Apêndice 2 - Notebook classificação de texto

1 Classificação de Texto do domínio de óleo e gás

O objetivo deste notebook é fazer um estudo sobre a classificação de textos. Serão usadas diversas técnicas para criar um modelo que classifique resumos de teses de doutorado e dissertação de mestrado. Serão usados documentos elaborados por técnicos da Petrobras, e da Biblioteca Digital de Teses e Dissertações. Esperamos que os modelos classifiquem corretamente os documentos nos seus respectivos domínios.

Baseado no post de Shivam Bansa
Shivam Bansal. **A Comprehensive Guide to Understand and ImplementText Classification in Python.** Analytics Vidhya. 23 de abr. de 2018.url:https : / / www . analyticsvidhya . com / blog / 2018 / 04 / a -comprehensive - guide - to - understand - and - implement - text -classification-in-python/(acesso em 14/08/2019)

[1]:
```
# Importando bibliotecas
import warnings
warnings.filterwarnings("ignore")
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
from random import shuffle
import pandas as pd
import numpy as np
import xgboost, textblob, string
from keras.preprocessing import text, sequence
import tensorflow as tf
from keras.models import Sequential
from keras.utils import to_categorical
from keras import layers, models, optimizers
from keras.callbacks import EarlyStopping
from keras.layers import Concatenate
from keras.layers import Layer
from keras.layers import Flatten
import tensorflow as tf
from keras.initializers import get
from bs4 import BeautifulSoup as bs
import nltk
from  nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
```

```python
import matplotlib.pyplot as plt
import gensim
from gensim.models import Word2Vec
from langdetect import detect
from langdetect import detect_langs
```

```
Using TensorFlow backend.
```

2 Preparando os dados

Lendo arquivos JSON com os dados das teses Petrobras no BDTD, das teses no BDTD com assunto “Petroleo” e teses de assunto opostos ao de interesse Petrobras (“Linguas, Letras e Artes”, “Arqueologia”, “Demografia”, . . .)

[2]:
```python
teses_Subject_petroleo = pd.read_json('BDTD/New_Subject_petroleo.json', orient =␣
 ↪'index')
teses_petrobras_BDTD = pd.read_json('Petrobras/New_teses_petrobras_BDTD.json',␣
 ↪orient = 'index')
tese_mesma_area_Large = pd.read_json('BDTD/teses_mesmas_areas_Large.json',␣
 ↪orient = 'index')
teses_areas_opostas_Large = pd.read_json('BDTD/teses_areas_opostas_Large.json',␣
 ↪orient = 'index')
```

[3]:
```python
# Número total de documentos
len(tese_mesma_area_Large) + len(teses_areas_opostas_Large)
```

[3]:
```
11532
```

[4]:
```python
# Unindo as teses de Petróleo
teses_petroleo = teses_Subject_petroleo
teses_petroleo = teses_petroleo.append(teses_petrobras_BDTD)
# Excluindo teses duplicadas
teses_petroleo = teses_petroleo[~teses_petroleo.index.duplicated(keep='first')]
```

[5]:
```python
# Unindo as teses de todas as áreas
teses_areas = tese_mesma_area_Large
teses_areas = teses_areas.append(teses_areas_opostas_Large)
# Excluindo teses duplicadas
teses_areas = teses_areas[~teses_areas.index.duplicated(keep='first')]
```

Acrescentando a classe nos dois DataFrame

[6]:
```python
teses_petroleo['classe'] = 'Petroleo'
teses_areas ['classe'] = 'Todas Areas'
```

Verificando a existência de teses duplicadas nas duas classes

[7]:
```python
# Unindo as duas classes de documentos
todos = teses_petroleo
todos = todos.append(teses_areas)
len(todos)
```

[7]: 13885

[16]:
```python
# Verificando a existência de documentos doplicados
todos = todos[~todos.index.duplicated(keep='first')]
len(todos)
```

[16]: 13845

[8]:
```python
#SEparando novamente
teses_petroleo = todos[todos['classe'] == 'Petroleo']
teses_areas = todos[todos['classe'] == 'Todas Areas']
```

Verificando balanceamento das duas classes

[9]:
```python
print ('teses_petroleo: ', len(teses_petroleo))
print ('teses_areas: ', len(teses_areas))
```

```
teses_petroleo:  2366
teses_areas:  11519
```

Verificando os campos de resumo

[10]:
```python
# Excluindo documentos sem resumo em português
teses_petroleo = teses_petroleo[(teses_petroleo['Resumo Português:'].notnull())]
teses_areas = teses_areas[(teses_areas['Resumo Português:'].notnull())]
```

[11]:
```python
# Função que recebe um texto e separa a parte português da parte em inglês
def separacao_port_engl(abstract):

    # Tokeniza os resumos em sentenças
    mix_sent = nltk.sent_tokenize(abstract)

    # Algumas sentenças vem unidas sem espaço.
    # Portanto é necessário encontra o ponto final para quebrar a sentença em␣
 ↪duas.
    new_mix = []
    for sent in mix_sent:
        position = sent.find('.')
        if position != len(sent)-1:
            sent_1 = sent[:position+1]
            sent_2 = sent[position+1:]
            new_mix.append(sent_1)
            new_mix.append(sent_2)
        else:
            new_mix.append(sent)

    mix_sent = new_mix

    # Para cada sentença, identificar se ela está em português ou inglês
    port = []
    engl = []
```

```
    for sent in mix_sent:
        try:
            if detect (sent) == 'pt':
                port.append(sent)
            else:
                engl.append (sent)
        except:
            pass

    # As sentenças são unidas novamente
    port = " ".join(port)
    engl = " ".join(engl)

    # A função retorna os resumos em cada idioma
    return(port, engl)
```

[12]:
```
# Separando português e inglês para teses petróleo
columns_pt = teses_petroleo['Resumo Português:'].apply(lambda x:␣
 ↪separacao_port_engl(x)[0])
columns_en = teses_petroleo['Resumo Português:'].apply(lambda x:␣
 ↪separacao_port_engl(x)[1])
teses_petroleo['Resumo Português:'] = columns_pt
teses_petroleo['Resumo inglês 2:'] = columns_en

# Separando português e inglês para teses das demais áreas
columns_pt = teses_areas['Resumo Português:'].apply(lambda x:␣
 ↪separacao_port_engl(x)[0])
columns_en = teses_areas['Resumo Português:'].apply(lambda x:␣
 ↪separacao_port_engl(x)[1])
teses_areas['Resumo Português:'] = columns_pt
teses_areas['Resumo inglês 2:'] = columns_en
```

Excluindo novamente as teses sem resumo em portugues

[13]:
```
teses_petroleo = teses_petroleo[(teses_petroleo['Resumo Português:'].notnull())]
teses_areas = teses_areas[(teses_areas['Resumo Português:'].notnull())]
```

Preprocessando o texto e retirando stopwords

[14]:
```
# Letras em minúsculas
teses_petroleo['Resumo Português:'] = teses_petroleo['Resumo Português:'].str.
 ↪lower()
teses_areas['Resumo Português:'] = teses_areas['Resumo Português:'].str.lower()
```

[16]:
```
# Preprocessando os textos
teses_petroleo['Resumo Português:'] = (teses_petroleo['Resumo Português:']
                                        .apply(gensim.utils.simple_preprocess)
                                        .str.join(" "))
teses_areas['Resumo Português:'] = (teses_areas['Resumo Português:']
                                     .apply(gensim.utils.simple_preprocess)
```

```
                                            .str.join(" "))
```

[17]:
```
# Importando as bibliotecas de stopwords
nltk.download('stopwords')

# Mapeando stopwords com NLTK
stopwordsIngles = stopwords.words("portuguese")

def remove_stopwords(abstract):
    without_stopwords = []
    for word in abstract:
        if word not in stopwordsIngles:
            without_stopwords.append(word)
    return(without_stopwords)

# Excluindo stopwords
teses_petroleo['Resumo Português:'] = teses_petroleo['Resumo Português:'].
 ↪apply(remove_stopwords)
teses_areas['Resumo Português:'] = teses_areas['Resumo Português:'].
 ↪apply(remove_stopwords)

# Unindo novamente o texto em uma única string
teses_petroleo['Resumo Português:'] = teses_petroleo['Resumo Português:'].str.
 ↪join(" ")
teses_areas['Resumo Português:'] = teses_areas['Resumo Português:'].str.join(" ")
```

[2]:
```
# Gravando textos preprocessados em um arquivo JSON
#teses_petroleo.to_json('BDTD/tese_petroleo_processada.json', orient = 'index')
#teses_areas.to_json('BDTD/tese_areas_processada.json', orient = 'index')
# lendo textos preprocessados de arquivos JSON
teses_petroleo = pd.read_json('BDTD/tese_petroleo_processada.json', orient =␣
 ↪'index')
teses_areas = pd.read_json('BDTD/tese_areas_processada.json', orient = 'index')
```

2.1 Dividindo o conjunto de treino, validação e de teste

Vamos dividir os dados em 80% treino e 20% teste

[3]:
```
#Função que recebe um dataframe com as teses e retorna dois dataframes com dados␣
 ↪de treino e teste.
#'train' é a fração dos dados para treino, o restante é para teste
def train_test(teses, train):
    corte_train = int(round((len(teses)*train),0))
    teses = teses.sample(frac=1)
    teses_train = teses[:corte_train]
    teses_test = teses[corte_train:]
    return(teses_train, teses_test)
```

[4]:
```python
# Os documentos são balanceados para ficarem com a mesma quantidades
teses_areas = teses_areas.sample(len(teses_petroleo))
```

[5]:
```python
print(len(teses_areas))
print(len(teses_petroleo))
```

```
2121
2121
```

[6]:
```python
# São separadas as frações para treino e teste
teses_petroleo_train, teses_petroleo_test = train_test(teses_petroleo, 0.8)
teses_areas_train, teses_areas_test = train_test(teses_areas, 0.8)
```

[7]:
```python
# Os dados de treino e teste são unindos e embaralhados
# Train
tese_train = teses_petroleo_train
tese_train = tese_train.append(teses_areas_train)
tese_train = tese_train.sample(frac=1).reset_index(drop=True)

#Test
tese_test = teses_petroleo_test
tese_test = tese_test.append(teses_areas_test)
tese_test = tese_test.sample(frac=1).reset_index(drop=True)
```

[8]:
```python
# Separando apenas os texto e classes para treinar os classificadores
train_x = tese_train['Resumo Português:']
train_y = tese_train['classe']

test_x = tese_test['Resumo Português:']
test_y = tese_test['classe']
```

[9]:
```python
# Codigficando as classes para as variáveis 0 e 1
encoder = preprocessing.LabelEncoder()
train_y = encoder.fit_transform(train_y)
test_y  = encoder.transform(test_y)
```

[10]:
```python
print ('Petrobras = ', encoder.transform(['Petroleo'])[0])
print ('Outro = ', encoder.transform(['Todas Areas'])[0])
```

```
Petrobras =  0
Outro =  1
```

3 Feature Engineering

O próximo passo é criar os atributos dos textos. Nesta etapa o texto bruto será transformado em vetores e novos atributos serão criados a partir dos dados atuais.

Count Vectors

Count Vector é uma notação de matriz, onde cada linha representa um documento, cada coluna representa um termo do corpus, e cada celula representa a frequência de um determinado termo em um documento em particular.

[11]:
```
# Criando um objeto Count Vector
count_vect = CountVectorizer(analyzer='word', token_pattern=r'\w{1,}')
count_vect.fit(tese_train['Resumo Português:'])

# Trasnsforma os dados de treino e teste usando o objeto Count Vector
xtrain_count =  count_vect.transform(train_x)
xtest_count =  count_vect.transform(test_x)
```

TF-IDF Vectors

TF-IDF score representa a importância relativa dos termos em um documento e no corpus inteiro. TF-IDF score é composto por:

TF(t) = (Número de vezes que o termo t aparece em um documento) / (Número total de termos em um documento) IDF(t) = log_e(Número total de documentos / Número de documentos que contém o termo t)

Os vetores TF-IDF podem ser gerados com diferentes níveis de tokens (palavrass, ccaracteres, n-grams)

[12]:
```
# word level tf-idf
tfidf_vect = TfidfVectorizer(analyzer='word', token_pattern=r'\w{1,}',␣
↪max_features=5000)
tfidf_vect.fit(tese_train['Resumo Português:'])
xtrain_tfidf =  tfidf_vect.transform(train_x)
xtest_tfidf =  tfidf_vect.transform(test_x)

# ngram level tf-idf
tfidf_vect_ngram = TfidfVectorizer(analyzer='word', token_pattern=r'\w{1,}',␣
↪ngram_range=(2,3), max_features=5000)
tfidf_vect_ngram.fit(tese_train['Resumo Português:'])
xtrain_tfidf_ngram =  tfidf_vect_ngram.transform(train_x)
xtest_tfidf_ngram =  tfidf_vect_ngram.transform(test_x)

# characters level tf-idf
tfidf_vect_ngram_chars = TfidfVectorizer(analyzer='char',␣
↪token_pattern=r'\w{1,}', ngram_range=(2,3), max_features=5000)
tfidf_vect_ngram_chars.fit(tese_train['Resumo Português:'])
xtrain_tfidf_ngram_chars =  tfidf_vect_ngram_chars.transform(train_x)
xtest_tfidf_ngram_chars =  tfidf_vect_ngram_chars.transform(test_x)
```

Word Embeddings

Word embeddings é uma forma de representação de palavras e documentos usando uma vetores. A posição das palavras em um espaço vetorial é aprendido do texto e é baseados nas palavras que o rodeiam. Word embeddings podem ser treinados usando como input o próprio corpus ou pode ser gerado usando modelos pré treinados como Glove, FastText ou Word2Vec.

implementando word2vec

[29]:
```
# unindo todos os textos
corpus = todos['Resumo Português:']
```

```python
corpus = corpus.str.cat(sep=' ')
```

[30]:
```python
# Criando uma lista de sentenças e embaralhando-as
corpus = nltk.sent_tokenize(corpus)
shuffle(corpus)
```

[31]:
```python
# Tokenizando as sentenças
corpus_processado = []
for sentence in corpus:
    corpus_processado.append(word_tokenize(sentence))
```

[32]:
```python
# Treinando um modelo de Word2Vec
BDTD_word2vec_50 = Word2Vec(corpus_processado, size=50, window=10, min_count=1,␣
 ↪workers=4, iter=100)
```

[33]:
```python
# Exemplo do vetor da palavra água
BDTD_word2vec_50.wv['água']
```

[33]:
```
array([-3.6960294e+00, -1.3259159e+00, -2.2412670e+00,  5.7865171e+00,
        2.2329526e+00, -5.5173335e+00, -5.9714718e+00,  7.3408980e+00,
        3.9766235e+00, -4.5603027e+00,  4.6718297e+00, -2.3560665e+00,
       -6.8443626e-01, -6.0587273e+00,  3.6310940e+00, -1.5324932e+01,
        2.0639896e+00,  6.0305029e-01,  3.7788615e+00,  4.0281076e+00,
        3.9048851e+00,  1.1163657e+00,  2.3122082e+00,  4.4901333e+00,
       -3.3812339e+00,  2.3412783e+00, -5.2094436e+00, -1.1195867e+00,
       -3.8026874e+00,  9.2307749e+00, -1.8853464e+00, -6.3019433e+00,
        3.5904100e+00, -2.6272061e+00, -4.8584223e+00,  7.7866459e+00,
       -1.3319616e+00,  5.1870914e+00, -5.6637077e+00, -1.5475802e+00,
        9.2607457e-03, -5.3038816e+00,  3.9777195e+00, -6.1717930e+00,
       -5.0633631e+00, -1.7807996e+00,  8.1977360e-03, -3.5111365e+00,
       -5.9699243e-01,  1.4175992e+00], dtype=float32)
```

[34]:
```python
# Vetores mais similares a palavra água
BDTD_word2vec_50.wv.similar_by_word('água')
```

[34]:
```
[('vazão', 0.730722188949585),
 ('solo', 0.6993394494056702),
 ('líquido', 0.687868058681488),
 ('ar', 0.6873413920402527),
 ('areia', 0.6845143437385559),
 ('argila', 0.6819689869880676),
 ('umidade', 0.6753614544868469),
 ('ozônio', 0.67515629529953),
 ('irrigação', 0.6692394018173218),
 ('óleo', 0.6592279076576233)]
```

[15]:
```python
# Gravando e lendo os modelos de embeddings
#BDTD_word2vec_50.save("Embeddings\BDTD_word2vec_50")
BDTD_word2vec_50 = Word2Vec.load("Embeddings\BDTD_word2vec_50")
```

[16]:
```
# Indexando as palavras presentes no modelo Word2Vec
word2index = {}
for index, word in enumerate(BDTD_word2vec_50.wv.index2word):
    word2index[word] = index
```

[17]:
```
# Indexando as palavras presentes no modelo Word2Vec
word2index = {}
for index, word in enumerate(BDTD_word2vec_50.wv.index2word):
    word2index[word] = index

# Função para indexar o texto usando os índices do modelo Word2Vec
def index_pad_text(text, maxlen, word2index):
    maxlen = 400
    new_text = []
    for sent in text:
        temp_sent = []
        for word in word_tokenize(sent):
            try:
                temp_sent.append(word2index[word])
            except:
                pass
        # Estebelecendo um limite máximo de palavras para cada resumo (padding)
        if len(temp_sent) > maxlen:
            temp_sent = temp_sent[:400]
        else:
            temp_sent += [0] * (maxlen - len(temp_sent))
        new_text.append(temp_sent)

    return np.array(new_text)

maxlen = 400
train_seq_x = index_pad_text(train_x, maxlen, word2index)
test_seq_x = index_pad_text(test_x, maxlen, word2index)
```

4 Model Building

A etapa final do framework de classificação de texto é treinar um classificador usando os atributos criados anteriormente. Os seguintes algoritmos de aprendizado de máquina foram implementados:

- Naive Bayes Classifier
- Linear Classifier - Logistic Regression
- Support Vector Machine
- Bagging Models - Random Forest
- Boosting Models - Xtereme Gradient Boosting
- 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

A função abaixo é usada para treinar os modelos. Ela aceita o classificador, o vetor de atributos dos dados de treinamento, as classes de treinamento, os atributos dos dados de teste e a informação se o classificador é uma rede neural. Com essas informações o modelo é treinado, a acurácia pe computada e, nos casos das redes neurais, um gráfico das épocas de treinamento é apresentado.

[18]:
```
def train_model(classifier, feature_vector_train, label, feature_vector_test,␣
↪is_neural_net=False):
    # fit the training dataset on the classifier

    if is_neural_net:
        callbacks = EarlyStopping(monitor='val_acc', patience=10,␣
↪restore_best_weights=True)
        history = classifier.fit(feature_vector_train,
                                 label, #to_categorical(label),
                                 epochs=1000,
                                 batch_size=64,
                                 validation_split=0.25,
                                 callbacks=[callbacks])

    # plot the loss
        # list all data in history
        print(history.history.keys())
        # summarize history for loss
        plt.plot(history.history['acc'])
        plt.plot(history.history['val_acc'])
        plt.title('model acc')
        plt.ylabel('acc')
        plt.xlabel('epoch')
        plt.legend(['acc', 'val_acc'], loc='upper left')
        plt.show()

    else:
        classifier.fit(feature_vector_train, label)

    # predict the labels on validation dataset
    predictions = classifier.predict(feature_vector_test)
    predictions = np.rint(predictions)

    num_classes = 2
    return (metrics.accuracy_score(predictions, test_y),
```

```
            tf.confusion_matrix(predictions, test_y, num_classes))
```

Naive Bayes

[19]:
```
# Naive Bayes on Count Vectors
accuracy, confusion = train_model(naive_bayes.MultinomialNB(), xtrain_count,␣
 ↪train_y, xtest_count)
print ("NB, Count Vectors: ", accuracy)
with tf.Session() as sess:
    print(confusion.eval())

# Naive Bayes on Word Level TF IDF Vectors
accuracy, confusion = train_model(naive_bayes.MultinomialNB(), xtrain_tfidf,␣
 ↪train_y, xtest_tfidf)
print ("NB, WordLevel TF-IDF Vectors: ", accuracy)
with tf.Session() as sess:
    print(confusion.eval())

# Naive Bayes on Ngram Level TF IDF Vectors
accuracy, confusion = train_model(naive_bayes.MultinomialNB(),␣
 ↪xtrain_tfidf_ngram, train_y, xtest_tfidf_ngram)
print ("NB, N-Gram Vectors: ", accuracy)
with tf.Session() as sess:
    print(confusion.eval())

# Naive Bayes on Character Level TF IDF Vectors
accuracy, confusion = train_model(naive_bayes.MultinomialNB(),␣
 ↪xtrain_tfidf_ngram_chars, train_y, xtest_tfidf_ngram_chars)
print ("NB, CharLevel Vectors: ", accuracy)
with tf.Session() as sess:
    print(confusion.eval())
```

```
NB, Count Vectors:  0.7405660377358491
[[324 120]
 [100 304]]
NB, WordLevel TF-IDF Vectors:  0.7275943396226415
[[336 143]
 [ 88 281]]
NB, N-Gram Vectors:  0.8926886792452831
[[387  54]
 [ 37 370]]
NB, CharLevel Vectors:  0.8360849056603774
[[382  97]
 [ 42 327]]
```

Linear Classifier - Logistic Regression

[20]:
```
# Linear Classifier on Count Vectors
```

```
accuracy, confusion = train_model(linear_model.LogisticRegression(),␣
 ↪xtrain_count, train_y, xtest_count)
print ("LR, Count Vectors: ", accuracy)
with tf.Session() as sess:
    print(confusion.eval())

# Linear Classifier on Word Level TF IDF Vectors
accuracy, confusion = train_model(linear_model.LogisticRegression(),␣
 ↪xtrain_tfidf, train_y, xtest_tfidf)
print ("LR, WordLevel TF-IDF Vectors: ", accuracy)
with tf.Session() as sess:
    print(confusion.eval())

# Linear Classifier on Ngram Level TF IDF Vectors
accuracy, confusion = train_model(linear_model.LogisticRegression(),␣
 ↪xtrain_tfidf_ngram, train_y, xtest_tfidf_ngram)
print ("LR, N-Gram Vectors: ", accuracy)
with tf.Session() as sess:
    print(confusion.eval())

# Linear Classifier on Character Level TF IDF Vectors
accuracy, confusion = train_model(linear_model.LogisticRegression(),␣
 ↪xtrain_tfidf_ngram_chars, train_y, xtest_tfidf_ngram_chars)
print ("LR, CharLevel Vectors: ", accuracy)
with tf.Session() as sess:
    print(confusion.eval())
```

```
LR, Count Vectors:  0.7452830188679245
[[325 117]
 [ 99 307]]
LR, WordLevel TF-IDF Vectors:  0.7417452830188679
[[311 106]
 [113 318]]
LR, N-Gram Vectors:  0.9198113207547169
[[386  30]
 [ 38 394]]
LR, CharLevel Vectors:  0.7971698113207547
[[335  83]
 [ 89 341]]
```

Support Vector Machine (SVM)

[21]:
```
# SVM on Count Vectors
accuracy, confusion = train_model(svm.SVC(gamma='scale'), xtrain_count, train_y,␣
 ↪xtest_count)
print ("SVM, Count Vectors: ", accuracy)
with tf.Session() as sess:
    print(confusion.eval())
```

```python
# SVM on Word Level TF IDF Vectors
accuracy, confusion = train_model(svm.SVC(gamma='scale'), xtrain_tfidf, train_y,␣
 ↪xtest_tfidf)
print ("SVM, Word Level TF IDF Vectors: ", accuracy)
with tf.Session() as sess:
    print(confusion.eval())

# SVM on Ngram Level TF IDF Vectors
accuracy, confusion = train_model(svm.SVC(gamma='scale'), xtrain_tfidf_ngram,␣
 ↪train_y, xtest_tfidf_ngram)
print ("SVM, Ngram Level TF IDF Vectors: ", accuracy)
with tf.Session() as sess:
    print(confusion.eval())

# SVM on Character Level TF IDF Vectors
accuracy, confusion = train_model(svm.SVC(gamma='scale'),␣
 ↪xtrain_tfidf_ngram_chars, train_y, xtest_tfidf_ngram_chars)
print ("SVM, Character Level TF IDFs Vectors: ", accuracy)
with tf.Session() as sess:
    print(confusion.eval())
```

```
SVM, Count Vectors:  0.7535377358490566
[[327 112]
 [ 97 312]]
SVM, Word Level TF IDF Vectors:  0.7452830188679245
[[311 103]
 [113 321]]
SVM, Ngram Level TF IDF Vectors:  0.9280660377358491
[[388  25]
 [ 36 399]]
SVM, Character Level TF IDFs Vectors:  0.847877358490566
[[354  59]
 [ 70 365]]
```

Bagging Model - Random Forest

[22]:
```python
# RF on Count Vectors
accuracy, confusion = train_model(ensemble.RandomForestClassifier(),␣
 ↪xtrain_count, train_y, xtest_count)
print ("RF, Count Vectors Vectors: ", accuracy)
with tf.Session() as sess:
    print(confusion.eval())

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

```
with tf.Session() as sess:
    print(confusion.eval())

# RF on Ngram Level TF IDF Vectors
accuracy, confusion = train_model(ensemble.RandomForestClassifier(),␣
 ↪xtrain_tfidf_ngram, train_y, xtest_tfidf_ngram)
print ("RF, Ngram Level TF IDF Vectors: ", accuracy)
with tf.Session() as sess:
    print(confusion.eval())

# RF on Character Level TF IDF Vectors
accuracy, confusion = train_model(ensemble.RandomForestClassifier(),␣
 ↪xtrain_tfidf_ngram_chars, train_y, xtest_tfidf_ngram_chars)
print ("RF, Character Level TF IDFs Vectors: ", accuracy)
with tf.Session() as sess:
    print(confusion.eval())
```

```
RF, Count Vectors Vectors:  0.6898584905660378
[[327 166]
 [ 97 258]]
RF, WordLevel TF-IDF Vectors:  0.7004716981132075
[[330 160]
 [ 94 264]]
RF, Ngram Level TF IDF Vectors:  0.8561320754716981
[[378  76]
 [ 46 348]]
RF, Character Level TF IDFs Vectors:  0.8042452830188679
[[360 102]
 [ 64 322]]
```

Boosting Model - Xtereme Gradient Boosting

[23]:
```
# Extereme Gradient Boosting on Count Vectors
accuracy, confusion = train_model(xgboost.XGBClassifier(), xtrain_count.tocsc(),␣
 ↪train_y, xtest_count.tocsc())
print ("Xgb, Count Vectors: ", accuracy)
with tf.Session() as sess:
    print(confusion.eval())

# Extereme Gradient Boosting on Word Level TF IDF Vectors
accuracy, confusion = train_model(xgboost.XGBClassifier(), xtrain_tfidf.tocsc(),␣
 ↪train_y, xtest_tfidf.tocsc())
print ("Xgb, WordLevel TF-IDF: ", accuracy)
with tf.Session() as sess:
    print(confusion.eval())

# Extereme Gradient Boosting on Ngram Level TF IDF Vectors
```

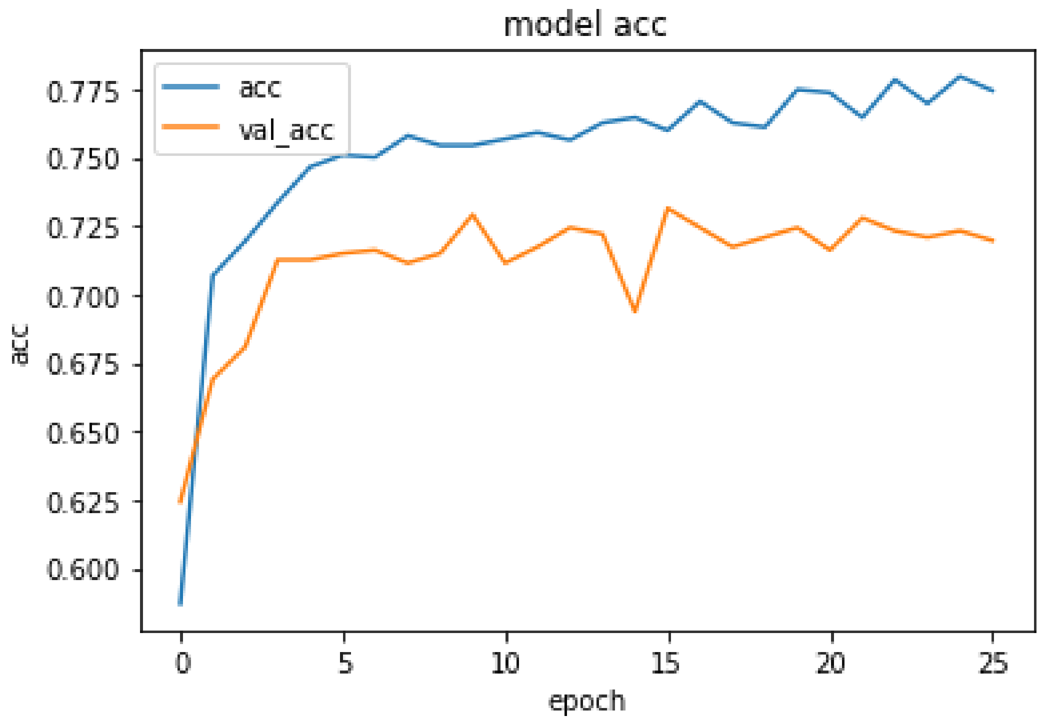
```
accuracy, confusion = train_model(xgboost.XGBClassifier(), xtrain_tfidf_ngram.
 ↪tocsc(), train_y, xtest_tfidf_ngram.tocsc())
print ("Xgb, Ngram Level Vectors: ", accuracy)
with tf.Session() as sess:
    print(confusion.eval())

# Extereme Gradient Boosting on Character Level TF IDF Vectors
accuracy, confusion = train_model(xgboost.XGBClassifier(),␣
 ↪xtrain_tfidf_ngram_chars.tocsc(), train_y, xtest_tfidf_ngram_chars.tocsc())
print ("Xgb, CharLevel Vectors: ", accuracy)
with tf.Session() as sess:
    print(confusion.eval())
```

```
Xgb, Count Vectors:  0.7358490566037735
[[318 118]
 [106 306]]
Xgb, WordLevel TF-IDF:  0.7476415094339622
[[311 101]
 [113 323]]
Xgb, Ngram Level Vectors:  0.9221698113207547
[[373  15]
 [ 51 409]]
Xgb, CharLevel Vectors:  0.8867924528301887
[[363  35]
 [ 61 389]]
```

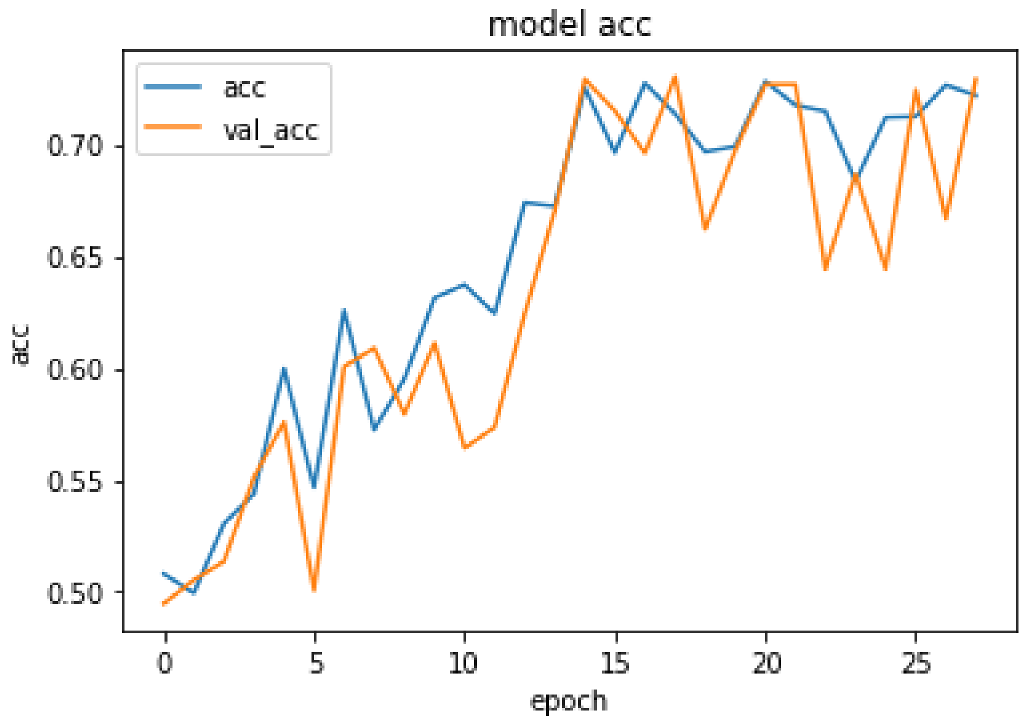
Redes neurais rasa - Mullti Layer Perceptron

[24]:
```
# Função para criar a arquitetura da rede neural
def create_model_architecture(input_size):
    model = Sequential()

    # create hidden layer
    model.add(layers.Dense(100,
                           input_shape=(input_size, ),
                           activation="sigmoid"))

    # create output layer
    model.add(layers.Dense(1, activation='sigmoid'))

    opt = optimizers.Adam()
    model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['acc'])

    return model

num_classes = 1

# Shallow Neural Network on Count Vectors Vectors
```

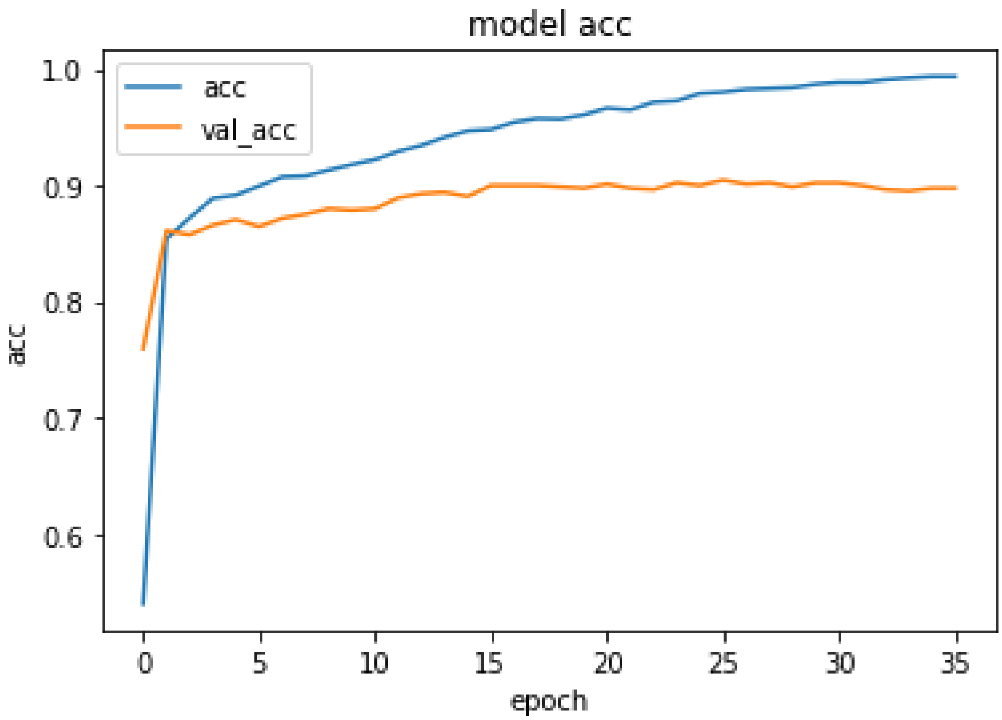
```
classifier = create_model_architecture(xtrain_count.shape[1])
accuracy, confusion = train_model(classifier, xtrain_count, train_y,␣
 ↪xtest_count, is_neural_net=True)
print ("Shallow Neural Network on Count Vectors Vectors",  accuracy)
with tf.Session() as sess:
    print(confusion.eval())


# Shallow Neural Network on Word Level TF IDF Vectors
classifier = create_model_architecture(xtrain_tfidf.shape[1])
accuracy, confusion = train_model(classifier, xtrain_tfidf, train_y,␣
 ↪xtest_tfidf, is_neural_net=True)
print ("Shallow Neural Network on Word Level TF IDF Vectors",  accuracy,)
with tf.Session() as sess:
    print(confusion.eval())

# Shallow Neural Network on Ngram Level TF IDF Vectors
classifier = create_model_architecture(xtrain_tfidf_ngram.shape[1])
accuracy, confusion = train_model(classifier, xtrain_tfidf_ngram, train_y,␣
 ↪xtest_tfidf_ngram, is_neural_net=True)
print ("Shallow Neural Network on Ngram Level TF IDF Vectors",  accuracy)
with tf.Session() as sess:
    print(confusion.eval())


# Shallow Neural Network on Character Level TF IDF Vectors
classifier = create_model_architecture(xtrain_tfidf_ngram_chars.shape[1])
accuracy, confusion = train_model(classifier, xtrain_tfidf_ngram_chars, train_y,␣
 ↪xtest_tfidf_ngram_chars, is_neural_net=True)
print ("Shallow Neural Network on Character Level TF IDF Vectors",  accuracy)
with tf.Session() as sess:
    print(confusion.eval())
```



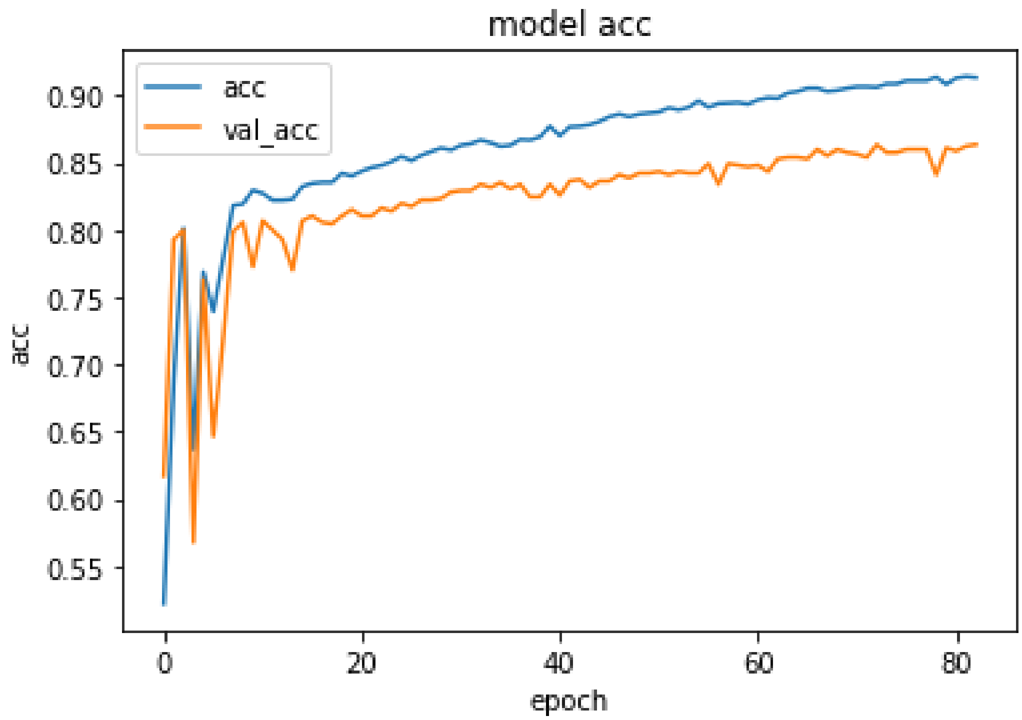

```
Shallow Neural Network on Count Vectors Vectors 0.7665094339622641
[[327 101]
 [ 97 323]]
```




```
Shallow Neural Network on Word Level TF IDF Vectors 0.7346698113207547
[[295  96]
 [129 328]]
```



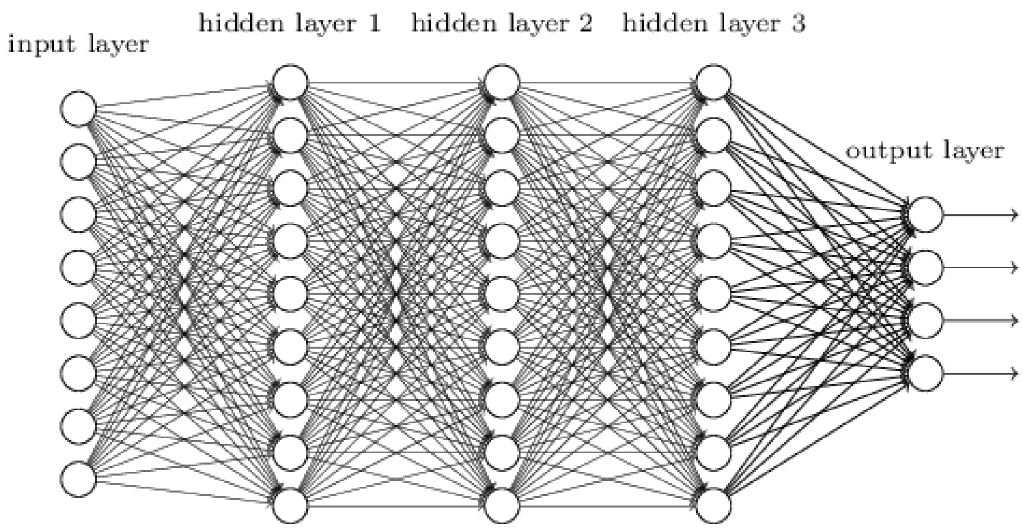

```
Shallow Neural Network on Ngram Level TF IDF Vectors 0.9186320754716981
[[390  35]
 [ 34 389]]
```




```
Shallow Neural Network on Character Level TF IDF Vectors 0.8832547169811321
[[365  40]
 [ 59 384]]
```

Deep Neural Networks

Redes neurais profundas são redes mais complexas em que as camadas escondidas realizam operações mais complexas. Diferentes tipos de redes podem ser aplicados aos problemas de classificação de texto.



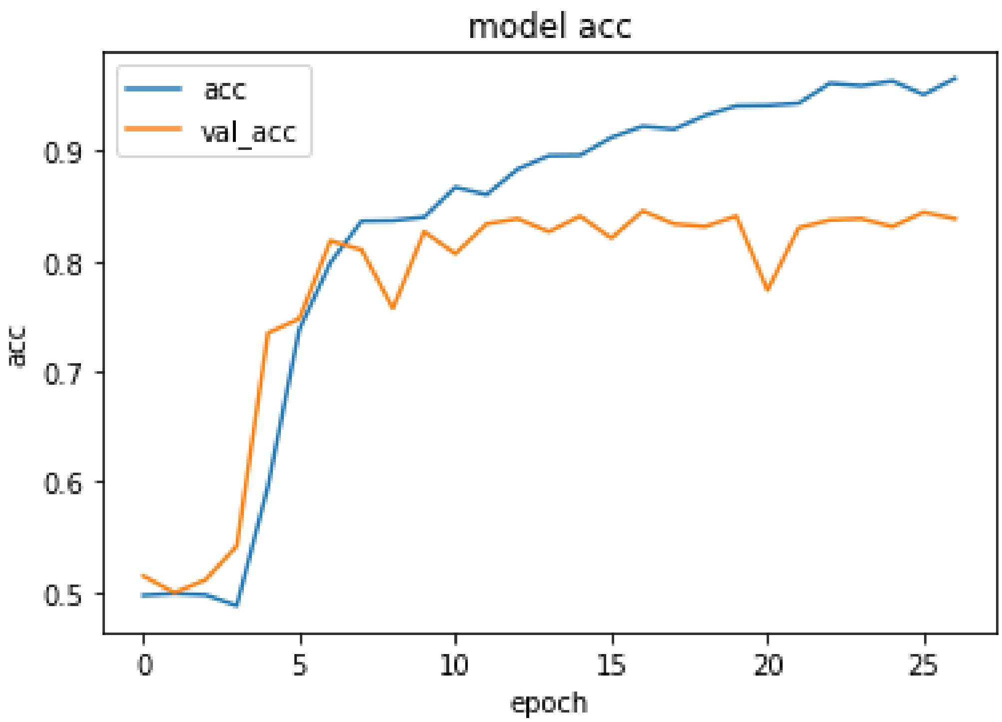
Convolutional Neural Network

[25]:
```
def create_cnn():
    # Add an Input Layer
    input_layer = layers.Input((maxlen, ))

    # Add the word embedding Layer
    embedding_layer = BDTD_word2vec_50.wv.
 ↪get_keras_embedding(train_embeddings=False)(input_layer)
    embedding_layer = layers.SpatialDropout1D(0.05)(embedding_layer)

    # Add the convolutional Layer e pooling layer
    conv_layer_1 = layers.Convolution1D(128, 5,␣
 ↪activation="relu")(embedding_layer)
    pooling_layer_1 = layers.MaxPooling1D(2)(conv_layer_1)
    pooling_layer_1 = layers.Dropout(0.15)(pooling_layer_1)

    conv_layer_2 = layers.Convolution1D(128, 5,␣
 ↪activation="relu")(pooling_layer_1)
    pooling_layer_2 = layers.MaxPooling1D(2)(conv_layer_2)
    pooling_layer_2 = layers.Dropout(0.15)(pooling_layer_2)

    conv_layer_3 = layers.Convolution1D(128, 5,␣
 ↪activation="relu")(pooling_layer_2)
    pooling_layer_3 = layers.GlobalMaxPooling1D()(conv_layer_3)
    pooling_layer_3 = layers.Dropout(0.15)(pooling_layer_3)

    # Add the output Layers
    output_layer1 = layers.Dense(512, activation="relu")(pooling_layer_3)
    output_layer2 = layers.Dense(512, activation="relu")(output_layer1)
    output_layer2 = layers.Dropout(0.12)(output_layer2)
```

```
    output_layer3 = layers.Dense(1, activation="sigmoid")(output_layer2)

    # Compile the model
    model = models.Model(inputs=input_layer, outputs=output_layer3)
    model.compile(optimizer=optimizers.Adam(), loss='binary_crossentropy',␣
 ↪metrics=['acc'])
    print(model.summary())
    return model

classifier = create_cnn()
accuracy, confusion = train_model(classifier, train_seq_x, train_y, test_seq_x,␣
 ↪is_neural_net=True)
classifier.save("model_cnn.h5")

print ("CNN, Word Embeddings",  accuracy)
with tf.Session() as sess:
    print(confusion.eval())
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 400)               0
_________________________________________________________________
embedding_1 (Embedding)      (None, 400, 50)           9289150
_________________________________________________________________
spatial_dropout1d_1 (Spatial (None, 400, 50)           0
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 396, 128)          32128
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 198, 128)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 198, 128)          0
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 194, 128)          82048
_________________________________________________________________
max_pooling1d_2 (MaxPooling1 (None, 97, 128)           0
_________________________________________________________________
dropout_2 (Dropout)          (None, 97, 128)           0
_________________________________________________________________
conv1d_3 (Conv1D)            (None, 93, 128)           82048
_________________________________________________________________
global_max_pooling1d_1 (Glob (None, 128)               0
_________________________________________________________________
dropout_3 (Dropout)          (None, 128)               0
_________________________________________________________________
```

```
dense_9 (Dense)              (None, 512)               66048
_________________________________________________________________
dense_10 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_4 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_11 (Dense)             (None, 1)                 513
=================================================================
Total params: 9,814,591
Trainable params: 525,441
Non-trainable params: 9,289,150
_________________________________________________________________
```




```
CNN, Word Embeddings 0.8620283018867925
[[336  29]
 [ 88 395]]
```

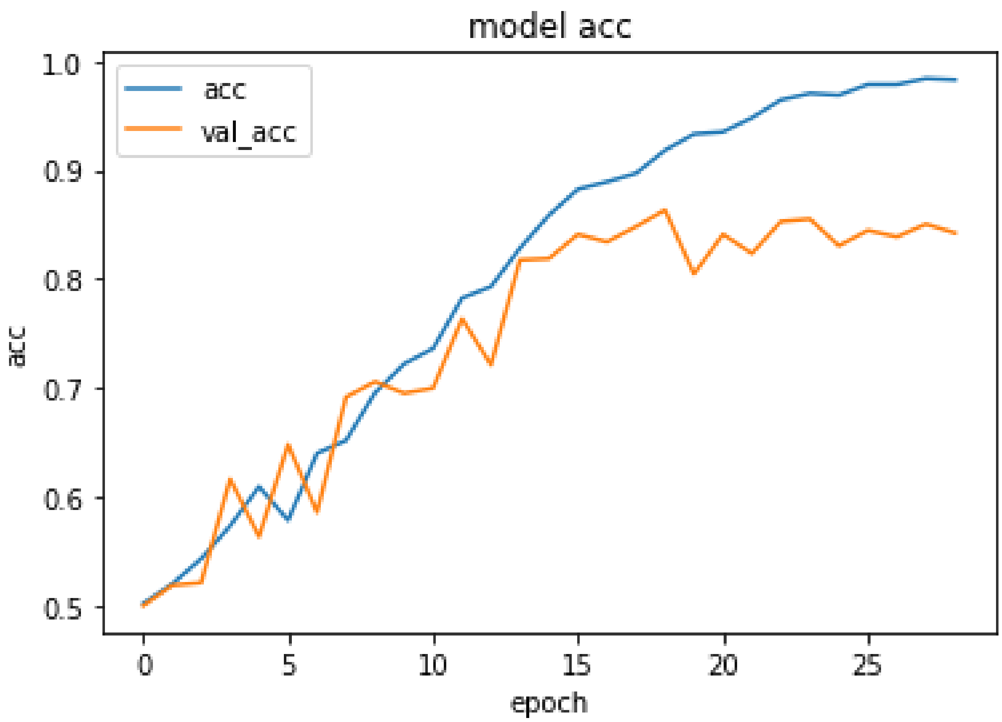
Recurrent Neural Network – LSTM

[26]:
```
def create_rnn_lstm():

    # Add an Input Layer
    input_layer = layers.Input((maxlen, ))
```

```
    # Add the word embedding Layer

    embedding_layer = BDTD_word2vec_50.wv.
 ↪get_keras_embedding(train_embeddings=True)(input_layer)

    embedding_layer = layers.SpatialDropout1D(0.05)(embedding_layer)

    # Add the LSTM Layer
    lstm_layer_1 = layers.LSTM(256, return_sequences=True)(embedding_layer)␣
 ↪#return_sequences=True
    lstm_layer_2 = layers.LSTM(128, return_sequences=True)(lstm_layer_1)
    lstm_layer_3 = layers.LSTM(64)(lstm_layer_2)

    # Add the output Layers
    output_layer1 = layers.Dense(256, activation="relu")(lstm_layer_3)
    output_layer1 = layers.Dropout(0.2)(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',␣
 ↪metrics=['acc'])
    print(model.summary())
    return model

classifier = create_rnn_lstm()
accuracy, confusion = train_model(classifier, train_seq_x, train_y, test_seq_x,␣
 ↪is_neural_net=True)
print ("RNN-LSTM, Word Embeddings",  accuracy)
with tf.Session() as sess:
    print(confusion.eval())
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 400)               0
_________________________________________________________________
embedding_2 (Embedding)      (None, 400, 50)           9289150
_________________________________________________________________
spatial_dropout1d_2 (Spatial (None, 400, 50)           0
_________________________________________________________________
lstm_1 (LSTM)                (None, 400, 256)          314368
_________________________________________________________________
lstm_2 (LSTM)                (None, 400, 128)          197120
_________________________________________________________________
lstm_3 (LSTM)                (None, 64)                49408
_________________________________________________________________
```

```
dense_12 (Dense)             (None, 256)               16640
_________________________________________________________________
dropout_5 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_13 (Dense)             (None, 1)                 257
=================================================================
Total params: 9,866,943
Trainable params: 9,866,943
Non-trainable params: 0
_________________________________________________________________
```




```
RNN-LSTM, Word Embeddings 0.8820754716981132
[[359  35]
 [ 65 389]]
```

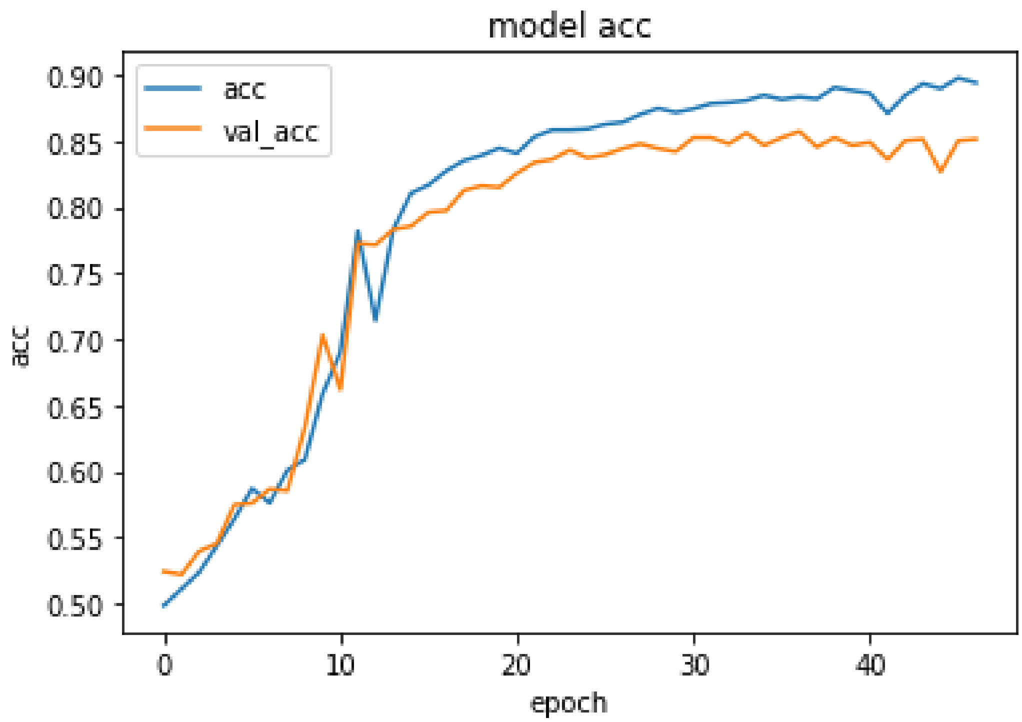
Recurrent Neural Network – GRU

[27]:
```python
def create_rnn_gru():
    # Add an Input Layer
    input_layer = layers.Input((maxlen, ))

    # Add the word embedding Layer
    embedding_layer = BDTD_word2vec_50.wv.
 ↪get_keras_embedding(train_embeddings=True)(input_layer)
```

```python
    embedding_layer = layers.SpatialDropout1D(0.05)(embedding_layer)

    # Add the GRU Layer
    gru_layer = layers.GRU(20, return_sequences=True)(embedding_layer)
    gru_layer_1 = layers.GRU(20)(gru_layer)

    # Add the output Layers
    output_layer1 = layers.Dense(50, activation="sigmoid")(gru_layer_1)
    output_layer1 = layers.Dropout(0.2)(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',␣
 ↪metrics=['acc'])
    print(model.summary())
    return model

classifier = create_rnn_gru()
accuracy, confusion = train_model(classifier, train_seq_x, train_y, test_seq_x,␣
 ↪is_neural_net=True)
print ("RNN-GRU, Word Embeddings",  accuracy)
with tf.Session() as sess:
    print(confusion.eval())
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         (None, 400)               0
_________________________________________________________________
embedding_3 (Embedding)      (None, 400, 50)           9289150
_________________________________________________________________
spatial_dropout1d_3 (Spatial (None, 400, 50)           0
_________________________________________________________________
gru_1 (GRU)                  (None, 400, 20)           4260
_________________________________________________________________
gru_2 (GRU)                  (None, 20)                2460
_________________________________________________________________
dense_14 (Dense)             (None, 50)                1050
_________________________________________________________________
dropout_6 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_15 (Dense)             (None, 1)                 51
=================================================================
Total params: 9,296,971
Trainable params: 9,296,971
```

```
Non-trainable params: 0
_________________________________________________________________
```




```
RNN-GRU, Word Embeddings 0.8726415094339622
[[335  19]
 [ 89 405]]
```

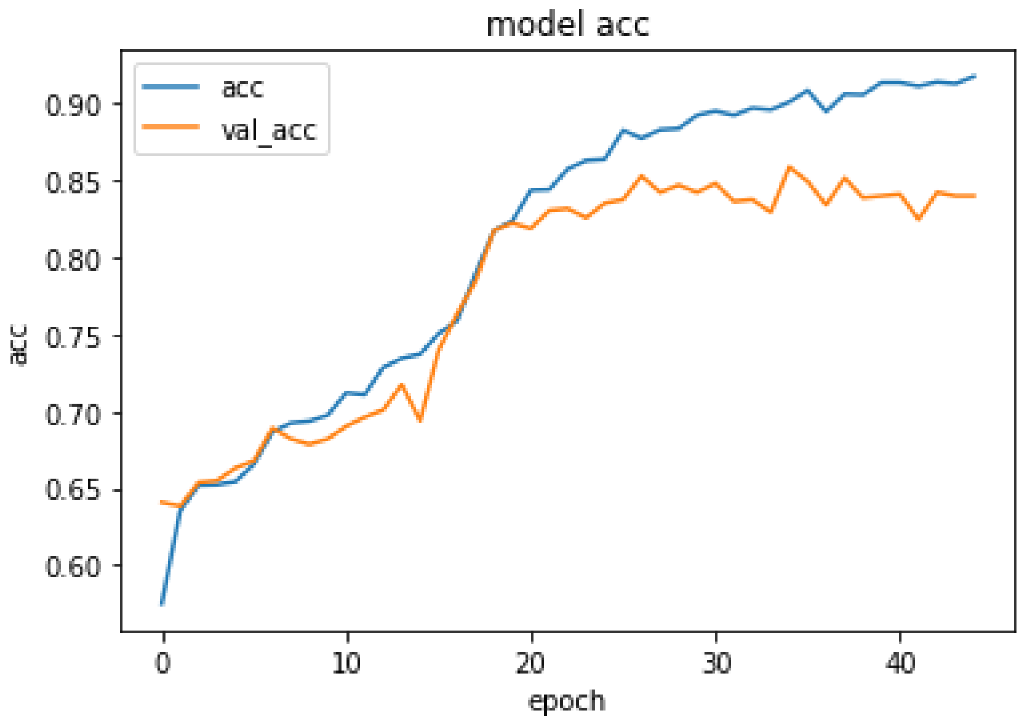
3.7.4 Bidirectional RNN

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

[28]:
```
def create_bidirectional_rnn():
    # Add an Input Layer
    input_layer = layers.Input((maxlen, ))

    # Add the word embedding Layer
    embedding_layer = BDTD_word2vec_50.wv.
 ↪get_keras_embedding(train_embeddings=True)(input_layer)

    embedding_layer = layers.SpatialDropout1D(0.05)(embedding_layer)

    # Add the GRU Layer
    lstm_layer = layers.Bidirectional(layers.GRU(20))(embedding_layer)
```

```python
    # Add the output Layers
    output_layer1 = layers.Dense(50, activation="relu")(lstm_layer)
    output_layer1 = layers.Dropout(0.2)(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',␣
 ↪metrics=['acc'])
    print(model.summary())
    return model

classifier = create_bidirectional_rnn()
accuracy, confusion = train_model(classifier, train_seq_x, train_y, test_seq_x,␣
 ↪is_neural_net=True)
print ("RNN-Bidirectional, Word Embeddings",  accuracy)
with tf.Session() as sess:
    print(confusion.eval())
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         (None, 400)               0
_________________________________________________________________
embedding_4 (Embedding)      (None, 400, 50)           9289150
_________________________________________________________________
spatial_dropout1d_4 (Spatial (None, 400, 50)           0
_________________________________________________________________
bidirectional_1 (Bidirection (None, 40)                8520
_________________________________________________________________
dense_16 (Dense)             (None, 50)                2050
_________________________________________________________________
dropout_7 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_17 (Dense)             (None, 1)                 51
=================================================================
Total params: 9,299,771
Trainable params: 9,299,771
Non-trainable params: 0
_________________________________________________________________
```



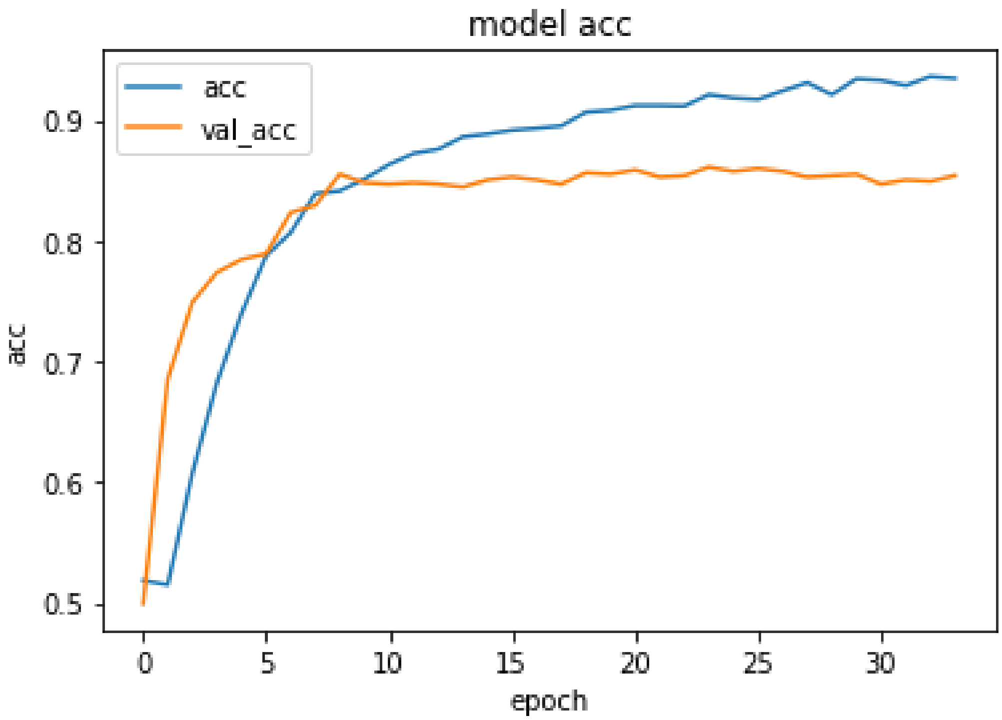

```
RNN-Bidirectional, Word Embeddings 0.8620283018867925
[[342  35]
 [ 82 389]]
```

3.7.5 Recurrent Convolutional Neural Network

[29]:
```
def create_rcnn():
    # Add an Input Layer
    input_layer = layers.Input((maxlen, ))

    # Add the word embedding Layer
    embedding_layer = BDTD_word2vec_50.wv.
 ↪get_keras_embedding(train_embeddings=True)(input_layer)

    embedding_layer = layers.SpatialDropout1D(0.05)(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="sigmoid")(embedding_layer)
```

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

    # Add the output Layers
    output_layer1 = layers.Dense(50, activation="sigmoid")(pooling_layer)
    output_layer1 = layers.Dropout(0.2)(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',␣
 ↪metrics=['acc'])
    print(model.summary())
    return model

classifier = create_rcnn()
accuracy, confusion = train_model(classifier, train_seq_x, train_y, test_seq_x,␣
 ↪is_neural_net=True)
print ("RCNN, Word Embeddings",  accuracy)
with tf.Session() as sess:
    print(confusion.eval())
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         (None, 400)               0
_________________________________________________________________
embedding_5 (Embedding)      (None, 400, 50)           9289150
_________________________________________________________________
spatial_dropout1d_5 (Spatial (None, 400, 50)           0
_________________________________________________________________
conv1d_4 (Conv1D)            (None, 398, 100)          15100
_________________________________________________________________
global_max_pooling1d_2 (Glob (None, 100)               0
_________________________________________________________________
dense_18 (Dense)             (None, 50)                5050
_________________________________________________________________
dropout_8 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_19 (Dense)             (None, 1)                 51
=================================================================
Total params: 9,309,351
Trainable params: 9,309,351
Non-trainable params: 0
_________________________________________________________________
```




```
RCNN, Word Embeddings 0.8679245283018868
[[362  50]
 [ 62 374]]
```

5 Conclusão

O modelo com o melhor resultado encontrado foi o algoritmo Support Vector Machine usando vetores TF-IDF com Bigramas e Trigramas.