COMP4432 Individual Project

Sentiment Analysis and Text Classification on IMDB Movie Reviews Dataset

JAHJA Darwin // 16094501d

Intro

Sentiment analysis is a challenging subject in machine learning. People express their emotions in language that is often obscured by sarcasm, ambiguity, and plays on words, all of which could be very misleading for both humans and computers. In this project, we are going to perform some sentimental analysis on the IMDB movie reviews to see how we can train a model that predicts whether a review is positive or negative.

In these 2 notebooks, we are going to combining pre-trained word2vec embeddings with Bidirectional Long short-term memory (BiLSTM) to build out our machine learning modal. The source code and report are based on two Jupyter Notebooks -- one for data preprocessing and Word2Vec Modelling, one for modelling -- for better understanding.

To give a short summary, in the beginning without using the word2vec embeddings (only tokenize), I reach ~70% accuracy with the testing dataset. With pre-trained word2vec embeddings and some parameter tunings, the final accurary have reached 78%. The result could probably still be improved with more training dataset and parameter tuning.

Importing packages and dataset

First, we need to import package and dataset to start.

As the data is not properly organized in a formatted way, I have processes all the datasets into `.csv` file.

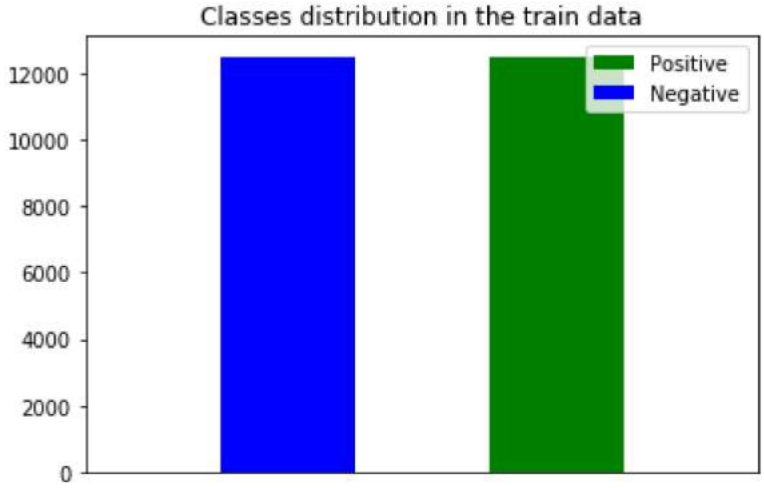
In [1]:
```
import warnings
warnings.filterwarnings('ignore')

# Data manipulation
import numpy as np
import pandas as pd
import re
import pickle

# Visualization
import matplotlib.pyplot as plt
import seaborn as sb

# Tools for preprocessing input data
from bs4 import BeautifulSoup
from nltk import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer

# Tools for creating ngrams and vectorizing input data
from gensim.models import Word2Vec, Phrases

# File system tool
import os
from pathlib import Path
```

In [2]:
```
# # Input data files are available in the "input/" directory
# # This process takes a lot of IO process and I've read and write all data in
to '.csv' file
# def load_data_from(directory, sublist):
#     results, texts = [], []
#     for label_type in sublist:
#         dir_name = os.path.join(directory, label_type)
#         for fname in os.listdir(dir_name):
#             if fname[-4:] == '.txt':
#                 f = open(os.path.join(dir_name, fname), encoding='utf8')
#                 texts.append(f.read())
#                 f.close()
#                 if label_type == 'neg':
#                     results.append(0)
#                 elif label_type == 'pos':
#                     results.append(1)
#     return results, texts

# # Note: change '\\' to '/' in unix based system
# imdb_dir = 'input\\imdb-movie-reviews-dataset\\aclImdb'
# train_dir = os.path.join(imdb_dir, 'train')
# test_dir = os.path.join(imdb_dir, 'test')

# sentiments, reviews = load_data_from(train_dir, ['pos', 'neg'])
# test_sentiments, test_reviews = load_data_from(test_dir, ['pos', 'neg'])
# _, unsup_reviews = load_data_from(train_dir, ['unsup'])

# train_data = pd.DataFrame({'review': reviews, 'sentiment': sentiments})
# test_data = pd.DataFrame({'review': test_reviews, 'sentiment': test_sentimen
ts})
# unsup_data = pd.DataFrame({'review': unsup_reviews})

# train_data.to_csv('resources/train.csv', index=False)
# test_data.to_csv('resources/test.csv', index=False)
# unsup_data.to_csv('resources/unsup.csv', index=False)
```

In [3]:
```
# Read files from csv
train_data = pd.read_csv('resources/train.csv')
test_data = pd.read_csv('resources/test.csv')
unsup_data = pd.read_csv('resources/unsup.csv')

datasets = [train_data, test_data, unsup_data]
titles = ['Train data', 'Test data', 'Unsup train data']

for dataset, title in zip(datasets,titles):
    print(f'\n{title}')
    dataset.info()
```

```
Train data
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 25000 entries, 0 to 24999
Data columns (total 2 columns):
review       25000 non-null object
sentiment    25000 non-null int64
dtypes: int64(1), object(1)
memory usage: 390.8+ KB

Test data
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 25000 entries, 0 to 24999
Data columns (total 2 columns):
review       25000 non-null object
sentiment    25000 non-null int64
dtypes: int64(1), object(1)
memory usage: 390.8+ KB

Unsup train data
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50000 entries, 0 to 49999
Data columns (total 1 columns):
review    50000 non-null object
dtypes: object(1)
memory usage: 390.8+ KB
```

In [4]:
```python
# Show class distribution
plt.hist(train_data[train_data.sentiment == 1].sentiment,
         bins=2, color='green', label='Positive')
plt.hist(train_data[train_data.sentiment == 0].sentiment,
         bins=2, color='blue', label='Negative')
plt.title('Classes distribution in the train data')
plt.xticks([])
plt.xlim(-0.5, 2)
plt.legend()
plt.show()
```



In [5]:
```python
# Prepare total number of review
all_reviews = np.array([], dtype=str)
for dataset in datasets:
    all_reviews = np.concatenate((all_reviews, dataset.review), axis=0)
print('Total number of reviews:', len(all_reviews))
```

```
Total number of reviews: 100000
```

Data Preprocessing

Let's see how an original review looks like:

In [6]:
```python
# Limiting some output
print(all_reviews[0][:500] + '...')
```

```
Bromwell High is a cartoon comedy. It ran at the same time as some other progr
ams about school life, such as "Teachers". My 35 years in the teaching profess
ion lead me to believe that Bromwell High's satire is much closer to reality t
han is "Teachers". The scramble to survive financially, the insightful student
s who can see right through their pathetic teachers' pomp, the pettiness of th
e whole situation, all remind me of the schools I knew and their students. Whe
n I saw the episode in which a s...
```

There are HTML tags (e.g. `<br />`), punctuations, abbreviations - all common issues when processing text from the Internet.

Apart from those issues, we need to deal with the common words (a.k.a stop words) that don't carry much meaning (e.g. in, on, to, etc.). Also, we also need to perform lemmatization to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form. (e.g. car, cars, car's, cars' => car).

Therefore, we need to clean and tidy them up using some useful and handy packages.

- **BeautifulSoup:** For cleaning up HTML Markup
- **Natural Language Toolkit (NLTK):** Tokenize words, Remove stop words and perform Lemmatize

Below we define some functions to perform the above operations:

In [7]:
```
stop_words = set(stopwords.words("english"))
lemmatizer = WordNetLemmatizer()

def clean_review(raw_review: str) -> str:
    # 1. Remove HTML
    review_text = BeautifulSoup(raw_review, "lxml").get_text()
    # 2. Remove non-letters
    letters_only = re.sub("[^a-zA-Z]", " ", review_text)
    # 3. Convert to lower case
    lowercase_letters = letters_only.lower()
    return lowercase_letters

def lemmatize(tokens: list) -> list:
    # Do Lemmatize
    lemmatized_tokens = list(map(lemmatizer.lemmatize, tokens))
    further_lemmatized_tokens = list(map(lambda x: lemmatizer.lemmatize(x, "v"
), lemmatized_tokens))
    return further_lemmatized_tokens

def preprocess(review: str, total: int, show_progress: bool = True) -> list:
    if show_progress:
        global counter
        counter += 1
        print('Processing... %6i/%6i'% (counter, total), end='\r')
    # 1. Clean text
    review = clean_review(review)
    # 2. Split into individual words
    tokens = word_tokenize(review)
    # 3. Remove stop words
    meaningful = list(filter(lambda x: not x in stop_words, tokens))
    # 4. Lemmatize
    lemmas = lemmatize(meaningful)
    # 5. Join the words back into one string separated by space,
    # and return the result.
    return lemmas
```

And let's preprocessing the dataset and see how are the processed review look like.

In [8]:
```
# Run preprocessing to the training dataset
counter = 0
clean_all_reviews = np.array(list(map(lambda x: preprocess(x, all_reviews.size
), all_reviews)))
```

```
Processing... 100000/100000
```

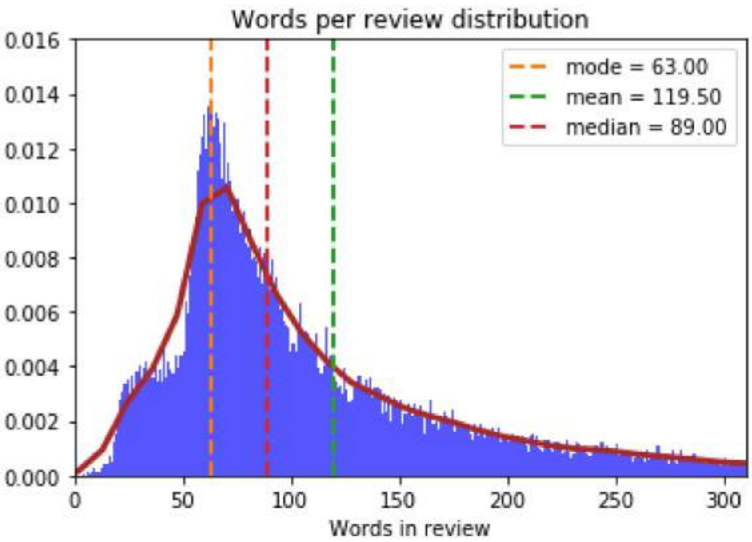
In [9]:
```
# Compare how a review has been processed
print(all_reviews[0])
print(clean_all_reviews[0])
```

```
Bromwell High is a cartoon comedy. It ran at the same time as some other progr
ams about school life, such as "Teachers". My 35 years in the teaching profess
ion lead me to believe that Bromwell High's satire is much closer to reality t
han is "Teachers". The scramble to survive financially, the insightful student
s who can see right through their pathetic teachers' pomp, the pettiness of th
e whole situation, all remind me of the schools I knew and their students. Whe
n I saw the episode in which a student repeatedly tried to burn down the schoo
l, I immediately recalled ......... at .......... High. A classic line: INSPEC
TOR: I'm here to sack one of your teachers. STUDENT: Welcome to Bromwell High.
I expect that many adults of my age think that Bromwell High is far fetched. W
hat a pity that it isn't!
['bromwell', 'high', 'cartoon', 'comedy', 'run', 'time', 'program', 'school',
'life', 'teacher', 'year', 'teach', 'profession', 'lead', 'believe', 'bromwel
l', 'high', 'satire', 'much', 'closer', 'reality', 'teacher', 'scramble', 'sur
vive', 'financially', 'insightful', 'student', 'see', 'right', 'pathetic', 'te
acher', 'pomp', 'pettiness', 'whole', 'situation', 'remind', 'school', 'know',
'student', 'saw', 'episode', 'student', 'repeatedly', 'try', 'burn', 'school',
'immediately', 'recall', 'high', 'classic', 'line', 'inspector', 'sack', 'on
e', 'teacher', 'student', 'welcome', 'bromwell', 'high', 'expect', 'many', 'ad
ult', 'age', 'think', 'bromwell', 'high', 'far', 'fetch', 'pity']
```

Let's also see how the word per review distribution for the training data.

In [10]:

```
X_train_data = clean_all_reviews[:train_data.shape[0]]
Y_train_data = train_data.sentiment.values

train_data['review_lenght'] = np.array(list(map(len, X_train_data)))
median = train_data['review_lenght'].median()
mean = train_data['review_lenght'].mean()
mode = train_data['review_lenght'].mode()[0]

fig, ax = plt.subplots()
sb.distplot(train_data['review_lenght'], bins=train_data['review_lenght'].max
(),
            hist_kws={"alpha": 0.66, "color": "blue"}, ax=ax,
            kde_kws={"color": "brown", 'linewidth': 3})
ax.set_xlim(left=0, right=np.percentile(train_data['review_lenght'], 95))
ax.set_xlabel('Words in review')
ymax = 0.016
plt.ylim(0, ymax)
ax.plot([mode, mode], [0, ymax], '--', label=f'mode = {mode:.2f}', linewidth=2
)
ax.plot([mean, mean], [0, ymax], '--', label=f'mean = {mean:.2f}', linewidth=2
)
ax.plot([median, median], [0, ymax], '--', label=f'median = {median:.2f}', lin
ewidth=2)
ax.set_title('Words per review distribution')
plt.legend()
plt.show()
```



Distributed Word Vectors

With the processed `clean_all_reviews` data, we can now proceed to construct our distributed word vectors model using the Word2vec algorithm.

Word2vec (Google, 2013) is a neural network implementation that learns distributed representations for words. It does not need labels in order to create meaningful representations. This is useful, since most data in the real world is unlabeled. If the network is given enough training data (tens of billions of words), it produces word vectors with intriguing characteristics. Words with similar meanings appear in clusters, and clusters are spaced such that some word relationships, such as analogies, can be reproduced using vector math. The famous example is that, with highly trained word vectors, "king - man + woman = queen."

Let's first use gensim's phrases to find bigrams or trigrams from our data:

In [11]:
```
%%time
bigrams = Phrases(sentences=clean_all_reviews)
```

```
Wall time: 21.9 s
```

In [12]:
```
%%time
trigrams = Phrases(sentences=bigrams[clean_all_reviews])
```

```
Wall time: 1min 10s
```

In [13]:
```
# Which will work like this (as an example)
print(bigrams['train station near new york'.split()])
```

```
['train_station', 'near', 'new_york']
```

And let's start to train our Word2Vec model. (This would take quite a long time, so we run it as an individual code block)

In [14]:
```
%%time
embedding_vector_size = 256
trigrams_model = Word2Vec(
    sentences=trigrams[bigrams[clean_all_reviews]],
    size=embedding_vector_size,
    min_count=3, window=5, workers=4)
```

```
Wall time: 10min 1s
```

In [15]:
```
print("Vocabulary size:", len(trigrams_model.wv.vocab))
```

```
Vocabulary size: 92287
```

Finally done! And now we can use our word2vec model to build a word embedding. Also we can use this model to define most similar words, calculate diffence between the words, etc.

In [16]:
```
# Find similar words
trigrams_model.wv.most_similar('planet')
```

Out[16]:
```
[('earth', 0.8660895228385925),
 ('galaxy', 0.7575183510780334),
 ('specie', 0.756401777267456),
 ('universe', 0.7411483526229858),
 ('civilization', 0.7340728044509888),
 ('asteroid', 0.7148683667182922),
 ('radiation', 0.7132657766342163),
 ('spaceship', 0.705843448638916),
 ('island', 0.7051829695701599),
 ('colony', 0.7034358978271484)]
```

In [17]:
```
# Find the difference
trigrams_model.wv.doesnt_match(['planet', 'earth', 'space', 'cat'])
```

Out[17]:
```
'cat'
```

Let's output the current model to the resources directory first.

In [18]:
```
# Dump the model as a pickles
pickle.dump(trigrams_model, open('resources/trigrams_model.pickle', 'wb'))
```

The next thing we need to do is to convert sentences to sentences with ngrams for vetorizing our sentences. And with the vetorized sentences, we can transform them into sequences to build the model. The vectorizing step will be done in part 2 since that process would take quite a bit of time.

In [19]:
```
%%time
# Load test data
test_start = train_data.shape[0]
test_end = test_start + test_data.shape[0]
X_test_data = clean_all_reviews[test_start:test_end]
Y_test_data = test_data.sentiment.values

# Convert sentences to sentences-with-n-grams
print('Convert sentences to sentences-with-n-grams...', end='\r')
X_train = trigrams[bigrams[X_train_data]]
print('Convert sentences to sentences-with-n-grams... (done)')

print('Convert sentences to sentences-with-n-grams...', end='\r')
X_test = trigrams[bigrams[X_test_data]]
print('Convert sentences to sentences-with-n-grams... (done)')

# Dump the formatted dataset also as pickles
pickle.dump([X_train, Y_train_data], open('resources/trainset.pickle', 'wb'))
pickle.dump([X_test, Y_test_data], open('resources/testset.pickle', 'wb'))
```

```
Convert sentences to sentences-with-n-grams... (done)
Convert sentences to sentences-with-n-grams... (done)
Wall time: 16.5 s
```

End of Part 1

Part 2

Transform sentences to sequences

First, we need to import packages for modelling, as well as loading the Word2Vec model and the preprocessed datasets we have done in part 1:

In [1]:
```
# Mute warning for better reading
import warnings
from tensorflow.compat.v1 import logging
warnings.filterwarnings('ignore')
logging.set_verbosity(logging.ERROR)

# Data loading and manipulation
import pickle
import numpy as np

# Tools for building a model
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense, LSTM, Dropout, Bidirectional
from keras.layers.embeddings import Embedding
from keras.preprocessing.sequence import pad_sequences

# Sklearn metric tools for assessing the quality of model prediction
from sklearn.metrics import accuracy_score, confusion_matrix

# Load the data from pickle
trigrams_model = pickle.load(open('resources/trigrams_model.pickle', 'rb'))
train = pickle.load(open('resources/trainset.pickle', 'rb'))
test = pickle.load(open('resources/testset.pickle', 'rb'))

X_train, Y_train = train[0], train[1]
X_test, Y_test = test[0], test[1]
```

```
Using TensorFlow backend.
```

With the sentences-with-n-grams, we can vetorize our sentences and later transforming them into sequences for training the model.

In [2]:
```
%%time
def vectorize_data(data, vocab: dict) -> list:
    print('Vectorize sentences...', end='\r')
    keys = list(vocab.keys())
    filter_unknown = lambda word: vocab.get(word, None) is not None
    encode = lambda review: list(map(keys.index, filter(filter_unknown, review
)))
    vectorized = list(map(encode, data))
    print('Vectorize sentences... (done)')
    return vectorized

input_length = 150
X_train_pad = pad_sequences(
    sequences=vectorize_data(X_train, vocab=trigrams_model.wv.vocab),
    maxlen=input_length,
    padding='post')
print('Transform sentences to sequences... (done)')
```

```
Vectorize sentences... (done)
Transform sentences to sequences... (done)
Wall time: 5min 55s
```

Building the model using BiLSTM

We are going to use Bidirectional LSTMs (BiLSTM), which is one of the RNN architectures for deep learning that are used in occasions where the learning problem is sequential. As the movie reviews are naturally sequential, we can try using this technique to train our model to predict whether it is possitive or not.

In BiLSTM, it involves duplicating the first recurrent layer in the network so that there are now two layers side-by-side, then providing the input sequence as-is as input to the first layer and providing a reversed copy of the input sequence to the second. It usually learns faster than one-directional approach.

So here let's define and configure how the model will work:

In [3]:
```python
def build_model(embedding_matrix: np.ndarray, input_length: int):
    model = Sequential()
    # Embedding layer (lookup table of trainable word vectors)
    model.add(Embedding(
        input_dim = embedding_matrix.shape[0],
        output_dim = embedding_matrix.shape[1],
        input_length = input_length,
        weights = [embedding_matrix],
        trainable=False))
    # Bidirectional LSTMs layer
    model.add(Bidirectional(LSTM(128, recurrent_dropout=0.1)))
    # Dropout and Dense twice
    model.add(Dropout(0.25))
    model.add(Dense(64))
    model.add(Dropout(0.3))
    model.add(Dense(1, activation='sigmoid'))
    # Give out a summary
    model.summary()
    return model

model = build_model(
    embedding_matrix=trigrams_model.wv.vectors,
    input_length=input_length)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_1 (Embedding)      (None, 150, 256)          23625472  
_________________________________________________________________
bidirectional_1 (Bidirection (None, 256)               394240    
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0         
_________________________________________________________________
dense_1 (Dense)              (None, 64)                16448     
_________________________________________________________________
dropout_2 (Dropout)          (None, 64)                0         
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 65        
=================================================================
Total params: 24,036,225
Trainable params: 410,753
Non-trainable params: 23,625,472
_________________________________________________________________
```

Before we start feeding the model with our training data set, let's split some data from the training set to build a validation set.

The idea of making this validation set for the model is that we can judge how well the model can generalize. Meaning, how well can the model able to predict on data that it's not seen while being trained.

The model does not "see" our validation set and is not in any way trained on it, but we as the architect and master of the hyperparameters tune the model according to this data. Therefore it indirectly influences the model because it directly influences our design decisions.

So here we use sklearn to split 5% of the training data for validation.

In [4]:
```
x_feed, x_valid, y_feed, y_valid = train_test_split(
    X_train_pad,
    Y_train,
    test_size=0.05,
    shuffle=True,
    random_state=42)
```

And let's feed in data and start training it!

In [5]:
```
# Here we use adam optimizer
model.compile(
    loss="binary_crossentropy",
    optimizer='adam',
    metrics=['accuracy'])

# And we train the model for 20 epochs
# (could be run as many times as needed, 20 might be enough in this case)
history = model.fit(
    x=x_feed,
    y=y_feed,
    validation_data=(x_valid, y_valid),
    batch_size=100,
    epochs=20)
```

```
Train on 23750 samples, validate on 1250 samples
Epoch 1/20
23750/23750 [==============================] - 183s 8ms/step - loss: 0.6508 -
acc: 0.6155 - val_loss: 0.5438 - val_acc: 0.7456
Epoch 2/20
23750/23750 [==============================] - 178s 8ms/step - loss: 0.5687 -
acc: 0.7145 - val_loss: 0.5190 - val_acc: 0.7648
Epoch 3/20
23750/23750 [==============================] - 176s 7ms/step - loss: 0.4996 -
acc: 0.7656 - val_loss: 0.4613 - val_acc: 0.7824
Epoch 4/20
23750/23750 [==============================] - 185s 8ms/step - loss: 0.4439 -
acc: 0.7956 - val_loss: 0.4446 - val_acc: 0.7920
Epoch 5/20
23750/23750 [==============================] - 207s 9ms/step - loss: 0.3942 -
acc: 0.8281 - val_loss: 0.4219 - val_acc: 0.8224
Epoch 6/20
23750/23750 [==============================] - 231s 10ms/step - loss: 0.3501 -
acc: 0.8477 - val_loss: 0.4065 - val_acc: 0.8312
Epoch 7/20
23750/23750 [==============================] - 252s 11ms/step - loss: 0.3153 -
acc: 0.8657 - val_loss: 0.4096 - val_acc: 0.8208
Epoch 8/20
23750/23750 [==============================] - 297s 13ms/step - loss: 0.2737 -
acc: 0.8836 - val_loss: 0.4669 - val_acc: 0.8096
Epoch 9/20
23750/23750 [==============================] - 306s 13ms/step - loss: 0.2317 -
acc: 0.9051 - val_loss: 0.4736 - val_acc: 0.8192
Epoch 10/20
23750/23750 [==============================] - 340s 14ms/step - loss: 0.1961 -
acc: 0.9202 - val_loss: 0.5664 - val_acc: 0.7968
Epoch 11/20
23750/23750 [==============================] - 341s 14ms/step - loss: 0.1723 -
acc: 0.9291 - val_loss: 0.5218 - val_acc: 0.8224
Epoch 12/20
23750/23750 [==============================] - 357s 15ms/step - loss: 0.1399 -
acc: 0.9449 - val_loss: 0.6103 - val_acc: 0.8200
Epoch 13/20
23750/23750 [==============================] - 368s 15ms/step - loss: 0.1209 -
acc: 0.9523 - val_loss: 0.6379 - val_acc: 0.8192
Epoch 14/20
23750/23750 [==============================] - 386s 16ms/step - loss: 0.0942 -
acc: 0.9629 - val_loss: 0.7227 - val_acc: 0.8176
Epoch 15/20
23750/23750 [==============================] - 400s 17ms/step - loss: 0.0904 -
acc: 0.9651 - val_loss: 0.7249 - val_acc: 0.8320
Epoch 16/20
23750/23750 [==============================] - 427s 18ms/step - loss: 0.0708 -
acc: 0.9740 - val_loss: 0.7448 - val_acc: 0.8304
Epoch 17/20
23750/23750 [==============================] - 474s 20ms/step - loss: 0.0779 -
acc: 0.9697 - val_loss: 0.7525 - val_acc: 0.8216
Epoch 18/20
23750/23750 [==============================] - 503s 21ms/step - loss: 0.0618 -
acc: 0.9756 - val_loss: 0.9215 - val_acc: 0.8072
Epoch 19/20
23750/23750 [==============================] - 496s 21ms/step - loss: 0.0552 -
acc: 0.9793 - val_loss: 0.8684 - val_acc: 0.8312
Epoch 20/20
23750/23750 [==============================] - 483s 20ms/step - loss: 0.0438 -
acc: 0.9844 - val_loss: 0.9865 - val_acc: 0.8088
```

Above we can see that we are getting a 80% accuracy with around 4% loss after 20 epoch, which is overall pretty good.

In [7]:
```
# Save our model and the training history to the dir (so we may refer it back
 later)
model.save('my_model.h5')
pickle.dump(history.history, open('resources/history', 'wb'))
```

Model performance

Before we check how our model perform, let's also transfrom the testing data:

In [8]:
```
%%time
X_test_pad = pad_sequences(
    sequences=vectorize_data(X_test, vocab=trigrams_model.wv.vocab),
    maxlen=input_length,
    padding='post')
print('Transform sentences to sequences... (done)')
```

```
Vectorize sentences... (done)
Transform sentences to sequences... (done)
Wall time: 7min 15s
```

Now, let's use our model to predict the sentiment of the movie reviews. We use the training data predictions as a comparison with the testing data predictions. Then we show both results with sklearn classification report.

In [10]:
```
%%time
# Predict using training and testing data respectively
Y_train_pred = model.predict_classes(X_train_pad)
Y_test_pred = model.predict_classes(X_test_pad)
```

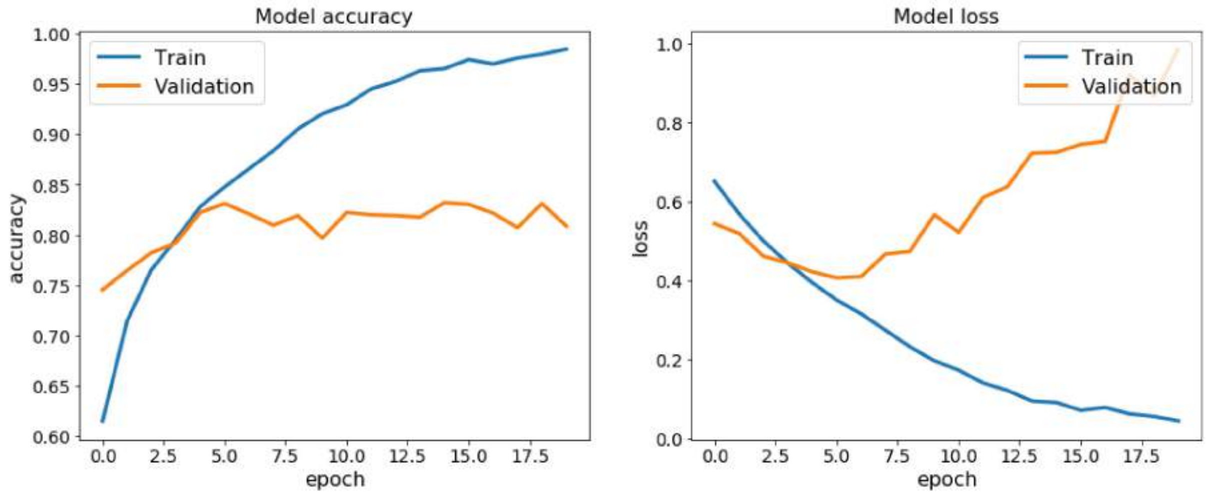
```
Wall time: 3min 1s
```

In [21]:
```
from sklearn.metrics import classification_report

# Train data report (for comparison)
print(classification_report(Y_train_pred, Y_train))
# Test data report (real thing)
print(classification_report(Y_test_pred, Y_test))
```

```
              precision    recall  f1-score   support

           0       0.99      0.97      0.98     12766
           1       0.97      0.99      0.98     12234

    accuracy                           0.98     25000
   macro avg       0.98      0.98      0.98     25000
weighted avg       0.98      0.98      0.98     25000

              precision    recall  f1-score   support

           0       0.88      0.74      0.80     14881
           1       0.69      0.85      0.76     10119

    accuracy                           0.78     25000
   macro avg       0.78      0.79      0.78     25000
weighted avg       0.80      0.78      0.79     25000
```

As we can see, we reach an accuracy of 78% with the testing dataset in the final. And let's see how the training progress goes:

In [22]:
```
fig, (axis1, axis2) = plt.subplots(nrows=1, ncols=2, figsize=(16,6))

# summarize history for accuracy
axis1.plot(history.history['acc'], label='Train', linewidth=3)
axis1.plot(history.history['val_acc'], label='Validation', linewidth=3)
axis1.set_title('Model accuracy', fontsize=16)
axis1.set_ylabel('accuracy')
axis1.set_xlabel('epoch')
axis1.legend(loc='upper left')

# summarize history for loss
axis2.plot(history.history['loss'], label='Train', linewidth=3)
axis2.plot(history.history['val_loss'], label='Validation', linewidth=3)
axis2.set_title('Model loss', fontsize=16)
axis2.set_ylabel('loss')
axis2.set_xlabel('epoch')
axis2.legend(loc='upper right')
plt.show()
```



It is still possible to try out different parameters which may lead to better result. And if possible, we can find more dataset to train that may imporve this model.

End