# Recursive Neural Network for Opinion Detection On Twitter[1]

GI15 Information Retrieval and Data Mining - Group Project

Yazhe Li
Yingkai Xu
Dexuan Zhang

13 April 2015

[1]Our system is on Github, download with Git clone https://github.com/liyazhe/IRDMProject.git

## Part I: Project Description

Social network platforms such as Twitter are very popular for political campaigning. So many people are tweeting about their favorite party or candidate. Analyzing the tweets can identify the political sentiment expressed by users and extract useful insight on political opinions.

The purpose of our Information Retrieval and Data Mining project is to build a system that summarizes political sentiment given some Twitter data.

### Data Description

The given dataset is a corpus of tweets during the 150 minutes of the first Presidential debate in 2008. Each data item contains the information of tweet id, data and time of the tweet, the content, the author's name as well as the nickname. Although there are up to eight labels slot in the dataset, each data item was manually labelled by mostly 3 to 5 human reviewers. They were labelled into four categories:

**negative** negative attitude towards the candidates

**positive** positive attitude towards the candidates

**mixed** mixed attitude

**others** Talking about other things

### Labelling the sentiment

Because the data is labelled by individual reviewers, it is common that there are different opinions, even contradicting opinions, on the classification of the tweets. This is especially true when the tweet content is not very clear about the orientation of the author. For example, the tweets below have been classified as positive by two reviewers and negative by an equal number of reviewers:

- *McCain in the membrane! #debate08*
- *rt: they are calling Palin "Bible Spice" now. LOL #tweetdebate*

While it is rare to have equal numbers of positive and negative labels for a data item, contradicting opinions are widely presented in the dataset. One the one hand, this reflects the ambiguity of the human language. On the other hand, it shows that the sense of the sentence largely depends on the context. Since tweet is often a short sentence, the context is missing. Hence, the sense is conveyed less effectively.

In order to form a target classification, one possibility is to calculate the mean of ratings and round it to give the final class. This method suits perfectly if we have a multi-label grading system such as the following:

0 for negative

1 for somewhat negative

2 for neutral

3 for somewhat positive

4 for positive

Unfortunately, the rating system of the dataset is not such a grading. Therefore, it is not suitable in our case. With the presence label “others”, it cannot be rearranged to a grading system.

What we propose is a voting mechanism that takes the label assigned by most reviewers. One difficulty with this mechanism is when there is a tie, such as the two examples we listed above. Obviously, taking either label systematically would result a bias, while randomly choosing a label would create a moving target. In our system, we choose the first solution and select whatever label comes first when there is a tie. It is also worth mention that the total number of occurrence of such situation is 326, which is representing about 10% of the whole dataset.

### Training and test split

Data is split into training and test set. We use 70% of the data for training and 30% for testing. Data items are randomly chosen to be in the training or testing set.

## Part II: System

This part of the report describes the sentiment analysis system that we build for predicting political sentiment on twitter data. The system is built using programming language Python 2.7. Several external packages have been used, including numpy, scikit learn, nltk and matplotlib. Please ensure that these packages are installed before running the system.

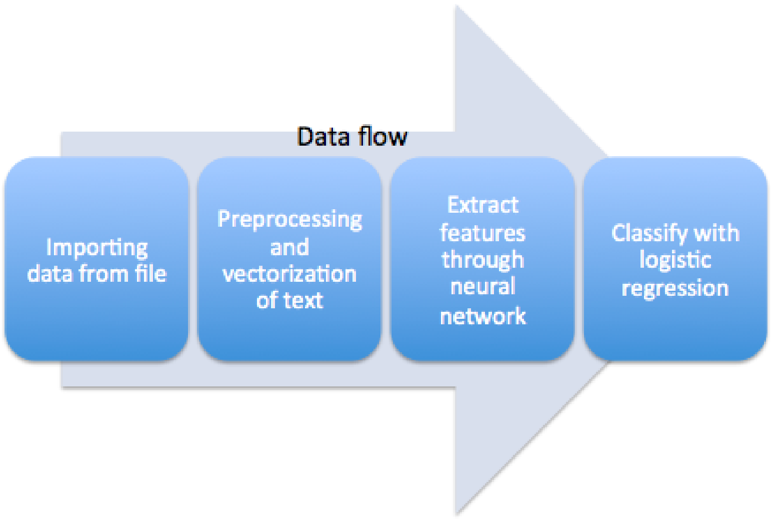

Data flow
Importing data from file
Preprocessing and vectorization of text
Extract features through neural network
Classify with logistic regression

Figure 1: Data flow through the sentiment analysis system

Figure 1 illustrates the data flow of the system. Data follows the following steps:

- **Importing data from file** Raw data is read in and stored in the runtime system as list of data point which contains the information needed for classification and evaluation.
- **Vectorisation of text** Preprocessing text before feeding into the classification system. Extracting features from context, for example time of the tweet and author.
- **Learning the feature vector** Semi-supervised recursive neural networks is used to learn the vector representation of the sentence.
- **Classification** With the vector representation of the sentence and context features, logistic regression is used to classify sentiment of the tweet.

## Importing data

Each data point is represented by a named tuple in the system. The tuple contains the following information:

1. tweet id
2. date
3. content
4. name
5. nickname
6. rating

Since each data item in the raw dataset could be manual rated by different reviewers, we provide a function to combine the manual ratings to form a target rating. As we've discussed in the previous section, a voting mechanism is used to form the target rating.
In our system, data importing functions are located in *corpus.py*.

## Vectorisation of strings

Before vectorise the tweet content, we first need a dictionary of vocabulary. The vocabulary is formed by parsing through the whole dataset. We save the vocabulary to a file in the data folder for later uses.

Tweet content is usually a short sentence. Upon receiving the content, the sentence is turned into a serie of tokens. By looking up the vocabulary, we transform the token to a number representing the place of that token in the dictionary. Hence, the whole sentence is transformed to a vector of numbers.

It is worth mention the tokenization of the tweeter data is slightly different than the usual text, because the use of hashtags, "@" and the fact that tweets often use informal wording. We use a tokenizer developed by Carnegie Mellon University that's specially designed for Twitter data. The difference can be visualised by the example below:

*@current #current How in this day and age can you deny help to anyone who wants to produce alternative fuel -4 McCain*

Normal tokenizer:
@, current, #, current, How, in, this, day, and, age, can, you, deny, help, to, anyone, who, wants, to, produce, alternative, fuel, -4, McCain

Twokenizer:
@current, #current, How, in, this, day, and, age, can, you, deny, help, to, anyone, who, wants, to, produce, alternative, fuel, -4, McCain

We can see that the Twokenizer function correctly identifies the topic which is identified by the hashtag and the referred user which is identified by '@'. Apart from the topic and reference, Twokenizer can also identify emotion expressions, for example :-), :(, ;D, ect.

Apart from the information in the tweet content, we also extract information about the author and when the it is tweeted. Because not everyone has a nicknames and the fact that nickname and name are almost the same, we choose to use author's name. These two features go into the recursive network with the vectorised text feature.

## Learning the feature using semi-supervised autoencoder

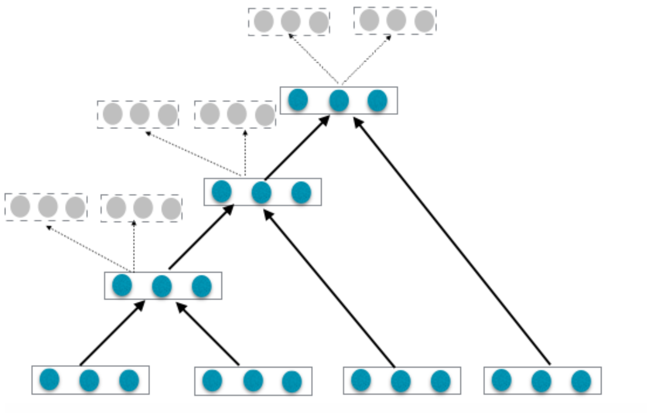
The machine learning model implemented with the system is a semi-supervised recursive autoencoder. This model combines the unsupervised learning method - the autoencoder and the supervised learning method - multi-label logistic regression in a deep neural network. The objective is that the system learns the correct word vector by going through the deep neural network. And these word vectors are used in the final sentiment prediction by a multi-label logistic regression.

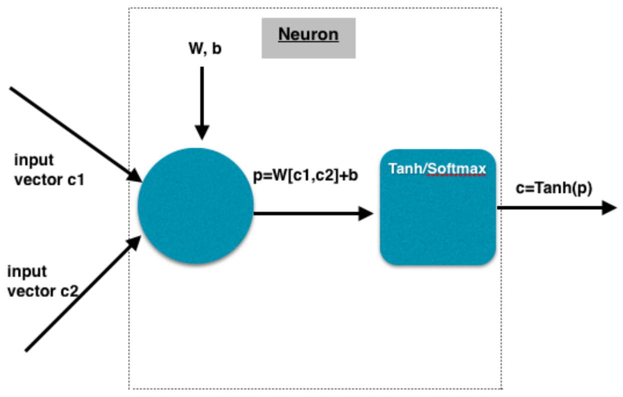
This approach is significantly different than the traditional supervised learning. In traditional supervised learning method, such as logistic regression or SVM, we need to manually extract features from the sentence. The most common features are ngram frequencies, word polarity, PMI measures, wordnet, etc. These features are summarised by human expert. In contrary, with our method, the recursive neural network are fed with raw sequence of words and finds the best feature representation of the classification problem for each sentence on its own.

Another difference is that unlike the traditional method, the recursive neural network results a dense vector. With the traditional supervised method, in order to get unigram frequencies feature, we can do a one hot encoding with the whole vocabulary. This gives us a vector of the size of the vocabulary, while for each sentence only a few of them are switched on. With the recursive neural network, the size of the vector representation of the sentence is a hyperparameter that can be determined by the user. Usually, the size of the vector is 25 - 1000.

## Description of the neural network

The recursive neural network used within our system is illustrated in Figure 2


(a) Structure of the recursive neural network



(b) A single neuron

Figure 2: Recursive neural network implemented by the system

Entering the neural network, each word is represented by a word vector $c_i$. $c_i$ can be found easily within the matrix $W_L$ which is formed by stacking together all the word vector of the vocabulary. The input to the unit in first layer of the network is the combination of the two adjacent words by stacking the two word vectors horizontally $[c_i, c_{i+1}]$. We use the *tanh* function in each neuron. And this results the output vector $p_1$ in the following form

$$p = tanh(W_1[c_i; c_{i+1}] + b_1)$$

where $W_1$ and $b_1$ are the two parameters that need to fit by the system.

To reconstruct the words $c_i$ and $c_{i+1}$ from $p_1$, we use the autoencoder. The reconstructed vector is

$$[c_i^{'}, c_{i+1}^{'}] = tanh(W_2 p + b_2)$$

where $W_2$ and $b_2$ are the two parameters that need to fit by the system and $c_i^{'}$ and $c_{i+1}^{'}$ are the reconstructed vectors of $c_i$ and $c_{i+1}$ respectively.

In parallel to the autoencoder, we have a supervised layer which takes the input $p$ and goes through a softmax function to predict the probability of each label given the vector at each node in the neural network.

$$d = softmax(W_{label}p)$$

The unit described above is repeated until the root of the sentence. At the root, we retrieve the feature vector of the sentence. This feature vector is then fed to the multi-class logistic regression to give the final prediction.

### Loss function

The loss function for the recursive neural network is the sum of the loss function of the autoencoder and that of the supervised unit.

1. Weighted reconstruction error
   The performance of the autoencoder unit is evaluated by the squared difference between the reconstruction of the vectors $c_i^{'}$ and $c_{i+1}^{'}$ and the original input vectors $c_i$ and $c_{i+1}$. If a node is not a leaf node, its error would represent a bigger importance than the leaf. In order to take into account the number of children the node, we use a weighted reconstruction error.

   $$E_{rec}([c_i; c_{i+1}]) = \frac{n_1}{n_1 + n_2} \left\| c_i^{'} - c_i \right\|^2 + \frac{n_2}{n_1 + n_2} \left\| c_{i+1}^{'} - c_{i+1} \right\|^2$$

   where $n_1$ and $n_2$ are the number of words under $c_i$ and $c_{i+1}$ respectively. One difficulty with this approach is that there can be numerous different structures of the neural network apply to the same data item. We can basically combine words in different sequence, and all of them give a valid structure of the network.

   A possible solution is to use the parse tree of the sentence for generating the structure of the network. However, building a parse tree itself needs to run a whole different algorithm and, in fact, it is very fast process. The approach we have in our system is simple build the network that leads to the smallest reconstruction error under the current parameter.

2. Squared error The supervised unit uses the squared error.

   $$E_{sup}(p, y) = \frac{1}{2} \|d - y\|^2$$

   where $d$ is the predicted probabilities and $y$ is the labels.

The overall loss function of the model is

$$J = \sum_{s \in \mathcal{D}} \alpha E_{rec}([c_i; c_{i+1}]_s) + (1 - \alpha) E_{sup}(p_s, y)$$

where $E_{rec}([c_i; c_{i+1}]_s)$ is the sum of weighted reconstruction error at each internal node for sentence $s$, $E_{sup}(p_s, y)$ is the sum of supervised error at each node for sentence $s$ and $\alpha$ is a hyperparameter which controls the proportion of the two error terms.

## Backpropagation

The optimisation problem illustrated above is solved by gradient descent. To calculate the gradient, we take the partial derivative of the loss against the parameter

$$\theta = \{[W_{11}, W_{12}], [W_{21}, W_{22}], W_L, W_{label}, b_1, [b_{21}, b_{22}], b_{label}\}$$

where $W_{11}$ and $W_{12}$ are square matrices that forms $W_1$, while $W_{21}$ and $W_{22}$ are square matrices that forms $W_2$.

$$\frac{\partial J}{\partial \theta} = \sum_{s \in \mathcal{D}} \alpha \frac{\partial E_{rec}([c_i, c_{i+1}]_s)}{\partial \theta} + (1-\alpha)\frac{\partial E_{sup}(p_s, y)}{\partial \theta}$$

To save space, we only derive the partial derivative for $W_{11}$ in this section, which is actually the most complex case. This will demonstrate how to do back propagation to the neural network. Take the root node $p_1$ of the tree, suppose it has two children $c_1$ and $c_2$. Let $n_1$ $n_2$ to be the number of leaves for $c_1$ and $c_2$ respectively. At the root level, a new reconstruction error is introduced $E_{rec}([c_1^{'}, c_2^{'}])$

$$\frac{\partial E_{rec}([c_1^{'}, c_2^{'}])}{\partial W_{11}} = \frac{2n_1}{n_1+n_2}(c_1^{'} - c_1)(\frac{\partial c_1^{'}}{\partial W_{11}} - \frac{\partial c_1}{\partial W_{11}}) + \frac{2n_2}{n_1+n_2}(c_2^{'} - c_2)(\frac{\partial c_2^{'}}{\partial W_{11}} - \frac{\partial c_2}{\partial W_{11}})$$

$c_1^{'}$ and $c_2^{'}$ are calculated as follows

$$c_1^{'} = tanh(W_{21}p_1 + b_{21})$$
$$c_2^{'} = tanh(W_{22}p_1 + b_{22})$$

So we can calculate the derivatives of them with regards to $W_{11}$.

$$\frac{\partial c_1^{'}}{\partial W_{11}} = tanh^{'}(W_{21}p_1 + b_{21})W_{21}\frac{\partial p}{\partial W_{11}}$$
$$\frac{\partial c_2^{'}}{\partial W_{11}} = tanh^{'}(W_{22}p_1 + b_{22})W_{22}\frac{\partial p}{\partial W_{11}}$$

We also know that for p

$$\begin{aligned}\frac{\partial p}{\partial W_{11}} &= \frac{\partial}{\partial W_{11}} tanh(W_{11}c_1 + W_{12}c_2 + b_1) \\ &= tanh^{'}(W_{11}c_1 + W_{12}c_2 + b_1)(c_1 + W_{11}\frac{\partial c_1}{\partial W_{11}} + W_{12}\frac{\partial c_2}{\partial W_{11}})\end{aligned}$$

Substitute the above equations to the original one and call the term below $parent_d(p_1)$,

$$parent_d(p_1) = \left(\frac{2n_1}{n_1+n_2}(c_1^{'} - c_1)tanh^{'}(W_{21}p_1 + b_{21})W_{21} + \frac{2n_2}{n_1+n_2}(c_2^{'} - c_2)tanh^{'}(W_{22}p_1 + b_{22})W_{22}\right) tanh^{'}(W_{11}c_1 + W_{12}c_2 + b_1)$$

It follows that

$$\frac{\partial E_{rec}([c_1^{'}, c_2^{'}])}{\partial W_{11}} = \left[parent_d(p_1)W_{11} - \frac{2n_1}{n_1+n_2}(c_1^{'} - c_1)\right]\frac{\partial c_1}{\partial W_{11}} + \left[parent_d(p_1)W_{12} - \frac{2n_2}{n_1+n_2}(c_2^{'} - c_2)\right]\frac{\partial c_2}{\partial W_{11}} + parent_d(p_1)c_1$$

For the root node, let $gradW_{11}(p_1) = parent_d(p_1)c_1$. Then we continue to the next level of the network where the root node would becomes $c1$ and $c_2$. Take $c_1$ for example, at this time, new reconstruction error $E_{rec}([c_{11}, c_{12}])$ is introduced to the node $c_1$. The above derivation is value for the new reconstruction error. In addition, we need to add the residual term $parent_d(p_1)W_{11} - \frac{2n_1}{n_1+n_2}(c_1^{'} - c_1)$ to the gradient. This would give us the following term

$$\begin{aligned} parent_d(c_1) = & \\ & \Big(\frac{2n_{11}}{n_{11}+n_{12}}(c_{11}^{'} - c_{11})tanh^{'}(W_{21}c_1 + b_{21})W_{21} \\ & + \frac{2n_{12}}{n_{11}+n_{12}}(c_{12}^{'} - c_{12})tanh^{'}(W_{22}c_1 + b_{22})W_{22} \\ & + parent_d(p_1)W_{11} - \frac{2n_1}{n_1+n_2}(c_1^{'} - c_1)\Big)tanh^{'}(W_{11}c_{11} + W_{12}c_{12} + b_1)\end{aligned}$$

Doing the above process recursively, the overall reconstruction error gradient is the sum of the gradient at each node. For the supervised error part, we could derive the same thing, only this time an additional term related to $W_lab$ is introduced into the $parent_d$. For the total gradient, we take the weighted sum of the gradient of reconstruction error and that of the supervised error. Realistically in the implementation, we first do a forward pass, where all the term tree structure and the vector representation of the nodes are generated. Then we do a back propagation to calculate the total loss and the gradient.

### Optimization

With the total loss and the gradient calculated, we can do gradient descent via a standard minFunc solver. In our system, we used the optimisation module that's provided by *scipy* package. We choose to use L-BGFS-B method, which provides an efficient way of solving the minimisation problem. The parameters of the solver can be adjusted via the model configuration file *model.json* in the data folder. For reference, we have limited the number of iteration to 70.

### Initialisation

To initialise the learning process, we generate the parameters of the neural network by sampling randomly from a uniform distribution between $-r$ and $r$, where $r$ is a small number, such as 0.001.

### Classification

With the feature vector found by the neural network, we then do a logistic regression to predict the label of each test sentence. The model used for the prediction is multi-class logistic regression. The model is fit with the training data with the cross-entropy loss

$$Loss = -\sum_k y_k^T log(f(x_k))$$

Then the test data goes through the fitted neural network and is fed into the fitted logistic regression function.

### Evaluation and analysis component

The system we build not only deals with the train and test process, it also has a component that carries out the cross-validation. We perform a 5-fold cross validation for the training data in order to find the optimal weight $\alpha$ that balances the reconstruction error and supervised error.
In addition, our system also provide a analysis component that projects the vector at each node of the tree to 2-dimension and visualise them on a diagram.

### Running the system

The detail of how to run the system is written in the *README.md* that can be found in the zip file.

## Part III: Experiments, results and discussions

In this section, we first use cross validate on the training set to find the best parameter for the recursive autoencoder model. Then, using the parameter found for the training set, we present the result for test set. Interesting insight discovered by the model are presented. Finally, we discuss how the model can be improved.

## Cross validation for parameter fitting

As we have discussed in the last section that the evaluation component of the system does a 5-fold cross validation on the training data in order to find the optimal $\alpha$. In Figure 3, the curve shows the cross validation result of $\alpha = 0.0, 0.2, 0.4, 0.6, 0.8, 1.0$

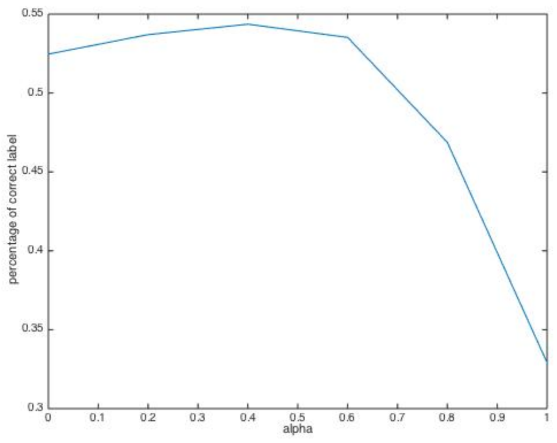


Figure 3: The cross validation result on function of $\alpha$

We can see here that the optimal is at $\alpha = 0.4$. And this is indeed the parameter that we use on the test set.

## Result on the test set

We train our machine learning model with the 70% of the whold data set and predict the label of the other 30%. The parameter used are the follows:

**- lowercase** True

**- number of dimension of the vector representation** 50

**- number of category to classify** 4

**- alpha** 0.4

**- lambdaW** $10^{-5}$

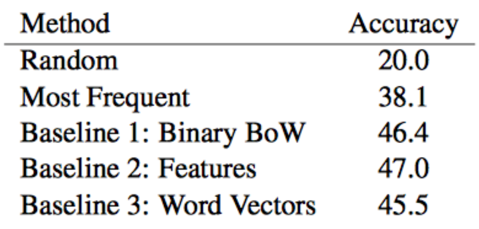
**- lambdaCat** $10^{-7}$

**- lambdaL** $10^{-4}$

**- iteration of the solver** 70

With the above configuration, we achieve an accuracy of 56.996%.

If we use a random baseline, this would give us 25% of accuracy. This signifies that our model indeed learned insight with the data as it performs significantly better than this baseline. Compare with the alternatives, We believe this is actually a pretty good result for a multi-label classification system. As we read through results from some of the papers, we can see that for a 5-class classifier, 45% is an average performance. Although we have a less challenging classification problem - only 4 classes are presented in the data, we also do significantly better than the average. (Figure 4)



| Method | Accuracy |
|---|---|
| Random | 20.0 |
| Most Frequent | 38.1 |
| Baseline 1: Binary BoW | 46.4 |
| Baseline 2: Features | 47.0 |
| Baseline 3: Word Vectors | 45.5 |

Figure 4: Performance for other classifiers on a 5-class problem

In the prediction result, we can see that sentences with a '+' or '-' sign are really easy to be classified. In fact, the token '+d' where d is a digit are among the highest scored token for positive label; and '-d' are among the highest for negative label. Some examples are provided below:

2 - "+5 Obama for alt energy, rural broadband, affordable college, healthcare #current #tweetdebate"
2 - "Obama +3 for wanting responsibility all the time #tweetdebate"
1 - "-5 Obama for continuing to say that McCain is right! Augh! Stop that! #tweetdebate"
1 - "-3 McCain for mispronounce of world leader's name. -3 Obama for trying to correct. #tweetdebate"

The system also successfully identifies some of the mixed altitude, which is usually quite difficult to do.
3 - "McCain definitely got his rear end handed to him tonight. Obama was simply on point in this debate."
3 - "Verdict: McCain exceeded all expectations. BHO: flat."

The most difficult thing in this classification is actually detect the off topic tweets, the system performs good as well
4 - "Jim Lehrer just directed the debate audience ... 30 seconds ..."
4 - "#current #tweetdebate can I vote for Jim Lehrer?"
4 - "#current #debate08 @wseltzer. Yes."

## Possible ways of improving the model

There are several aspects where we can further improve the semi-autoencoder model.
First of all, the preprocessing of the text can be more careful. As the tweet can be very informal, people tend to use permute words to mean the same thing. Here are some examples from our Tweeter corpus:
wayyyy
gaaaaaaah
Haaaa...
SLAAAAAAMMM
baaaaack
Heeeeeeeeeeeey

Apart from this, people also tend to use permutation of word whether intentionally or unintentionally due to a typographical error. Therefore, it is reasonable to think that with further preprocessing, the system can achieve a better result.

Improvement of the performance could also come from using a pre-learnt word embedding. Since learning the word vector is an unsupervised task, we do not need to restrain ourselves in using strictly the training data. In fact, we can initialise our word vectors $W_L$ by a readily formed word embedding library. In the following learning process within our system, the word vectors are pulled to the right direction for the prediction to work well.

## Conclusion

We presented a novel algorithm - semi-supervised autoencoder - that can accurately predict sentence-level sentiment analysis. Without using any hand-crafted features, our model achieves good performance on Tweeter datasets. This system is implemented within our sentiment analysis system which is downloadable via Github. In addition, further improvements can be made to make the system even more accurate. We are looking forward to more advancement of technics in deep learning for information retrieval and data mining tasks.