Report
Team Entropy

Members:
Abhishek Pratap Singh (214101002)
Harsh Bijwe (214101020)
Kishore M (214101062)
Murtaza Saifee (214101031)

Fake news has become a serious issue with the explosion of decentralised information generation. To effectively tackle it, one needs an automated solution. There have been various efforts at tackling the problem.
Majority of extant fake news detection techniques focus on exogenous signals. They generally ignore the endogenous preference of the user when he/she decides to spread a piece of fake news.
In the given paper, the problem of exploiting user preference for fake news detection is looked at.

Reason for looking at endogenous preference of user:

- **Confirmation bias:** A user is more likely to spread fake news if it confirms his/her existing beliefs/preferences. So the historical data of user provides a rich information about users' preferences
- **Fact checking is labour intensive** and requires domain experts. So this alternative method is more useful.

Major components of integrating user preference:

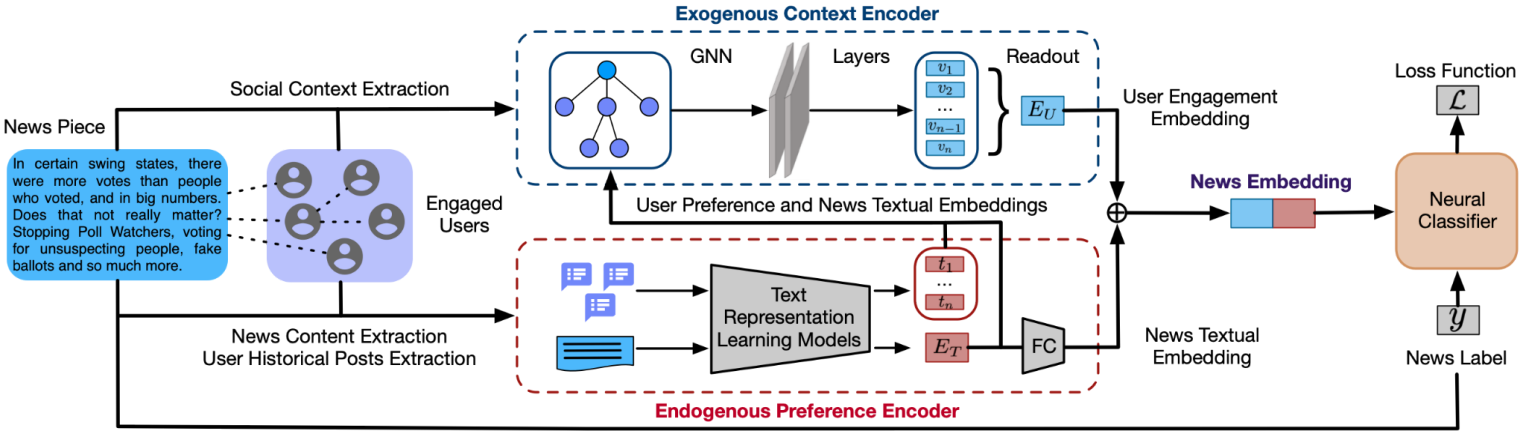
1. **User endogenous preference** in which we encode news content and user historical posts
2. **User exogenous context,** a tree-structured propagation graph for each news based on its sharing cascading on social media
3. **Integrate the endogenous and exogenous information**: we take the vector representation of news and users as their node features and employ Graph Neural networks to learn a joint user engagement embedding

Now this user engagement embedding and news textual embedding are used to train a neural classifier to detect fake news.

Dataset

The **FakeNewsNet dataset** is used. Additionally for the users who have interacted with the newspiece in the dataset, we have **collected profile information**. The additional information crawled from twitter includes details about **user profile, previous 200 tweets for each user.**

Architecture of proposed method



The proposed architecture proceeds in 3 major steps:

1. **Endogenous Preference Encoding**
   This step includes using the **historical posts of a user** to encode his/her preference. We have previous 200 tweets for each user. Some preprocessing over the text of tweets is done which involves removing special characters, urls. Next step is **encoding news textual information** and user preferences. 2 methods of generating embeddings are employed. First is using pretrained **spacy word2vec vectors** and other is using **pretrained BERT embeddings**.

2. **Exogenous Context Extraction**
   User exogenous context is **composed of all users that are engaged with the news.** We **build a news propagation graph** using retweet information of news pieces. To build a propagation graph, we define a news piece as v_1, and $\{v_2, \dots, v_n\}$ as a list of users that retweeted v_1 ordered by time. Here are two rules to determine news propagation path:
   a. For account v_i, if v_i retweets the same news later than at least one of the following accounts in $\{v_1, \dots, v_n\}$, we estimate the news spreads from the account with the latest timestamp to account v_i.
   b. If account v_i does not follow any accounts in the retweet sequences including the source account, we can conservatively estimate the news spread from the accounts with the most number of followers

3. **Information fusion**
   Fusing the user features with a news propagation graph could boost fake news detection performance. GNN can encode both node feature and graph structure.
   Author's have used a hierarchical information fusion approach. First step is **fusing endogenous and exogenous information using GNN**. The news textual embedding and user preference are taken as node features. Given news propagation graph, GNN aggregates the features of its adjacent nodes to **learn the embedding of a node**. A **mean pooling over all node embeddings** is done to get the **graph embedding.**

As news content contains more explicit signals of news credibility, we fuse the news textual embedding and user engagement embedding by concatenation. This forms the ultimate news embedding.

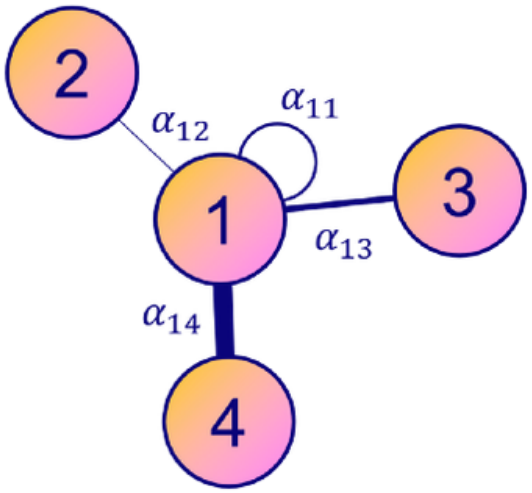
This **ultimate news embedding is fed into a 2-layer multilayer perceptron (MLP).**

Our work

The model provided by the author's was GNN. We ran the GNN on both dataset and with different preprocessing steps. Two available options for the dataset are gossipcop and politifact. Two different ways of preprocessing include: spacy word2vec is used for embedding and other where BERT is used for generating embeddings.

The improvement we suggested is to include **Multihead attention** into the graph itself. The intuition behind this using this is that:-

In GCN, every neighbour has the same importance.
Self attention in GNNs relies on a simple idea: **nodes should not all have the same importance**.
In GAT however we are looking for giving more attention to a particular node which is learnt by Neural Network itself.



$$h_1 = \alpha_{11}\mathbf{W}x_1 + \alpha_{12}\mathbf{W}x_2 + \alpha_{13}\mathbf{W}x_3 + \alpha_{14}\mathbf{W}x_4$$

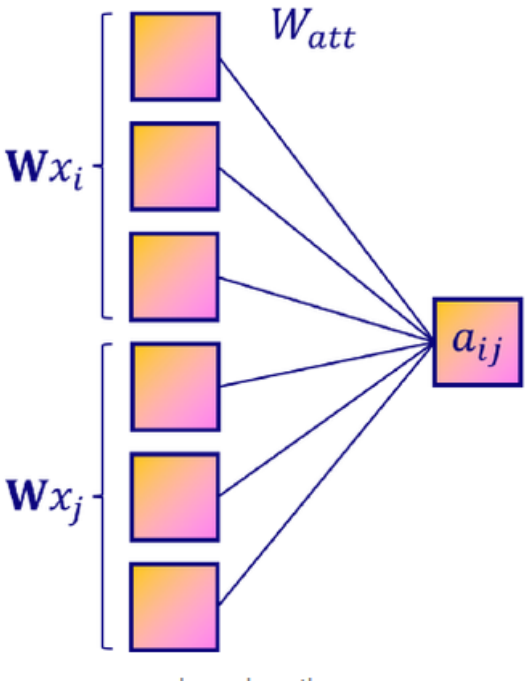
There are 3 steps in the process:

1. **Linear transformation**
2. **Activation function**
3. **Softmax normalisation**

Linear transformation

We want to calculate the importance of each connection, so we need pairs of hidden vectors. An easy way to create these pairs is to concatenate vectors from both nodes.

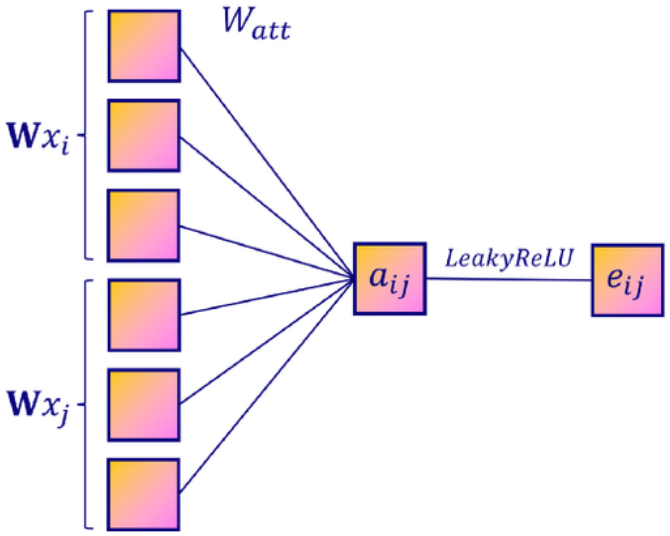
$$a_{ij} = W_{att}^{t}[\mathbf{W}x_i \parallel \mathbf{W}x_j]$$



Activation function

We're building a neural network, so the second step is to add an activation function. In this case, the authors of the paper chose the *LeakyReLU* function.

$$e_{ij} = LeakyReLU(a_{ij})$$

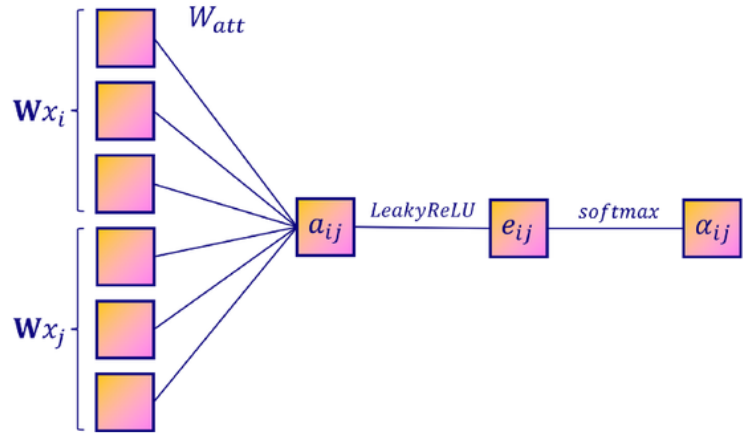


Softmax normalisation

The output of our neural network is **not normalised**, which is a problem since we want to compare these scores. To be able to say if node 2 is more important to node 1 than node 3 ($\alpha_{12} > \alpha_{13}$), we need to share the same scale.

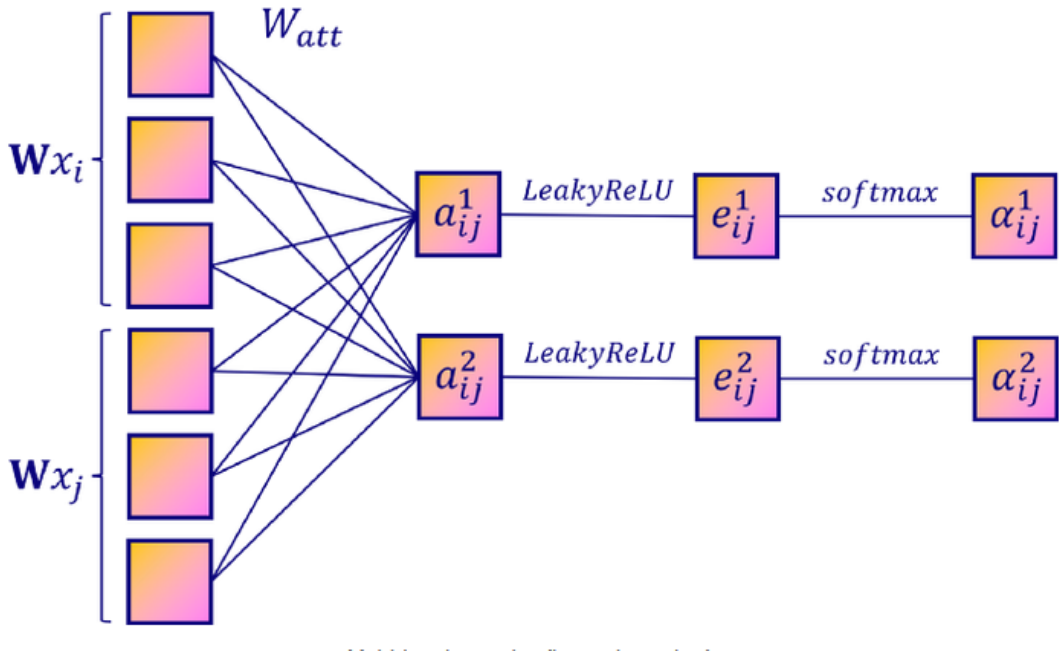
A common way to do it with neural networks is to use the ***softmax*** function. Here, we apply it to every neighbouring node:

$$\alpha_{ij} = softmax_j(e_{ij}) = \frac{exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} exp(e_{ik})}$$



GAT + Multihead Attention:

In GATs, multi-head attention consists of **replicating the same 3 steps several times** in order to average or concatenate the results. That's it. Instead of a single h_i, we get one hidden vector h_i^k per attention head. We can do Average or Sum the different h_i^k and normalise the result by the number of attention heads n.



In practice, we use the concatenation scheme when it's a hidden layer, and the average scheme when it's the last layer of the network.

Results and observation

For gossipcop the number of epochs used is 40
For politifact the number of epochs used is 200. The size of the dataset in politifact is small, this is the reason more epochs are needed.

Model evaluation values for GCN (provided model)

Dataset + Embedding	Test Accuracy	Test F1 score
Gossipcop + word2vec	0.94	0.94
Gossipcop + BERT	0.83	0.85
Politifact + word2vec	0.72	0.84
Politifact + BERT	0.82	0.81

Interesting observation:

- When working on politifact with BERT embeddings, the training loss goes to zero. This means our model is overfitting. This is also evident from the fact that test accuracy in the BERT case is less than that of the case when word2vec is used. In word2vec case, training loss does not become zero, rather it is 0.18.

Model evaluation values for GAT with multihead attention (Our model)

Dataset + Embedding	Test Accuracy	Test F1 score
Gossipcop + word2vec	0.94	0.93
Gossipcop + BERT	0.85	0.85
Politifact + word2vec	0.77	0.80
Politifact + BERT	0.82	0.82

Generating Endogenous Preference encoding

We also have written **code for generating user preference embedding**. For this, the tweet text for previous 200 tweets is extracted for each user. The text is then preprocessed which involves removing special characters, urls, etc. All the unique words in the tweets are stored in a list of strings.
For generating embedding, we have used pretrained spacy word2vec embedding. The size of output embedding is 300. The embedding generated over all the words is then averaged. This averaged embedding over all the words in the tweet gives us the user preference embedding.
The embedding generated is stored as a .npy file. There is also mapping which gives the userId of tweets whose embedding is stored.
[**Note**: As this take ~400 hrs to execute, we stop after tweets for 10 users have been read, to show the working of this file]