Submitted by:

Akanksha Bindal (903227687) abindal@gatech.edu
Ankit Khedia (903245659) akhedia3@gatech.edu

Sequential Prediction Model

Sequential event prediction refers to a wide class of problems in which a set of initially hidden events are sequentially revealed. The goal is to use the set of revealed events,to predict the remaining (hidden). Usually there is a“sequence database” of past event sequences that we can use to design the predictions. Predictions for the next event are updated each time a new event is revealed. There are many examples of sequential prediction problems. Medical conditions occur over a timeline, and the conditions that the patient has experienced in the past can be used to predict conditions that will come. Music recommender systems, e.g. Pandora, use a set of songs for which the user has revealed his or her preference to construct a suitable playlist. The playlist is modified as new preferences are revealed. Online grocery stores use the customer’s current shopping cart to recommend other items. The recommendations are updated as items are added to the basket. Recommender systems are a particularly interesting example of sequential event prediction because the predictions are expected to influence the sequence and any realistic algorithm should take this into account . Rather than a sequence of single items, the data comprise a sequence of sets of conditions. The sequential event prediction problems we consider here are different from time-series prediction problems, that one might handle with a Markov chain. Only the set of past items are useful for predicting the remaining sequence

Types of sequential prediction models

a. Recurrent Neural Networks:

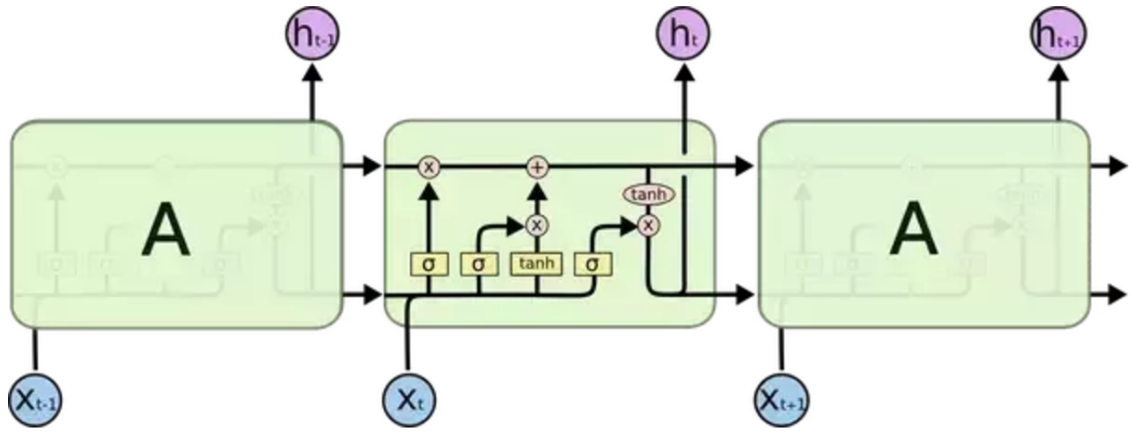
A recurrent neural network (RNN) is a class of artificial neural network where connections between units form a directed cycle. This creates an internal state of the network which allows it to exhibit dynamic temporal behavior. Unlike feedforward neural networks, RNNs can use their internal memory to process arbitrary sequences of inputs. This makes them applicable to tasks such as unsegmented connected handwriting recognition or speech recognition. Since RNNs shows a temporal behavior, it can be very well used for sequence prediction based on past seen inputs.

b. LSTM Neural Networks:

A powerful type of neural network designed to handle sequence dependence is called recurrent neural networks. The Long Short-Term Memory network or LSTM network is a type of recurrent neural network used in deep learning because very large architectures can be successfully trained. The Long Short-Term Memory network, or LSTM network, is a recurrent neural network that is trained using Backpropagation Through Time and overcomes the vanishing gradient problem. As such, it can be used to create large recurrent networks that in turn can be used to address difficult sequence problems in machine learning and achieve state-of-the-art results. Instead of neurons, LSTM networks have memory blocks that are connected through layers. A block has components that make it smarter than a classical neuron and a memory for recent sequences. A block contains gates that manage the block's state and output. A block operates upon an input sequence and each gate within a block uses the sigmoid activation units to control whether they are triggered or not, making the change of state and addition of information flowing through the block conditional. There are three types of gates within a unit: Forget Gate: conditionally decides what information to throw away from the block. Input Gate: conditionally decides which values from the input to update the memory state. Output Gate: conditionally decides what to output based on input and the memory of the block. Each unit is like a mini-state machine where the gates of the units have weights that are learned during the training procedure. You can see how you may achieve sophisticated learning and memory from a layer of LSTMs, and it is not hard to imagine how higher-order abstractions may be layered with multiple such layers. LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn.

In [1]:
```
from IPython.display import Image
Image(filename='images/lstm.png')
```

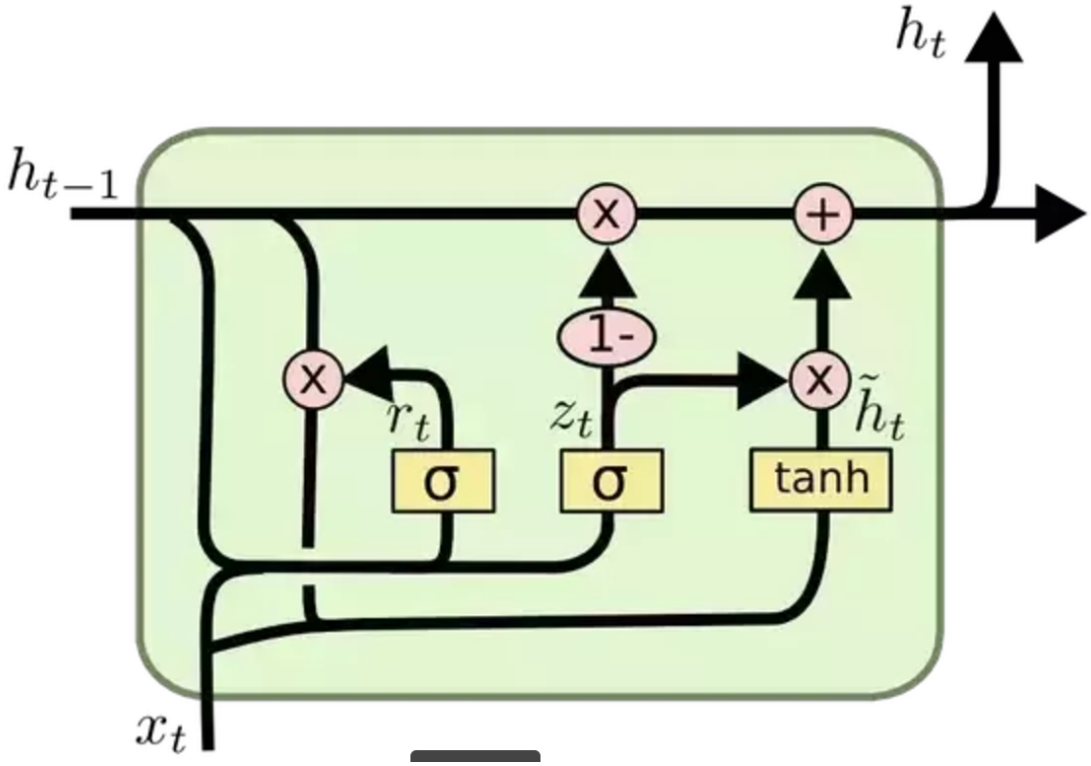
Out[1]:



c. GRU Neural Networks: The idea behind a GRU layer is quite similar to that of a LSTM layer. A GRU has two gates, a reset gate , and an update gate . Intuitively, the reset gate determines how to combine the new input with the previous memory, and the update gate defines how much of the previous memory to keep around. If we set the reset to all 1's and update gate to all 0's we again arrive at our plain RNN model. The basic idea of using a gating mechanism to learn long-term dependencies is the same as in a LSTM, but there are a few key differences: A GRU has two gates, an LSTM has three gates.GRUs don't possess and internal memory that is different from the exposed hidden state. They don't have the output gate that is present in LSTMs. The input and forget gates are coupled by an update gate and the reset gate is applied directly to the previous hidden state. Thus, the responsibility of the reset gate in a LSTM is really split up into both and . We don't apply a second nonlinearity when computing the output.

In [4]:
```
from IPython.display import Image

Image(filename='images/gru.png')
```

Out[4]:



Why is language detection important?

Language identification has become increasingly important, as more and more textual data is making its way online. When processing multilingual document collections, appropriate language annotations can be used to bootstrap shallow machine translation, parts-of-speech tagging, topic labeling or information retrieval. Automatic "language guessing" systems have been described in the past, achieving very high accuracy

State of Art

Ngrams

Most of the literature in automatic language identification uses some variation of character-based n-gram model. Cavnar and Trenkle [1994] use a language n-gram profile, built upon the most frequent character n-grams in a text, with n running from 1 to 5. Then, they use an ad hoc “out-of-place” ranking distance measure to classify a newly arrived text into one of the existing profiles. They report 90% to 100% precision using a 300 n-grams profile, in detecting a text of 300 characters, and 95% to 100% for longer texts. Dunning [1994] states that real-life applications need a precise language classification for texts up to 20 characters, showing the practical limitations of methods based on characteristic sequences and common words. He also criticises the approach of Cavnar and Trenkle [1994] since it depends on tokenisation and uses an empirical measure whose statistical properties are difficult to derive. Therefore, he proposes a method based on Markov chains and Bayesian models, similar to the one proposed here, that uses a log-probability sum as distance measure between a text and a language model.

A comparison of trigram-based and short-word-based techniques for language identification is described by Grefenstette [1995]. He uses a non-smoothed MLE probability distribution, and then compares one technique in which the distribution runs over trigrams and another in which the distribution runs over “common words” (5 letters or less, frequent words). The results show that trigrams are better for short sentences and that when larger sentences are considered, both methods perform equally well. A short survey on language identification containing some pointers to other related work can be found in Muthusamy and Spitz [1997]. Given that language identification is an extremely simple problem compared to other NLP fields, it has the status of a “solved problem” that could be thought in undergraduate courses [McNamee, 2005]. The solution presented in this report aims at achieving state-of-the-art accuracy, that is to say correctly identify near to 100% of the sentences in a test set even for short sentences, using a relatively simple approach.

How do Ngrams work?

N-Gram is an N-character slice of a longer string. Typically one would slice the string into a set of overlapping N-grams. In our system we will use N-Grams of various lengths simultaneously. Note: We would want this to be configurable. We will also append blanks to the beginning and end of strings in order to help with matching the beginning-of-word and end-of-word situations. We will represent this using the _ character. Given the word TEXT we would obtain the following N-Grams

bi-grams *T, TE, EX, XT, T* tri-grams *TE, TEX, EXT, XT*, T **quad-grams *TEX, TEXT, EXT*, XT**, T___ In general a string of length kk, padded with blanks, will have k+1k+1 bi-grams, k+1k+1 tri-grams, k+1k+1 quad-grams and so on.

Human language has some words which are used more than others. For example you can imagine that in a document the word the will occur more frequently than the word aardvark. Moreover there is always a small set of words which dominates most of the language in terms of frequency of use. This is true both for words in a particular language and also for words in a particular category. Thus words which appear in a sporty document will be different from the words that appear in a political document and words which are in the English language will obviously be different from words which are in the French language.

Well it turns out that these small fragments of words (N-grams) also obey (approximately) the same property. Thus given a document in a particular language we will always find a set of N-grams which dominate - you can visualise these N-grams as a language palette from which words are formed. These set of N-grams will be different for each language. If we use this language detector on small fragments of text we are less susceptible to noise hence making our language detection more resilient.

From other language detection frameworks implemented we know that we should expect the following results:

The top 300 or so N-grams are almost always highly correlated to the language. Thus the language profile of a sporty document will be very similar to the language profile generated from a political document in the same language. This gives us confidence that if we train the system on the Declaration of Human Right we will still be able to classify documents to the correct language even though they might have completely different topics. The highest ranking N-grams are mostly uni-grams and simply reflect the distribution of characters in a language. After uni-grams N-grams representing prefixes and suffixes should be the most popular. Starting at around rank 300 or so, an N-gram frequency profile begins to become specific to the topic.

Markov Models

Hidden Markov Models (HMM) are commonly used in spoken language identification (Zissman and Singer, 1994; Lamel and Gauvain, 1994) but they are also used for written language (Xafopoulos et al., 2004; Ueda and Nakagawa, 1990). Nevertheless, this task can be performed with visible Markov Models (MM) which is the first of the three systems compared in this work. For each language that the system must know about, a model is trained from a text corpora, and stored for later comparison with unidentified text. In these models each state si represents a character trigram c1c2c3. Thus, the parameters of the MM are the transition probability and the initial probability: Aij: transition probability from the state i to the state j . As from one state c1c2c3 it is only possible to go to another state c2c3c4(' the probability transition is, in fact, the 4-gram probability P(c4/c1c2c3) Pi(i): probability of starting a sequence in state i . These probabilities are estimated via MLE, i.e. computing the relative frequency of each transition or initial state in the training data: Very simple smoothing is performed, counting a fixed small number of occurrences for trigrams not appearing in the training set. To classify a new text, the system computes the sequence probability using each language model it has been trained for. Then the system chooses the language that gives the largest probability.

Problem Definition

We have corpus for text in different languages. We are given a test string and we need to find the language of the given string.

Approach

We plan to use LSTM models for the purpose. LSTMs are very good at generating text given some seed. They generate text character by character. We predict the character by finding out probability for each of the characters after a sequence and predict the character which has the highest probability to occur after a given sequence.

We use the same LSTM but with some variations. Given a seed value, we find out the probability of the characters of our test strings one by one and multiply them to find the overall probability for each characters.

In [1]:
```
from __future__ import print_function
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.layers import LSTM
from keras.optimizers import RMSprop
from keras.utils.data_utils import get_file
import numpy as np
import random
import sys
from sklearn.cross_validation import train_test_split
```

```
Using TensorFlow backend.
/Users/akankshabindal/anaconda/envs/env/lib/python2.7/site-packages/skl
earn/cross_validation.py:44: DeprecationWarning: This module was deprec
ated in version 0.18 in favor of the model_selection module into which
 all the refactored classes and functions are moved. Also note that the
interface of the new CV iterators are different from that of this modul
e. This module will be removed in 0.20.
  "This module will be removed in 0.20.", DeprecationWarning)
```

In [2]:
```
#reading corpus
with open('subset1/eng.txt','rt') as f:
    text1 = f.read().lower()
with open('subset1/frn.txt','rt') as f:
    text2 = f.read().lower()
print('corpus length:', len(text1))
print('corpus length:', len(text2))
```

```
corpus length: 10746
corpus length: 12009
```

We designed 2 LSTM for English and French language with 128 nodes and trained it with corresponding corpus. We divided corpus into 80-20 train test set. The training set was formed using 40 sequence of characters from corpus and 1 following character. The step size means that we skip that many characters for picking up next sentence for test or train sets We have used stepsize of 1 for training corpus and step size of 20 for test. This helps in getting greater number of training sets The trained neural network could predict probabilities of each character given previous 40 characters(seed). We selected 100 5-character strings from test set randomly to form our test strings and then tried to find out Pr(string|french) and Pr(string|english) and assign a string to that class which has higher probability. We tried many different ways to find out the probability Pr(string|model). Our models needs a seed of 40 character to predict the probability of next character. So, we used the 40 characters preceding the test string in the corpus as our seed for finding prediction probability of first character of the string. Similarily, we use last 39 characters and the first character of sting for finding prediction probability of second character of the string and so on. We multiply these probabilities to get final values of probability Pr(string|model). We tried varying the seed accordingly in different experiments to see the accuracy as discussed in next sections.

In [5]:
```
#generating english model
chars_eng = sorted(list(set(text1)))
chars_fr = sorted(list(set(text2)))
chars = sorted(list(set(chars_eng + chars_fr)))
print('total chars:', len(chars))
char_indices = dict((c, i) for i, c in enumerate(chars))
indices_char = dict((i, c) for i, c in enumerate(chars))


##training set prepare
maxlen= 40
step = 1
sentences_eng_tr = []
next_chars_eng_tr = []
for i in range(0, int(0.8*len(text1)) - maxlen, step):
    sentences_eng_tr.append(text1[i: i + maxlen])
    next_chars_eng_tr.append(text1[i + maxlen])

print('nb sequences:', len(sentences_eng_tr))
##test set prepare
sentences_eng_test = []
string_eng_test= []
step_test=20
for i in range(int(0.8*len(text1)) - maxlen,len(text1) - maxlen-step_tes
t, step_test):
    sentences_eng_test.append(text1[i: i + maxlen])
    string_eng_test.append(text1[i+maxlen:i+maxlen+5])
print('nb sequences:', len(sentences_eng_test))

print('Vectorization...')
X_eng = np.zeros((len(sentences_eng_tr), maxlen, len(chars)), dtype=np.b
ool)
y_eng = np.zeros((len(sentences_eng_tr), len(chars)), dtype=np.bool)
for i, sentence in enumerate(sentences_eng_tr):
    for t, char in enumerate(sentence):
        X_eng[i, t, char_indices[char]] = 1
    y_eng[i, char_indices[next_chars_eng_tr[i]]] = 1


# build the model: a single LSTM
print('Build model...')
model = Sequential()
model.add(LSTM(128, input_shape=(maxlen, len(chars))))
model.add(Dense(len(chars)))
model.add(Activation('softmax'))

optimizer = RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metr
ics = ['accuracy'])
```

```
total chars: 44
nb sequences: 8556
nb sequences: 107
Vectorization...
Build model...
```

In [6]:
```
#generating french model



##training set prepare
maxlen= 40
step = 1
sentences_fr_tr = []
next_chars_fr_tr = []
for i in range(0, int(0.8*len(text2)) - maxlen, step):
    sentences_fr_tr.append(text2[i: i + maxlen])
    next_chars_fr_tr.append(text2[i + maxlen])

print('nb sequences:', len(sentences_fr_tr))
##test set prepare
sentences_fr_test = []
string_fr_test= []
step_test=20
for i in range(int(0.8*len(text2)) - maxlen,len(text2) - maxlen, step_te
st):
    sentences_fr_test.append(text2[i: i + maxlen])
    string_fr_test.append(text2[i+maxlen:i+maxlen+5])
print('nb sequences:', len(sentences_fr_test))

print('Vectorization...')
X_fr = np.zeros((len(sentences_fr_tr), maxlen, len(chars)), dtype=np.boo
l)
y_fr = np.zeros((len(sentences_fr_tr), len(chars)), dtype=np.bool)
for i, sentence in enumerate(sentences_fr_tr):
    for t, char in enumerate(sentence):
        X_fr[i, t, char_indices[char]] = 1
    y_fr[i, char_indices[next_chars_fr_tr[i]]] = 1


# build the model: a single LSTM
print('Build model...')
model2 = Sequential()
model2.add(LSTM(128, input_shape=(maxlen, len(chars))))
model2.add(Dense(len(chars)))
model2.add(Activation('softmax'))

optimizer = RMSprop(lr=0.01)
model2.compile(loss='categorical_crossentropy', optimizer=optimizer, met
rics=["accuracy"])
```

```
nb sequences: 9567
nb sequences: 121
Vectorization...
Build model...
```

In [8]:
```
#saving the test strings for running other models
import csv
sentences_fr_test =sentences_fr_test[:100]
string_fr_test=string_fr_test[:100]
sentences_eng_test =sentences_eng_test[:100]
string_eng_test=string_eng_test[:100]

sentences_test = sentences_fr_test + sentences_eng_test
string_test = string_fr_test + string_eng_test

with open("test_50.csv", 'wt') as f:
    w = csv.writer(f, dialect='excel')
    w.writerow(string_test)
```

In [7]:
```
def vectorize(sentence, chars , char_indices ):
    X = np.zeros((1, maxlen, len(chars)), dtype=np.bool)
    for t, char in enumerate(sentence):
        X[0, t, char_indices[char]] = 1
    return X
```

In [8]:
```
#return an array of probability of predicting each character of test str
ing
def get_scores_for_model(seed,next_string, model,chars,char_indices):
    sentence =seed
    prob_char=0.0
    x = np.zeros((5, maxlen, len(chars)))
    x[0] = vectorize(seed,chars,char_indices)
    x[1] = vectorize(seed[1:]+next_string[:1],chars,char_indices)
    x[2] = vectorize(seed[2:]+next_string[:2],chars,char_indices)
    x[3] = vectorize(seed[3:]+next_string[:3],chars,char_indices)
    x[4] = vectorize(seed[4:]+next_string[:4],chars,char_indices)
    probs_term = []
    for i in range(5):
        preds = model.predict(x[i].reshape(1,40,44), verbose=0)[0]
        preds = np.log(preds)
        probs_term.append(preds[char_indices[next_string[i]]])

    return probs_term
```

In [9]:
```
import csv
# train the model, output generated text after each iteration
for iteration in range(1, 2):
    print()
    #print('-' * 50)
    print('Iteration', iteration)
    a1=model.fit(X_eng, y_eng,
              batch_size=128,
              epochs=20)
    model_eng_json = model.to_json()
    with open("model/model_eng_"+str(iteration)+".json", "w") as json_fi
le:
        json_file.write(model_eng_json)
    # serialize weights to HDF5
    model.save_weights("weights/model_eng_"+ str(iteration) +".h5")
    print("Saved model to disk")

    sentences_fr_test =sentences_fr_test[:100]
    string_fr_test=string_fr_test[:100]
    sentences_eng_test =sentences_eng_test[:100]
    string_eng_test=string_eng_test[:100]

    #sentence=
    prob_english=[]
    for i in range(len(sentences_eng_test)):
        prob_english.append(get_scores_for_model(sentences_eng_test[i],
string_eng_test[i], model,chars, char_indices))
    for i in range(len(sentences_fr_test)):
        prob_english.append(get_scores_for_model(sentences_fr_test[i], s
tring_fr_test[i], model, chars, char_indices))
    #sentence = text1[start_index: start_index + maxlen]
    #print (prob_english)

    with open("prob_eng/output_prob_" + str(iteration) + ".csv", 'wt') a
s f:
        w = csv.writer(f, dialect='excel')
        for row in prob_english:
            w.writerow(row)
    with open("model/loss_eng" + ".csv", 'a') as f:
        w = csv.writer(f, dialect='excel')
        w.writerow(a1.history['loss'])

        #print()
```

```
Iteration 1
Epoch 1/20
8556/8556 [==============================] - 17s - loss: 2.8177 - acc:
 0.2114
Epoch 2/20
8556/8556 [==============================] - 16s - loss: 2.2196 - acc:
 0.3663
Epoch 3/20
8556/8556 [==============================] - 18s - loss: 1.8978 - acc:
 0.4494
Epoch 4/20
8556/8556 [==============================] - 18s - loss: 1.6614 - acc:
 0.5144
Epoch 5/20
8556/8556 [==============================] - 18s - loss: 1.4858 - acc:
 0.5630
Epoch 6/20
8556/8556 [==============================] - 18s - loss: 1.3195 - acc:
 0.6030
Epoch 7/20
8556/8556 [==============================] - 18s - loss: 1.1911 - acc:
 0.6378
Epoch 8/20
8556/8556 [==============================] - 19s - loss: 1.0780 - acc:
 0.6663
Epoch 9/20
8556/8556 [==============================] - 19s - loss: 0.9691 - acc:
 0.6967
Epoch 10/20
8556/8556 [==============================] - 18s - loss: 0.8899 - acc:
 0.7255
Epoch 11/20
8556/8556 [==============================] - 19s - loss: 0.8073 - acc:
 0.7528
Epoch 12/20
8556/8556 [==============================] - 19s - loss: 0.7331 - acc:
 0.7686
Epoch 13/20
8556/8556 [==============================] - 16s - loss: 0.6771 - acc:
 0.7888
Epoch 14/20
8556/8556 [==============================] - 15s - loss: 0.6199 - acc:
 0.8072
Epoch 15/20
8556/8556 [==============================] - 16s - loss: 0.5750 - acc:
 0.8178
Epoch 16/20
8556/8556 [==============================] - 17s - loss: 0.5426 - acc:
 0.8310
Epoch 17/20
8556/8556 [==============================] - 19s - loss: 0.4938 - acc:
 0.8454
Epoch 18/20
8556/8556 [==============================] - 19s - loss: 0.4747 - acc:
 0.8483
Epoch 19/20
8556/8556 [==============================] - 19s - loss: 0.4453 - acc:
```

```
 0.8554
Epoch 20/20
8556/8556 [==============================] - 19s - loss: 0.4194 - acc:
 0.8709
Saved model to disk
```

In [10]:
```
import csv
# train the model, output generated text after each iteration


for iteration in range(1, 2):
    print()
    #print('-' * 50)
    print('Iteration', iteration)
    a2=model2.fit(X_fr, y_fr,
              batch_size=128,
              epochs=20)
    model_fr_json = model2.to_json()
    with open("model/model_fr_"+str(iteration)+".json", "w") as json_fil
e:
        json_file.write(model_fr_json)
    # serialize weights to HDF5
    model2.save_weights("weights/model_fr_"+ str(iteration) +".h5")
    print("Saved model to disk")


    sentences_fr_test =sentences_fr_test[:100]
    string_fr_test=string_fr_test[:100]
    sentences_eng_test =sentences_eng_test[:100]
    string_eng_test=string_eng_test[:100]
    #sentence=
    prob_fr=[]
    for i in range(len(sentences_eng_test)):
        prob_fr.append(get_scores_for_model(sentences_eng_test[i], strin
g_eng_test[i], model2,chars, char_indices))
    for i in range(len(sentences_fr_test)):
        prob_fr.append(get_scores_for_model(sentences_fr_test[i], string
_fr_test[i], model2, chars, char_indices))
    #sentence = text1[start_index: start_index + maxlen]
    #print (prob_english)
    with open("prob_fr/output_prob_" + str(iteration) + ".csv", 'wt') as
 f:
        w = csv.writer(f, dialect='excel')
        for row in prob_fr:
            w.writerow(row)
    with open("model/loss_fr" + ".csv", 'a') as f:
        w = csv.writer(f, dialect='excel')
        w.writerow(a2.history['loss'])

    #print()
```

```
Iteration 1
Epoch 1/20
9567/9567 [==============================] - 20s - loss: 2.6876 - acc:
 0.2336
Epoch 2/20
9567/9567 [==============================] - 19s - loss: 2.1182 - acc:
 0.3607
Epoch 3/20
9567/9567 [==============================] - 19s - loss: 1.8451 - acc:
 0.4386
Epoch 4/20
9567/9567 [==============================] - 19s - loss: 1.6625 - acc:
 0.4839
Epoch 5/20
9567/9567 [==============================] - 16s - loss: 1.5075 - acc:
 0.5278
Epoch 6/20
9567/9567 [==============================] - 16s - loss: 1.3679 - acc:
 0.5709
Epoch 7/20
9567/9567 [==============================] - 16s - loss: 1.2498 - acc:
 0.6033
Epoch 8/20
9567/9567 [==============================] - 16s - loss: 1.1348 - acc:
 0.6449
Epoch 9/20
9567/9567 [==============================] - 17s - loss: 1.0332 - acc:
 0.6708
Epoch 10/20
9567/9567 [==============================] - 17s - loss: 0.9459 - acc:
 0.6959
Epoch 11/20
9567/9567 [==============================] - 21s - loss: 0.8725 - acc:
 0.7196
Epoch 12/20
9567/9567 [==============================] - 18s - loss: 0.7940 - acc:
 0.7428
Epoch 13/20
9567/9567 [==============================] - 18s - loss: 0.7319 - acc:
 0.7611
Epoch 14/20
9567/9567 [==============================] - 18s - loss: 0.6793 - acc:
 0.7761
Epoch 15/20
9567/9567 [==============================] - 19s - loss: 0.6354 - acc:
 0.7927
Epoch 16/20
9567/9567 [==============================] - 18s - loss: 0.5767 - acc:
 0.8146
Epoch 17/20
9567/9567 [==============================] - 19s - loss: 0.5534 - acc:
 0.8188
Epoch 18/20
9567/9567 [==============================] - 18s - loss: 0.5230 - acc:
 0.8283
Epoch 19/20
9567/9567 [==============================] - 19s - loss: 0.4909 - acc:
```

```
 0.8401
Epoch 20/20
9567/9567 [==============================] - 18s - loss: 0.4771 - acc:
 0.8429
Saved model to disk

/Users/akankshabindal/anaconda/envs/env/lib/python2.7/site-packages/ipy
kernel/__main__.py:14: RuntimeWarning: divide by zero encountered in lo
g
```

In [124]:
```
#Load existing model from disk for retraining from some checkpoint in ca
se of model breakdown


from keras.models import model_from_json
json_file = open('model/model_eng_2.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
model3 = model_from_json(loaded_model_json)
# load weights into new model
model3.load_weights("weights/model_eng_2.h5")
print("Loaded model from disk")
optimizer = RMSprop(lr=0.01)
model3.compile(loss='categorical_crossentropy', optimizer=optimizer)
# evaluate loaded model on test data
#loaded_model.compile(loss='binary_crossentropy', optimizer='rmsprop', m
etrics=['accuracy'])
```

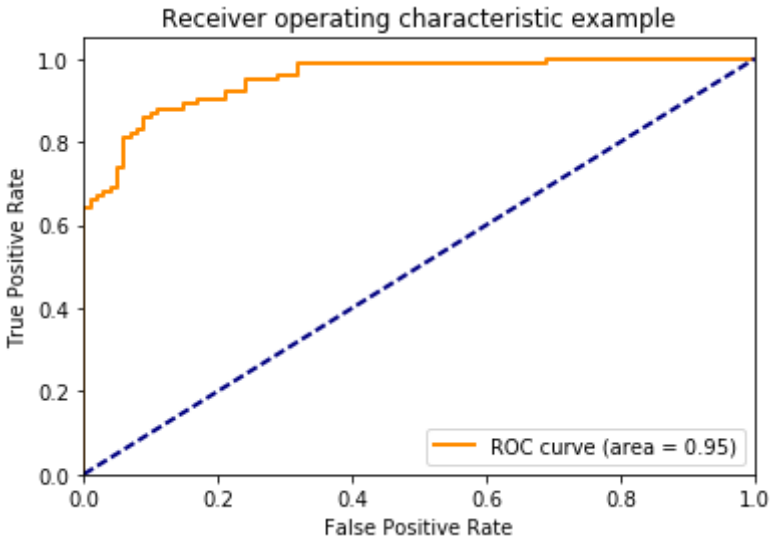
```
Loaded model from disk
```

In [193]:
```
#Load existing model from disk for retraining from some checkpoint in ca
se of model breakdown



from keras.models import model_from_json
json_file = open('model/model_eng_1.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
model3 = model_from_json(loaded_model_json)
model3.load_weights("eng.h5")
```

In [11]:
```
#Code to predict french(1)/english(0) based on probability values stored
 for each test string

def predict_language(prob_english, prob_fr):
    y = [1] * len(prob_english)
    probs_eng = []
    probs_frn = []
    for i in range(len(prob_english)):
        sum_english = 0
        sum_fr = 0
        for j in range(len(prob_english[0])):
            sum_english += float(prob_english[i][j])
            sum_fr += float(prob_fr[i][j])
        sum_english = np.exp(sum_english)
        sum_fr = np.exp(sum_fr)
        if sum_english < sum_fr:
            y[i] = 0
        if sum_fr == 0.0:
            sum_fr = 0.0000000001
            #print(str(i) + "Zero")
        probs_eng.append(sum_english)
        probs_frn.append(sum_fr)
    return y, probs_eng, probs_frn
```

In [13]:
```
import csv
for iter in range(1,2):
    print("----------------- Iteration: " + str(iter) + " --------------
------")
    prob_english=[]
    with open("prob_eng/output_prob_" + str(iter) + ".csv", 'rt') as f:
            w = csv.reader(f, delimiter=',')
            prob_english = list(w)

    prob_fr=[]
    with open("prob_fr/output_prob_" + str(iter) + ".csv", 'rt') as f:
            w = csv.reader(f, delimiter=',')
            prob_fr = list(w)

    #print((prob_english), len(prob_fr))
    true_labels = ([1] * 100) + ([0] * 100)
    #print(true_labels.shape)

    from sklearn.metrics import roc_auc_score, confusion_matrix, accurac
y_score, auc, roc_curve
    #try:
    pred_y, logprob_english, logprob_fr = predict_language(prob_english,
 prob_fr)
    #except:
    #    continue
    #x = [e for i, e in enumerate(logprob_fr) if e == 0]
    #print(x)
    print(min(logprob_fr), max(logprob_fr))
    y_hat = np.array([a/b for (a, b) in zip(logprob_english,
logprob_fr)])
    #roc_auc_score(true_labels, y_hat)

    print (confusion_matrix(true_labels, pred_y))
    print (accuracy_score(true_labels, pred_y))
    #print (y_hat)
    try:
        fpr, tpr, _ = roc_curve(true_labels, y_hat)
    except:
        continue
    roc_auc = auc(fpr, tpr)

    import matplotlib.pyplot as plt
    plt.figure()
    lw = 2
    plt.plot(fpr, tpr, color='darkorange',
             lw=lw, label='ROC curve (area = %0.2f)' % roc_auc)
    plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic example')
    plt.legend(loc="lower right")
    plt.show()
```

```
----------------- Iteration: 1 --------------------
2.65220778046e-26 0.908649133562
[[90 10]
 [14 86]]
0.88
```



Challenges Faced

It may be expected that when distinguishing languages belonging to the same philogenetic family (e.g. romance languages, Germanic languages) the precision of the systems will be lower. If these languages also share sociocultural environment, the task will be even harder. As French and English are both Roman languages, the model needs to be robust to detect the nuances between the two languages.

We initially tried multiplying probabilities of each of the character given previous characters and then taking the log for y_hat calculation. However, due to arithmetic overflow (multiplying two extremely small numbers), our accuracy was not increasing beyond 56% with ROC of 50. So we took the log at each character probability and then proceeded to add these values to determine the final probability for each test string. This method drastically improved our accuracy and ROC-AUC values.

Results- Is this model good?

We ran it for 20 epochs and obtained ROC-Auc value of 0.95 and accuracy value of 88% Confusion matrix: [[90 10] [14 86]]

It manages to distinguish between English and French for small sized text strings (5 character strings). This is a good discriminative model as the script of both the languages is same (Roman Script) and it still manages to give decent accuracy.

What are at least three alternatives to language detection that you can think of or find on the internet? What are the pros and cons of each approach?

1. Ngrams

We refer the Ngram Language profiling blog as described in the project description. One of the most important parts of an N-Gram language detector is the creation of language profiles. There are many datasets out there which can be used for this purpose, but for this post we will use the Universal Declaration of Human Rights Database since i) it has been translated into over 460 different languages and dialects and ii) it's free. We run experiments on the Ngram Language Profiling of the UDHR database in Experiment 15 We find that N grams leads to 80 percent accuracy on the same test set as used in our base example [[83 17] [23 77]] 0.8

Pro:

1. Language-independent method and no prior knowledge of the languages is needed
2. Building n-gram models may be computationally expensive if we have a large language set for which we need to build character models.
3. [Rapport] shows us that a simpler model, i.e. a character model instead of a word model, is better fit to the task of language detection.
4. Character n-gram models can also be applied to resource-poor languages that do not dispose of large corpora, as a model built upon some hundred sentences is already very accurate
5. Manning and Sch¨utze [1999] affirm that a trigram model performs as well as a human in predicting the next word of a text and Dunning [1994] also uses trigrams for language identification.
6. Bigram models could also be used without dramatically decreasing precision, and that using 4-gram and 5-gram models does not improve the accuracy of the method.

Cons:

1. Precision is very low for the smallest texts but rises as the amount of text to classify becomes larger, becoming comparable to the precision of the other systems[11]
2. N-grams only work because letters are not drawn iid in a language

2. Trigrams and character set heuristic:

We use the franc tool of natural language detection which is also based on the UDHR, the most translated document in the world. Franc is a derivate from the guess language which attempts to determine the natural language of a selection of Unicode (utf-8) text. Guess_language uses heuristics based on the character set and trigrams in a sample text to detect the language. It works better with longer samples and will be confused if the sample text includes markup such as HTML tags. Supported languages franc-min 81 franc 186 franc-all 384 Processing involved: Create a list of trigrams in content sorted by frequency Convert to normalized unicode.Remove non-alpha chars and compress runs of spaces

Pros:

1. We find franc gives incredible results of upto 99.5 % accuracy. This could be because franc is trained on large corpus size and the test string length is also much longer. If we pass shorter length strings to franc, it ouputs 'UNDEFINED'
2. Manning and Sch¨utze [1999] affirm that a trigram model performs as well as a human in predicting the next word of a text and Dunning [1994] also uses trigrams for language identification.

Cons:

1. franc supports many languages, so make sure to pass it big documents, to get reliable results. it doesn't work for smaller strings. So our test set was modified to 40 character strings instead of the initial 5 character string.
2. It supports only languages which have > 1Million speakers. So training corpus had to be large.

3. BAG of WORDS

Bag-of-words, depends on searching through a large dictionary and essentially doing template matching.

Pros:

Computationally faster for table look-up but memory intensive.

Cons:

There are two main drawbacks here:

1. each language would have to have an extensive dictionary of words on file, which would take a relatively long time to search through, and
2. bag-of-words will fail if none of the words in the training set are included in the testing set.
3. A bag-of-words classifier, would need a full dictionary for EACH language in order to guarantee that a language could be detected based on whichever sentence it was given
4. With words, you have no information at all when given a word not in the dictionary, while with letter N-grams you often have at least a few useful letter combinations within that word.

4. HMM:

Ergodic, continuous-observation, hidden Markov models (HMMs) were used to perform automatic language classification and detection. State observation probability densities were modeled as tied Gaus- sian mixtures. The algorithm was evaluated on four multilanguage speech databases: a three language subset of the Spoken Language Library, a three language subset of a five language Rome Laboratory database, the 20 language CCITT database, and the ten language OGI telephone speech database. The paper shows English vs French discriminative accuracy

to be 83 % and 70 % for the two different types of models used, [Lincoln] and [Muthuswamy]. As the multistate HMMs require training more parameters than the single state HMM, it is possible that the amount of training data available was simply insufficient for the multistate HMM. However, it is also likely that the contribution of transition probabilities to the forward decoding calculation was dwarfed by the contribution of the observation likelihoods. Experiments that enhanced the contribution of the transition probabilities by using variable frame rate analysis to reduce the observation rate had little effect on per- formance. Better training techniques that strike a better balance between static and transitional information should be the subject of future research.

Pros:

Able to classify shorter text woth more precision. Probably because its global sequence probability optimisation captures language features (length of the words, frequent words or stems) that can not be dealt with by the other systems. This system is the fastest, performing the classification about 4 times faster than ngrams [11] Because the Lincoln system requires no language-specific phonological knowledge or hand-labeled training data, it is easily extended to new languages. Future efforts should be focused on determining whether such simple statistical approaches to language ID can be refined, or whether systems incorporating sophisticated phonological knowledge are required to improve performance.

Cons:

Generally, performance of a single state HMM (i.e, a static Gaussian mixture classifier comparable to the multistate HMMs, indicating that sequential modeling capabilities of HMMs were not exploited.

5 ways of improving accuracy

We consider the experiment with changing START string to blank spaces as the baseline. We obtained ROC-Auc value of 80% and accuracy value of 74.5% Some ways of improving accuracy: The experiments for each of these are conducted in the Extra Credit section below along with the results for each experiment.

1. Varying number of iterations: By increasing the number of iterations we expect the LSTM model to better differentiate between the languages. This is because the loss corresponding to each model decreases and accuracy eventually improves. (ROC-Auc value of 91% and accuracy value of 85%)
2. Increasing LSTM layers: We expect the model to become more robust to inputs of different languages and model the nuances between languages. (ROC-Auc value of 93% and accuracy value of 84.5%)
3. Using GPU for training:Handles larger datasets with better efficacy. We can train large datasets for more iterations and it could achieve better results in lesser time.
4. Increasing test string size: It is difficult to ascertain the language of smaller strings as the characters are Romanic. If we increase the test string size there is greater chance of increasing the precision of our model as language specific content is more. (ROC-Auc value of 94% and accuracy value of 85%)
5. Decreasing NODE SIZE: We can intuitively reason that increasing the node sizes may lead to overfitting as the corpus size is very small. (ROC-Auc value of 96% and accuracy value of 88.5%)
6. Change optimisation(Adagrad): We can think that changing the method of convergence from RMS to AdaGrad or some other method of optimisation might lead to more robustness in our final model. (ROC-Auc value of 95% and accuracy value of 86.5%)
7. Increasing corpus size: The current literature on language detection models work with larger corpus of data. So we reason that increasing the corpus size could lead to much better discrimination among languages expecially for the all languages model experiment.
8. Decreasing training set "sentence length" dimension: We find that decreasing sentence length in the training set from 40 to 5 increases accuracy. This is expected as decreasing this dimension leads to an increase in corpus size of the training set. More data is always better!!!! (ROC-Auc value of 97% and accuracy value of 89.5%)

Some inferences after the Extra credits part and confirming with [11]: ¶

1. Influence of the train set size is not important when this size is bigger than approximately 50 kwords [Experiment 20]
2. The amount of text to classify is crucial, but it is not necessary to have very long texts to achieve a good precision. For texts over 500 characters, all the systems get a precision higher than 95%, and for texts of 5000 characters the precision is higher than 99% with all systems, reaching 100% in many cases
3. It is important to highlight that for small texts there is a big difference (Franc is unable to identify 5 character strings)
4. The more languages the system has to recognise, the less precision it will have.
5. It may be expected that when distinguishing languages belonging to the same philogenetic family (e.g. romance languages, Germanic languages) the precision of the systems will be lower. If these languages also share sociocultural environment, the task will be even harder. As French and English are both Roman languages, the model needs to be robust to detect the nuances between the two languages.

Future work:

Some further work that could be realized is adapting the systems to recognise multilingual texts and to detect intra–document language changes. It can be very useful when dealing with systems applied to mail or news processing where there are usually insertions of languages different from the main one (replying a mail in another language, quoting someone...). This phenomenon happens very often with languages that share the same social environment, specially in multilingual regions. Furthermore, it could be interesting to study the behaviour of the systems when the text to classify comes from a noisy source and contains some contextual errors. This errors could arise from an OCR system or from the unsupervised typewriting of an e-mail, among others. In addition, the presented experiments have been performed under a closed–world assumption (i.e. all possible languages for input text are known to the system). This may be enough for applications restricted to a bilingual or multilingual environment, but when moving to an unrestricted domain (e.g. Internet) the possibility that the input text is written in a language unknown to the system should be considered. Finally, since the tested systems tend to fail when distinguishing similar languages (e.g. Spanish and Catalan), further research could be done to solve these cases, maybe in the line including in the system the ability to deal with some specific morphological features.

Extra Credits:

Experiments Performed

All the following experiments are in the code/experiments folder

Experiment 1: Varying number of iterations(Exp_1_iter_20.ipynb)

We changed number of iterations from 1 to 20 and recorded no improvement over subsequent iterations as the accuracy falls from 89 to lowest 79 and gradually improves to 85. We obtained final ROC-Auc value of 91% and accuracy value of 85%

Experiments with different test/training sets

Experiment 2: Change training set dimension from 40 to 5(Exp_2_maxlen_5.ipynb)

We changed training set dimension from 40 to 5 to train the model for short sentences and The accuracy was low. It was expected since more information about previous states in LSTM helps in improving accuracy. We obtained ROC-Auc value of 97% and accuracy value of 89.5%

Experiment 3: Change train step size (Exp_3_tr_stepsize.ipynb) We changed train step size from 1 to 5 train the model for short sentences and The accuracy was low. It was expected since it reduced the number of training examples. We obtained ROC-Auc value of 86% and accuracy value of 82%

Experiment 4: Change test step size (Exp_4_test_stepsize.ipynb) We changed test step size from 20 to 5 train the model for short sentences and The accuracy was low. We obtained ROC-Auc value of 86% and accuracy value of 79.5%

Experiment 5: Taking seed from following characters (Exp_5_seed_back.ipynb) We changed seed from 40 characters preceding to string to 20 characters preceding and 20 characters following the test string . The accuracy was low. It was expected since LSTM also learn sequences and give a better prediction if it finds a similar sequence in test which was the case initially. We obtained ROC-Auc value of 79% and accuracy value of 69%

Experiment 6: Changing seed to blank spaces(Exp_6_spaces.ipynb) We changed seed from 40 characters preceding to 40 spaces. The accuracy was low. It was expected since LSTM also learn sequences and give a better prediction if it finds a similar sequence in test which was the case initially.We obtained ROC-Auc value of 80% and accuracy value of 74.5%

Experiment 7: Changing seed to blank spaces followed by a character $ (non existent in the corpus) (Exp_7_non_existent.ipynb) We changed seed from 40 characters preceding to 39 spaces followed by a non existent character from the corpus- Dollar sign.

It was an attempt to simulate START using $. The accuracy was low. It was expected since LSTM also learn sequences and give a better prediction if it finds a similar sequence in test which was the case initially.We obtained ROC-Auc value of 67% and accuracy value of 61%

Experiments involving changing of hyperparameters

Experiment 8: Increasing LSTM layers (Exp_8_IncLayers.ipynb) We added one extra dense layer to existing architecture. We obtained ROC-Auc value of 93% and accuracy value of 84.5%

Experiment 9: Decreasing node size (Exp_9_DecNodeSize.ipynb) We decreased number of nodes from 128 to 64. We obtained ROC-Auc value of 96% and accuracy value of 88.5%

Experiment 10: Changing optimizer (Exp_10_opt.ipynb) We changed optimiser from to Adagrad. We obtained ROC-Auc value of 95% and accuracy value of 86.5%

Experiment 11: Changing activation function (Exp_11_act.ipynb) We changed activation function from to Sigmoid . We obtained ROC-Auc value of 95% and accuracy value of 86.5%

Experiment 12: Changing Dropout (Exp_12_dropout.ipynb) We changed dropout from to binary cross entropy. We obtained ROC-Auc value of 96% and accuracy value of 88.5%

Early Stopping

In machine learning, early stopping is a form of regularization used to avoid overfitting. Such methods update the learner so as to make it better fit the training data with each iteration. Up to a point, this improves the learner's performance on data outside of the training set. Past that point, however, improving the learner's fit to the training data comes at the expense of increased generalization error. Early stopping rules provide guidance as to how many iterations can be run before the learner begins to over-fit. Early stopping rules have been employed in many different machine learning methods, with varying amounts of theoretical foundation. We implement two variations of early stopping

Experiment 13. Early Stopping if loss falls below a threshold value

(Exp_13_14_EarlyStopping.ipynb)

The LSTM model stops training when there is no change in loss between subsequent epochs. This helps in stopping training early when no improvement in accuracy of the model is taking place.

Example: EarlyStopping(monitor='loss', patience=2, verbose=1),

Experiment 14. Early Stopping if no change in loss between certain # of epochs

(Exp_13_14_EarlyStopping.ipynb)

The LSTM model stops training when there is no change in loss between subsequent epochs. This helps in stopping training early when no improvement in accuracy of the model is taking place.

Example: EarlyStopping(monitor='loss', patience=2, verbose=1),

Implement All Language Detector Model

We peform language detection over the following 23 languages: czc.txt - Czech (Cesky) dns.txt - Danish (Dansk) dut.txt - Dutch (Nederlands) eng.txt - English frn.txt - French (Français) ger.txt - German (Deutsch) grk.txt - Ellinika' (Greek) hng.txt - Hungarian itn.txt - Italian jpn.txt - Japanese (Nihongo) lat.txt - Latvian lit.txt - Lithuanian (Lietuviskai) ltn.txt - Latin (Latina) ltn1.txt - Latin (Latina) lux.txt - Luxembourgish (Lëtzebuergeusch) mls.txt - Maltese por.txt - Portuguese rmn1.txt - Romani rum.txt - Romanian (Româna) rus.txt - Russian (Russky) spn.txt - Español (Spanish) ukr.txt - Ukrainian (Ukrayins'ka) yps.txt - YapeseResults: [[0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0] [3 0 0 0 2 0 0 0 0 0 0 0 0 2 0 0 0 0 1 0 0 2 0] [0 0 5 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 2 0] [0 0 0 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] [0 1 0 2 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] [0 0 0 0 1 5 0 1 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0] [0 0 0 0 0 0 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 9 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 1 0 1 0 0 0 0] [0 0 0 0 0 0 0 0 0 10 0 0 0 0 0 0 0 0 0 0 0 0 0] [0 0 0 0 0 1 0 0 0 0 9 0 0 0 0 0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0 0 0 0 9 0 0 1 0 0 0 0 0 0 0 0] [1 1 0 0 0 0 0 0 0 0 0 0 4 1 0 0 2 0 1 0 0 0 0] [0 0 1 0 1 0 0 0 0 0 0 0 1 4 0 0 0 0 1 0 2 0 0] [0 0 1 0 0 1 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 1 1 0 1 3 1 0 0 0 0 2 0 0 0 0 1] [0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 6 0 0 0 1 1 0] [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 9 0 0 0 0 0] [0 0 0 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 5 0 0 1 0] [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 7 0 3 0] [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 8 0 0] [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 8 0] [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 8]] Accuracy: 64.7826086957 We find that Romanic languages are not classified very effectively. English, French, German, Dutch, Latin, Latin1, Maltese, Romanian

Experiment15: Training for 20 epochs to compare against our baseline methods

(Exp_15_23Languages.ipynb)

```
We get an accuracy of 64.78 with Non Roman script languages being classified
 almost perfectly
```

Experiment16: Training for 50 epochs to check for improvement

(Exp_16_23Languages.ipynb)

```
We get an accuracy of 62.6 with Non Roman script languages still being class
ified almost perfectly. However roman script languages are classified worse
 in this case.
```

Experiment17: Training for 20 epochs with original french.txt (without the modified literals)

(Exp_17_23Languages_originalfrn.ipynb)

```
We get an accuracy of 64.78 with the confusion matrix exactly similar to Exp
eriment 1. This shows that changing the literals had no/little effect on the
 final model.
```

Experiment 18. Different LSTMs (changing the time series parameter)

(Exp_18_LSTMtime.ipynb)

We train an LSTM with varying time series parameter over different Train Shape for each sentence length varying from Maxlen=10...0 The idea behind this approach is to not fix the size of the input in the LSTM and make it robust to different train string lengths.

We find that the accuracy of the model drops to 77.5 % with an ROC AUC of 0.88. The model trains pretty quickly as the size of the train data decreases.

Experiment 19. Models: Ngram

(Exp_19_ngram.ipynb)

We run four experiments:

Experiment 19.1: one with original French text(We find 79.5% accuracy.) Experiment 19.2: one with modified French text (We find 80% accuracy) Intuitively, this makes sense as the accuracy should albeit slightly improve if the language literals are modified to be similar

Experiment 19.3: We tried to run the ngram model on our test set for all languages and the accuracy was around 35.62% when the test string size was 5. Experiment 19.4: But when test string size is increased to 50, the accuracy increases to 79.56%. This is probably because it required a longer test string size to discriminate effectively between so many languages.

Experiment 20. Models: Trigram (Franc)

(Exp_20_franc.ipynb)

Experiment 20.1: We train our test set over the franc-min package which supports 81 languages. We whitelist our command to check amongst only English and French languages. As franc supports many languages, it needs larger test string size passed to enable a valid classification. We used our test set size of 5 characters and our model output UNDEFINED every time.

In order to overcome this problem, we change our test string size to 50. We notice brilliant results with an accuracy of 99.5 % for our test set. This is probably because our training corpus is very large(~ 1 Million speakers) and that leads to almost perfect classification.

Experiment 20.2: When training over all languages, we obtain accuracy of 63. But we think this is because the languages which aren't classified have not been listed under the franc package. For the other languages we obtain almost perfect classification.

Experiment 21. Test Size 50

(Exp_21_test50.ipynb)

To ascertain that test size string is not the only criterion for the increase in Franc model's accuracy of (99.5%) we perform experiment over our original approach with test size now 50 instead of 5. We find the accuracy decreases from 88 to 85 and roc decreases from 95 to 94 [[87 13] [17 83]] 0.85 As opposed to almost perfect classification by Franc method. Thus the almost perfect classification is largely due to the large training corpus in franc.

Experiment 22. GRU

(Exp_22_GRU.ipynb)

We train a Gated Recurrent Unit instead of the baseline LSTM with the same parameters. We find an accuracy of 87 % with an roc of 95. [[86 14] [12 88]] 0.87 We notice every epoch is computed much faster than before. GRU is related to LSTM as both are utilizing different way if gating information to prevent vanishing gradient problem. Here are some pin-points about GRU vs LSTM- The GRU unit controls the flow of information like the LSTM unit, but without having to use a memory unit. It just exposes the full hidden content without any control. GRU is relatively new, and from our perspective, the performance is on par with LSTM, but computationally more efficient (less complex structure as pointed out). We are also seeing it being used more and more.

References:

1. http://cloudmark.github.io/Language-Detection-Implemenation/ (http://cloudmark.github.io/Language-Detection-Implemenation/)
2. https://github.com/wooorm/franc (https://github.com/wooorm/franc)
3. http://www1.cs.columbia.edu/~fadi/candidacy/LID/zissman93.pdf (http://www1.cs.columbia.edu/~fadi/candidacy/LID/zissman93.pdf)
4. William B. Cavnar and John M. Trenkle. N-gram-based text categorization. InProceedings of SDAIR, pages 161–175, 1994.
5. Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. In Proceedings of ACL, pages 310–318, 1996.
6. Philip Clarkson and Ronald Rosenfeld. Statistical language modeling using the cmu-cambridge toolkit. In Proceedings of ESCA Eurospeech, pages 2707–2710,1997.
7. Ted Dunning. Statistical identification of language. Technical report, New Mexico State University, 1994.
8. Gregory Grefenstette. Comparing two langage identification schemes. In JADT,1995.
9. Slava M. Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. IEEE Transactions on Acoustics,Speech, and Signal Processing, 35(3):400–401, 1987.
10. Philipp Koehn. Europarl: A parallel corpus for statistical machine translation.1995
11. http://www.lsi.upc.edu/~nlp/papers/padro04a.pdf (http://www.lsi.upc.edu/~nlp/papers/padro04a.pdf)
12. http://www.inf.ufrgs.br/~ceramisch/download_files/courses/Master_FRANCE/ENSIMAG_2008_2/Ingenierie_d (http://www.inf.ufrgs.br/~ceramisch/download_files/courses/Master_FRANCE/ENSIMAG_2008_2/Ingenierie_d
13. http://colah.github.io/posts/2015-08-Understanding-LSTMs/ (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)
14. http://mourafiq.com/2016/05/15/predicting-sequences-using-rnn-in-tensorflow.html (http://mourafiq.com/2016/05/15/predicting-sequences-using-rnn-in-tensorflow.html)