07_sec_filings_return_prediction

September 29, 2021

1 RNN & Word Embeddings for SEC Filings to Predict Returns

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 apply an RNN model to SEC filings to learn custom word embeddings (see Chapter 16) and predict the returns over the week after the filing date.

1.1 Imports & Settings

[1]:
```
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
%matplotlib inline

from pathlib import Path
from time import time
from collections import Counter
from datetime import datetime, timedelta
from tqdm import tqdm

import numpy as np
import pandas as pd
from scipy.stats import spearmanr
import yfinance as yf

from gensim.models.word2vec import LineSentence
from gensim.models.phrases import Phrases, Phraser

from sklearn.model_selection import train_test_split

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (Dense, GRU, Bidirectional,
                                     Embedding, BatchNormalization, Dropout)
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.metrics import RootMeanSquaredError, MeanAbsoluteError
```

```python
import tensorflow.keras.backend as K

import matplotlib.pyplot as plt
import seaborn as sns
```

[3]:
```python
gpu_devices = tf.config.experimental.list_physical_devices('GPU')
if gpu_devices:
    print('Using GPU')
    tf.config.experimental.set_memory_growth(gpu_devices[0], True)
else:
    print('Using CPU')
```

```
Using CPU
```

[4]:
```python
np.random.seed(42)
tf.random.set_seed(42)
```

[5]:
```python
idx = pd.IndexSlice
sns.set_style('whitegrid')
```

[6]:
```python
def format_time(t):
    m, s = divmod(t, 60)
    h, m = divmod(m, 60)
    return f'{h:02.0f}:{m:02.0f}:{s:02.0f}'
```

[7]:
```python
deciles = np.arange(.1, 1, .1).round(1)
```

1.2 Get stock price data

1.2.1 Paths

[8]:
```python
data_path = Path('..', 'data', 'sec-filings')
```

[9]:
```python
results_path = Path('results', 'sec-filings')

selected_section_path = results_path / 'ngrams_1'
ngram_path = results_path / 'ngrams'
vector_path = results_path / 'vectors'

for path in [vector_path, selected_section_path, ngram_path]:
    if not path.exists():
        path.mkdir(parents=True)
```

1.2.2 Get filing info

[10]:
```
filing_index = (pd.read_csv(data_path / 'filing_index.csv',
                            parse_dates=['DATE_FILED'])
                .rename(columns=str.lower))
filing_index.index += 1
```

[11]:
```
filing_index.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 22631 entries, 1 to 22631
Data columns (total 11 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   cik           22631 non-null  int64
 1   company_name  22631 non-null  object
 2   form_type     22631 non-null  object
 3   date_filed    22631 non-null  datetime64[ns]
 4   edgar_link    22631 non-null  object
 5   quarter       22631 non-null  int64
 6   ticker        22631 non-null  object
 7   sic           22461 non-null  object
 8   exchange      20619 non-null  object
 9   hits          22555 non-null  object
 10  year          22631 non-null  int64
dtypes: datetime64[ns](1), int64(3), object(7)
memory usage: 1.9+ MB
```

[12]:
```
filing_index.head()
```

[12]:
```
       cik                company_name form_type date_filed  \
1  1000180                 SANDISK CORP      10-K 2013-02-19
2  1000209      MEDALLION FINANCIAL CORP      10-K 2013-03-13
3  1000228              HENRY SCHEIN INC      10-K 2013-02-13
4  1000229         CORE LABORATORIES N V      10-K 2013-02-19
5  1000232  KENTUCKY BANCSHARES INC  KY      10-K 2013-03-28

                                    edgar_link  quarter ticker   sic exchange  \
1  edgar/data/1000180/0001000180-13-000009.txt        1   SNDK  3572   NASDAQ
2  edgar/data/1000209/0001193125-13-103504.txt        1   TAXI  6199   NASDAQ
3  edgar/data/1000228/0001000228-13-000010.txt        1   HSIC  5047   NASDAQ
4  edgar/data/1000229/0001000229-13-000009.txt        1    CLB  1389     NYSE
5  edgar/data/1000232/0001104659-13-025094.txt        1   KTYB  6022      OTC

  hits  year
1    3  2013
2    0  2013
3    3  2013
```

```
4    2  2013
5    0  2013
```

[13]:
```
filing_index.ticker.nunique()
```

[13]:
```
6630
```

[14]:
```
filing_index.date_filed.describe()
```

[14]:
```
count                     22631
unique                      980
top       2014-03-31 00:00:00
freq                        442
first     2013-01-02 00:00:00
last      2016-12-30 00:00:00
Name: date_filed, dtype: object
```

1.2.3 Download stock price data using Yfinance

`yfinance` can be unstable so that connections drop; if you experience this you may want to store intermediate results so you don't have to start over.

[ ]:
```
yf_data, missing = [], []
for i, (symbol, dates) in enumerate(filing_index.groupby('ticker').date_filed,␣
 ↪1):

    if i % 250 == 0:
        print(i, len(yf_data), len(set(missing)), flush=True)

    ticker = yf.Ticker(symbol)
    for filing, date in dates.to_dict().items():
        start = date - timedelta(days=93)
        end = date + timedelta(days=31)
        df = ticker.history(start=start, end=end)
        if df.empty:
            missing.append(symbol)
        else:
            yf_data.append(df.assign(ticker=symbol, filing=filing))
```

[ ]:
```
yf_data = pd.concat(yf_data).rename(columns=str.lower)
```

[ ]:
```
yf_data.to_hdf(results_path / 'sec_returns.h5', 'data/yfinance')
```

[ ]:
```
yf_data = pd.read_hdf(results_path / 'sec_returns.h5', 'data/yfinance')
```

[ ]:
```
yf_data.ticker.nunique()
```

[ ]:
```
yf_data.info()
```

1.2.4 Get (some) missing prices from Quandl

[ ]:
```
to_do = (filing_index.loc[~filing_index.ticker.isin(yf_data.ticker.unique()),
                          ['ticker', 'date_filed']])
```

[ ]:
```
to_do.date_filed.min()
```

[ ]:
```
quandl_tickers = (pd.read_hdf('../data/assets.h5', 'quandl/wiki/prices')
                  .loc[idx['2012':, :], :]
                  .index.unique('ticker'))
quandl_tickers = list(set(quandl_tickers).intersection(set(to_do.ticker)))
```

[ ]:
```
len(quandl_tickers)
```

[ ]:
```
to_do = filing_index.loc[filing_index.ticker.isin(quandl_tickers), ['ticker',␣
↪'date_filed']]
```

[ ]:
```
to_do.info()
```

[ ]:
```
ohlcv = ['adj_open', 'adj_high', 'adj_low', 'adj_close', 'adj_volume']
```

[ ]:
```
quandl = (pd.read_hdf('../data/assets.h5', 'quandl/wiki/prices')
          .loc[idx['2012': , quandl_tickers], ohlcv]
          .rename(columns=lambda x: x.replace('adj_', '')))
```

[ ]:
```
quandl.info()
```

[ ]:
```
quandl_data = []
for i, (symbol, dates) in enumerate(to_do.groupby('ticker').date_filed, 1):
    if i % 100 == 0:
        print(i, end=' ', flush=True)
    for filing, date in dates.to_dict().items():
        start = date - timedelta(days=93)
        end = date + timedelta(days=31)
        quandl_data.append(quandl.loc[idx[start:end, symbol], :].
↪reset_index('ticker').assign(filing=filing))
quandl_data = pd.concat(quandl_data)
```

[ ]:
```
quandl_data.to_hdf(results_path / 'sec_returns.h5', 'data/quandl')
```

1.2.5 Combine, clean and persist

[ ]:
```
data = (pd.read_hdf(results_path / 'sec_returns.h5', 'data/yfinance')
        .drop(['dividends', 'stock splits'], axis=1)
        .append(pd.read_hdf(results_path / 'sec_returns.h5',
                            'data/quandl')))
```

[ ]:
```
data = data.loc[:, ['filing', 'ticker', 'open', 'high', 'low', 'close',␣
↪'volume']]
```

[ ]:
```
data.info()
```

[ ]:
```
data[['filing', 'ticker']].nunique()
```

[ ]:
```
data.to_hdf(results_path / 'sec_returns.h5', 'prices')
```

1.3 Copy filings with stock price data

[16]:
```
data = pd.read_hdf(results_path / 'sec_returns.h5', 'prices')
```

[17]:
```
filings_with_data = data.filing.unique()
len(filings_with_data)
```

[17]:
```
16758
```

1.3.1 Remove short and long sentences

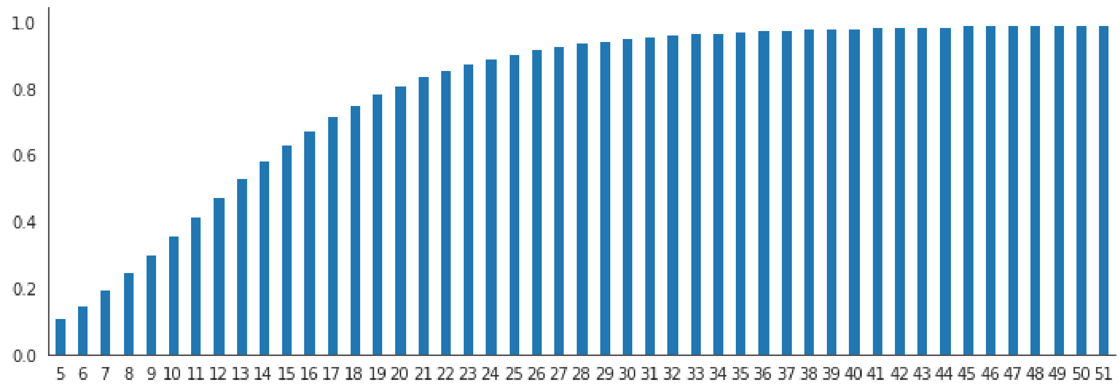
[18]:
```
min_sentence_length = 5
max_sentence_length = 50
```

[19]:
```
sent_length = Counter()
for i, idx in enumerate(filings_with_data, 1):
    if i % 500 == 0:
        print(i, end=' ', flush=True)
    text = pd.read_csv(data_path / 'selected_sections' / f'{idx}.csv').text
    sent_length.update(text.str.split().str.len().tolist())
    text = text[text.str.split().str.len().between(min_sentence_length,␣
↪max_sentence_length)]
    text = '\n'.join(text.tolist())
    with (selected_section_path / f'{idx}.txt').open('w') as f:
        f.write(text)
```

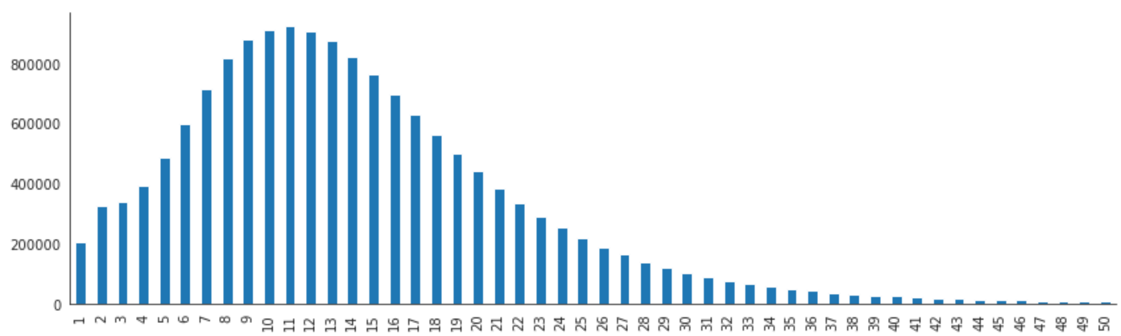
```
500 1000 1500 2000 2500 3000 3500 4000 4500 5000 5500 6000 6500 7000 7500 8000
8500 9000 9500 10000 10500 11000 11500 12000 12500 13000 13500 14000 14500 15000
15500 16000 16500
```

[20]:
```
sent_length = pd.Series(dict(sent_length.most_common()))
```

```
[21]: with sns.axes_style("white"):
          sent_length.sort_index().cumsum().div(sent_length.sum()).loc[5:51].plot.
       ↪bar(figsize=(12, 4), rot=0)
          sns.despine();
```




```
[22]: with sns.axes_style("white"):
          sent_length.sort_index().loc[:50].plot.bar(figsize=(14, 4))
          sns.despine();
```



1.3.2 Create bi- and trigrams

Combine all filings

```
[23]: files = selected_section_path.glob('*.txt')
      texts = [f.read_text() for f in files]
      unigrams = ngram_path / 'ngrams_1.txt'
      unigrams.write_text('\n'.join(texts))
```

```
[23]: 1827326308
```

```
[24]: texts = unigrams.read_text()
```

This takes quite some time; last attempt was 30 min per iteration.

[25]:
```python
n_grams = []
start = time()
for i, n in enumerate([2, 3]):
    sentences = LineSentence(ngram_path / f'ngrams_{n-1}.txt')
    phrases = Phrases(sentences=sentences,
                      min_count=25,  # ignore terms with a lower count
                      threshold=0.5,  # accept phrases with higher score
                      max_vocab_size=4000000,  # prune of less common words to␣
↪limit memory use
                      delimiter=b'_',  # how to join ngram tokens
                      scoring='npmi')

    s = pd.DataFrame([[k.decode('utf-8'), v] for k, v in phrases.
↪export_phrases(sentences)],
                     columns=['phrase', 'score']).assign(length=n)

    n_grams.append(s.groupby('phrase').score.agg(['mean', 'size']))
    print(n_grams[-1].nlargest(5, columns='size'))

    grams = Phraser(phrases)
    sentences = grams[sentences]
    (ngram_path / f'ngrams_{n}.txt').write_text('\n'.join([' '.join(s) for s in␣
↪sentences]))

    src_dir = results_path / f'ngrams_{n-1}'
    target_dir = results_path / f'ngrams_{n}'
    if not target_dir.exists():
        target_dir.mkdir()

    for f in src_dir.glob('*.txt'):
        text = LineSentence(f)
        text = grams[text]
        (target_dir / f'{f.stem}.txt').write_text('\n'.join([' '.join(s) for s␣
↪in text]))
    print('\n\tDuration: ', format_time(time() - start))

n_grams = pd.concat(n_grams).sort_values('size', ascending=False)
n_grams.to_parquet(results_path / 'ngrams.parquet')
```

```
                         mean    size
phrase
year ended           0.824560  456420
results operations   0.727928  390446
table contents       0.946177  341318
company s            0.588563  312218
financial condition  0.768172  310234
```

```
        Duration:  00:29:02
                                                    mean    size
phrase
year_ended december                             0.803816  397878
financial_condition results_operations          0.781534  145569
material_adverse effect                         0.876534  130986
net income                                      0.506277  130149
interest income                                 0.558110  101746

        Duration:  00:56:36
```

[26]:
```
n_grams.groupby(n_grams.index.str.replace('_', ' ').str.count(' ')).size()
```

[26]:
```
phrase
1    28636
2     9970
3     2334
dtype: int64
```

1.3.3 Convert filings to integer sequences based on token count

[27]:
```
sentences = (ngram_path / 'ngrams_3.txt').read_text().split('\n')
```

[28]:
```
n = len(sentences)
```

[29]:
```
token_cnt = Counter()
for i, sentence in enumerate(sentences, 1):
    if i % 500000 == 0:
        print(f'{i/n:.1%}', end=' ', flush=True)
    token_cnt.update(sentence.split())
token_cnt = pd.Series(dict(token_cnt.most_common()))
token_cnt = token_cnt.reset_index()
token_cnt.columns = ['token', 'n']
```

```
3.5% 6.9% 10.4% 13.9% 17.4% 20.8% 24.3% 27.8% 31.3% 34.7% 38.2% 41.7% 45.2%
48.6% 52.1% 55.6% 59.1% 62.5% 66.0% 69.5% 73.0% 76.4% 79.9% 83.4% 86.8% 90.3%
93.8% 97.3%
```

[30]:
```
token_cnt.to_parquet(results_path / 'token_cnt')
```

[31]:
```
token_cnt.n.describe(deciles).apply(lambda x: f'{x:,.0f}')
```

[31]:
```
count      205,450
mean           919
std         13,442
min              1
```

```
10%             1
20%             2
30%             4
40%             6
50%            12
60%            25
70%            40
80%            82
90%           269
max     1,926,643
Name: n, dtype: object
```

[32]:
```python
token_cnt.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 205450 entries, 0 to 205449
Data columns (total 2 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   token   205450 non-null  object
 1   n       205450 non-null  int64
dtypes: int64(1), object(1)
memory usage: 3.1+ MB
```

[33]:
```python
token_cnt.nlargest(10, columns='n')
```

[33]:

	token	n
0	million	1926643
1	business	1210252
2	company	1061550
3	products	1035118
4	sales	864927
5	net	853937
6	including	794889
7	market	794453
8	costs	772244
9	increase	756554

[34]:
```python
token_cnt.sort_values(by=['n', 'token'], ascending=[False, True]).head()
```

[34]:

	token	n
0	million	1926643
1	business	1210252
2	company	1061550
3	products	1035118
4	sales	864927

```
[35]: token_by_freq = token_cnt.sort_values(by=['n', 'token'], ascending=[False,␣
      ↪True]).token
      token2id = {token: i for i, token in enumerate(token_by_freq, 3)}
```

```
[36]: len(token2id)
```

```
[36]: 205450
```

```
[37]: for token, i in token2id.items():
          print(token, i)
          break
```

```
million 3
```

```
[43]: def generate_sequences(min_len=100, max_len=20000, num_words=25000, oov_char=2):
          if not vector_path.exists():
              vector_path.mkdir()
          seq_length = {}
          skipped = 0
          for i, f in tqdm(enumerate((results_path / 'ngrams_3').glob('*.txt'), 1)):
              file_id = f.stem
              text = f.read_text().split('\n')
              vector = [token2id[token] if token2id[token] + 2 < num_words else␣
      ↪oov_char
                        for line in text
                        for token in line.split()]
              vector = vector[:max_len]
              if len(vector) < min_len:
                  skipped += 1
                  continue
              seq_length[int(file_id)] = len(vector)
              np.save(vector_path / f'{file_id}.npy', np.array(vector))
          seq_length = pd.Series(seq_length)
          return seq_length
```

```
[44]: seq_length = generate_sequences()
```

```
16758it [01:00, 279.15it/s]
```

```
[45]: pd.Series(seq_length).to_csv(results_path / 'seq_length.csv')
```

```
[46]: seq_length.describe(deciles)
```

```
[46]: count    16535.000000
      mean     10946.423163
      std       5217.386029
      min        121.000000
```

```
10%       4090.000000
20%       6159.000000
30%       7805.800000
40%       9229.000000
50%      10687.000000
60%      12124.000000
70%      13780.800000
80%      15909.400000
90%      19193.200000
max      20000.000000
dtype: float64
```

[47]:
```python
seq_length.sum()
```

[47]: 180999107

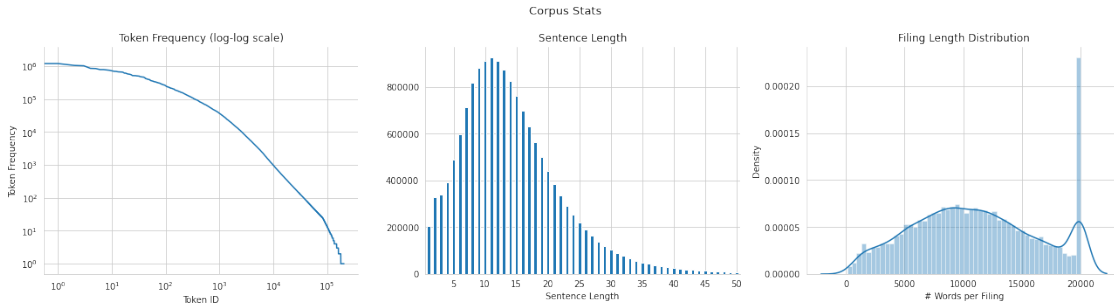
[48]:
```python
fig, axes = plt.subplots(ncols=3, figsize=(18,5))
token_cnt.n.plot(logy=True, logx=True, ax=axes[0], title='Token Frequency␣
 ↪(log-log scale)')
sent_length.sort_index().loc[:50].plot.bar(ax=axes[1], rot=0, title='Sentence␣
 ↪Length')

n=5
ticks = axes[1].xaxis.get_ticklocs()
ticklabels = [l.get_text() for l in axes[1].xaxis.get_ticklabels()]
axes[1].xaxis.set_ticks(ticks[n-1::n])
axes[1].xaxis.set_ticklabels(ticklabels[n-1::n])
axes[1].set_xlabel('Sentence Length')

sns.distplot(seq_length, ax=axes[2], bins=50)
axes[0].set_ylabel('Token Frequency')
axes[0].set_xlabel('Token ID')

axes[2].set_xlabel('# Words per Filing')
axes[2].set_title('Filing Length Distribution')

fig.suptitle('Corpus Stats', fontsize=13)
sns.despine()
fig.tight_layout()
fig.subplots_adjust(top=.85)
fig.savefig(results_path / 'sec_seq_len', dpi=300);
```




```
[49]: files = vector_path.glob('*.npy')
      filings = sorted([int(f.stem) for f in files])
```

1.4 Prepare Model Data

1.4.1 Create weekly forward returns

```
[50]: prices = pd.read_hdf(results_path / 'sec_returns.h5', 'prices')
      prices.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1405358 entries, 2013-09-17 to 2015-01-23
Data columns (total 7 columns):
 #   Column  Non-Null Count    Dtype
---  ------  --------------    -----
 0   filing  1405358 non-null  int64
 1   ticker  1405358 non-null  object
 2   open    1405304 non-null  float64
 3   high    1405322 non-null  float64
 4   low     1405322 non-null  float64
 5   close   1405323 non-null  float64
 6   volume  1405323 non-null  float64
dtypes: float64(5), int64(1), object(1)
memory usage: 85.8+ MB
```

```
[51]: fwd_return = {}
      for filing in filings:
          date_filed = filing_index.at[filing, 'date_filed']
          price_data = prices[prices.filing==filing].close.sort_index()

          try:
              r = (price_data
                   .pct_change(periods=5)
                   .shift(-5)
                   .loc[:date_filed]
                   .iloc[-1])
```

```
    except:
        continue
    if not np.isnan(r) and -.5 < r < 1:
        fwd_return[filing] = r
```

[52]:
```
len(fwd_return)
```

[52]: 16352

1.4.2 Combine returns with filing data

[53]:
```
y, X = [], []
for filing_id, fwd_ret in fwd_return.items():
    X.append(np.load(vector_path / f'{filing_id}.npy') + 2)
    y.append(fwd_ret)
y = np.array(y)
```

[54]:
```
len(y), len(X)
```

[54]: (16352, 16352)

[55]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.1)
```

1.4.3 Pad sequences

In the second step, we 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:

[56]:
```
maxlen = 20000
```

[57]:
```
X_train = pad_sequences(X_train,
                        truncating='pre',
                        padding='pre',
                        maxlen=maxlen)

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

[58]:
```
X_train.shape, X_test.shape
```

[58]: ((14716, 20000), (1636, 20000))

1.5 Define Model Architecture

[59]:
```
K.clear_session()
```

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.

[60]:
```
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:

[61]:
```
input_dim = X_train.max() + 1
```

[62]:
```
rnn = Sequential([
    Embedding(input_dim=input_dim,
              output_dim=embedding_size,
              input_length=maxlen,
             name='EMB'),
    BatchNormalization(name='BN1'),
    Bidirectional(GRU(32), name='BD1'),
    BatchNormalization(name='BN2'),
    Dropout(.1, name='DO1'),
    Dense(5, name='D'),
    Dense(1, activation='linear', name='OUT')
])
```

The resulting model has over 2 million parameters.

[63]:
```
rnn.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
EMB (Embedding)              (None, 20000, 100)        2500000
_________________________________________________________________
BN1 (BatchNormalization)     (None, 20000, 100)        400
_________________________________________________________________
BD1 (Bidirectional)          (None, 64)                25728
_________________________________________________________________
BN2 (BatchNormalization)     (None, 64)                256
_________________________________________________________________
DO1 (Dropout)                (None, 64)                0
_________________________________________________________________
D (Dense)                    (None, 5)                 325
_________________________________________________________________
```

```
OUT (Dense)                  (None, 1)                 6
=================================================================
Total params: 2,526,715
Trainable params: 2,526,387
Non-trainable params: 328
_________________________________________________________________
```

[64]:
```
rnn.compile(loss='mse',
            optimizer='Adam',
            metrics=[RootMeanSquaredError(name='RMSE'),
                     MeanAbsoluteError(name='MAE')])
```

1.6 Train model

[65]:
```
early_stopping = EarlyStopping(monitor='val_MAE',
                               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:

[66]:
```
training = rnn.fit(X_train,
                   y_train,
                   batch_size=32,
                   epochs=100,
                   validation_data=(X_test, y_test),
                   callbacks=[early_stopping],
                   verbose=1)
```

```
Epoch 1/100
460/460 [==============================] - 387s 840ms/step - loss: 0.1059 -
RMSE: 0.3255 - MAE: 0.2010 - val_loss: 0.0085 - val_RMSE: 0.0920 - val_MAE:
0.0614
Epoch 2/100
460/460 [==============================] - 382s 831ms/step - loss: 0.0223 -
RMSE: 0.1494 - MAE: 0.0850 - val_loss: 0.0075 - val_RMSE: 0.0867 - val_MAE:
0.0529
Epoch 3/100
460/460 [==============================] - 378s 823ms/step - loss: 0.0120 -
RMSE: 0.1094 - MAE: 0.0651 - val_loss: 0.0071 - val_RMSE: 0.0841 - val_MAE:
0.0520
Epoch 4/100
460/460 [==============================] - 386s 839ms/step - loss: 0.0092 -
RMSE: 0.0961 - MAE: 0.0575 - val_loss: 0.0067 - val_RMSE: 0.0821 - val_MAE:
0.0494
Epoch 5/100
460/460 [==============================] - 384s 835ms/step - loss: 0.0085 -
RMSE: 0.0919 - MAE: 0.0550 - val_loss: 0.0068 - val_RMSE: 0.0822 - val_MAE:
```

```
0.0503
Epoch 6/100
460/460 [==============================] - 384s 835ms/step - loss: 0.0082 -
RMSE: 0.0905 - MAE: 0.0543 - val_loss: 0.0068 - val_RMSE: 0.0826 - val_MAE:
0.0507
Epoch 7/100
460/460 [==============================] - 384s 834ms/step - loss: 0.0079 -
RMSE: 0.0891 - MAE: 0.0540 - val_loss: 0.0074 - val_RMSE: 0.0862 - val_MAE:
0.0549
Epoch 8/100
460/460 [==============================] - 385s 837ms/step - loss: 0.0077 -
RMSE: 0.0876 - MAE: 0.0532 - val_loss: 0.0082 - val_RMSE: 0.0907 - val_MAE:
0.0593
Epoch 9/100
460/460 [==============================] - 378s 822ms/step - loss: 0.0074 -
RMSE: 0.0860 - MAE: 0.0528 - val_loss: 0.0087 - val_RMSE: 0.0934 - val_MAE:
0.0629
```

1.7 Evaluate the Results

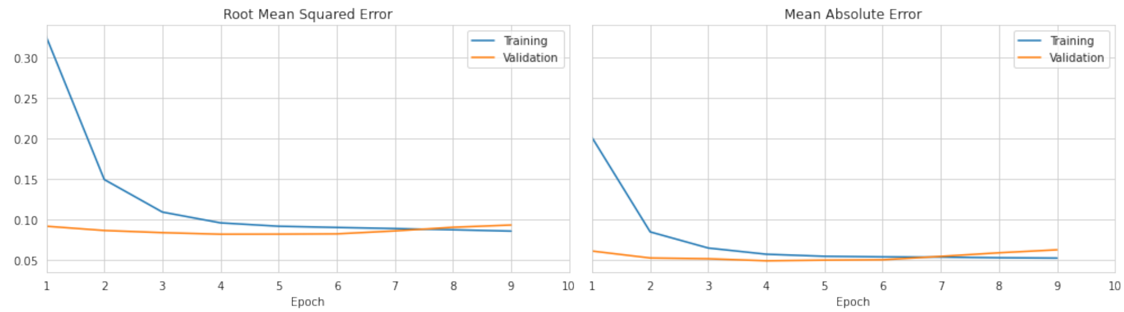
[67]:
```
df = pd.DataFrame(training.history)
df.to_csv(results_path / 'rnn_sec.csv', index=False)
```

[68]:
```
df.index += 1
```

[69]:
```
fig, axes = plt.subplots(ncols=2, figsize=(14, 4), sharey=True)
plot_data = (df[['RMSE', 'val_RMSE']].rename(columns={'RMSE': 'Training',
                                                       'val_RMSE':␣
 ↪'Validation'}))
plot_data.plot(ax=axes[0], title='Root Mean Squared Error')

plot_data = (df[['MAE', 'val_MAE']].rename(columns={'MAE': 'Training',
                                                     'val_MAE': 'Validation'}))
plot_data.plot(ax=axes[1], title='Mean Absolute Error')

for i in [0, 1]:
    axes[i].set_xlim(1, 10)
    axes[i].set_xlabel('Epoch')
fig.tight_layout()
fig.savefig(results_path / 'sec_cv_performance', dpi=300);
```



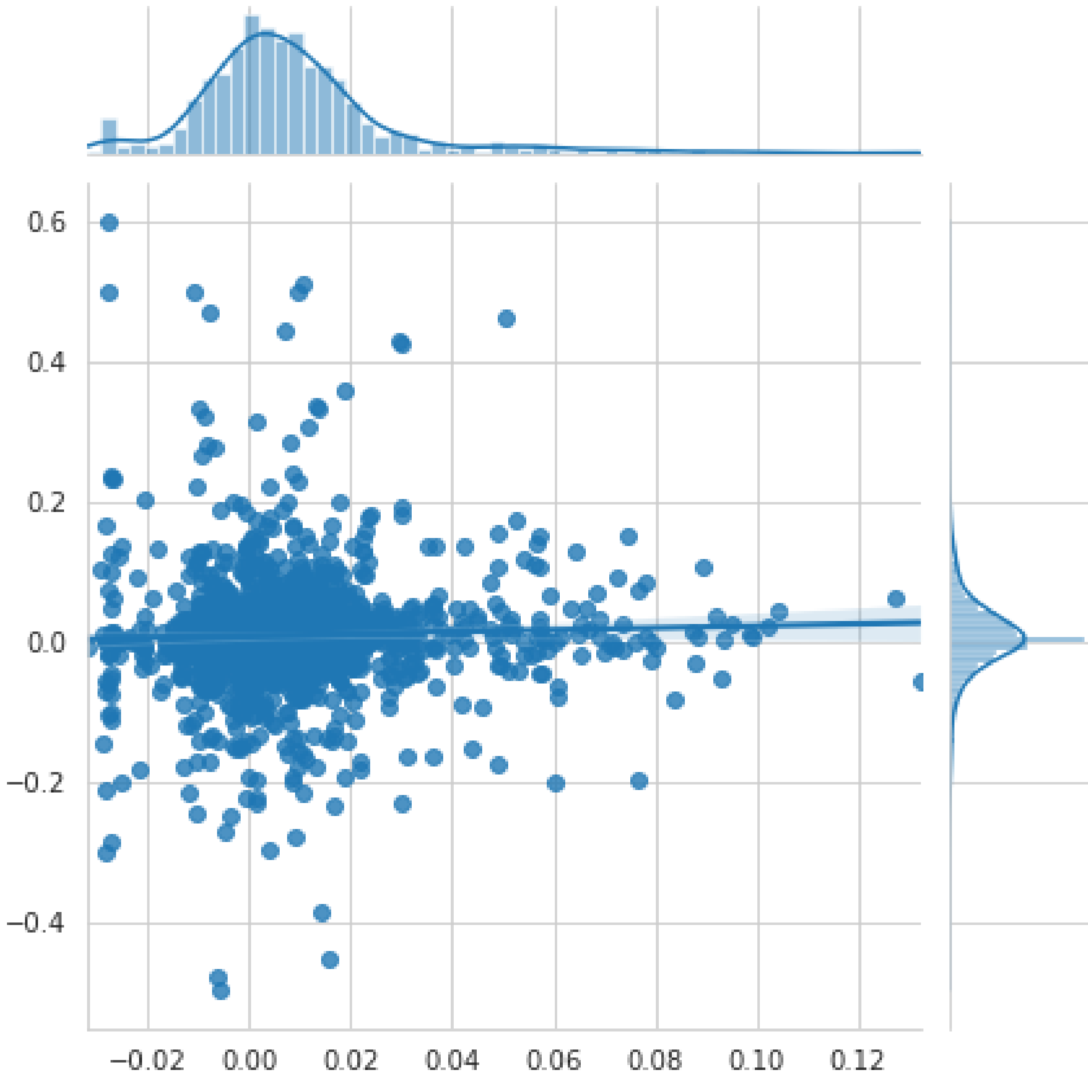
[70]:
```python
y_score = rnn.predict(X_test)
```

[71]:
```python
rho, p = spearmanr(y_score.squeeze(), y_test)
```

[75]:
```python
print(f'Information Coefficient: {rho*100:.2f} ({p:.2%})')
```

```
Information Coefficient: 7.65 (0.20%)
```

[74]:
```python
g = sns.jointplot(y_score.squeeze(), y_test, kind='reg');
```


0.6
0.4
0.2
0.0
−0.2
−0.4
−0.02
0.00
0.02
0.04
0.06
0.08
0.10
0.12

```
[ ]:
```