Capstone Documentation

Release 0.1

Kapil Chiravarambath

Feb 13, 2019

CONTENTS

Contents:

CHAPTER

ONE

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!

CHAPTER

TWO

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

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.

CHAPTER

THREE

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
    },
    "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.

CHAPTER

FOUR

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

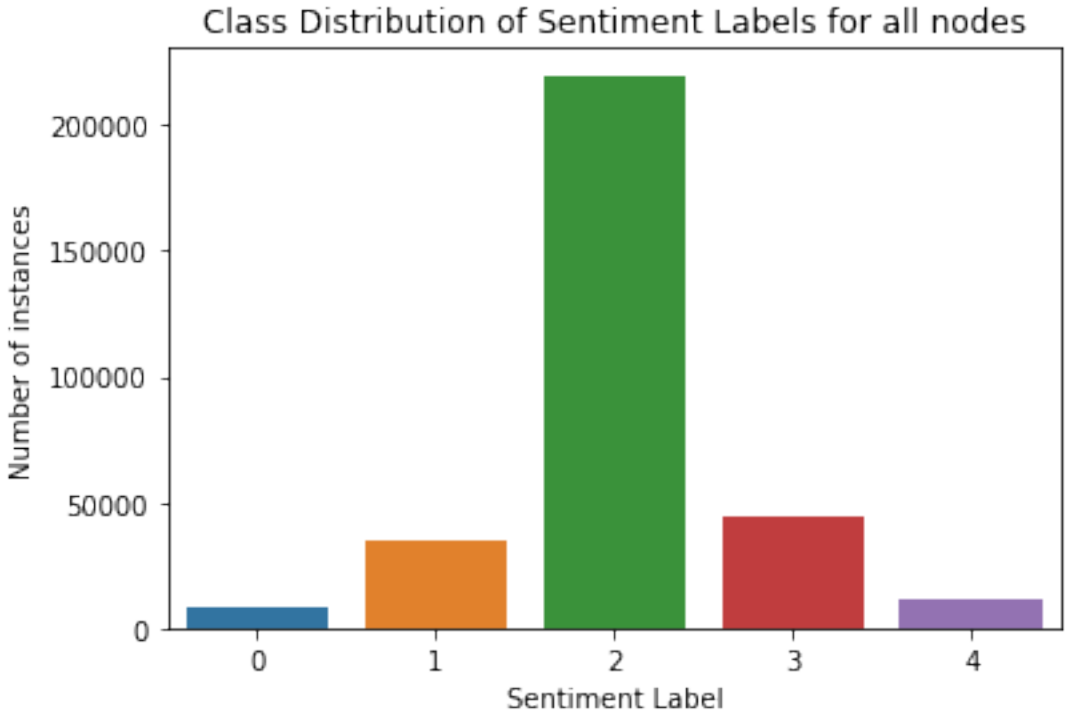
```
# Imports
import os
import sys
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

```
# 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: {0}'.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')
```

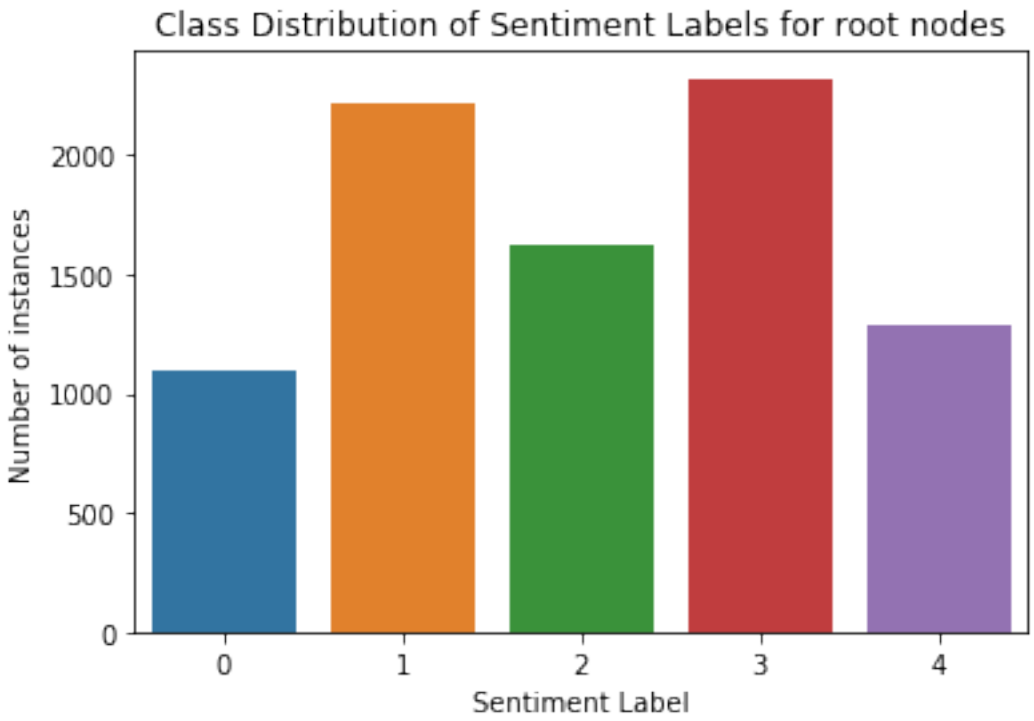


The class distribution is very heavily skewed towards neutral values. This class imbalance will influence the training of neural network and needs to be handled.

```
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: {0}'.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.

CHAPTER

FIVE

INFERENTIAL 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(left height, 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
```

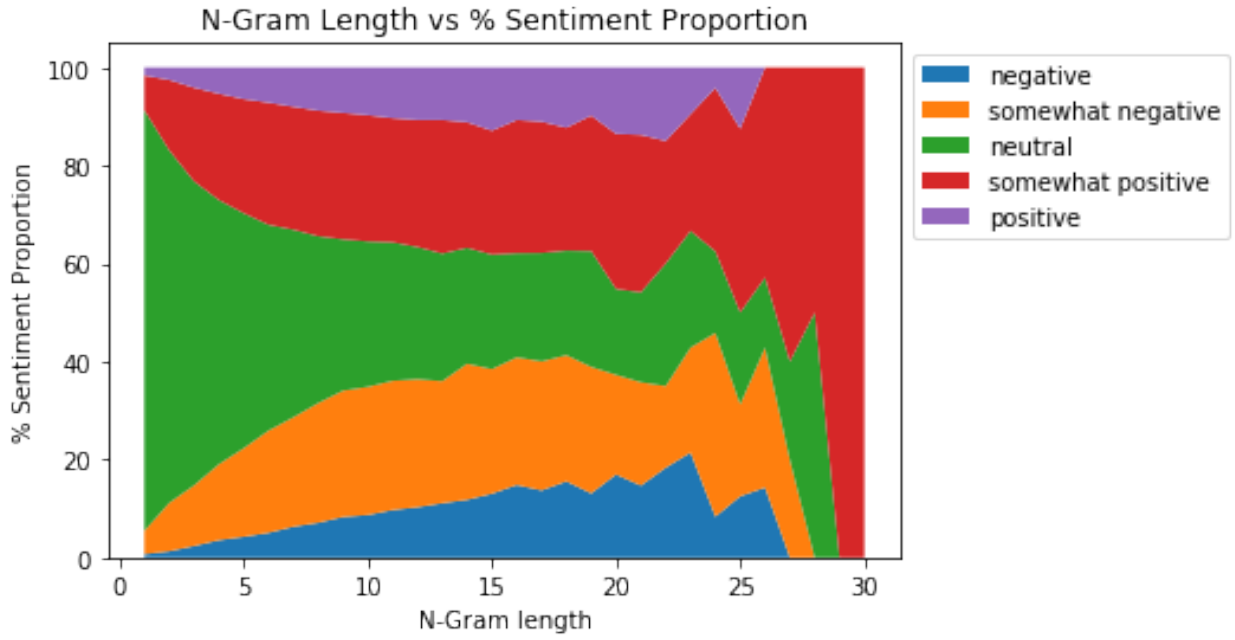
```
# Set path to model code
PROJ_ROOT = os.pardir
sys.path.append(PROJ_ROOT)
from src.features.tree import Tree
from src.models.data_manager import DataManager
```

```
# 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 sentence associated with it.

CHAPTER

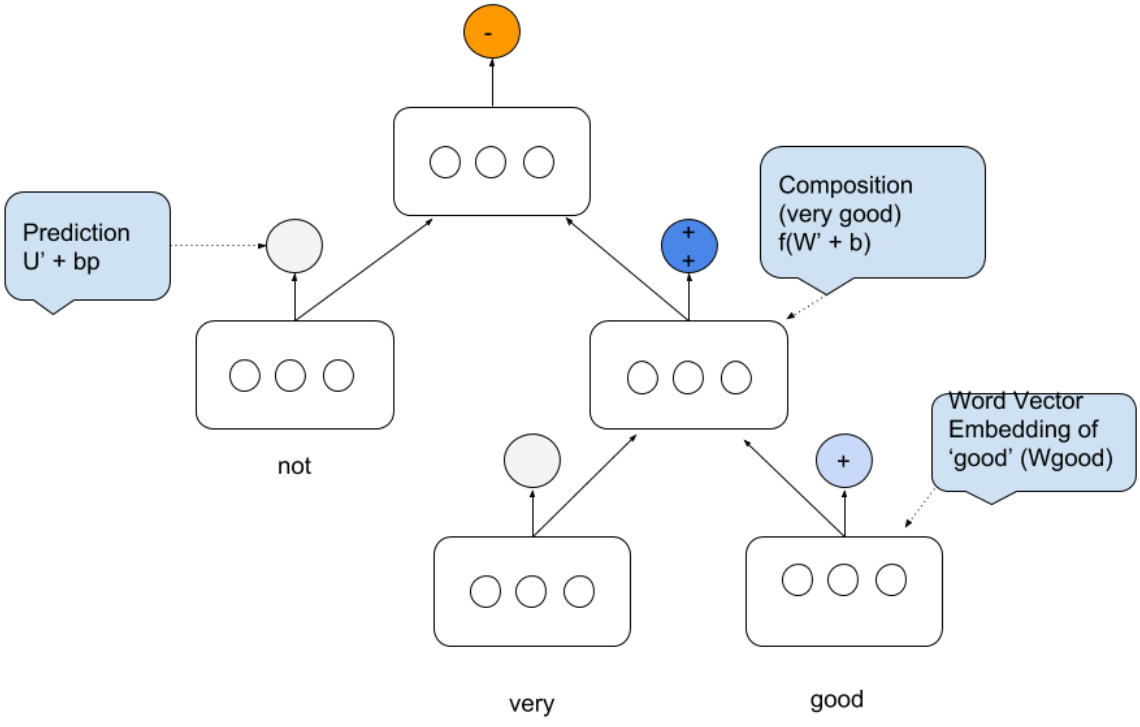
SIX

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:

$zs = W * X + b$ (Standard term)

$zt = X^T * T * X$ (Neural Tensor term)

$a = f(zs + zt)$ (Composition function)

where: * W: Weights to be computed by the model of shape [d, 2*:math:'d'] for Composition step. X: Word embeddings for input words stacked together. X is a column vector of shape [2*:math:d, 1]* b: bias term for the node of shape [d, 1] * T: Tensor of dimension [2*:math:d, 2*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, label_size] where label_size is the number of classes. * bs: Bias term for Projection Step of shape [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,

$:raw-latex:*delta*_{i, s} = U^{T}(y_i - t_i) :raw-latex:‘otimes ‘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 $k = 1, ..., d$ 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, > $\delta_{p2,out} = \left(W^T \delta_{p2,com} + S\right) \otimes f'\left(\begin{bmatrix} a \\ p1 \end{bmatrix}\right)$

where,

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

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

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

where,

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

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

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

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

CHAPTER

SEVEN

BASELINE MODELS

We look at bag of words models as baseline models for comparison with the RNTN model.

We evaluate the models for both root level and full tree node accuracy scores.

7.1 Extracting Phrases from the Treebank

The Sentiment Treebank dataset is in form of parsed trees. Here we generate all sub-phrases and their associated sentiments for evaluating full accuracy.

```
# Imports
import os
import sys
import numpy as np
import pandas as pd
```

```
# 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 sub-phrases for a single tree
def get_phrases(node):
    if node.isLeaf:
        return (np.asarray([node.word]), np.asarray([node.label]))
    else:
        left_phrases, left_labels = get_phrases(node.left)
        right_phrases, right_labels = get_phrases(node.right)
        curr_phrases = np.concatenate([np.asarray([node.text()]), left_phrases, right_
↪phrases])
        curr_labels = np.concatenate([np.asarray([node.label]), left_labels, right_
↪labels])
        return (curr_phrases, curr_labels)
```

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

```
# Get sub-phrases for every tree
X_data_train = []
y_data_train = []
for i in range(len(x_train)):
    X_tree, y_tree = get_phrases(x_train[i].root)
    X_data_train = np.concatenate([X_data_train, X_tree])
    y_data_train = np.concatenate([y_data_train, y_tree])

dt_train = pd.DataFrame(data={'phrase': X_data_train, 'label': y_data_train})
dt_train.to_csv('../src/data/processed/train_phrases.csv')
```

```
# or run only the following
dt_train = pd.read_csv('../src/data/processed/train_phrases.csv')
X_data_train = np.ravel(dt_train[['phrase']])
y_data_train = np.ravel(dt_train[['label']])
```

```
# Get sub-phrases for every cross validation set tree
X_data_dev = []
y_data_dev = []
for i in range(len(x_dev)):
    X_tree, y_tree = get_phrases(x_dev[i].root)
    X_data_dev = np.concatenate([X_data_dev, X_tree])
    y_data_dev = np.concatenate([y_data_dev, y_tree])

dt_dev = pd.DataFrame(data={'phrase': X_data_dev, 'label': y_data_dev})
dt_dev.to_csv('../src/data/processed/dev_phrases.csv')
```

```
# or run only the following
dt_dev = pd.read_csv('../src/data/processed/dev_phrases.csv')
X_data_dev = np.ravel(dt_dev[['phrase']])
y_data_dev = np.ravel(dt_dev[['label']])
```

7.2 Building vocabulary

The vocabulary is built using a CountVectorizer that extracts words and pre-processes them (lemmatization). The fit_transform method returns the one-hot encoded version of the sentences with the frequency of the word occurence as the components of the generated sentence vector (rows).

```
# Build vocabulary using CountVectorizer
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer()
X_data = vectorizer.fit_transform(np.concatenate([X_data_train, X_data_dev]))
X_data = X_data.tocsc()  # some versions of sklearn return COO format
y_data = np.concatenate([y_data_train, y_data_dev])
```

7.3 Cross validation

The dev/train split is already specified in the trained dataset. Here we use Predefined Split to specify which data is cross-validation test set and which is training data.

```
# Use Predefined split as train, dev data is already separate
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
```

(continues on next page)

(continued from previous page)

```
from sklearn.model_selection import PredefinedSplit, GridSearchCV

# Prepare data for training
validation_set_indexes = [-1] * len(X_data_train) + [0] * len(X_data_dev)
cv = PredefinedSplit(test_fold=validation_set_indexes)
```

7.4 Naive Bayes Model

The Naive Bayes model provides a good baseline as it only makes independence assumption. However, we do not expect it to do well against sentences with negation as it does not take structure of the sentence in account. It will also mark sentences with higher positive word counts more positively and similarly negative word counts will give negative sentiment as the prediction.

```
# Simple naive bayes classifier
from sklearn.metrics import make_scorer, accuracy_score, f1_score, log_loss

# Use MultinomialNB classifier
from sklearn.naive_bayes import MultinomialNB
nb_clf = MultinomialNB()

# Find the best hyper-parameter using GridSearchCV
params = {'alpha': [.1, 1, 10]}
nb_model = GridSearchCV(nb_clf, params, scoring=make_scorer(accuracy_score), cv=cv)
```

```
# Train model
nb_model.fit(X_data, y_data)
print(nb_model.best_params_)
```

```
{'alpha': 1}
```

We load the test dataset to compare. Phrases are extracted out of each sentence as we know their sentiment labels.

```
# Get sub-phrases for every test set tree
X_data_test = []
y_data_test = []
for i in range(len(x_test)):
    X_tree, y_tree = get_phrases(x_test[i].root)
    X_data_test = np.concatenate([X_data_test, X_tree])
    y_data_test = np.concatenate([y_data_test, y_tree])

dt_test = pd.DataFrame(data={'phrase': X_data_test, 'label': y_data_test})
dt_test.to_csv('../src/data/processed/test_phrases.csv')
```

```
# or run only the following
dt_test = pd.read_csv('../src/data/processed/test_phrases.csv')
X_data_test = np.ravel(dt_test[['phrase']])
y_data_test = np.ravel(dt_test[['label']])
```

Generate a word frequency count vector for each sentence.

```
# Vectorize
X_data_test_vec = vectorizer.transform(X_data_test)
```

```
# Score model
# Print the accuracy on the test and training dataset
#training_accuracy = model.score(X_data.reshape(-1,1), y_data)
#test_accuracy = model.score(X_data_test_vec, y_data_test.astype(int))
y_pred = nb_model.predict(X_data_test_vec)
y_true = y_data_test.astype(int)
y_probs = nb_model.predict_proba(X_data_test_vec)

#print("Accuracy on training data: {:2f}".format(training_accuracy))
print("Accuracy on full test data (NB):     {:2f}".format(accuracy_score(y_true, y_
↪pred)))
```

```
Accuracy on full test data (NB):     0.735557
```

The model gives a good accuracy score. Next we look at where the misclassifications are happening.

```
# Classification Report
print(classification_report(y_true, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.44      0.09      0.15      2008
           1       0.57      0.19      0.29      9255
           2       0.76      0.97      0.86     56548
           3       0.54      0.29      0.38     10998
           4       0.60      0.17      0.26      3791

   micro avg       0.74      0.74      0.74     82600
   macro avg       0.58      0.34      0.39     82600
weighted avg       0.70      0.74      0.68     82600
```

```
# F1-score
print(f1_score(y_true, y_pred, average='weighted'))
```

```
0.6833294980935176
```

```
# Confusion Matrix
print(confusion_matrix(y_true, y_pred))
```

```
[[  177   606  1170    52     3]
 [  168  1779  7030   250    28]
 [   50   595 54967   856    80]
 [    7   131  7347  3194   319]
 [    0    29  1506  1616   640]]
```

The misclassifications show where the problem lies. The class imbalance is causing the classifier to make more 'neutral' sentiment predictions. Even for more extreme values, the classifications error towards neutral state.

The weighted F1 Score gives a good overall measure to directly evaluate the models, the other is log loss as shown below.

```
# Log loss per sample
print(log_loss(y_true, y_probs))
```

```
0.8609283543125711
```

7.5 Fixing imbalance with oversampling

To reduce misclassification due to imbalance, we try oversampling the minority classes. Undersampling would not be a good choice as the vocabulary matrix is sparse and this will result in model classifying most of the 1-grams not seen as neutral.

```
from imblearn.over_sampling import RandomOverSampler
from collections import Counter

ros = RandomOverSampler(random_state=42)
X_os, y_os = ros.fit_resample(X_data_test_vec, y_data_test.astype(int))

print('After Rebalance: {0}'.format(Counter(y_os)))
```

```
After Rebalance: Counter({2: 56548, 3: 56548, 1: 56548, 4: 56548, 0: 56548})
```

```
# Train model
nb_rebal_model = MultinomialNB()
nb_rebal_model.fit(X_os, y_os)

# Score model after rebalance
y_pred = nb_rebal_model.predict(X_data_test_vec)
y_true = y_data_test.astype(int)
y_probs = nb_rebal_model.predict_proba(X_data_test_vec)
```

```
#print("Accuracy on training data: {:2f}".format(training_accuracy))
print("Accuracy on full test data (NB):     {:2f}".format(accuracy_score(y_true, y_
↪pred)))
```

```
Accuracy on full test data (NB):     0.338172
```

Oversampling does not improve the accuracy of the classification at all. We will use the original unbalanced sample as the baseline.

7.6 Root Level Evaluation Metrics

Next we examine how the root level accuracy and classification metrics are, which gives us the overall prediction for a sentence.

```
# Build root train data set
x_root_train = [tree.text() for tree in x_train]
y_root_train = [tree.root.label for tree in x_train]
x_root_dev = [tree.text() for tree in x_dev]
y_root_dev = [tree.root.label for tree in x_dev]
x_root_all = x_root_train + x_root_dev
y_root_all = y_root_train + y_root_dev
```

```
# Vectorize x
x_root_train_vec = vectorizer.transform(x_root_all)
```

```
# Train model for root nodes
nb_root = MultinomialNB()
nb_root.fit(x_root_train_vec, y_root_all)
```

```
MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True)
```

```
# Build root test data set
x_root_test = [tree.text() for tree in x_test]
y_root_test = [tree.root.label for tree in x_test]
x_root_test_vec = vectorizer.transform(x_root_test)
```

```
# Score model
# Print the accuracy on the test dataset
y_pred = nb_model.predict(x_root_test_vec)
y_true = y_root_test
y_probs = nb_model.predict_proba(x_root_test_vec)

print("Accuracy on root test data (NB):     {:2f}".format(accuracy_score(y_true, y_
↪pred)))
```

```
Accuracy on root test data (NB):     0.319457
```

```
# Classification Report
print(classification_report(y_true, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.65      0.11      0.18       279
           1       0.51      0.23      0.32       633
           2       0.21      0.75      0.33       389
           3       0.40      0.32      0.36       510
           4       0.66      0.19      0.30       399

   micro avg       0.32      0.32      0.32      2210
   macro avg       0.49      0.32      0.30      2210
weighted avg       0.48      0.32      0.31      2210
```

```
# F1-score
print(f1_score(y_true, y_pred, average='weighted'))
```

```
0.3088168253391177
```

```
# Confusion Matrix
print(confusion_matrix(y_true, y_pred))
```

```
[[ 30  78 165   5   1]
 [ 13 147 435  33   5]
 [  3  37 290  54   5]
 [  0  19 300 163  28]
 [  0   5 165 153  76]]
```

```
# Log loss per sample
print(log_loss(y_true, y_probs))
```

```
2.4381252006114305
```

The root level accuracy is much lower as expected. This also aligns with a max accuracy of about 45% for the best model in the paper.

Here, too we see distinct effect of too many neutral words on the overall accuracy of the model.

CHAPTER

EIGHT

MODEL EVALUATION

We first look at accuracy metrics and try to identify what is mis-classified most.

8.1 Evaluation Setup

Load the model and test data.

```
# Imports
import os
import sys
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix,␣
↪f1_score, log_loss
```

```
# Add src to path
PROJ_ROOT = os.pardir
sys.path.append(PROJ_ROOT)
from src.features.tree import Tree
from src.models.data_manager import DataManager
from src.models.rntn import RNTN
from src.features.tree import Tree
```

```
# Best Model
model_name = 'RNTN_30_tanh_35_5_None_50_0.001_0.01_9645'

# Instantiate model
model_rntn = RNTN(model_name=model_name)
```

```
# Load test data for full tree
x_test = DataManager().x_test
```

```
# Function to get sub-phrases for a single tree
def get_phrases(node):
    if node.isLeaf:
        return (np.asarray([str(node)]), np.asarray([node.label]))
    else:
        left_phrases, left_labels = get_phrases(node.left)
        right_phrases, right_labels = get_phrases(node.right)
        curr_phrases = np.concatenate([np.asarray([str(node)]), left_phrases, right_
↪phrases])
```

(continues on next page)

(continued from previous page)

```
        curr_labels = np.concatenate([np.asarray([node.label]), left_labels, right_
↪labels])
        return (curr_phrases, curr_labels)

X_data = []
y_data = []

for i in range(len(x_test)):
    X_tree, y_tree = get_phrases(x_test[i].root)
    X_data = np.concatenate([X_data, X_tree])
    y_data = np.concatenate([y_data, y_tree])

dt_test = pd.DataFrame(data={'phrase': X_data})
dt_test.to_csv('../src/data/processed/test_phrases_raw.csv')
```

```
X_trees_data = [Tree(t) for t in X_data]
```

8.2 Full Tree Accuracy

The test data contains the each sentence and its sub-phrase and associated ground truth label. We use the model predict function to look at how each node is predicted.

```
# Call models predict method
y_pred = model_rntn.predict(np.asarray(X_trees_data).reshape(-1, 1))
y_true = y_data.astype(int)
```

```
INFO:tensorflow:Restoring parameters from C:UserscskapgithubSpringboardcapstone_
↪1srcmodels../../models//RNTN_30_tanh_35_5_None_50_0.001_0.01_9645/RNTN_30_
↪tanh_35_5_None_50_0.001_0.01_9645.ckpt
```

```
# Calculate the probabilities for
y_probs = model_rntn.predict_proba(np.asarray(X_trees_data).reshape(-1, 1))
```

```
INFO:tensorflow:Restoring parameters from C:UserscskapgithubSpringboardcapstone_
↪1srcmodels../../models//RNTN_30_tanh_35_5_None_50_0.001_0.01_9645/RNTN_30_
↪tanh_35_5_None_50_0.001_0.01_9645.ckpt
```

```
# Accuracy
print("Accuracy on full test data (RNTN):     {:2f}".format(accuracy_score(y_true, y_
↪pred)))
```

```
Accuracy on full test data (RNTN):     0.664661
```

RNTN model accuracy is less than the accuracy of Naive Bayes model. Lets look closer at what is mis-classified and also compute other metrics.

```
# Classification Report
print(classification_report(y_true, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.32      0.29      0.31      2008
           1       0.38      0.46      0.42      9255
```

(continues on next page)

(continued from previous page)

```
           2       0.88      0.76      0.82     56548
           3       0.30      0.44      0.36     10998
           4       0.48      0.54      0.51      3791

   micro avg       0.66      0.66      0.66     82600
   macro avg       0.47      0.50      0.48     82600
weighted avg       0.71      0.66      0.68     82600
```

We now start seeing why this model does better with predicting sentiments over Naive Bayes. Even though the accuracy is lower, the per-class model is less confused about classification. It is not classifying everything is neutral, rather the positive sentiments are mostly misclassified as slightly positive, which will make prediction more reliable.

```
# Confusion Matrix
print(confusion_matrix(y_true, y_pred))
```

```
[[  591   819   250   288    60]
 [  791  4270  2178  1810   206]
 [  294  4601 43158  8036   459]
 [  116  1183  3355  4852  1492]
 [   36   219   337  1169  2030]]
```

```
# F1-score
print(f1_score(y_true, y_pred, average='weighted'))
```

```
0.6836731505540418
```

```
# Log loss per sample
print(log_loss(y_true, y_probs))
```

```
1.0353974476756531
```

Again, we see a better F1 score with RNTN model due to better classification in minority classes. The average Log loss is slightly higher, due to lesser accuracy of the model.

8.3 Root Level Evaluation Metrics

```
# Call models predict method
y_pred = model_rntn.predict(np.asarray(x_test).reshape(-1,1))
y_true = [t.root.label for t in x_test]
```

```
INFO:tensorflow:Restoring parameters from C:UserscskapgithubSpringboardcapstone_
↪1srcmodels../../models//RNTN_30_tanh_35_5_None_50_0.001_0.01_9645/RNTN_30_
↪tanh_35_5_None_50_0.001_0.01_9645.ckpt
```

```
# Calculate probabilities for log loss
y_probs = model_rntn.predict_proba(np.asarray(x_test).reshape(-1, 1))
```

```
INFO:tensorflow:Restoring parameters from C:UserscskapgithubSpringboardcapstone_
↪1srcmodels../../models//RNTN_30_tanh_35_5_None_50_0.001_0.01_9645/RNTN_30_
↪tanh_35_5_None_50_0.001_0.01_9645.ckpt
```

```
# Accuracy
print("Accuracy on root test data (RNTN):     {:2f}".format(accuracy_score(y_true, y_
↪pred)))
```

(continues on next page)

(continued from previous page)

```
Accuracy on root test data (RNTN):     0.373756
```

Root accuracy for RNTN is significantly higher than baseline. This can be explained as extreme sentiments are not misclassified as neutral as much as the nearer class of slightly positive/negative sentiments.

```
# Classification Report
print(classification_report(y_true, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.29      0.34      0.31       279
           1       0.42      0.48      0.45       633
           2       0.23      0.01      0.01       389
           3       0.29      0.30      0.29       510
           4       0.44      0.68      0.54       399

   micro avg       0.37      0.37      0.37      2210
   macro avg       0.33      0.36      0.32      2210
weighted avg       0.34      0.37      0.33      2210
```

```
# Confusion Matrix
print(confusion_matrix(y_true, y_pred))
```

```
[[ 96 120   2  40  21]
 [140 301   4 130  58]
 [ 54 141   3 124  67]
 [ 32 119   4 153 202]
 [ 11  33   0  82 273]]
```

```
# F1-score
print(f1_score(y_true, y_pred, average='weighted'))
```

```
0.33485051111248126
```

```
# Log loss per sample
print(log_loss(y_true, y_probs))
```

```
1.9482420690090614
```

All metrics show better performance as compared to root sentiment predictions for the baseline model.

CHAPTER

NINE

MODEL VISUALIZATION

```
# Imports
import os
import sys
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.manifold import TSNE
from IPython.core.display import display, HTML, Image

# Set project root
PROJ_ROOT = os.pardir
sys.path.append(PROJ_ROOT)
from src.features.tree import Tree
from src.models.data_manager import DataManager
from src.models.rntn import RNTN
from src.models.predict_model import predict_model
```

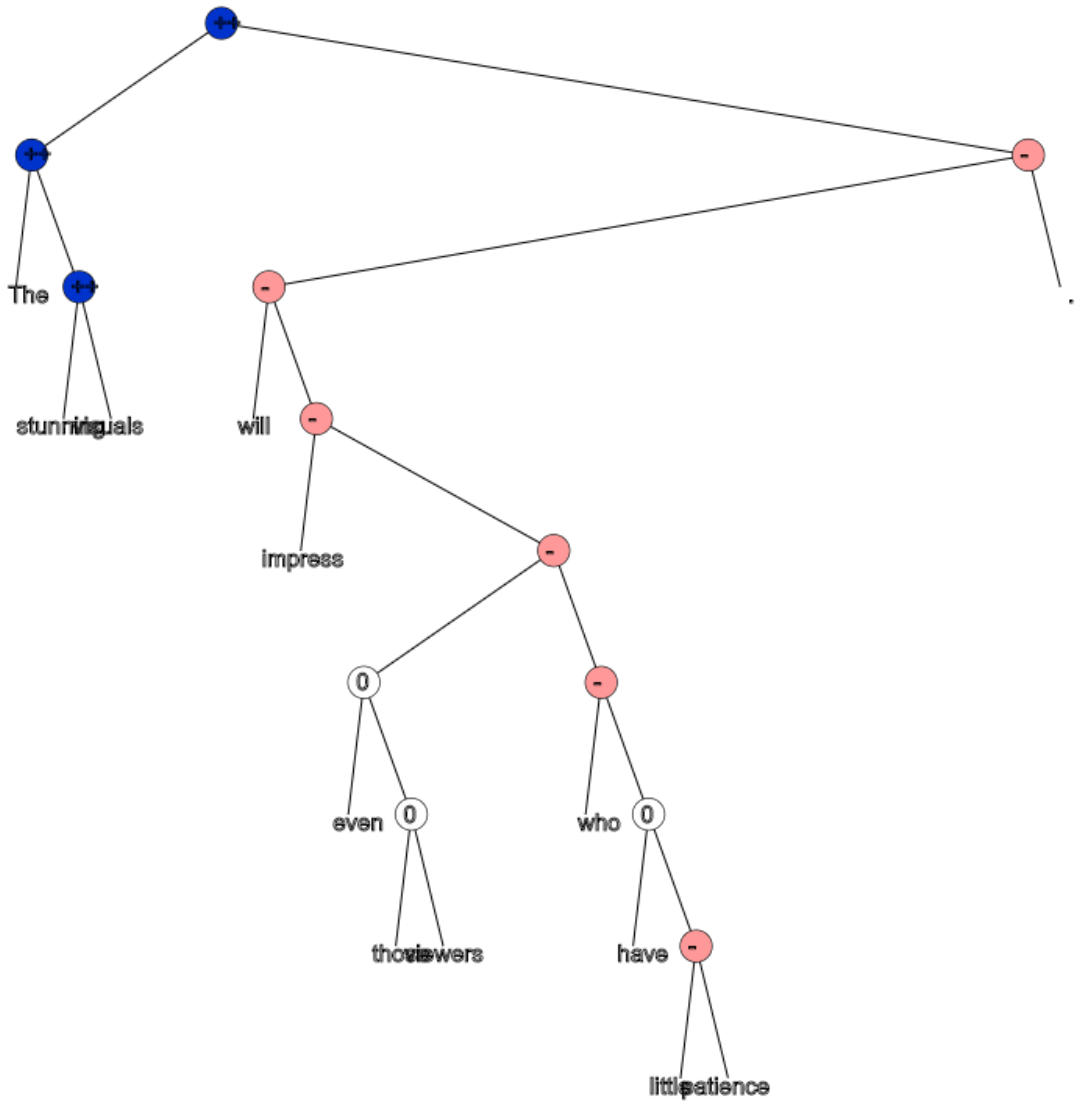
We look at some of the reviews from the test set, to examine how the trained model makes predictions.

9.1 Example: Positive Sentiment

This is an example, where the positive sentiment from the left subtree dominates the right subtree sentiment.

```
def build_str(txt):
    return     \
        '<html><body><h3>Sentiment tree:</h3> \
        <canvas id="treeCanvas" width="800" height="800" style="border:1px solid
↪#000000;"></canvas> \
        <script type=text/javascript src="../src/webapp/static/tree.js"></script> \
        <script type="text/javascript"> \
            draw_tree(\'' + txt + '\', 800, 800); \
        </script></body></html>'
```

```
pos_txt = 'The stunning visuals will impress even those viewers who have little␣
↪patience.'
#y, predict_txt = predict_model(pos_txt)
#pos_display_txt = build_str(str(predict_txt))
# display(HTML(pos_display_txt))
display(Image(filename='../docs/pos_sent_Visualizations.png'))
```

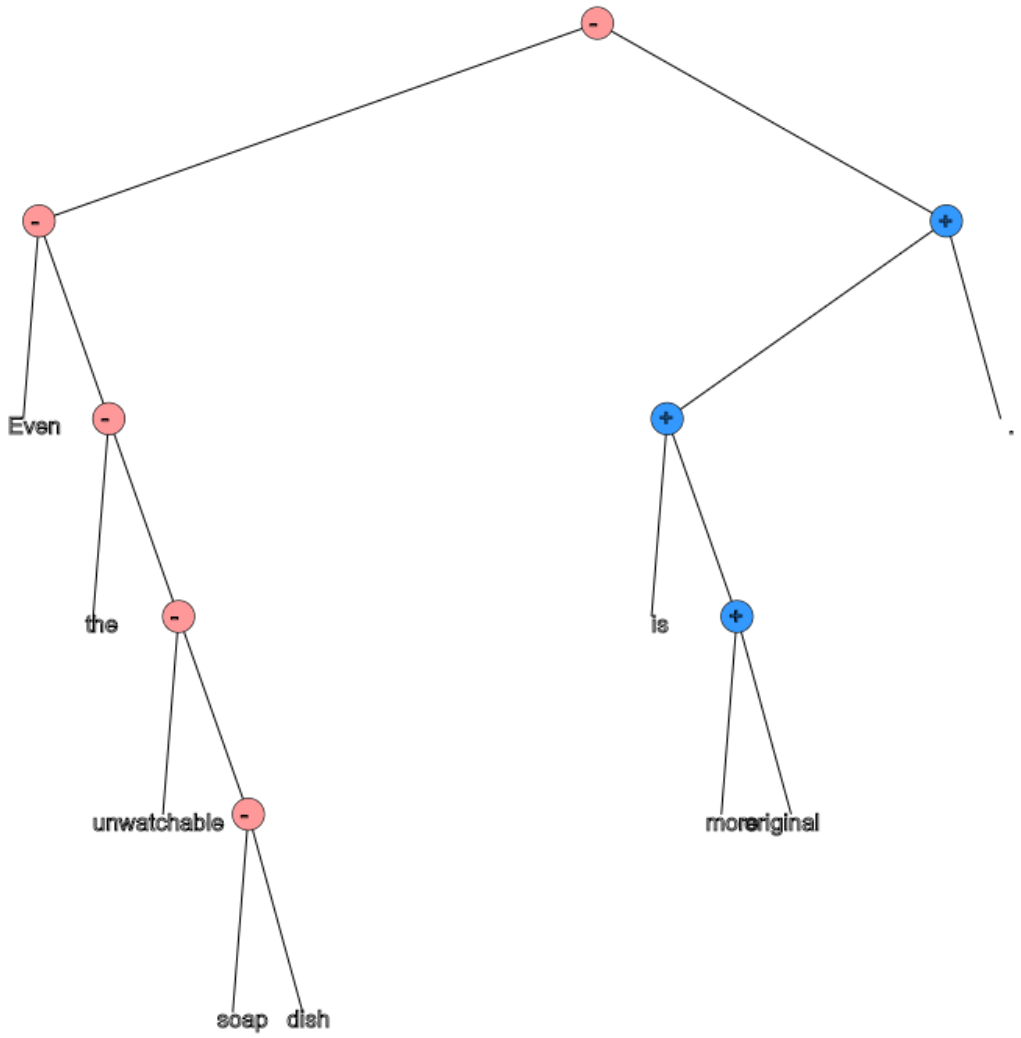


As seen with this example, the statement is overall very positive even though a part of the sentence is negative.

9.2 Example: Negative Sentiment

A great example, where the partial phrase expresses positive sentiment (more original), but the sentence is correctly classified as negative.

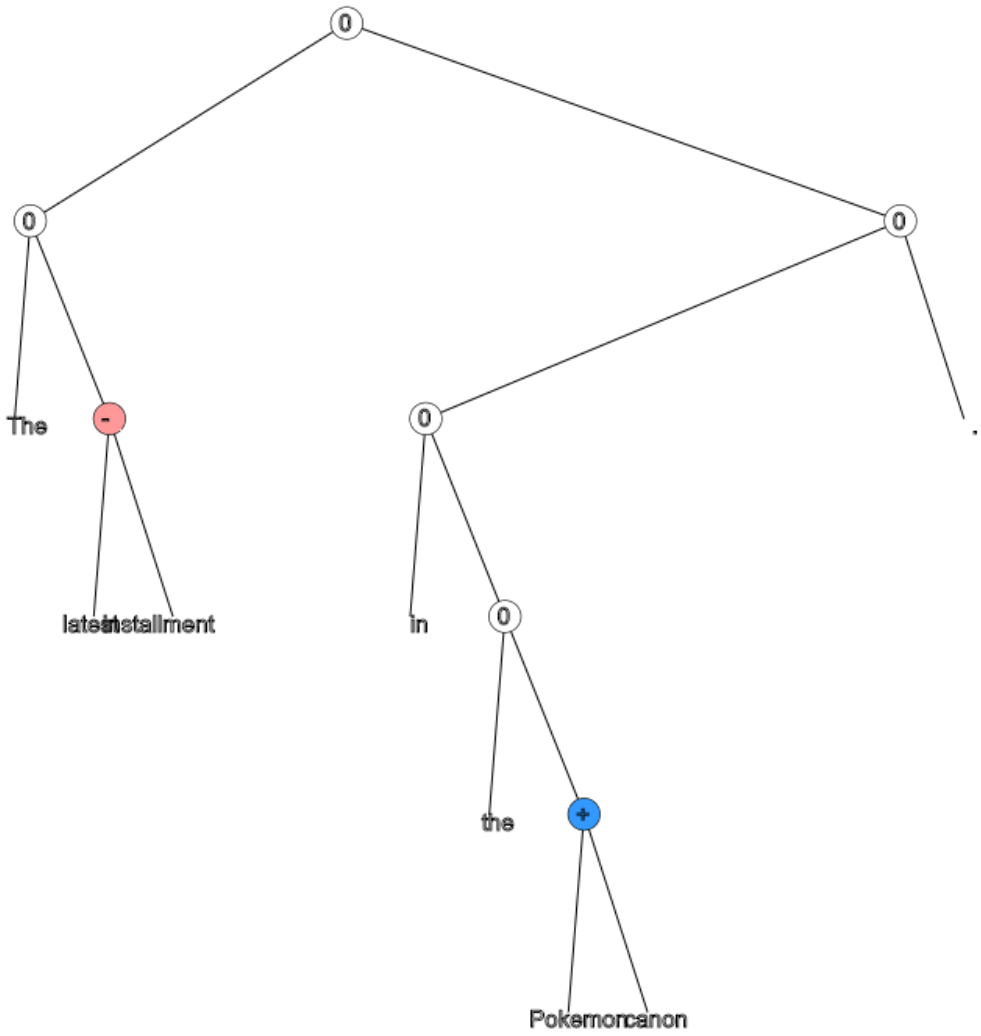
```
pos_txt = "Even the unwatchable soap dish is more original."
# y, predict_txt = predict_model(pos_txt)
# pos_display_txt = build_str(str(predict_txt))
# display(HTML(pos_display_txt))
display(Image(filename='../docs/neg_sen_Visualizations.png'))
```



9.3 Example: Neutral Sentiment

This sentence expresses no opinion and is correctly classified as neutral.

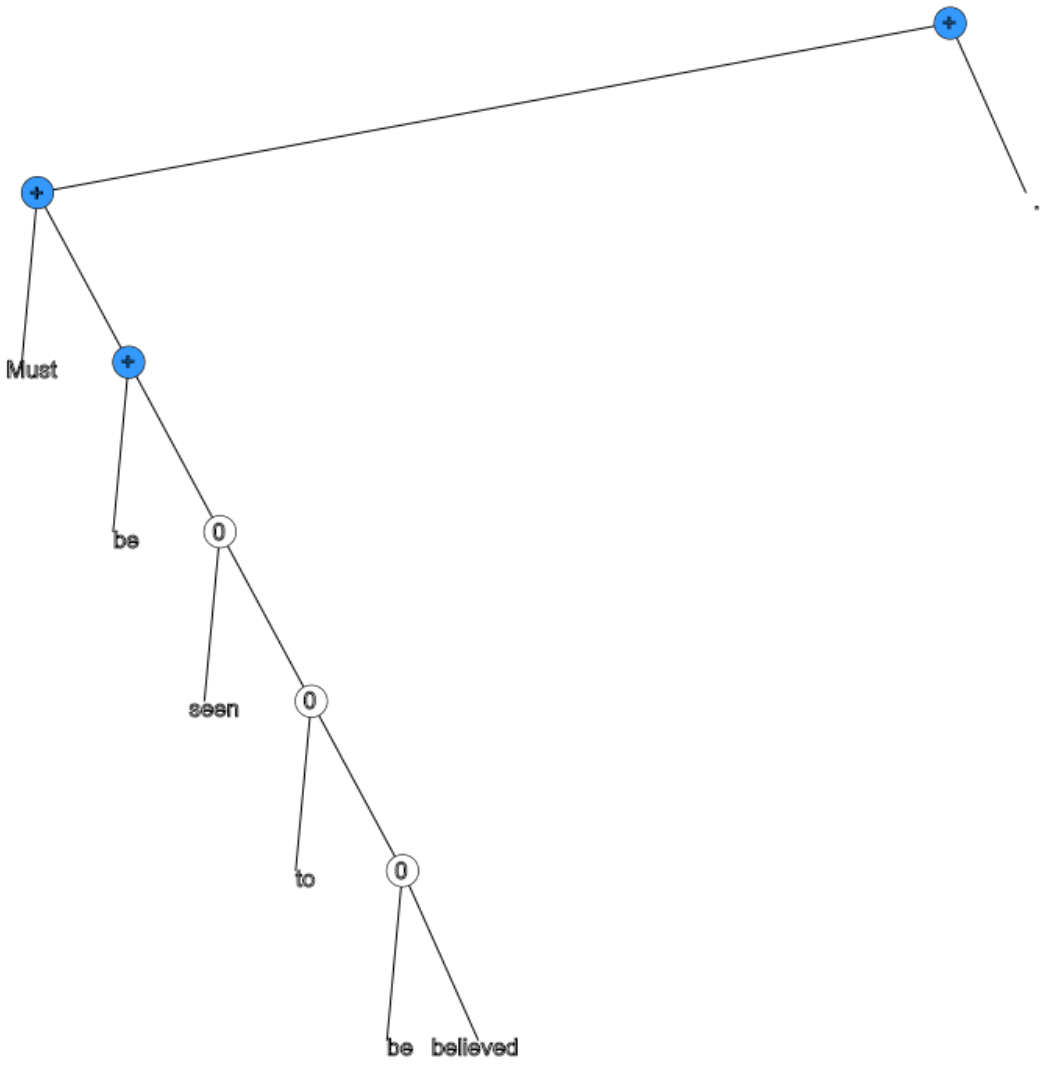
```
pos_txt = 'The latest installment in the Pokemon canon.'
# y, predict_txt = predict_model(pos_txt)
# pos_display_txt = build_str(str(predict_txt))
# display(HTML(pos_display_txt))
display(Image(filename='../docs/neutral_sen_Visualizations.png'))
```



9.4 Example: No Sentiment Words

This is a great example of sentiment prediction, even when there are no sentiment words!

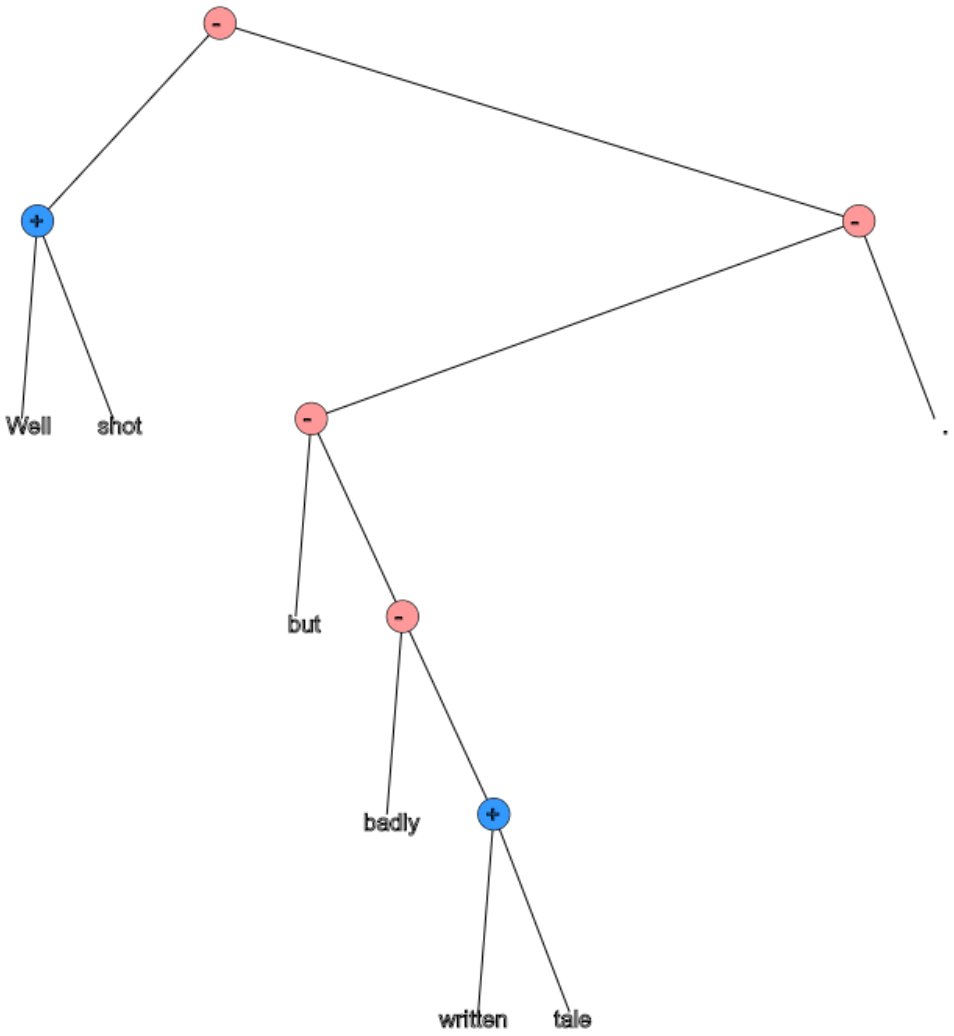
```
pos_txt = 'Must be seen to be believed.'
# y, predict_txt = predict_model(pos_txt)
# pos_display_txt = build_str(str(predict_txt))
# display(HTML(pos_display_txt))
display(Image(filename='../docs/no_sen_Visualizations.png'))
```



9.5 Example: Mixed Sentiments

This example shows how mixed sentiments are expressed in a parse tree, the left subtree is slightly positive, while the right subtree changes the its sentiment due to strong effect of 'badly'.

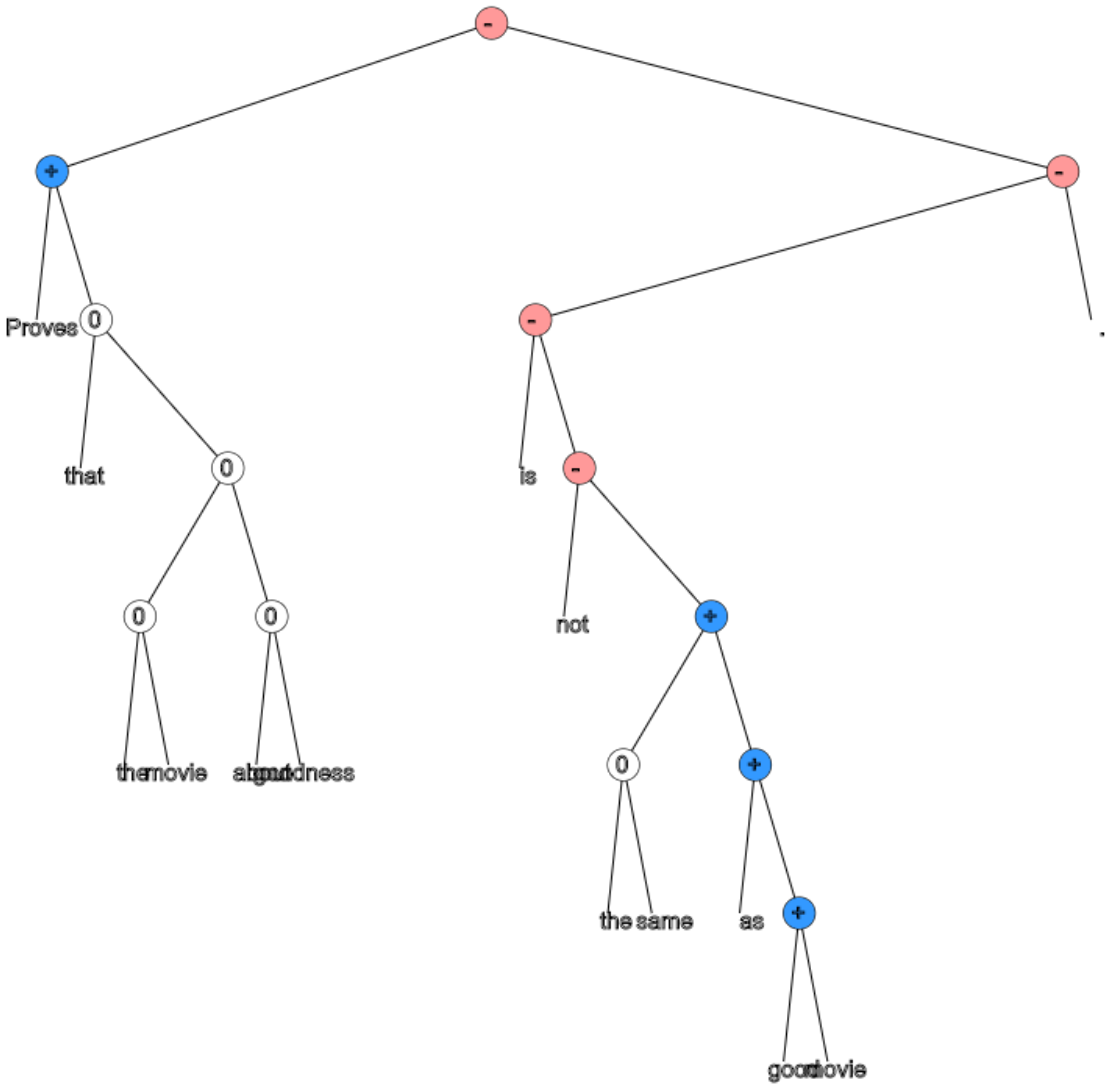
```
pos_txt = "Well shot but badly written tale."
# y, predict_txt = predict_model(pos_txt)
# pos_display_txt = build_str(str(predict_txt))
# display(HTML(pos_display_txt))
display(Image(filename='../docs/mixed_sen_Visualizations.png'))
```



9.6 Example: Sentence Negation

This example shows strong effect of the word 'not' in flipping the sentiment.

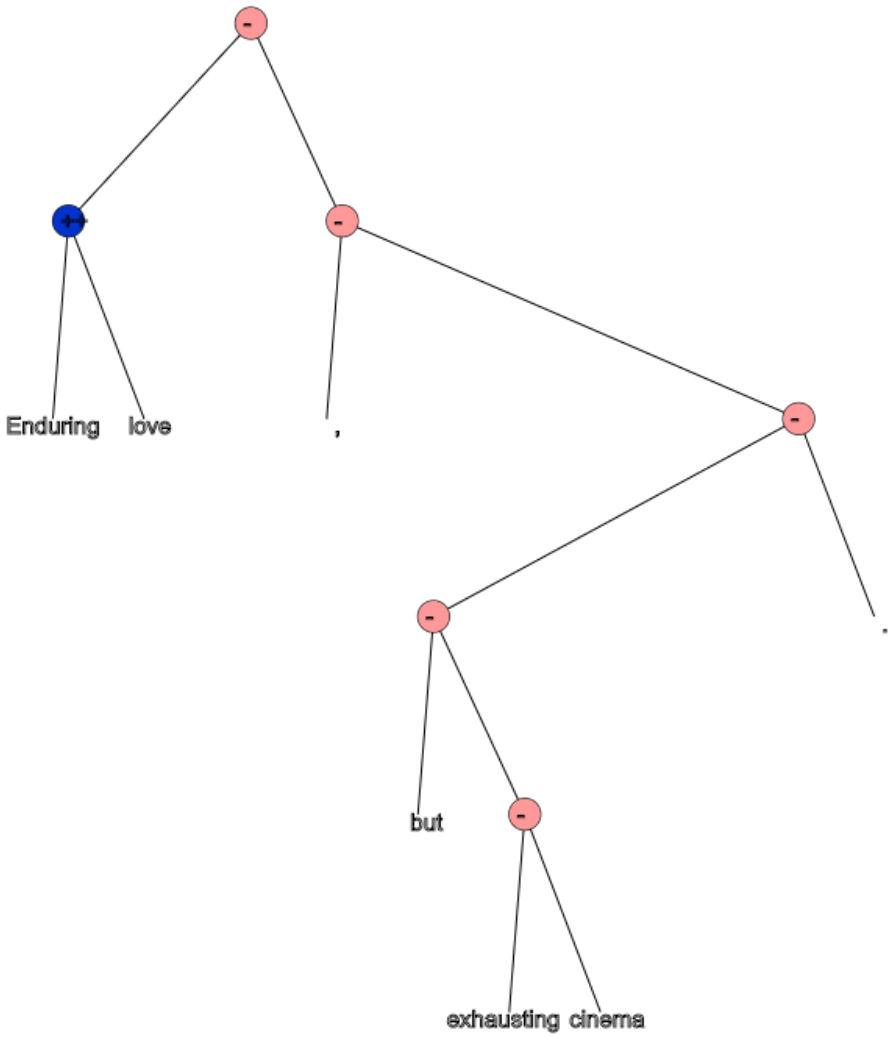
```
pos_txt = "Proves that the movie about goodness is not the same as good movie."
# y, predict_txt = predict_model(pos_txt)
# pos_display_txt = build_str(str(predict_txt))
# display(HTML(pos_display_txt))
display(Image(filename='../docs/sen_neg_Visualizations.png'))
```



9.7 Sentence Orientation flip

Here, we see strong effect of the word 'but' in flipping the overall sentiment.

```
pos_txt = "Enduring love, but exhausting cinema."
# y, predict_txt = predict_model(pos_txt)
# pos_display_txt = build_str(str(predict_txt))
# display(HTML(pos_display_txt))
display(Image(filename='../docs/sen_flip_Visualizations.png'))
```



9.8 Similar Words: t-SNE Visualization

One way of validating the generated word vectors is to look at how close they are to each other. Since the word-embeddings in the trained model live in high-dimensional space, these have to be mapped to 2-D space for visualization. t-SNE (t-Stochastic Neighbor Embedding) allows for reducing the vector to 2-D space.

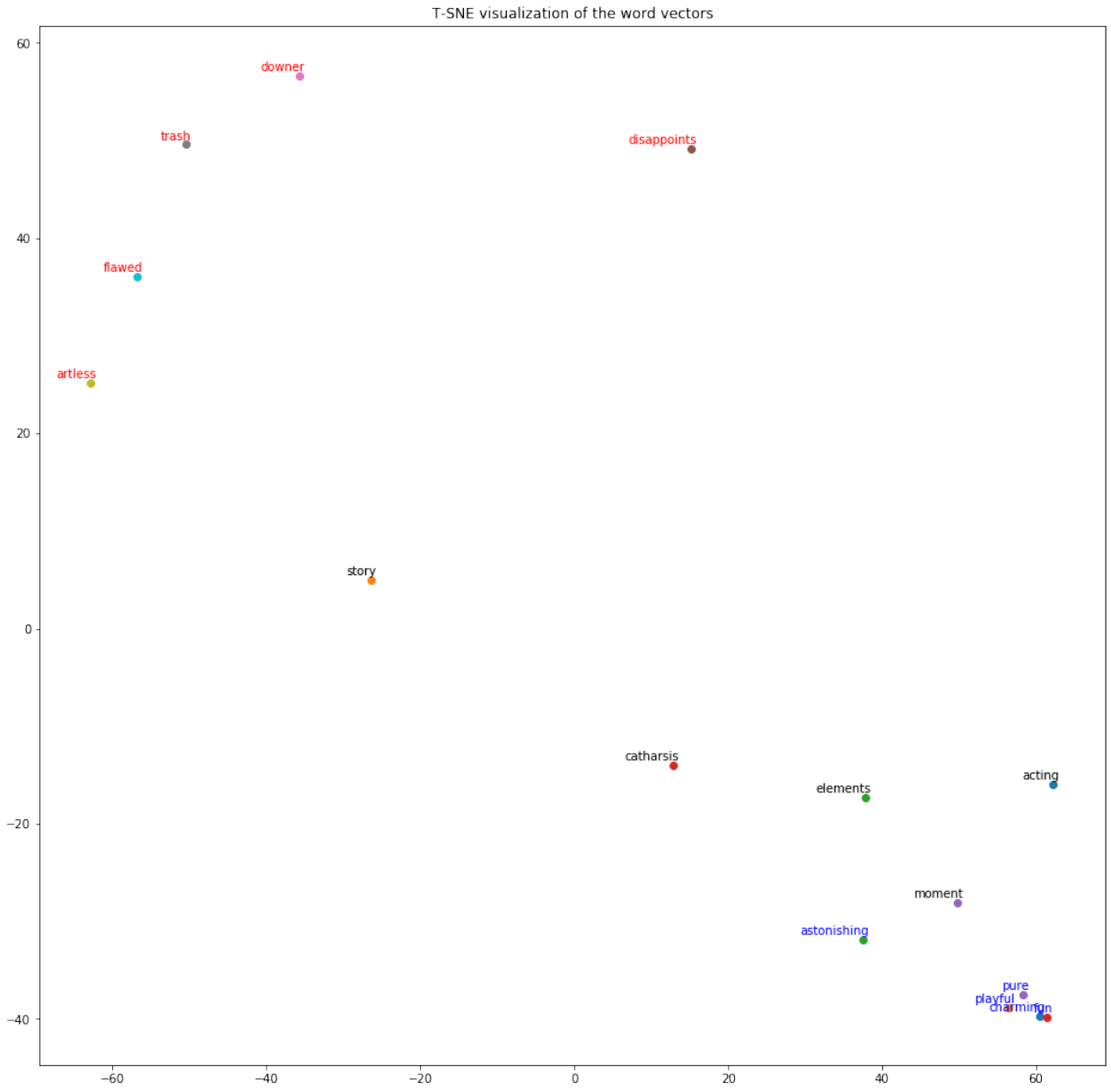
We look at a few randomly chosen words from each class and see how close these are in 2-D space.

```
# Load word embeddings from the model
model_name = 'RNTN_30_tanh_35_5_None_50_0.001_0.01_9645'

# Load model
rntn_model = RNTN(model_name=model_name)
L, vocab = rntn_model.get_word_embeddings()
```

```
# Choose words from each class
pos_words = ['charming', 'playful', 'astonishing', 'fun', 'pure']
neg_words = ['disappoints', 'downer', 'trash', 'artless', 'flawed']
neutral_words = ['acting', 'story', 'elements', 'catharsis', 'moment']

pos_idx = [vocab[i] for i in pos_words]
neg_idx = [vocab[i] for i in neg_words]
neutral_idx = [vocab[i] for i in neutral_words]
```

```
# Generate 2-d word embeddings
tsne_model = TSNE()
tsne_embeddings = tsne_model.fit_transform(np.transpose(L))
x = tsne_embeddings[:, 0]
y = tsne_embeddings[:, 1]
```

```
# Show the embeddings
plt.figure(figsize=(16, 16))
for i in range(5):
    idx = pos_idx[i]
    plt.scatter(x[idx],y[idx])
    plt.annotate(pos_words[i],
                 xy=(x[idx], y[idx]),
                 xytext=(5, 2),
                 textcoords='offset points',
                 ha='right',
                 va='bottom', color='blue', label='positive')

for i in range(5):
    idx = neg_idx[i]
    plt.scatter(x[idx],y[idx])
    plt.annotate(neg_words[i],
                 xy=(x[idx], y[idx]),
                 xytext=(5, 2),
                 textcoords='offset points',
                 ha='right',
                 va='bottom', color='red', label='negative')

for i in range(5):
    idx = neutral_idx[i]
    plt.scatter(x[idx],y[idx])
    plt.annotate(neutral_words[i],
                 xy=(x[idx], y[idx]),
                 xytext=(5, 2),
                 textcoords='offset points',
                 ha='right',
                 va='bottom', label='neutral')

plt.title('T-SNE visualization of the word vectors')
plt.show()
```



We see three distinct clusters, one for each of the three groups. The blue cluster represents the positive words, the red cluster the negative words and the black cluster represents neutral sentiment.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search