

04_sentiment_analysis

September 29, 2021

1 LSTM & Word Embeddings for Sentiment Classification

RNNs are commonly applied to various natural language processing tasks. We've already encountered sentiment analysis using text data in part three of [this book](#).

We are now going to illustrate how to apply an RNN model to text data to detect positive or negative sentiment (which can easily be extended to a finer-grained sentiment scale). We are going to use word embeddings to represent the tokens in the documents. We covered word embeddings in Chapter 15, Word Embeddings. They are an excellent technique to convert text into a continuous vector representation such that the relative location of words in the latent space encodes useful semantic aspects based on the words' usage in context.

We saw in the previous RNN example that Keras has a built-in embedding layer that allows us to train vector representations specific to the task at hand. Alternatively, we can use pretrained vectors.

1.1 Imports & Settings

```
[1]: %matplotlib inline
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
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()
```

1.2 Load Reviews

To keep the data manageable, we will illustrate this use case with the IMDB reviews dataset, which contains 50,000 positive and negative movie reviews evenly split into a train and a test set, and with balanced labels in each dataset. The vocabulary consists of 88,586 tokens.

The dataset is bundled into Keras and can be loaded so that each review is represented as an integer-encoded sequence. We can limit the vocabulary to `num_words` while filtering out frequent and likely less informative words using `skip_top`, as well as sentences longer than `maxlen`. We can also choose `oov_char`, which represents tokens we chose to exclude from the vocabulary on frequency grounds, as follows:

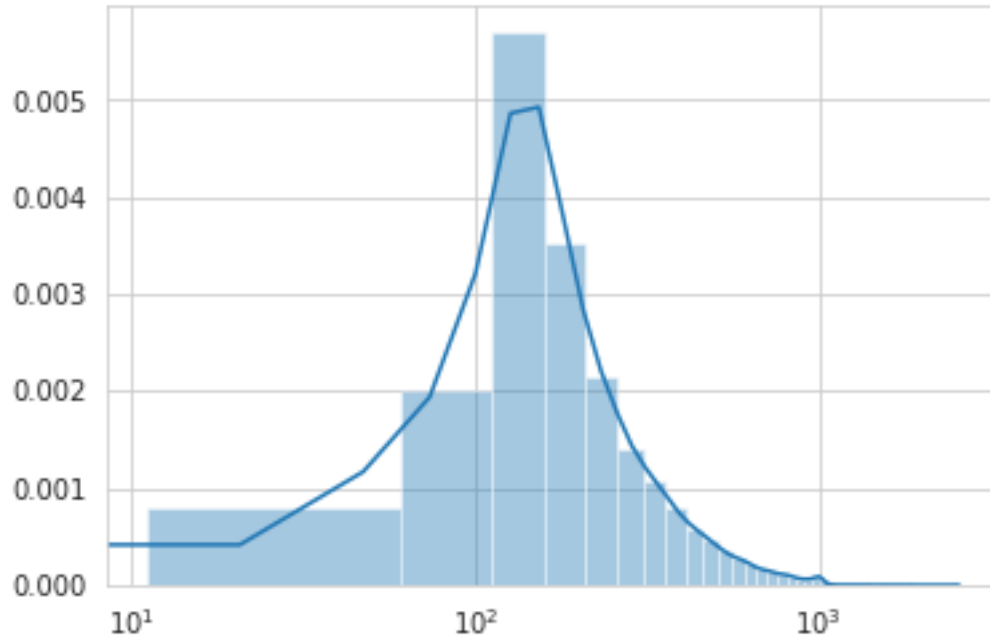
```
[3]: vocab_size = 20000
```

```
[4]: (X_train, y_train), (X_test, y_test) = imdb.load_data(seed=42,
                                                         skip_top=0,
                                                         maxlen=None,
                                                         oov_char=2,
                                                         index_from=3
                                                         num_words=vocab_size)
```

```
[5]: ax = sns.distplot([len(review) for review in X_train])
      ax.set(xscale='log');
```

```
/usr/local/lib/python3.5/dist-packages/scipy/stats/stats.py:1713: FutureWarning:
Using a non-tuple sequence for multidimensional indexing is deprecated; use
`arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpreted
as an array index, `arr[np.array(seq)]`, which will result either in an error or
a different result.
```

```
    return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```



1.3 Prepare Data

In the second step, convert the lists of integers into fixed-size arrays that we can stack and provide as input to our RNN. The `pad_sequence` function produces arrays of equal length, truncated, and padded to conform to `maxlen`, as follows:

```
[6]: maxlen = 100
```

```
[7]: X_train_padded = pad_sequences(X_train,
                                     truncating='pre',
                                     padding='pre',
                                     maxlen=maxlen)

X_test_padded = pad_sequences(X_test,
                              truncating='pre',
                              padding='pre',
                              maxlen=maxlen)
```

```
[8]: X_train_padded.shape, X_test_padded.shape
```

```
[8]: ((25000, 100), (25000, 100))
```

1.4 Define Model Architecture

Now we can define our RNN architecture. The first layer learns the word embeddings. We define the embedding dimension as previously using the `input_dim` keyword to set the number of tokens

that we need to embed, the `output_dim` keyword, which defines the size of each embedding, and how long each input sequence is going to be.

1.4.1 Custom Loss Metric

```
[9]: # source: https://github.com/keras-team/keras/issues/3230
def auc(y_true, y_pred):
    ptas = tf.stack([binary_PTA(y_true, y_pred, k) for k in np.linspace(0, 1,
    ↪1000)], axis=0)
    pfas = tf.stack([binary_PFA(y_true, y_pred, k) for k in np.linspace(0, 1,
    ↪1000)], axis=0)
    pfas = tf.concat([tf.ones((1,)), pfas], axis=0)
    binSizes = -(pfas[1:] - pfas[:-1])
    s = ptas * binSizes
    return K.sum(s, axis=0)

def binary_PFA(y_true, y_pred, threshold=K.variable(value=0.5)):
    """prob false alert for binary classifier"""
    y_pred = K.cast(y_pred >= threshold, 'float32')
    # N = total number of negative labels
    N = K.sum(1 - y_true)
    # FP = total number of false alerts, alerts from the negative class labels
    FP = K.sum(y_pred - y_pred * y_true)
    return FP / (N + 1)

def binary_PTA(y_true, y_pred, threshold=K.variable(value=0.5)):
    """prob true alerts for binary classifier"""
    y_pred = K.cast(y_pred >= threshold, 'float32')
    # P = total number of positive labels
    P = K.sum(y_true)
    # TP = total number of correct alerts, alerts from the positive class labels
    TP = K.sum(y_pred * y_true)
    return TP / (P + 1)
```

```
[10]: embedding_size = 100
```

Note that we are using GRUs this time, which train faster and perform better on smaller data. We are also using dropout for regularization, as follows:

```
[11]: rnn = Sequential([
    Embedding(input_dim=vocab_size, output_dim= embedding_size,
    ↪input_length=maxlen),
    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)         2000000
-----
gru_1 (GRU)                  (None, 32)               12768
-----
dense_1 (Dense)              (None, 1)                33
=====
Total params: 2,012,801
Trainable params: 2,012,801
Non-trainable params: 0
-----
```

The resulting model has over 2 million parameters.

We compile the model to use our custom AUC metric, which we introduced previously, and train with early stopping:

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

```
[17]: rnn_path = 'models/imdb.gru.weights.best.hdf5'
checkpointer = ModelCheckpoint(filepath=rnn_path,
                               monitor='val_auc',
                               save_best_only=True,
                               save_weights_only=True,
                               period=5)
```

```
[18]: early_stopping = EarlyStopping(monitor='val_auc',
                                     mode='max',
                                     patience=5,
                                     restore_best_weights=True)
```

Training stops after eight epochs and we recover the weights for the best models to find a high test AUC of 0.9346:

```
[19]: 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 [=====] - 125s 5ms/step - loss: 0.3562 -
acc: 0.8514 - auc: 0.8717 - val_loss: 0.4181 - val_acc: 0.8311 - val_auc: 0.8704
Epoch 2/25
25000/25000 [=====] - 112s 4ms/step - loss: 0.2964 -
acc: 0.8833 - auc: 0.8924 - val_loss: 0.3363 - val_acc: 0.8575 - val_auc: 0.8817
Epoch 3/25
25000/25000 [=====] - 112s 4ms/step - loss: 0.2551 -
acc: 0.9006 - auc: 0.9054 - val_loss: 0.3450 - val_acc: 0.8585 - val_auc: 0.8823
Epoch 4/25
25000/25000 [=====] - 113s 5ms/step - loss: 0.2231 -
acc: 0.9145 - auc: 0.9129 - val_loss: 0.3393 - val_acc: 0.8583 - val_auc: 0.8822
Epoch 5/25
25000/25000 [=====] - 112s 4ms/step - loss: 0.1997 -
acc: 0.9258 - auc: 0.9179 - val_loss: 0.3517 - val_acc: 0.8558 - val_auc: 0.8815
Epoch 6/25
25000/25000 [=====] - 117s 5ms/step - loss: 0.1780 -
acc: 0.9335 - auc: 0.9221 - val_loss: 0.3980 - val_acc: 0.8443 - val_auc: 0.8787
Epoch 7/25
25000/25000 [=====] - 122s 5ms/step - loss: 0.1571 -
acc: 0.9436 - auc: 0.9251 - val_loss: 0.3949 - val_acc: 0.8511 - val_auc: 0.8782
Epoch 8/25
25000/25000 [=====] - 112s 4ms/step - loss: 0.1422 -
acc: 0.9484 - auc: 0.9279 - val_loss: 0.3815 - val_acc: 0.8507 - val_auc: 0.8780

```

```
[19]: <keras.callbacks.History at 0x7f01c4c8a400>
```

1.5 Evaluate Results

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

```
[22]: y_score = rnn.predict(X_test_padded)
      y_score.shape
```

```
[22]: (25000, 1)
```

```
[23]: roc_auc_score(y_score=y_score.squeeze(), y_true=y_test)
```

```
[23]: 0.9346154079999999
```

```
[ ]:
```