After the GRU layer we will concatenate both, the average pooling and max pooling of the hidden representation and the last hidden state of GRU in order to prevent our model from forgetting infromations. This architecture is described in the following paper: https://arxiv.org/pdf/1801.06146.pdf (https://arxiv.org/pdf/1801.06146.pdf). There is also the possibility to get rid of the last hidden state from our model at all, this kind of architecture, that uses max-pooling or avg-pooling is depicted in the paper: https://arxiv.org/pdf/1705.02364.pdf (https://arxiv.org/pdf/1705.02364.pdf).

Building and training the model

Let's start with importing all indispensable libraries.

In [2]:
```
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 tensorboardX import SummaryWriter
```

Now, we are going to load the tarining and validation sets, but we will use only the clean_review column and label column.

In [3]:
```
# Import the dataset. Use clean_review and label columns
train_dataset = pd.read_csv('drugreview/drugreview_feat_clean/train_fe
                            usecols=['clean_review', 'rating'])

# Change columns order
train_dataset['label'] = train_dataset.rating >= 5
train_dataset = train_dataset[['clean_review', 'label']]
```

In [4]:
```
# Depict the first 5 rows of the training set
train_dataset = train_dataset.dropna()
train_dataset.head()
```

Out[4]:

	clean_review	label
1	okay anxiety gotten worse past couple years po...	True
6	reading possible effects scary medicine gave l...	True
9	clonazepam effective controlling agitation pro...	True
11	experienced effects considering anorexia nervo...	True
12	i've gianvi months skin clear didn't...	True

In [6]:
```
# Import the dataset. Use clean_review and label columns
val_dataset = pd.read_csv('drugreview/drugreview_feat_clean/val_feat_c
                          usecols=['clean_review', 'rating'])

# Change columns order

val_dataset['label'] = val_dataset.rating >= 5
val_dataset = val_dataset[['clean_review', 'label']]
```

In [7]:
```
# Depict the first 5 rows of the validation set
val_dataset = val_dataset.dropna()
val_dataset.head()
```

Out[7]:

	clean_review	label
1	4yrs having nexaplon implant mental physical h...	False
4	l5 s1 lumbar herniated disc surgery weeks surg...	True
5	far lot acne clear tea tree broke decided birt...	True
6	insulin works fine trouble pen pain pen jammed...	False
7	nexplanon option work iud painful insert pills...	True

Below we will use the BatchIterator class defined in the previous 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.

In [9]:
```
train_iterator = BatchIterator(train_dataset, batch_size=256, vocab_cr
                               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 [10]:
```
val_iterator = BatchIterator(val_dataset, batch_size=256, vocab_create
                             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 [11]:
```
for batches in train_iterator:
    print(batches.keys())
    break
```

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

Notice that the output batch has the dimensions: (batch_size, seq_len)

In [12]:
```
for batches in train_iterator:
    # Unpack the dictionary of batches
    input_seq, target, x_lengths = batches['input_seq'], batches['targ
    print('input_seq shape: ', input_seq.size())
    print('target shape: ', target.size())
    print('x_lengths shape: ', x_lengths.size())
    break
```

```
input_seq shape:  torch.Size([256, 16])
target shape:  torch.Size([256])
x_lengths shape:  torch.Size([256])
```

In [13]:
```
for batches in val_iterator:
    # Unpack the dictionary of batches
    input_seq, target, x_lengths = batches['input_seq'], batches['targ
    print('input_seq shape: ', input_seq.size())
    print('target shape: ', target.size())
    print('x_lengths shape: ', x_lengths.size())
    break
```

```
input_seq shape:  torch.Size([256, 17])
target shape:  torch.Size([256])
x_lengths shape:  torch.Size([256])
```

Next step is to build the biGRU model.

In [21]:
```
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.
    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, embedding_dim, output_
                 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.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, becuase
        # we will concatenate max_pooling ,avg_pooling and last hidden
        self.linear = nn.Linear(self.hidden_size * 3, self.output_size


    def forward(self, input_seq, input_lengths, hidden=None):
        """Forward propagate through the neural network model.

        Parameters
        ----------
        input_seq: torch.Tensor
            Batch of input sequences.
        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 and last hidden state t
        concat_out = torch.cat([last_hidden, max_pool, avg_pool], dim=

        #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
```

```
        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, target, x_lengths = batches['input_seq'], batch

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

            self.optimizer.zero_grad()

            pred = self.forward(input_seq, x_lengths)
            loss = self.loss_fn(pred, target)
            loss.backward()
            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, target, x_lengths = batches['input_seq'], b

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

                pred = self.forward(input_seq, 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 = []

            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 [22]:

```
# Initialize parameters
hidden_size = 8
vocab_size = len(train_iterator.word2index)
embedding_dim = 200
output_size = 2
n_layers = 1
dropout = 0.5
```

```
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, embedding_dim, output_size, n_l
              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=1)

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
```

100% 87.11it/s]

```
Epoch [6/20]: Train accuracy: 0.801. Train loss: 0.4363. Evaluation
accuracy: 0.821. Evaluation loss: 0.4061

Start epoch [7/20]
```

Training: 177/177 [00:13<00:00,

100% 12.01it/s]

```
Iteration: 100. Average training loss: 0.4240. Accuracy: 0.806
```

Evaluation: 45/45 [00:00<00:00,

100% 86.68it/s]

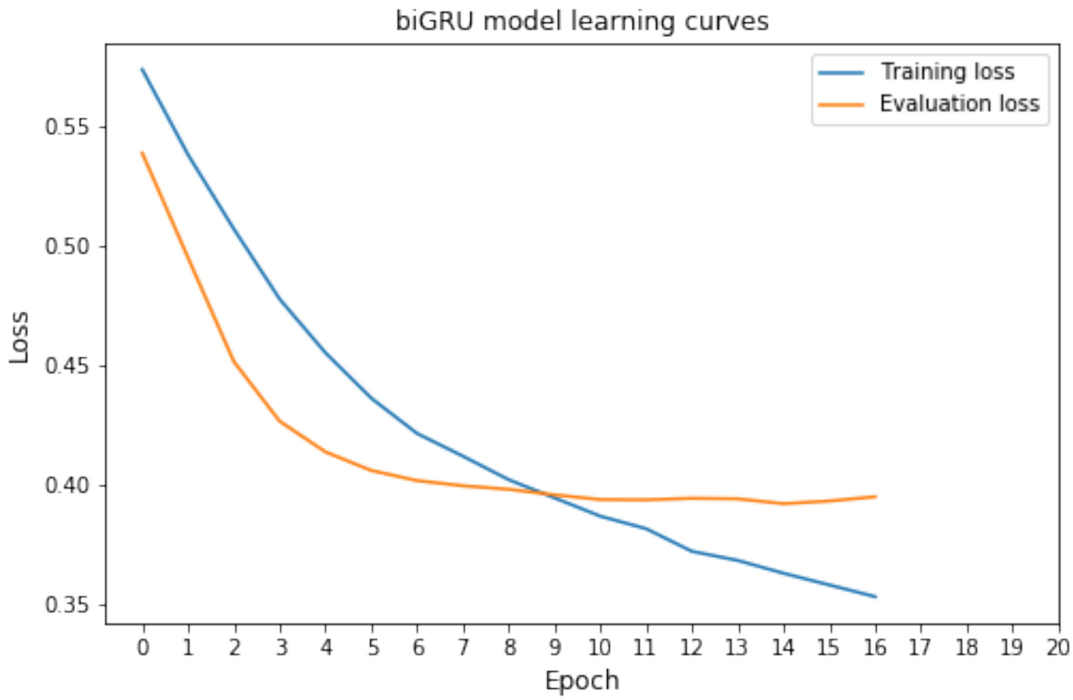
In [19]:

```
# Add the dataset initial loss
#print(len(train_losses_list))
#train_avg_loss_list.insert(0, train_losses_list[0])
#eval_avg_loss_list.insert(0, train_losses_list[0])
#del eval_avg_loss_list[0]
#del train_avg_loss_list.remove[0]
```

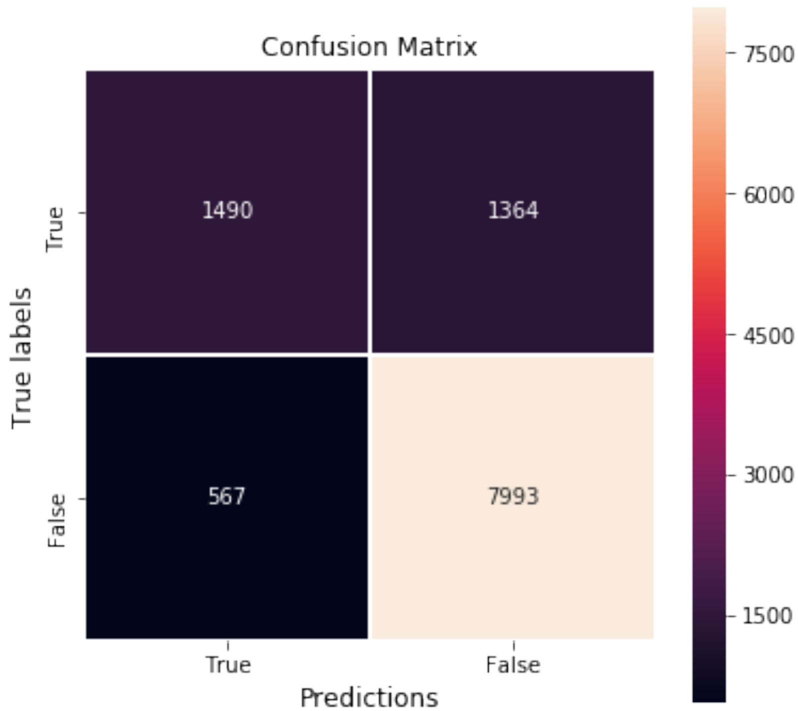
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
last)
<ipython-input-19-24f2f9e83acb> in <module>
      4 #eval_avg_loss_list.insert(0, train_losses_list[0])
      5 del eval_avg_loss_list[0]
----> 6 del train_avg_loss_list.remove[0]

TypeError: 'builtin_function_or_method' object does not support item
deletion
```

In [23]:
```
# 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 model learning curves')
plt.xticks(ticks=range(21))
plt.legend()
plt.show()
```



In [24]:
```
# 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 model achieved the validation accuracy of 0.878, while the training accuracy was 0.908. The model's best state was saved to the *checkpoint.pt* file in the current directory. The training wasn't stopped by EarlyStopping object because the validation loss changes were too small and fluctuated near the same value.

The training process that is presented above regards the model with the tuned hyperparameters. The steps we went through when doing hyperparameters fine-tuning are listed in the next section.

The summary - final set of hyperparameters

Considering all above training trials, we can draw the following conclusions:

- increasing the dropout probability helps in reducing the model's overfitting.
- spatial dropout works better in terms of decreasing the variance problem than the traditional dropout.
- the most improvement in reducing overfitting is due to the reduction of hidden_size.
- using stacked GRU doesn't improve in our case the model's performance.
- reducing the batch_size doesn't significantly affect the model's learning ability what is rather unexpected while increasing the batch_size does improve a bit the model's performance.

The following are the hyperparameters that will be used to finaly train our neural network:

- hidden_size = 8
- embedding_dim = 200
- n_layers = 1
- dropout = 0.5
- learning_rate = 0.001
- epochs = 20
- spatial_dropout = True
- batch_size = 256
- min_word_count = 3
- max_seq_len = 0.9

Below we will use the *tensorboardX* to create the graph of our neural network model that has been depicted at the top of this notebook. You can encounter torch._C.Value issue while using *add.graph()* method, to tackle that I recommend following the *github* thread devoted to this topic:
[https://github.com/lanpa/tensorboardX/issues/483 (https://github.com/lanpa/tensorboardX/issues/483)](https://github.com/lanpa/tensorboardX/issues/483)
namely, you can try to build tensorboardX from source with:
git clone [https://github.com/lanpa/tensorboardX (https://github.com/lanpa/tensorboardX)](https://github.com/lanpa/tensorboardX) && cd tensorboardX && python setup.py install

In [25]:
```
hidden_size = 4
vocab_size = len(train_iterator.word2index)
embedding_dim = 200
n_layers = 2
output_size = 2
spatial_dropout = True
dropout = 0.5

writer = SummaryWriter('runs/exp-1')

for batch in train_iterator:
    input_seq, _, x_lengths = batch['input_seq'], batch['target'], bat

with SummaryWriter(comment='Model graph') as w:
    w.add_graph(BiGRU(hidden_size, vocab_size, embedding_dim, output_s
                        spatial_dropout, bidirectional=True), (input_sec
```

```
graph(%self : ClassType<BiGRU>,
      %input_seq : Long(256, 50),
      %lengths.1 : Long(256)):
  %1 : ClassType<Embedding> = prim::GetAttr[name="embedding"](%self
)
  %weight.1 : Tensor = prim::GetAttr[name="weight"](%1)
  %5 : ClassType<GRU> = prim::GetAttr[name="gru"](%self)
  %6 : Tensor = prim::GetAttr[name="weight_ih_l0"](%5)
  %7 : Tensor = prim::GetAttr[name="weight_hh_l0"](%5)
  %8 : Tensor = prim::GetAttr[name="bias_ih_l0"](%5)
  %9 : Tensor = prim::GetAttr[name="bias_hh_l0"](%5)
  %10 : Tensor = prim::GetAttr[name="weight_ih_l0_reverse"](%5)
  %11 : Tensor = prim::GetAttr[name="weight_hh_l0_reverse"](%5)
  %12 : Tensor = prim::GetAttr[name="bias_ih_l0_reverse"](%5)
  %13 : Tensor = prim::GetAttr[name="bias_hh_l0_reverse"](%5)
  %14 : Tensor = prim::GetAttr[name="weight_ih_l1"](%5)
  %15 : Tensor = prim::GetAttr[name="weight_hh_l1"](%5)
  %16 : Tensor = prim::GetAttr[name="bias_ih_l1"](%5)
  %17 : Tensor = prim::GetAttr[name="bias_hh_l1"](%5)
```

The generalization error

In [26]:
```
# Import the dataset. Use clean_review and label columns
test_dataset = pd.read_csv('drugreview/drugreview_feat_clean/test_feat
                           usecols=['clean_review', 'rating'])

# Change columns order
test_dataset['label'] = test_dataset.rating >= 5
test_dataset = test_dataset[['clean_review', 'label']]
```

In [27]:
```
test_dataset = test_dataset.dropna()
test_dataset.head()
```

Out[27]:

	clean_review	label
0	i've tried antidepressants years citalopr...	True
1	son crohn's disease asacol complaints sho...	True
2	quick reduction symptoms	True
3	contrave combines drugs alcohol smoking opioid...	True
4	birth control cycle reading reviews type simil...	True

In [29]:
```
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 [30]:
```
_, test_avg_loss, test_accuracy, test_conf_matrix = model.evaluate_moc
```

Evaluation: 189/189 [00:02<00:00,

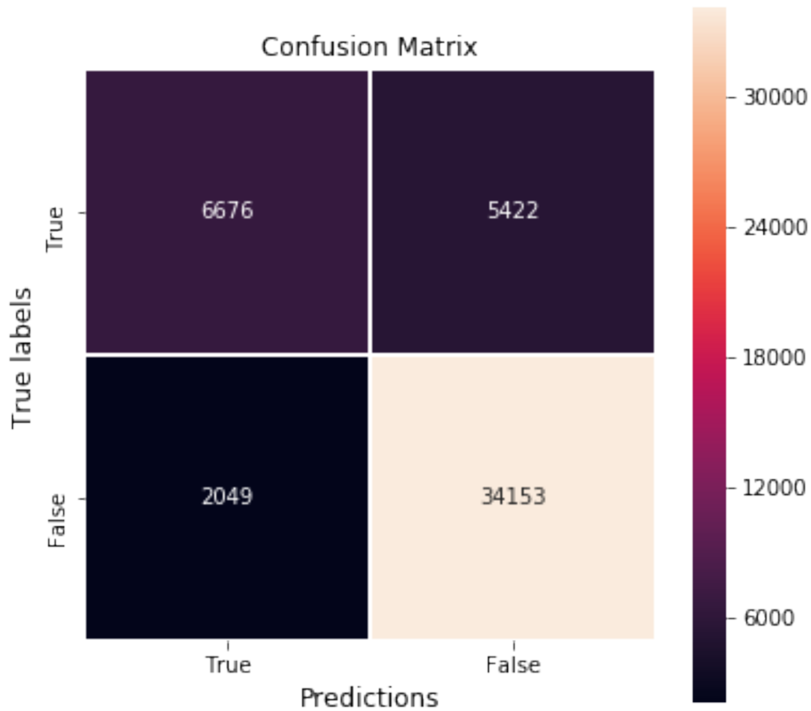
100% 100.98it/s]

```
Iteration: 100. Average evaluation loss: 0.3597. Accuracy: 0.84
```

In [31]:
```
print('Test accuracy: {:.3f}. Test error: {:.3f}'.format(test_accuracy
```

```
Test accuracy: 0.845. Test error: 0.358
```

In [32]:
```
# 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()
```



The generalization accuracy of the biGRU model equals 0.845. As we can see on the above plot of the confusion matrix the both, positive and negative classes were similarly numerous, and the prediction mistakes amount (TN, FP) is also very similar, so model learned both classes in the same detail.

In [ ]: