ReadVideo

Anindita Nath
Gerardo Cervantes

Department of Computer Science
University of Texas at El Paso

November 29, 2018

This is a report on the final project of the course *CS-5319, Natural Language Processing*, at the *University of Texas at El Paso*

Table of Contents

1 Objective

The primary objective of this project is to develop an improved domain-specific transcription system for Computer Science audios/videos in English.

2 Modular Diagram

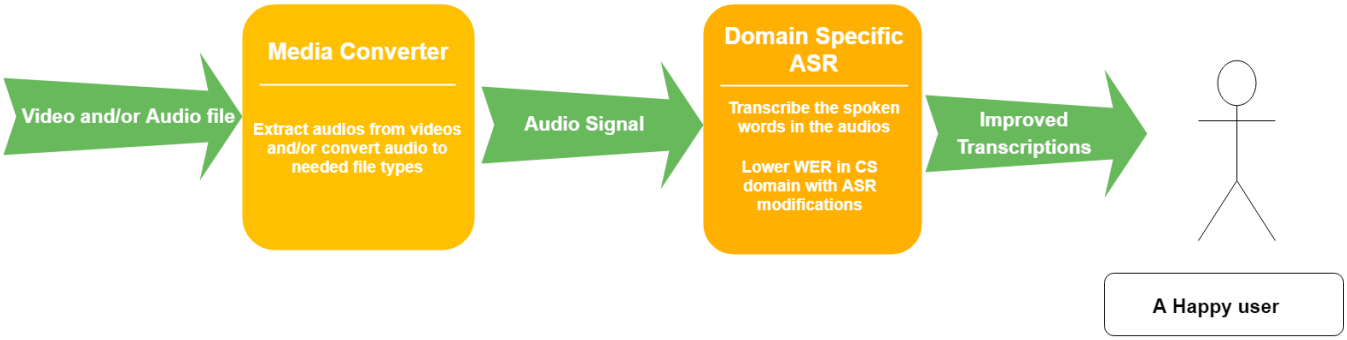

Figure 1: Modular Diagram

3 Overall Functionality

The Automatic Speech Recognition(ASR) system will transcribe the words spoken in the input audios/extracted audios. The output of the ASR system is the text corresponding to the spoken word. Language of the audios is, for now, limited only to English.

The system will be able to convert words spoken in English in an audio file to its corresponding text format. It is intended to generate better transcriptions for Computer Science related videos than the existing general transcriber. The system will also be able to capture Computer Science terminology in transcriptions. It will be able to accept both audio and video files as input since it will auto-extract audios from the video files.

4 Features

The following are the chief features of our ASR:

- *ReadVideo* will be very easy to use because it will work by just providing it the path to the video/audio file.
- It will be a light software, easy to download and needs no internet connection for its functioning.
- It will accept both video and audio files.
- Offline videos/audios will be accepted as well.
- The system will also be usable on most audio/video files. Most common formats of videos, .mp4, .avi, .mov, .wmv and .flv, will be accepted as input.

- The conversion of a general ASR to a domain-specific ASR is an interesting task and if implemented successfully, will benefit the Computer Science students and/or researchers immensely. This is the novel and also the most challenging part of the project.

5 Implementation Details- Current State and Issues Faced

5.1 The Basics

5.1.1 Download and Install *KALDI*

After installing *git* in our system, we cloned the *KALDI* project from its official GitHub repository at `https://github.com/kaldi-asr/kaldi.git`. The installation instructions are provided in *KALDI/INSTALL* file. This also necessitated installing loads of pre-requisites. However, the process was not quite smooth for various reasons as enumerated in *Section 5.1.3*.

5.1.2 How *KALDI* works

Following is a schematic diagram of different components of *KALDI*:

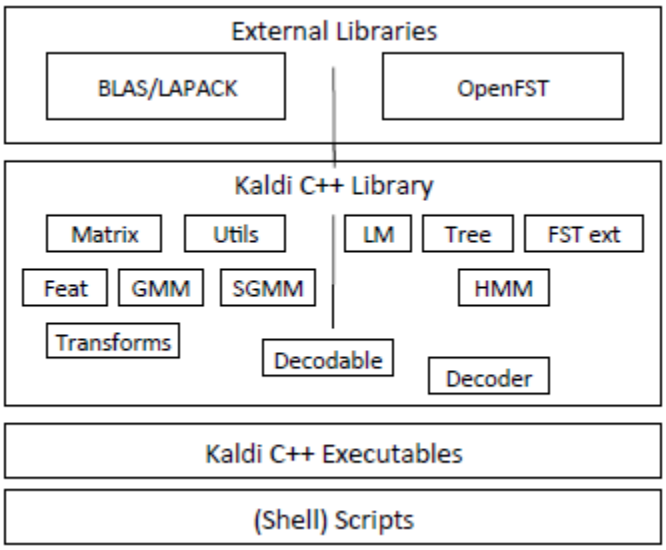

Figure 2: KALDI Schematic Diagram

Following is a brief walk-through [2]:

- Feature Extraction: provides support to create standard MFCC and PLP features, setting reasonable defaults with options to tweak for customized needs (e.g., the number of mel bins, minimum and maximum frequency cutoffs, etc.). Most commonly used feature extraction approaches include VTLN, cepstral mean and variance normalization, LDA, TC/MLLT, HLDA, and so on.
- Acoustic Modeling: supports conventional models (i.e. diagonal GMMs) and Subspace Gaussian Mixture Models (SGMMs) and also supports extensions to new kinds of models.
- Phonetic Decision Trees: provides general support for a wide range of approaches. The conventional approach is, in each HMM-state of each monophone, to have a decision tree that asks questions about, say, the left and right phones.

- Language Modeling: uses an FST-based framework so that it is, at least theoritically easy to use any language model that can be represented as an FST. For building LMs from raw text, users may use the IRSTLM toolkit.
- Creating Decoding Graphs: uses Weighted Finite State Transducers (WFSTs) for all training and decoding algorithms.
- Decoders: includes simple to highly optimized, on-the-fly language model rescoring and lattice generation.

5.1.3 Issues

We tried to get *KALDI* set up on our *Windows OS* since that is the readily available architecture that we use for all development purposes in our lab as well as our personal PCs. *KALDI*, however, failed to run on *Windows OS* despite successful installation. Problems started when we tried to run or create models. The problems faced were open problems, yet unresolved, which is understandable since this inclusion is very recent. However, this delayed the start of the project, and also required for us to install *LINUX OS* in our systems. We installed *Ubuntu* through virtual box, *Oracle VMWare*. But the hard-disk limits and especially, low RAM and non-availability of multiple parallel processors in a virtual environment made the installation process quite cumbersome and prolonged, more than anticipated.

5.2 Media Converter

We created a media converter that can run from the command-line for converting most video/audio files to the corresponding audio files required by respective *KALDI* model. The bash script found in Appendix C, takes in the path to the video file. The media converter uses *FFmpeg* a free command-line software for handling video and audio files. *FFmpeg* converts most video files or audio files to the *.flac* format required. *Sox* then runs on the audio output of *FFmpeg* and changes the sample rate of the audio to be 16k and the bits per sample to be 16. *KALDI* supports bits per sample of either 8, 16, or 32.

5.3 Librispeech Corpus

One of the data-sets used for training was LibriSpeech from OpenSLR. A smaller version of this dataset was used due to memory limits. After doing some research, we found that 17 gigabytes of memory is needed to create a ASR model with 100 hours of the LibriSpeech OpenSLR data. We did not have that amount of memory, so we opted in using less data to create the ASR models.

There was 1 thousand hours of data in the LibiriSpeech corpus from OpenSLR, only a portion of that was used for the model. Only 5 hours of the LibriSpeech data was used for training, approximately 5 hours of data was used for test, and 3 hours used for dev set.

Digging deeper, we found that this corpus contained audio-book data, more specifically recordings of humans reading books and their corresponding transcripts. Since, evaluation of the system would be done with YouTube Computerphile videos/audios, which included monologues of humans lecturing on or discussing about Computer Science topics, we figured that TEDLIUM corpus could be a better suited option for training data.

5.4 TEDLIUM Corpus

Included in the *KALDI/egs* directory, is the *tedlium* folder. The script *run.sh* first downloads the entire TED talk corpus, 32 GB, and creates various *KALDI* models as per requirements.

The script *ted_download_lm.sh* downloads pre-built language models trained on the Cantab-TEDLIUM text data and TEDLIUM acoustic training data. To build own language models, the script *tedlium/local/ted_train_lm.sh* is run. We built a 4 gram language model. Acoustic model was based on chain TDNN model. LDA was used instead of PCA for i-vector features. The audios were segmented into 3 seconds utterances. Training data included 212 hours of audios. Evaluating on the test set yielded a word-error rate of 15%, while training for 4 epochs only. However, this took a long time and hence, evaluation with Computerphile audios could not be done in time. Meanwhile, we played with a pre-trained *TEDLIUM* model as described below.

Kaldi GStreamer server[1]. This is a real-time full-duplex speech recognition server, based on the *KALDI* toolkit and the GStreamer framework and implemented in Python. This provides support for speech segmentation, i.e., a long speech signal is broken into shorter segments based on silences, supports arbitrarily long speech input and can be used with *KALDI*'s GMM as well as online DNN models.

The DNN-based online decoder requires a newer GStreamer plugin that is not in the *KALDI* code base and has to be compiled separately. It's available at https://github.com/alumae/gst-kaldi-nnet2-online. The DNN-based models for English, pre-trained on the TEDLIUM speech corpus and combined with a generic English language model provided by *Cantab Research* were downloaded. The *download-tedlium-nnet2.sh* script downloads the models (approximately, 1.5 GB).

We evaluated this model with Computerphile audio whose results are detailed in *Section 6.4.1*.

5.5 Gathering Computer-Science Domain Data

To know how well our ASR models perform when transcribing an audio from the Computer Science domain, we used videos on YouTube uploaded by Computerphile. Computerphile videos are videos where a speaker tries to explain Computer Science topics to a general audience. We downloaded 5 Computerphile videos and their respective transcriptions to test how well the predicted transcriptions were to the original.

5.6 Gathering Transcriptions from YouTube

To be able to use transcriptions from YouTube, we needed to access the transcripts available with the videos. To get transcripts, we made a Python script that generates transcriptions from YouTube, refer Appendix B. YouTube transcriptions can be accessed from the corresponding URL, providing the video-id and the language via an *http request*. The transcriptions given are in XML, which were parsed. We cleaned the transcriptions to fit a format closer to what the transcriptions of the training data look like. Pre-processing steps included removing punctuation and new-line characters. Apostrophes in the transcripts came out as either " or &#39, so they were replaced by actual quote characters. Regular expressions were used for these pre-processing.

5.7 Generating Computer Science Words

One technique we wanted to attempt was to increase the probability of a domain-specific word appearing in the language model. For this, we needed a list of Computer Science words.

To find Computer Science domain-specific words, a few techniques were applied. We first generated a small list of Computer Science words hand-picked from memory, as well as looking at a few Computer Science documents to get more Computer Science words. From this small list of Computer Science words, more words were generated by using WordNet. The list of hand-picked Computer Science words and a bigger list of generated Computer Science words can be found in Appendix D. At the very end, duplicates were removed so that the list of Computer Science words were unique. The code can be found in Appendix A.

5.7.1 WordNet

WordNet was used to get the synsets of every word in the small list of hand-picked Computer Science words. This list of generated synsets were added to the list of Computer Science words.

5.7.2 Word Embeddings

The other techniques used to generate similar Computer Science words was to use word embeddings to get the top 5 closest words in the vector space. Pre-trained word embbedings from *GloVe* were used. The word embeddings from *GloVe* had 50 dimensions and was generated from Wikipedia 2014 + Gigaword 5.

6 Performance Analysis

6.1 Performance Metric

Different pre-trained/freely available acoustic and language models available with the corpora, included in the official *KALDI* Github site, used different metrics for comparing the performance of the models. While some used word-error rate, others used confidence scores, perplexity and/or likelihood estimates. To have an uniform base for comparison, we used a customized Python script, refer Appendix F, to compute the word-error rate between the transcribed text and the reference text when the models were evaluated with Computerphile audios. It can be easily installed through *pip* and run in command-line as *wer ref hyp* where *ref* is the reference text file name and *hyp* is the transcribed text file name.

6.2 Results

The following enumerates some of our evaluation results:

Corpus	Algorithm	Audio	Word-Error Rate
Yes/No	Monophone	Yes/No audios	0.00%
Librispeech	Monophone	Librispeech	57.64%
Librispeech	Tri1	Librispeech(test)	30.82%
Librispeech	Tri2b	Librispeech(test)	24.44%
Librispeech	Tri1	CPAudio1	143.85%
TED-Lium	chain TDNN	TED-Lium(test)	15.0%
TED-Lium	DNN-pretrained	CPAudio2	22.96%

Performance Analysis

6.3 Preliminary KALDI experiment - Yes and No in Hebrew

A preliminary experiment on KALDI was done to ensure that the model was installed and running successfully. The model detects the word 'yes' and 'no' in Hebrew. The dataset involves an individual recording the word multiple times. The model used for that was monophone. The data is available for download at http://openslr.org/resources/1/waves_yesno.tar.gz, containing 60 audios. 31 of the audio files are for training while the other 29 are used as test data. The model is able to achieve a Word-Error Rate (WER) of 0.00% mainly because it is an easy problem to solve with a vocabulary size of only 2 words.

6.4 Model analyses

6.4.1 Librispeech Model

As mentioned in the table, Performance Analysis, CPAudio1 is the Computerphile video: Why Do We Need IP Addresses? - Computerphile. Monophone feature type is Delta. Tri1 features type is Delta. Tri2b uses (delta+delta-deltas). The Tri2b model uses LDA and MLLT.

Transcription Prediction

WELL AS I ¡UNK¿ ROUN AN ROUN ¡UNK¿ ¡UNK¿ ¡UNK¿ OR MY ADDRESS ¡UNK¿ TO ¡UNK¿ THE ¡UNK¿ AND SON UNLESS IT IS ONE WHO KNEW NEITHER HIS IS THE ¡UNK¿ OF ITS OWN THAT ITS IMAGES IN QUESTION

Original Transcription

we had this idea of a machine having or a network card or a wifi card having a 'mac' address which I understand to be a unique address to that dev not necessarily that device but certainly to that network interface (that's probably the best word for it is it?) So the question is why do we need IP addresses if we've got mac addresses? It's an interesting question

The Tri1 model was tested on a Computerphile video, a sample of the transcriptions and predictions can be seen in Appendix E. In the predictions you can see some words line up like the *word address* and *question*. The model also did not recognize some words and put them as unknown words. The results might have not been as good as we hoped because the acoustic model in the ASR was trained with very little amount of data rather than what was expected. You can see that there are similarities in the transcript predictions and the actual transcriptions for the file.

6.4.2 TEDLIUM Model

As mentioned in the table, Performance Analysis, CPAudio2 is the Computerphile audio: *bill-gates*.

Transcription Prediction

when i was a kid the disaster we worry about most was a nuclear war. that's why we had a bear like this down our basement filled with cans of food and water. nuclear attack came we were supposed to go downstairs hunker down and eat out of that barrel. today the greatest risk of global catastrophe. don't look like this instead it looks like this. if anything kills over ten million people in

the next few decades it's most likely to be a highly infectious virus rather than a war. not missiles that microbes now part of the reason for this is that we have invested a huge amount in nuclear deterrence we've actually invested very little in a system to stop an epidemic. we're not ready for the next epidemic.

Original Transcription

When I was a kid, the disaster we worried about most was a nuclear war. That's why we had a barrel like this down in our basement, filled with cans of food and water. When the nuclear attack came, we were supposed to go downstairs, hunker down, and eat out of that barrel. Today the greatest risk of global catastrophe doesn't look like this. Instead, it looks like this. If anything kills over 10 million people in the next few decades, it's most likely to be a highly infectious virus rather than a war. Not missiles, but microbes. Now, part of the reason for this is that we've invested a huge amount in nuclear deterrents. But we've actually invested very little in a system to stop an epidemic. We're not ready for the next epidemic.

The pre-trained DNN based model built with TEDLIUM corpus data was tested on a Computerphile audio. This yielded an word-error rate of 22.96%. Performance surely improved than the previous model. The reasons for such high performance were:

- running a more sophisticated acoustic model based on deep neural networks rather than Gaussian models.
- structure of TED-talks. They were mostly lecture or presentation talks or monologues, well-structured and designed to engage the audience. This essentially matched the structure of our audios/videos in the evaluation set, downloaded from YouTube's Computerphile site.

7 Conclusion

7.1 Pitfalls

- Different metrics were implemented in the freely available scripts for building various KALDI models. Hence, performance analysis/comparison could not be based on a single metric. Though, we used a customized script to compute Word-Error Rate, it could not be integrated as an output of the ASR models.
- Most of the data used for speech-recognition and that are freely available in OpenSLR are either small-sized or not of the desired format, e.g. audio-book data, digit recognition data, etc. which did not prove to be good training corpora for our evaluation audios, that included lecture audios or presentation talks.
- Most useful form of corpora were only available at a price from LDC.
- Non-availability of suitable architectural/hardware support (GPUs or AWS), required to run large-sized corpora or to build DNN based models, was a major handicap. Even a simple networks model took days to train. Since *KALDI* failed to run on *Windows OS*, we could only run it through a virtual *Ubuntu OS* that had minimum RAM(1024MB) and an insufficient memory of only 32 GB.

7.2 Skills Acquired

Though we were not able to achieve our main objectives, we were able to apply many useful skills. While implementing the project, we learned:

- how to make *KALDI* work with different types of English corpora.
- what is required to make an ASR model and how to build its different parts.
- how to evaluate the pre-trained models with custom audios.
- how to directly download audios form YouTube videos, extract audios from videos, convert to various audio formats and write various batch scripts.
- how to modify a lexicon, making it more domain-specific using word-embeddings and integrate it with existing language models.
- how to run and edit various shell scripts.
- to work with Linux Virtual box environment

8 Future Work

- Integrate our domain specific words-list into the trained models, by converting them into required formats.
- Experiment with more domain-specific lexicons.
- Experiment with advanced deep learning models like *theano-based RNN-LSTM models*.
- Run models with large-sized corpora on an more efficient/faster architecture with higher computation abilities (GPUs or AWS platform).
- Extend this functionality to other domains and/or other languages.

9 Roles & Responsibilities

Sl.	Task	Person Hours	Task Completion Time	Person Responsible
1	Download and Install KALDI	2	48	Anindita Nath & Gerardo Cervantes
2	Scripts for direct audio download from YouTube, auto-extraction of audios from videos, conversion of audios in desired format (using ffmpeg and sox)	0.5	0.5	Anindita Nath
3	Trained and Tested with small OpenSLR Hebrew data set of 'yes' and'no'	3	3	Gerardo Cervantes
4	Script to generate list of Computer Science terms from word-embeddings	0.5	1.5	Gerardo Cervantes
5	Download Computerphile videos with transcriptions in required format	1	2	Anindita Nath & Gerardo Cervantes
6	Download librispeech/mini-librispeech corpus	0.5	5	Gerardo Cervantes
7	Train and test on split dev and test data set of same corpus	3	5	Gerardo Cervantes
8	Test Librispeech model on Computerphile audios	3	5	Gerardo Cervantes
8	Download TED corpus and create DNN based KALDI model	3	24	Anindita Nath
9	Test above model on test data from same corpus	3	10	Anindita Nath
10	Test pre-trained DNN based KALDI model on Computerphile audios	3	5	Anindita Nath
11	Use modified lexicon to improve domain-specific performance	In progress		Anindita Nath
12	Final project report	3	5	Anindita Nath & Gerardo Cervantes
13	Final project presentation	In progress		Anindita Nath & Gerardo Cervantes

Roles and Responsibilities

10 References

[1] Tanel Alumäe. *Full-duplex Speech-to-text System for Estonian*. Kaunas, Lithuania, 2014.

[2] Daniel Povey et al. "The kaldi speech recognition toolkit". In: *In IEEE 2011 workshop*. 2011.

Appendices

A Generating domain specific words code

```
Appendix A
# -*- coding: utf-8 -*-
"""

@author: Gerardo Cervantes
"""

from nltk.corpus import wordnet
from gensim.models import KeyedVectors

def get_related_word(starting_words):
    related_word_list = []
    #Word embeddings in gensim format
    gloveModel = KeyedVectors.load_word2vec_format('../gensim_glove.6b.50d.txt')

    for starting_word in starting_words:
        related_words = get_wordnet_synsets(starting_word)
        similar_word_embeddings = gloveModel.similar_by_word(starting_word, 5)
        similar_words = [w_embedding for w_embedding,similarity in similar_word_embeddi
ngs]
        print('similar words: ', similar_words)
        related_word_list = related_word_list + related_words + similar_words

    return set(related_word_list)


def get_wordnet_synsets(word):
    syns = wordnet.synsets(word)
    synsets = [synword.lemmas()[0].name() for synword in syns]
    return synsets

#Used for converting from glove format to gensim which is used in this case
def convert_glove_to_gensim(src_glove, dst_gensim):
    from gensim.scripts.glove2word2vec import glove2word2vec
    glove2word2vec(src_glove, dst_gensim)

if __name__ == '__main__':
    cs_words = ['computer', 'hash', 'array', 'programming', 'number', 'float', 'double'
, 'integer', 'encrypt',
                'password', 'user', 'machine', 'learning', 'network']

    related_words = get_related_word(cs_words)
    print(related_words)
```

B Gathering transcripts from YouTube code

```python
# -*- coding: utf-8 -*-
"""
Created on ~

@author: Gerardo Cervantes
"""

import requests
import re

import xml.etree.cElementTree as ET

def send_http_request(video_id, lang):
    request = requests.get('http://video.google.com/timedtext?lang=' + lang + '&v=' + video_id)

    return request.text

def parse_transcriptions_xml(xml):
    root = ET.fromstring(xml)
    transcript_text = ''
    for child in root:
        if child.tag == 'text':
            transcript_text = transcript_text + ' ' + child.text
            print(child.text)

    return transcript_text

def remove_transcription_punctuation(text):
    text = re.sub(r'[.,]','', text)
    text = re.sub(r'[\n]',' ', text)
    text = re.sub(r'"','\'', text)
    text = re.sub(r''','\'', text)
    text = re.sub(r'  ',' ', text)

    return text

if __name__ == '__main__':
    video_id = 'iGPXkxeOfdk'
    lang = 'en'

    xml_content = send_http_request(video_id, lang)

    transcript = parse_transcriptions_xml(xml_content)

    #Remove punctuation
    transcript = remove_transcription_punctuation(transcript)
    print(transcript)
    with open('transcript', 'w') as text_file:
        text_file.write(transcript)
```

```
#Media_converter.sh


ffmpeg -i $1 output.flac
sox output.flac -b 16 output2.flac rate 16k
rm output.flac
```

Hand-picked Computer Science words

['computer', 'hash', 'array', 'programming', 'number', 'float', 'double', 'integer', 'encrypt',

'password', 'user', 'machine', 'learning', 'network']

Generated Computer Science words

{'nonzero', 'ice-cream_soda', 'non-negative', 'passwords', 'software', 'authentication', 'cryptographic', 'internet', 'skills', 'application', 'password', 'learning', 'car', 'technology', 'numeral', 'login', 'channel', 'straight', 'bivalent', 'integers', 'decrypt', 'least', 'user', 'determine', 'single', 'phone_number', 'upside', 'electronic', 'device', 'memorize', 'scheduling', 'encrypts', 'combining', 'act', 'count', 'floated', 'gun', 'sail', 'encrypted', 'drug_user', 'using', 'username', 'users', 'double', 'practical', 'server', 'formula_1', 'floating', 'sha-1', 'tying', 'integer', 'eruditeness', 'sophisticated', 'non-zero', 'ranging', 'types', 'net', 'creating', 'align', 'hash', 'array', 'duplicate', 'calculator', 'enables', 'interactive', 'annotate', 'code', 'channels', 'teach', 'experience', 'learn', 'third', 'float', 'air_bladder', 'numbers', 'only', 'encrypting', 'ten', 'network', 'networks', 'double_over', 'multiplication', 'computers', 'floats', 'format', 'issue', 'range', 'number', 'interface', 'teaching', 'doubling', 'hashish', 'machine', 'cable', 'triple', 'other', 'total', 'doubly', 'programming', 'program', 'knowledge', 'exploiter', 'used', 'computer', 'machines', 'hashes', 'md5'}

E Transcription prediction comparison

Model Tri1 prediction:

WELL AS I <UNK> ROUN AN ROUN <UNK> <UNK> <UNK> OR MY ADDRESS <UNK> TO <UNK> THE <UNK> AND SON UNLESS IT IS ONE WHO KNEW NEITHER HIS IS THE <UNK> OF ITS OWN THAT ITS IMAGES IN QUESTION IS DIVIDED INTO THINGS OUTSIDE WITH THE KING OF <UNK> AMONG THE DRESS OF SOMETHING LIGHT SAID FIGHTS OF BENEATH THAT MY ADDRESS THERE SO THEY TELLS ABOUT A FOOT FEED IT IS A MOST OFFICE WHICH CAN BE USED TO INDICATE THINGS LIKE THIS DISMAL TO CAST UPON THE <UNK> OF WENT ON ANY VISITED ALL THE INDICATING WHAT YOU THINK OF WHAT IS THE MONEY THATCHER SEALED SLICES ON HIS PULPIT TO RESORT TO POINT OUT TO DEBTS OF ALL NEW KINDS MONEY NEEDED OF ALL MEN EVEN INTO <UNK> AMENITIES WOODRUFF IT ONLY TO ASTONISH AND <UNK> CITED IN THE SIDE OF IT ADDRESSED BY FITTED THE SQUABBLE IN SIX THOUGHTS SO NICE NEEDED HEIGHTS THAT THESE WERE THE HANDS OF A HOT WHEN A THIEF HAD CAUGHT OCCUPIED WHEN IT WAS ALL THE OF PHYSICAL THINGS BECAUSE IT WAS OF THE SULTAN COLLEGE HOPPET HALL POT COVENANTED THAT INDABA ADOWN LIAISON THEM OUT EYES WITH A SUDDEN IS SITTING WITH WHAT FINE EXAMPLES SAID MISTER SOFA IN MANY CASES SIT ON FAST DOWN STEVENS AWOKE ONE NIGHT IN BETWEEN THE MEDLICOTT <UNK> WHAT OF THAT WHITE GOWNS UNITED X FOURTEEN TERESA <UNK> BECAUSE SIXTY ONE FIVE SWALLOW RACES WAS FATAL WUTHLESS ON WITH HIS FACE AT THE SIGHT OF THE AMA

Youtube Transcription

we had this idea of a machine having or a network card or a wifi card having a 'mac' address which I understand to be a unique address to that dev not necessarily that device but certainly to that network interface (that's probably the best word for it is it?) So the question is why do we need IP addresses if we've got mac addresses? It's an interesting question is because they do different things I'd say's probably the glib answer to it A MAC address looks something like; six bytes would be an ethernet mac address there are some details about the first few bits at the start of this which can be used to indicate things like this is multicast and broadcast couple of things and then you've essentially got: These indicate the; well you can think of it as the manufacturer so you apply to the standards body I think it's the IEEE to get a block You pay them some money and you get a block and then you can mint your own mac addresses from that and then these would refer normally to the station on the network on the ethernet so that gives you a fairly big address space you can address quite a lot in six bytes So it used to be the case that these were burnt into the hardware of the ethernet card back in the day when it was all physical things with physical wires there'd be something on the card which had this hard hard coded into in in the hardware nowadays and I mean for several years particularly with WiFi it's been possible to set this through software in many cases so you can't in fact guarantee one-to-one mapping between the network card which is what it used to be what it identified and the mac address for its you can change the mac address network card particularly in wifi - was one of the reasons why spoofing on wifi is so much is actually easy than people think On the other hand an IP address is a four byte number in Hex that might be something like that I'm not gonna try & translate that to decimal in my head an example of a decimal one would be 10 dot zero dot zero dot one So that's a dotted quad notation these are bytes each of these goes from 0 to 255 - When you have Hollywood doing films the number of times that erm CSI's a particular example of this as well the number of times that people will do trace routes and other such things to IP addresses that start you know 700 and something and it's like it's completely meaningless [It's a bit like the Hollywood phone numbers: 555] Yeah Possibly - maybe it's been done deliberately I would like it would be nice to think it was done deliberately so you've got these two

F ASR Evaluation Script

```
#
Copyright
2017-2018
Ben
Lambert
```

```
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at

#     http://www.apache.org/licenses/LICENSE-2.0

# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""
Primary code for computing word error rate and other metrics from ASR
output.
"""
from __future__ import division

from functools import reduce
from collections import defaultdict
from edit_distance import SequenceMatcher

from termcolor import colored

# Some defaults
print_instances_p = False
print_errors_p = False
files_head_ids = False
files_tail_ids = False
confusions = False
min_count = 0
wer_vs_length_p = True

# For keeping track of the total number of tokens, errors, and matches
ref_token_count = 0
error_count = 0
```

```
match_count = 0
counter = 0
sent_error_count = 0

# For keeping track of word error rates by sentence length
# this is so we can see if performance is better/worse for longer
# and/or shorter sentences
lengths = []
error_rates = []
wer_bins = defaultdict(list)
wer_vs_length = defaultdict(list)
# Tables for keeping track of which words get confused with one another
insertion_table = defaultdict(int)
deletion_table = defaultdict(int)
substitution_table = defaultdict(int)
# These are the editdistance opcodes that are condsidered 'errors'
error_codes = ['replace', 'delete', 'insert']


# TODO - rename this function.  Move some of it into evaluate.py?
def main(args):
    """Main method - this reads the hyp and ref files, and creates
    editdistance.SequenceMatcher objects to compute the edit distance.
    All the statistics necessary statistics are collected, and results are
    printed as specified by the command line options.
    This function doesn't not check to ensure that the reference and
    hypothesis file have the same number of lines.  It will stop after the
    shortest one runs out of lines.  This should be easy to fix...
    """
    global counter
    set_global_variables(args)

    counter = 0
    # Loop through each line of the reference and hyp file
    for ref_line, hyp_line in zip(args.ref, args.hyp):
        processed_p = process_line_pair(ref_line, hyp_line,
case_insensitive=args.case_insensitive,

remove_empty_refs=args.remove_empty_refs)
        if processed_p:
            counter += 1
    if confusions:
        print_confusions()
    if wer_vs_length_p:
        print_wer_vs_length()
    # Compute WER and WRR
```

```
    if ref_token_count > 0:
        wrr = match_count / ref_token_count
        wer = error_count / ref_token_count
    else:
        wrr = 0.0
        wer = 0.0
    # Compute SER
    if counter > 0:
        ser = sent_error_count / counter
    else:
        ser = 0.0
    print('Sentence count: {}'.format(counter))
    print('WER: {:10.3%} ({:10d} / {:10d})'.format(wer, error_count,
ref_token_count))
    print('WRR: {:10.3%} ({:10d} / {:10d})'.format(wrr, match_count,
ref_token_count))
    print('SER: {:10.3%} ({:10d} / {:10d})'.format(ser, sent_error_count,
counter))


def process_line_pair(ref_line, hyp_line, case_insensitive=False,
remove_empty_refs=False):
    """Given a pair of strings corresponding to a reference and hypothesis,
    compute the edit distance, print if desired, and keep track of results
    in global variables.
    Return true if the pair was counted, false if the pair was not counted
due
    to an empty reference string."""
    # I don't believe these all need to be global.  In any case, they
shouldn't be.
    global error_count
    global match_count
    global ref_token_count
    global sent_error_count

    # Split into tokens by whitespace
    ref = ref_line.split()
    hyp = hyp_line.split()
    id_ = None

    # If the files have IDs, then split the ID off from the text
    if files_head_ids:
        id_ = ref[0]
        ref, hyp = remove_head_id(ref, hyp)
    elif files_tail_ids:
        id_ = ref[-1]
```

```
            ref, hyp = remove_tail_id(ref, hyp)

        if case_insensitive:
            ref = list(map(str.lower, ref))
            hyp = list(map(str.lower, hyp))
        if remove_empty_refs and len(ref) == 0:
            return False

        # Create an object to get the edit distance, and then retrieve the
        # relevant counts that we need.
        sm = SequenceMatcher(a=ref, b=hyp)
        errors = get_error_count(sm)
        matches = get_match_count(sm)
        ref_length = len(ref)

        # Increment the total counts we're tracking
        error_count += errors
        match_count += matches
        ref_token_count += ref_length

        if errors != 0:
            sent_error_count += 1

        # If we're keeping track of which words get mixed up with which others,
    call track_confusions
        if confusions:
            track_confusions(sm, ref, hyp)

        # If we're printing instances, do it here (in roughly the align.c
    format)
        if print_instances_p or (print_errors_p and errors != 0):
            print_instances(ref, hyp, sm, id_=id_)

        # Keep track of the individual error rates, and reference lengths, so we
        # can compute average WERs by sentence length
        lengths.append(ref_length)
        if len(ref) > 0:
            error_rate = errors * 1.0 / len(ref)
        else:
            error_rate = float("inf")
        error_rates.append(error_rate)
        wer_bins[len(ref)].append(error_rate)
        return True

    def set_global_variables(args):
        """Copy argparse args into global variables."""
```

```
    global print_instances_p
    global print_errors_p
    global files_head_ids
    global files_tail_ids
    global confusions
    global min_count
    global wer_vs_length_p
    # Put the command line options into global variables.
    print_instances_p = args.print_instances
    print_errors_p = args.print_errors
    files_head_ids = args.head_ids
    files_tail_ids = args.tail_ids
    confusions = args.confusions
    min_count = args.min_word_count
    wer_vs_length_p = args.print_wer_vs_length

def remove_head_id(ref, hyp):
    """Assumes that the ID is the begin token of the string which is common
    in Kaldi but not in Sphinx."""
    ref_id = ref[0]
    hyp_id = hyp[0]
    if ref_id != hyp_id:
        print('Reference and hypothesis IDs do not match! '
              'ref="{}" hyp="{}"\n'
              'File lines in hyp file should match those in the ref
file.'.format(ref_id, hyp_id))
        exit(-1)
    ref = ref[1:]
    hyp = hyp[1:]
    return ref, hyp

def remove_tail_id(ref, hyp):
    """Assumes that the ID is the final token of the string which is common
    in Sphinx but not in Kaldi."""
    ref_id = ref[-1]
    hyp_id = hyp[-1]
    if ref_id != hyp_id:
        print('Reference and hypothesis IDs do not match! '
              'ref="{}" hyp="{}"\n'
              'File lines in hyp file should match those in the ref
file.'.format(ref_id, hyp_id))
        exit(-1)
    ref = ref[:-1]
    hyp = hyp[:-1]
    return ref, hyp
```

```
def print_instances(ref, hyp, sm, id_=None):
    """Print a single instance of a ref/hyp pair."""
    print_diff(sm, ref, hyp)
    if id_:
        print(('SENTENCE {0:d}  {1!s}'.format(counter + 1, id_)))
    else:
        print('SENTENCE {0:d}'.format(counter + 1))
    # Handle cases where the reference is empty without dying
    if len(ref) != 0:
        correct_rate = sm.matches() / len(ref)
        error_rate = sm.distance() / len(ref)
    elif sm.matches() == 0:
        correct_rate = 1.0
        error_rate = 0.0
    else:
        correct_rate = 0.0
        error_rate = sm.matches()
    print('Correct          = {0:6.1%}  {1:3d}
({2:6d})'.format(correct_rate, sm.matches(), len(ref)))
    print('Errors           = {0:6.1%}  {1:3d}
({2:6d})'.format(error_rate, sm.distance(), len(ref)))

def track_confusions(sm, seq1, seq2):
    """Keep track of the errors in a global variable, given a sequence
matcher."""
    opcodes = sm.get_opcodes()
    for tag, i1, i2, j1, j2 in opcodes:
        if tag == 'insert':
            for i in range(j1, j2):
                word = seq2[i]
                insertion_table[word] += 1
        elif tag == 'delete':
            for i in range(i1, i2):
                word = seq1[i]
                deletion_table[word] += 1
        elif tag == 'replace':
            for w1 in seq1[i1:i2]:
                for w2 in seq2[j1:j2]:
                    key = (w1, w2)
                    substitution_table[key] += 1

def print_confusions():
    """Print the confused words that we found... grouped by insertions,
deletions
    and substitutions."""
    if len(insertion_table) > 0:
```

```
        print('INSERTIONS:')
        for item in sorted(list(insertion_table.items()), key=lambda x:
x[1], reverse=True):
            if item[1] >= min_count:
                print('{0:20s} {1:10d}'.format(*item))
    if len(deletion_table) > 0:
        print('DELETIONS:')
        for item in sorted(list(deletion_table.items()), key=lambda x: x[1],
reverse=True):
            if item[1] >= min_count:
                print('{0:20s} {1:10d}'.format(*item))
    if len(substitution_table) > 0:
        print('SUBSTITUTIONS:')
        for [w1, w2], count in sorted(list(substitution_table.items()),
key=lambda x: x[1], reverse=True):
            if count >= min_count:
                print('{0:20s} -> {1:20s}   {2:10d}'.format(w1, w2, count))

# TODO - For some reason I was getting two different counts depending on how
I count the matches,
# so do an assertion in this code to make sure we're getting matching
counts.
# This might slow things down.
def get_match_count(sm):
    "Return the number of matches, given a sequence matcher object."
    matches = None
    matches1 = sm.matches()
    matching_blocks = sm.get_matching_blocks()
    matches2 = reduce(lambda x, y: x + y, [x[2] for x in matching_blocks],
0)
    assert matches1 == matches2
    matches = matches1
    return matches


def get_error_count(sm):
    """Return the number of errors (insertion, deletion, and substitutiions
    , given a sequence matcher object."""
    opcodes = sm.get_opcodes()
    errors = [x for x in opcodes if x[0] in error_codes]
    error_lengths = [max(x[2] - x[1], x[4] - x[3]) for x in errors]
    return reduce(lambda x, y: x + y, error_lengths, 0)

# TODO - This is long and ugly.  Perhaps we can break it up?
# It would make more sense for this to just return the two strings...
```

```
def print_diff(sm, seq1, seq2, prefix1='REF:', prefix2='HYP:', suffix1=None,
suffix2=None):
    """Given a sequence matcher and the two sequences, print a Sphinx-style
    'diff' off the two."""
    ref_tokens = []
    hyp_tokens = []
    opcodes = sm.get_opcodes()
    for tag, i1, i2, j1, j2 in opcodes:
        # If they are equal, do nothing except lowercase them
        if tag == 'equal':
            for i in range(i1, i2):
                ref_tokens.append(seq1[i].lower())
            for i in range(j1, j2):
                hyp_tokens.append(seq2[i].lower())
        # For insertions and deletions, put a filler of '***' on the other
one, and
        # make the other all caps
        elif tag == 'delete':
            for i in range(i1, i2):
                ref_token = colored(seq1[i].upper(), 'red')
                ref_tokens.append(ref_token)
            for i in range(i1, i2):
                hyp_token = colored('*' * len(seq1[i]), 'red')
                hyp_tokens.append(hyp_token)
        elif tag == 'insert':
            for i in range(j1, j2):
                ref_token = colored('*' * len(seq2[i]), 'red')
                ref_tokens.append(ref_token)
            for i in range(j1, j2):
                hyp_token = colored(seq2[i].upper(), 'red')
                hyp_tokens.append(hyp_token)
        # More complicated logic for a substitution
        elif tag == 'replace':
            seq1_len = i2 - i1
            seq2_len = j2 - j1
            # Get a list of tokens for each
            s1 = list(map(str.upper, seq1[i1:i2]))
            s2 = list(map(str.upper, seq2[j1:j2]))
            # Pad the two lists with False values to get them to the same
length
            if seq1_len > seq2_len:
                for i in range(0, seq1_len - seq2_len):
                    s2.append(False)
            if seq1_len < seq2_len:
                for i in range(0, seq2_len - seq1_len):
                    s1.append(False)
```

```
            assert len(s1) == len(s2)
            # Pair up words with their substitutions, or fillers
            for i in range(0, len(s1)):
                w1 = s1[i]
                w2 = s2[i]
                # If we have two words, make them the same length
                if w1 and w2:
                    if len(w1) > len(w2):
                        s2[i] = w2 + ' ' * (len(w1) - len(w2))
                    elif len(w1) < len(w2):
                        s1[i] = w1 + ' ' * (len(w2) - len(w1))
                # Otherwise, create an empty filler word of the right width
                if not w1:
                    s1[i] = '*' * len(w2)
                if not w2:
                    s2[i] = '*' * len(w1)
            s1 = map(lambda x: colored(x, 'red'), s1)
            s2 = map(lambda x: colored(x, 'red'), s2)
            ref_tokens += s1
            hyp_tokens += s2
    if prefix1: ref_tokens.insert(0, prefix1)
    if prefix2: hyp_tokens.insert(0, prefix2)
    if suffix1: ref_tokens.append(suffix1)
    if suffix2: hyp_tokens.append(suffix2)
    print(' '.join(ref_tokens))
    print(' '.join(hyp_tokens))

def mean(seq):
    """Return the average of the elements of a sequence."""
    return float(sum(seq)) / len(seq) if len(seq) > 0 else float('nan')

def print_wer_vs_length():
    """Print the average word error rate for each length sentence."""
    values = wer_bins.values()
    avg_wers = list(map(lambda x: (x[0], mean(x[1])), values))
    for length, _ in sorted(avg_wers, key=lambda x: x[1]):
        print('{0:5d} {1:f}'.format(length, avg_wers[length]))
    print('')
```