Assignment

정주호

2022712955
인공지능융합학과

테이블이나 그림 추가 가능합니다.
추가적인 섹션이 필요하다면 추가하셔도 됩니다.

1. Problem Definition

Covid19에 관련된 가짜 뉴스를 Graph Neural Network기법을 활용하여 분류하는 과제이다. 해당 문제의 해결이 중요한 이유는 Covid19의 등장과 함께 Covid19에 관련된 근거가 없는 루머나 여론몰이식의 가짜 뉴스들이 SNS를 통해 검열이 안되고 빠르게 확산되었다. 이런 검열 없는 루머, 가짜 뉴스의 확산은 해당 지식을 분별할 수 있는 능력이 없는 일반인들, 청소년들에게 잘못된 정보를 전달하여 신체적, 정신적, 금전적 피해를 야기시켰다. SNS의 발전으로 뉴스매체의 정보 전달보다 SNS에서의 전달 속도가 더 빠르고, 대부분의 첫 이슈가 SNS에서 나오며, 그 파급력이 일반 뉴스 보도매체보다 훨씬 크다[1]. 이번 이태원 사태에서도 보도 초기에 가짜 뉴스와 루머와 혐오 발언들이 SNS상에서 빠르게 확산되어 큰 문제가 되었고, 결국 유투브와 인스타그램에서 자체적으로 검열 시스템이 작동된 사태가 있었다. 따라서 뉴스 기사가 정확한 팩트인지, 가짜 뉴스인지를 구분하는 문제가 중요하다. 이 과제에서는 가짜 뉴스 기사 분별을 GNN의 Node Classification으로 수행하고자한다.

2. Data

2.1. Data Description

데이터는 Covid19에 대한 가짜 뉴스를 판별하기 위한 데이터셋이다. Covid19와 관련이 있는 데이터셋과 (FakeCovid (Shahi and Nandini, 2020), CoAID (Cui and Lee, 2020)), Non-Covid19와 관련된 (fact-checking 웹사이트 : Politifact, Snopes) 뉴스 기사들을 크롤링해서 수집하였다. 데이터는 아래와 같은 구조(Columns)로 이루어져있다.
뉴스기사별로 고유의 id값이 부여되고, 해당 뉴스기사가 일반뉴스라면 0, 가짜뉴스라면 1로 라벨링 되어있다. Published_date은 뉴스의 출간 날짜이고, net_keywords, keybert_keyowords는 각각 Bert-Ner, keyBERT를 이용해서 claim text에서 keyword를 추출한 것이다.

Columns

- `id` - 고유 id
- `label` - label (test.csv 에는 없음) / 0은 일반 뉴스, 1은 가짜 뉴스
- `claim` - fakenews claim
- `published_date` - 출간 날짜
- `ner_keywords` - BERT-NER를 이용하여 `claim` text에서 keyword 추출
- `keybert_keywords` - KeyBERT를 이용하여 `claim` text에서 keyword 추출
- `youtube0~9` - `ner_keywords` or `keybert_keywords`로 Youtube 검색하여 상위 10개의 관련도 높은 비디오의 title과 description 추출

2.2. Data Preprocessing

뉴스의 Claim을 전처리하기 위해서 크게 두가지 텍스트 전처리 과정을 진행하였다.

2.2.1 Tokenizer

먼저 뉴스의 Claim이 영어 텍스트이기 때문에 이 문장을 단어 단위로 token화를 진행해주었다. 이때 Tokenizer로는 다양한 Tokenizer를 활용해서 그 성능을 비교해보았고, 사용한 Tokenizer는 아래 3번에 자세하게 설명하였다.

2.2.2 Stop words 제거 및 대문자 소문자 변환

뉴스 Claim를 문장 단위로 인식하고 이를 단어의미 단위의 토큰으로 token화를 진행하였다. 이때 영어 단어적으로 의미가 없는 stop word를 제외시켜주었고, tokenizer에서 토큰화할때, 대문자들을 전부 소문자로 바꿔서 토큰화해주는 파라미터가 있는데 이를 False로 두어 소문자와 대문자의 구분을 살려주었다. 그 이유는 해당 텍스트가 뉴스이기 때문에 대문자로 갖는 의미가 있을 것이라 생각했기 때문이다.

3. Analysis

3.1. Counter Vectorizer

Count Vectorize는 단어에 피처 값을 부여할 때, 각 문장에서 해당 단어가 나타나는 횟수(count)를 반영하여 부여한다. 즉, 단어의 빈도수가 높을 수록 중요한 단어로 반영된다

➔ 본 과제에서의 활용: GCN의 입력 Feature로 전체 뉴스(5277개)의 Claim에서 단어들을 토큰화하고 각 단어에 피쳐값을 부여할 때 Count Vectorizer를 사용하였다. 이 Count Vectorizer에서 나온 Feature들을 모델의 Input Feature로 넣어주었다. 이때 Max Features의 크기를 설정해줄 수 있는데, 3000개에서부터 순차적으로 줄여가면서 실험을 진행하였고 max_features=2000일 때 성능이 가장 좋았다.

3.2. Tfidf Vectorizer

TF-IDF(Term Frequency-Inverse Document Frequency)의 약자로 Counter Vectorizer의 문제점을 보완하였다. 카운터 기반의 벡터화는 단순히 단어의 빈도수만을 고려하기 때문에, 문장에서 어쩔 수 없이 자주 나올 수 밖에 없는 'the' 'a'등의 단어를 가장 중요하다고 판단할 수 있다. Tfidf Vectorizer는 카운터 벡터화처럼 개별 문장에서 자주 등장하는 단어에 높은 가중치를 부여하면서 동시에 모든 문서에서 전반적으로 자주 등장하는 단어에 대해서 패널티를 주는 방식으로 작동한다. 가령, 모든 문장에서 자주 등장 할 수 밖에 없는 'the', 'a'등은 모든 문장에서 자주 등장함으로 중요하지 않기 때문에 패널티를 부여한다.

➔ 본 과제에서의 활용: GCN의 입력 Feature로 전체 뉴스(5277개)의 Claim에서 단어들을 토큰화하고 각 단어에 피쳐값을 부여할 때 TF-IDF Vectorizer를 사용하였다. 이 Tfidf Vectorizer에서 나온 Feature들을 모델의 Input Feature로 넣어주었다. 이때 3.1에서의 실험 결과를 바탕으로 max_features=2000으로 맞춰주었다.

3.3. Pretrained BERT Ouput Layer Feature

BERT는 2018년에 발표되었지만 여전히 NLP분야와 그외 다양한 분야에서 강력하게 작동한다[4] BERT이전의 모델들은 문장을 학습할 때 순차적인 방향(왼쪽에서 오른쪽으로)으로 문장을 학습하고 읽지만, BERT는 Bidirectional Transformer로 양방향으로 학습하고 예측이 가능하다. BERT는 NLP 11분야에서 모두 State of art성능을 보였다.

➔ 본 과제에서의 활용: 뉴스의 Claim을 먼저 Pretrained BERT Tokenizer로 토큰화한다.

	id	label	claim	bert
0	0	0	Did a Vermont Woman Post a Joke About Trump Br...	[did, a, vermont, woman, post, a, joke, about,...
1	1	1	Were Baseball Players Photographed Kneeling to...	[were, baseball, players, photographed, kneeli...
2	2	0	Wisconsin state Rep. John Nygren hits pay dirt...	[wisconsin, state, rep, ., john, ny, ##gren, h...

그 다음 Bert tokenizer의 convert_tokens_to_ids를 사용하여 전체 단어 토근에 대한 정수 인코딩을 수행한다. 이후 전체 뉴스 Claims(5277개)의 문장의 워드 토큰 중 가장 큰 길이를 계산하고 (96개) 전체 5277개의 서로 다른 개수의 문장 워드 토큰을 zero padding을 통하여 길이를 맞춰주었다.

```
tokenized_text = [tokenizer.tokenize(sen) for sen in train_df.claim.values]

max_lenth = 0
for i in tokenized_text:
  if len(i) >= max_lenth:
    max_lenth = len(i)

print(max_lenth)
```

```
96
```

```
from keras_preprocessing.sequence import pad_sequences

print('padding')
MAX_LEN = 100
input_ids = [tokenizer.convert_tokens_to_ids(x) for x in tokenized_text]
# input_ids = pad_sequences(input_ids, maxlen = MAX_LEN, dtype='long', truncating='post', padding='post')
input_ids[0]
print('input_ids[0]:',input_ids[0])
```

```
padding
input_ids[0]: [2106, 1037, 8839, 2450, 2695, 1037, 8257, 2055, 8398, 5026, 2067, 8864, 1029]
```

4. Model

4.1. Classifier

4.1.1 Graph Convolutional Networks(GCN)[2]

GCN은 노드 v의 이웃 노드들과(neighbor) 자기 자신(self node) 모두에 대해서 동일한 파라미터를 사용한다는 것이 핵심이다.

4.1.2 GraphSAGE[3]

GraphSAGE는 GCN과 다르게 노드와 이웃 노드들의 가중치 값을 더하지 않고 Concat한다는 것이 핵심이다.

4.2. Design Consideration

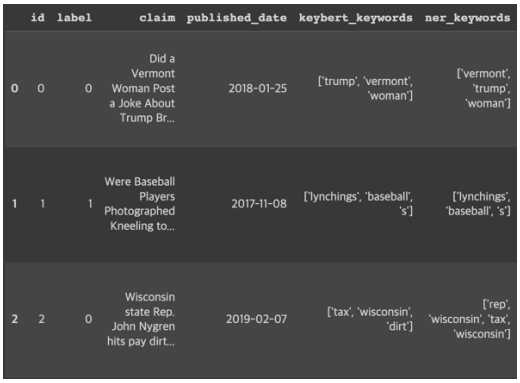
Classifier로 위에서 언급한 GCN, GraphSAGE모델을 둘 다 사용하고 성능 비교 실험을 하면서 둘 중에 하나의 모델을 확정지었다.

	Accuracy
GCN	0.63
GraphSAGE-layer3	0.684
GraphSAGE-layer2	0.707

➔ GraphSAGE를 최종 Classifier 모델로 확정지었다. 이때 GCN과 GraphSAGE에서 layer는 짧게는 2개부터 많게는 4개까지 순차적으로 쌓아가면서 실험을 진행하였고 이를 비교하면서 최종 모델 GraphSAGE, 최종 Layer=3을 확정지었다.

```
class GCN(nn.Module):
    def __init__(self, in_feats, h_feats,num_classes):
        super(GCN, self).__init__()
        self.conv1 = GraphConv(in_feats, h_feats)
        self.conv2 = GraphConv(h_feats, num_classes)

    def forward(self, g, in_feat):
        h = self.conv1(g, in_feat)
        h = F.relu(h)
        h = self.conv2(g, h)
        return h


from dgl.nn import SAGEConv

# build a two-layer GraphSAGE model
class GraphSAGE(nn.Module):
    def __init__(self, in_feats, h_feats,h_feat,num_classes):
        super(GraphSAGE, self).__init__()
        self.conv1 = SAGEConv(in_feats, h_feats, 'mean')
        self.conv2 = SAGEConv(h_feats, h_feat, 'mean')
        self.conv3 = SAGEConv(h_feat, num_classes, 'mean')
        self.dropout = nn.Dropout(p=0.4)

    def forward(self, g, in_feat):
        h = self.conv1(g, in_feat)
        h = F.relu(h)
        h = self.dropout(h)
        h = self.conv2(g, h)
        h = F.relu(h)
        h = self.dropout(h)
        h = self.conv3(g, h)
        return h
```

4.3. Graph Building



	id	label	claim	published_date	keybert_keywords	ner_keywords
0	0	0	Did a Vermont Woman Post a Joke About Trump Br...	2018-01-25	['trump', 'vermont', 'woman']	['vermont', 'trump', 'woman']
1	1	1	Were Baseball Players Photographed Kneeling to...	2017-11-08	['lynchings', 'baseball', 's']	['lynchings', 'baseball', 's']
2	2	0	Wisconsin state Rep. John Nygren hits pay dirt...	2019-02-07	['tax', 'wisconsin', 'dirt']	['rep', 'wisconsin', 'tax', 'wisconsin']

데이터에서 id를 Node로, Claim에서 추출한 feature를 Node Feature로 사용하였다. 또한 Edge의 경우 두 뉴스기사가 Keybert_keywords나 ner_keywords에서 겹치는 부분이 있으면 연결하였다. 이때, Keybert_keywords와 ner_keywords중 하나만 사용하여 edge를 구성하는 것보다, 둘다 사용하여 두 뉴스기사의 키워드가 겹치면 edge를 연결하는 방법의 정확도가 더 높았다.

	Counter Vectorizer or Tfidf Vectorizer	Bert Tokenizer
Node	5277	5277
Edge	1382295	1382295
Node Feature	100	768
Label	0,1	0,1

Train Mask: Test Mask : Validation Mask = 3800 : 1055: 422

➔ **이렇게 구성하게 된 배경은 '뉴스 기사의 Keyword가 같다면 뉴스의 내용, 성격이 비슷할거다' 라는 가정에 근거하였다.**

Edge	Accuracy
Keybert_kewords	0.632
Ner_kewords	0.64
Both	0.684

5. Experiments

5.1. Settings

먼저 Train에 사용된 데이터셋은 Train.csv파일에 있는 5277개의 데이터셋을 사용했다. 이 데이터셋을 Trainset 70%, Testset 20%, Valset 10%로 나누어서 실험을 진행했다. 학습은 프로세스는 Trainset으로 모델을 학습하면서 Valset으로 모델을 검증하고, train_loss와 val_loss에 기반하여 파라미터를 선정하였다. 또한 Unseen dataset에 대한 모델 성능 검증을 위해 Testset을 예측하고, 모델의 예측값 y_pred 와 Testset의 실제 label값과의 정확도를 비교하여서 최종 모델을 선정하였다.

```
[ ]  nids = np.random.permutation(np.arange(g.number_of_nodes()))
     test_size = int(len(nids) * 0.2)
     test_id = nids[:test_size]

     train_id = nids[test_size:]
     val_size = int(len(train_id) * 0.1)

     val_id = train_id[:val_size]
     train_id = train_id[val_size:]


     print("train_size: ", len(train_id), "test_size: ", len(test_id),"val_size: ", len(val_id) )

     train_size:  3800 test_size:  1055 val_size:  422
```

➔ 학습을 통해 확정한 모델을 가지고 Test.csv파일에 있는 1856개의 unlabeled된 데이터셋을 예측하는 방식으로 진행하였다.

5.2. Performance Metrics

평가 지표의 기준으로는 Accuracy를 사용했다. Accuracy는 True를 True로, False를 False로 분류하는 정확도를 의미한다. Accuracy는 데이터의 비율에 큰영향을 받기 때문에 원래 평가 지표로 Accuracy만을 고려하진 않는다. 그러나 본 프로젝트에서는 Kaggle에서 높은 성능을 받아야함으로 Accuracy를 기준으로 평가지표로 설정하였다. 그리고 나머지 Precision, recall, F1-score에 대해서는 모델이 하나의 label로만 찍고 있지는 않은지, 0과1의 라벨을 어떤 분포로 예측하고 있는지를 참고하여 활용하였다.

```
              precision    recall  f1-score   support

           0       0.81      0.65      0.72       661
           1       0.56      0.74      0.64       394

    accuracy                           0.68      1055
   macro avg       0.68      0.69      0.68      1055
weighted avg       0.71      0.68      0.69      1055

[Test result] loss: 0.718, test acc: 0.682
```

5.3. Results

모델을 학습하면서 train_loss, val_loss가 동시에 줄어드는지를 확인하면서 학습의 overfitting을 평가하였고, Test_accuracy를 보면서 unseen dataset에 대해서 모델의 정확도를 평가하였다.

Best Model Performance

TfidfVectorizer(Max_features=2000) / GraphSAGE(layer=3, out_featurs=256,64, dropout=0.4)

```
In epoch 0, loss: 0.568, train acc: 0.675, val acc: 0.618
In epoch 1, loss: 0.539, train acc: 0.708, val acc: 0.600
In epoch 2, loss: 0.511, train acc: 0.750, val acc: 0.661
In epoch 3, loss: 0.487, train acc: 0.765, val acc: 0.623
In epoch 4, loss: 0.464, train acc: 0.795, val acc: 0.664
In epoch 5, loss: 0.448, train acc: 0.797, val acc: 0.640
In epoch 6, loss: 0.429, train acc: 0.812, val acc: 0.694
In epoch 7, loss: 0.416, train acc: 0.822, val acc: 0.659
In epoch 8, loss: 0.399, train acc: 0.828, val acc: 0.694
In epoch 9, loss: 0.388, train acc: 0.833, val acc: 0.668
In epoch 10, loss: 0.369, train acc: 0.849, val acc: 0.692
In epoch 11, loss: 0.367, train acc: 0.846, val acc: 0.668
In epoch 12, loss: 0.351, train acc: 0.856, val acc: 0.687
In epoch 13, loss: 0.346, train acc: 0.856, val acc: 0.678
In epoch 14, loss: 0.332, train acc: 0.873, val acc: 0.701
[Test result] loss: 0.641, test acc: 0.686
```

➔ 학습을 보면, Train accuracy와 Validation accuracy, Test Accuarcy모두 올라가는 것을 볼 수 있다. 그러나 Train loss가 0.33으로 계속 줄어가지만 Test Loss는 0.641보다 줄어들지 않았다. 이는 Trainset에 대한 모델의 학습은 잘 이루어지지만, Validation/Test에 대해서 모델의 학습이 어느정도 이상의 성능 향상이 일어나지 않고 있다. 이 문제를 해결하기 위해서 Dropout Layer를 추가로 두어서 Test acc를 향상시켰다. 더 좋은 방법으로는 Validation set에 대한 val loss를 함께 줄여가는 식의 학습 방법을 Model Check point를 통해서 Val loss가 가장 낮을때의 Best point를 저장해서 사용하는 방법이 있다.

Best Model Performance

Tfidf Vectorizer (Max_features=2000) / GraphSAGE (layer=1, out_features= 128, dropout=0.4)

```
In epoch 0, loss: 0.700, train acc: 0.486, val acc: 0.526 (best 0.526)
In epoch 1, loss: 0.624, train acc: 0.665, val acc: 0.578 (best 0.578)
In epoch 2, loss: 0.577, train acc: 0.689, val acc: 0.595 (best 0.595)
In epoch 3, loss: 0.549, train acc: 0.692, val acc: 0.588 (best 0.588)
In epoch 4, loss: 0.515, train acc: 0.725, val acc: 0.630 (best 0.630)
In epoch 5, loss: 0.487, train acc: 0.756, val acc: 0.621 (best 0.621)
In epoch 6, loss: 0.467, train acc: 0.772, val acc: 0.637 (best 0.637)
In epoch 7, loss: 0.442, train acc: 0.789, val acc: 0.640 (best 0.640)
In epoch 8, loss: 0.427, train acc: 0.808, val acc: 0.656 (best 0.656)
In epoch 9, loss: 0.412, train acc: 0.816, val acc: 0.652 (best 0.652)
In epoch 10, loss: 0.394, train acc: 0.828, val acc: 0.668 (best 0.668)
In epoch 11, loss: 0.383, train acc: 0.834, val acc: 0.654 (best 0.654)
In epoch 12, loss: 0.367, train acc: 0.840, val acc: 0.673 (best 0.673)
In epoch 13, loss: 0.356, train acc: 0.850, val acc: 0.649 (best 0.649)
In epoch 14, loss: 0.346, train acc: 0.856, val acc: 0.659 (best 0.659)
In epoch 15, loss: 0.339, train acc: 0.859, val acc: 0.645 (best 0.645)
In epoch 16, loss: 0.328, train acc: 0.864, val acc: 0.664 (best 0.664)
In epoch 17, loss: 0.317, train acc: 0.869, val acc: 0.671 (best 0.671)
In epoch 18, loss: 0.309, train acc: 0.874, val acc: 0.649 (best 0.649)
In epoch 19, loss: 0.303, train acc: 0.877, val acc: 0.664 (best 0.664)
In epoch 20, loss: 0.293, train acc: 0.881, val acc: 0.673 (best 0.673)
In epoch 21, loss: 0.287, train acc: 0.882, val acc: 0.664 (best 0.664)
In epoch 22, loss: 0.277, train acc: 0.892, val acc: 0.668 (best 0.668)
In epoch 23, loss: 0.274, train acc: 0.893, val acc: 0.656 (best 0.656)
In epoch 24, loss: 0.266, train acc: 0.898, val acc: 0.673 (best 0.673)
In epoch 25, loss: 0.261, train acc: 0.898, val acc: 0.673 (best 0.673)
In epoch 26, loss: 0.258, train acc: 0.902, val acc: 0.685 (best 0.685)
In epoch 27, loss: 0.253, train acc: 0.900, val acc: 0.666 (best 0.666)
In epoch 28, loss: 0.243, train acc: 0.908, val acc: 0.671 (best 0.671)
In epoch 29, loss: 0.240, train acc: 0.911, val acc: 0.671 (best 0.671)
In epoch 30, loss: 0.237, train acc: 0.909, val acc: 0.666 (best 0.666)
In epoch 31, loss: 0.232, train acc: 0.909, val acc: 0.666 (best 0.666)
In epoch 32, loss: 0.228, train acc: 0.913, val acc: 0.652 (best 0.652)
In epoch 33, loss: 0.222, train acc: 0.917, val acc: 0.680 (best 0.680)
In epoch 34, loss: 0.220, train acc: 0.918, val acc: 0.656 (best 0.656)
[Test result] loss: 0.708, test acc: 0.707
```

➔ 마찬가지로 Trainset에 대해서는 loss는 줄고 Accuracy는 90프로대까지 오르지만 Val accuracy는 따라가지 못하는 것을 확인할 수 있다. Val accuracy가 줄고 있는 것이 아니라 같이 오르고 있긴 하기때문에 학습이 되고 있고, 완전한 overfitting이 일어난 것은 아니라고 판단된다. 그러나 Train, Test, Validation을 전부 Random permutation을 통해서 무작위로 뽑았는데 Train accuracy와 Validation accuracy가

오르는 속도가 다르기 때문에 GraphSAGE 모델의 학습 과정에서 Validation loss를 고려하지 않고 모델이 학습하고 있기 때문이라고 결론지었다. 이후 이를 보완할 수 있는 모델 학습 방법을 도입하여 해결하려고 한다.

➔ 최종 Performance: TestSet에 대해서 70.7% 정확도

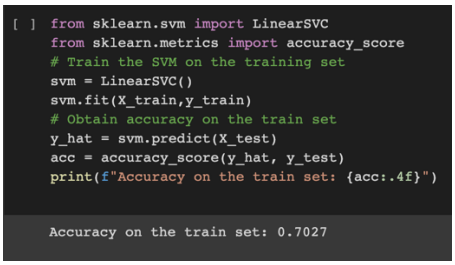
6. Discussion and Limitation

6.1. 학습의 문제

위에서 언급한대로 먼저 학습과정에서의 문제점이 있다. Train Test Validation을 Random shuffle하여 비율대로 나누어서 데이터에 편향은 없었다. 그러나 모델이 학습하는 과정에서 Train accuracy와 Validation accuracy가 둘 다 같이 올라서 모델이 학습은 하고 있는 것을 확인하였지만, Train accuracy가 가파르게 오르는데 반면, Validation accuracy는 천천히 오르는 것을 확인하였다. 이는 모델이 학습할 때 validation set에 대해서 고려하지 않고, Trainset에 대해서만(Train loss를 줄이는 방향으로) 맞춰서 학습이 일어나고 있기 때문이다. Dropout layer를 모든 conv layer에 추가하면서 이 문제를 해결하려 노력했고, dropout의 비율을 변경시켜가면서 그 간격을 줄일 수 있었다. 그럼에도 불구하고 위에서 언급하였듯이 train performance와 validation performance의 차이가 여전히 존재하기때문에 이는 Model checkpoint등의 학습 method를 추가하여 해결할 수 있다고 생각한다.

6.2. Bert Feature의 사용의 문제

Baseline을 SVM 모델로 하고, Model Input을 Claim값만 넣고 Unseen data를 예측했는데 정확도가 73프로가 나왔다. 이 말은 해당 뉴스의 Claim의 내용이 가짜뉴스인지 진짜뉴스인지를 구분하는데 굉장히 큰 연관이 있다는 것을 의미한다. 본프로젝트는 Kaggle의 성능을 올리는 것이 중요하다. 본강의에서는 Graph 모델들과 학습, method에 대해서 주로 다뤘다면, 이번 프로젝트에서는 Kaggle의 성능을 올리는 방법을 총동원해서 좋은 Feature를 뽑고, 이를 잘 활용하는 것이 중요하다.



```
[ ] from sklearn.svm import LinearSVC
    from sklearn.metrics import accuracy_score
    # Train the SVM on the training set
    svm = LinearSVC()
    svm.fit(X_train,y_train)
    # Obtain accuracy on the train set
    y_hat = svm.predict(X_test)
    acc = accuracy_score(y_hat, y_test)
    print(f"Accuracy on the train set: {acc:.4f}")

    Accuracy on the train set: 0.7027
```

따라서 NLP에서 가장 많이 사용되는 Bert모델을 사용하여 뉴스의 Claim을 전처리하고, 토큰화하고, Feature를 추출하는 과정을 진행하였다. 이 Bert모델의 output결과의 마지

막 layer의 값(shape=768)을 GraphSAGE의 노드 Feature로 넣어주었다.

```
Number of layers: 3   (initial embeddings + 12 BERT layers)
Number of batches: 1
Number of tokens: 100
Number of hidden units: 768
```

➔ 그러나 아래 결과에서 확인할 수 있듯이 BERT output을 활용하여 GraphSAGE의 Input feature로 넣어줘서 학습한 결과 모델이 전혀 학습을 하지 못하는 것을 볼 수 있다. 분명 Claim을 Bert로 Embedding하여서 Graph의 Node Feature로 넣어줘서 Node classification을 하면 더 좋은 결과가 있을거라 기대했지만 모델이 전혀 학습하고 있지 못하는 것을 보아, (1) pretrained Bert의 last layer ouput을 잘못 가져왔거나, (2) Bert output feature를 잘못 처리한 것 같다.

```
In epoch 15, loss: 2.207, train acc: 0.501, val acc: 0.474 (best 0.474)
In epoch 16, loss: 2.809, train acc: 0.515, val acc: 0.557 (best 0.557)
In epoch 17, loss: 2.465, train acc: 0.494, val acc: 0.509 (best 0.509)
In epoch 18, loss: 3.649, train acc: 0.514, val acc: 0.547 (best 0.547)
In epoch 19, loss: 1.861, train acc: 0.499, val acc: 0.469 (best 0.469)
In epoch 20, loss: 3.040, train acc: 0.509, val acc: 0.540 (best 0.540)
In epoch 21, loss: 4.247, train acc: 0.497, val acc: 0.486 (best 0.486)
In epoch 22, loss: 3.472, train acc: 0.517, val acc: 0.545 (best 0.545)
In epoch 23, loss: 3.811, train acc: 0.499, val acc: 0.450 (best 0.450)
In epoch 24, loss: 3.392, train acc: 0.506, val acc: 0.538 (best 0.538)
In epoch 25, loss: 1.561, train acc: 0.501, val acc: 0.512 (best 0.512)
In epoch 26, loss: 2.518, train acc: 0.522, val acc: 0.536 (best 0.536)
In epoch 27, loss: 3.333, train acc: 0.487, val acc: 0.467 (best 0.467)
In epoch 28, loss: 3.127, train acc: 0.513, val acc: 0.538 (best 0.538)
In epoch 29, loss: 2.997, train acc: 0.499, val acc: 0.467 (best 0.467)
In epoch 30, loss: 3.246, train acc: 0.513, val acc: 0.552 (best 0.552)
In epoch 31, loss: 1.750, train acc: 0.518, val acc: 0.505 (best 0.505)
In epoch 32, loss: 2.411, train acc: 0.536, val acc: 0.564 (best 0.564)
In epoch 33, loss: 2.441, train acc: 0.497, val acc: 0.453 (best 0.453)
In epoch 34, loss: 1.893, train acc: 0.541, val acc: 0.571 (best 0.571)
In epoch 35, loss: 2.102, train acc: 0.491, val acc: 0.462 (best 0.462)
In epoch 36, loss: 1.904, train acc: 0.529, val acc: 0.547 (best 0.547)
In epoch 37, loss: 1.534, train acc: 0.505, val acc: 0.474 (best 0.474)
In epoch 38, loss: 1.517, train acc: 0.546, val acc: 0.557 (best 0.557)
In epoch 39, loss: 0.934, train acc: 0.535, val acc: 0.505 (best 0.505)
In epoch 40, loss: 0.995, train acc: 0.530, val acc: 0.559 (best 0.559)
In epoch 41, loss: 1.130, train acc: 0.509, val acc: 0.521 (best 0.521)
In epoch 42, loss: 0.732, train acc: 0.547, val acc: 0.533 (best 0.533)
In epoch 43, loss: 0.798, train acc: 0.553, val acc: 0.531 (best 0.531)
In epoch 44, loss: 1.639, train acc: 0.521, val acc: 0.550 (best 0.550)
In epoch 45, loss: 0.754, train acc: 0.564, val acc: 0.564 (best 0.564)
In epoch 46, loss: 1.231, train acc: 0.532, val acc: 0.576 (best 0.576)
In epoch 47, loss: 1.412, train acc: 0.508, val acc: 0.467 (best 0.467)
In epoch 48, loss: 1.004, train acc: 0.563, val acc: 0.583 (best 0.583)
In epoch 49, loss: 0.793, train acc: 0.547, val acc: 0.498 (best 0.498)
[Test result] loss: 1.005, test acc: 0.500
```

```
from tqdm import tqdm
sentence_vec = []

model = BertModel.from_pretrained('bert-base-uncased',
                                  output_hidden_states = True,
                                  )
model.eval()

for i,j in zip(tqdm(input_ids),attention_masks):
  tokens_tensor = torch.tensor([i])
  segments_tensors = torch.tensor([j])


  with torch.no_grad():
    outputs = model(tokens_tensor, segments_tensors)
    hidden_states = outputs[2]


  token_vecs = hidden_states[-2][0]
  # # Calculate the average of all 36 token vectors.
  sentence_embedding = torch.mean(token_vecs, dim=0)
  sentence_vec.append(sentence_embedding)
```

```
Some weights of the model checkpoint at bert-base-uncased were not used
- This IS expected if you are initializing BertModel from the checkpoin
- This IS NOT expected if you are initializing BertModel from the check
100%|■■■■■■■■■■| 5277/5277 [38:44<00:00,  2.27it/s]
```

➔ (2)번을 조금 더 구체적으로 설명하면, BERT의 output의 shape이 (#layer, #batch, #token, #feature)인데, 여기서 #token=100으로 설정했고, #feature=768이기 때문에 이 한문장에 대해서 이 100개의 값을 평균내야지 sentence vector가 된다고 생각했다. 한문장에 대해서 (100,768)이 생성되는데 이를 평균내서 (768)을 만들었고 전체 5277개의 News Claim이 전부 768개의 vector를 갖게 되었다. 이 과정에서 저 (100,768)의 벡터를 평균내면 그 문장의 Sentence Vector가 된다는 잘못된 생각을 한 것 같다.

7. References

[1] What is Twitter, a social network or a news media?

[2] Kipf, T. N., & Welling, M. (2016). Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907.

[3] GraphSage: Inductive Representation Learning on Large Graphs(2017, Neurips)
[4] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.
[5]https://riverkangg.github.io/nlp/nlp-bertWordEmbedding/
[6]https://medium.com/@claude.feldges/text-classification-with-tf-idf-lstm-bert-a-quantitative-comparison-b8409b556cb3
[7]https://riverkangg.github.io/nlp/nlp-bertWordEmbedding/