Build a biGRU neural network model with additional features

In this notebook, we are going to build a bidirectional Gated Recurrent Unit neural network model, which will use the clean_review feature as in the previous notebook, but this time we will also use additional features that we created in the first tutorial. The extra features that will be passed to the model are the following: polarity, subjectivity, word count, Part-Of-Speech tags ratio, uppercase words ratio and digits ratio.

In the end, the model will be evaluated on the test set to determine the generalization error.

Building and training the model

Let's start with importing all indispensable libraries.

In [1]:
```
from batch_iterator import BatchIterator
from early_stopping import EarlyStopping
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch import device
from tqdm import tqdm_notebook
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import StandardScaler
```

Now, we are going to load the tarining and validation sets. We will import sets with all columns except the review column.

In [2]:
```
# Import the dataset.
train_dataset = pd.read_csv('drugreview/drugreview_feat_clean/train_fe
                            usecols=['clean_review', 'polarity', 'subj
                                     'PROPN', 'VERB', 'NOUN', 'PUNCT'

# Change columns order
train_dataset['label'] = train_dataset.rating >= 5
train_dataset = train_dataset[['clean_review', 'polarity', 'subjectivi
                               'PROPN', 'VERB', 'NOUN', 'PUNCT', 'ADJ'
```

In [3]:
```
# Depict the first 5 rows of the training set
train_dataset = train_dataset.dropna()
train_dataset.head()
```

Out[3]:

	clean_review	polarity	subjectivity	word_count	UPPERCASE	DIGITS	PROPN	VERB	N(
1	okay anxiety gotten worse past couple years po...	0.12980	0.4067	150.0	0.06000	0.00000	0.0	0.0	
6	reading possible effects scary medicine gave l...	0.07983	0.5347	90.0	0.07776	0.01111	0.0	0.0	
9	clonazepam effective controlling agitation pro...	0.23700	0.6855	118.0	0.03390	0.00848	0.0	0.0	
11	experienced effects considering anorexia nervo...	0.50630	0.5750	47.0	0.10640	0.02127	0.0	0.0	
12	i've gianvi months skin clear didn't...	-0.10710	0.3894	54.0	0.05554	0.01852	0.0	0.0	

Before we create batches of our data, we have to normalize the numerical features so that we remove the possibility that one variable is the orders of magnitude greater than other variables, which might cause that the first one dominates other features in the dataset and this is something we don't want to happen in our model.

The polarity is within the range [-1.0, 1.0], and the subjectivity is within the range [0.0, 1.0], thus these both features don't require the scaling.

In [4]:
```
# Instantiate the StandardScaler
train_scaler = StandardScaler()
# Scale the features
train_dataset.iloc[:, 3:11] = train_scaler.fit_transform(train_dataset
```

In [5]:
```
train_dataset.head()
```

Out[5]:

	clean_review	polarity	subjectivity	word_count	UPPERCASE	DIGITS	PROPN	VEI
1	okay anxiety gotten worse past couple years po...	0.12980	0.4067	1.485427	0.051652	-0.809541	-0.21187	-0.3016
6	reading possible effects scary medicine gave l...	0.07983	0.5347	0.123681	0.516477	-0.242778	-0.21187	-0.3016
9	clonazepam effective controlling agitation pro...	0.23700	0.6855	0.759163	-0.631453	-0.376944	-0.21187	-0.3016
11	experienced effects considering anorexia nervo...	0.50630	0.5750	-0.852236	1.266060	0.275523	-0.21187	-0.3016
12	i've gianvi months skin clear didn't...	-0.10710	0.3894	-0.693366	-0.065078	0.135235	-0.21187	-0.3016

In [7]:
```
# Import the dataset. Use clean_review and label columns
val_dataset = pd.read_csv('drugreview/drugreview_feat_clean/val_feat_c
                          usecols=['clean_review', 'polarity', 'subjec
                                   'PROPN', 'VERB', 'NOUN', 'PUNCT'

# Change columns order
val_dataset['label'] = val_dataset.rating >= 5
val_dataset = val_dataset[['clean_review', 'polarity', 'subjectivity',
                           'PROPN', 'VERB', 'NOUN', 'PUNCT', 'ADJ', 'l
```

In [8]:
```
# Depict the first 5 rows of the validation set
val_dataset = val_dataset.dropna(0)
val_dataset.head()
```

Out[8]:

	clean_review	polarity	subjectivity	word_count	UPPERCASE	DIGITS	PROPN	VERB	NO
1	4yrs having nexaplon implant mental physical h...	0.1217	0.4553	137.0	0.02919	0.007298	0.0	0.0	
4	l5 s1 lumbar herniated disc surgery weeks surg...	0.1459	0.3792	69.0	0.04350	0.028990	0.0	0.0	
5	far lot acne clear tea tree broke decided birt...	0.2375	0.5540	85.0	0.11770	0.011765	0.0	0.0	
6	insulin works fine trouble pen pain pen jammed...	-0.0958	0.5500	47.0	0.08510	0.000000	0.0	0.0	
7	nexplanon option work iud painful insert pills...	-0.0353	0.4426	135.0	0.05927	0.007410	0.0	0.0	

In [9]:
```
# Instantiate the StandardScaler
val_scaler = StandardScaler()
# Scale the features
val_dataset.iloc[:, 3:11] = val_scaler.fit_transform(val_dataset.iloc[
```

In [10]:
```
val_dataset.head()
```

Out[10]:

	clean_review	polarity	subjectivity	word_count	UPPERCASE	DIGITS	PROPN	VER
1	4yrs having nexaplon implant mental physical h...	0.1217	0.4553	1.142500	-0.762432	-0.445881	-0.217982	-0.29772
4	l5 s1 lumbar herniated disc surgery weeks surg...	0.1459	0.3792	-0.332093	-0.387531	0.650841	-0.217982	-0.29772
5	far lot acne clear tea tree broke decided birt...	0.2375	0.5540	0.014870	1.556398	-0.220035	-0.217982	-0.29772
6	insulin works fine trouble pen pain pen jammed...	-0.0958	0.5500	-0.809167	0.702326	-0.814859	-0.217982	-0.29772
7	nexplanon option work iud painful insert pills...	-0.0353	0.4426	1.099130	0.025619	-0.440219	-0.217982	-0.29772

Below we will use the BatchIterator class defined in the *vocabulary* notebook to create the vocabulary, trim sequences in terms of the rare word occurrence and the length, map words to their numerical representation (word2index), furthermore BatchIterator sorts dataset examples, generates batches, performs sequence padding and enables to use it instance to iterate through all batches.

We will use the min_word_count=3 and max_seq_len=0.9 as in the previous model. The batch_size entry value will be the 256, but it will turn out during the fine-tuning process (that is not presented) that for the dataset with more features the model achieves superior performance with the smaller size of the batch (batch_size=64) which helps in preventing overfitting.

In [11]:
```
train_iterator = BatchIterator(train_dataset, batch_size=64, vocab_cre
                               word2index=None, sos_token='<SOS>', eos
                               pad_token='<PAD>', min_word_count=3, ma
                               use_pretrained_vectors=True, glove_path
                               weights_file_name='glove/weights_train.
```

```
Trimmed vocabulary using as minimum count threashold: count = 3.00
14773/39267 tokens has been retained
Trimmed input strings vocabulary
Trimmed input sequences lengths to the length of: 59
Mapped words to indices
Start creating glove_word2vector dictionary
Extracted 12312/14777 of pre-trained word vectors.
2465 vectors initialized to random numbers
Weights vectors saved into glove/weights_train.npy
Batches created
```

In [12]:
```
val_iterator = BatchIterator(val_dataset, batch_size=64, vocab_createc
                             word2index=train_iterator.word2index, sos
                             unk_token='<UNK>', pad_token='<PAD>', mir
                             max_seq_len=0.9, use_pretrained_vectors=T
                             glove_name='glove.6B.100d.txt', weights_f
```

```
Trimmed vocabulary using as minimum count threashold: count = 3.00
7720/19770 tokens has been retained
Trimmed input strings vocabulary
Trimmed input sequences lengths to the length of: 58
Mapped words to indices
Start creating glove_word2vector dictionary
Extracted 12475/15036 of pre-trained word vectors.
2561 vectors initialized to random numbers
Weights vectors saved into glove/weights_val.npy
Batches created
```

We have to check out how batches that we created look like before we pass them into the model. For the record, the set of batches for input and output variables is returned as a dictionary, thus we will just look at the dictionary keys to find out how to extract particular variables.

In [13]:
```
for batches in train_iterator:
    print(batches.keys())
    break
```

```
dict_keys(['input_feat', 'input_seq', 'target', 'x_lengths'])
```

As we can see above we can distinguish the following batches: *input_feat* that comprises all additional features, *input_seq* that contains cleaned reviews, *target batch* that includes target labels and *x_lengths* batch that encompasses input sequences lengths.

Notice that the *input_seq* batch has the dimensions: (batch_size, seq_len), while *input_feat* batch has the shape of (batch_size, num_features).

In [14]:
```
for batches in train_iterator:

    # Unpack the dictionary of batches
    input_seq, input_feat, target, x_lengths = batches['input_seq'], b
                                               batches['target'], batc

    print('input_seq shape: ', input_seq.size())
    print('input_feat shape: ', input_feat.size())
    print('target shape: ', target.size())
    print('x_lengths shape: ', x_lengths.size())
    break
```

```
input_seq shape:  torch.Size([64, 49])
input_feat shape:  torch.Size([64, 10])
target shape:  torch.Size([64])
x_lengths shape:  torch.Size([64])
```

In [15]:
```
for batches in val_iterator:
    # Unpack the dictionary of batches
    input_seq, input_feat, target, x_lengths = batches['input_seq'], b
                                               batches['target'], batc
    print('input_seq shape: ', input_seq.size())
    print('input_feat shape: ', input_feat.size())
    print('target shape: ', target.size())
    print('x_lengths shape: ', x_lengths.size())
    break
```

```
input_seq shape:  torch.Size([64, 31])
input_feat shape:  torch.Size([64, 10])
target shape:  torch.Size([64])
x_lengths shape:  torch.Size([64])
```

Now we are going to build a biGRU model that will use the *input_feat* batch as additional features vector.

In [16]:
```
class BiGRU(nn.Module):
    """BiDirectional GRU neural network model.

    Parameters
    ----------
```

```
    hidden_size: int
        Number of features in the hidden state.
    vocab_size: int
        The size of the vocabulary.
    n_extra_feat: int
        Number of additional features.
    embedding_dim: int
        The size of each embedding vector.
    output_size: int
        Number of classes.
    n_layers: int, optional (default=1)
        Number of stacked recurrent layers.
    dropout: float, optional (default=0.2)
        Probability of an element of the tensor to be zeroed.
    spatial_dropout: boolean, optional (default=True)
        Whether to use the spatial dropout.
    bidirectional: boolean, optional (default=True)
        Whether to use the bidirectional GRU.

    """

    def __init__(self, hidden_size, vocab_size, n_extra_feat, embeddin
                 spatial_dropout=True, bidirectional=True):

        # Inherit everything from the nn.Module
        super(BiGRU, self).__init__()

        # Initialize attributes
        self.hidden_size = hidden_size
        self.vocab_size = vocab_size
        self.n_extra_feat = n_extra_feat
        self.embedding_dim = embedding_dim
        self.output_size = output_size
        self.n_layers = n_layers
        self.dropout_p = dropout
        self.spatial_dropout = spatial_dropout
        self.bidirectional = bidirectional
        self.n_directions = 2 if self.bidirectional else 1

        # Initialize layers
        self.embedding = nn.Embedding(self.vocab_size, self.embedding_
        self.dropout = nn.Dropout(self.dropout_p)
        if self.spatial_dropout:
            self.spatial_dropout1d = nn.Dropout2d(self.dropout_p)
        self.gru = nn.GRU(self.embedding_dim, self.hidden_size, num_la
                          dropout=(0 if n_layers == 1 else self.dropou
                          bidirectional=self.bidirectional)
        # Linear layer input size is equal to hidden_size * 3 + n_extr
        # we will concatenate max_pooling ,avg_pooling, last hidden st
        self.linear = nn.Linear(self.hidden_size * 3 + self.n_extra_fe
```

```
    def forward(self, input_seq, input_feat, input_lengths, hidden=Non
        """Forward propagate through the neural network model.

        Parameters
        ----------
        input_seq: torch.Tensor
            Batch of input sequences.
        input_feat: torch.Tensor
            Batch of additional features.
        input_lengths: torch.LongTensor
            Batch containing sequences lengths.
        hidden: torch.FloatTensor, optional (default=None)
            Tensor containing initial hidden state.

        Returns
        -------
        torch.Tensor
            Logarithm of softmaxed input tensor.

        """
        # Extract batch_size
        self.batch_size = input_seq.size(0)

        # Embeddings shapes
        # Input: (batch_size,  seq_length)
        # Output: (batch_size, seq_length, embedding_dim)
        emb_out = self.embedding(input_seq)

        if self.spatial_dropout:
            # Convert to (batch_size, embedding_dim, seq_length)
            emb_out = emb_out.permute(0, 2, 1)
            emb_out = self.spatial_dropout1d(emb_out)
            # Convert back to (batch_size, seq_length, embedding_dim)
            emb_out = emb_out.permute(0, 2, 1)
        else:
            emb_out = self.dropout(emb_out)

        # Pack padded batch of sequences for RNN module
        packed_emb = nn.utils.rnn.pack_padded_sequence(emb_out, input_

        # GRU input/output shapes, if batch_first=True
        # Input: (batch_size, seq_len, embedding_dim)
        # Output: (batch_size, seq_len, hidden_size*num_directions)
        # Number of directions = 2 when used bidirectional, otherwise
        # shape of hidden: (n_layers x num_directions, batch_size, hid
        # Hidden state defaults to zero if not provided
        gru_out, hidden = self.gru(packed_emb, hidden)
        # gru_out: tensor containing the output features h_t from the
        # gru_out comprises all the hidden states in the last layer ("
```

```
        # For biGRu gru_out is the concatenation of a forward GRU repr
        # hidden (h_n) comprises the hidden states after the last time

        # Extract and sum last hidden state
        # Input hidden shape: (n_layers x num_directions, batch_size,
        # Separate hidden state layers
        hidden = hidden.view(self.n_layers, self.n_directions, self.ba
        last_hidden = hidden[-1]
        # last hidden shape (num_directions, batch_size, hidden_size)
        # Sum the last hidden state of forward and backward layer
        last_hidden = torch.sum(last_hidden, dim=0)
        # Summed last hidden shape (batch_size, hidden_size)

        # Pad a packed batch
        # gru_out output shape: (batch_size, seq_len, hidden_size*num_
        gru_out, lengths = nn.utils.rnn.pad_packed_sequence(gru_out, b

        # Sum the gru_out along the num_directions
        if self.bidirectional:
            gru_out = gru_out[:,:,:self.hidden_size] + gru_out[:,:,sel

        # Select the maximum value over each dimension of the hidden r
        # Permute the input tensor to dimensions: (batch_size, hidden,
        # Output dimensions: (batch_size, hidden_size)
        max_pool = F.adaptive_max_pool1d(gru_out.permute(0,2,1), (1,))

        # Consider the average of the representations (mean pooling)
        # Sum along the batch axis and divide by the corresponding len
        # Output shape: (batch_size, hidden_size)
        avg_pool = torch.sum(gru_out, dim=1) / lengths.view(-1,1).type

        # Concatenate max_pooling, avg_pooling, hidden state and input
        concat_out = torch.cat([last_hidden, max_pool, avg_pool, input

        # concat_out = self.dropout(concat_out)
        out = self.linear(concat_out)
        return F.log_softmax(out, dim=-1)


    def add_loss_fn(self, loss_fn):
        """Add loss function to the model.

        """
        self.loss_fn = loss_fn


    def add_optimizer(self, optimizer):
        """Add optimizer to the model.

        """
```

```
        self.optimizer = optimizer


    def add_device(self, device=torch.device('cpu')):
        """Specify the device.

        """
        self.device = device


    def train_model(self, train_iterator):
        """Perform single training epoch.

        Parameters
        ----------
        train_iterator: BatchIterator
            BatchIterator class object containing training batches.

        Returns
        -------
        train_losses: list
            List of the training average batch losses.
        avg_loss: float
            Average loss on the entire training set.
        accuracy: float
            Models accuracy on the entire training set.

        """
        self.train()

        train_losses = []
        losses = []
        losses_list = []
        num_seq = 0
        batch_correct = 0

        for i, batches in tqdm_notebook(enumerate(train_iterator, 1),
            input_seq, input_feat, target, x_lengths = batches['input_
                                                       batches['target

            input_seq.to(self.device)
            input_feat.to(self.device)
            target.to(self.device)
            x_lengths.to(self.device)

            self.optimizer.zero_grad()

            pred = self.forward(input_seq, input_feat, x_lengths)
            loss = self.loss_fn(pred, target)
            loss.backward()
            losses.append(loss.data.cpu().numpy())
```

```
            losses.append(loss.data.cpu().numpy())
            self.optimizer.step()

            losses_list.append(loss.data.cpu().numpy())

            pred = torch.argmax(pred, 1)

            if self.device.type == 'cpu':
                batch_correct += (pred.cpu() == target.cpu()).sum().it

            else:
                batch_correct += (pred == target).sum().item()

            num_seq += len(input_seq)

            if i % 100 == 0:
                avg_train_loss = np.mean(losses)
                train_losses.append(avg_train_loss)

                accuracy = batch_correct / num_seq

                print('Iteration: {}. Average training loss: {:.4f}. A
                      .format(i, avg_train_loss, accuracy))

                losses = []

            avg_loss = np.mean(losses_list)
            accuracy = batch_correct / num_seq

        return train_losses, avg_loss, accuracy


    def evaluate_model(self, eval_iterator, conf_mtx=False):
        """Perform the one evaluation epoch.

        Parameters
        ----------
        eval_iterator: BatchIterator
            BatchIterator class object containing evaluation batches.
        conf_mtx: boolean, optional (default=False)
            Whether to print the confusion matrix at each epoch.

        Returns
        -------
        eval_losses: list
            List of the evaluation average batch losses.
        avg_loss: float
            Average loss on the entire evaluation set.
        accuracy: float
            Models accuracy on the entire evaluation set.
        conf_matrix: list
```

```
            Confusion matrix.

        """
        self.eval()

        eval_losses = []
        losses = []
        losses_list = []
        num_seq = 0
        batch_correct = 0
        pred_total = torch.LongTensor()
        target_total = torch.LongTensor()

        with torch.no_grad():
            for i, batches in tqdm_notebook(enumerate(eval_iterator, 1
                input_seq, input_feat, target, x_lengths = batches['in
                                                           batches['ta

                input_seq.to(self.device)
                input_feat.to(self.device)
                target.to(self.device)
                x_lengths.to(self.device)

                pred = self.forward(input_seq, input_feat, x_lengths)
                loss = self.loss_fn(pred, target)
                losses.append(loss.data.cpu().numpy())
                losses_list.append(loss.data.cpu().numpy())

                pred = torch.argmax(pred, 1)

                if self.device.type == 'cpu':
                    batch_correct += (pred.cpu() == target.cpu()).sum(

                else:
                    batch_correct += (pred == target).sum().item()

                num_seq += len(input_seq)

                pred_total = torch.cat([pred_total, pred], dim=0)
                target_total = torch.cat([target_total, target], dim=0

                if i % 100 == 0:
                    avg_batch_eval_loss = np.mean(losses)
                    eval_losses.append(avg_batch_eval_loss)

                    accuracy = batch_correct / num_seq

                    print('Iteration: {}. Average evaluation loss: {:.
                          .format(i, avg_batch_eval_loss, accuracy))

                    losses = []
```

```
                losses = []

            avg_loss_list = []

            avg_loss = np.mean(losses_list)
            accuracy = batch_correct / num_seq

            conf_matrix = confusion_matrix(target_total.view(-1), prec

        if conf_mtx:
            print('\tConfusion matrix: ', conf_matrix)

        return eval_losses, avg_loss, accuracy, conf_matrix
```

Now we will instantiate the model, add loss function, optimizer, and device to it and begin the training.

In [17]:
```
# Initialize parameters
hidden_size = 8
vocab_size = len(train_iterator.word2index)
n_extra_feat = 10
embedding_dim = 200
output_size = 2
n_layers = 1
dropout = 0.5
learning_rate = 0.001
epochs = 20
spatial_dropout = True

# Check whether system supports CUDA
CUDA = torch.cuda.is_available()

model = BiGRU(hidden_size, vocab_size, n_extra_feat, embedding_dim, ou
              spatial_dropout, bidirectional=True)

# Move the model to GPU if possible
if CUDA:
    model.cuda()

model.add_loss_fn(nn.NLLLoss())

optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
model.add_optimizer(optimizer)

device = torch.device('cuda' if CUDA else 'cpu')

model.add_device(device)
```

```
# Instantiate the EarlyStopping
early_stop = EarlyStopping(wait_epochs=2)

train_losses_list, train_avg_loss_list, train_accuracy_list = [], [],
eval_avg_loss_list, eval_accuracy_list, conf_matrix_list = [], [], []

for epoch in range(epochs):

    print('\nStart epoch [{}/{}]'.format(epoch+1, epochs))

    train_losses, train_avg_loss, train_accuracy = model.train_model(t

    train_losses_list.append(train_losses)
    train_avg_loss_list.append(train_avg_loss)
    train_accuracy_list.append(train_accuracy)

    _, eval_avg_loss, eval_accuracy, conf_matrix = model.evaluate_mode

    eval_avg_loss_list.append(eval_avg_loss)
    eval_accuracy_list.append(eval_accuracy)
    conf_matrix_list.append(conf_matrix)

    print('\nEpoch [{}/{}]: Train accuracy: {:.3f}. Train loss: {:.4f}
          .format(epoch+1, epochs, train_accuracy, train_avg_loss, eva

    if early_stop.stop(eval_avg_loss, model, delta=0.003):
        break
```

```
Start epoch [1/20]
```

Training: 708/708 [00:23<00:00,

100% 27.97it/s]

```
Iteration: 100. Average training loss: 0.6353. Accuracy: 0.667
Iteration: 200. Average training loss: 0.5643. Accuracy: 0.708
Iteration: 300. Average training loss: 0.5496. Accuracy: 0.722
Iteration: 400. Average training loss: 0.5272. Accuracy: 0.732
Iteration: 500. Average training loss: 0.5155. Accuracy: 0.738
Iteration: 600. Average training loss: 0.5143. Accuracy: 0.742
Iteration: 700. Average training loss: 0.5254. Accuracy: 0.743
```

Evaluation: 179/179 [00:00<00:00,

100% 231.99it/s]

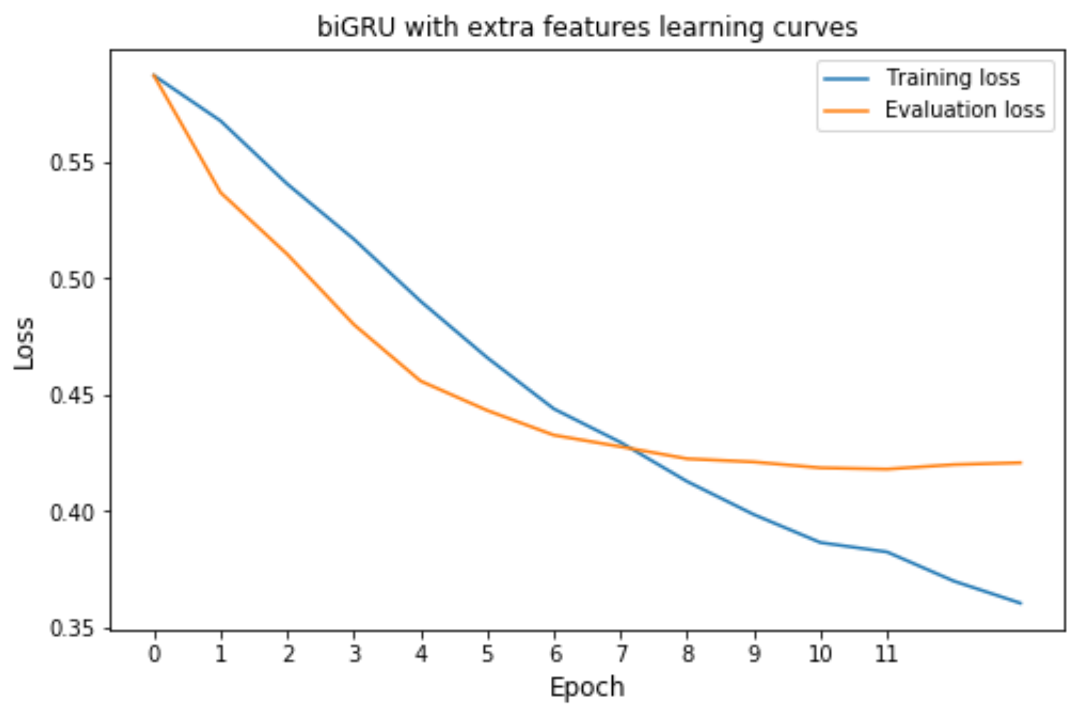
```
Iteration: 100. Average evaluation loss: 0.4819. Accuracy: 0.77
```

The training was stopped by EarlyStopping object because the loss didn't improve for 2 epochs. The best performance of the model was achieved at the 13th epoch with the validation accuracy of 0.843 and the loss of 0.3604. As we can see using extracted features improve the model's predictive ability on the validation set from the value of 0.820 (previous model) to 0.813.

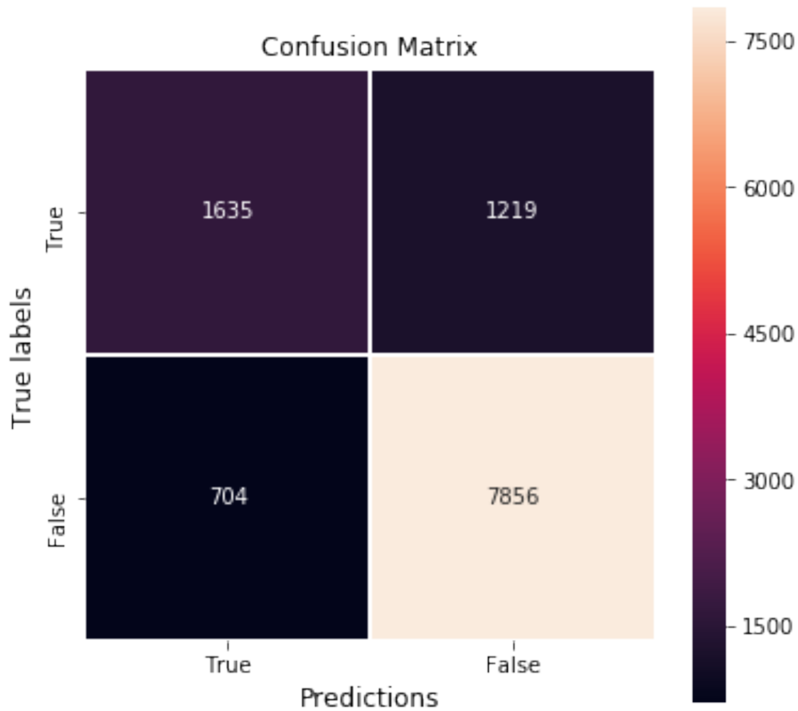
The model's best state was saved to the checkpoint.pt file in the current directory.

In [17]:
```
# Add the dataset initial loss
train_avg_loss_list.insert(0, train_losses_list[0][0])
eval_avg_loss_list.insert(0, train_losses_list[0][0])
```

In [18]:
```
# Plot the training and the validation learning curve
plt.figure(figsize=(8,5))
plt.plot(train_avg_loss_list, label='Training loss')
plt.plot(eval_avg_loss_list, label='Evaluation loss')
plt.xlabel('Epoch', size=12)
plt.ylabel('Loss', size=12)
plt.title('biGRU with extra features learning curves')
plt.xticks(ticks=range(12))
plt.legend()
plt.show()
```



In [18]:
```
# Confusion matrix
plt.figure(figsize=(6,6))
ax = sns.heatmap(conf_matrix, fmt='d', annot=True, linewidths=1, squar
ax.set_xlabel('Predictions', size=12)
ax.set_ylabel('True labels', size=12)
ax.set_title('Confusion Matrix', size=12);
ax.xaxis.set_ticklabels(['True', 'False'])
ax.yaxis.set_ticklabels(['True', 'False'])
ax.set_ylim(2,0)
plt.show()
```



The generalization error

In [19]:
```
# Import the dataset. Use clean_review and label columns
test_dataset = pd.read_csv('drugreview/drugreview_feat_clean/test_feat
                           usecols=['clean_review', 'polarity', 'subj
                                    'PROPN', 'VERB', 'NOUN', 'PUNCT',

# Change columns order
test_dataset['label'] = test_dataset.rating >= 5
test_dataset = test_dataset[['clean_review', 'polarity', 'subjectivity
                             'PROPN', 'VERB', 'NOUN', 'PUNCT', 'ADJ',
```

In [20]:
```
test_dataset = test_dataset.dropna()
test_dataset.head()
```

Out[20]:

	clean_review	polarity	subjectivity	word_count	UPPERCASE	DIGITS	PROPN	VERB	N
0	i've tried antidepressants years citalopr...	0.0000	0.4400	68.0	0.02942	0.00000	0.02942	0.1765	0.
1	son crohn's disease asacol complaints sho...	0.5670	0.6000	48.0	0.00000	0.00000	0.04166	0.1875	0.
2	quick reduction symptoms	0.3333	0.5000	4.0	0.00000	0.00000	0.00000	0.0000	0.
3	contrave combines drugs alcohol smoking opioid...	0.1390	0.5000	143.0	0.06995	0.00000	0.04895	0.2238	0.
4	birth control cycle reading reviews type simil...	0.2610	0.5503	149.0	0.06714	0.01342	0.00000	0.1879	0.

In [21]:
```
# Instantiate the StandardScaler
test_scaler = StandardScaler()
# Scale the features
test_dataset.iloc[:, 3:11] = test_scaler.fit_transform(test_dataset.il
```

In [22]:
```
test_iterator = BatchIterator(test_dataset, batch_size=256, vocab_crea
                              word2index=train_iterator.word2index, sc
                              unk_token='<UNK>', pad_token='<PAD>', mi
                              max_seq_len=0.9, use_pretrained_vectors=
                              glove_name='glove.6B.100d.txt', weights_
```

```
Trimmed vocabulary using as minimum count threashold: count = 3.00
15210/40911 tokens has been retained
Trimmed input strings vocabulary
Trimmed input sequences lengths to the length of: 59
Mapped words to indices
Batches created
```

In [23]:
```
_, test_avg_loss, test_accuracy, test_conf_matrix = model.evaluate_moc
```

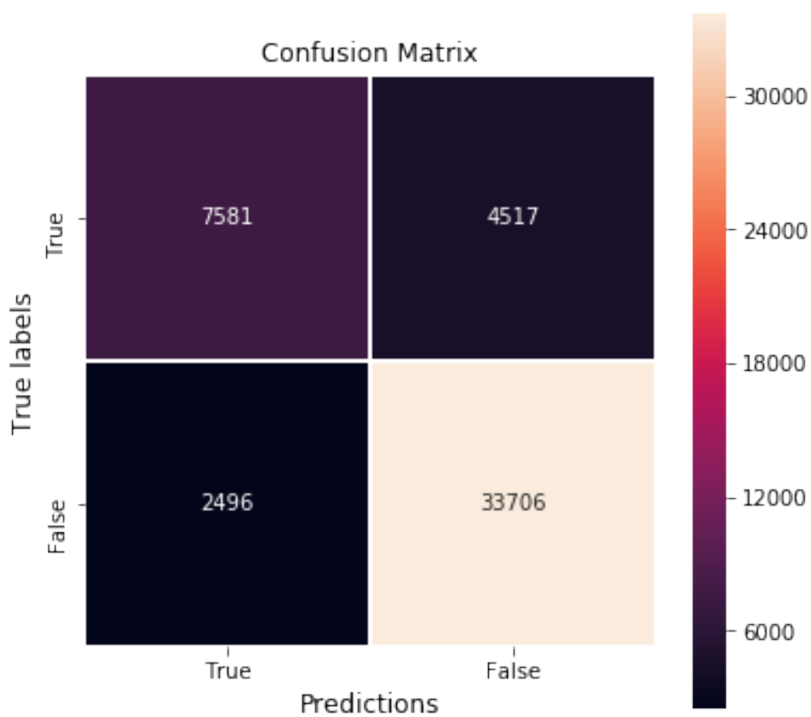
Evaluation: 100% 189/189 [00:02<00:00, 89.50it/s]

```
Iteration: 100. Average evaluation loss: 0.3371. Accuracy: 0.86
```

In [24]:
```
print('Test accuracy: {:.3f}. Test error: {:.3f}'.format(test_accuracy
```

```
Test accuracy: 0.855. Test error: 0.340
```

In [25]:
```
# Confusion matrix
plt.figure(figsize=(6,6))
ax = sns.heatmap(test_conf_matrix, fmt='d', annot=True, linewidths=1,
ax.set_xlabel('Predictions', size=12)
ax.set_ylabel('True labels', size=12)
ax.set_title('Confusion Matrix', size=12);
ax.xaxis.set_ticklabels(['True', 'False'])
ax.yaxis.set_ticklabels(['True', 'False'])
ax.set_ylim(2,0)
plt.show()
```



Thanks to using additional features our model achieved the generalization error of 0.855, which is 0.1 higher than in the model without using extracted features. Even a superior result is possible, but the precise process of hyperparameters fine-tuning will be required.

In [ ]: