

<https://github.com/Raj-Sanjay-Shah/Twignet>

TextGCN:

The paper differs from traditional graphical neural networks in the manner it forms the graph.

The graph is formed in the following manner:

Nodes: Consists of Documents and unique words.

Number of nodes = Corpus size + Vocabulary size

X(feature matrix) is an n*n identity matrix

The edges are built based on word occurrence in documents (document-word edges) and word co-occurrence in the whole corpus (word-word edges). The weight of document-word edges are TF_IDF values. For weights of word-word edges we use a fixed size sliding window to calculate a Point wise Mutual Information measure.

$$A_{ij} = \begin{cases} \text{PMI}(i, j) & i, j \text{ are words, } \text{PMI}(i, j) > 0 \\ \text{TF-IDF}_{ij} & i \text{ is document, } j \text{ is word} \\ 1 & i = j \\ 0 & \text{otherwise} \end{cases}$$

The PMI value of a word pair i, j is computed as

$$\text{PMI}(i, j) = \log \frac{p(i, j)}{p(i)p(j)}$$

$$p(i, j) = \frac{\#W(i, j)}{\#W}$$

$$p(i) = \frac{\#W(i)}{\#W}$$

$\#W(i)$ is the number of sliding windows in a corpus that contain word i

$\#W(i, j)$ is the number of sliding windows that contain both word i and j

$\#W$ is the total number of sliding windows in the corpus.

A two-layer GCN can allow message passing among nodes that are at maximum two steps away. Although there are no direct document-document edges in the graph, the two-layer GCN allows the information exchange between pairs of documents.

$$E_1 = \tilde{A}XW_0$$

$$E_2 = \tilde{A} \text{ReLU}(\tilde{A}XW_0)W_1$$

$$\text{Softmax}(E_2)$$

Results:

Table 2: Test Accuracy on document classification task. We run all models 10 times and report mean \pm standard deviation. Text GCN significantly outperforms baselines on 20NG, R8, R52 and Ohsumed based on student t -test ($p < 0.05$).

Model	20NG	R8	R52	Ohsumed	MR
TF-IDF + LR	0.8319 \pm 0.0000	0.9374 \pm 0.0000	0.8695 \pm 0.0000	0.5466 \pm 0.0000	0.7459 \pm 0.0000
CNN-rand	0.7693 \pm 0.0061	0.9402 \pm 0.0057	0.8537 \pm 0.0047	0.4387 \pm 0.0100	0.7498 \pm 0.0070
CNN-non-static	0.8215 \pm 0.0052	0.9571 \pm 0.0052	0.8759 \pm 0.0048	0.5844 \pm 0.0106	0.7775 \pm 0.0072
LSTM	0.6571 \pm 0.0152	0.9368 \pm 0.0082	0.8554 \pm 0.0113	0.4113 \pm 0.0117	0.7506 \pm 0.0044
LSTM (pretrain)	0.7543 \pm 0.0172	0.9609 \pm 0.0019	0.9048 \pm 0.0086	0.5110 \pm 0.0150	0.7733 \pm 0.0089
Bi-LSTM	0.7318 \pm 0.0185	0.9631 \pm 0.0033	0.9054 \pm 0.0091	0.4927 \pm 0.0107	0.7768 \pm 0.0086
PV-DBOW	0.7436 \pm 0.0018	0.8587 \pm 0.0010	0.7829 \pm 0.0011	0.4665 \pm 0.0019	0.6109 \pm 0.0010
PV-DM	0.5114 \pm 0.0022	0.5207 \pm 0.0004	0.4492 \pm 0.0005	0.2950 \pm 0.0007	0.5947 \pm 0.0038
PTE	0.7674 \pm 0.0029	0.9669 \pm 0.0013	0.9071 \pm 0.0014	0.5358 \pm 0.0029	0.7023 \pm 0.0036
fastText	0.7938 \pm 0.0030	0.9613 \pm 0.0021	0.9281 \pm 0.0009	0.5770 \pm 0.0049	0.7514 \pm 0.0020
fastText (bigrams)	0.7967 \pm 0.0029	0.9474 \pm 0.0011	0.9099 \pm 0.0005	0.5569 \pm 0.0039	0.7624 \pm 0.0012
SWEM	0.8516 \pm 0.0029	0.9532 \pm 0.0026	0.9294 \pm 0.0024	0.6312 \pm 0.0055	0.7665 \pm 0.0063
LEAM	0.8191 \pm 0.0024	0.9331 \pm 0.0024	0.9184 \pm 0.0023	0.5858 \pm 0.0079	0.7695 \pm 0.0045
Graph-CNN-C	0.8142 \pm 0.0032	0.9699 \pm 0.0012	0.9275 \pm 0.0022	0.6386 \pm 0.0053	0.7722 \pm 0.0027
Graph-CNN-S	–	0.9680 \pm 0.0020	0.9274 \pm 0.0024	0.6282 \pm 0.0037	0.7699 \pm 0.0014
Graph-CNN-F	–	0.9689 \pm 0.0006	0.9320 \pm 0.0004	0.6304 \pm 0.0077	0.7674 \pm 0.0021
Text GCN	0.8634 \pm 0.0009	0.9707 \pm 0.0010	0.9356 \pm 0.0018	0.6836 \pm 0.0056	0.7674 \pm 0.0020

Advantages of using GCN for text classification:

- Multiple benchmark datasets demonstrate that a vanilla Text GCN without any external word embeddings or knowledge outperforms state-of-the-art methods for text classification.
- Text GCN also learns predictive word and document embeddings.
- The improvement of Text GCN over state-of-the-art comparison methods become more prominent as we lower the percentage of training data, suggesting the robustness of Text GCN to less training data in text classification.

Limitations of GCN for text classification: GCN model is inherently transductive, in which test document nodes (without labels) are included in GCN training. Thus Text GCN can not quickly generate embeddings and make predictions for unseen test documents.

Code:

Understanding the file structure:

- data/corpus/mr.txt: Contains all the documents with each document being one line of the file in its raw format.
- data/corpus/mr.clean.txt: Contains all the documents with each document being one line of the file in its cleaned format(post removal of rare words and stop words).
- data/mr.txt: Contains 'document_id', 'train/test' and '1/0'(binary class).
- data/mr.train.index: Shuffled indices of the train set documents.
- data/mr.test.index: Shuffled indices of the test set documents.
- data/mr_shuffle.txt: Contains the indices of all the documents in shuffled manner (appended lists data/mr.train.index and data/mr.test.index)
- data/corpus/mr_shuffle.txt: Contains the documents in the same shuffled manner according to the indices of the file above.

- data/corpus/mr_vocab.txt: Contains the entire vocabulary.
- data/corpus/mr_vocab_def.txt: Stores the dictionary meaning of words in our vocabulary.
- data/corpus/mr_word_vectors.txt: Stores the embeddings for words of the vocabulary.
- data/corpus/mr_labels.txt: Contains the unique labels of the data.
- data/mr.real_train.name: Contains 90% of the train set (id, train, 1/0)

The implementation of the python scripts are as follows:

remove_words.py

1. Read the data/corpus/mr.txt line by line. Each line consists of a document
2. Store the frequency of each unique word in all the documents in a dictionary
3. Remove stop words from each document and remove less frequent words (threshold=5). This is done for all datasets except mr. Store the list of cleaned documents in data/corpus/mr.clean.txt
4. Calculate the metadata about corpus like minimum document size, maximum document size and average document size.

build_graph.py:

1. Set the embedding dimension size as 1000
2. Make four lists:
 - a. doc_name_list = [document_id, train/test, 1/0]
 - b. doc_train_list = [document_id, train, 1/0]
 - c. doc_test_list = [document_id, test, 1/0]
 - d. doc_content_list // read cleaned data from the previous step, data/corpus/mr.clean.txt
3. Get all the row numbers/ids of the doc_train_list elements in the doc_name_list into the list train_ids and do a similar process for test_ids. Randomly shuffle both lists and write to data/mr.train.index and data/mr.test.index respectively.
4. Make a new list containing the shuffled train and test ids and write into data/mr_shuffle.txt. The corpus of the same indices is written into data/corpus/mr_shuffle.txt.
5. Then we make a dictionary of {word, frequency} and also build a list of unique words (vocabulary) and write the list into data/corpus/mr_vocab.txt.
6. Make a dictionary word_doc_list that stores { word, list of documents the word appears in}. We also make two more dictionaries of
 - a. word_doc_freq = {word, number of documents the word appears in}
 - b. word_id_map = {word, index of word in vocabulary}
7. (*Commented Code*) Using wordnet.synset we get the definition of all the words in our vocabulary and store it in data/corpus/mr_vocab_def.txt. We run the tf-idf vectorizer on the word definitions to build a document- word matrix(corpus_size, 1000) and limit the max number of features to 1000 (builds a vocabulary that only considers the top max_features ordered by term frequency across the corpus). We effectively convert each word in the vocabulary to an embedding based on the word meanings as the corpus. We write the word vectors to data/corpus/mr_word_vectors.txt.

Word_Vocabulary	Meaning	Word-1	Word-2	Word-3...	Word-1000
Word 1	Meaning(Document)	tf-idf_word1	tf-idf_word2	tf-idf_word3	tf-idf_word1000

8. We get the unique labels from the data/corpus/mr_shuffle.txt file and write to data/corpus/mr_labels.txt. We separate out 90% of the train set and store it in data/mr_real_train.name.
9. Then we add the embedding value for each word present in the document from the previous step and normalize it using the document length.
for each word in document,

$$document_{vector} = (\sum word_{vector}) / document_{length}$$

For each document i we create:

row	column(word_embedding_dimension)	data($document_{vector}$)
Document id, i	1	1st value
Document id, i	...1000	1000th value

We input this data into a compressed sparse row matrix. `csr_matrix((data_x, (row_x, col_x)), shape=(real_train_size, word_embeddings_dim))`. Note that `real_train_size` is 90% of the total train set size. We then one hot encode the label and store it in a list. For mr dataset this list is of the form `[[1,0], [1,0], [0,1]...[1,0]]`. We perform the same steps for the entirety of the test set.

10. We create another sparse matrix of the dimensions (train_size + vocab_size, word_embeddings_dim) and the label matrix of dimension (train_size + vocab_size, label_size). Note that the one hot encoding array consists of all zeros for the vocabulary.
11. The next step is to find the word co-occurrence in the same window. We first create a list of all the windows of size (window_size). We then create a dictionary {word, number of windows the word appears in}. We then create a dictionary - word_pair_count consisting of {(word1_id, word2_id), the number of windows the words occur together in}.
12. We then calculate the PMI between words used from the data collected from the above step.

```
row = [ ]
col = [ ]
weight = [ ]
for key in word_pair_count:
    temp = key.split(',')
    i = int(temp[0])
    j = int(temp[1])
    count = word_pair_count[key]
    word_freq_i = word_window_freq[vocab[i]]
    word_freq_j = word_window_freq[vocab[j]]
    pmi = log((1.0 * count / num_window) /
              (1.0 * word_freq_i * word_freq_j / (num_window * num_window)))
    if pmi <= 0:
```

```

        continue
    row.append(train_size + i)
    col.append(train_size + j)
    weight.append(pmi)

```

13. Then we construct a dictionary with {(document_id,word_id), frequency}. Subsequently, we add document, word weights to the row, col, and weight lists above using the following code:

```

for i in range(len(shuffle_doc_words_list)):
    doc_words = shuffle_doc_words_list[i]
    words = doc_words.split()
    doc_word_set = set()
    for word in words:
        if word in doc_word_set:
            continue
        j = word_id_map[word]
        key = str(i) + ',' + str(j)
        freq = doc_word_freq[key]
        if i < train_size:
            row.append(i)
        else:
            row.append(i + vocab_size)
            col.append(train_size + j)
            idf = log(1.0 * len(shuffle_doc_words_list) /
                    word_doc_freq[vocab[j]])
            weight.append(freq * idf)
            doc_word_set.add(word)

```

14. With this we finally obtain the adjacency matrix:

```

node_size = train_size + vocab_size + test_size
adj = sp.csr_matrix(
    (weight, (row, col)), shape=(node_size, node_size))

```

15. We dump the four sparse matrices and matrices of respective labels formed till now using pickle:

```

f = open("data/ind.{}.x".format(dataset), 'wb')
pkl.dump(x, f)
f.close()
f = open("data/ind.{}.y".format(dataset), 'wb')
pkl.dump(y, f)
f.close()
f = open("data/ind.{}.tx".format(dataset), 'wb')
pkl.dump(tx, f)
f.close()
f = open("data/ind.{}.ty".format(dataset), 'wb')
pkl.dump(ty, f)
f.close()
f = open("data/ind.{}.allx".format(dataset), 'wb')
pkl.dump(allx, f)
f.close()
f = open("data/ind.{}.ally".format(dataset), 'wb')

```

```

pkl.dump(ally, f)
f.close()
f = open("data/ind.{}.adj".format(dataset), 'wb')
pkl.dump(adj, f)
f.close()

```

train.py:

1. Load the corpus from the file. We load/generate the following data structures:
 - a. features: (train_size + vocab_size + test_size, word_embeddings_dim) Matrix (N,1000)
 - b. y_train, y_val, y_test: Individual label arrays.
 - c. train_mask, test_mask, val_mask: these are masking arrays of features and labels.
 - d. train_size, test_size
 - e. Adjacency matrix(adj): We make adj symmetric in the following manner:

```
adj = adj + adj.T.multiply(adj.T > adj) - adj.multiply(adj.T > adj)
```

```
adj=
  1      2      0
1/5      1    1/2
3/10    1/5      0
```

```
adj.T=
1    1/5    3/10
2      1    1/5
0    1/2      0
```

```
adj.T>adj
false false true
true  false false
false true  false
```

```
adj.T.multiply(adj.T>adj)
0      0    3/10
2      0      0
0    1/2      0
```

```
adj.multiply(adj.T>adj)
0      0      0
1/5    0      0
0    1/5      0
```

```
adj + adj.T.multiply(adj.T>adj) - adj.multiply(adj.T>adj)
```

```

  1      2    3/10
  2      1    1/2
3/10    1/2      0

```

The need to make the matrix symmetric in such fashion comes from the fact that we added the tf-idf value to (document, word) entry in the matrix but did not add it to (word, document).

2. We edit the feature matrix (features) to make it into an identity matrix of (train_size + vocab_size + test_size, train_size + vocab_size + test_size) size.
3. We then normalize the adjacency matrix in the following manner:

```
def normalize_adj(adj):  
    """Symmetrically normalize adjacency matrix."""  
    adj = sp.coo_matrix(adj)  
    rowsum = np.array(adj.sum(1))  
    d_inv_sqrt = np.power(rowsum, -0.5).flatten()  
    d_inv_sqrt[np.isinf(d_inv_sqrt)] = 0.  
    d_mat_inv_sqrt = sp.diags(d_inv_sqrt)  
    return adj.dot(d_mat_inv_sqrt).transpose().dot(d_mat_inv_sqrt).tocoo()
```

4. We build a GCN model with the following parameters:
 - a. num_support (1): Number of supporting matrices (adj in this case)
 - b. number of features
 - c. Labels
 - d. Dropout (default to 0)
5. The built model has the following properties:
 - a. Adamoptimizer
 - b. Learning rate=0.02
 - c. Embedding dimension of hidden layer = 200
 - d. Masked_softmax_cross_entropy error
 - e. Relu activation function

FastGCN:

The proposed approach, coined FastGCN, not only rids the reliance on the test data but also yields a controllable cost for per-batch computation. Faster than GraphSAGE

Drawbacks with GCN:

1. Requires both train and test data at the same time.
2. The recursive neighborhood expansion across layers poses time and memory challenges for training with large, dense graphs.
3. Non additive nature of loss function makes it difficult to use SGD. Also, data samples are not independent.

In [mathematics](#), an integral transform maps a [function](#) from its original [function space](#) into another function space via [integration](#), where some of the properties of the original function might be more easily characterized and manipulated than in the original function space. The transformed function can generally be mapped back to the original function space using the inverse transform.

What this paper does is view graph convolutions differently and interpret them as integral transforms of embedding functions under probability measures.

GAT:

1. Self attention Layers
2. Feed forward layers

Uses self attention for determining the importance of an edge

<https://dsgittr.com/blogs/gat/>