IMDB - Word2Vec with LSTM

> ## NLP(Natural Language Processing)

```
import warnings
warnings.filterwarnings('ignore')
```

Import Keras

- Keras Version 확인

```
import keras

keras.__version__
```

```
'2.4.3'
```

I. IMDB Data_Set

1) Load IMDB Data_Set

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
!ls -l /content/drive/MyW Drive/ColabW Notebooks/datasets/IMDB.zip
```

```
-rw------- 1 root root 60711700 Mar 21 01:09 '/content/drive/My Drive/Colab Notebooks/dataset
```

```
!unzip /content/drive/MyW Drive/ColabW Notebooks/datasets/IMDB.zip
```

2) 'texts' and 'labels' Data

- 'texts' : 문자열 리스트(영화 감상평)
- 'lebels' : 감상평 리뷰(긍정/부정)

```
import os

imdb_dir = 'aclImdb'
train_dir = os.path.join(imdb_dir, 'train')

labels = []
texts = []

for label_type in ['neg', 'pos']:
    dir_name = os.path.join(train_dir, label_type)
    for fname in os.listdir(dir_name):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname), encoding = 'utf8')
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)
```

```
len(labels), len(texts)
```

```
(25000, 25000)
```

II. Tensor Transformation

1) X_train and X_valid : (25000, 2000)

- vectorization
  - (25000, 2000)

```
import numpy as np
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences

maxlen = 2000               # 2000개 단어까지 적용
max_words = 10000           # 빈도 높은 10000개 단어 사용

tokenizer = Tokenizer(num_words = max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
print('%s개의 고유한 토큰을 찾았습니다.' % len(word_index))

# (25000, 2000)으로 패딩
data = pad_sequences(sequences, maxlen = maxlen)
```

```
labels = np.asarray(labels)
print('데이터 텐서의 크기:', data.shape)
print('레이블 텐서의 크기:', labels.shape)


# 샘플 데이터 랜덤화
indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
labels = labels[indices]


# 데이터를 훈련 세트와 검증 세트로 분할
training_samples = 15000           # 훈련 샘플은 15000개
validation_samples = 10000         # 검증 샘플은 10000개

X_train = data[:training_samples]
y_train = labels[:training_samples]
X_valid = data[training_samples: training_samples + validation_samples]
y_valid = labels[training_samples: training_samples + validation_samples]
```

```
88582개의 고유한 토큰을 찾았습니다.
데이터 텐서의 크기: (25000, 2000)
레이블 텐서의 크기: (25000,)
```

III. Word2Vec

https://tfhub.dev/google/Wiki-words-250-with-normalization/2

1) Load Pretrained Word2Vec

```
import tensorflow_hub as hub

embeddings_index = hub.load('https://tfhub.dev/google/Wiki-words-250/2')
```

- Word2Vec 매핑 확인

```
embeddings_index(['apple'])
```

```
<tf.Tensor: shape=(1, 250), dtype=float32, numpy=
array([[-7.81319588e-02, -7.97718316e-02,  1.65636316e-02,
        -1.08001232e-02, -5.10360440e-03,  1.73767412e-03,
        -5.22104278e-02, -3.89753021e-02,  3.56903672e-02,
        -3.47909741e-02, -1.01491222e-02,  1.17565657e-03,
         1.01802059e-01,  1.75360963e-02,  3.36469710e-02,
         2.79656947e-02,  9.57141817e-02, -7.82085657e-02,
         5.06314561e-02, -1.66016668e-01,  2.88206208e-02,
         6.76635057e-02,  9.70917642e-02,  1.79236010e-02,
        -5.42766303e-02, -1.56506345e-01, -5.30809052e-02,
```

```
-1.09254161e-03, -1.59554277e-02, -6.70691356e-02,
 6.15172908e-02,  4.47090678e-02,  4.07696068e-02,
-3.83969024e-02,  6.96176291e-02, -5.60147781e-03,
-2.26747449e-02, -3.67878452e-02, -5.66431917e-02,
-2.18681507e-02, -8.86453837e-02, -2.22746611e-01,
-2.07957737e-02,  8.30694276e-04, -3.93166617e-02,
 5.93367852e-02,  4.33623493e-02,  2.82799695e-02,
-3.33397463e-02,  4.76312377e-02, -1.70567650e-02,
-2.29284037e-02, -6.83813961e-03,  9.81669277e-02,
 1.08622149e-01,  2.33869702e-02,  9.60301422e-03,
-5.18808290e-02,  9.87230614e-03, -8.69225897e-03,
-7.11247921e-02,  7.64381886e-02, -2.43284814e-02,
-1.53728379e-02, -1.62205976e-02,  6.06783573e-03,
-7.78824538e-02, -4.63543087e-02, -6.10455871e-02,
 3.76226790e-02, -3.59688103e-02, -2.22809706e-03,
 7.61934593e-02, -5.42140864e-02,  8.81553534e-03,
 1.02721518e-02, -2.33051814e-02,  5.69647923e-02,
-1.97233241e-02,  5.40813841e-02, -1.06068708e-01,
-4.93893884e-02,  7.71986991e-02,  7.54192844e-02,
-9.59494933e-02,  1.31888255e-01, -9.32045653e-02,
 3.91011648e-02,  9.07641053e-02, -1.92628205e-02,
-2.80017368e-02,  1.24782704e-01, -4.39715805e-03,
-6.76676631e-02,  4.82756756e-02, -8.52152854e-02,
 2.47792583e-02, -1.15225026e-02, -5.91450334e-02,
 5.01944311e-02,  6.20031133e-02, -5.91100939e-02,
-6.27844781e-02,  1.29692315e-03, -1.05377071e-01,
-9.93670970e-02,  3.99318524e-05,  1.49191227e-02,
 4.78936266e-03, -1.33457212e-02,  3.32747735e-02,
 1.46604422e-02,  4.45817932e-02, -6.77856291e-03,
 8.64909738e-02, -5.00659496e-02,  6.05830625e-02,
-5.85512519e-02, -7.41328374e-02, -4.07381281e-02,
-4.62190509e-02, -2.91668661e-02, -4.27213088e-02,
-8.50341916e-02, -7.91059509e-02, -6.71506599e-02,
 1.28828017e-02, -1.12139061e-02, -2.05859430e-02,
 2.96257101e-02, -1.02398887e-01, -7.77662396e-02,
-1.40929092e-02, -9.11662430e-02, -6.77241236e-02,
-7.13218153e-02, -2.31174957e-02, -5.25913984e-02,
 5.41850226e-03,  4.89527360e-02, -1.23757459e-02,
 9.59252100e-03, -2.92501189e-02,  6.46212921e-02,
-6.40202463e-02,  1.71411615e-02, -1.26278475e-01,
 6.52319491e-02, -6.71177506e-02, -1.82377342e-02,
 1.48606636e-02,  5.96186668e-02, -3.32654566e-02,
 1.99540146e-03, -5.77870682e-02,  5.53884953e-02,
 1.13689369e-02, -7.79620459e-05, -5.87577112e-02,
 7.56517053e-02,  4.84508127e-02, -3.61890532e-02,
-1.79408665e-03,  2.08812468e-02,  2.92749479e-02,
-9.95378569e-02, -2.77001262e-02,  1.80020705e-02,
 2.97355093e-02, -1.23234659e-01, -5.09306304e-02,
-3.30761299e-02. -5.26242852e-02. -3.00878249e-02.
```

2) 임베딩 행렬 생성

- (10000, 250)

```
embedding_dim = 250

embedding_matrix = np.zeros((max_words, embedding_dim))

for word, i in word_index.items():
```

```
for word, i in word_index.items():
    embedding_vector = embeddings_index([word])
    if i < max_words:
        if embedding_vector is not None:
            # 임베딩 인덱스에 없는 단어는 모두 0이 됨
            embedding_matrix[i] = embedding_vector
```

- 확인

```
embedding_matrix.shape
```

```
(10000, 250)
```

IV. Keras Embedding Modeling

1) Model Define

- 모델 신경망 구조 정의
  - Embedding Dimension : 250

```
from keras import models
from keras import layers

imdb = models.Sequential()

imdb.add(layers.Embedding(max_words,
                          embedding_dim,
                          input_length = maxlen))

imdb.add(layers.LSTM(16))
imdb.add(layers.Dropout(0.5))
imdb.add(layers.Dense(1, activation = 'sigmoid'))
```

```
imdb.layers[0].set_weights([embedding_matrix])
imdb.layers[0].trainable = False
```

- 모델 구조 확인

```
imdb.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 2000, 250)         2500000
```

```
_________________________________________________________________
lstm (LSTM)                  (None, 16)                17088
_________________________________________________________________
dropout (Dropout)            (None, 16)                0
_________________________________________________________________
dense (Dense)                (None, 1)                 17
=================================================================
Total params: 2,517,105
Trainable params: 17,105
Non-trainable params: 2,500,000
_________________________________________________________________
```

2) Model Compile

- 모델 학습방법 설정

```
imdb.compile(loss = 'binary_crossentropy',
             optimizer = 'adam',
             metrics = ['accuracy'])
```

3) Model Fit

- 약 25분

```
%%time

Hist_imdb = imdb.fit(X_train, y_train,
                     epochs = 100,
                     batch_size = 512,
                     validation_data = (X_valid, y_valid))
```

```
Epoch 1/100
30/30 [==============================] - 46s 458ms/step - loss: 0.6900 - accuracy: 0.5329
Epoch 2/100
30/30 [==============================] - 13s 428ms/step - loss: 0.6761 - accuracy: 0.5988
Epoch 3/100
30/30 [==============================] - 13s 430ms/step - loss: 0.6192 - accuracy: 0.6785
Epoch 4/100
30/30 [==============================] - 13s 430ms/step - loss: 0.5849 - accuracy: 0.7134
Epoch 5/100
30/30 [==============================] - 13s 429ms/step - loss: 0.5638 - accuracy: 0.7251
Epoch 6/100
30/30 [==============================] - 13s 430ms/step - loss: 0.5228 - accuracy: 0.7574
Epoch 7/100
30/30 [==============================] - 13s 430ms/step - loss: 0.5018 - accuracy: 0.7676
Epoch 8/100
30/30 [==============================] - 13s 429ms/step - loss: 0.5008 - accuracy: 0.7676
Epoch 9/100
30/30 [==============================] - 13s 430ms/step - loss: 0.4818 - accuracy: 0.7826
Epoch 10/100
```

```
30/30 [==============================] - 13s 430ms/step - loss: 0.5094 - accuracy: 0.7571
Epoch 11/100
30/30 [==============================] - 13s 431ms/step - loss: 0.4938 - accuracy: 0.7785
Epoch 12/100
30/30 [==============================] - 13s 431ms/step - loss: 0.4713 - accuracy: 0.7851
Epoch 13/100
30/30 [==============================] - 13s 431ms/step - loss: 0.4673 - accuracy: 0.7931
Epoch 14/100
30/30 [==============================] - 13s 431ms/step - loss: 0.4664 - accuracy: 0.7919
Epoch 15/100
30/30 [==============================] - 13s 434ms/step - loss: 0.4507 - accuracy: 0.8012
Epoch 16/100
30/30 [==============================] - 13s 432ms/step - loss: 0.4560 - accuracy: 0.7988
Epoch 17/100
30/30 [==============================] - 13s 433ms/step - loss: 0.4550 - accuracy: 0.7970
Epoch 18/100
30/30 [==============================] - 13s 431ms/step - loss: 0.4479 - accuracy: 0.8013
Epoch 19/100
30/30 [==============================] - 13s 430ms/step - loss: 0.4367 - accuracy: 0.8071
Epoch 20/100
30/30 [==============================] - 13s 431ms/step - loss: 0.4329 - accuracy: 0.8119
Epoch 21/100
30/30 [==============================] - 13s 432ms/step - loss: 0.4331 - accuracy: 0.8051
Epoch 22/100
30/30 [==============================] - 13s 431ms/step - loss: 0.4266 - accuracy: 0.8141
Epoch 23/100
30/30 [==============================] - 13s 431ms/step - loss: 0.4400 - accuracy: 0.8041
Epoch 24/100
30/30 [==============================] - 13s 429ms/step - loss: 0.4255 - accuracy: 0.8135
Epoch 25/100
30/30 [==============================] - 13s 431ms/step - loss: 0.4170 - accuracy: 0.8177
Epoch 26/100
30/30 [==============================] - 13s 431ms/step - loss: 0.4183 - accuracy: 0.8205
Epoch 27/100
30/30 [==============================] - 13s 431ms/step - loss: 0.4284 - accuracy: 0.8102
Epoch 28/100
30/30 [==============================] - 13s 430ms/step - loss: 0.4394 - accuracy: 0.8024
Epoch 29/100
30/30 [==============================] - 13s 431ms/step - loss: 0.4094 - accuracy: 0.8159
```

4) 학습 결과 시각화

- Loss Visualization

```
import matplotlib.pyplot as plt

epochs = range(1, len(Hist_imdb.history['loss']) + 1)

plt.figure(figsize = (9, 6))
plt.plot(epochs, Hist_imdb.history['loss'])
plt.plot(epochs, Hist_imdb.history['val_loss'])
plt.title('Training & Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(['Training Loss', 'Validation Loss'])
plt.show()
```

```
plt.grid()
plt.show()
```

- Accuracy Visualization

```
import matplotlib.pyplot as plt

epochs = range(1, len(Hist_imdb.history['accuracy']) + 1)

plt.figure(figsize = (9, 6))
plt.plot(epochs, Hist_imdb.history['accuracy'])
plt.plot(epochs, Hist_imdb.history['val_accuracy'])
plt.title('Training & Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(['Training Accuracy', 'Validation Accuracy'])
plt.grid()
plt.show()
```

5) Model Evaluate

- Loss & Accuracy

```
loss, accuracy = imdb.evaluate(X_valid, y_valid)

print('Loss = {:.5f}'.format(loss))
print('Accuracy = {:.5f}'.format(accuracy))
```

```
313/313 [==============================] - 11s 36ms/step - loss: 0.4214 - accuracy: 0.8216
Loss = 0.42139
Accuracy = 0.82160
```

#

#

#

The End

#

#

#