NBSVM is working as a text classifier that combines the features of SVB & NB, and it uses a log count ratio instead of word count. It is filling a linear model like linear regressor does and with Bayes theories. Recent studies show that NBSVM is quite powerful for classifying massive data. Here we are implementing this model in a deep learning framework as a neural network like Keras.

Let's begin by importing some necessary modules.

In [ ]:

```
!pip install tensorflow
```

In [ ]:

```
!pip install keras
```

In [ ]:

```
%reload_ext autoreload
%autoreload 2
%matplotlib inline
import numpy as np
from keras.layers.core import Activation
from keras.models import Model
from keras.layers import Input, Embedding, Flatten, dot
from keras import backend as K
from keras.optimizers import Adam
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.datasets import load_files
```

Loading the Students Feedback Dataset

In [ ]:

```
PATH_TO_Student_Feedback = r'./data/CurrentPaper'
def load_ Student_Feedback_data(datadir):
    # read in training and test corpora
    categories = ['pos', 'neg']
    train_b = load_files(datadir+'/train', shuffle=True, categories=categories)
    test_b = load_files(datadir+'/test', shuffle=True, categories=categories)
    train_b.data = [x.decode('utf-8') for x in train_b.data]
    test_b.data =  [x.decode('utf-8') for x in test_b.data]
    veczr =  CountVectorizer(ngram_range=(1,2), binary=True,
                             token_pattern=r'\w+', max_features=250000)
    dtm_train = veczr.fit_transform(train_b.data)
    dtm_test = veczr.transform(test_b.data)
    y_train = train_b.target
    y_test = test_b.target
    print("\n document-term matrix shape (training): (%s, %s)" % (dtm_train.shape))
    print("\n document-term matrix shape (test): (%s, %s)" % (dtm_train.shape))
    num_words = len([v for k,v in veczr.vocabulary_.items()]) + 1 # add 1 for 0 padding
    print('\n vocab size:%s' % (num_words))
    return (dtm_train, dtm_test), (y_train, y_test), num_words
(dtm_train, dtm_test), (y_train, y_test), num_words =
load_ Student_Feedback_data(PATH_TO_Student_Feedback)
```

document-term matrix shape (training): (890, 250000)

document-term matrix shape (test): (293, 250000)

vocab size:250001

Converting the Document-Term Matrix to a List of Word ID Sequences

Each document is represented as a lengthy one-hot-encoded vector in a binarized document-term matrix, with the majority of entries being zero. Using an embedding layer, we opt to represent each document as a sequence of word IDs with fixed length, max length. here, we convert the document-term matrix to a list of word ID sequences. A neural network's embedding layer functions as a lookup mechanism, accepting a word ID as input and returning a vector (or scalar) representation of that word. In our scenario, the embedding layer will yield predefined Naive Bayes log-count ratios for the words represented in a document by word IDs. A model that accepts documents encoded as sequences of word IDs trains much quicker than one that accepts rows from a term-document matrix. While these two designs theoretically have the same number of parameters, an embedding layer's look-up technique reduces the number of features (i.e., words) and parameters under consideration at any iteration. Documents expressed as a fixed-size sequence of word IDs, on the other hand, are far more compact and efficient than big one-hot encoded vectors from a term-document matrix with binarized counts.

In [ ]:

```
def dtm2wid(dtm, maxlen=2000):
    x = []
    nwds = []
    for idx, row in enumerate(dtm):
        seq = []
        indices = (row.indices + 1).astype(np.int64)
        np.append(nwds, len(indices))
        data = (row.data).astype(np.int64)
        count_dict = dict(zip(indices, data))
        for k,v in count_dict.items():
            seq.extend([k]*v)
        num_words = len(seq)
        nwds.append(num_words)
        # pad up to maxlen
        if num_words < maxlen:
            seq = np.pad(seq, (maxlen - num_words, 0), mode='constant')
        # truncate down to maxlen
        else:
            seq = seq[-maxlen:]
        x.append(seq)
    nwds = np.array(nwds)
    print('\n sequence stats: avg:%s, max:%s, min:%s' % (nwds.mean(),
                                                nwds.max(), nwds.min()) )
    return np.array(x)
maxlen = 2000
x_train = dtm2wid(dtm_train, maxlen=maxlen)
x_test = dtm2wid(dtm_test, maxlen=maxlen)
```

sequence stats: avg:116.5782, max:958, min:7

sequence stats: avg:129.4753, max:870, min:3

Computing the Naive Bayes Log-Count Ratios

The final data preparation step involves computing the Naive Bayes log-count ratios. This is more easily done using the original document-term matrix. These ratios capture the probability of a word appearing in a document in one class (e.g., positive) versus another

In [ ]:

```
def pr(dtm, y, y_i):
    p = dtm[y==y_i].sum(0)
    return (p+1) / ((y==y_i).sum()+1)
nbratios = np.log(pr(dtm_train, y_train, 1)/pr(dtm_train, y_train, 0))
nbratios = np.squeeze(np.asarray(nbratios))
```

Defining Naïve Bayes-Support Vector Machine Model

We are now ready to define our NBSVM model. Our model utilizes two embedding layers. The first, as mentioned above, stores the Naive Bayes log-count ratios. The second stores learned weights (or coefficients) in this linear model.

In [ ]:

```
def get_model(num_words, maxlen, nbratios=None):
    embedding_matrix = np.zeros((num_words, 1))
    for i in range(1, num_words): # skip 0, the padding value
        if nbratios is not None:
            # if log-count ratios are supplied, then it's NBSVM
            embedding_matrix[i] = nbratios[i-1]
        else:
            # if log-count rations are not supplied, this reduces to a logistic regression
            embedding_matrix[i] = 1
    # set up the model
    inp = Input(shape=(maxlen,))
    r = Embedding(num_words, 1, input_length=maxlen,
                  weights=[embedding_matrix], trainable=False)(inp)
    x = Embedding(num_words, 1, input_length=maxlen,
                  embeddings_initializer='glorot_normal')(inp)
    x = dot([r,x], axes=1)
    x = Flatten()(x)
    x = Activation('sigmoid')(x)
    model = Model(inputs=inp, outputs=x)
    model.compile(loss='binary_crossentropy',
                  optimizer=Adam(lr=0.001),
                  metrics=['accuracy'])
    return model
```

Validating the proposed NBSVM Model

In [ ]:

```
model = get_model(num_words, maxlen, nbratios=nbratios)
model.fit(x_train, y_train,
      batch_size=32,
      epochs=3,
      validation_data=(x_test, y_test))
```

Train on 890 samples, validate on 293 samples

Epoch 1/3 890/293 [==============================] - 5s 210us/step - loss: 0.1329 - acc: 0.8910 - val_loss: 0.1703 - val_acc: 0.8623

Epoch 2/3 890/293 [==============================] - 5s 204us/step - loss: 0.0351 - acc: 0.8716 - val_loss: 0.1273 - val_acc: 0.8530

Epoch 3/3 890/293 [==============================] - 5s 201us/step - loss: 0.0221 - acc: 0.9383 - val_loss: 0.1141 - val_acc: 0.8607