05_sentiment_analysis_pretrained_embeddings

September 29, 2021

1 Sentiment analysis with pretrained word vectors

In Chapter 15, Word Embeddings, we discussed how to learn domain-specific word embeddings. Word2vec, and related learning algorithms, produce high-quality word vectors, but require large datasets. Hence, it is common that research groups share word vectors trained on large datasets, similar to the weights for pretrained deep learning models that we encountered in the section on transfer learning in the previous chapter.

We are now going to illustrate how to use pretrained Global Vectors for Word Representation (GloVe) provided by the Stanford NLP group with the IMDB review dataset.

[1]:
```
%matplotlib inline
from pathlib import Path
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime, date
from sklearn.metrics import mean_squared_error, roc_auc_score
from sklearn.preprocessing import minmax_scale
from keras.callbacks import ModelCheckpoint, EarlyStopping
from keras.datasets import imdb
from keras.models import Sequential, Model
from keras.layers import Dense, LSTM, GRU, Input, concatenate, Embedding,␣
 ↪Reshape
from keras.preprocessing.sequence import pad_sequences
from keras.preprocessing.text import Tokenizer
import keras
import keras.backend as K
import tensorflow as tf
```

```
Using TensorFlow backend.
```

[2]:
```
sns.set_style('whitegrid')
np.random.seed(42)
K.clear_session()
```

1.1 Load Reviews

We are going to load the IMDB dataset from the source for manual preprocessing.

Data source: Stanford IMDB Reviews Dataset

[3]:
```
path = Path('data/aclImdb/')
```

[4]:
```
files = path.glob('**/*.txt')
len(list(files))
```

[4]:
```
50000
```

[5]:
```
files = path.glob('*/**/*.txt')
data = []
for f in files:
    _, _, data_set, outcome = str(f.parent).split('/')
    data.append([data_set, int(outcome=='pos'), f.read_text(encoding='latin1')])
```

[6]:
```
data = pd.DataFrame(data, columns=['dataset', 'label', 'review']).sample(frac=1.
 ↪0)
```

[7]:
```
train_data = data.loc[data.dataset=='train', ['label', 'review']]
test_data = data.loc[data.dataset=='test', ['label', 'review']]
```

[8]:
```
train_data.label.value_counts()
```

[8]:
```
1    12500
0    12500
Name: label, dtype: int64
```

[9]:
```
test_data.label.value_counts()
```

[9]:
```
1    12500
0    12500
Name: label, dtype: int64
```

1.2 Prepare Data

1.2.1 Tokenizer

Keras provides a tokenizer that we use to convert the text documents to integer-encoded sequences, as shown here:

[10]:
```
num_words = 10000
t = Tokenizer(num_words=num_words,
              lower=True,
              oov_token=2)
t.fit_on_texts(train_data.review)
```

[11]:
```
vocab_size = len(t.word_index) + 1
vocab_size
```

[11]: 88586

[12]:
```
train_data_encoded = t.texts_to_sequences(train_data.review)
test_data_encoded = t.texts_to_sequences(test_data.review)
```

[13]:
```
max_length = 100
```

1.2.2 Pad Sequences

We also use the pad_sequences function to convert the list of lists (of unequal length) to stacked sets of padded and truncated arrays for both the train and test datasets:

[14]:
```
X_train_padded = pad_sequences(train_data_encoded,
                            maxlen=max_length,
                            padding='post',
                           truncating='post')
y_train = train_data['label']
X_train_padded.shape
```

[14]: (25000, 100)

[15]:
```
X_test_padded = pad_sequences(test_data_encoded,
                            maxlen=max_length,
                            padding='post',
                           truncating='post')
y_test = test_data['label']
X_test_padded.shape
```

[15]: (25000, 100)

1.3 Load Embeddings

Assuming we have downloaded and unzipped the GloVe data to the location indicated in the code, we now create a dictionary that maps GloVe tokens to 100-dimensional real-valued vectors, as follows:

[16]:
```
# load the whole embedding into memory
glove_path = Path('data/glove/glove.6B.100d.txt')
embeddings_index = dict()

for line in glove_path.open(encoding='latin1'):
    values = line.split()
    word = values[0]
    try:
        coefs = np.asarray(values[1:], dtype='float32')
```

```
    except:
        continue
    embeddings_index[word] = coefs
```

[17]:
```
print('Loaded {:,d} word vectors.'.format(len(embeddings_index)))
```

```
Loaded 399,883 word vectors.
```

There are around 340,000 word vectors that we use to create an embedding matrix that matches the vocabulary so that the RNN model can access embeddings by the token index:

[18]:
```
embedding_matrix = np.zeros((vocab_size, 100))
for word, i in t.word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
```

[19]:
```
embedding_matrix.shape
```

[19]:
```
(88586, 100)
```

1.4 Define Model Architecture

The difference between this and the RNN setup in the previous example is that we are going to pass the embedding matrix to the embedding layer and set it to non-trainable, so that the weights remain fixed during training:

[20]:
```
embedding_size = 100
```

[21]:
```
rnn = Sequential([
    Embedding(input_dim=vocab_size,
              output_dim= embedding_size,
              input_length=max_length,
              weights=[embedding_matrix],
              trainable=False),
    GRU(units=32,  dropout=0.2, recurrent_dropout=0.2),
    Dense(1, activation='sigmoid')
])
rnn.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 100, 100)          8858600
_________________________________________________________________
gru_1 (GRU)                  (None, 32)                12768
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 33
```

```
=================================================================
Total params: 8,871,401
Trainable params: 12,801
Non-trainable params: 8,858,600
_________________________________________________________________
```

[23]:
```python
rnn.compile(loss='binary_crossentropy',
            optimizer='RMSProp',
            metrics=['accuracy'])
```

[24]:
```python
rnn_path = 'models/imdb.gru_pretrained.weights.best.hdf5'
checkpointer = ModelCheckpoint(filepath=rnn_path,
                               monitor='val_loss',
                               save_best_only=True,
                               save_weights_only=True,
                               period=5)
```

[25]:
```python
early_stopping = EarlyStopping(monitor='val_loss',
                               patience=5,
                               restore_best_weights=True)
```

[26]:
```python
rnn.fit(X_train_padded,
        y_train,
        batch_size=32,
        epochs=25,
        validation_data=(X_test_padded, y_test),
        callbacks=[checkpointer, early_stopping],
        verbose=1)
```

```
Train on 25000 samples, validate on 25000 samples
Epoch 1/25
25000/25000 [==============================] - 63s 3ms/step - loss: 0.6392 -
acc: 0.6218 - val_loss: 0.5223 - val_acc: 0.7461
Epoch 2/25
25000/25000 [==============================] - 62s 2ms/step - loss: 0.5263 -
acc: 0.7458 - val_loss: 0.4666 - val_acc: 0.7781
Epoch 3/25
25000/25000 [==============================] - 63s 3ms/step - loss: 0.4834 -
acc: 0.7722 - val_loss: 0.4407 - val_acc: 0.7918
Epoch 4/25
25000/25000 [==============================] - 62s 2ms/step - loss: 0.4550 -
acc: 0.7869 - val_loss: 0.4264 - val_acc: 0.7986
Epoch 5/25
25000/25000 [==============================] - 63s 3ms/step - loss: 0.4395 -
acc: 0.7958 - val_loss: 0.4131 - val_acc: 0.8066
Epoch 6/25
25000/25000 [==============================] - 63s 3ms/step - loss: 0.4247 -
acc: 0.8037 - val_loss: 0.4040 - val_acc: 0.8096
```

```
Epoch 7/25
25000/25000 [==============================] - 63s 3ms/step - loss: 0.4113 -
acc: 0.8117 - val_loss: 0.4022 - val_acc: 0.8100
Epoch 8/25
25000/25000 [==============================] - 63s 3ms/step - loss: 0.4043 -
acc: 0.8129 - val_loss: 0.3950 - val_acc: 0.8144
Epoch 9/25
25000/25000 [==============================] - 63s 3ms/step - loss: 0.3985 -
acc: 0.8156 - val_loss: 0.3990 - val_acc: 0.8120
Epoch 10/25
25000/25000 [==============================] - 63s 3ms/step - loss: 0.3917 -
acc: 0.8216 - val_loss: 0.3938 - val_acc: 0.8176
Epoch 11/25
25000/25000 [==============================] - 67s 3ms/step - loss: 0.3888 -
acc: 0.8233 - val_loss: 0.3930 - val_acc: 0.8179
Epoch 12/25
25000/25000 [==============================] - 66s 3ms/step - loss: 0.3817 -
acc: 0.8265 - val_loss: 0.3871 - val_acc: 0.8192
Epoch 13/25
25000/25000 [==============================] - 66s 3ms/step - loss: 0.3794 -
acc: 0.8296 - val_loss: 0.4039 - val_acc: 0.8134
Epoch 14/25
25000/25000 [==============================] - 63s 3ms/step - loss: 0.3753 -
acc: 0.8311 - val_loss: 0.3843 - val_acc: 0.8237
Epoch 15/25
25000/25000 [==============================] - 63s 3ms/step - loss: 0.3682 -
acc: 0.8328 - val_loss: 0.3906 - val_acc: 0.8196
Epoch 16/25
25000/25000 [==============================] - 67s 3ms/step - loss: 0.3675 -
acc: 0.8374 - val_loss: 0.3822 - val_acc: 0.8262
Epoch 17/25
25000/25000 [==============================] - 65s 3ms/step - loss: 0.3641 -
acc: 0.8373 - val_loss: 0.3799 - val_acc: 0.8253
Epoch 18/25
25000/25000 [==============================] - 64s 3ms/step - loss: 0.3630 -
acc: 0.8382 - val_loss: 0.3875 - val_acc: 0.8205
Epoch 19/25
25000/25000 [==============================] - 63s 3ms/step - loss: 0.3586 -
acc: 0.8417 - val_loss: 0.3820 - val_acc: 0.8267
Epoch 20/25
25000/25000 [==============================] - 63s 3ms/step - loss: 0.3600 -
acc: 0.8387 - val_loss: 0.3862 - val_acc: 0.8262
Epoch 21/25
25000/25000 [==============================] - 63s 3ms/step - loss: 0.3560 -
acc: 0.8395 - val_loss: 0.3847 - val_acc: 0.8249
Epoch 22/25
25000/25000 [==============================] - 64s 3ms/step - loss: 0.3542 -
acc: 0.8408 - val_loss: 0.3815 - val_acc: 0.8264
```

[26]: <keras.callbacks.History at 0x7f16815803c8>

[29]:
```
rnn.load_weights(rnn_path)
```

[30]:
```
y_score = rnn.predict(X_test_padded)
roc_auc_score(y_score=y_score.squeeze(), y_true=y_test)
```

[30]: 0.910672304

[ ]:
```
```