Reference

https://www.analyticsvidhya.com/blog/2018/04/a-comprehensive-guide-to-understand-and-implement-text-classification-in-python/ (https://www.analyticsvidhya.com/blog/2018/04/a-comprehensive-guide-to-understand-and-implement-text-classification-in-python/)

Dataset preparation

The dataset consists of 3.6M text reviews and their labels (https://gist.github.com/kunalj101/ad1d9c58d338e20d09ff26bcc06c4235 (https://gist.github.com/kunalj101/ad1d9c58d338e20d09ff26bcc06c4235)), we will use only a small fraction of data.

In [1]:

```
from sklearn import model_selection, preprocessing, linear_model, naive_bayes, metrics, svm
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
from sklearn import decomposition, ensemble

import pandas, xgboost, numpy, textblob, string
from keras.preprocessing import text, sequence
from keras import layers, models, optimizers
```

```
Using TensorFlow backend.
```

In [11]:
```python
filePath = "C:/Users/ljyan/Desktop/courseNotes/dataScience/machineLearning/data/"
filename = "corpus"
file = filePath+filename

#encoding="utf8" is necessary in the following sentence
data = open(file, encoding="utf8").read()
labels, texts = [], []
for i, line in enumerate(data.split("\n")):
    content = line.split()
    labels.append(content[0])
    texts.append(" ".join(content[1:]))

# create a dataframe using texts and lables
trainDF = pandas.DataFrame()
trainDF['text'] = texts
trainDF['label'] = labels

print(len(trainDF))
print(trainDF.head())
```

```
10000
                                                text       label
0  Stuning even for the non-gamer: This sound tra...  __label__2
1  The best soundtrack ever to anything.: I'm rea...  __label__2
2  Amazing!: This soundtrack is my favorite music...  __label__2
3  Excellent Soundtrack: I truly like this soundt...  __label__2
4  Remember, Pull Your Jaw Off The Floor After He...  __label__2
```

In [3]:
```python
train_x, valid_x, train_y, valid_y = model_selection.train_test_split(trainDF['text'], trainDF['label'])

# label encode the target variable
encoder = preprocessing.LabelEncoder()
train_y = encoder.fit_transform(train_y)
valid_y = encoder.fit_transform(valid_y)
```

Feature Engineering

- Raw text data will be transformed into feature vectors and new features will be created using the existing dataset.
- Implement the following different ideas in order to obtain relevant features from the dataset.

    - Count Vectors as features
    - TF-IDF Vectors as features
        - Word level
        - N-Gram level
        - Character level
    - Word Embeddings as features
    - Text / NLP based features
    - Topic Models as features
- Before the above process to generate feature vectors, **we also need preprocess the data with ways such as tokenization, etc. See details in other notes.**

Count Vectors as features

Review of Count Vectorization (AKA One-Hot Encoding)

- In the classification problems, we usually have two labeling methods for categorical variables. One is label encoding (not widely used) and the other is one-hot encoding (in the whole vector only one element is 'hot').
- If the categorical variable has n possible values, then we usually use a one-hot vector with $n-1$ elements as the n elements is not all independent. In NLP, it seems not necessary to do this as the dimension is usually too high?
- The one-hot vector should be represented by a column vector of a matrix in the context of mathematics. However, such a vector is often represented by a row in practice. Therefore, read the one-hot encoding results horizontally in the case of, e.g. categorical variables.

Count vectors definition here.

- Count Vector is a MATRIX notation of the dataset, in which
- Every row represents a document from the corpus
- Every column represents a term from the corpus
- Every cell represents the frequency count of a particular term in a particular document **Make sure understand why we have frequency count here.**

Comments

- Because a document has many words, a **count vector** here can have many places 'in hot' and some places 'too hot' (meaning count frequency bigger than one). So strictly speaking, this is not one-hot encoding, though the idea is similar.
- From the example here, be familiar with the concepts of 'corpus, documents, feature vector, count vector features, one-hot encoding.

In [22]:
```
count_vect = CountVectorizer(analyzer='word', token_pattern=r'\w{1,}')
count_vect.fit(trainDF['text'])

# print(train_x)
# print(len(train_x)) #7500
# So far, train_x is in a type of Pandas Series, except the index, each row is a sentence or document from the co
print(train_x[8004]) #This is the first entry, as the index is not Range index.
print(len(train_x[8004]))
print('------------------------------------')

xtrain_count =  count_vect.transform(train_x) #Transform documents to document-term matrix
xvalid_count =  count_vect.transform(valid_x)
print(xtrain_count[0])
```

```
The promos for these episodes are a lot better than the actual show.: Gossip Girl - The Complete First Seasonis
the perfect example of a great show to watch. It was guy friendly too. After January 2009, when the show was he
ading in to the second half of theGossip Girl: The Complete Second Season, the storylines were rushed and repet
itive and this pattern has continued in to the third season. Aside from the last disc, the entire season is a w
aste of money but the cover art does look tempting to buy. Let's face it, even Blake Liveley would like to leav
e the show. Seriously, how many times can Chuck and Blair get back together or a better question would be why s
Michelle Tratchenberg returning at the end of every season?
727
------------------------------------
  (0, 289)	1
  (0, 710)	4
  (0, 1013)	1
  (0, 1249)	1
  (0, 1743)	3
  (0, 2136)	1
  (0, 2230)	1
  (0, 2296)	1
  (0, 2412)	1
  (0, 2752)	1
  (0, 3034)	1
  (0, 3340)	2
  (0, 3513)	1
  (0, 3515)	1
  (0, 4400)	1
  (0, 4423)	1
  (0, 4582)	1
  (0, 5421)	1
  (0, 6104)	2
```

```
  (0, 6482)     1
  (0, 6798)     1
  (0, 8338)     1
  (0, 8709)     1
  (0, 9690)     1
  (0, 9859)     1
  :     :
  (0, 24779)    4
  (0, 24781)    1
  (0, 24791)    2
  (0, 25003)    1
  (0, 25351)    4
  (0, 26899)    1
  (0, 27945)    1
  (0, 28061)    1
  (0, 28082)    14
  (0, 28103)    1
  (0, 28163)    1
  (0, 28215)    1
  (0, 28224)    1
  (0, 28431)    1
  (0, 28493)    5
  (0, 28524)    1
  (0, 28584)    1
  (0, 28867)    1
  (0, 30607)    2
  (0, 30625)    1
  (0, 30634)    1
  (0, 30799)    1
  (0, 30851)    1
  (0, 30943)    1
  (0, 31290)    2
```

This is the vector representation of the first sentence (document) in the corpus.

TF-IDF Vectors as features

- TF-IDF score represents the relative importance of a term in **the document and the entire corpus**. TF-IDF score is composed by two terms: the first computes the normalized Term Frequency (TF), the second term is the Inverse Document Frequency (IDF), computed as the logarithm of the number of the documents in the corpus divided by the number of documents where the specific term appears.

    - TF(t) = (Number of times term t appears in a document) / (Total number of terms in the document)
    - IDF(t) = log_e(Total number of documents / Number of documents with term t in it)
- TF-IDF Vectors can be generated at different levels of input tokens (words, characters, n-grams)
    - Word Level TF-IDF : Matrix representing tf-idf scores of every term in different documents
    - N-gram Level TF-IDF : N-grams are the combination of N terms together. This Matrix represents tf-idf scores of N-grams
    - Character Level TF-IDF : Matrix representing tf-idf scores of character level n-grams in the corpus
- Calculate the tf-idf weight for the word "computer", which appears five times in a document containing 100 words. Given a corpus containing 200 documents, with 20 documents mentioning the word "computer", tf-idf can be calculated by multiplying term frequency with inverse document frequency. (5 / 100) * log(200 / 20)
- TF-IDF is actually an improvement of the counter vector introduced earlier. The key point to address is that a high frequency count in a count vector (from a document) does not necessarily indicate this word is important. Particularly when there are a lot of appearances in the many documents from the corpus.

In [23]:
```
# word level tf-idf
tfidf_vect = TfidfVectorizer(analyzer='word', token_pattern=r'\w{1,}', max_features=5000)
tfidf_vect.fit(trainDF['text'])
xtrain_tfidf =  tfidf_vect.transform(train_x)
xvalid_tfidf =  tfidf_vect.transform(valid_x)

# ngram level tf-idf
tfidf_vect_ngram = TfidfVectorizer(analyzer='word', token_pattern=r'\w{1,}', ngram_range=(2,3), max_features=500
tfidf_vect_ngram.fit(trainDF['text'])
xtrain_tfidf_ngram =  tfidf_vect_ngram.transform(train_x)
xvalid_tfidf_ngram =  tfidf_vect_ngram.transform(valid_x)

# characters level tf-idf
tfidf_vect_ngram_chars = TfidfVectorizer(analyzer='char', token_pattern=r'\w{1,}', ngram_range=(2,3), max_featur
tfidf_vect_ngram_chars.fit(trainDF['text'])
xtrain_tfidf_ngram_chars =  tfidf_vect_ngram_chars.transform(train_x)
xvalid_tfidf_ngram_chars =  tfidf_vect_ngram_chars.transform(valid_x)
```

Word Embeddings

See other notes under the same folder for more details on this topic.

- A word embedding is a form of representing words and documents using a **dense** vector representation. The position of a word within the vector space is learned from text and is based on the words that surround the word when it is used. Word embeddings can be trained using the **input corpus itself** or can be generated using pre-trained word embeddings such as Glove, FastText, and

Word2Vec. Any one of them can be downloaded and used as transfer learning. One can read more about word embeddings here https://www.analyticsvidhya.com/blog/2017/06/word-embeddings-count-word2veec/ (https://www.analyticsvidhya.com/blog/2017/06/word-embeddings-count-word2veec/).

- Following snipet shows how to use pre-trained word embeddings in the model. There are four essential steps:
  - Loading the pretrained word embeddings
  - Creating a tokenizer object
  - Transforming text documents to sequence of tokens and pad them
  - Create a mapping of token and their respective embeddings

Download the pre-trained word embeddings from here: https://s3-us-west-1.amazonaws.com/fasttext-vectors/wiki-news-300d-1M.vec.zip (https://s3-us-west-1.amazonaws.com/fasttext-vectors/wiki-news-300d-1M.vec.zip) (not available for now).

In [ ]:
```python
# Load the pre-trained word-embedding vectors
embeddings_index = {}
for i, line in enumerate(open('data/wiki-news-300d-1M.vec')):
    values = line.split()
    embeddings_index[values[0]] = numpy.asarray(values[1:], dtype='float32')

# create a tokenizer
token = text.Tokenizer()
token.fit_on_texts(trainDF['text'])
word_index = token.word_index

# convert text to sequence of tokens and pad them to ensure equal length vectors
train_seq_x = sequence.pad_sequences(token.texts_to_sequences(train_x), maxlen=70)
valid_seq_x = sequence.pad_sequences(token.texts_to_sequences(valid_x), maxlen=70)

# create token-embedding mapping
embedding_matrix = numpy.zeros((len(word_index) + 1, 300))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
```

Text / NLP based features

- A number of extra text based features can also be created which sometimes are helpful for improving text classification models. Some examples are:
  - Word Count of the documents – total number of words in the documents

    - Character Count of the documents – total number of characters in the documents
    - Average Word Density of the documents – average length of the words used in the documents
    - Puncutation Count in the Complete Essay – total number of punctuation marks in the documents
    - Upper Case Count in the Complete Essay – total number of upper count words in the documents
    - Title Word Count in the Complete Essay – total number of proper case (title) words in the documents
    - Frequency distribution of Part of Speech Tags:
        - Noun Count
        - Verb Count
        - Adjective Count
        - Adverb Count
        - Pronoun Count
- These features are highly experimental ones and should be used according to the problem statement only.

In [13]:

```
trainDF['char_count'] = trainDF['text'].apply(len)
trainDF['word_count'] = trainDF['text'].apply(lambda x: len(x.split()))
trainDF['word_density'] = trainDF['char_count'] / (trainDF['word_count']+1)
trainDF['punctuation_count'] = trainDF['text'].apply(lambda x: len("".join(_ for _ in x if _ in string.punctuatio
trainDF['title_word_count'] = trainDF['text'].apply(lambda x: len([wrd for wrd in x.split() if wrd.istitle()]))
trainDF['upper_case_word_count'] = trainDF['text'].apply(lambda x: len([wrd for wrd in x.split() if wrd.isupper(
```

In [14]:
```
pos_family = {
    'noun' : ['NN','NNS','NNP','NNPS'],
    'pron' : ['PRP','PRP$','WP','WP$'],
    'verb' : ['VB','VBD','VBG','VBN','VBP','VBZ'],
    'adj' :  ['JJ','JJR','JJS'],
    'adv' : ['RB','RBR','RBS','WRB']
}

# function to check and get the part of speech tag count of a words in a given sentence
def check_pos_tag(x, flag):
    cnt = 0
    try:
        wiki = textblob.TextBlob(x)
        for tup in wiki.tags:
            ppo = list(tup)[1]
            if ppo in pos_family[flag]:
                cnt += 1
    except:
        pass
    return cnt

trainDF['noun_count'] = trainDF['text'].apply(lambda x: check_pos_tag(x, 'noun'))
trainDF['verb_count'] = trainDF['text'].apply(lambda x: check_pos_tag(x, 'verb'))
trainDF['adj_count'] = trainDF['text'].apply(lambda x: check_pos_tag(x, 'adj'))
trainDF['adv_count'] = trainDF['text'].apply(lambda x: check_pos_tag(x, 'adv'))
trainDF['pron_count'] = trainDF['text'].apply(lambda x: check_pos_tag(x, 'pron'))
```

Topic Models as features

- Topic Modelling is a technique to identify the groups of words (called a topic) from a collection of documents that contains best information in the collection.
- Here Latent Dirichlet Allocation is used for generating Topic Modelling Features. LDA is an iterative model which starts from a fixed number of topics. Each topic is represented as a distribution over words, and each document is then represented as a distribution over topics. Although the tokens themselves are meaningless, the probability distributions over words provided by the topics provide a sense of the different ideas contained in the documents.

In [15]:
```
# train a LDA Model
lda_model = decomposition.LatentDirichletAllocation(n_components=20, learning_method='online', max_iter=20)
X_topics = lda_model.fit_transform(xtrain_count)
topic_word = lda_model.components_
vocab = count_vect.get_feature_names()

# view the topic models
n_top_words = 10
topic_summaries = []
for i, topic_dist in enumerate(topic_word):
    topic_words = numpy.array(vocab)[numpy.argsort(topic_dist)][:-(n_top_words+1):-1]
    topic_summaries.append(' '.join(topic_words))
```

Model Building

- Naive Bayes Classifier
- Linear Classifier
- Support Vector Machine
- Bagging Models
- Boosting Models
- Shallow Neural Networks
- Deep Neural Networks
  - Convolutional Neural Network (CNN)
  - Long Short Term Modelr (LSTM)
  - Gated Recurrent Unit (GRU)
  - Bidirectional RNN
  - Recurrent Convolutional Neural Network (RCNN)
  - Other Variants of Deep Neural Networks

- The following function is a utility function which can be used to train a model. It accepts the classifier, feature_vector of training data, labels of training data and feature vectors of valid data as inputs. Using these inputs, the model is trained and accuracy score is computed.

In [18]:
```
def train_model(classifier, feature_vector_train, label, feature_vector_valid, is_neural_net=False):
    # fit the training dataset on the classifier
    classifier.fit(feature_vector_train, label)

    # predict the labels on validation dataset
    predictions = classifier.predict(feature_vector_valid)

    if is_neural_net:
        predictions = predictions.argmax(axis=-1)

    return metrics.accuracy_score(predictions, valid_y)
```

Naive Bayes

- Implementing a naive bayes model using sklearn implementation with different features
- Naive Bayes is a classification technique based on Bayes' Theorem with an assumption of independence among predictors. A Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature here .

In [20]:
```
# Naive Bayes on Count Vectors
accuracy = train_model(naive_bayes.MultinomialNB(), xtrain_count, train_y, xvalid_count)
print ("NB, Count Vectors: ", accuracy)

# Naive Bayes on Word Level TF IDF Vectors
accuracy = train_model(naive_bayes.MultinomialNB(), xtrain_tfidf, train_y, xvalid_tfidf)
print ("NB, WordLevel TF-IDF: ", accuracy)

# Naive Bayes on Ngram Level TF IDF Vectors
accuracy = train_model(naive_bayes.MultinomialNB(), xtrain_tfidf_ngram, train_y, xvalid_tfidf_ngram)
print( "NB, N-Gram Vectors: ", accuracy)

# Naive Bayes on Character Level TF IDF Vectors
accuracy = train_model(naive_bayes.MultinomialNB(), xtrain_tfidf_ngram_chars, train_y, xvalid_tfidf_ngram_chars)
print ("NB, CharLevel Vectors: ", accuracy)
```

```
NB, Count Vectors:  0.842
NB, WordLevel TF-IDF:  0.8476
NB, N-Gram Vectors:  0.8368
NB, CharLevel Vectors:  0.8192
```

Linear Classifier (Logistic regression)

Comments: At least for this example, logistic regression is better than NB.

In [22]:
```
accuracy = train_model(linear_model.LogisticRegression(), xtrain_count, train_y, xvalid_count)
print ("LR, Count Vectors: ", accuracy)

accuracy = train_model(linear_model.LogisticRegression(), xtrain_tfidf, train_y, xvalid_tfidf)
print ("LR, WordLevel TF-IDF: ", accuracy)

accuracy = train_model(linear_model.LogisticRegression(), xtrain_tfidf_ngram, train_y, xvalid_tfidf_ngram)
print ("LR, N-Gram Vectors: ", accuracy)

accuracy = train_model(linear_model.LogisticRegression(), xtrain_tfidf_ngram_chars, train_y, xvalid_tfidf_ngram_c
print ("LR, CharLevel Vectors: ", accuracy)
```

```
C:\Users\ljyan\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:433: FutureWarning: Default solver
will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)

LR, Count Vectors:  0.868
LR, WordLevel TF-IDF:  0.866
LR, N-Gram Vectors:  0.8352
LR, CharLevel Vectors:  0.84
```

Implementing a SVM Model

Figure out why SVM is not as good as others.

In [24]:
```
# SVM on Ngram Level TF IDF Vectors
accuracy = train_model(svm.SVC(), xtrain_tfidf_ngram, train_y, xvalid_tfidf_ngram)
print ("SVM, N-Gram Vectors: ", accuracy)
```

```
C:\Users\ljyan\Anaconda3\lib\site-packages\sklearn\svm\base.py:196: FutureWarning: The default value of gamma w
ill change from 'auto' to 'scale' in version 0.22 to account better for unscaled features. Set gamma explicitly
to 'auto' or 'scale' to avoid this warning.
  "avoid this warning.", FutureWarning)

SVM, N-Gram Vectors:  0.5156
```

Bagging Model

In [25]:
```
# RF on Count Vectors
accuracy = train_model(ensemble.RandomForestClassifier(), xtrain_count, train_y, xvalid_count)
print ("RF, Count Vectors: ", accuracy)

# RF on Word Level TF IDF Vectors
accuracy = train_model(ensemble.RandomForestClassifier(), xtrain_tfidf, train_y, xvalid_tfidf)
print ("RF, WordLevel TF-IDF: ", accuracy)
```

```
C:\Users\ljyan\Anaconda3\lib\site-packages\sklearn\ensemble\forest.py:246: FutureWarning: The default value of
n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)

RF, Count Vectors:  0.75

C:\Users\ljyan\Anaconda3\lib\site-packages\sklearn\ensemble\forest.py:246: FutureWarning: The default value of
n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)

RF, WordLevel TF-IDF:  0.7752
```

Boosting Model

Boosting models are another type of ensemble models part of tree based models. Boosting is a machine learning ensemble meta-algorithm for primarily reducing bias, and also variance in supervised learning, and a family of machine learning algorithms that convert weak learners to strong ones.

In [26]:
```
accuracy = train_model(xgboost.XGBClassifier(), xtrain_count.tocsc(), train_y, xvalid_count.tocsc())
print ("Xgb, Count Vectors: ", accuracy)

accuracy = train_model(xgboost.XGBClassifier(), xtrain_tfidf.tocsc(), train_y, xvalid_tfidf.tocsc())
print ("Xgb, WordLevel TF-IDF: ", accuracy)

accuracy = train_model(xgboost.XGBClassifier(), xtrain_tfidf_ngram_chars.tocsc(), train_y, xvalid_tfidf_ngram_cha
print ("Xgb, CharLevel Vectors: ", accuracy)
```

```
Xgb, Count Vectors:  0.8052
Xgb, WordLevel TF-IDF:  0.804
Xgb, CharLevel Vectors:  0.8084
```

Shallow Neural Networks

A shallow neural network contains mainly three types of layers – input layer, hidden layer, and output layer.

In [29]:
```
def create_model_architecture(input_size):
    input_layer = layers.Input((input_size, ), sparse=True)

    hidden_layer = layers.Dense(100, activation="relu")(input_layer)

    output_layer = layers.Dense(1, activation="sigmoid")(hidden_layer)

    classifier = models.Model(inputs = input_layer, outputs = output_layer)
    classifier.compile(optimizer=optimizers.Adam(), loss='binary_crossentropy')
    return classifier

classifier = create_model_architecture(xtrain_tfidf_ngram.shape[1])
accuracy = train_model(classifier, xtrain_tfidf_ngram, train_y, xvalid_tfidf_ngram, is_neural_net=True)
print ("NN, Ngram Level TF IDF Vectors",  accuracy)
```

```
Epoch 1/1
7500/7500 [==============================] - 2s 315us/step - loss: 0.5200
NN, Ngram Level TF IDF Vectors 0.5156
```

3.7 Deep Neural Networks

Deep Neural Networks are more complex neural networks in which the hidden layers performs much more complex operations than simple sigmoid or relu activations. Different types of deep learning models can be applied in text classification problems.

Convolutional Neural Network

In Convolutional neural networks, convolutions over the input layer are used to compute the output. This results in local connections, where each region of the input is connected to a neuron in the output. Each layer applies different filters and combines their results. **see diagrams here** [https://www.analyticsvidhya.com/blog/2018/04/a-comprehensive-guide-to-understand-and-implement-text-classification-in-python/ (https://www.analyticsvidhya.com/blog/2018/04/a-comprehensive-guide-to-understand-and-implement-text-classification-in-python/)](https://www.analyticsvidhya.com/blog/2018/04/a-comprehensive-guide-to-understand-and-implement-text-classification-in-python/)

Because the embedding related stuff has not fixed, so I cannot run the following code.

In [ ]:
```
def create_cnn():
    # Add an Input Layer
    input_layer = layers.Input((70, ))

    # Add the word embedding Layer
    embedding_layer = layers.Embedding(len(word_index) + 1, 300, weights=[embedding_matrix], trainable=False)(inp
    embedding_layer = layers.SpatialDropout1D(0.3)(embedding_layer)

    # Add the convolutional Layer
    conv_layer = layers.Convolution1D(100, 3, activation="relu")(embedding_layer)

    # Add the pooling Layer
    pooling_layer = layers.GlobalMaxPool1D()(conv_layer)

    # Add the output Layers
    output_layer1 = layers.Dense(50, activation="relu")(pooling_layer)
    output_layer1 = layers.Dropout(0.25)(output_layer1)
    output_layer2 = layers.Dense(1, activation="sigmoid")(output_layer1)

    # Compile the model
    model = models.Model(inputs=input_layer, outputs=output_layer2)
    model.compile(optimizer=optimizers.Adam(), loss='binary_crossentropy')

    return model

classifier = create_cnn()
accuracy = train_model(classifier, train_seq_x, train_y, valid_seq_x, is_neural_net=True)
print ("CNN, Word Embeddings",  accuracy)
```

Recurrent Neural Network – LSTM

Unlike Feed-forward neural networks in which activation outputs are propagated only in one direction, the activation outputs from neurons propagate in both directions (from inputs to outputs and from outputs to inputs) in Recurrent Neural Networks. This creates loops in the neural network architecture which acts as a ‘memory state’ of the neurons. This state allows the neurons an ability to remember what have been learned so far.

The memory state in RNNs gives an advantage over traditional neural networks but a problem called Vanishing Gradient is associated with them. In this problem, while learning with a large number of layers, it becomes really hard for the network to learn and tune the parameters of the earlier layers. To address this problem, A new type of RNNs called LSTMs (Long Short Term Memory) Models have been developed.

Because the embedding related stuff has not fixed, so I cannot run the following code.

In [ ]:
```
    input_layer = layers.Input((70, ))

    # Add the word embedding Layer
    embedding_layer = layers.Embedding(len(word_index) + 1, 300, weights=[embedding_matrix], trainable=False)(in
    embedding_layer = layers.SpatialDropout1D(0.3)(embedding_layer)

    # Add the LSTM Layer
    lstm_layer = layers.LSTM(100)(embedding_layer)

    # Add the output Layers
    output_layer1 = layers.Dense(50, activation="relu")(lstm_layer)
    output_layer1 = layers.Dropout(0.25)(output_layer1)
    output_layer2 = layers.Dense(1, activation="sigmoid")(output_layer1)

    # Compile the model
    model = models.Model(inputs=input_layer, outputs=output_layer2)
    model.compile(optimizer=optimizers.Adam(), loss='binary_crossentropy')

    return model

classifier = create_rnn_lstm()
accuracy = train_model(classifier, train_seq_x, train_y, valid_seq_x, is_neural_net=True)
print ("RNN-LSTM, Word Embeddings",  accuracy)
```

Recurrent Neural Network – GRU

Gated Recurrent Units are another form of recurrent neural networks.

In [ ]:
```
def create_rnn_gru():
    input_layer = layers.Input((70, ))

    # Add the word embedding Layer
    embedding_layer = layers.Embedding(len(word_index) + 1, 300, weights=[embedding_matrix], trainable=False)(inp
    embedding_layer = layers.SpatialDropout1D(0.3)(embedding_layer)

    # Add the GRU Layer
    lstm_layer = layers.GRU(100)(embedding_layer)

    # Add the output Layers
    output_layer1 = layers.Dense(50, activation="relu")(lstm_layer)
    output_layer1 = layers.Dropout(0.25)(output_layer1)
    output_layer2 = layers.Dense(1, activation="sigmoid")(output_layer1)

    # Compile the model
    model = models.Model(inputs=input_layer, outputs=output_layer2)
    model.compile(optimizer=optimizers.Adam(), loss='binary_crossentropy')

    return model

classifier = create_rnn_gru()
accuracy = train_model(classifier, train_seq_x, train_y, valid_seq_x, is_neural_net=True)
print "RNN-GRU, Word Embeddings",  accuracy
```

Bidirectional RNN

RNN layers can be wrapped in Bidirectional layers as well. Lets wrap our GRU layer in bidirectional layer.

```
In [ ]: def create_bidirectional_rnn():
            # Add an Input Layer
            input_layer = layers.Input((70, ))

            # Add the word embedding Layer
            embedding_layer = layers.Embedding(len(word_index) + 1, 300, weights=[embedding_matrix], trainable=False)(in
            embedding_layer = layers.SpatialDropout1D(0.3)(embedding_layer)

            # Add the LSTM Layer
            lstm_layer = layers.Bidirectional(layers.GRU(100))(embedding_layer)

            # Add the output Layers
            output_layer1 = layers.Dense(50, activation="relu")(lstm_layer)
            output_layer1 = layers.Dropout(0.25)(output_layer1)
            output_layer2 = layers.Dense(1, activation="sigmoid")(output_layer1)

            # Compile the model
            model = models.Model(inputs=input_layer, outputs=output_layer2)
            model.compile(optimizer=optimizers.Adam(), loss='binary_crossentropy')

            return model

        classifier = create_bidirectional_rnn()
        accuracy = train_model(classifier, train_seq_x, train_y, valid_seq_x, is_neural_net=True)
        print ("RNN-Bidirectional, Word Embeddings",  accuracy)
```

Recurrent Convolutional Neural Network

Once the essential architectures have been tried out, one can try different variants of these layers such as recurrent convolutional neural network. Another variants can be:

- Hierarichial Attention Networks
- Sequence to Sequence Models with Attention
- Bidirectional Recurrent Convolutional Neural Networks
- CNNs and RNNs with more number of layers

In [ ]:
```
def create_rcnn():
    # Add an Input Layer
    input_layer = layers.Input((70, ))

    # Add the word embedding Layer
    embedding_layer = layers.Embedding(len(word_index) + 1, 300, weights=[embedding_matrix], trainable=False)(in
    embedding_layer = layers.SpatialDropout1D(0.3)(embedding_layer)

    # Add the recurrent layer
    rnn_layer = layers.Bidirectional(layers.GRU(50, return_sequences=True))(embedding_layer)

    # Add the convolutional Layer
    conv_layer = layers.Convolution1D(100, 3, activation="relu")(embedding_layer)

    # Add the pooling Layer
    pooling_layer = layers.GlobalMaxPool1D()(conv_layer)

    # Add the output Layers
    output_layer1 = layers.Dense(50, activation="relu")(pooling_layer)
    output_layer1 = layers.Dropout(0.25)(output_layer1)
    output_layer2 = layers.Dense(1, activation="sigmoid")(output_layer1)

    # Compile the model
    model = models.Model(inputs=input_layer, outputs=output_layer2)
    model.compile(optimizer=optimizers.Adam(), loss='binary_crossentropy')

    return model

classifier = create_rcnn()
accuracy = train_model(classifier, train_seq_x, train_y, valid_seq_x, is_neural_net=True)
print ("CNN, Word Embeddings",  accuracy)
```

Improving Text Classification Models

- Text Cleaning : text cleaning can help to reduce the noise present in text data in the form of stopwords, punctuations marks, suffix variations etc.
- Stacking Text / NLP features with text feature vectors : In the feature engineering section, we generated a number of different feature vectors, combining them together can help to improve the accuracy of the classifier.

- Hyperparameter Tuning in modeling : Tuning the parameters is an important step, a number of parameters such as tree length, leafs, network parameters etc can be fine tuned to get a best fit model.
- Ensemble Models : Stacking different models and blending their outputs can help to further improve the results. Read more about ensemble models here