Build a self-attention Transformer model

In this notebook, we will build the Transformer model for the classification task. The main architecture of the Transformer is derived from the paper: https://arxiv.org/pdf/1706.03762.pdf (https://arxiv.org/pdf/1706.03762.pdf), but to be able to perform text classification we have to re-build the model a bit by applying the Max or Avg Pooling according to https://arxiv.org/pdf/1705.02364.pdf (https://arxiv.org/pdf/1705.02364.pdf), where instead of using hidden representations we will us the last Transfomer block output.

The Transformer is solely based on the self-attention mechanism, disposing recurrent units or convolution layers at all, thanks to which that architecture is superior in terms of the prediction quality and the training time. The Transformer allows for significantly more parallelization and keeps also the ability of discerning long-term dependencies. To increase the generalization performance of the model we will use the label smoothing method.

The model is going to be trained on the clean_review column from the training dataset. In the end, the model will be evaluated on the test set to determine the generalization error.

We will perform the hyperparameter fine-tuning and visualize model's learning curves to compare the model's performance while working on different set of parameters.

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 tensorboardX import SummaryWriter
```

To train the model we will use the clean_review column from the training set as well as the label column.

In [2]:
```
# 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 [3]:
```
# Depict the first 5 rows of the training set
train_dataset = train_dataset.dropna()
train_dataset.head()
```

Out[3]:

	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

To fine-tune the hyperparameters we will evaluate the model on a validation set.

In [5]:
```
# 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 [6]:
```
# Depict the first 5 rows of the validation set
val_dataset = val_dataset.dropna()
val_dataset.head()
```

Out[6]:

	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

Now we will use the BatchIterator class to preprocess the text data and generate batches.

In [7]:
```
batch_size = 32
```

In [8]:
```
train_iterator = BatchIterator(train_dataset, batch_size=batch_size, v
                               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 [9]:
```
val_iterator = BatchIterator(val_dataset, batch_size=batch_size, vocab
                             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
```

Now we will check out if the batches look correctly.

In [10]:
```
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([32, 14])
target shape:  torch.Size([32])
x_lengths shape:  torch.Size([32])
```

In [11]:
```
# Extract the maximum sequence length

max_len = 0

for batches in train_iterator:
    x_lengths = batches['x_lengths']
    if max(x_lengths) > max_len:
        max_len = int(max(x_lengths))
```

In [12]:
```
print('Maximum sequence length: {}'.format(max_len))
```

```
Maximum sequence length: 60
```

Let's start implementing the Transformer model.

In [13]:
```
class MultiHeadAttention(nn.Module):
```

```
    """Implementation of the Multi-Head-Attention.

    Parameters
    ----------
    dmodel: int
        Dimensionality of the input embedding vector.
    heads: int
        Number of the self-attention operations to conduct in parallel
    """

    def __init__(self, dmodel, heads):

        super(MultiHeadAttention, self).__init__()

        assert dmodel % heads == 0, 'Embedding dimension is not divisi

        self.dmodel = dmodel
        self.heads = heads
        # Split dmodel (embedd dimension) into 'heads' number of chunk
        # each chunk of size key_dim will be passed to different atten
        self.key_dim = dmodel // heads

        # keys, queries and values will be computed at once for all he
        self.linear = nn.ModuleList([
            nn.Linear(self.dmodel, self.dmodel, bias=False),
            nn.Linear(self.dmodel, self.dmodel, bias=False),
            nn.Linear(self.dmodel, self.dmodel, bias=False)])

        self.concat = nn.Linear(self.dmodel, self.dmodel, bias=False)


    def forward(self, inputs):
        """ Perform Multi-Head-Attention.

        Parameters
        ----------
        inputs: torch.Tensor
            Batch of inputs - position encoded word embeddings ((batch

        Returns
        -------
        torch.Tensor
            Multi-Head-Attention output of a shape (batch_size, seq_le
        """

        self.batch_size = inputs.size(0)

        assert inputs.size(2) == self.dmodel, 'Input sizes mismatch, d
            .format(self.dmodel, inputs.size(2))
```

```
        # Inputs shape (batch_size, seq_length, embedding_dim)
        # Map input batch allong embedd dimension to query, key and va
        # a shape of (batch_size, heads, seq_len, key_dim (dmodel // h
        # where 'heads' dimension corresponds o different attention he
        query, key, value = [linear(x).view(self.batch_size, -1, self.
                             for linear, x in zip(self.linear, (inputs

        # Calculate the score (batch_size, heads, seq_len, seq_len)
        # for all heads at once
        score = torch.matmul(query, key.transpose(-2, -1)) / np.sqrt(s

        # Apply softmax to scores (batch_size, heads, seq_len, seq_len
        soft_score = F.softmax(score, dim = -1)

        # Multiply softmaxed score and value vector
        # value input shape (batch_size, heads, seq_len, key_dim)
        # out shape (batch_size, seq_len, dmodel (key_dim * heads))
        out = torch.matmul(soft_score, value).transpose(1, 2).contiguo
            .view(self.batch_size, -1, self.heads * self.key_dim)

        # Concatenate and linearly transform heads to the lower dimens
        # out shape (batch_size, seq_len, dmodel)
        out = self.concat(out)

        return out


class PositionalEncoding(nn.Module):
    """Implementation of the positional encoding.

    Parameters
    ----------
    max_len: int
        The maximum expected sequence length.
    dmodel: int
        Dimensionality of the input embedding vector.
    dropout: float
        Probability of an element of the tensor to be zeroed.
    padding_idx: int
        Index of the padding token in the vocabulary and word embeddin

    """

    def __init__(self, max_len, dmodel, dropout, padding_idx):

        super(PositionalEncoding, self).__init__()

        self.dropout = nn.Dropout(dropout)

        # Create pos_encoding, positions and dimensions matrices
        # with a shape of (max_len, dmodel)
```

```
        self.pos_encoding = torch.zeros(max_len, dmodel)
        positions = torch.repeat_interleave(torch.arange(float(max_ler
        dimensions = torch.arange(float(dmodel)).repeat(max_len, 1)

        # Calculate the encodings trigonometric function argument (max
        trig_fn_arg = positions / (torch.pow(10000, 2 * dimensions / c

        # Encode positions using sin function for even dimensions and
        # cos function for odd dimensions
        self.pos_encoding[:, 0::2] = torch.sin(trig_fn_arg[:, 0::2])
        self.pos_encoding[:, 1::2] = torch.cos(trig_fn_arg[:, 1::2])

        # Set the padding positional encoding to zero tensor
        if padding_idx:
            self.pos_encoding[padding_idx] = 0.0

        # Add batch dimension
        self.pos_encoding = self.pos_encoding.unsqueeze(0)


    def forward(self, embedd):
        """Apply positional encoding.

        Parameters
        ----------
        embedd: torch.Tensor
            Batch of word embeddings ((batch_size, seq_length, dmodel

        Returns
        -------
        torch.Tensor
            Sum of word embeddings and positional embeddings (batch_si
        """

        # embedd shape (batch_size, seq_length, embedding_dim)
        # pos_encoding shape (1, max_len, dmodel = embedd_dim)
        embedd = embedd + self.pos_encoding[:, :embedd.size(1), :]
        embedd = self.dropout(embedd)

        # embedd shape (batch_size, seq_length, embedding_dim)
        return embedd


class LabelSmoothingLoss(nn.Module):
    """Implementation of label smoothing with the Kullback-Leibler div

    Example:
    label_smoothing/(output_size-1) = 0.1
    confidence = 1 - 0.1 = 0.9
```

```
True labels      Smoothed one-hot labels
    |0|               [0.9000, 0.1000]
    |0|               [0.9000, 0.1000]
    |1|               [0.1000, 0.9000]
    |1|     label     [0.1000, 0.9000]
    |0|   smoothing   [0.9000, 0.1000]
    |1|     ---->     [0.1000, 0.9000]
    |0|               [0.9000, 0.1000]
    |0|               [0.9000, 0.1000]
    |0|               [0.9000, 0.1000]
    |1|               [0.1000, 0.9000]

Parameters
----------
output_size: int
     The number of classes.
label_smoothing: float, optional (default=0)
    The smoothing parameter. Takes the value in range [0,1].

"""

def __init__(self, output_size, label_smoothing=0):

    super(LabelSmoothingLoss, self).__init__()

    self.output_size = output_size
    self.label_smoothing = label_smoothing
    self.confidence = 1 - self.label_smoothing

    assert label_smoothing >= 0.0 and label_smoothing <= 1.0, \
    'Label smoothing parameter takes values in the range [0, 1]'

    self.criterion = nn.KLDivLoss()


def forward(self, pred, target):
    """Smooth the target labels and calculate the Kullback-Leibler

    Parameters
    ----------
    pred: torch.Tensor
        Batch of log-probabilities (batch_size, output_size)
    target: torch.Tensor
        Batch of target labels (batch_size, seq_length)

    Returns
    -------
    torch.Tensor
        The Kullback-Leibler divergence Loss.

    """
```

```
        # Create a Tensor of targets probabilities of a shape that equ
        # with label_smoothing/(output_size-1) value that will corresp
        one_hot_probs = torch.full(size=pred.size(), fill_value=self.l

        # Fill the tensor at positions that correspond to the true lab
        # with the modified value of maximum probability (confidence).
        one_hot_probs.scatter_(1, target.unsqueeze(1), self.confidence

        # KLDivLoss takes inputs (pred) that contain log-probs and tar
        return self.criterion(pred, one_hot_probs)
```

In [14]:
```
class TransformerBlock(nn.Module):
    """Implementation of single Transformer block.

    Transformer block structure:
    x --> Multi-Head --> Layer normalization --> Pos-Wise FFNN --> Lay
      |   Attention   |                        |                  |
      |_______________|                        |__________________|
     residual connection                       residual connection

    Parameters
    ----------
    dmodel: int
        Dimensionality of the input embedding vector.
    ffnn_hidden_size: int
        Position-Wise-Feed-Forward Neural Network hidden size.
    heads: int
        Number of the self-attention operations to conduct in parallel
    dropout: float
        Probability of an element of the tensor to be zeroed.
    """

    def __init__(self, dmodel, ffnn_hidden_size, heads, dropout):

        super(TransformerBlock, self).__init__()

        self.attention = MultiHeadAttention(dmodel, heads)
        self.layer_norm1 = nn.LayerNorm(dmodel)
        self.layer_norm2 = nn.LayerNorm(dmodel)

        self.ffnn = nn.Sequential(
                nn.Linear(dmodel, ffnn_hidden_size),
                nn.ReLU(),
                nn.Dropout(dropout),
                nn.Linear(ffnn_hidden_size, dmodel))


    def forward(self, inputs):
        """Forward propagate through the Transformer block
```

```
            Forward propagate through the Transformer block.

        Parameters
        ----------
        inputs: torch.Tensor
            Batch of embeddings.

        Returns
        -------
        torch.Tensor
            Output of the Transformer block (batch_size, seq_length, d
        """
        # Inputs shape (batch_size, seq_length, embedding_dim = dmodel
        output = inputs + self.attention(inputs)
        output = self.layer_norm1(output)
        output = output + self.ffnn(output)
        output = self.layer_norm2(output)

        # Output shape (batch_size, seq_length, dmodel)
        return output


class Transformer(nn.Module):
    """Implementation of the Transformer model for classification.

    Parameters
    ----------
    vocab_size: int
        The size of the vocabulary.
    dmodel: int
        Dimensionality of the embedding vector.
    max_len: int
        The maximum expected sequence length.
    padding_idx: int, optional (default=0)
        Index of the padding token in the vocabulary and word embeddin
    n_layers: int, optional (default=4)
        Number of the stacked Transformer blocks.
    ffnn_hidden_size: int, optonal (default=dmodel * 4)
        Position-Wise-Feed-Forward Neural Network hidden size.
    heads: int, optional (default=8)
        Number of the self-attention operations to conduct in parallel
    pooling: str, optional (default='max')
        Specify the type of pooling to use. Available options: 'max' o
    dropout: float, optional (default=0.2)
        Probability of an element of the tensor to be zeroed.
    """

    def __init__(self, vocab_size, dmodel, output_size, max_len, paddi
                 ffnn_hidden_size=None, heads=8, pooling='max', dropou

        super(Transformer, self).__init__()
```

```
        if not ffnn_hidden_size:
            ffnn_hidden_size = dmodel * 4

        assert pooling == 'max' or pooling == 'avg', 'Improper pooling

        self.pooling = pooling
        self.output_size = output_size

        self.embedding = nn.Embedding(vocab_size, dmodel)

        self.pos_encoding = PositionalEncoding(max_len, dmodel, dropou

        self.tnf_blocks = nn.ModuleList()

        for n in range(n_layers):
            self.tnf_blocks.append(
                TransformerBlock(dmodel, ffnn_hidden_size, heads, drop

        self.tnf_blocks = nn.Sequential(*self.tnf_blocks)

        self.linear = nn.Linear(dmodel, output_size)


    def forward(self, inputs, input_lengths):
        """Forward propagate through the Transformer.

        Parameters
        ----------
        inputs: torch.Tensor
            Batch of input sequences.
        input_lengths: torch.LongTensor
            Batch containing sequences lengths.

        Returns
        -------
        torch.Tensor
            Logarithm of softmaxed class tensor.
        """
        self.batch_size = inputs.size(0)

        # Input dimensions (batch_size, seq_length, dmodel)
        output = self.embedding(inputs)
        output = self.pos_encoding(output)
        output = self.tnf_blocks(output)
        # Output dimensions (batch_size, seq_length, dmodel)

        if self.pooling == 'max':
            # Permute to the shape (batch_size, dmodel, seq_length)
            # Apply max-pooling, output dimensions (batch_size, dmodel
            output = F.adaptive_max_pool1d(output.permute(0,2,1), (1,)
```

```
            output = F.adaptive_max_pool1d(output.permute(0,2,1), (1,)
        else:
            # Sum along the batch axis and divide by the corresponding
            # Output shape: (batch_size, dmodel)
            output = torch.sum(output, dim=1) / input_lengths.view(-1,

        output = self.linear(output)

        return F.log_softmax(output, 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, 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
```

In [15]:

```
# Initialize parameters
vocab_size = len(train_iterator.word2index)
dmodel = 64
output_size = 2
padding_idx = train_iterator.word2index['<PAD>']
n_layers = 4
ffnn_hidden_size = dmodel * 2
heads = 8
pooling = 'max'
dropout = 0.5
label_smoothing = 0.1
learning_rate = 0.001
epochs = 30

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

```
model = Transformer(vocab_size, dmodel, output_size, max_len, padding_
                    ffnn_hidden_size, heads, pooling, dropout)

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

# Add loss function
if label_smoothing:
    loss_fn = LabelSmoothingLoss(output_size, label_smoothing)
else:
    loss_fn = nn.NLLLoss()

model.add_loss_fn(loss_fn)

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)

# Create the parameters dictionary and instantiate the tensorboardX Su
params = {'batch_size': batch_size,
          'dmodel': dmodel,
          'n_layers': n_layers,
          'ffnn_hidden_size': ffnn_hidden_size,
          'heads': heads,
          'pooling': pooling,
          'dropout': dropout,
          'label_smoothing': label_smoothing,
          'learning_rate': learning_rate}

train_writer = SummaryWriter(comment=f' Training, batch_size={batch_si
ffnn_hidden_size={ffnn_hidden_size}, heads={heads}, pooling={pooling},
label_smoothing={label_smoothing}, learning_rate={learning_rate}'.form

val_writer = SummaryWriter(comment=f' Validation, batch_size={batch_si
ffnn_hidden_size={ffnn_hidden_size}, heads={heads}, pooling={pooling},
label_smoothing={label_smoothing}, learning_rate={learning_rate}'.form

# Instantiate the EarlyStopping
early_stop = EarlyStopping(wait_epochs=3)

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):

    try:
```

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

        train_losses, train_avg_loss, train_accuracy = model.train_moc

        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_

        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: {:
              .format(epoch+1, epochs, train_accuracy, train_avg_loss,

        train_writer.add_scalar('Training loss', train_avg_loss, epoch
        val_writer.add_scalar('Validation loss', eval_avg_loss, epoch)

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

    finally:
        train_writer.close()
        val_writer.close()
```

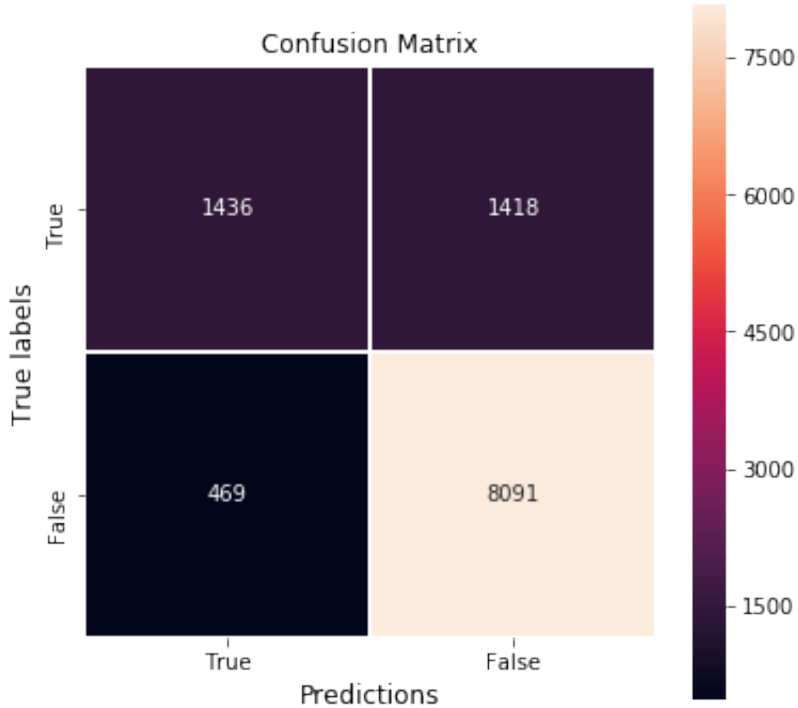
```
Epoch [16/30]: Train accuracy: 0.862. Train loss: 0.0780. Evaluatio
n accuracy: 0.836. Evaluation loss: 0.0906

Start epoch [17/30]
```

Training: 100% 1415/1415 [01:58<00:00, 11.87it/s]

```
Iteration: 100. Average training loss: 0.0796. Accuracy: 0.858
Iteration: 200. Average training loss: 0.0785. Accuracy: 0.859
Iteration: 300. Average training loss: 0.0804. Accuracy: 0.858
Iteration: 400. Average training loss: 0.0805. Accuracy: 0.858
Iteration: 500. Average training loss: 0.0745. Accuracy: 0.860
Iteration: 600. Average training loss: 0.0755. Accuracy: 0.861
Iteration: 700. Average training loss: 0.0771. Accuracy: 0.861
Iteration: 800. Average training loss: 0.0821. Accuracy: 0.861
Iteration: 900. Average training loss: 0.0795. Accuracy: 0.860
Iteration: 1000. Average training loss: 0.0789. Accuracy: 0.859
```

In [16]:
```
# 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 [17]:
```
# 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 [18]:
```
test_dataset = test_dataset.dropna()
test_dataset.head()
```

Out[18]:

	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 [20]:
```
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
Start creating glove_word2vector dictionary
Extracted 13890/17168 of pre-trained word vectors.
3278 vectors initialized to random numbers
Weights vectors saved into glove/weights_train.npy
Batches created
```

In [21]:
```
_, test_avg_loss, test_accuracy, test_conf_matrix = model.evaluate_moc
```

Evaluation: 100% 189/189 [00:38<00:00, 4.86it/s]

```
Iteration: 100. Average evaluation loss: 0.0836. Accuracy: 0.85
```

In [22]:
```
print('Test accuracy: {:.3f}. Test error: {:.3f}'.format(test_accuracy
```

```
Test accuracy: 0.852. Test error: 0.082
```

In [23]:
```
# 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()
```