kaldi 资料归纳和总结

wbglearn（吴本谷） version 0.7

2016 年 9 月 5 日

目录

目录.....2
0 开篇前的话.....4
1 kaldi 的介绍.....5
1.1 kaldi 简介.....5
1.2 kaldi 的特色.....5
1.3 kaldi 的声学模型.....5
1.4 kaldi 所用到的库介绍:.....5
2 kaldi 的安装和出现错误的解决方案.....6
2.1 安装前的配置.....6
2.2 安装.....6
2.2.1 Unbunt.....6
2.2.2 Cygwin.....6
2.2.3 服务器或者工作站.....7
3 kaldi 的使用.....8
3.1 总述.....8
3.2 kaldi 里各种数据库的介绍.....15
3.3 yesno 的例子.....16
3.4 timit.....16
3.5 rm.....16
3.6 voxforge.....17
3.7 kaldi 上使用 GPU 以及如何安装 cuda.....17
3.8 可以使用的公开数据库.....19
3.9 如何使用自己的数据集.....19
3.10 kaldi 上第一个中文数据库.....20
3.11 对各位语音识别新手的建议.....20
3.12 如何使用 kaldi 工具包使用小数字语料库创建一个简单的 ASR 系统.....23
4 kaldi 主页上的翻译.....34
4.1 数据准备.....35
4.2 特征提取.....51
4.3 声学建模代码.....53
4.4 kaldi 里解码图的构建.....55
4.5 Karel 的深度神经网络训练实现.....59
4.6 Kaldi 中的关键词搜索.....63
4.8 在线识别.....75
4.9 决策树是如何在 kaldi 中使用.....79
4.10 Decision tree internals.....88
4.11 HMM 拓扑结构和转移模型.....92
4.12 kaldi 里的聚类机制.....100
4.13 Decoding-graph creation recipe (training time).....102
4.14 Decoding-graph creation recipe (test time).....104
4.15 kaldi tutorial.....112
5 附录.....136
5.1 kaldi 上搭建 TIMIT 基线系统.....136

5.2 kaldi 里的 voxforge..... 139
5.3 在 vs 2013 中编译 kaldi..... 144
5.4 kaldi 学习联盟群第一次讨论记录.....145
6 其他资料和资源.....153
7 版本更新日志.....154

0 开篇前的话

首先，先打 2 个广告。希望大家看到文档的同时，可以积极参与讨论。

第一个广告：kaldi 学习 语音深度学习群，群号是：367623211。欢迎大家的加入。

第二个广告：语音识别论坛，论坛地址：http://www.threedweb.cn/forum-76-1.html 论坛欢迎大家的发帖和跟帖，欢迎大家积极讨论。

其次，关注我博客（http://blog.csdn.net/wbgxx333）的人应该知道本文档是建立在群里大家的共同努力完成的，特别是对 kaldi 主页翻译的部分，感谢每个参与翻译的人，感谢你们的辛勤劳动和付出，希望可以对学习 kaldi 的人有用。

最后，希望大家可以积极参与讨论，让群和论坛全部活跃起来，也可以使语音识别逐渐成为大家的生活的一部分。

备注：此文档不定期更新，如果您发现有任何问题和疑问，欢迎随时在群里跟我联系或者通过邮箱（wbglearn@gmail.com 或者 354475072@qq.com)与我联系，我们将逐渐完善该文档。

2014.8.16

最近都在对 kaldi 的脚本和源码有兴趣，欢迎大家积极交流。大家的看的时候希望做下笔记，然后写下你自己的理解，希望你可以发给我，我将放在我们这里，供大家学习，希望越多的人加入进来，为未来的人铺好路。此外，大家在学习 kaldi 的过程一定要注重对代码和脚本的学习，学的深一点，不要仅仅会运行。祝大家学习愉快！

2016.9.8

Kaldi 新手群：279295537。欢迎各位新手加入。Kaldi 学习群现在人员满了，需要回答问题才能入群。

1 kaldi 的介绍

1.1 kaldi 简介

kaldi是一个开源的语音识别工具箱，是基于c++编写的，可以在windows和unix平台上编译。

1.2 kaldi 的特色

与文本无关的LVCSR系统；
基于FST的训练和解码；
最大似然训练；
各种各样的线性和映射变换；
有VTLN，SAT的脚本；

1.3 kaldi 的声学模型

支持标准的机器学习训练模型：
线性变换如：LDA HLDA,MLLT/STC；
说话人自适应:fMLLR,MLLR；
支持GMM,SGMMs,DNN

1.4 kaldi 所用到的库介绍:

1.OpenFst：Weighted Finite State Transducer library　(加权有限状态转换器)
2.ATLAS/CLAPACK:标准的线性代数库
3.sph2pipe:由 sph 文件转成其他音频文件
4.srilm:语言模型的工具箱
5.sctk：score benchmark（评价 ASR 基准）

2 kaldi 的安装和出现错误的解决方案

2.1 安装前的配置

安装前你需要对你的 linux 进行配置，你需要安装的软件有：
apt-get
subversion
automake
autoconf
libtool
g++
zlib
libatal
wget
具体安装方法如下：
（1）sudo apt-get install libtool
（2）sudo apt-get install autoconf
（3）sudo apt-get install wget
（4）sudo apt-get install perl
（5）sudo apt-get install subversion
（6）sudo apt-get install build-essential
（7）sudo apt-get install gfortran
（8）sudo apt-get install libatlas-dev
（9）sudo apt-get install libatlas-base-dev
（10）sudo apt-get install zlib1g-dev　(中间有个数字1）
（11）如果报function gensub never defined，则需要安装gawk
　　安装办法：apt-get install gawk

2.2 安装

2.2.1 Unbunt

1. git clone clone https://github.com/kaldi-asr/kaldi.git
2.先回到tool目录下，在命令行输入：make。
Note that "make" takes a long time; you can speed it up by running make
in parallel, for instance "make -j 4" 4是机器的cpu核心数，说明比较好。
3.make 完后，在src目录下：
　　./configure
　　make depend
　　make

2.2.2 Cygwin

32 位的系统稳定，不会出现错误。据说 64 位的会出现 bug。
kaldi在cygwin里安装出现minsize.o错误的解决办法：
在openfst 里面执行 configure --enable-static --disable-shared
然后make 然后到tools 里面make .

备注：这部分有待完善，将在后面的版本中更新。

2.2.3 服务器或者工作站

实测CentOS7.0, 工作站安装。

I 安装前准备

安装前你需要对你的centos进行配置，步骤如下：

yum check-update； 检查更新

yum install -y； 安装找到的更新，确保已安装的软件是最新版的

依赖软件的安装（如果已经安装可以跳过）。

（1）sudo yum install libtool

（2）sudoyum install autoconf

（3）sudo yum install wget

（4）sudo yum install perl

（5）sudo yum install subversion

（6）sudo yum install zlib

（7）sudo yum groupinstall "Development Tools"

II 安装

（1）静态库安装方法（默认）：

cd tools; make

./install_atlas.sh（这里不行就需要安装 atlas-devel.i686 和 atlas-devel.x86_64）

cd ../src; ./configure; %此步骤中只能用configure， 加入--shared会报错：

make depend; make

（2）动态库安装方法：

cd tools; make

将 install_atlas.sh 中的 ../configure $opt --prefix=`pwd`/install || exit 1;

改为：

../configure $opt --prefix=`pwd`/install --shared|| exit 1;

./install_atlas.sh

cd ../src; ./configure --shared 这里如果报错，请将--shared去掉

make depend; make

（待写）

备注：由于各个平台和你自己电脑配置的问题，如果大家遇到什么问题，欢迎在群里交流。

3 kaldi 的使用

3.1 总述

在跑 kaldi 里的样例时，你需要注意三个脚本：cmd.sh path.sh run.sh。下面分别来说，

Cmd.sh 脚本为：

```
# "queue.pl" uses qsub.  The options to it are
# options to qsub.  If you have GridEngine installed,
# change this to a queue you have access to.
# Otherwise, use "run.pl", which will run jobs locally
# (make sure your --num-jobs options are no more than
# the number of cpus on your machine.

#a) JHU cluster options
#export train_cmd="queue.pl -l arch=*64"
#export decode_cmd="queue.pl -l arch=*64,mem_free=2G,ram_free=2G"
#export mkgraph_cmd="queue.pl -l arch=*64,ram_free=4G,mem_free=4G"

#export cuda_cmd=run.pl


#b) BUT cluster options
#export train_cmd="queue.pl -q all.q@@blade -l
ram_free=1200M,mem_free=1200M"
#export decode_cmd="queue.pl -q all.q@@blade -l
ram_free=1700M,mem_free=1700M"
#export decodebig_cmd="queue.pl -q all.q@@blade -l
ram_free=4G,mem_free=4G"

#export cuda_cmd="queue.pl -q long.q@@pco203 -l gpu=1"
#export cuda_cmd="queue.pl -q long.q@pcspeech-gpu"
#export mkgraph_cmd="queue.pl -q all.q@@servers -l
ram_free=4G,mem_free=4G"

#c) run it locally...
export train_cmd=run.pl
```

export decode_cmd=run.pl
export cuda_cmd=run.pl
export mkgraph_cmd=run.pl

大家可以很清楚的看到有 3 个分类分别对应 a，b，c。a 和 b 都是集群上去运行这个样子，c 就是我们需要的。我们在虚拟机上运行的。你需要修改这个脚本。

Path.sh 的内容：
export KALDI_ROOT=`pwd`/../../..
export PATH=$PWD/utils/:$KALDI_ROOT/src/bin:$KALDI_ROOT/tools/openfst/bin:$KALDI_ROOT/tools/irstlm/bin/:$KALDI_ROOT/src/fstbin/:$KALDI_ROOT/src/gmmbin/:$KALDI_ROOT/src/featbin/:$KALDI_ROOT/src/lm/:$KALDI_ROOT/src/sgmmbin/:$KALDI_ROOT/src/sgmm2bin/:$KALDI_ROOT/src/fgmmbin/:$KALDI_ROOT/src/latbin/:$KALDI_ROOT/src/nnetbin:$KALDI_ROOT/src/nnet-cpubin/:$KALDI_ROOT/src/kwsbin:$PWD:$PATH
export LC_ALL=C
export IRSTLM=$KALDI_ROOT/tools/irstlm

在这里一般只要修改 export KALDI_ROOT=`pwd`/../../..改为你安装 kaldi 的目录，有时候不修改也可以，大家根据实际情况。

Run.sh 里大家需要指定你的数据在什么路径下，你只需要修改：
如：
#timit=/export/corpora5/LDC/LDC93S1/timit/TIMIT # @JHU
timit=/mnt/matylda2/data/TIMIT/timit # @BUT
修改为你的 timit 所在的路径。
其他的数据库都一样。

此外，voxforge 或者 vystadial_cz 或者 vystadial_en 这些数据库都提供下载，没有数据库的可以利用这些来做实验。

最后，来解释下 run.sh 脚本。我们就用 timit 里的 s5 来举例阐述：

```
#!/bin/bash

#
# Copyright 2013 Bagher BabaAli,
#                2014 Brno University of Technology (Author: Karel Vesely)
#
# TIMIT, description of the database:
# http://perso.limsi.fr/lamel/TIMIT_NISTIR4930.pdf
#
# Hon and Lee paper on TIMIT, 1988, introduces mapping to 48 training phonemes,
```

```
# then re-mapping to 39 phonemes for scoring:
# http://repository.cmu.edu/cgi/viewcontent.cgi?article=2768&context=compsci
#

. ./cmd.sh
[ -f path.sh ] && . ./path.sh
set -e

# Acoustic model parameters
numLeavesTri1=2500
numGaussTri1=15000
numLeavesMLLT=2500
numGaussMLLT=15000
numLeavesSAT=2500
numGaussSAT=15000
numGaussUBM=400
numLeavesSGMM=7000
numGaussSGMM=9000

decode_nj=5
train_nj=30


echo
============================================
echo "                    Data & Lexicon & Language Preparation
"
echo
============================================================

#timit=/export/corpora5/LDC/LDC93S1/timit/TIMIT # @JHU
timit=/mnt/matylda2/data/TIMIT/timit # @BUT

local/timit_data_prep.sh $timit || exit 1

local/timit_prepare_dict.sh

# Caution below: we insert optional-silence with probability 0.5, which is the
# default, but this is probably not appropriate for this setup, since silence
# appears also as a word in the dictionary and is scored.    We could stop this
# by using the option --sil-prob 0.0, but apparently this makes results worse.
# (-> In sclite scoring the deletions of 'sil' are not scored as errors)
utils/prepare_lang.sh --position-dependent-phones false --num-sil-states 3 \
  data/local/dict "sil" data/local/lang_tmp data/lang

local/timit_format_data.sh

echo
============================================================
```

这个就是之前说的你需要修改的，修改好才可以运行成功，否则会报错

声学模型的参数，这部分我们基本默认不改，除非你自己训练自己的数据库

nj 是指你需要运行 jobs 的数量，可以根据你机器的 CPU 的数量来改变。一般不超过 CPU 的数量

这部分是数据准备部分，在 kaldi 主页翻译中，@v 给大家翻译的足够详细，具体的可以去看那篇译文。

这就是你需要修改的 timit 数据路径。

```
    echo "            MFCC Feature Extration & CMVN for Training and Test set
"
    echo
=====================================================================
```

这部分是提取特征部分，这里只提取了 mfcc，kaldi 里支持 mfcc，plp，pitch 等特征，然后做一

```
    # Now make MFCC features.
    mfccdir=mfcc


    for x in train dev test; do
      steps/make_mfcc.sh --cmd "$train_cmd" --nj 30 data/$x exp/make_mfcc/$x
$mfccdir
      steps/compute_cmvn_stats.sh data/$x exp/make_mfcc/$x $mfccdir
    done
```

这部分是单音素的训练和解码部分。语音识别最基础的部分。

```
    echo
=====================================================================
    echo "                          MonoPhone Training & Decoding
"
    echo
=====================================================================

    steps/train_mono.sh    --nj "$train_nj" --cmd "$train_cmd" data/train data/lang
exp/mono

    utils/mkgraph.sh --mono data/lang_test_bg exp/mono exp/mono/graph

    steps/decode.sh --nj "$decode_nj" --cmd "$decode_cmd" \
      exp/mono/graph data/dev exp/mono/decode_dev

    steps/decode.sh --nj "$decode_nj" --cmd "$decode_cmd" \
      exp/mono/graph data/test exp/mono/decode_test
```

这部分是三音素的训练和解码部分

```
    echo
=====================================================================
    echo "          tri1 : Deltas + Delta-Deltas Training & Decoding
"
    echo
=====================================================================

    steps/align_si.sh --boost-silence 1.25 --nj "$train_nj" --cmd "$train_cmd" \
      data/train data/lang exp/mono exp/mono_ali

    # Train tri1, which is deltas + delta-deltas, on train data.
    steps/train_deltas.sh --cmd "$train_cmd" \
      $numLeavesTri1 $numGaussTri1 data/train data/lang exp/mono_ali exp/tri1

    utils/mkgraph.sh data/lang_test_bg exp/tri1 exp/tri1/graph
```

```
    steps/decode.sh --nj "$decode_nj" --cmd "$decode_cmd" \
      exp/tri1/graph data/dev exp/tri1/decode_dev

    steps/decode.sh --nj "$decode_nj" --cmd "$decode_cmd" \
      exp/tri1/graph data/test exp/tri1/decode_test

    echo
=====================================================================
    echo "                         tri2 : LDA + MLLT Training &
"
    echo
=====================================================================

    steps/align_si.sh --nj "$train_nj" --cmd "$train_cmd" \
       data/train data/lang exp/tri1 exp/tri1_ali

    steps/train_lda_mllt.sh --cmd "$train_cmd" \
      --splice-opts "--left-context=3 --right-context=3" \
      $numLeavesMLLT $numGaussMLLT data/train data/lang exp/tri1_ali exp/tri2

    utils/mkgraph.sh data/lang_test_bg exp/tri2 exp/tri2/graph

    steps/decode.sh --nj "$decode_nj" --cmd "$decode_cmd" \
      exp/tri2/graph data/dev exp/tri2/decode_dev

    steps/decode.sh --nj "$decode_nj" --cmd "$decode_cmd" \
      exp/tri2/graph data/test exp/tri2/decode_test

    echo
=====================================================================
    echo "                    tri3 : LDA + MLLT + SAT Training & Decoding
"
    echo
=====================================================================

    # Align tri2 system with train data.
    steps/align_si.sh --nj "$train_nj" --cmd "$train_cmd" \
      --use-graphs true data/train data/lang exp/tri2 exp/tri2_ali

    # From tri2 system, train tri3 which is LDA + MLLT + SAT.
    steps/train_sat.sh --cmd "$train_cmd" \
      $numLeavesSAT $numGaussSAT data/train data/lang exp/tri2_ali exp/tri3

    utils/mkgraph.sh data/lang_test_bg exp/tri3 exp/tri3/graph

    steps/decode_fmllr.sh --nj "$decode_nj" --cmd "$decode_cmd" \
      exp/tri3/graph data/dev exp/tri3/decode_dev

    steps/decode_fmllr.sh --nj "$decode_nj" --cmd "$decode_cmd" \
```

这部分是在三音素模型的基础上做了 LDA+MLLT 变换。

这部分是在三音素模型的基础上做了 LDA+MLLT+SAT 变换。

```
  exp/tri3/graph data/test exp/tri3/decode_test

echo
=====================================================================
echo "                    SGMM2 Training & Decoding
"
echo
=====================================================================

steps/align_fmllr.sh --nj "$train_nj" --cmd "$train_cmd" \
  data/train data/lang exp/tri3 exp/tri3_ali

#exit 0 # From this point you can run DNN : local/run_dnn.sh

steps/train_ubm.sh --cmd "$train_cmd" \
  $numGaussUBM data/train data/lang exp/tri3_ali exp/ubm4

steps/train_sgmm2.sh --cmd "$train_cmd" $numLeavesSGMM
$numGaussSGMM \
  data/train data/lang exp/tri3_ali exp/ubm4/final.ubm exp/sgmm2_4

utils/mkgraph.sh data/lang_test_bg exp/sgmm2_4 exp/sgmm2_4/graph

steps/decode_sgmm2.sh --nj "$decode_nj" --cmd "$decode_cmd"\
  --transform-dir exp/tri3/decode_dev exp/sgmm2_4/graph data/dev \
  exp/sgmm2_4/decode_dev

steps/decode_sgmm2.sh --nj "$decode_nj" --cmd "$decode_cmd"\
  --transform-dir exp/tri3/decode_test exp/sgmm2_4/graph data/test \
  exp/sgmm2_4/decode_test

echo
=====================================================================
echo "                    MMI + SGMM2 Training &
"
echo
=====================================================================

steps/align_sgmm2.sh --nj "$train_nj" --cmd "$train_cmd" \
  --transform-dir exp/tri3_ali --use-graphs true --use-gselect true data/train \
  data/lang exp/sgmm2_4 exp/sgmm2_4_ali

steps/make_denlats_sgmm2.sh --nj "$train_nj" --sub-split "$train_nj" --cmd
"$decode_cmd"\
  --transform-dir exp/tri3_ali data/train data/lang exp/sgmm2_4_ali \
  exp/sgmm2_4_denlats

steps/train_mmi_sgmm2.sh --cmd "$decode_cmd" \
  --transform-dir exp/tri3_ali --boost 0.1 --drop-frames true \
```

这部分是在三音素模型的基础上做了 sgmm2，sgmm2 是 povey 提出的。

这部分是在三音素模型的基础上做了 sgmm2+MMI。

```
  data/train data/lang exp/sgmm2_4_ali exp/sgmm2_4_denlats \
  exp/sgmm2_4_mmi_b0.1

for iter in 1 2 3 4; do
  steps/decode_sgmm2_rescore.sh --cmd "$decode_cmd" --iter $iter \
    --transform-dir exp/tri3/decode_dev data/lang_test_bg data/dev \
    exp/sgmm2_4/decode_dev exp/sgmm2_4_mmi_b0.1/decode_dev_it$iter

  steps/decode_sgmm2_rescore.sh --cmd "$decode_cmd" --iter $iter \
    --transform-dir exp/tri3/decode_test data/lang_test_bg data/test \
    exp/sgmm2_4/decode_test exp/sgmm2_4_mmi_b0.1/decode_test_it$iter
done

echo
============================================================
echo "                              DNN Hybrid Training & D
"
echo
============================================================

# DNN hybrid system training parameters
dnn_mem_reqs="mem_free=1.0G,ram_free=0.2G"
dnn_extra_opts="--num_epochs 20 --num-epochs-extra 10 --add-layers-period 1
--shrink-interval 3"

steps/nnet2/train_tanh.sh --mix-up 5000 --initial-learning-rate 0.015 \
  --final-learning-rate 0.002 --num-hidden-layers 2   \
  --num-jobs-nnet "$train_nj" --cmd "$train_cmd" "${dnn_train_extra_opts[@]}"
\
  data/train data/lang exp/tri3_ali exp/tri4_nnet

[ ! -d exp/tri4_nnet/decode_dev ] && mkdir -p exp/tri4_nnet/decode_dev
decode_extra_opts=(--num-threads 6 --parallel-opts "-pe smp 6 -l
mem_free=4G,ram_free=0.7G")
steps/nnet2/decode.sh --cmd "$decode_cmd" --nj "$decode_nj"
"${decode_extra_opts[@]}" \
  --transform-dir exp/tri3/decode_dev exp/tri3/graph data/dev \
  exp/tri4_nnet/decode_dev | tee exp/tri4_nnet/decode_dev/decode.log

[ ! -d exp/tri4_nnet/decode_test ] && mkdir -p exp/tri4_nnet/decode_test
steps/nnet2/decode.sh --cmd "$decode_cmd" --nj "$decode_nj"
"${decode_extra_opts[@]}" \
  --transform-dir exp/tri3/decode_test exp/tri3/graph data/test \
  exp/tri4_nnet/decode_test | tee exp/tri4_nnet/decode_test/decode.log


echo
============================================================
```

这部分是 povey 版本的 dnn 模型，不建议使用。

这部分是 dnn+sgmm 模型

```
    echo "                    System Combination (DNN+SGMM)
"
    echo
================================================================================

    for iter in 1 2 3 4; do
       local/score_combine.sh --cmd "$decode_cmd" \
         data/dev data/lang_test_bg exp/tri4_nnet/decode_dev \
         exp/sgmm2_4_mmi_b0.1/decode_dev_it$iter
exp/combine_2/decode_dev_it$iter

       local/score_combine.sh --cmd "$decode_cmd" \
         data/test data/lang_test_bg exp/tri4_nnet/decode_test \
         exp/sgmm2_4_mmi_b0.1/decode_test_it$iter
exp/combine_2/decode_test_it$iter
    done

    echo
================================================================================
    echo "                    DNN Hybrid Training & Decoding (Karel'
"
    echo
================================================================================

    local/run_dnn.sh

    echo
================================================================================
    echo "                    Getting Results [see RESULTS file]
"
    echo
================================================================================
    bash RESULTS dev
    bash RESULTS test
    echo
================================================================================
    echo "Finished successfully on" `date`
    echo
================================================================================
    exit 0
```

这部分是 karel 的 dnn 模型。通用的深度学习模

这部分是得到上述模型的最后的识别结果

此外，有人修改这个 dnn 的脚本，具体修改和详细的请看：http://blog.csdn.net/sytxsybm/article/details/18556403，我会在附录5.1中给出。

3.2 kaldi 里各种数据库的介绍

1、babel : IARPA Babel program 语料库来自巴比塔项目，主要是对低资源语言的语音识别和关键词检索例子，包括普什语，波斯语，土耳其语，越南语等等。据文献上讲效果不太好，wer 达到50以上。

2、sre08：" Speaker Recognition Evaluations" 说话人识别。

3、aurora4: 主页：http://aurora.hsnr.de/ 研究各种噪音的。带噪音的语音识别--健壮的语音识别项目。包括说话人分离，音乐分离，噪声分离。

4、hkust:香港大学的普通话语音识别

5、callhome_egyptian: 埃及的阿拉伯语语音识别

6、chime_wsj0: chime 挑战项目数据，这个挑战是对电话，会议，远距离麦克风数据进行识别。

7、fisher_englist：英语的双声道话音。

8、gale_arabic:全球自动语言开发计划中的阿拉伯语。

9、gp:global phone项目，全球电话语音：19种不同的语言，每种15-20小时的语音

10、lre:包括说话人识别，语种识别

11、wsj:wall street journal 华尔街日报语料库，似乎所有的脚本都是这个东西开始的。

12、swbd:Switchboard 语料库

13、tidigits:男人，女人，孩子说的不同的数字串语音的识别训练，

14、voxforge:开源语音收集项目

15、timit:不同性别，不同口音的美国英语发音和词汇标注，包括 Texas Instruments (TI) 和 Massachusetts Institute of Technology (MIT), 所以叫timit

16、tedlium: 数据在这里

http://www.openslr.org/resources/7/TEDLIUM_release1.tar.gz

TED talks英语语音数据，由Laboratoire d'Informatique de l'Université du Maine (LIUM) 创建

17、vystadial_cz:dataset of telephone conversations in Czech 希腊人搞的电话语音识别数据

18、vystadial_en:dataset of telephone conversations in English 希腊人搞的电话语音识别数据

19、yesno: 各种yes,no 两个词的语音识别，归入命令词语音识别吧。

20、rm:DARPA Resource Management Continuous Speech Corpora 。

3.3 yesno 的例子

1. 把 waves_yesno.zip.gz 复制到 yesno/s3 目录下，然后使用
sudo yumzip waves_yesno.zip.gz
tar –xvf waves_yesno.tar
2.运行./run.sh。

3.4 timit

1 去 run.sh 里修改 timit 的目录，修改为你自己 timit 数据所在的目录；
2 根据之前的修改 cmd.sh 和 path.sh；
3 运行 run.sh 即可。

3.5 rm

1 去 run.sh 里修改 rm 的目录，修改为你自己 rm 数据所在的目录；

2 根据之前的修改 cmd.sh 和 path.sh；
3 运行 run.sh 即可。

3.6 voxforge

1.用 getdata.sh 收集数据，数据大小大约为 11G 左右，大家保证有足够多的硬盘空间；
2.根据之前的修改 cmd.sh 和 path.sh;
3.你可以顺利运行 run.sh 脚本。

此外，附录中有个别人跑 voxforge 的例子，需要的人可以去附录 5.2 看一下。

3.7 Kaldi 上使用 GPU 以及如何安装 cuda

补充：如何查询自己的 gpu 型号支不支持 cuda？
http://www.nvidia.cn/object/cuda_learn_products_cn_old.html

本博客是在@冒顿和群里的一个同学的指导下完成的。特此感谢……

众所周知，kaldi 对硬件的要求，希望大家一开始都弄个好的机器，方便后面自己在 kaldi 里大展身手。下面是怎么去装 cuda 和在 kaldi 中怎么使用。

特别提醒，如果显卡是 gtx970 或者 980 的话就不是下面这个驱动啊。大家要注意的哦。网站为：

http://developer.download.nvidia.com/compute/cuda/6_5/rel/installers/cuda-repo-ubuntu1404-6-5-prod_6.5-19_amd64.deb。

我网盘的地址：http://pan.baidu.com/s/1i3mpdad。也可以去这个地址下载。

首先说明我们的显卡是 geforce ，其他的显卡是否一样不知道。以下是安装步骤和注意事项：

1、 安装软件： apt-get install ppa-purge
2、 增加安装源： apt-add-repository ppa:xorg-edgers
3、更新安装源： apt-get update
4、安装 nvidai 显卡驱动： apt-get install nvidia-current nvidia-settings
5、下载 cuda 安装包，这里说明下，如果是 12.04 就修改 1404 为 1204 哈。
wget http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1404/x86_64/cuda-repo-ubuntu1404_6.5-14_amd64.deb
6、安装 cuda 安装包，这个也是啊，上面下了什么就改什么名字啊。
dpkg -i cuda-repo-ubuntu1404_6.5-14_amd64.deb
7. apt-get update
8、安装 cuda：apt-get install cuda
9、在 kaldi 的 src 中 make clean，主要是为了清楚我们之前的编译结果

10、./configure ：重新生成配置文件，这时会自动检查是否安装了 cuda，这里会有显示是不是有 cuda。

到 src/cudamatrix 中 Makefile 中修改 CUDA_ARCH =-gencode 那一行. 我用的是 geforce 9600 显卡计算能力为 1.1，则修改 compute=11,code=sm_11, 把 TESTFILES 改为 BINFILES

11、make all 即可

运行 src/cudamatrix/目录下运行 ./cu-vector-test，如果不报错，则表明可以用 GPU 进行矩阵运算了。

如果此方法不通，大家尝试去这里看下：
http://blog.csdn.net/abcjennifer/article/details/23016583

后面我将更新 ubuntu 14.04+cuda 6.5.谢谢……

2014.11.19 晚更新：

平台：64 位 ubuntu 14.04 物理机 显卡是 gtx 980，装 cuda 6.5.感谢@神牛的指导。

1.安装所需要的库：

[plain] view plaincopy

```
1.  sudo apt-get install freeglut3-dev build-essential libx11-dev libxmu-dev
2.  libxi-dev libgl1-mesa-glx libglu1-mesa libglu1-mesa-dev
```

2.删除所来的 nvidia 驱动的库：

sudo apt-get remove --purge nvidia*

sudo nano /etc/modprobe.d/blacklist.conf

在终端的末尾添加：

blacklist amd76x_edac （这个本身就有，就在这后面添加后面的哈）

blacklist vga16fb

blacklist nouveau

blacklist rivafb

blacklist nvidiafb

blacklist rivatv

3.这里首先要下载 cuda_6.5.19_linux_64.run，这个我放到我百度网盘里哈。地址：http://pan.baidu.com/s/1nt1aaed。也可以去官网下载。

关掉图像界面，按 Ctrl+Alt+F1，登陆输入账号和密码，然后：

sudo service lightdm stop

chmod +x cuda_6.5.19_linux_64.run

sudo ./cuda_6.5.19_linux_64.run

这样后就会显示 cuda 的一些东西，你直接按 enter 到底，然后根据那个输入 acept，y 等等这些啊，尽量同意那些啊。最后就告诉你：

Driver: Installed

Toolkit: Installed In /Usr/Local/Cuda-6.5

Samples: Installed.

这样基本就安装好了啊。

然后再 kaldi 的 src 目录下./configure ，会显示 cuda 是不是安装好了。然后 make all，最后到 src 的 cudamatrix 目录下运行：./cu-vector-test，就会显示 gpu 什么的啊。

至此，你就大功告成了啊。你可以运行 kaldi 里的 dnn 模块啊。enjoy it。

下面就可以使用 gpu 来运行我们的 kaldi 程序了。相信你很期待这个速度吧，加油吧……

欢迎使用其他显卡的同学分享你们的经验，此博文将收录到 kaldi 全部资料 V0.5 里。

3.8 可以使用的公开数据库

首先感谢 povey 和他的同事的贡献，现在在英文数据集下已经有很多的数据了，期待某天中文数据集可以这么开源。下面分别说下可以下的数据集

librispeech：大约为 1000 小时的英文数据，可以在 http://www.openslr.org/12/下载。

此外，还有其他的一些数据都可以在这个网站上下载：http://www.openslr.org/resources.php

当然，在这提醒大家的时候，如果自己没有足够的资源，比如内存或者 gpu 或者服务器，最好还是别去尝试这些数据。毕竟这些耗费的硬件也很多。作为学生或者初学者，建议使用 timit 等一些小数据的数据。

这里还分享一个重要的资料就是：http://www.kaldi-asr.org/downloads/tree/trunk/

这里有很多经过 kaldi 跑出来的结果，如果没有这些数据的话可以下载来看看，也是一个非常好的资料。具体的大家可以去官网资料。

3.9 如何使用自己的数据集

这里暂时这么说下这个问题。以来最近遇到几个问这个问题的。拿到一个新的数据集，对于 kaldi 来说，

第一步要做的就是数据准备阶段，这个具体可以看下后面翻译中《数据准备》这部分的内容，看如何去准备和怎么去准备，这里可能需要自己写些脚本。

第二步就是去套用 kaldi 里本身的提取特征和 gmm 或者 dnn 的脚本来跑一下自己的数据，然后去观察下效果。

第三步根据你得到的效果一步一步去看是不是真的好，之前就遇到有人问单音素的效果比三音素的效果好等等各类的问题，这些需要你不断的去学习，不断的寻找为什么，如果你的语音识别基础足够好的话，你应该可以找到这些问题的。

大概的过程就是这些了，当然遇到一个懂的人会帮助很大，没有的话就去群里多问问，相信总会解决的。

3.10 kaldi 上第一个中文数据库

今天在清华大学 cslt 实验室王东老师的分享下，kaldi 终于有了免费的中文语音识别的例子，网址为：https://github.com/kaldi-asr/kaldi/tree/master/egs/thchs30。各位可以根据这个来训练自己的模型。

再次谢谢王东老师的付出。

此外，清华大学 cslt 分享的数据库还有很多，网址为：

http://pan.baidu.com/s/1dEhUghz

3.11 对各位语音识别新手的建议

由于工作的原因，很长时间不能更新博客和管理 kaldi 群，每天看着 kaldi 群的人数不断增长，由衷的为从事语音感到自豪，希望在我博客和群里能得到你们想要的，但我同时拒绝伸手党。这几年语音的发展很迅速，所以导致更多的人来学习：

下面主要从 2 个方面来说明，一个是从学生角度，一个是从工业角度。希望以后问怎么学习和怎么研究的人会越来越少，当然我这里也只是抛砖引玉。

学生角度

如果你是本科生，你或许付出的努力会更多。如果你是研究生，最好是数学和英语好，要不然你同样付出很多努力。

1、基础部分

默认你是研究生，你学过基本的矩阵理论和随机过程，一些简单的模式识别原理，这样你或者利于理解识别的原理。默认你能够阅读一般的英文文献，不要害怕专有名词，等你见多了，自然能看懂。也行英文六级过了，一般的文章能看得懂，只要语句知道怎么翻译。这就是数学和英文基础，希望你具备。

此外，默认你具有一些计算机基础，比如 c 语言和 c++，和其他的一些脚本语言如 Python，shell，perl 等基础。如果你不会的话，不要怕，去网上听公开课，也许你听的够多，也就会了。此外，你一定要会 linux 的基本命令，仅仅是基本命令，慢慢的你也会其他高阶的命令，这个只要你慢慢用。

如果你不具备这些，可以早点学；如果你具备，那你已经达到基础的一步。如果你觉得特别难，赶紧放弃吧。其他的路也是一条很好的路，其实学语音识别需要的东西太多，所以你要有足够的思想准备。如果你的老师或者其他人只丢给

你一个课题，如果你根本不感冒，那就劝你直接放弃吧。也许你学习其他的东西，也可以活得很好。再次强调，这也许是一条不归路，或者说很长很长的路，远比你之前遇到的都难。

2.进阶

哦，忘记一个非常重要的东西，就是数据结构和算法。如果你没有学过，赶紧学，这个东西非常非常重要。

这也许是第一个进阶的 东西，这个东西很奇妙，如果你有兴趣，你可以做的很好。

接下来你需要看一些语音的基础知识，知道语音的一些基本处理或者信号的一些基本处理。如果你根本不懂傅里叶变换或者你根本不知道为什么那样，或许你需要思考。这个是大学的信号与系统，数字信号处理，通信原理等等的一些基础知识。

然后，你可以看下语音识别的一些基本知识，当然这个中文的课本也没一个说的好的。你可以先看看机器学习，比如逻辑斯特回归，支持向量机，高斯混合模型，隐马尔科夫模型。当然还有信号检测和估计的一些理论，多维高斯模型和最大似然估计等等。结合你的中文课本和 htk 中文版前三章，试试看，能不能更好的理解。当然你可以看 kaldi 主页的内容，看看一些简单的东西，比如课外的一些语音课程，这个只有 ppt 和一些材料，很少有视频，如果你们学校有老师讲那就更好。比如上海交通大学的俞凯老师。

这里，默认你已经知道一些基本的原理了，也许你不够具体，不要怕，继续走，也许某一个瞬间就明白了。多思考多想想，这个是最好的建议。

3.高阶

你可以去看看 kaldi 的脚本和代码或者 htk 的脚本和代码或者其他平台，再根据你的理论试试结合，如果你不懂，多问问，也许你慢慢就积累了，慢慢就会了。如果有人带你，那就更好了。慢慢的你就成为高手了。

这时候你理解了 GMM-HMM 那套后，就去看 DNN 那一套吧，甚至更高深的东西。

如果你是学生，发现上面跟你的差别特别大，如果你没兴趣的话，最好放弃，别为难自己。要学的的确很多，但为了你未来的路好走，或许你必须这样辛苦。

工业角度

如果你只是想使用语音识别，你最好去调别人家的 SDK。如果你的确需要去做，你可以看看上面的那么多条，你具备吗？

其次，你真的很想很想做，第一个问题来，语料有吗?买，的确很贵，不买，做不出来。然后，你有人吗？没有几个懂得人慢慢去弄也很麻烦，所以的确不是刚需，你可以尝试去调别人的 SDK。

经常会被问的问题：

1.我就想做个我说一句话，然后就识别为文字。

有时候问这个问题，我的确挺气的。你们知道这背后有多少东西，没有那么简单

就搞出来的。统一回答：你需要找到训练语料，然后训练一个模型，然后再弄解码，然后再出来。这个需要训练模型，需要语料，你有吗？

2.不是有 kaldi 或者 htk 了吗？再说王东老师都开源中文的识别例子了？自己可以搞了。
然而并不是，王东老师弄这个例子是为了促进中文语音识别的发展，真的特别感谢王东老师。我当时读研的时候什么数据库都没，就自己看。但这个语料仅仅只是科研，你想要工业应用，还差十万八千里。开源的的确存在，你需要有自己的数据库，然后利用开源来搭建一个适合自己情景的系统。

3.机器真的要 GPU 吗？
现在都知道 dnn 或者 LSTM 的效果好，这些都是用显卡和语料烧出来的，也就是用钱喂出来的。所以这个的确需要钱……

基本对于工业的人来说已经够了，应该足够自己做判断了。再次强调语音识别还是一个高门槛的事情，谨慎入。当然土豪就无所谓了。

以上对于大家来说，可以做判断了，聪明的人做聪明的判断。
希望更多的人加入到语音这个大家庭来，共同繁荣语音事业。

后续会根据反馈情况，逐步反馈。

看了群里 n 多初级的问题，这里汇总下：
1.请问有没有可能做爬虫去网上下载训练数据？商业数据还是太贵了…
答：
网上的训练数据问题：1）采样率是否统一，有没有做过重采样；2）说话人是否足够；3）是否有标注；4）若有声读物标注是一长段，不太适合训练，需搞短点；5）电影里的背景声音问题；6）新闻的倒是有可能用，但也需要很多人工切分，处理等。

2.自己的数据怎么弄？
kaldi 中有很多的例子，你可以参考 kaldi 里的任何一个例子，英文的可以参考的很多，中文的就参考王东老师提供的 thchus30。主要是数据准备阶段，其他的都可以复用脚本，但不一定适合自己，别忘了调整参数。

3.12 如何使用 kaldi 工具包使用小数字语料库创建一个简单的 ASR 系统

1.介绍

对于初学者来说，文章介绍了如何循序渐进的在 Kaldi 工具包上使用自己的一组简单的数据创建一个简单 ASR（自动语音识别）系统。当我开始学习 kaldi 时，我是非常喜欢读这样的文章。这是在所有基于语音识别对象和脚本编程的情况下，我作为一个业余的经历。

如果你曾经浏览过 Kaldi 官网的教程，或许会觉得有点失落。别担心，我的作品（工作）可能是你不错的选择。您将学习如何安装 Kaldi，如何使它工作以及如何使用您自己的语料运行 ASR 系统。然后，你会得到你的第一个语音解码结果。

首先，要搞懂什么 Kaldi 实际上是什么，为什么你需要使用它而不是别的平台。在我看来，学习 Kaldi 需要非常深厚的语音识别和 ASR 系统的知识。这也是很好的了解脚本语言（bash,Perl，python）的基础知识。 C ++可能在未来是很有用的（或许你会想在源代码中做一些修改）。

相关知识详见:
http://kaldi.sourceforge.net/about.htmlhttp://kaldi.sourceforge.net/tutorial_prereqs.html

2.环境

要求 1--采用 Linux 系统

虽然可以在 Windows 上使用 Kaldi，不过大部分人的经验让我确信使用 Linux 将使你的工作量与问题少很多。我使用的 Ubuntu 的 14.10。这是一个资源丰富而稳定的 Linux 系统，我强烈推荐。当你终于有了你自己能够正常运行的 Linux 系统，请打开一个终端，并安装一些必要的东西（如果你还没有的话）：

必备的

Atlas- 在线性代数领域的自动化和优化计算

Autoconf- 在不同的操作系统上进行软件编译

Automake- 创建可移植的 Makefile 文件

git- 分布式版本控制系统

libtool-创建静态和动态库

svn- 版本控制系统，对 kaldi 的下载和安装非常重要

wget- 采用 HTTP,HTTPS 和 FTP 协议进行数据的转换

zlib- 数据压缩

gawk- 一种用于在文件和数据流中进行搜索和处理的编程语言

可能需要安装的

bash- Unix 的 shell 和脚本程序语言

grep- 命令行使用程序，用于正则表达式搜索纯文本数据

make- 自动从源程序中生成可执行程序和库

perl- 动态程序语言，可以完美的处理文本文件程序

到此，操作系统和 liunx 所需的所有工具都已经准备好

3.下载 Kaldi

仔细阅读这个网址上的指示：http://kaldi.sourceforge.net/install.html。如果你不清楚怎么使用 GIT，请阅读:http://kaldi.sourceforge.net/tutorial_git.html。我将 kaldi 安装在这个目录下面：/home/{user}/kaldi-trunk.

4.Kaldi 的目录结构

Kaldi-trunk-kaldi 的主目录，包含以下部分：

egs- 例子脚本，帮助你快速建立 ASR 系统，里面包含 30 个比较流行的语音库（文档附录在每一个工程里面）

misc- 额外的工具和应用，对于一般的 kaldi 功能没有必要

src- kaldi 的源代码

tools- 有用的组成部分和外部工具

windows- 在 windows 上运行 kaldi 的工具.

很明显，最重要的部分是 egs。在这里你可以创建你自己的 ASR 系统。

5.自己的例程

对于本教程的目的，假设您想我一样有一组简单的数据集（如下所述，在 6.1。声音数据部分）。然后尝试将我的每一个操作植入到你自己的项目。如果你完全没有任何音频数据或你想按照我的方式学习，可随时录制自己的声音-这对于学习 ASR 会更有帮助。现在我们开始。

前提：你已经有了一定数量的包含不同说话人的的数字音频数据，每一个音频文件是一个完整的句子。

目的：你想把你的音频数据分成训练部分和测试部分，搭建一个 ASR 系统并且对它进行训练和测试，得到一些解码结果。

首要任务：首先在 kaldi/egs/目录下创建一个名为 digits 的文件夹，这是你存放有关你工程的所有文件的地方。

6.数据准备

6.1.音频数据

假设你想根你自己的音频数据搭建一个 ASR 系统。例如，你有 100 个音频文件，文件格式为 WAV。每一个文件包含 3 个英文数字，一个接一个。这些文件已一种可以识别的方式命名（例如 1_5_6.WAV，意味着这个语料中为 1,5,6 的发音）并把它放在可以识别特定说话人的文件夹中（有可能你记录的同一个说话人在不同的信噪比下的音频，记住把他们放在在不同的文件夹中）。

所以总结起来，我的示范性的数据集看起来是这样的：

10 个不用得说话人（ASR 系统必须优不同的说话人进行训练和测试，说话人越多越好）

每一个说话人有 10 个句子

总共 100 个句子（100 个.wav 的文件放在 10 个特定说话人的文件夹中，每一个文件夹中邮 10 条句子）

一共 300 个单词（数字 1—9）

每一个句子包含 3 个英文数字。

无论你的第一个数据集是什么样子的，设定为我例程的格式。小心大数据集合复杂的语法，最好从简单的开始。开始阶段，使用只包含数字的句子是最好的。

任务

进入到 kaldi-trunk/egs/digits 目录下创建 digits_audio 文件夹，在 kaldi-trunk/egs/digits/digits_audio 文件夹下再创建两个文件夹 train 和 test。选择一个说话人的句子来代表测试数据集。用选中的说话人的说话人编码（speakerID）作为名字在 kaldi-trunk/egs/digits/digits_audio/test 目录下建立一个新的文件夹。然后把有关这个说话人的音频文件全都放进去。

把剩下的 9 个说话人的音频文件放在 train 下，这就是你的训练数据集。同样为每一个说话人建立二级文件夹。

6.2.声学数据

现在你需要创建一些 text 文件使 kaldi 与你的音频文件进行联系，下面这些文件是必须创建的：

任务：在 kaldi-trunk/egs/digit 目录下，创建一个文件夹 data。然后再该文件夹下创建子文件价 test 和 train。在子文件夹下创建如下文件（所以现在你要在 test 和 train 下一相同的方式创建子文件，但是这些子文件涉及两个不同的数据集）：

a.)spk2gender 此文件夹表明说话人的性别。向我们假设的一样，‘speakerID’是每一个说话人的独有的名字（再这样情况下，它也可以是‘recordingID’，每一个说话人在录制中只有一个音频数据文件）。在我的例子中分别有 5 个男士和 5 个女士。（f=男士，m=女士）

PATTERN: <speakerID><gender>
----- exemplary spk2gender starts -----
cristine f
dad m
josh m
july f
and so on...
----- exemplary spk2gender ends -----

b.)wav.scp 这个文件连接每一句话，如果你坚持我的命名方式，‘utteranceID’与‘speakerID’（说话人文件夹名字）是一样的，只不过是*.wav 文件名没有以‘.wav’结尾（来看一下下面的例子）

PATTERN: <uterranceID><full_path_to_audio_file>
----- exemplary wav.scp starts -----
dad_4_4_2
/home/{user}/kaldi-trunk/egs/digits/digits_audio/train/dad/4_4_2.wav

july_1_2_5
/home/{user}/kaldi-trunk/egs/digits/digits_audio/train/july/1_2_5.wav
july_6_8_3
/home/{user}/kaldi-trunk/egs/digits/digits_audio/train/july/6_8_3.wav
and so on...
----- exemplary wav.scp ends -----

c.) test 该文件包含每一个句子所对应的文本信息

PATTERN: <uterranceID><text_transcription>
----- exemplary text starts -----
dad_4_4_2 four four two
july_1_2_5 one two five
july_6_8_3 six eight three
and so on...
----- exemplary text ends -----

d.) utt2spk 这个文件夹告诉 ASR 系统哪一个句子属于哪个特定的说话人

PATTERN: <uterranceID><speakerID>
----- exemplary utt2spk starts -----
dad_4_4_2 dad
july_1_2_5 july
july_6_8_3 july
and so on...
----- exemplary utt2spk ends -----

e.) corpus.txt 这个文件夹有一个稍微不同的目录，在 kaldi-trunk/egs/digits/data 下创建另一个文件夹‘local’。在 kaldi-trunk/egs/digits/data/local 下创建一个文件 corpus.txt，它应该包含每一个单个句子的所对应的出现在你的 ASR 系统中的文本信息（在我们的例子中它应该包含来至于 100 个句子的 100 行信息）。

PATTERN: <text_transcription>
----- exemplary corpus.txt starts -----
one two five
six eight three
four four two
and so on...
----- exemplary corpus.txt ends -----

6.3.语言数据

本节中所涉及到的语言模型文件也是 ASR 系统中必要的一部分。具体参考 http://kaldi.sourceforge.net/data_prep.html（每一个文件都有详细的描述）。

任务：在 kaldi-trunk/egs/digits/data/local 目录下，创建一个新的文件夹‘dict’。在 kaldi-trunk/egs/digits/data/dict 在创建如下文件：

a.)lexicon.txt 这个文件包含你的字典里的每一个单词的音素

PATTERN: <word><phone 1><phone 2> ...
----- exemplary lexicon.txt starts -----
!SIL sil
<UNK> spn

eight ey t
five f ay v
four f ao r
nine n ay n
one hh w ah n
one w ah n
seven s eh v ah n
six s ih k s
three th r iy
two t uw
zero z ih r ow
zero z iy r ow
----- exemplary lexicon.txt ends -----

b.) nonsilence_phones.txt 这个文件列出了你工程中的所有的非静音音素
PATTERN: <phone>
----- exemplary nonsilence_phones.txt starts -----
ah
ao
ay
eh
ey
f
hh
ih
iy
k
n
ow
r
s
t
th
uw
w
v
z
----- exemplary nonsilence_phones.txt ends -----

c.) silence_phones.txt 这个文件列出了静音音素
PATTERN: <phone>
----- exemplary silence_phones.txt starts -----
sil
spn
----- exemplary silence_phones.txt ends -----

d.) optional_silence.txt 这个文件列出了可选择的静音音素

PATTERN: <phone>
----- exemplary optional_silence.txt starts -----
sil
----- exemplary optional_silence.txt ends -----

7.工程定稿

运行脚本前的最后一章，你的工程将会变得完整。

7.1.工具附件

你需要添加在例子脚本中广泛使用的 kaldi 工具箱。

任务：在 kaldi-trunk/egs/wsj/s5 目录下拷贝出两个文件夹（注意拷贝所有内容）：‘utils’和‘steps’，并把它们放在你的 kaldi-trunk/egs/digits 目录下。你还可以为你的这些目录建立连接。你可以在 kaldi-trunk/egs/voxforge/s5 中找到相似的例子。

7.2.评分脚本

这个脚本可以帮助你得到解码结果。

任务：从 kaldi-trunk/egs/voxforge/local 目录下拷贝 score.sh 脚本到你工程中相同的位置（注意，必须是相同的位置：kaldi-trunk/egs.digits/local）

7.3. SRILM 安装

你还需要加载在我的 SRILMl 例子中使用到的语言模型工具包。

任务：关于如何安装，请仔细阅读 kaldi-trunk/tools/install_srilm.sh 里面的所有内容。

7.4. 配置文件

在这里，创建配置文件不是必须的，但是它对你将来的学习是一个好的习惯。

任务：在目录 kaldi-trunk/egs/digits 目录下创建一个名为‘conf’的文件夹。在 kaldi-trunk/egs/digits/conf 目录下创建两个文件（关于一些在解码和 mfcc 特征提取过程中的配置修改——从/egs/voxforge 下拷贝）

a.) decode.config

----- exemplary decode.config starts -----
first_beam=10.0
beam=13.0
lattice_beam=6.0
----- exemplary decode.config ends -----

b.) mfcc.conf

----- exemplary mfcc.conf starts -----
--use-energy=false
----- exemplary mfcc.conf ends -----

8.运行脚本

你的第一个在 kaldi 环境下的 ASR 系统基本上已经完成，最后的工作就是准备运行脚本来搭建你自己设定的 ASR 系统。为了方便大家理解，我在已经准备好的脚本上做了一些注释。

MONO-单音素训练，

TRI1-简单的三音素训练（第一个三音素训练），这两种方法足以显示出仅使用数字词汇和小规模的训练数据集在解码结果上的不同。

任务：在 kaldi-trunk/egs/digits 目录下创建 3 个脚本：

a.) cmd.sh

```
----- cmd.sh script starts here ----------------------------------------------------
# Setting local system jobs (local CPU - no external clusters)
export train_cmd=run.pl
export decode_cmd=run.pl
----- cmd.sh script ends here ------------------------------------------------------
```

b.) path.sh

```
----- path.sh script starts here ----------------------------------------------------
# Defining Kaldi root directory
export KALDI_ROOT=`pwd`/../..

# Setting paths to useful tools
export
PATH=$PWD/utils/:$KALDI_ROOT/src/bin:$KALDI_ROOT/tools/openfst/bin:
$KALDI_ROOT/src/fstbin/:$KALDI_ROOT/src/gmmbin/:$KALDI_ROOT/src/f
eatbin/:$KALDI_ROOT/src/lm/:$KALDI_ROOT/src/sgmmbin/:$KALDI_ROO
T/src/sgmm2bin/:$KALDI_ROOT/src/fgmmbin/:$KALDI_ROOT/src/latbin/:$P
WD:$PATH

# Defining audio data directory (modify it for your installation directory!)
export DATA_ROOT="/home/{user}/kaldi-trunk/egs/digits/digits_audio"

# Variable that stores path to MITLM library
export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$(pwd)/tools/mitlm-svn/lib

# Variable needed for proper data sorting
export LC_ALL=C
----- path.sh script ends here ------------------------------------------------------
```

c.) run.sh

```
----- run.sh script starts here ----------------------------------------------------
#!/bin/bash
. ./path.sh || exit 1
. ./cmd.sh || exit 1
nj=1            # number of parallel jobs - 1 is perfect for such a small data set
```

```
lm_order=1      # language model order (n-gram quantity) - 1 is enough for digits grammar

# Safety mechanism (possible running this script with modified arguments)
. utils/parse_options.sh || exit 1
[[ $# -ge 1 ]] && { echo "Wrong arguments!"; exit 1; }
# Removing previously created data (from last run.sh execution)
rm -rf exp mfcc data/train/spk2utt data/train/cmvn.scp data/train/feats.scp data/train/split1 data/test/spk2utt data/test/cmvn.scp data/test/feats.scp data/test/split1 data/local/lang data/lang data/local/tmp data/local/dict/lexiconp.txt
echo
echo "===== PREPARING ACOUSTIC DATA ====="
echo
# Needs to be prepared by hand (or using self written scripts):
#
# spk2gender    [<speaker-id><gender>]
# wav.scp       [<uterranceID><full_path_to_audio_file>]
# text          [<uterranceID><text_transcription>]
# utt2spk       [<uterranceID><speakerID>]
# corpus.txt    [<text_transcription>]
# Making spk2utt files
utils/utt2spk_to_spk2utt.pl data/train/utt2spk > data/train/spk2utt
utils/utt2spk_to_spk2utt.pl data/test/utt2spk > data/test/spk2utt
echo
echo "===== FEATURES EXTRACTION ====="
echo
# Making feats.scp files
mfccdir=mfcc
# utils/validate_data_dir.sh data/train     # script for checking if prepared data is all right
# utils/fix_data_dir.sh data/train          # tool for data sorting if something goes wrong above
steps/make_mfcc.sh --nj $nj --cmd "$train_cmd" data/train exp/make_mfcc/train $mfccdir
```

```
steps/make_mfcc.sh --nj $nj --cmd "$train_cmd" data/test exp/make_mfcc/test
$mfccdir
# Making cmvn.scp files
steps/compute_cmvn_stats.sh data/train exp/make_mfcc/train $mfccdir
steps/compute_cmvn_stats.sh data/test exp/make_mfcc/test $mfccdir
echo
echo "===== PREPARING LANGUAGE DATA ====="
echo
# Needs to be prepared by hand (or using self written scripts):
#
# lexicon.txt               [<word><phone 1><phone 2> ...]
# nonsilence_phones.txt     [<phone>]
# silence_phones.txt        [<phone>]
# optional_silence.txt      [<phone>]
# Preparing language data
utils/prepare_lang.sh data/local/dict "<UNK>" data/local/lang data/lang
echo
echo "===== LANGUAGE MODEL CREATION ====="
echo "===== MAKING lm.arpa ====="
echo
loc=`which ngram-count`;
if [ -z $loc ]; then
        if uname -a | grep 64 >/dev/null; then
            sdir=$KALDI_ROOT/tools/srilm/bin/i686-m64
      else
            sdir=$KALDI_ROOT/tools/srilm/bin/i686
        fi
        if [ -f $sdir/ngram-count ]; then
                echo "Using SRILM language modelling tool from $sdir"
                export PATH=$PATH:$sdir
        else
                echo "SRILM toolkit is probably not installed.
                   Instructions: tools/install_srilm.sh"
                exit 1
```

```
        fi
fi
local=data/local
ngram-count -order $lm_order -write-vocab $local/tmp/vocab-full.txt -wbdiscount
-text $local/corpus.txt -lm $local/tmp/lm.arpa
echo
echo "===== MAKING G.fst ====="
echo
lang=data/lang
cat $local/tmp/lm.arpa | arpa2fst - | fstprint | utils/eps2disambig.pl | utils/s2eps.pl |
fstcompile --isymbols=$lang/words.txt --osymbols=$lang/words.txt
--keep_isymbols=false --keep_osymbols=false | fstrmepsilon | fstarcsort
--sort_type=ilabel > $lang/G.fst
echo
echo "===== MONO TRAINING ====="
echo
steps/train_mono.sh --nj $nj --cmd "$train_cmd" data/train data/lang exp/mono    ||
exit 1
echo
echo "===== MONO DECODING ====="
echo

utils/mkgraph.sh --mono data/lang exp/mono exp/mono/graph || exit 1
steps/decode.sh --config conf/decode.config --nj $nj --cmd "$decode_cmd"
exp/mono/graph data/test exp/mono/decode
echo
echo "===== MONO ALIGNMENT ====="
echo
steps/align_si.sh --nj $nj --cmd "$train_cmd" data/train data/lang exp/mono
exp/mono_ali || exit 1
echo
echo "===== TRI1 (first triphone pass) TRAINING ====="
echo
```

steps/train_deltas.sh --cmd "$train_cmd" 2000 11000 data/train data/lang exp/mono_ali exp/tri1 || exit 1

echo

echo "===== TRI1 (first triphone pass) DECODING ====="

echo

utils/mkgraph.sh data/lang exp/tri1 exp/tri1/graph || exit 1

steps/decode.sh --config conf/decode.config --nj $nj --cmd "$decode_cmd" exp/tri1/graph data/test exp/tri1/decode

echo

echo "===== run.sh script is finished ====="

echo

----- run.sh script ends here -----

9.结果

任务：现在你所要做的事情就是运行脚本 run.sh。如果我再这个任务中出现了什么错误，log 会指导你去修改错误。

另外你会发现在终端界面上有一些解码结果，然后去看一下新生成的 kaldi-trunk/egs/digits/exp。你会发现这些文件中也有 mono 和 tri 的结果，目录结构也是一样的。来到 mono/decode 目录下，你会发现以 wer_{number}命名方式的结果文件夹。解码过程的 logs 文件也会在 kaldi-trunk/egs/digits/exp 目录下的 log 文件夹下找到。

10.总结

这仅仅是一个简单的例子，这个简单的例子的意义在于想你展示了怎样去在 kaldi 上创建任何形式的工程以及在使用这些工具时以一种更好的方式去思考。就个人而言，我开始时是看的 kaldi 开发者的教程，当我成功安装上 kaldi 之后，我运行了一些脚本（包括 Yesno, Voxforge, LibriSpeech 等，它们相对来说比较简单，并且有免费的语音库去下载，我将这三个作为我自己脚本的一个基础）。

确保你仔细阅读了 kaldi 的官方网站：http://kaldi-asr.org ，上面对于刚开始接触 kaldi 的人有两个非常重要的部分：

http://kaldi.sourceforge.net/tutorial.html-关于如何搭建 ASR 系统几乎是手把手的教程；

http://kaldi.sourceforge.net/data_prep.html-非常详细的介绍了如何在 kaldi 中使用你自己的数据。

更多有用的网站：

https://sites.google.com/site/dpovey/kaldi-lectures-kaldi-主要开发者的讲座

http://www.superlectures.com/icassp2011/category.php?lang=en&id=131 -视频版

http://www.diplomovaprace.cz/133/thesis_oplatek.pdf-一些关于 kaldi 在语音识别方面的硕士毕业论文。

4 kaldi 主页上的翻译

kaldi 主页上的翻译的事情一直是我希望做的事情，希望期待更多的人参与。这里先列出部分重点内容，欢迎随时补充：

kaldi 翻译重点：

1.数据准备： http://kaldi.sourceforge.net/data_prep.html @V

2.特征提取： http://kaldi.sourceforge.net/feat.html @煮8戒

3.声学模型： http://kaldi.sourceforge.net/model.html @老那

4.解码部分： http://kaldi.sourceforge.net/graph.html @karinafl

5.FST 部分： http://kaldi.sourceforge.net/fst_algo.html @木鱼

6.解码在训练部分：http://kaldi.sourceforge.net/graph_recipe_train.html @燕子飞

7.解码在测试部分：http://kaldi.sourceforge.net/graph_recipe_test.html

8.深度学习在 kaldi 上的应用（karel 的例子）：http://kaldi.sourceforge.net/dnn1.html @wbglearn

9.关键词检索： http://kaldi.sourceforge.net/kws.html

10.在线识别： http://kaldi.sourceforge.net/online_programs.html @冒顿

11.kaldi 手册 ：http://kaldi.sourceforge.net/tutorial.html (这个需要5个人）

2014.8.16：

再此说明下，如果您想翻译 kaldi 主页中的任何一篇，欢迎跟我联系。因为您翻译的过程了不仅仅锻炼了您自己的英译汉的说明，而且还深刻理解了下您翻译的内容。开源需要每个人的努力，欢迎您的加入。

4.1 数据准备

译者：V (shiwei@sz.pku.edu.cn)　水平有限，如有错误请多包涵。@wbglearn校对。

介绍

在运行完示例脚本后（见Kaldi tutorial），你可能会想用自己的数据在Kaldi上跑一下。本节主要讲述如何准备相关数据。我们假设本页的读者使用的是最新版本的示例脚本（即在脚本目录下被命名为s5的那些，例如egs/rm/s5）。另外，除了阅读本页所述内容外，你还可以查看脚本目录下的那些数据准备相关的脚本。（译者：结合起来看更易理解。）在顶层的run.sh 脚本（例如 egs/rm/s5/run.sh）中，最前面的几行命令都是和数据准备相关的，代表数据准备的不同步骤。子目录local/下的脚本都是和数据集相关的。例如，Resource Management（RM） 数据集相应的脚本就是local/rm_data_prep.sh。对RM数据集来说，这几行数据准备的命令为：

```
local/rm_data_prep.sh /export/corpora5/LDC/LDC93S3A/rm_comp || exit 1;
utils/prepare_lang.sh data/local/dict '!SIL' data/local/lang data/lang || exit 1;
local/rm_prepare_grammar.sh || exit 1;
```

而对于 WSJ 来说，命令为：

```
wsj0=/export/corpora5/LDC/LDC93S6B
wsj1=/export/corpora5/LDC/LDC94S13B
local/wsj_data_prep.sh $wsj0/??-{?,??}.? $wsj1/??-{?,??}.?  || exit 1;
local/wsj_prepare_dict.sh || exit 1;
utils/prepare_lang.sh data/local/dict "<SPOKEN_NOISE>" data/local/lang_tmp data/lang || exit 1;

local/wsj_format_data.sh || exit 1;
```

在WSJ的示例脚本中，上述命令之后还有一些训练语言模型的命令（根据标注重新训练语言模型，而不是使用LDC提供的）， 但是上述几条命令是最重要的。

数据准备阶段的输出包含两部分。一部分与“数据”相关（保存在诸如data/train/之类的目录下），另一部分则与“语言”相关（保存在诸如data/lang/之类的目录下）。“数据”部分与数据集的录音相关，而“语言”部分则与语言本身更相关的内容，例如发音字典、音素集合以及其他Kaldi需要的关于音素的额外信息。如果你想用已有的识别系统和语言模型对你的数据进行解码，那么你只需要重写“数据”部分。

数据准备-- 数据部分.

举个数据准备阶段中的关于“数据”部分例子，请查看任何一个示例脚本目录下的“data/train”目录（假设你已经运行过一遍这些脚本了）。注意：目录名字“data/train”本身没有什么特别的。一些被命名为其他名字的目录，如“data/eval2000”（为一个测试集建立的），有几乎差不多的目录结构和文件格式（说“几乎”是因为在测试集的目录下，可能含有“sgm”和“glm”文件，用于sclite评分）。我们以Switchboard数据为例，对应脚本在egs/swbd/s5下

```
s5# ls data/train
cmvn.scp  feats.scp  reco2file_and_channel  segments  spk2utt  text
utt2spk  wav.scp
```

不是所有的文件都同等重要。如果要设置简单点，分段（segmentation）信息是不必要的（即一个文件里只有一段发音），你只需要自己创建“utt2spk”、“text”和“wav.scp”，“segments”和“reco2file_and_channel”是可选的， 根据实际需要决定是否创建。剩下的就都交给标准脚本。

下面我们会详细描述该目录下的这些文件。首先从那些需要你手动创建的文件开始。

需要手动创建的文件

文件“text”包含每段发音的标注。

```
s5# head -3 data/train/text
sw02001-A_000098-001156 HI UM YEAH I'D LIKE TO TALK ABOUT HOW
YOU DRESS FOR WORK AND
sw02001-A_001980-002131 UM-HUM
sw02001-A_002736-002893 AND IS
```

每行的第一项是发音编号（utterance-id），可以是任意的文本字符串，但是如果在你的设置中还包含说话人信息， 你应该把说话人编号（speaker-id）作为发音编号的前缀。这对于音频文件的排序非常重要。发音编号后面跟着的是每段发音的标注。你不用保证这里出现的每一个字都出现在你的词汇表中。词汇表之外的词会被映射到data/lang/oov.txt中。注意：尽管在这个特别的例子中，我们用下划线分割了发音编号中的“说话人”和“发音”部分，但是通常用破折号（“-”）会更安全一点。这是因为破折号的ASCII值更小。有人向我指出说，如果使用下划线，并且说话人编号的长度不一，在某些特殊的情况下， 如果使用标准"C"语言风格对字符串进行排序，说话人编号和对应的发音编号会被排成不同的顺序。另外一个很重要的文件是wav.scp。在Switchboard例子中，

```
s5# head -3 data/train/wav.scp
sw02001-A /home/dpovey/kaldi-trunk/tools/sph2pipe_v2.5/sph2pipe -f wav -p -c
1 /export/corpora3/LDC/LDC97S62/swb1/sw02001.sph |
sw02001-B /home/dpovey/kaldi-trunk/tools/sph2pipe_v2.5/sph2pipe -f wav -p -c
2 /export/corpora3/LDC/LDC97S62/swb1/sw02001.sph |
```

这个文件的格式是：

```
<recording-id> <extended-filename>
```

其中，“extended-filename”可能是一个实际的文件名，或者就像本例中所述那样，是一段提取wav格式文件的命令。 extended-filename末尾的管道符号表明，整个命令应该被解释为一个管道。等会我们会解释什么是“recording-id”， 但是首先，我们需要指出，如果“segments”文件不存在，“wav.scp”每一行的第一项就是发音编号。

在Switchboard设置中，我们有“segments”文件，所以下面我们就讨论一下这个文件。

```
s5# head -3 data/train/segments
sw02001-A_000098-001156 sw02001-A 0.98 11.56
sw02001-A_001980-002131 sw02001-A 19.8 21.31
sw02001-A_002736-002893 sw02001-A 27.36 28.93
```

"segments"文件的格式是：

```
<utterance-id> <recording-id> <segment-begin> <segment-end>
```

其中，segment-begin和segment-end以秒为单位。它们指明了一段发音在一段录音中的时间偏移量。“recording-id” 和在“wav.scp”中使用的是同一个标识字符串。再次声明一下，这只是一个任意的标识字符串，你可以随便指定。文件"reco2file_and_channel"只是在你用NIST的sclite工具对结果进行评分（计算错误率）的时候使用：

```
s5# head -3 data/train/reco2file_and_channel
sw02001-A sw02001 A
sw02001-B sw02001 B
sw02005-A sw02005 A
```

格式是：

```
<recording-id> <filename> <recording-side (A or B)>
```

filename通常是.sph文件的名字，当然需要去掉sph这个后缀；但是也可以是任何其他你在“stm”文件中使用的标识字符串。 “录音方”（recording side）则是一个电话对话中两方通话（A或者B）的概念。如果不是两方通话，那么为保险起见最好使用“A”。如果你并没有“stm”文件，或者你根本不知道这些都是什么东西，那么你可能就不需要reco2file_and_channel"文件。

最后一个需要你手动创建的文件是“utt2spk”。该文件指明某一段发音是哪一个说话人发出的。

```
s5# head -3 data/train/utt2spk
sw02001-A_000098-001156 2001-A
sw02001-A_001980-002131 2001-A
sw02001-A_002736-002893 2001-A
```

格式是：

```
<utterance-id> <speaker-id>
```

注意一点，说话人编号并不需要与说话人实际的名字完全一致——只需要大概能够猜出来就行。在这种情况下，我们假定每一个说话方（电话对话的每一方）对应一个说话人。这可能不完全正确—— 有时一个说话人会把电话交给另外一个说话人，或者同一个说话人会在不同的对话中出现——但是上述假定对我们来

说也足够用了。如果你完全没有关于说话人的信息，你可以把发音编号当做说话人编号。那么对应的文件格式就变为<utterance-id> <utterance-id>。

在一些示例脚本中还出现了另外一个文件，它在Kaldi的识别系统的建立过程中只是偶尔使用。在Resource Management

（RM）设置中该文件是这样的：

```
s5# head -3 ../../rm/s5/data/train/spk2gender
adg0 f
ahh0 m
ajp0 m
```

这个文件根据说话人的性别，将每个说话人编号映射为“m”或者“f”。

上述所有文件都应该被排序。如果没有排序，你在运行脚本的时候就会出现错误。在 The Table concept 中我们解释了为什么需要这样。这与（Kaldi的）I/O框架有关，归根到底是因为排序后的文件可以在一些不支持 fseek()的流中，例如，含有管道的命令，提供类似于随机存取查找的功能。许多Kaldi程序都会从其他Kaldi命令中读取多个管道流，读入各种不同类型的对象，然后对不同输入做一些类似于“合并然后排序”的处理。既然要合并排序， 当然需要输入是经过排序的。小心确保你的shell环境变量LC_ALL定义为“C”。例如，在bash中，你需要这样做：

```
export LC_ALL=C
```

如果你不这样做，这些文件的排序方式会与C++排序字符串的方式不一样，Kaldi就会崩溃。这一点我已经再三强调过了！

如果你的数据中包含NIST提供的测试集，其中有“stm”和“glm”文件可以用作计算WER，那么你可以直接把这些文件拷贝到数据目录下，并分别命名为“stm”和“glm”。注意，我们把评分脚本score.sh（可以计算WER）放到local/下， 这意味着该脚本是与数据集相关的。不是所有的示例设置下的评分脚本都能识别stm和glm文件。能够使用这些文件的一个例子在Switchboard设置里，即i.e.egs/swbd/s5/local/score_sclite.sh。如果检测到你有stm和glm文件该脚本会被顶层的评分脚本egs/swbd/s5/local/score.sh调用。

不需要手动创建的文件

数据目录下的其他文件可以由前述你提供的文件所生成。你可以用如下的一条命令创建“spk2utt”文件（ 这是一条从egs/rm/s5/local/rm_data_prep.sh中摘取的命令）：

```
utils/utt2spk_to_spk2utt.pl data/train/utt2spk > data/train/spk2utt
```

这是因为 utt2spk 和 spk2utt 文件中包含的信息是一样的。spk2utt 文件的格式是：

<speaker-id> <utterance-id1> <utterance-id2> ...

下面我们讲一讲 feats.scp 文件.

```
s5# head -3 data/train/feats.scp
sw02001-A_000098-001156
/home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/raw_mfcc_train.1.ark:24
```

```
sw02001-A_001980-002131 /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/raw_mfcc_train.1.ark:54975
sw02001-A_002736-002893 /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/raw_mfcc_train.1.ark:62762
```

这个文件指向已经提取好的特征——在这个例子中我们所使用的是 MFCC。feats.scp 文件的格式是：

```
<utterance-id> <extended-filename-of-features>
```

每一个特征文件保存的都是 Kaldi 格式的矩阵。在这个例子中，矩阵的维度是 13（译者注：即列数；行数则和你的文件长度有关，标准情况下帧长 20ms，帧移 10ms，所以一行特征数据对应 10ms 的音频数据。但在 Kaldi 中，实际返回的帧数可能比你预想的要小一点。例如，你的音频是 5.68s，返回的通常是 565 帧而不是 568。这和 Kaldi 中处理不足以分帧的剩余数据的方式有关，目前这种方式是兼容 HTK 的。你需要一直向下传递 window size 和 frame shift 的值，才能准确算出每一帧的 timing。这种方式在实际使用中有不少问题。）。第一行的“extended filename”，

即/home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/raw_mfcc_train.1.ark:24，意思是，打开存档（archive）文件

/home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/raw_mfcc_train.1.ark， fseek()定位到 24（字节），然后开始读数据。

feats.scp 文件由如下命令创建：

```
steps/make_mfcc.sh --nj 20 --cmd "$train_cmd" data/train exp/make_mfcc/train $mfccdir
```

该句被顶层的“run.sh”脚本调用。命令中一些 shell 变量的定义，请查阅对应 run.sh。

$mfccdir 是.ark 文件将被写入的目录，由用户自定义。

data/train 下最后一个未讲到的文件是“cmvn.scp”。该文件包含了倒谱均值和方差归一化的统计量，以说话人编号为索引。每个统计量集合都是一个矩阵，在本例中是 2 乘以 14 维。在我们的例子中，有：

```
s5# head -3 data/train/cmvn.scp
2001-A /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/cmvn_train.ark:7
2001-B /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/cmvn_train.ark:253
2005-A /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/cmvn_train.ark:499
```

与 feats.scp 不同，这个 scp 文件是以说话人编号为索引，而不是发音编号。该文件由如下的命令创建：

```
steps/compute_cmvn_stats.sh data/train exp/make_mfcc/train $mfccdir
```

(这个例句来自 egs/swbd/s5/run.sh).

因为数据准备阶段的错误会影响后续脚本的运行，所以我们有一个脚本来判断数据目录的文件格式是否正确。运行如下的例子：

```
utils/validate_data_dir.sh data/train
```

你可能会发现下面这个命令也很有用：

```
utils/fix_data_dir.sh data/train
```

（当然可对任何数据目录使用该命令，而不只是 data/train）。该脚本会修复排序错误，并会移除那些被指明需要特征数据或标注，但是却找不到被需要的数据的那些发音（utterances）。

数据准备-- “lang”目录

现在我们关注一下数据准备的“lang”这个目录。

```
s5# ls data/lang
L.fst  L_disambig.fst  oov.int  oov.txt  phones  phones.txt  topo  words.txt
```

除 data/lang，可能还有其他目录拥有相似的文件格式：例如有个目录被命名为“data/lang_test”，其中包含和 data/lang 完全一样的信息，但是要多一个 G.fst 文件。该文件是一个 FST 形式的语言模型：

```
s5# ls data/lang_test
G.fst  L.fst  L_disambig.fst  oov.int  oov.txt  phones  phones.txt  topo  words.txt
```

注意，lang_test/ 由拷贝 lang/目录而来，并加入了 G.fst。每个这样的目录都似乎只包含为数不多的几个文件。但事实上不止如此，因为其中 phones 是一个目录而不是文件：

```
s5# ls data/lang/phones
context_indep.csl  disambig.txt          nonsilence.txt          roots.txt  silence.txt
context_indep.int  extra_questions.int   optional_silence.csl    sets.int   word_boundary.int
context_indep.txt  extra_questions.txt   optional_silence.int    sets.txt   word_boundary.txt
disambig.csl       nonsilence.csl        optional_silence.txt    silence.csl
```

phones 目录下有许多关于音素集的信息。同一类信息可能有三种不同的格式，分别以.csl、.int 和.txt 结尾。幸运的是，作为一个 Kaldi 用户，你没有必要去一一手动创建所有这些文件，因为我们有一个脚本“utils/prepare_lang.sh”能够根据更简单的输入为你创建所有这些文件。在讲述该脚本和所谓更简单的输入之前，有必要先解释一下“lang”目录下到底有些什么内容。之后我们将解释如何轻松创建该目录。如果用户不需要理解 Kaldi 是如何工作的，而是秉着快速建立识别系统的目的，那么可以跳过下面的 Creating the "lang" directory 这一节

“lang”目录下的内容

首先是有文件 hones.txt 和 words.txt。这些都是符号表（symbol-table）文件，符合 OpenFst 的格式定义。其中每一行首先是一个文本项，接着是一个数字项：

```
s5# head -3 data/lang/phones.txt
<eps> 0
SIL 1
SIL_B 2
```

```
s5# head -3 data/lang/words.txt
<eps> 0
!SIL 1
-'S 2
```

在 Kaldi 中，这些文件被用于在这些音素符号的文本形式和数字形式之间进行转换。

大多数情况下，只有脚本 utils/int2sym.pl、utils/sym2int.pl 和 OpenFst 中的程序

fstcompile 和 fstprint 会读取这些文件。

文件 L.fst 是 FST 形式的发音字典（L，见 <http://www.cs.nyu.edu/~mohri/pub/hbka.pdf> "Speech Recognition with Weighted Finite-State Transducers" by Mohri, Pereira and Riley, in Springer Handbook on SpeechProcessing and Speech Communication, 2008），

其中，输入是音素，输出是词。文件 L_disambig.fst 也是发音字典，但是还包含了为消歧而引入的符号，诸如#1、#2 之类，以及为自环（self-loop）而引入的 #0 。#0 能让消岐符号“通过”（pass through）整个语法（译者注：n 元语法，即我们的语言模型。另外前面这句话我是在不知道该怎么翻译）。更多解释见 Disambiguation symbols 。但是不管明白与否，你其实不用自己手动去引入这些符号。

文件 data/lang/oov.txt 仅仅只有一行：

```
s5# cat data/lang/oov.txt
<UNK>
```

在训练过程中，所有词汇表以外的词都会被映射为这个词（译者注：UNK 即 unknown）。“<UNK>”本身并没有特殊的地方，也不一定非要用这个词。重要的是需要保证这个词的发音只包含一个被指定为“垃圾音素”（garbage phone）的音素。该音素会与各种口语噪声对齐。在我们的这个特别设置中，该音素被称为 <SPN>，就是“spoken noise”的缩写：

```
s5# grep -w UNK data/local/dict/lexicon.txt
<UNK> SPN
```

文件 oov.int 包含<UNK>的整数形式（从 words.txt 中提取的），在本设置中是 221。你可能已经注意到了，在 Resource Management 设置中，oov.txt 里有一个静音词，在 RM 设置中被称为“!SIL”。在这种情况下，我们从词汇表中任意选一个词（放入 oov.txt）——因为训练集中没有 oov 词，所以选哪个都不起作用。

文件 data/lang/topo 则含有如下数据：

```
s5# cat data/lang/topo
<Topology>
<TopologyEntry>
<ForPhones>
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73
```

```
74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119
120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138
139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157
158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176
177 178 179 180 181 182 183 184 185 186 187 188
    </ForPhones>
    <State> 0 <PdfClass> 0 <Transition> 0 0.75 <Transition> 1 0.25 </State>
    <State> 1 <PdfClass> 1 <Transition> 1 0.75 <Transition> 2 0.25 </State>
    <State> 2 <PdfClass> 2 <Transition> 2 0.75 <Transition> 3 0.25 </State>
    <State> 3 </State>
    </TopologyEntry>
    <TopologyEntry>
    <ForPhones>
    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
    </ForPhones>
    <State> 0 <PdfClass> 0 <Transition> 0 0.25 <Transition> 1 0.25 <Transition> 2
0.25 <Transition> 3 0.25 </State>
    <State> 1 <PdfClass> 1 <Transition> 1 0.25 <Transition> 2 0.25 <Transition> 3
0.25 <Transition> 4 0.25 </State>
    <State> 2 <PdfClass> 2 <Transition> 1 0.25 <Transition> 2 0.25 <Transition> 3
0.25 <Transition> 4 0.25 </State>
    <State> 3 <PdfClass> 3 <Transition> 1 0.25 <Transition> 2 0.25 <Transition> 3
0.25 <Transition> 4 0.25 </State>
    <State> 4 <PdfClass> 4 <Transition> 4 0.75 <Transition> 5 0.25 </State>
    <State> 5 </State>
    </TopologyEntry>
    </Topology>
```

这个文件指明了我们所用 HMM 模型的拓扑结构。在这个例子中，一个“真正”的音素内含 3 个发射状态，呈标准的三状态从左到右拓扑结构——即“Bakis”模型。（发射状态即能“发射”特征矢量的状态，与之对应的就是那些“假”的仅用于连接其他状态的非发射状态）。音素 1 到 20 是各种静音和噪声。之所以会有这么多，是因为对词中的不同位置的同一音素进行了区分（word-position-dependency）。这种情况下，实际上这里的静音和噪声音素大多数都用不上。不考虑在词位话，应该只有 5 个代表静音和噪声的音素。所谓静音音素（silence phones）有更复杂的拓扑结构。每个静音音素都有一个起始发射状态和一个结束发射状态，中间还有另外三个发射状态。你不用手动创建 data/lang/topo。

data/lang/phones/下有一系列的文件，指明了音素集合的各种信息。这些文件大多数有三个不同版本：一个 “.txt”形式，如：

```
    s5# head -3 data/lang/phones/context_indep.txt
    SIL
    SIL_B
    SIL_E
```

一个“.int”形式，如:

```
s5# head -3 data/lang/phones/context_indep.int
1
2
3
```

以及一个“.csl”形式，内含一个冒号分割的列表：

```
s5# cat data/lang/phones/context_indep.csl
1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20
```

三种形式的文件包含的是相同的信息，所以我们只关注人们更易阅读的“.txt”形式。文件"context_indep.txt"包含一个音素列表，用于建立文本无关的模型。也就是说，对这些音素，我们不会建立需要参考左右音素的上下文决策树。实际上，我们建立的是更小的决策树，只参考中心音素和 HMM 状态。这依赖于“roots.txt”，下面将会介绍到。关于决策树的更深入讨论见 How decision trees are used in Kaldi 。

文件 context_indep.txt 包含所有音素，包括那些所谓“假”音素：例如静音（SIL），口语噪声（SPN），非口语噪声（NSN）和笑声（LAU）：

```
# cat data/lang/phones/context_indep.txt
SIL
SIL_B
SIL_E
SIL_I
SIL_S
SPN
SPN_B
SPN_E
SPN_I
SPN_S
NSN
NSN_B
NSN_E
NSN_I
NSN_S
LAU
LAU_B
LAU_E
LAU_I
LAU_S
```

因为考虑了词位，这些音素都有许多变体。不是所有的变体都会被实际使用。这里， SIL 代表静音词，会被插入到发音字典中（是一个词而不是一个词的一部分，可选的）；SIL_B 则代表一个静音音素，应该出现在一个词的开端（这种情况应该永不出现）；SIL_I 代表词内静音（也很少存在）；SIL_E 代表词末静音（不应该存在）；而 SIL_S 则表示一种被视为“单独词”（singleton word）的静

音，意指这个音素只对应一个词——当你的发音字典中有“静音词”且标注中有明确的静音段时会有用。

silence.txt 和 onsilence.txt 分别包含静音音素列表和非静音音素列表。这两个集合是互斥的，且如果合并在一起，应该是音素的总集。在本例中，silence.txt 与 context_indep.txt 的内容完全一致。我们说“非静音”音素，是指我们将要估计各种线性变换的音素。所谓线性变换是指全局变换，如 LDA 和 MLLT，以及说话人自适应变换，如 fMLLR。 根据之前的实验，我们相信，加入静音对这些变换没有影响。我们的经验是，把噪声和发声噪声都列为“静音”音素，而其他传统的音素则是“非静音”音素。在 Kaldi 中我们还没有通过实验找到一个最佳的方法来这样做。

```
s5# head -3 data/lang/phones/silence.txt
SIL
SIL_B
SIL_E
s5# head -3 data/lang/phones/nonsilence.txt
IY_B
IY_E
IY_I
```

disambig.txt 包含一个“消岐符号”列表（见 Disambiguation symbols):

```
s5# head -3 data/lang/phones/disambig.txt
#0
#1
#2
```

这些符号会出现在 phones.txt 中，被当做音素使用。

optional_silence.txt 只含有一个音素。该音素可在需要的时候出现在词之间：

```
s5# cat data/lang/phones/optional_silence.txt
SIL
```

（可选静音列表中的）音素出现在词之间的机制是，在发音字典的 FST 中，可选地让该音素 出现在每个词的词尾（以及每段发音的段首）。该音素必须在 phones/中指明而不是仅仅出现在 L.fst 中。这个原因比较复杂，这里就不讲了。

文件 sets.txt 包含一系列的音素集，在聚类音素时被分组（被当做同一个音素），以便建立文本相关问题集（在 Kaldi 中，建立决策树时使用自动生成的问题集，而不是具有语言语义的问题集）。本设置中，sets.txt 将每个音素的所有不同词位的变体组合为一行：

```
s5# head -3 data/lang/phones/sets.txt
SIL SIL_B SIL_E SIL_I SIL_S
```

```
SPN SPN_B SPN_E SPN_I SPN_S
NSN NSN_B NSN_E NSN_I NSN_S
```

文件 extra_questions.txt 包含那些自动产生的问题集之外的一些问题：

```
s5# cat data/lang/phones/extra_questions.txt
IY_B B_B D_B F_B G_B K_B SH_B L_B M_B N_B OW_B AA_B TH_B P_B OY_B R_B UH_B AE_B S_B T_B AH_B V_B W_B Y_B Z_B CH_B AO_B DH_B UW_B ZH_B EH_B AW_B AX_B EL_B AY_B EN_B HH_B ER_B IH_B JH_B EY_B NG_B
IY_E B_E D_E F_E G_E K_E SH_E L_E M_E N_E OW_E AA_E TH_E P_E OY_E R_E UH_E AE_E S_E T_E AH_E V_E W_E Y_E Z_E CH_E AO_E DH_E UW_E ZH_E EH_E AW_E AX_E EL_E AY_E EN_E HH_E ER_E IH_E JH_E EY_E NG_E
IY_I B_I D_I F_I G_I K_I SH_I L_I M_I N_I OW_I AA_I TH_I P_I OY_I R_I UH_I AE_I S_I T_I AH_I V_I W_I Y_I Z_I CH_I AO_I DH_I UW_I ZH_I EH_I AW_I AX_I EL_I AY_I EN_I HH_I ER_I IH_I JH_I EY_I NG_I
IY_S B_S D_S F_S G_S K_S SH_S L_S M_S N_S OW_S AA_S TH_S P_S OY_S R_S UH_S AE_S S_S T_S AH_S V_S W_S Y_S Z_S CH_S AO_S DH_S UW_S ZH_S EH_S AW_S AX_S EL_S AY_S EN_S HH_S ER_S IH_S JH_S EY_S NG_S
SIL SPN NSN LAU
SIL_B SPN_B NSN_B LAU_B
SIL_E SPN_E NSN_E LAU_E
SIL_I SPN_I NSN_I LAU_I
SIL_S SPN_S NSN_S LAU_S
```

你可以看到，所谓一个问题就是一组音素。前 4 个问题是关于普通音素的词位信息，后面五个则是关于“静音音素”的。 “静音”音素也可能没有像_B 这样的后缀，比如 SIL。这些可被作为发音字典中可选的静音词的表示，即不会出现在某个词中，而是单独成词。在具有语调和语气的设置中，extra_questions.txt 可以包含与之相关的问题集。

word_boundary.txt 解释了这些音素与词位的关联情况：

```
s5# head    data/lang/phones/word_boundary.txt
SIL nonword
SIL_B begin
SIL_E end
SIL_I internal
SIL_S singleton
SPN nonword
SPN_B begin
```

这和音素中的后缀（_B 等等）是相同的信息，但我们并不想给音素的文本形式附加这样的强制限制——记住一件事，Kaldi 的可执行程序从不使用音素的文本形式，而是整数形式。（译者注：即 Kaldi 内部使用的都是音素的整数标号来表示和传递音素）。所以我们使用文件 word_boundary.txt 来指明各音素与词位间的对应关系。 建立这种对应关系的原因是因为我们需要这些信息从音素网格

中恢复词的边界（例如，lattice-align-words 需要读取 word_boundary.txt 的整数版本 word_boundary.int）。找出词的边界是有用的，其中之一是用作 NIST 的 sclite 评分，该工具需要词的时间标记。还有其他的后续处理需要这些信息。

roots.txt 文件包含如何建立音素上下文决策树的信息：

```
head data/lang/phones/roots.txt
shared split SIL SIL_B SIL_E SIL_I SIL_S
shared split SPN SPN_B SPN_E SPN_I SPN_S
shared split NSN NSN_B NSN_E NSN_I NSN_S
shared split LAU LAU_B LAU_E LAU_I LAU_S
...
shared split B_B B_E B_I B_S
```

暂时你可以忽略”shared“和”split“——这些与我们建立决策树时的具体选项有关(更多信息见 How decision trees are used in Kaldi). 像 SIL SIL_B SIL_E SIL_I SIL_S 这样，几个音素出现在同一行的意义是，在决策树中它们都有同一个“共享根”（shared root），因此状态可在这些音素间共享。对于带语气语调的系统，通常所有与语气和语调相关的音素变体都会出现在同一行。此外，一个 HMM 中的 3 个状态（对静音来说有 5 个状态）共享一个根，且决策树的建立过程需要知道状态（的共享情况）。 HMM 状态间共享决策树根节点，这就是 roots 文件中“shared”代表的意思。

建立"lang"目录

data/lang/目录下有很多不同的文件，所以我们提供了一个脚本为你创建这个目录，你只需要提供一些相对简单的输入信息：

```
utils/prepare_lang.sh data/local/dict "<UNK>" data/local/lang data/lang
```

这里，输入目录是 data/local/dict/，<UNK>需要在字典中，是标注中所有 OOV 词的映射词（映射情况会写入 data/lang/oov.txt 中）。data/local/lang/只是脚本使用的一个临时目录，data/lang/才是输出文件将会写入的地方。

作为数据准备者，你需要做的事就是创建 data/local/dict/这个目录。该目录包含以下信息：

```
s5# ls data/local/dict
extra_questions.txt    lexicon.txt nonsilence_phones.txt    optional_silence.txt
silence_phones.txt
```

（实际上还有一些文件我们没有列出来，但那都是在创建目录时所遗留下的临时文件，可以忽略）。下面的这些命令可以让你知道这些文件中大概都有些什么：

```
s5# head -3 data/local/dict/nonsilence_phones.txt
IY
B
D
s5# cat data/local/dict/silence_phones.txt
SIL
SPN
NSN
```

```
LAU
s5# cat data/local/dict/extra_questions.txt
s5# head -5 data/local/dict/lexicon.txt
!SIL SIL
-'S S
-'S Z
-'T K UH D EN T
-1K W AH N K EY
```

正如你看到的，本设置（Switchboard）中，这个目录下的内容都非常简单。我们只是分别列出了“真正”的音素和“静音”音素，一个叫 extra_questions.txt 的空文件，以及一个有如下格式的 lexicon.txt：

```
<word> <phone1> <phone2> ...
```

注意：lexicon.txt 中，如果一个词有不同发音，则会在不同行中出现多次。如果你想使用发音概率，你需要建立 exiconp.txt 而不是 lexicon.txt。lexiconp.txt 中第二域就是概率值。

注意，一个通常的作法是，对发音概率进行归一化，使最大的那个概率值为 1，而不是使同一个词的所有发音概率加起来等于 1。 这样可能会得到更好的结果。 如果想在顶层脚本中找一个与发音概率相关的脚本，请在 egs/wsj/s5/run.sh 目录下搜索 pp。

需要注意的是，在这些输入中，没有词位信息，即没有像_B 和_E<这样的后缀。

这是因为脚本 prepare_lang.sh 会添加这些后缀。

从空的 extra_questions.txt 件中你会发现，可能还有些潜在的功能我们没有利用。

这其中就包括重音和语调标记。 对具有不同重音和语调的同一音素，你可能会想用不同的标记去表示。为展示如何这样做，我们看看在另外一个设置 egs/wsj/s5/的这些文件。结果如下：

```
s5# cat data/local/dict/silence_phones.txt
SIL
SPN
NSN
s5# head data/local/dict/nonsilence_phones.txt
S
UW UW0 UW1 UW2
T
N
K
Y
Z
AO AO0 AO1 AO2
AY AY0 AY1 AY2
```

```
SH
s5# head -6 data/local/dict/lexicon.txt
!SIL SIL
<SPOKEN_NOISE> SPN
<UNK> SPN
<NOISE> NSN
!EXCLAMATION-POINT  EH2 K S K L AH0 M EY1 SH AH0 N P OY2 N T
"CLOSE-QUOTE  K L OW1 Z K W OW1 T
s5# cat data/local/dict/extra_questions.txt
SIL SPN NSN
S UW T N K Y Z AO AY SH W NG EY B CH OY JH D ZH G UH F V ER AA IH M DH L AH P OW AW HH AE R TH IY EH
UW1 AO1 AY1 EY1 OY1 UH1 ER1 AA1 IH1 AH1 OW1 AW1 AE1 IY1 EH1
UW0 AO0 AY0 EY0 OY0 UH0 ER0 AA0 IH0 AH0 OW0 AW0 AE0 IY0 EH0
UW2 AO2 AY2 EY2 OY2 UH2 ER2 AA2 IH2 AH2 OW2 AW2 AE2 IY2 EH2
s5#
```

你可能已经注意到了，nonsilence_phones.txt 中的某些行，一行中有多个音素。这些是同一元音的与重音相关的不同表示。 注意，在 CMU 版的字典中，每个音素有 4 种表示：例如，UW UW0 UW1 UW2。基于某些原因。其中一种表示没有数字后缀。行中音素的顺序没有关系。通常，我们建议将每个“真实音素”的不同形式都组织在单独的一行中。我们使用 CMU 字典中的重音标记。文件 extra_questions.txt 中只有一个问题包含所有的“静音”音素（实际上这是不必要的，只是脚本 prepare_lang.sh 会添加这么一个问题），以及一个涉及不同重音标记的问题。这些问题对利用重音标记信息来说是必要的，因为在 nonsilence_phones.txt 中每个音素的不同重音表示都在同一行，这确保了他们在 data/lang/phones/roots.txt 和 data/lang/phones/sets.txt 也属同一行，这又反过来确保了它们共享同一个（决策）树根，并且不会有决策问题弄混它们。因此，我们需要提供一个特别的问题，能为决策树的建立过程提供一种区分音素的方法。 注意：我们在 sets.txt 和 roots.txt 中将音素分组放在一起的原因是，这些同一音素的不同重音变体可能缺乏足够的数据去稳健地估计一个单独的决策树，或者是产生问题集时需要的聚类信息。 像这样把它们组合在一起，我们可以确保当数据不足以对它们分别估计决策树时，这些变体能在决策树的建立过程中“聚集在一起”（stay together）。

写到这里我们需要提一点，脚本 utils/prepare_lang.sh 支持很多选项。下面是该脚本的用法，可让你们了解这些选项都有哪些：

```
usage: utils/prepare_lang.sh <dict-src-dir> <oov-dict-entry> <tmp-dir> <lang-dir>
e.g.: utils/prepare_lang.sh data/local/dict <SPOKEN_NOISE> data/local/lang data/lang
options:
```

```
        --num-sil-states <number of states>             # default: 5, #states in
silence models.
        --num-nonsil-states <number of states>          # default: 3, #states in
non-silence models.
        --position-dependent-phones (true|false)        # default: true; if true, use
_B, _E, _S & _I
                                                        # markers on
phones to indicate word-internal positions.
        --share-silence-phones (true|false)             # default: false; if true,
share pdfs of
                                                        # all
non-silence phones.
        --sil-prob <probability of silence>             # default: 0.5 [must have
0 < silprob < 1]
```

一个可能的重要选项是--share-silence-phones。该选项默认是 false。 如果该选项被设为 true， 所有静音音素——如静音、发声噪声、噪声和笑声——的概率密度函数（PDF，高斯混合模型）都会共享，只有模型中的转移概率不同。现在还不清楚为什么这样做有用，但我们发现这对 IARPA 的 BABEL 项目中的广东话数据集非常有效。该数据集非常乱，其中有很长的未标注的部分，我们试着将其与一个特别标记的音素对齐。我们怀疑训练数据可能没能成功正确对齐，而且基于某些不明原因，将上述选项设置为 true 则改变了结果。

另外一个可能的重要选项是“--sil-prob”。 通常，对这些选项我们所作的实验都不多，所以对具体如何设置也不能给出非常详细的建议。

创建语言模型或者语法文件

前面的关于如何创建 lang/目录的教程没有涉及如何产生 G.fst 文件。该文件是语言模型——或者称为语法——的有限状态转换器格式的表示，我们解码时需要它。 实际上，在一些设置中，为做不同的测试，我们可能会有许多“lang”目录。这些目录中有不同的语言模型和字典。以华尔街日报（WSJ）的设置为例：

```
s5# echo data/lang*
data/lang data/lang_test_bd_fg data/lang_test_bd_tg data/lang_test_bd_tgpr
data/lang_test_bg \
 data/lang_test_bg_5k data/lang_test_tg data/lang_test_tg_5k data/lang_test_tgpr
data/lang_test_tgpr_5k
```

根据我们使用的语言模型的不同——是统计语言模型还是别的种类的语法形式——生成 G.fst 的步骤会不同。 在 RM 设置中，使用的是二元语法，只允许某些词对。我们将总概率值 1 分配给所有向外的弧，以确保每个语法状态的概率和为 1。在 local/rm_data_prep.sh 中有这样一句代码：

```
local/make_rm_lm.pl $RMROOT/rm1_audio1/rm1/doc/wp_gram.txt  >
$tmpdir/G.txt || exit 1;
```

脚本 local/make_rm_lm.pl 会建立一个 FST 格式的语法文件（文本格式，不是二进制格式）。

该文件包含如下形式的行：

```
s5# head data/local/tmp/G.txt
0	1	ADD	ADD	5.19849703126583
0	2	AJAX+S	AJAX+S	5.19849703126583
0	3	APALACHICOLA+S	APALACHICOLA+S
5.19849703126583
```

到 www.openfst.org 上查阅更多关于 OpenFst 的信息（他们有一个很详细的教程）。 脚本 local/rm_prepare_grammar.sh 会将文本格式的语法文件转换为二进制文件 G.fst。所用命令如下：

```
fstcompile --isymbols=data/lang/words.txt --osymbols=data/lang/words.txt
--keep_isymbols=false \
        --keep_osymbols=false $tmpdir/G.txt > data/lang/G.fst
```

如果你要建立自己的语法文件，你也应做类似的事。

注意：这种过程只适用于一类语法：用上述方法你不能创建上下文无关的语法，因为这类语法不能被表示为 OpenFst 格式。 在 WFST 框架下还是有办法这么做（见 Mike Riley 最近关于 push down transducers 的研究工作），但是在 Kaldi 中我们还没实现这些功能。

在 WSJ 设置中，我们使用了一个统计语言模型。脚本 local/wsj_format_data.sh 将 WSJ 数据库提供的 ARPA 格式的语言模型文件转换为 OpenFst 格式的。 脚本中关键的命令如下：

```
    gunzip -c $lmdir/lm_${lm_suffix}.arpa.gz | \
      utils/find_arpa_oovs.pl $test/words.txt  > $tmpdir/oovs_${lm_suffix}.txt
   ...
    gunzip -c $lmdir/lm_${lm_suffix}.arpa.gz | \
       grep -v '<s> <s>' | \
       grep -v '</s> <s>' | \
       grep -v '</s> </s>' | \
       arpa2fst - | fstprint | \
       utils/remove_oovs.pl $tmpdir/oovs_${lm_suffix}.txt | \
       utils/eps2disambig.pl | utils/s2eps.pl | fstcompile --isymbols=$test/words.txt \
          --osymbols=$test/words.txt  --keep_isymbols=false
--keep_osymbols=false | \
         fstrmepsilon > $test/G.fst
```

这里，变量$test 的值形如 data/lang_test_tg。最重要的一条命令是 arpa2fst， 这是一个 Kaldi 程序。该程序将 ARPA 格式的语言模型转换为一个加权有限状态转换器（实际上是一个接收器）。grep 命令移除语言模型中“不可用”的 N 元语法。程序 remove_oovs.pl

移除包含集外词的 N 元语法（如果不移除会引起 fstcompile 崩溃）。

eps2disambig.pl

将回退弧上的<eps>（ε）符号转换为一个特殊的符号#0，以保证语法文件是确定的（determinizable），见 Disambiguation symbols. 如果你不知道“回退弧”是什么，

你可以参考关于回退 N 元语法的文献，例如 Goodman 的“A bit of progress in language modeling”，以及我们前面引用的 Mohri 的论文。 命令 s2eps.pl 将句首和句末

符号<s> 和 </s> 转换为 epsilon（<eps>）， 意即“没有符号”。fstcompile 是一个 OpenFst 命令，可将文本形式的 FST 转换为二进制形式的。 fstrmepsilon 也是一个 OpenFst 命令，可将 FST 中由<s> 和 </s> 替换而来的少量的<eps>符号移除掉。

4.2 特征提取

本翻译原文 http://kaldi.sourceforge.net/feat.html，由@煮八戒翻译，@wbglearn 校对和修改。

简介

我们做特征提取和波形读取的这部分代码，其目的是为了得到标准的 MFCC(译注：梅尔倒谱系数)和 PLP(译注：感知线性预测系数)特征，设置合理的默认值但留了一部分用户最有可能想调整的选项（如梅尔滤波器的个数，最小和

最大截止频率等等）。这部分代码只读取 wav 文件里的 pcm(译注：脉冲编码调制)数据。这类文件通常带.wav 或.pcm 后缀（虽然有时.pcm 后缀会用于 sph 文件；这种情况下必须转换该文件）。假如源数据不是 wav 类文件，则用户可自由选择命令行工具来转换，而我们提供的 sph2pipe 工具已能满足一般的情况。

命令行工具 compute-mfcc-feats 和 compute-plp-feats 计算特征；同其它 Kaldi 工具一样，不带参数地运行它们会给出一个选项列表。例子脚本里显示了这些工具的用法。

计算 MFCC 特征

这里我们介绍如何使用命令行工具 compute-mfcc-feats 计算 MFCC 参数。该程序需要两个命令行参数：rspecifier 是用来读.wav 数据（以发音为索引）和 wspecifier 是用来写特征（以发音为索引）；参见 The Table concept 和 Specifying Table formats: wspecifiers and rspecifiers 获取更多关于这些术语的解释。典型的用法是，将数据写入一个大的“archive”文件，也写到一个“scp”文件以便随机存取；参见 Writing an archive and a script file simultaneously 解释。程序没有添加增量功能（如需添加，参见 add-deltas）。它接收选项-channel 来选择通道（如–channel=0, –channel=1），该选项在读取立体声数据时很有用。

计算 MFCC 特征由 Mfcc 类型的对象完成，它有 Compute()函数可以根据波形计算特征。

完整的 MFCC 计算如下所示：

- 计算出一个文件中帧的数目（通常帧长 25ms 帧移 10ms）。
- 对每一帧：
  - 提取数据，可选做 dithering（注：直译为抖动，在这里可以理解为类似归一化的预处理），预加重和去除直流偏移，还可以和加窗函数相乘（此处支持多种选项，如汉明窗）
  - 计算该点能量（假如用对数能量则没有 C0）
  - 做 FFT（译注：快速傅里叶变换）并计算功率谱
  - 计算每个梅尔滤波器的能量；如 23 个部分重叠的三角滤波器，其中心在梅尔频域等间距
  - 计算对数能量并作余弦变换，根据要求保留系数（如 13 个）
  - 选做倒谱变换；它仅仅是比例变换，确保系数在合理范围。

上下截止频率根据三角滤波器界定，由选项–low-freq 和–high-freq 控制，通常分别设置为 0Hz 和奈奎斯特频率附近，如对 16kHz 采样的语音设置为–low-freq=20 和 –high-freq=7800。

Kaldi 的特征和 HTK 的特征在很多方面不同，但是几乎所有这些不同归结于有不同的默认值。用选项–htk-compat=true 并正确设置参数，能得到同 HTK 非常接近的特征。一个可能重要的选项是我们不支持能量最大归一化。这是因为我们希望能把无状态方式应用到归一化方法，且希望从原理上计算一帧帧特征仍能给出相同结果。但是程序 compute-mfcc-feats 里有–subtract-mean 选项来提取特征的均值。对每个语音做此操作；每个说话人可以有不同的方式来提取特征均值。（如在脚本里搜“cmvn”，表示倒谱均值和方差归一化）。

计算 PLP 特征

计算 PLP 特征的算法与 MFCC 的算法前期是一样的。稍后我们也许会在此部分增加些内容，但目前参见 Hynek Hermansky《语音的感知线性预测（PLP）分析》，Journal of the Acoustical Society of America, vol. 87, no. 4, pages 1738–1752 (1990).

特征级声道长度归一化（VTLN）

程序 compute-mfcc-feats 和 compute-plp-feats 接收一个 VTLN 弯折因子选项。在目前的脚本中，这仅用作线性版的 VTLN 的初始化线性转换的一种方法。VTLN 通过移动三角频率箱的中心频率的位置来实现。移动频率箱的弯折函数是一个在频域空间分段线性的函数。为理解它，记住以下数量关系：

0 <= low-freq <= vtln-low < vtln-high < high-freq <= nyquist

此处，low-freq 和 high-freq 分别是用于标准 MFCC 或 PLP 计算的最低和最高频率（忽略更低和更高的频率）。vtln-low 和 vtln-high 是用于 VTLN 的截止频率，它们的功能是确保所有梅尔滤波器有合适的宽度。

我们实现的 VTLN 弯折函数是一个分段线性函数，三个部分映射区间 [low-freq, high-freq]至[low-freq, high-freq]。记弯折函数为 W(f)，f 是频率。中段映射 f 到 f/scale，scale 是 VTLN 弯折因子（通常范围为 0.8 到 1.2）。x 轴上低段和中段的连接点是满足 min(f, W(f)) = vtln-low 的 f 点。x 轴上中段和高端的连接点是满足 max(f, W(f)) = vtln-high 的 f 点。要求低段和高段的斜率和偏移是连续的且 W(low-freq)=low-freq， W(high-freq)=high-freq。这个弯折函数和 HTK 的不同；HTK 的版本中，"vtln-low"和"vtln-high"的数量关系是 x 轴上可以不连续的点，这意味着变量"vtln-high"必须基于弯折因子的可能范围的先验知识谨慎选择（否则梅尔滤波器可能为空）。

一个合理的设置如下（以 16kHz 采样的语音为例）；注意这反映的是我们理解的合理值，并非任何非常细致的调试实验的结果。

low-freq	vtln-low	vtln-high	high-freq	nyquist
40	60	7200	7800	8000

4.3 声学建模代码

（翻译：那兴宇）

简介

我们先来简单说说 Kaldi 中用于建模的代码的思路，以及为什么这样设计。我们的目标是使 Kaldi 支持传统模型，也就是对角阵高斯混合模型（GMM）和子空间高斯混合模型（SGMM），同时也要便于扩展到新的模型。在上一轮代码设计过程中，我们设计了一个虚基类，从其中衍生出 GMM 类和 SGMM 类，然后编写了一个命令行工具用于操作这两种模型。但是根据我们的经验，基类并不像事先想象的那样能派上大用场，因为这两种模型之间的差异太大（例如，它们支持的自适应算法不同）。为了构建我们想象中具有“普适性”的代码，使它能够处理

不同模型类别，我们不得不不断的扩展基类。最终，我们的命令行工具已经很难修改和维护了。

在重设计代码的时候，我们决定使用更“现代化”的软件工程方法，不为了增强普适性而过于强调类的层级结构，而是专注于创建简单的、可复用的组件。 例如，我们的解码器代码是很通用的，因为它的需求很少，它只要我们从简单的基类 DecodableInterface 中创造一个衍生的实例。这个实例操作起来类似于由一个句子的声学似然度组成的矩阵（参见 Kaldi 工具中的解码器）。同时，每个命令行工具更简单易用，例如 gmm-align 用于给定对角 GMM 获得一句话的状态级对齐结果。总的想法是，要实现某种新技术只要编写一个新的命令行程序，而不是增加已有的命令行程序的复杂度。

对角 GMM

DiagGmm 类表示一个对角协方差矩阵的高斯混合模型。一组 DiagGmm 的对象，由从 0 开始的“pdf- ids”索引组合在一起，组成声学模型。这个声学模型是由 AmDiagGmm 类实现的。尽管这个声学模型类的接口比单个 GMM 丰富，但你可以简单地把 AmDiagGmm 看作是 DiagGmm 的矢量。把声学模型表示为独立模型的集合，每个模型用于表示一个 pdf，我们可以想象，这不是对所有模型都适用的构建方法。例如，SGMM 就不能这样表示。而且，如果我们想实现不同状态之间 GMM 子成分的捆绑，就不能把每个 pdf 单独表示了。
（译者注：pdf，科技文献中写作 p.d.f.，是概率密度函数的缩写。）

独立 GMM

理论上，DiagGmm 类是一个简单的、被动的对象，负责存储高斯混合模型的参数，并且有一个用于计算似然度的成员函数。它完全不知道自己会被如何使用，它只提供获取其成员的接口。它不获取累计量 （Accumulation），不更新参数（对于参数更新，参见 MlEstimateDiagGmm 类）。DiagGmm 类存储参数的方式是：方差的倒数，以及均值乘以方差倒数的结果。这意味着，通过简单的内积操作就可以计算似然度。与 HTK 的 gconst 不同，Kaldi 中的 gconst 是独立于均值的。

由于以这种方式调整高斯参数很不方便，我们提供了 DiagGmmNormal 类，用于以简单直观的方式存储参数。同时，我们提供了 DiagGmm 和 DiagGmmNormal 实例互相转换的函数。大部分的参数更新代码作用于 DiagGmmNormal 类。

基于 GMM 的声学模型

AmDiagGmm 类表示一组 DiagGmm 对象，由 pdf-id 索引。该类不表示 HMM-GMM 模型，仅仅是一组 GMM 的集合。将其与 HMM 组合在一起由其他代码负责，主要是负责拓扑结构和转移概率部分的代码以及负责整合解码图的代码（参见 HMM 拓扑与转移建模）。在这里需要指出的是，在声学模型中，我们没有只存储一个 AmDiagGmm 对象，而是同时保存了一个 TransitionModel 对象和一个

AmDiagGmm 对象。这是为了避免在存储器上写入太多的独立文件。这样做的原因是，高斯模型和转移概率的更新往往是同时进行的。这样做的初衷是，在创建其他类型的模型时，我们可以将一个 TransitionModel 和一个目标模型的对象保存在同一个文件中。这样，那些只需要读取转移模型的程序（如创建解码图）就可以直接读取模型文件，而不需要知道其中声学模型的类型。

AmDiagGmm 类是一个相对简单的对象，并不负责模型估计（参见 AccumAmDiagGmm）和变换矩阵估计等工作。在 Kaldi 中，有其他的代码来负责这些工作，参见 Kaldi 中的特征空间与模型空间变换。

满协方差 GMM

对于满协方差矩阵的 GMM，我们创建了 FullGmm 类， 其作用与 DiagGmm 相似，区别只在与协方差矩阵的形式。 这个类主要是用来在 SGMM 建模中训练满协方差的广义背景模型（UBM）。唯一用于满方差 GMM 的命令行工具是用于训练全局混合模型的，也就是 UBM。我们没有编写 AmDiagGmm 类的满方差版本以及对应的命令行工具。不过要实现这个功能也不是难事。

子空间高斯混合模型（SGMM）

子空间高斯混合模型（SGMM）由 AmSgmm 类实现。这个类从本质上实现了``The Subspace Gaussian Mixture Model – a Structured Model for Speech Recognition"（by D. Povey, Lukas Burget et. al, Computer Speech and Language, 2011.）这篇论文中的方法。AmSgmm 类表示一个完整的 pdf 的集合。Kaldi 中没有用于表示一个 SGMM 的类（如用于 GMM 的 DiagGmm）。SGMM 参数的估计是由 MleAmSgmmAccs 类 和 MleAmSgmmUpdater 类完成的。

对于如何训练一个基于 SGMM(子空间高斯混合模型)系统的示例脚本，参见 egs/rm/s1/steps/train_ubma.sh，egs/rm/s1/steps/train_sgmma.sh 和 egs/rm/s1/steps/decode_sgmma.sh。

4.4 kaldi 里解码图的构建

首先，我们不详细介绍有限状态转换以及它在语音识别中的应用。需要了解的，可以看"Speech Recognition with Weighted Finite-State Transducers" by Mohri, Pereira and Riley (in Springer Handbook on Speech Processing and Speech Communication, 2008)。我们要用的主要方法那里面都有介绍，但是一些细节，尤其是像怎样处理混淆符，怎样处理 weight-pushing 可能与那篇论文中的就不一样了。

Overview of graph creation

整个解码网络都是围绕 HCLG = H o C o L o G 的图来建立的。

- G 是用来编码语法或者语言模型的接收器(i.e：它的输入和输出符号是一样的)
- L 是发声词典；输出是词，输入是音素。
- C 表示上下文关系，输出是音素，输入是上下文相关的音素例如 N 个音素组成的窗。具体看 Phonetic context windows.里面的介绍。
- H 包含 HMM 的定义；输出符表示上下文相关的音素，输入是 transitions-ids(转移 id)，transitions-ids 是编码 pdf-id 或者其他信息(自转或向后转)，具体看(Integer identifiers used by TransitionModel)

以上是标准方法，然而还有好多细节没有说明。我们想确保输出是确定化的和最小化的，为了让 HCLG 确定化，我们插入了消歧符。关于消歧符更多的介绍，看下面的 Disambiguation symbols。

我们也想让 HCLG 尽可能的随机。传统的方法是通过"push-weights"来达到这种效果。我们确保随机性方法与传统的不同，是基于确保没有消除随机性的图构建步骤；具体看 Preserving stochasticity and testing it。

如果我们用一行公式来总结我们的方法（很明显一行不能覆盖所有的细节），那一行应该是下面这样，asl=="add-self-loops"

rds=="remove-disambiguation-symbols"， H' 是 H 不带自环的：

HCLG = asl(min(rds(det(H' o min(det(C o min(det(L o G)))))))

Weight-pushing 没有被我们采用。相反，我们的目标是只要 G 是随机的，这样就可以确保图的构建过程中没有消除结果随机性的。当然，G 不会是绝对随机的，因为带补偿的 Arpa 语言模型是被用 FSTs 表示，但是至少我们的方法确保不随机性的部分不会增加，不会比开始变的更糟糕；这种方法避免了"push-weights"操作失败或者是变更坏的危险。

Disambiguation symbols

消歧符是在被插在词典中音素序列末尾的类似 #1, #2, #3 的符号。在词典中当一个音素序列是另一个音素序列的前缀，或者是这个音素序列出现在多个词中，就需要把这些符号加在它的后面。这些符号用来确保 L o G 输出时是确定化的。我们也插入混淆符在两个其它的地方。我们把#0 加在语言模型 G 的补偿弧上，当删除静音后(确定化的方法删除静音)，确保 G 是确定化的。我们也加#-1 在出现在上下文 FST C 左边的静音的地方，这种情况出现在句子的开始。这对于解决当有的词是用空音素(<s>)表示的问题是必要的。

下面是关于怎样证明图编译的中间过程 (e.g. LG, CLG, HCLG) 是确定化的一个概述。这对于确保我们的方法永远不失败是很重要的。我们这里讲的确定化是删除静音后的确定化。主要步骤是: 首先保证 G 必须是确定化的，这就是为什么需要#0 (这里 G 确实是确定化的)。然后对于任何确定化的 G，我们想让 L 也是这样，那么 L o G 也是确定化的。[对于 C 来说也是一样，把右边的 G 换成 L o G 即可]。这里还有很多理论的细节需要充实，但是我认为对于 L 有以下两点属性就够了：

- 必须是函数形式的

  相当于：对于任何的输入序列在 L 中必须有唯一的输出序列

  相当于：对于任何线性接受器 A，A o L 是线性转换或是空。
- L 具有双胞胎属性，比如：同一个输入符号序列不可能对应两个可达到的状态，也就是它们有相同输入序列但不同权重或者不同输出序列的自环。

这对转换器 C 同样适用。我们认为我们的脚本和代码创建的转换器都具有这些属性。

The ContextFst object

ContextFst 类的对象 (C)是一个动态建立的 FST 对象，用来表示一个从上下文相关的音素到上下文独立的音素的转换器。这个对象是用来避免我们在上下文中罗列所有可能的音素，当上下文窗长 (N)或音素数量比平常大很多时罗列所有音素的方法将会很困难。ContextFst::ContextFst 转换器需要上下文窗宽(N)和中间音素位置(P)这些都会在之前的 Phonetic context windows 中有进一步的解释。对于三音素系统他们的常用值是 3 和 1。特殊符号的整数 id 也是需要的，"subsequential symbol"(上文中提到的$)，当所有音素都被处理完后 FST 输出 N-P-1 次$(用来确保 context FST 的输出是确定化的)。除此之外也要整数 id 的音素和消歧符列表。ContextFst 输出端的词表包括音素，和加上 subsequential symbol 的消歧符。输入端的词表是自动生成的(除了静音，它未被使用), 相当于内容有上下文相关的音素，混淆符，以及一些特殊符号在传统方法中用 #-1 代替静音，然而这里我们把他们看成别的混淆符(这是用来确保确定化的, i.e.删除静音)。subsequential symbol '$'和输入端没有太多联系(对传统方法来说也一样)。输入端混淆符的 id 和输出端的相关符号的整数 id 不是绝对一样。ContextFst 类对象有一个 ILabelInfo()的函数，是用来对 std::vector<std::vector<int32> >型对象返回引用，使用户能求出输入端每个符号的"meaning"。相关的解释可以查找 The ilabel_info object。

这是 ContextMatcher 类的特殊对象"Matcher"，它是在组合算法中被使用 (Matcher 是组合算法用来做弧查找，更多细节请看 the OpenFst documentation)。 ContextMatcher 让 ContextFst 的对象使用更高效通过避免分配更多不必要的状态((the issue is that with the normal matcher, every time we want any arc out of a state, we would have to allocate the destination states of all other arcs out of that state)。当组合的左边参数是 ContxtFst 型时，这是一个相关函数，ComposeContextFst()是用来执行 FST 组合操作的。也有一个函数很类似 ComposeContext()，但是用来创建 ContextFst 对象。

Avoiding weight pushing

我们处理权重前推的整个问题上和传统方法上有微小的差别。权重前推对于 log 半环在加速搜索是有效的(权重前推是一个 FST 操作确保每个状态出弧概率的和在合适的情况下相加为 1)。然而在一些情况下 权重前推有负面影响。这个

问题来自用 FSTs 表示统计语言模型，通常相加的结果大于 1，因为一些词被记了两次（直接的或通过回退弧）。

我们决定避免可能的权重前推，所以使用了不同的方法来处理问题。1.定义"stochastic" FST，它的权重和为 1，读者可以假设这里只考虑 log 半环，而不是 tropical，所以"sum to one"就是加和的意思，不是取最大。要确保图创建的每一过程都具有那种属性：如果前一个过程是随机的，后一个过程也将是随机的。也就是说，如果 G 是随机的，LG 也是随机的；如果 LG 是随机的，那么 det(LG)也是随机的；如果 det(LG)是随机的，那么 min(det(LG))也是随机的，等等。意思就是，每个独立的操作必须在一定情况下保持随机性。现在通过一种琐碎但是很有用的方法可以实现：例如：we could just try the push-weights algorithm and if it seems to be failing because, say, the original G fst summed up to more than one, then we throw up our hands in horror and announce failure. This would not be very helpful.

我们想保持随机性在某方面，也就是说首先，对于 G，所有状态中最小和最大的，以及这些状态的出弧概率与最后的概率的和。这是我们程序"fstisstochastic"能实现的。如果 G 是随机的，所有的这些数字就都是 1（你也从程序中看到是 0，因为我们在 log 空间中执行的操作；这里是 log 半环）。我们想保持随机性在一下方面：当最小或最大从来不变的最坏；换句话说，他们从来不会离 1 很远。事实上，这很自然会发生，当我们的算法用局部的方法维保随机性。还有好多算法需要保持随机性，包括：

- Minimization 最小化操作
- Determinization 确定化操作
- Epsilon removal 删除空
- Composition 组合操作(with particular FSTs on the left) There are also one or two minor algorithms that need to preserve stochasticity, like adding a subsequential-symbol loop. Minimization naturally preserves stochasticity, as long as we don't do any weight pushing as part of it (we use our program "fstminimizeencoded" which does minimization without weight pushing). Determinization preserves stochasticity as long as we do it in the same semiring that we want to preserve stochasticity in (this means the log semiring; this is why we use our program fstdeterminizestar with the option --determinize-in-log=true). Regarding epsilon removal: firstly, we have our own version of epsilon removal "RemoveEpsLocal()" (fstrmepslocal), which doesn't guarantee to remove all epsilons but does guarantee to never "blow up". This algorithm is unusual among FST algorithms in that, to to what we need it to do and preserve stochasticity, it needs to "keep track of" two semirings at the same time. That is, if it is to preserve equivalence in the tropical semiring and stochasticity in the log semiring, which is what we need in practice, it

actually has to "know about" both semirings simultaneously. This seems to be an edge case where the "semiring" concept somewhat breaks down. Composition on the left with the lexicon L, the context FST C and the H tranducer (which encodes the HMM structure) all have to preserve stochasticity. Let's discuss this the abstract: we ask, when composing A o B, what are sufficient properties that A must have so that A o B will be stochastic whenever B is stochastic? We believe these properties are:

- For any symbol sequence that can appear on the input of B, the inverse of A must accept that sequence (i.e. it must be possible for A to output that sequence), and:
- For any such symbol sequence (say, S), if we compose A with an unweighted linear FST with S on its input, the result will be stochastic.

These properties are true of C, L and H, at least if everything is properly normalized (i.e. if the lexicon weights sum to one for any given word, and if the HMMs in H are properly normalized and we don't use a probability scale). However, in practice in our graph creation recipes we use a probability scale on the transition probabilities in the HMMs (similar to the acoustic scale). This means that the very last stages of graph creation typically don't preserve stochasticity. Also, if we are using a statistical language model, G will typically not be stochastic in the first place. What we do in this case is we measure at the start how much it "deviates from stochasticity" (using the program fstisstochastic), and during subsequent graph creation stages (except for the very last one) we verify that the non-stochasticity does not "get worse" than it was at the beginning.

至少如果 • 每个都能被恰当的归一的话(例如，如果对任何给定的词，词典权重之和都为 1，如果 H 中的 HMMs 状态被恰当的归一，而没用概率尺度表示)，这些属性对 C,L,H 也都是真实存在的。然而我们在建图的实际应用中用概率尺度表示 HMMs 的转移概率（类似声学尺度）。这就是说图构建的最后的过程不能保持随机性。如果我们用了一个随机的语言模型，G 首先不会是随机的。

4.5 Karel 的深度神经网络训练实现

Karel's DNN training implementation
综述

Karel Vesely 的 DNN 实现需要以下的技术：

- 基于 RBMS(受限波尔兹蔓机)，每一层进行预训练
- 每一帧进行交叉熵训练

- 用格子框架通过 sMBR 准则（状态的最小贝叶斯风险），对序列的区分性训练

这个系统主要是建立在 LDA-MLLT-fMLLR 特征上（从辅助的 GMM 模型得到的）。整个 DNN 训练是在一个用 CUDA 的单 GPU 上运行的，但是 cuda 矩阵库是为了没有 GPU 的机器上运行的，但是似乎慢 10 倍。（备注：这个例子只支持单 GPU 或者单 CPU，而 povey 的例子支持多 GPU 或者多 CPU）

我们用 CUDA (4.2, 5.0, 5.5) 发布版本来运行和测试我们例子。

标准数据库(rm/wsj/swbd)的脚本在：egs/{rm,wsj}/s5/local/run_dnn.sh egs/swbd/s5b/local/run_dnn.sh。

DNN 训练阶段

Pre-training（预训练）

每一层的 RBM 预训练是参考 http://www.cs.toronto.edu/~hinton/absps/guideTR.pdf。由于 DNN 的输入是高斯特征，我们的第一个 RBM 用的是 Gaussian-Bernoulli 单元，这层学习率小，和更多的训练次数。接下来的 RBMs 模型都用 Bernoulli-Bernoulli 单元。.我们是用马尔科夫链蒙特卡洛采样 (CD-1)组成的对比散度算法。在这些实验里，调整学习率是非常重要的。然而，为了更好的调整，我们需要做很多次的 fine-tuning 实验。在这个样例中我们使用 L2 正则化来增加 RBM 训练的混合率，在前 50h 的数据上动量项(momentum)是从 0.5 到 0.9 是逐步线性增加的。为了有效的保持学习速率不变，我们需要通过一个因子(1-mmt)来减小学习率。

在原始的 TIMIT 的实验里，我们有大约 200h 的数据，相对应我们的迭代次数是 50。因此我们在 switchboard 数据库上每层训练一次。根据我们跟 Toronto 里的人的讨论，我们拥有的数据越多，RBM 的预训练就越不重要。但是能让我们更好的初始化深度网络，使得反向传播算法能有一个好的初始化位置（点）。

在 RBM 训练中，我们在句子级别和帧级别上分别置乱来模仿从训练数据分布里提取样本，每一个 mini-batches 更新一次。.当我们训练 Gaussian-Bernoulli 单元，RBM 对权重扩张非常敏感，特别是有很大的学习率和许多隐含单元。为了避免这样，我们在 min-batches 里比较训练数据和重建数据的变化。如果差别很大，权重将重新调整，学习率将暂时的减小。

Frame-level cross-entropy training

在这个阶段，我们把每一帧分成三音素状态来训练 DNN。这个是通过 mini-batch 随机梯度下降算法来进行，默认的学习率为 0.008, minibatch 的大小为 256;我们不使用动量项或者正则化。

学习率在最初的几次迭代中是保持不变的，当神经网络不在提高，我们在每次训练时将学习率减半，直到它再次停止提高。

就像在 RBMs 里，我们用句子级别和帧级别的置乱来模仿从训练数据分布中日趋样本，在训练工具里就做了帧级别的置乱，在 GPU 存储空间中。

一般所有的训练工具需要从–feature-transform 接受参数，这个是进行临近特征变换的神经网络，这个通常在 GPU 里完成。在这个脚本里，我们用来把数据分块和归一化数据，但是通常是由像瓶颈特征提取这样的更多的复杂变换。

Sequence-discriminative training (sMBR)

这个阶段，神经网络被训练用来正确的对整个句子分类，这个是比帧级别的训练更加贴近普通的 ASR 的目标。sMBR 的目标就是最大化我们期望的正确率，但需要从参考的 transcriptions 的状态标签获得。我们使用 lattice framework 来表示 competing hypothesis。

这个训练是通过对 per-utterance 进行随机梯度下降更新。我们使用更低的学习速率 1e-5 不变跑 3-5 次。第一次训练后，重新生成 lattices 收敛会更快。

我们支持 MMI, BMMI, MPE and sMBR 训练。所有的方法使用非常相似，只有 sMBR 仅仅好一点点。在 sMBR 优化中，我们排除了静音帧。更多的细节 http://www.danielpovey.com/files/2013_interspeech_dnn.pdf。

代码

DNN 的代码在 src/nnet,，工具在 src/nnetbin.。因为所有的计算都通过 GPU，所有的代码都建立在 src/cudamatrix 基础上。

神经网络的表示

神经网络 Nnet 通过 Component 向量操作的。 一个 Component 与神经网络的一层不是很准确的对应。对于每一层分别有 Components 对应映射变换和非线性变换。Nnet 最重要的思想是：

- Nnet::Propagate：从输入传到输出
- Nnet::Backpropagate：将目标函数的导数通过网络后向传播
- Nnet::Feedforward：从输入传播到输出，用 2 个 flipping buffers 节省内存存储
- Nnet::SetTrainOptions：设置训练的综合参数，学习率， 动量项，L1,L2 代价。通过 const 引用， Components and buffers 是可以接受的。

请记住 DNN 是可以扩展的，我们定义了二个接口：

- Component：神经网络一层的接口，这个不需要包含可训练的参数（非线性和权重是分开 Components）。

- UpdatableComponent :通过添加支持的可训练的参数扩展的 Component。Component and UpdatableComponent 最重要的虚拟方法有：
- Component::PropagateFnc： 在前向传播中计算数据变换
- Component::BackpropagateFnc :在后向传播中转化导数
- UpdatableComponent::Update :从导数和输入中计算梯度，然后更新

Component 的实现通过：

- Sigmoid :计算逻辑 sigmoid 变换和他的导数
- Softmax： 计算 softmax 变换，假设导数已经是 wrt.激活函数了 (BackpropagateFnc 复制原来的导数)

UpdatableComponent 的实现通过：

- AffineTransform :含有一个线性变换和一个偏置项
- Rbm： 与 Restricted Boltzmann Machine 对应，包含它的参数和单元类型(当添加到 DBN，就转化 AffineTransform 和 Sigmoid)

目标函数的表示

每一帧的目标函数是通过 classes Mse (均方误差), Xent (交叉熵)来想实现的。他们的导数是通过 methods Mse::Eval, Xent::Eval and Xent::EvalVec 计算的，每个 epoch 的目标值的统计性是累积的。

Frame-level shuffling

The frame-level shuffling 是通过 Cache 和 CacheTgtMat 完成的。在 Cache 里我们假设 targets 编码成向量的索引，在 CacheTgtMat 目标中是矩阵。

Extending network by a new component

用一个新的成分来扩展 NN 框架是非常简单的，所有的可以通过以下做到：

- 1) 定义新的入口：Component::ComponentType
- 2) 在表 Component::kMarkerMap 定义新的行
- 3)添加动态的构造函数的调用到像工厂函数 Component::Read
- 4)实现界面 Component:: 或者 UpdatableComponent::

Binary tools

这些训练工作都在 src/nnetbin，最重要的工具是：

- nnet-train-xent-hardlab-frmshuff.cc：帧级别的训练，对后验概率和目标之前的交叉熵做最小化，允许在时间轴上做拼接，帧级别的置乱是在特征转换后完成，进行一次训练。学习率减半是在 bash 脚本上完成的。
- nnet-forward.cc：这个工具通过神经网络传播输入特征，他可以去除运行中的 softmax 和基于三音素状态的数目来通过先验划分。这个工具常常用来特征提取中的解码部分，如果你运行在一个没有 GPU 的机器，一个 CPU 就被代替。
- rbm-train-cd1-frmshuff.cc：训练 RBM，可以通过逐步线性增加动量项来计算迭代的时候
- nnet-train-mmi-sequential.cc：可以用来 MMI/bMMI 训练
- nnet-train-mpe-sequential.cc：可以用来 MPE / sMBR 训练

4.6 Kaldi 中的关键词搜索

介绍

本文描述了在kaldi中的关键词搜索模块。我们实现了以下功能：

- 对网络进行索引以便快速检索关键词
- 使用代理关键词以处理表外词（OOV）问题

在以下的文档中，我们将重点讨论为特殊目的而进行的基于词的关键词检索，但我们同样支持子词级别的关键词检索。我们的语音识别模块和关键词检索模块都使用加权有限状态转换器wfst，这个算法也支持使用符号表将词/子词映射到整数。

本文剩下的部分结构如下：在Typical Kaldi KWS system部分，我们描述了kaldi kws 系统的基本模块；在Proxy keywords部分，我们解释了我们如何使用代理关键词处理不在词汇表中的关键词；最后在Babel scripts部分，我们介绍了为iarpa bable项目所做的kws相关的脚本。

Typical Kaldi KWS system

我们在文章"Quantifying the Value of Pronunciation Lexicons for Keyword Search in Low Resource Languages", G. Chen, S. Khudanpur, D. Povey, J. Trmal, D. Yarowsky and O. Yilmaz中可以看到kaldi kws 系统的例子。一般来说，一个kws系统包括两个部分：一个lvcsr 模块解码检索集合并且产生相应的网格，一个kws模块生成网格索引并从索引中查找关键词。

我们的基础lvcsr 系统是一个sgmm+mmi 系统，我们使用标准的PLP分析器抽取13维的语音特征，然后用一个典型的最大似然估计进行语音训练，以一个平滑的上下文无关的音素HMM做初始值开始，以说话人自适应（SAT）的状态集群三音素hmm-gmm做为输出结束。在说话人变换训练集的统一背景模型（UBM)进行训练，然后得到用于训练HMM发射概率的子空间高斯混合模型（SGMM），最后，所有的训练语音使用SGMM系统进行解码，然后对sgmm的参数进行bmmi训练。更详细的细节参见egs/babel/s5b/run-1-main.sh

我们在sgmm_mmi系统之外还创建了一些附加的系统，如：一个混合深度神经网络(DNN).详情见egs/babel/s5b/run-2a-nnet-gpu.sh,一个瓶颈特殊(BNF)系统，详情见egs/babel/s5b/run-8a-kaldi-bnf.sh 等等。所有这些系统都是对相同的检索集合进行解码并且生成网格，随后送到kws 模块进行索引和检索。我们在检索结果上而不是在网格上将这些系统组织起来。

由上述 lvscr 系统产生的网格，使用在文章"Lattice indexing for spoken term detection", D. Can, M. Saraclar, Audio, Speech, and Language Processing 中描述的网格索引技术进行处理。所有待检索集语句中的网格都被从单一加权有限状态转换成一个单广义因数变送器结构，将每个词的开始时间，结束时间和网格后验概率这三维数据存储起来。给一个词或短语，我们创建他的简单有限状态机，可以得到这个关键词/短语并且将他与因数变送器得到关键词/短语在检索集合中所有出现过的地方，和一个语句的 ID 号，开始时间，结束时间，以及每个地方网格的后验概率。所有检索出来的结果以他们的后验概率进行排序，并且使用论文"Rapid and Accurate Spoken Term Detection"中的方法来对每个实例判断是或否。

Proxy keywords 代理关键词

我们的代理关键词产生过程在论文 "Using Proxies for OOV Keywords in the Keyword Search Task", G. Chen, O. Yilmaz, J. Trmal, D. Povey, S. Khudanpur 做了描述。我们最早使用这个方法解决词网格中的 oov 问题，如果关键词不在 lvcsr 系统的词汇表中，尽管实际上这个词在谈话中已经被提到了，但他也不会出现在检索集的网格中。这是 lvcsr 为基础的关键词检索系统的一个老问题，并且已经有办法解决它，比如，创建一个子词系统。我们的办法是寻找语音中与词汇表中相似的词，并且使用这些词做为代理关键词替换那些表外词汇。这样做的好处是我们不用创建额外的子词系统。在即将到来的 interspeech 的论文"Low-Resource Open Vocabulary Keyword Search Using Point Process Models", C. Liu, A. Jansen, G. Chen, K. Kintzley, J. Trmal, S. Khudanpur 中我们证明这技术可以同一个建立在点处理模型上的音素检索模型相媲美。代理关键词是一种模糊检索方法，他在处理表外词的同时也可以加强 IV 关键词性能。

通常的代理关键词处理过程可以用以下公式产生：

$$K' = \text{Project}\left(\text{ShortestPath}\left(\text{Prune}\left(\text{Prune}\left(K \circ L_2 \circ E'\right) \circ L_1^{-1}\right)\right)\right)$$

这里K为原始的关键词，L_2是包含发音的K词汇。如果K是表外词，这个词汇可以使用G2P工具得到。E'是编辑距离转换，表示音素与训练集的差异程度。L_1是原始词汇。K'是包含多个IV词的WFST，是与原始发音K相似的发音词。我们把他当成原始词进行检索。

注意两个修正阶段是必不可少的，尤其当你的词汇表非常大的时候。我们同时实现了一个简单的组合算法，只产生必须的组合状态（比如，不产生以后会被修正的状态）。这避免了在计算$K \circ L_2 \circ E'$和L_1^{-1}组合的时候使用内存太多。

Babel scripts

A highlevel look 概述

我们为IARPA Babel 项目建立了一个“一键”脚本。如果你在Babel 上进行实验时想用我们的脚本，你可以用以下几步建立一个SGMM+MMI的关键词检索系统（假设你的工作目录是 egs/babel/s5b/)

1、安装F4DE 并且将他设置在你的path.sh中。

2、修改cmd.sh以保证可以运行在你的集群 。

3、将conf/languages中的一个文件连接到./lang.conf,

比如 "ln -s conf/languages/105-turkish-limitedLP.official.conf lang.conf"

4、修改lang.conf中的JHU集群中的数据使它指向你的数据文件。

5、运行run-1-main.sh ,建议lvcsr系统。

6、运行run-2-segmentation.sh, 产生eval 数据的分割

7、运行run-4-anydecode.sh,解码eval 数据，生成索引检索关键词

同时，你可以建立DNN系统，BNF系统，Semi-supervised 系统等等语音识别系统。关键词检索任务在run-4-anydecode.sh中执行。我们将在下面介绍如何进行关键词检索的细节，你可以把这些方法用于其它的数据上。我们假设你已经解码了检索集合并且产生的相应的网格。

关键词检索数据准备

Prepare KWS data

一般来说，我们在检索数据的目录中建立kws的数据目录。例如，如果你有一个叫做dev10h.uem的检索集。数据所在的目录为data/dev10.uem/. 我们在在这个目录下面创建kws 数据目录，如data/dev10h.uem/kws. 在创建kws目录之前，你必须有三个文件，一个包含检索信息的ecf文件，一个关键词列表kwlist文件，一个打分文件rttm. 有时你必须自己创建这些文件，例如，你可以创建一个rttm文件指定模型中的检索的参数。下面我们列出这些文件的格式：ECF文件的例子：

```
    <ecf source_signal_duration="483.825" language="" version="Excluded noscore regions">
  <excerpt audio_filename="YOUR_AUDIO_FILENAME" channel="1" tbeg="0.000" dur="483.825" source_type="splitcts"/>
</ecf>
```

KWLIST 文件的例子：

```
    <kwlist ecf_filename="ecf.xml" language="tamil" encoding="UTF-8" compareNormalize="" version="Example keywords">
  <kw kwid="KW204-00001">
      <kwtext>       </kwtext>
  </kw>
```

```
  <kw kwid="KW204-00002">
    <kwtext>        </kwtext>
  </kw>
</kwlist>
```

RTTM 文件的例子：

```
SPEAKER YOUR_AUDIO_FILENAME 1 5.87 0.370 <NA> <NA> spkr1 <NA>
LEXEME YOUR_AUDIO_FILENAME 1 5.87 0.370 lex spkr1 0.5
SPEAKER YOUR_AUDIO_FILENAME 1 8.78 2.380 <NA> <NA> spkr1 <NA>
LEXEME YOUR_AUDIO_FILENAME 1 8.78 0.300 lex spkr1 0.5
LEXEME YOUR_AUDIO_FILENAME 1 9.08 0.480 lex spkr1 0.5
LEXEME YOUR_AUDIO_FILENAME 1 9.56 0.510 lex spkr1 0.5
LEXEME YOUR_AUDIO_FILENAME 1 10.07 0.560 lex spkr1 0.5
LEXEME YOUR_AUDIO_FILENAME 1 10.63 0.350 lex spkr1 0.5
LEXEME YOUR_AUDIO_FILENAME 1 10.98 0.180 lex spkr1 0.5
```

准备好这些文件后，你可以开始准备 kws 数据目录。如果你只是想进行基本关键词检索，运行以下命令：

```
local/kws_setup.sh \
  --case_insensitive $case_insensitive \
  --rttm-file $my_rttm_file \
  $my_ecf_file $my_kwlist_file data/lang $dataset_dir
```

如果你要对表外词汇进行模糊检索，你可以运行以下几个命令，这些命令首先收集相似音素，然后训练 G2P 模型，创建 KWS 数据目录：

```
#Generate the confusion matrix
#NB, this has to be done only once, as it is training corpora dependent,
#instead of search collection dependent
if [ ! -f exp/conf_matrix/.done ] ; then
  local/generate_confusion_matrix.sh --cmd "$decode_cmd" --nj $my_nj \
    exp/sgmm5/graph exp/sgmm5 exp/sgmm5_ali exp/sgmm5_denlats \
exp/conf_matrix
  touch exp/conf_matrix/.done
fi
confusion=exp/conf_matrix/confusions.txt

if [ ! -f exp/g2p/.done ] ; then
  local/train_g2p.sh  data/local exp/g2p
  touch exp/g2p/.done
fi
local/apply_g2p.sh --nj $my_nj --cmd "$decode_cmd" \
  --var-counts $g2p_nbest --var-mass $g2p_mass \
  $kwsdatadir/oov.txt exp/g2p $kwsdatadir/g2p
L2_lex=$kwsdatadir/g2p/lexicon.lex

L1_lex=data/local/lexiconp.txt
local/kws_data_prep_proxy.sh \
  --cmd "$decode_cmd" --nj $my_nj \
  --case-insensitive true \
  --confusion-matrix $confusion \
  --phone-cutoff $phone_cutoff \
  --pron-probs true --beam $beam --nbest $nbest \
  --phone-beam $phone_beam --phone-nbest $phone_nbest \
  data/lang  $data_dir $L1_lex $L2_lex $kwsdatadir
```

Indexing and searching 索引和检索

在这一阶段，我们假设你已经解码了检索集合并且生成了相应的网格。运行以下脚本将进行索引和检索：

```
    local/kws_search.sh --cmd "$cmd" \
  --max-states ${max_states} --min-lmwt ${min_lmwt} \
  --max-lmwt ${max_lmwt} --skip-scoring $skip_scoring \
  --indices-dir $decode_dir/kws_indices $lang_dir $data_dir $decode_dir
```

如果你的KWS数据目录有一个额外的ID号，比如 oov(当你进行不同的kws时，这很有用，这种情况下，你的目录可以是这样的data/dev10h.uem/kws_oov)，你必须使用extraid 选项：

```
    local/kws_search.sh --cmd "$cmd" --extraid $extraid    \
  --max-states ${max_states} --min-lmwt ${min_lmwt} \
  --max-lmwt ${max_lmwt} --skip-scoring $skip_scoring \
  --indices-dir $decode_dir/kws_indices $lang_dir $data_dir $decode_dir
```

4.7 kaldi 中的关键词检索(之前的版本)

（@刘诗涵翻译）

Keyword Search in Kaldi

Introduction

本部分主要对 kaldi 中关键字搜索模块的内容做相关介绍。我们会简要介绍在论文"Lattice indexing for spoken term detection", D. Can, M. Saraclar, Audio, Speech, and Language Processing 中提出的关键字搜索算法，但是这里会着重介绍搜索算法的具体实现过程以及我们对算法做的一些延伸。

Lattice Indexing

在此我们只是简单的介绍一下算法，如果想了解更多的细节，你可以去查看相关网格索引（lattice indexing）的文章。

Related Semiring

索引算法是基于一种特殊设计的半环（semiring），此种半环可以同时存储时间和权重（置信度）的信息。强烈建议读者去仔细阅读理解半环的详细定义，但是以防读者忘记相关概念，我们在此对其进行简单回顾：

定义 1：幺半群（注 monoid: 幺半群是指一个带有可结合二元运算和单位元的代数结构）是一个三元组$\left(\mathrm{k},\otimes,\ \overline{1}\right)$，其中$\otimes$是在数据集$\mathrm{k}$上的一个封闭的二元结合运算，$\overline{1}$是$\otimes$中的单位元。如果二元运算$\otimes$在数据集$\mathrm{k}$是可交换的那么此幺半群也是可交换的。

定义 2：半环（semiring：在抽象代数中，半环是类似于环但没有加法逆元的代数结构）是一个五元组$\left(\mathrm{k},\oplus,\otimes,\ \overline{0},\overline{1}\right)$，并且符合如下条件：

1.$\left(\mathrm{k},\otimes,\ \overline{1}\right)$是一个幺半群；

2.$\left(\mathrm{k},\oplus,\ \overline{0},\right)$是一个可交换幺半群；

3.$\otimes$对$\oplus$符合分配率，即

$\forall a,b,c\in k,a\otimes\left(b\oplus c\right)=a\otimes b\oplus a\otimes c$且$\left(\mathrm{a}\oplus b\right)\otimes c=a\otimes c\oplus b\otimes c$；

$\forall a\in k,\overline{0}\otimes a=a\otimes\overline{0}=\overline{0}$

下面介绍在算法中使用到的 5 种半环（semiring）：

对数半环（Log semiring）：对数半环（Log semiring）的定义如下：

$$\mathrm{L}=\left(\mathrm{R}\cup\{-\infty,+\infty\},\oplus_{\log},+,+\infty,0\right),$$

其中$\forall a,b\in\mathrm{R}\cup\{-\infty,+\infty\}$；

$a\oplus_{\log}b=-\log\left(e^{-a}+e^{-b}\right)$，而$e^{-\infty}=0$，$-\log(0)=\infty$。

热带半环 （Tropical semiring）：此种半环的格式为

$$T=\left(\mathrm{R}\cup\{-\infty,+\infty\},\ \min,+,+\infty,0\right),$$

其中操作符$\min$（得到最小值）是用来做 tropical 加法（tropical addition）的。

北极半环（Arctic semiring）: 此种半环的格式为

$$T=\left(\mathrm{R}\cup\{-\infty,+\infty\},\ \max,+,+\infty,0\right),$$

其中操作符$\max$（得到最大值）是用来做 arctic 加法（Arctic addition）。因为其操作正好与热带半环（tropical semiring）相反，因此得名北极半环（arctic semiring）。$\forall a_1,a_2\in k_A,\mathrm{b}_1,\mathrm{b}_2\in k_B$，则$\left(a_1,\mathrm{b}_1\right)\oplus_{\times}\left(a_2,\mathrm{b}_2\right)=\left(a_1\oplus_{k_A}a_2,\ \mathrm{b}_1\oplus_{k_B}b_2\right)$

产品半环（Product semiring）:此种半环的格式为：

$$A\times B=\left(k_A\times k_B,\oplus_{\times},\otimes_{\times},\overline{0}_{k_A}\times\overline{0}_{k_B},\overline{1}_{k_A}\times\overline{1}_{k_B}\right),$$

其中$A=\left(k_A,\oplus_{k_A},\otimes_{k_A},\overline{0}_{k_A},\overline{1}_{k_A}\right)$和$B=\left(k_B,\oplus_{k_B},\otimes_{k_B},\overline{0}_{k_B},\overline{1}_{k_B}\right)$是偏序半环；$\oplus_{\times}$和$\otimes_{\times}$式能量算子符号（component-wise operators），例如：

词典半环（Lexicographic semiring）: 词典半环的定义为：

$$A*B=\left(k_A\times k_B,\oplus_{*},\otimes_{*},\overline{0}_{k_A}\times\overline{0}_{k_B},\overline{1}_{k_A}\times\overline{1}_{k_B}\right),$$

其中$A=\left(k_A,\oplus_{k_A},\otimes_{k_A},\overline{0}_{k_A},\overline{1}_{k_A}\right)$和$B=\left(k_B,\oplus_{k_B},\otimes_{k_B},\overline{0}_{k_B},\overline{1}_{k_B}\right)$是偏序半环；$\oplus_{*}$是字典算符优先符号，例如：$\forall a_1,a_2\in k_A,\mathrm{b}_1,\mathrm{b}_2\in k_B$，则

$$(a_1,\mathrm{b}_1)\oplus(a_2,\mathrm{b}_2)=\begin{cases}(a_1,\mathrm{b}_1\oplus_{k_B} b_2),\text{if } a_1=a_2\\(a_1,\mathrm{b}_1),\text{if } a_1=(a_1\oplus_{k_A} a_2)\neq a_2\\(a_2,\mathrm{b}_2),\text{if } a_1\neq(a_1\oplus_{k_A} a_2)=a_2\end{cases}$$ 。

Algorithm

索引算法是把由 ASR 过程中生成的网格（Lattice）转变为加权系数转换器（weighted factor transducer），这样就可以通过结合结果转换器（resulting transducer）与经过合理定义的关键词转换器（keywords transducers）（也包括关键句子，但是为了简单在接下来的文中仅使用关键字），就可以得到关键字的时间信息和表示置信度的分数（scores indicating the confidence）. resulting factor transducer 也称作索引，有如下性质：每一条路径代表原网格中的部分路径，而路径上的权重信息则表示相关时间信息和得分/置信度。需要强调的是最后索引上的权重的回推到整条路径，独立的每条边的权重没有意义。

从原始网格中建立如上的所以大概分为以下五个步骤：

权重推进和聚类 （Weight Pushing and Clustering）: 这两步有时候可以理解为预处理。在原始网格中的权重信息通常对应于由语言模型和声学模型所分配的联合概率，而权重推进步骤把这些权重信息转化成所需要的基于网格的后验概率。聚类的目的是把连接相同的字以及相同时间段内的边合并。聚类是通过“标签”实现的：如果有几条边属于相同的类（连接相同的字以及有时间上的重叠），那么就给这些边相同的输出“标签”（输入标签仅仅是简单的单词）。当在转换器（transducer）上面进行确定化处理时才会对不同类之间进行真正的合并。

因子生成（Factor Generation）: 正像我们前面所提及的，在最后索引中的每条路径都是原网格的部分路径，这是通过因子生成（factor generation）这步来实现的。对于每一个网格，我们新增加两个状态，起始状态和终止状态。网格中的每个状态，新增加一条从起始状态到此状态的路径，以及此状态到终止状态的路径。在此步中还对半环做了修改，从对数半环 L 变化到半环 $L\times T\times T'$，其中 T 是热带半环（tropical semiring），T' 是北极半环（arctic semiring），$\times$ 表示产品半环（product semiring）。其中使用对数半环 L 来表示置信度得分，热带半环 T 表示部分路径的起始时间，以及北极半环 T' 表示部分路径的结束时间。需要注意在新增加的起始状态和终止状态之间添加路径的时候，必须保证推进权重（“pushed” weight）中的时间信息代表此部分路径的起始和结束时间。

因子合并（Factor merging）: 此步骤完成在聚类步骤中提到的对聚类进行合并，是通过确定化编码处理来实现的。在完成合并后，就没有北极半环（arctic

semiring）并且产品半环（product semiring）也变为词典半环（lexicographic semiring）。即，把$L \times T \times T^{'}$变为$T*T*T$，同时去掉在聚类步骤中生成的聚类标签。

因子消歧（Factor Disambiguation）：在结果传感器（resulting transducer）中的每条连接最后状态的边均对应着在原始网格中不同的独立的部分路径，通过消歧因子使其能在后续步骤中保持相互独立。

优化（optimization）：此步中包括空符号移除（epsilon removal），确定化和最小化等的优化操作。

通过上述的几个步骤，就可以把每个网格转化为小的索引，通过结合不同的小的索引变为一个大的索引。进一步的优化可以在最后完成的大的索引上进行。

Implementation Details

因为 kaldi 是基于 openfst 的工具箱，这就使得其实现变得相对简单。直接从 Kaldi:CompactLattice 开始，在其中收集时间信息和置信度得分信息，然后编译使其生成为因子传感器。通常情况下使用在索引文章中所描述的算法，但是会需要做一些本地自适应和修改。在此部分中我们列出此些修改。

半环定义（semiring definition）

我们对在 openfst 中现有的大部分半环进行了模块化的处理，除了北极半环（arctic semiring），是因为北极半环没有可以直接使用的半环。模块化的半环如下所示：

```
// The T*T*T semiring
typedef fst::LexicographicWeight<TropicalWeight, TropicalWeight>
StdLStdWeight;
typedef fst::LexicographicWeight<TropicalWeight, StdLStdWeight>
StdLStdLStdWeight;
typedef fst::ArcTpl<StdLStdLStdWeight> StdLStdLStdArc;

// The LxTxT' semiring
typedef fst::ProductWeight<TropicalWeight, ArcticWeight>
StdXStdprimeWeight;
typedef fst::ProductWeight<LogWeight, StdXStdprimeWeight>
LogXStdXStdprimeWeight;
typedef fst::ArcTpl<LogXStdXStdprimeWeight> LogXStdXStdprimeArc;
```

```
// Rename the weight and arc types to make them look more "friendly".
typedef StdLStdLStdWeight KwsLexicographicWeight;
typedef StdLStdLStdArc KwsLexicographicArc;
typedef fst::VectorFst<KwsLexicographicArc> KwsLexicographicFst;
typedef LogXStdXStdprimeWeight KwsProductWeight;
typedef LogXStdXStdprimeArc KwsProductArc;
typedef fst::VectorFst<KwsProductArc> KwsProductFst;
```

其中 fst::ArcticWeight 是在 lat/arctic-weight.h 中定义的。

权重推进（Weight Pushing）

在原始的算法中，权重推进是在预处理步骤完成的，但是我们发现如果在因子生成（factor generation）步骤中完成权重推进会更高效。计算先验概率 alphas、后验概率 betas，通过手动完成权重推进。我们把此步向后移，这样就可以使得在因子生成（factor generation）步骤中计算最短路径的时候可以重用 alphas 和 betas 的信息。对于一条给定的弧，假定 c 为它的 cost，通过如下步骤修改 cost：

```
non-final:
c <-- c - beta[destination-state of arc] + beta[start-state of arc]
final:
c <-- c + beta[start-state of arc]
```

在完成修改之后把 betas 置为 0，通过如下公式来修改 alphas：

```
alpha[s] <--- alpha[s] + beta[s] - beta[initial-state]
```

现在“好像”是通过 pushed FST 计算出 alphas 和 betas。

静音消除（Long Silence Removal）

在因子生成（factor generation）步骤中形成的传感器（transducer）可以使得我们可以搜索原网格中的任意一条部分路径，但是在这其中只有部分路径是有意义的。因此可以在因子传感器（factor transducer）的顶层使用一些滤波技术。在我们实现的过程中使用基于词间静音长度的滤波器。如果两个词间的静音长度太长（在 babel 项目中此时间设为 0.5 秒），那么这两个单词之间也许是不连续的，我们可以在索引中去掉此路径。如果两个单词之间的静音时间很长，则把指向下个单词的弧转而指向立即状态，然后在整个传感器上运行 Connect（），此步会去掉没有意义的路径。

消歧符号（Disambiguation Symbol）

在 Dogan 和 Murat's 的文章中，在完成索引上的最后优化后，去掉输入标签域的消歧符号，使得在搜索时间内完成与关键字 FST 的结合。但是这样使得在搜索的结果上很难做进一步的优化（通过结合索引与关键字 FST 形成结果 FST），

因为来自相同句子的不同实例可能会结合。在我们实现的过程中并没有完全去掉消歧符号。我们把消歧符号与句子的 ID 编码在一起，从最后边中去掉原始的消歧符号，并且用编码后的标签去替代最后边的输出标签。这样我们就可以在结果 FST 中进行进一步优化（例如确定化和最小化），因为有时候结果 FST（resulting FST）会非常大。

Running KWS in Kaldi

在 kaldi 上运行 KWS

在 kaldi 中运行 KWS 时有两个重要的文件：lattice-to-kws-index 和 kws-search。从他们各自的名字中我们也可以分辨其各自的作用，lattice-to-kws-index 是读入网格信息输出索引，kws-search 则是在索引上进行关键字的搜索。

lattice-to-kws-index 可以通过下面的命令来运行：

```
lattice-to-kws-index ark:utter_id "ark:gzip -cdf lat.1.gz|" ark:1.index
```

其中 lat.1.gz 是 kaldi 中网格的 ARK 格式文件（注：.ark 格式可以是 text 或 binary 格式，（你可以写为 text 格式，命令行要加‘t’，binary 是默认的格式）文件里面数据的格式是：key（如句子的 id）空格后接数据。摘自博客 my9084 的博客 http://blog.sina.com.cn/u/1145070023）， 1.index 包含在 achieve 中每个网格的一个索引。Utter_id 是对每个句子的 ID 建立的符号表，如下所示：

```
10713_A_20111024_220917_000665 1
10713_A_20111024_220917_001321 2
10713_A_20111024_220917_002097 3
10713_A_20111024_220917_002550 4
10713_A_20111024_220917_003311 5
10713_A_20111024_220917_003903 6
10713_A_20111024_220917_004616 7
10713_A_20111024_220917_004976 8
10713_A_20111024_220917_005656 9
10713_A_20111024_220917_006296 10
.....
```

在检索之前，为了得到更多准确的时间信息，网格通常需要“对齐”，有个时候对某些特定的词赋予“惩罚”系数（penalty）。在检索之后，会有一个合并的过程这样使得在原 archive 中的所有网格可以形成单个索引。一些实例脚本如下：

```
lattice-add-penalty "ark:gzip -cdf lat.1.gz|" ark:- |\
   lattice-align-words $word_boundary $model    ark:- ark:- |\
 lattice-scale --acoustic-scale=$acwt --lm-scale=$lmwt ark:- ark:- |\
   lattice-to-kws-index ark:utter_id ark:- ark:- |\
kws-index-union ark:- "ark:|gzip -c > index.1.gz"
```

为了在已编译好的索引上搜索关键字，关键字的编译为 FST 格式，如下：

```
transcripts-to-fsts ark:keywords.int ark:keywords.fsts
```

其中 keywords.int 文件由一系列形如<关键字 id，关键字内容>所组成，如下：

```
KW101-0001 2669 4878
KW101-0002 3419
KW101-0003 6792
KW101-0004 867 9757
KW101-0005 2055 8016
KW101-0007 5959 5450 16796
KW101-0008 1560 867 9818
KW101-0009 4213
KW101-0010 4305
.....
```

在上述文件中关键字已经映射到其对应的符号上，其文本格式如下所示：

```
KW101-0001  冇 嘸
KW101-0002  化疗
KW101-0003  屋企人
KW101-0004  个 时间
KW101-0005  保障 成本
KW101-0006  彭鱼
KW101-0007  好 多 野
KW101-0008  今 个 星期日
KW101-0009  吃饭
KW101-0010  后卫
```

在编译完关键字的 FST 后，搜索仅需要一条命令：

```
kws-search "ark:gzip -cdf index.1.gz|" ark:keywords.fsts \
   "ark,t:|int2sym.pl -f 2 utter_id > result.1"
```

我们把搜索的结果存储在 result.1 文件中，此文件中的每一行都包括五个部分，对应单独的一条搜索结果。下图所示的是在 result.1 文件中的前几行内容，其中每行的第一部分表示关键字的 ID，第二部分为搜索句子的 ID，第三部分是相对于所搜索句子起始位置的关键字出现的起始帧，第四部分为相对的关键字的结束帧，最后一部分为搜索结果“否定”的对数概率（"negated" log probability）：

```
KW101-0001-A 10713_A_20111024_220917_026395 163 194 2.876953
KW101-0001-A 10713_A_20111024_220917_008532 154 187 5.734375
KW101-0003-A 10733_A_20111021_141006_001440 448 510 1.737305
KW101-0010-A 10713_A_20111024_220917_028639 471 522 7.064453
KW101-0010-A 10733_A_20111021_141006_043478 224 275 8.200195
```

```
KW101-0012-A 10713_A_20111024_220917_026395 255 314 7.344727
KW101-0016-A 10733_A_20111021_141006_028914 738 800 0.1992188
KW101-0016-A 10733_A_20111021_141006_028914 738 803 4.556641
KW101-0016-A 10713_A_20111024_220917_020531 224 258 4.624023
KW101-0016-A 10733_A_20111021_141006_020971 233 273 8.793945
```

所有以上所提及的步骤都可以通过现有脚本来实现，脚本的位置在egs/wsj/s5/run.sh。

Proxy Keywords

在 Kaldi 中的关键字模块中，引入代理关键字来处理不在词汇表内（out Of ocabulary，OOV）的关键字的问题。这里的“代理”指的是与原 OOV 关键字发音较接近的字。代理关键字可以通过下述公式产生：

$$K^{'} = \text{project}\left(\text{ShortestPath}\left(\text{K} \circ L_2 \circ E^{'} \circ \left(L_1\right)^{-1}\right)\right)$$

其中 K 是原关键字，L_2 是由类似于 Sequitur 的 G2P 工具生成的 OOV 词典，$E^{'}$ 是包含音素模糊（phone confusion）的距离编辑传感器（edit Distance transducer），L_1 是原来的词典部分关键词可能不在此词典中。假设 OOV 词典 L_2 可以有标准的工具生成，对于每个因素的模糊因素都可以收集到，代理关键字的生成可以通过下列步骤完成：

首先，要生成距离编辑传感器（edit Distance transducer）。假设我们已经得到了 phone_confusion，可以通过运行 egs/babel/s5 中的如下命令来实现：

```
local/build_edit_distance_fst.pl \
   --confusion-matrix phone_confusion phones.txt - |\
   fstcompile --isymbols=phones.txt --osymbols=phones.txt - Edit.fst
```

文件 phones.txt 是包含在词典中要使用到的因素的文件。Phone_confusion 文件包含每对因素组合因素模糊（phone confusion）的“否定”对数概率（"negated" log probability），如下：

```
o oj 7.18083119904456
v tS 4.35670882668959
u ow 5.20766299827991
dZ S 4.91998092582813
h dZ 6.21460809842219
z A 4.18965474202643
iw U 3.93182563272433
b uj 6.40025744530882
j N 3.56751859710967
D z 3.85014760171006
```

这两个词典都需要编译为 FST 格式

```
cat oov.lex | utils/make_lexicon_fst.pl - |\
```

```
    fstcompile --isymbols=phones.txt --osymbols=words.txt - |\
    fstinvert | fstarcsort --sort_type=olabel > L2.fst
cat original.lex | utils/make_lexicon_fst.pl - |\
    fstcompile --isymbols=phones.txt --osymbols=words.txt - |\
    fstarcsort --sort_type=ilabel > L1'.fst
fstcompose L2.fst Edit.fst |\
    fstarcsort --sort_type=olabel > L2xE.fst
```

最后可以通过下述命令调用 generate-proxy-keywords 来生成代理关键字。

```
generate-proxy-keywords --verbose=1 \
    --cost-threshold=$beam --nBest=$nbest \
    L2xE.fst L1'.fst ark:oov_keywords.int ark:proxy.fsts
```

所有上述的步骤都被整合在一个脚本里，这个脚本就是：

egs/babel/s5/local/generate_proxy_keywords.sh

在生成代理关键字后，可以当作普通的关键在原索引上进行搜索。但是需要注意的是对返回的搜索结果可能会需要不同的处理（例如：它们可能需要不同的阈值）以及进行调优。

4.8 在线识别

（@冒顿翻译）

在kaldi 的工具集里有好几个程序可以用于在线识别。这些程序都位在src/onlinebin文件夹里，他们是由src/online文件夹里的文件编译而成(你现在可以用make ext 命令进行编译).这些程序大多还需要tools文件夹中的portaudio 库文件支持， portaudio 库文件可以使用tools文件夹中的相应脚本文件下载安装。

这些程序罗列如下：

online-gmm-decode-faster: 从麦克风中读取语音，并将识别结果输出到控制台

online-wav-gmm-decode-faster:读取wav文件列表中的语音，并将识别结果以指定格式输出。

online-server-gmm-decode-faster:从UDP连接数据中获取语音MFCC向量，并将识别结果打印到控制台。

online-net-client :从麦克风录音，并将它转换成特征向量，并通过UDP连接发送给online-server-gmm-decode-faster

online-audio-server-decode-faster :从tcp连接中读取原始语音数据，并且将识别结果返回给客户端

online-audio-client :读出wav文件列表，并将他们通过tcp 发送到online-audio-server-decode-fater,得到返回的识别结果后，将他存成指定的文件格式

代码中还有一个java版的online-audio-client ,他包含更多的功能，并且有一个界面。另外，还有一个与GStreamer 1.0兼容的插件，他可以对输入的语音进行识别，并输出的文字结果。这个插件基于onlinefasterDecoder,也可以做为在线识别程序

在线语音服务器。

online-server-gmm-decode-faster 和online-audio-server-decode-faster的主要区别是各自的输入不同，前者接受特征向量做为输入，而后者则接收原始音频。后者的好处是他可以处理任意客户端直接传过来的数据：不管他是互联网上的计算机还是一部移动设备。最主要的是客户端不需要知道训练模型使用了何种特征集，只要设定好预先定义的采样率和位深度，他就可以跟服务器进行通信。使用特征向量做为输入的服务器的一个好处是，在客户端和服务器之间传输的数据比音频服务器小得多，而这只需使用简单的音频代码就可以实现。

在online-audio-client和online-audio-server-decode-faster之间的通信包括两个步骤：第一步客户端将原始音频数据包发送给服务器，第二步服务器将识别结果发回给客户端。这两步可以同步进行，这意味着解码器不用等音频数据发送完毕，就可以在线输出结果，这给未来新的应用提供了很多可以扩展的功能。

音频数据

音频数据格式必须是采样率16KHz，16-bit 位深，单声道，线性PCM编码的硬编码数据。

通信协议数据分为块和前缀，每一个块包含4字节指示此块的长度。这个长度值也是一个long型little-endian值,可以是正也可以是负（因为是16-bit采样)。最后一个长度为0的包被当作音频流的结束，这将迫使解码器导出所有剩余的结果，并且结束识别过程

识别结果

服务器返回两种类型的结果。时间对齐结果和部分结果。

解码器识别出每一个词都会立即发送一个部分结果，而当解码器识别到一次发音结束时则会发送一个时间对齐结果（这可能是也可能不是一次停顿或者句子结束）。

每一个部分结果包会有以字符串：“PARTIAL:"开头，后面跟一个词，每一个词发送一次不同的部分结果包。时间对齐结果包会以字符串："RESULT:"开头。后面是以逗号为分格的key=value参数列表，这些参数列表包含一些有用信息。

NUM:后面词的个数
FORMAT :返回结果的格式，目前只支持WSEC格式（词-开始-结束-置信度),未来会允许修改
RECO-DUR: 识别语音所使用的时间。（以秒计的浮点型值）
INPUT-DUR:输入语音的时间长度（以秒计的浮点型值)
你可以用识别时间除以输入时间得到解码器的实时识别速度。当服务器将识别结果全部返回后会发送一个"RESULT:DONE", 这种情况下服务器会等待输入或是断开。

在数据头部后面包含有NUM行以FORMAT格式组成的词行，现在的WSEC格式，只是简单以逗号分隔的四个部分：词，起始时间，终止时间（以秒计的浮点数），置信度（0-1之间浮点数）要记住这些词只是在词典中的编码，因此客户端必须执行转换操作。online-audio-client 在生成webvtt文件时不进行任何字符转换，因此你需要用iconv将结果文件转换成UTF8

服务器返回的结果的例子如下：
```
RESULT:NUM=3,FORMAT=WSEC,RECO_DUR=1.7,INPUT_DUR=3.22
one,0.4,1.2,1
two,1.4,1.9,1
three,2.2,3.4,0.4
RESULT:DONE
```

使用举例：

命令行启动服务器：
```
online-audio-server-decode-faster --verbose=1 --rt-min=0.5 --rt-max=3.0 --max-active=6000 --beam=72.0 --acoustic-scale=0.0769 final.mdl graph/HCLG.fst graph/words.txt '1:2:3:4:5' 5010 graph/word_boundary_phones.int final.mat
```

各参数含义如下：
final.mdl :声音模型文件
HCLG.fst :完全的fst
words.txt:发音词典（将词的id号映射到对应的字符上)
'1:2:3:4:5' ：sil 的id号
5010 ：服务器的端口号
word_boundary_phones.int ：词对齐时使用的音素边界信息
final.mat ：特征的LDA矩阵

命令行启动客户端

```
online-audio-client --htk --vtt localhost 5010 scp:test.scp
```

各参数含义如下

?–htk 结果存成htk 标注文件

?–vtt 结果存成webvtt文件

?localhost 服务器地址

?5010 服务器端口号

?scp:test.scp Wav文件的列表文件

命令行方式启动java客户端

```
java -jar online-audio-client.jar
```

或者直接在图形界面上双点jar文件

命令行方式运行识别wav文件

```
online-wav-gmm-decode-faster --verbose=1 --rt-min=0.8 --rt-max=0.85\
                --max-active=4000 --beam=12.0 --acoustic-scale=0.0769 \
                scp:$decode_dir/input.scp $ac_model/model $ac_model/HCLG.fst
\
                $ac_model/words.txt '1:2:3:4:5' ark,t:$decode_dir/trans.txt \
                ark,t:$decode_dir/ali.txt $trans_matrix;;
```

各参数含义如下：

GStreamer 插件

Kaldi 工具集为GStream 多媒体流框架提供了一个插件（1.0或兼容版)。此插件可以做为过滤器，接收原始音频做为输入并将识别结果做为输出。这个插件的主要好处是他使得kaldi 在线语音识别功能对所有支持GStreamer1.0的编程语言都可用(包括python,ruby,java,vala等等)。这也使得可以把kaldi 在线解码器集成到支持GStreamer 通信标准的应用程序中

安装

GStreamer 插件的源码位于src/gst-plugin 目录。要完成编译，kaldi工具集的其它部分必须使用共享库进行编译。因此在配置的时候必须使用--shared 参数，同进需要编译在线扩展（make ext).确保支撑GStreamer 1.0开发头文件的包已经安装在你的系统中。在Debian Jessie系统版本中，需要包libgstreamer1.0-dev。在Debian Wheezy版本中，GStreamer 1.0 可以从backports源中下载到。需要安装gstreamer1.0-plugins-good和gstreamer1.0-tools这两个包。演示程序还需要PulseAudio Gstreamer插件 ，包名：gstreamer1.0-pulseaudio

最后，在src/gst-plugin目录中运行make depend和make。这样就会生成一个文件src/gst-plugin/libgstkaldi.so，他包含了GStreamer 插件。为确保GStreamer 可以找到kaldi 插件，必须把src/gst-plugin目录加到他的插件搜索目录。因此需要把这个目录加入环境变量中

export GST_PLUGIN_PATH=$KALDI_ROOT/src/gst-plugin

当然，你需要把$KALDI_ROOT修改成你自己文件系统中kaldi所在 的根目录。

4.9 决策树是如何在 kaldi 中使用

How decision trees are used in Kaldi

介绍

决策树在 Kaldi 中如何使用

介绍（Introduction）

本页将对声学决策树在 kaldi 中如何被创建和使用，以及如何在训练和解码图构建过程进行运用给出一个概述性的解释。对于构建决策树代码的内部描述，请参见 Decision tree internals；对于构建解码图方法的详细信息，可以参见 Decoding graph construction in Kaldi。

实现的基本算法就是自顶向下的贪婪分裂，通过问一些问题，比如说左边的音素，右边的音素，中心音素以及当前的状态等等，我们会得到很多可以把数据进行分裂的路径。我们实现的算法与标准算法非常相似，请参见 Young，Odell 和 Woodland 的这篇论文”Tree-based State Tying for High Accuracy Acoustic Modeling” 。假设我们对数据建模时采用单高斯将它们分成两部分，在这个算法

中，我们通过选择局部最优的问题进行数据分裂，也就是使得似然值增加最大的那个问题。与标准算法实现不同的地方包括可以自由配置树的根节点；对 HMM 状态和中心音素相关问题提问的能力；以及实际上在 Kaldi 脚本中默认情况下，问题集是通过对数据自顶向下的二分聚类自动生成的，这就意味着不需要手动去创建问题集。关于树的根节点的配置：可能是把一个共享的群组里面所有音素分裂的统计量，或者独立的音素，或者每个音素的 HMM 状态，作为树的根节点来进行分裂，或者把音素组作为树的根节点（注：多个音素共享一棵树的根节点）。对于如何用标准的脚本配置根节点，请参见 Data preparation。实际上，我们一般让每棵树的根节点都对应一个真实的音素（real phone），意思就是说我们把每个音素的词位置相关、发音相关或者音调相关的所有变种都放进一个音素组，作为决策树的根节点。

本页下面主要给出相关代码层面的一些详细信息。

音素上下文窗（Phonetic context windows）

这里我们解释一下在代码中我们怎样描述一个音素的上下文。一棵特殊的决策树将有两个整型值，分别描述的是上下文窗的宽度和中心位置。下表简单说明了这两个值：

Name in code	Name in command-line arguments	Value (triphone)	Value (monophone)
N	--context-width=?	3	1
P	--central-position=?	1	0

N 是上下文窗的宽度，P 是设计的中心音素的标记。一般 P 就是窗的中心（因此叫中心位置）；举例说，当 N=3 时我们一般设 P=1，但是我们也可以从 0 到 N-1 自由选择；比如，P=2 和 N=3 意味着有左上下文有两个音素，并且没有右上下文。在代码中，当我们讨论中心音素时，我们总是认为讨论的是第 P 个音素，可能是也可能不是上下文窗中心的那个音素。

一个用来表示典型的 triphone 上下文窗的整型向量可能是：

```
//probably not valid C++vector<int32> ctx_window = { 12, 15, 21 };
```

假设 N=3 和 P=1，这个表示有一个右上下文 21 和一个左上下文 12 的音素 15。我们处理端点位置上下文的一个方式是使用 0（0 不是一个合法的音素，因为在 OpenFst 中 0 是为空符号 epsilon 而保留的），所以比如：

```
vector<int32> ctx_window = { 12, 15, 0};
```

表示有一个左上下文 12 和没有右上下文的音素 15，因为音素 15 是句子的结尾。在句子结尾这种特殊的地方，0 这种方式的使用可能有一点意外，因为最后一个“音素”实际上是后续符号“$”（参见 Making the context transducer），但是为了在决策树代码中的便利，我们不把后续符号放进上下文窗，而是把 0 放进去。注意，如果此时我们 N=3 和 P=2，那上述的上下文窗是非法的，因为第 P 个元素是一个不能表示任何真实音素的 0；当然同样如果我们有一个 N=1 的树，上面

的窗都是不合法的，因为那些窗的大小都是错误的。在单音素的情况下，我们可以有一个如下的窗：

```
vector<int32> ctx_window = { 15 };
```

因此单音素系统只是被当成上下文相关系统的一种特殊情况，窗的大小 N 等于 1，并且还有一棵什么都不做的树（注：经过这棵树后没有任何参数被绑定）。

树的构建过程（The tree building process）

在这部分我们给出 Kaldi 中树构建过程的一个概述。

即使是单音素系统也有一个决策树，但是是一个无用的树。参见返回这样一个无用树的函数 MonophoneContextDependency() 和 MonophoneContextDependencyShared()。这两个函数被命令行程序 gmm-init-mono 调用；它主要的输入参数是 HmmTopology 对象，并且输出一棵树，这棵树通常会被以 ContextDependency 类型的对象写到一个叫做“tree”的文件中，以及模型文件（模型文件包含一个 TransitionModel 对象和一个 AmDiagGmm 对象）。如果程序 gmm-init-mono 接受一个叫-shared-phones 的可选参数，它将会在指定的音素序列间共享 pdfs（注：输出概率密度函数，比如高斯），否则它会使得所有的音素都是独立的。

从一个扁平的初始（注：所有的高斯都是用全局均值和方差初始化的）开始训练一个单音素系统后，我们拿单音素对齐的结果和使用函数 AccumulateTreeStats()（被 acc-tree-stats 调用）来累积训练决策树的统计量。这个程序不限于读取单音素的对齐结果；它也能读取上下文相关的对齐结果，因此我们也可以基于 triphone 对齐结果来构建树。构建树的统计量以 BuildTreeStatsType 类型（参见 Statistics for building the tree）被写到磁盘。函数 AccumulateTreeStats() 输入 N 和 P 的值，N 和 P 就是上文解释过的上下文窗的大小和中心音素位置。命令行程序会默认地将 N 和 P 设为 3 和 1，但是也可以使用–context-width 和 –central-position 可选参数进行覆盖。程序 acc-tree-stats 输入一个上下文无关的音素列表（比如，silence），但是即使存在上下文无关的音素，这个也不是必需的；它只是减少统计量大小的一个机制。对于上下文无关的音素，程序将会累积一个没有定义 keys 的相关的统计量，keys 是跟左右音素对应的（注：在代码中会把一个音素不同的上下文和 pdf-class 分别作为不同的 key，然后累积每个 key 的统计量）（c.f. Event maps）。

当统计量被积累后，我们使用程序 build-tree 来构建树。这个程序输出一棵树。程序 build-tree 需要三样东西：

- 统计量（BuildTreeStatsType 类型）
- 问题集配置（Questions 类型）
- roots 文件（参见下面）

统计量一般从程序 acc-tree-stats 得到；问题集配置类可以用程序 compile-questions 输出，compile-questions 输入一个声学问题集的拓扑列表（在我们的脚本中，这些都是自动地从构建树的统计量通过程序 cluster-phones 得到）（注：cluster-phones 输入构建树的统计量可以得到一个声学问题集）。roots 文件指定了将要在决策树聚类过程中共享根节点的音素集，并且对每个音素集指出下面两个东西：

- “shared”或者“not-shared”指出是每个 pdf-class（也就是一般情况下的 HMM 状态）都有不同的根节点，还是所有 pdf-class 共享一个根节点。如果是“shared”，对于所有的 HMM 状态（比如在正常的 HMM 拓扑下所有的三个状态）将只会有一个树根节点；如果是“not-shared”，将会有三个树根节点，每个 pdf-class 有一个。
- “split”或者“not-split”指出对于根节点要不要根据问题进行决策树分裂（对于 silence，我们一般不分裂）。如果该行指定“split”（正常情况），那么我们进行决策树分裂。如果指定“not-split”，那么就不会进行分裂，因此根节点就被无分裂地保留。

下面将对这个怎样使用方面做一些阐述：

- 如果我们指定“shared split”，即使所有的三个 HMM 状态有一个根节点，不同的 HMM 状态仍然可以到达不同的叶子节点，因为树可以像对声学上下文的问题提问一样对 pdf-class 的问题进行提问。
- 对于 roots 文件中同一行出现的所有音素，我们总是让它们共享根节点。如果你不想共享音素的根节点，你只要把它们放在不同的行。

下面是 roots 文件的一个例子；假设音素 1 是 silence，并且其他的音素都有不同的根节点。

```
not-shared not-split 1 shared split 2... shared split 28
```

当我们有比如位置和声调相关的音素时，将多个音素放在同一行会非常有用；这样每个“真实的“音素将关联到一个整数的音素 ID 集合。在这种情况下我们将 particular underlying（注：这个不知道怎么翻译）音素的所有变种共享一个根节点。下面是来自 egs/wsj/s5 脚本中 Wall Street Journal 的 roots 文件的一个例子（这个例子中音素是用文本表示的，而不是整数形式；但在被 Kaldi 读取之前会被转换成整数形式（注：就是会把音素映射成整数的 ID））：

```
not-shared not-split SIL SIL_B SIL_E SIL_I SIL_S SPN SPN_B SPN_E S
PN_I SPN_S NSN NSN_B NSN_E NSN_I NSN_S shared split AA_B AA_E
AA_I AA_S AA0_B AA0_E AA0_I AA0_S AA1_B AA1_E AA1_I AA1_S A
A2_B AA2_E AA2_I AA2_S shared split AE_B AE_E AE_I AE_S AE0_B A
E0_E AE0_I AE0_S AE1_B AE1_E AE1_I AE1_S AE2_B AE2_E AE2_I AE
2_S shared split AH_B AH_E AH_I AH_S AH0_B AH0_E AH0_I AH0_S A
H1_B AH1_E AH1_I AH1_S AH2_B AH2_E AH2_I AH2_S shared split AO_
B AO_E AO_I AO_S AO0_B AO0_E AO0_I AO0_S AO1_B AO1_E AO1_I
```

```
AO1_S AO2_B AO2_E AO2_I AO2_S shared split AW_B AW_E AW_I AW_S AW0_B AW0_E AW0_I AW0_S AW1_B AW1_E AW1_I AW1_S AW2_B AW2_E AW2_I AW2_S shared split AY_B AY_E AY_I AY_S AY0_B AY0_E AY0_I AY0_S AY1_B AY1_E AY1_I AY1_S AY2_B AY2_E AY2_I AY2_S shared split B_B B_E B_I B_S
                    \shared split CH_B CH_E CH_I CH_S shared split D_B D_E D_I D_S
```

当创建这个 roots 文件时，你应该确保在每一行至少有一个音素是可见的（注：有对应的训练样本）。比如上面的情况，如果音素 AY 至少在声调和词位置的某些连接中可见，那就没问题。

在这个例子中，对于 slience 等音素我们有很多的词位置相关的变种。它们将共享它们的 pdf's，因为它们都在同一行，并且是"not-split"，但是它们可能会有不同的状态转移参数。实际上，silence 的大多数变种都不可能用到，因为 silence 不可能出现在词与词之间；这只是为了防止以后有人做一些奇怪的事而不会过时。

我们用从之前创建的模型（比如，单音素模型）得到的对齐结果来对混合高斯参数进行初始化；对齐的结果会被程序 convert-ali 从一棵树转换到另一棵（注：应该就是说对齐的 transition 不变，但状态绑定的参数可能因为决策树的不同而变化）。

PDF 标号（PDF identifiers）

PDF 标号（pdf-id）是一个从 0 开始的数字，用做概率密度函数（p.d.f.）的序号。系统中每一个 p.d.f.都有自己的 pdf-id，并且是连续的（在一个 LVCSR 系统中一般会有几千个）。在树首先被构建时，它们就会被赋值。对于每一个 pdf-id 对应的是哪个音素，可能知道也可能不知道，这取决于树是怎样被构建的。

上下文相关对象（Context dependency objects）

ContextDependencyInterface 对象是树的一个虚基类，指定了如何与构建解码图代码进行交互。这个接口只包含四个函数：

- ContextWidth()返回树需要的 N（上下文窗的大小）的值。
- CentralPosition()返回树需要的 P（窗中心位置）的值
- NumPdfs()返回树定义的 pdfs 的数量；pdfs 的编号从 0 到 NumPdfs()-1。
- Compute()是对某个特殊的上下文计算它对应的 pdf-id 的函数

ContextDependencyInterface::Compute()函数的声明如下：

```
class ContextDependencyInterface { ... virtualbool Compute(conststd::vector<int32>&phoneseq, int32 pdf_class, int32 *pdf_id)const;}
```

如果能计算得到上下文和 pdf-class 对应的 pdf-id，函数返回 true。返回 false 时表明出现了一些错误或者是不匹配。使用这个函数的一个例子：

```
ContextDependencyInterface *ctx_dep = ... ; vector<int32> ctx_window = { 12, 15, 21 }; // not valid C++ \int32 pdf_class = 1; // probably central state of 3-state HMM.int32 pdf_id; if(!ctx_dep-> Compute(ctx_window, pdf_class, &pdf_id)) KALDI_ERR <<"Something went wrong!"elseKALDI_LOG <<"Got pdf-id, it is "<< pdf_id;
```

目前唯一继承 ContextDependencyInterface 的类就是 ContextDependency，ContextDependency 有少量更丰富的接口；唯一主要的添加就是函数 GetPdfInfo，被用于 TransitionModel 类算出一个特殊的 pdf 可能对应哪些音素（这个函数的功能可以被 ContextDependencyInterface 接口遍历所有的上下文而实现）。

ContextDependency 对象实际上是对 EventMap 对象的简单组合封装；请参见 Decision tree internals。我们希望尽可能地隐藏树的真正实现，使得以后需要重构代码时变得非常简单。

决策树的一个例子（An example of a decision tree）

决策树文件的格式不是以人们的可读性为首要目标而创建的，但由于大家需要我们在这里试着解释如何去解读这个文件。请看下面的例子，这个是一个来自 Wall Street Journal 脚本中 triphone 的决策树。它以这个对象的名字 ContextDependency 开始（注：在代码中整个树是一个 ContextDependency 对象）；然后是 N（上下文窗的大小），这里是 3；接着是 P（上下文窗的中心位置），这里是 1。文件剩下的部分包含单个 EventMap 对象。EventMap 是一个可能包含指向其他 EventMap 指针的多态类型。更多详细信息，请参见 Event maps。这个文件表示一棵决策树或多棵决策树的集合，并将一个键值对集合（比如，left-phone=5, central-phone=10, right-phone=11, pdf-class=2（注：注意这里是四个键值对，表示一个中心音素是 10，上文是音素 5，下文是音素 11 的 triphone 的第 2 个状态））映射到一个 pdf-id（比如，158）。简单来说，一个决策树包含三种基本类型：一个是 SplitEventMap（就像决策树中的分支判断），一个是 ConstantEventMap（就像决策树的叶子节点，只包含一个表示 pdf-id 的数字），和一个是 TableEventMap（就像是一个包含其他 EventMaps 的一个查找表）。SplitEventMap 和 TableEventMap 都有一个需要它们判断的 key，这个值可能是 0，1 或者 2，分别表示左上下文音素，中心音素和右上下文音素，也可能是-1，表示 pdf-class 的标号（注：如果 HMM 的每个状态都有对应的 pdf，则 pdf-class 可理解为 HMM 的第几个状态）。一般情况，pdf-class 的值与 HMM 状态的序号是相同的，比如 0，1 或 2。请尝试不要因此而感到困惑：key 是-1，value 是 0，1 或 2，但它们与上下文窗中音素的 keys 0，1 或 2 是没有任何关系的（注：上下文窗中 0，1 和 2 表示的是窗中音素的位置）。SplitEventMap 有一系列值可以触发决策树的 yes 分支。下面是一种 quasi-BNF 符号表示的决策树文件格式。

```
EventMap := ConstantEventMap | SplitEventMap | TableEventMap | "NULL "ConstantEventMap := "CE"<numeric pdf-id>SplitEventMap := "SE"<key-to-split
```

```
-on>"[" yes-value-list "]""{" EventMap EventMap "}"TableEventMap := "TE"<key-to-split-on><table-size>"(" EventMapList ")"
```

在下面的例子中，树顶层的 EventMap 是一个以 key 1 进行分裂的 SplitEventMap，也就是按中心音素分裂。在方括号中是一系列连续范围的 phone-ids。然而，这些并不表示一个问题，它们只是音素分裂的一种方法，因此我们可以得到每个音素真正的决策树（注：音素真正的决策树是根据音素上下文和 pdf-class 进行决策的，对中心音素的决策只是为了找到这个音素对应的真正的决策树）。问题在于这棵树是通过“shared roots”方式创建的，所以有很多与同一音素不同词位置和音调标识相关的 phone-ids，它们都共享树的根节点。在这种情况下在树的顶层我们不能使用 TableEventMap，否则我们就不得不将每棵树重复好几遍（因为 EventMap 是一棵纯树，而不是一个通用的图，它没有指针共享的机制）。文件后面的一些“SE”标签也是 quasi-tree 的一部分，它们都是首先按中心音素进行分裂（当我们顺着文件往下看时我们进入了树的更深处；注意这个花括号“{”一直是打开的，还没有关闭）。然后我们看到字符串“TE -1 5 (CE 0 CE 1 CE 2 CE 3 CE 4) ”，表示通过 TableEventMap 对 pdf-class -1 进行分裂（实际上就是，HMM-position），并且返回从 0 到 4 的值。这 5 个值表示的是静音和噪声音素 SIL，NSN 和 SPN 的 5 个 pdf-ids。在我们的设定中，这三个非语音音素的 pdfs 是共享的（只有转移矩阵是不同的）。注意：对于这些音素我们用 5 状态而不是 3 状态的 HMM，所以这里有 5 个不同的 pdf-ids。接下来是“SE -1 [0] ”，这可以被认为是这棵树中第一个真正的问题。我们可以从上面的 SE 问题看出这个问题被应用于中心音素为 4 到 19 时候，也就是音素 AA 的不同版本（注：原文写的是 5 到 19，不过我认为原文有问题，改成了 4 到 19）。这个问题问的是 pdf-class（key -1）是不是 0（即是不是最左边的 HMM-state）。下一个问题是“SE 2 [220 221 222 223]”，问的是音素右上下文是不是音素“M”不同形式中的一个（这是一个非常有效的问题，因为我们是在最左边的 HMM-state）；如果问题的答案是 yes，我们继续问“SE 0 [104 105 106 107… 286 287]”，这是一个关于音素左上下文的问题（注：原文写的是右上下文，但应该是左上下文）；如果答案是 yes，则 pdf-id 就是 5（“CE 5”），否则就是 696（“CE 696”）。

```
s3# copy-tree --binary=false exp/tri1/tree - 2>/dev/null | head -100

ContextDependency 3 1 ToPdf SE 1 [ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 ] { SE 1 [ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 4
```

```
1 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 6
4 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 8
7 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107
 108 109 110 111 ] { SE 1 [ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
 64 65 66 67 ] { SE 1 [ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 ] { SE 1 [ 1 2 3 ] { TE -1 5 ( CE 0 CE 1 CE 2 CE 3 CE 4 ) SE -1
[ 0 ] { SE 2 [ 220 221 222 223 ] { SE 0 [ 104 105 106 107 112 113 114
115 172 173 174 175 208 209 210 211 212 213 214 215 264 265 266 267 2
80 281 282 283 284 285 286 287 ] { CE 5 CE 696 }SE 2 [ 4 5 6 7 8 9 1
0 11 12 13 14 15 16 17 18 19 36 37 38 39 40 41 42 43 44 45 46 47 48 4
9 50 51 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 1
47 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 26
8 269 270 271 288 289 290 291 292 293 294 295 296 297 298 299 300 301
 302 303 ]
```

下面是一个更简单的例子：来自 Resource Management 脚本的单音素决策树。顶层的 EventMap 是一个 TableEventMap（“TE 0 49 …”）。key 0 是音素位置 0，表示中心（并且只有这一个）音素，因为上下文窗大小（N）为 1。TE 的条目数量是 49（音素的数量加 1）。表中第一个 EventMap 是 NULL，因为没有序号为 0 的音素。下一个 EventMap 是一个有三个元素的 TableEventMap，关联到第一个音素的三个 HMM 状态（技术上来说，是 pdf-class）：“TE -1 3 (CE 0 CE 1 CE 2)”。

```
s3# copy-tree --binary=falseexp/mono/tree - 2>/dev/null| head -5ContextDepe
ndency 10 ToPdf TE 049 ( NULL TE -13 ( CE 0 CE 1 CE 2 ) TE -13 ( C
E 3 CE 4 CE 5 ) TE -13 ( CE 6 CE 7 CE 8 ) TE -13 ( CE 9 CE 10 CE
11 ) TE -13 ( CE 12 CE 13 CE 14 )
```

输入符号信息对象（The ilabel_info object）

CLG 图（请参见 Decoding graph construction in Kaldi）在它的输入符号位置上有表示上下文相关音素的符号（辅助符号和可能的空符号也一样）。在图中它们总是用整型的标签来表示。在代码和文件名中，我们使用一个叫做 ilable_info 的对象。ilable_info 对象跟 ContextFst 对象有很密切的联系，请参见 see The ContextFst object。就跟许多其他的 Kaldi 类型一样，ilabel_info 也是一个通用的（STL）类型，但是为了可以辨别出是 ilabel_info，我们使用与之相同的变量名。就是下面定义的类型：

```
std::vector<std::vector<int32>>  ilabel_info;
```

它是一个以 FST 输入标签为索引的 vector，给每一个输入标签一个对应的音素上下文窗（参见上文，Phonetic context windows）。比如，假设符号 1500 是左上下文是 12 和右上下文是 4 的音素 30，我们有：

```
// not valid C++ ilabel_info[1500] == { 4, 30, 12 };
```

在单音素的情况下，我们就会像这样：

```
ilabel_info[30] == { 28 }
```

处理辅助符号会有点特殊（参见 Disambiguation symbols 或者上面引用的 Springer Handbook 文献，该文献解释了这些辅助符号是什么）。如果一条 ilabel_info 记录对应到一个辅助符号，我们就把辅助符号的符号表序号取负值放进去（注意这跟辅助符号打印形式#0，#1，#2 等等里面的数字是不一样的，它是跟这些辅助符号在符号表文件中的顺序相关的数字，这个符号表文件在我们现在的脚本中叫做 phones_disambig.txt）。比如，

```
ilabel_info[5] == { -42 }
```

意味着在 HCLG 中符号数 5 对应到整数 id 是 42 的辅助符号。为了编程方便我们对这些 id 取负号，因此解析 ilable_info 对象的程序不需要给一个辅助符号的列表就可以在单音素情况下将它们跟真实的音素进行区分。有两个额外特殊情况：

```
ilabel_info[0] == { }; // epsilon\ilabel_info[1] == { 0 }; // disambig symbo
l #-1;// we use symbol 1, but don't consider this hardwired.
```

第一个是正常的空符号，我们给它一个空的 vector 作为它的 ilabel_info。这个符号一般不会出现在 CLG 的左边（注：应该是说不会作为 CLG 的输入符号）。第二个是一个特殊的辅助符号，打印形式叫做“#-1”。在 epsilons 被用做标准（Springer Handbook）脚本中 C 转换器输入符号的时候，我们使用辅助符号“#-1”。它可以确保有空音素表示的词的 CLG 网络可以被确定化。

程序 fstmakecontextsyms 可以创建一个与 ilabel_info 对象打印形式对应的符号表；这个主要用于调试和诊断错误。

4.10 Decision tree internals

Decision tree internals

这页将描述音素决策树代码的内部机制，这个是用通用的数据结构和算法来实现的。

从更高层的解释整个算法是如何实现的和怎么在 kaldi 使用的，可以看 How decision trees are used in Kaldi。

Event maps

决策树建立代码的核心概念就是 event map,用类 EventMap 来描述。不要被事件这个词误认为这是发生在特定事件的。event 仅仅代表一个(key,value)对的一个集合，要求没有 key 是重复的。从概念上讲，它是可以用类型 std::map<int32,int32>来描述。事实上，为了更加有效的使用，我们通过 typedef，把它表示为 a sorted vector of pairs。

```
Typedef std::vector<std::pair<EventKeyType,EventValueType> > EventType;
```

这里，EventKeyType 和 EventValueType 仅仅 typedefs to int32，但是如果我们可以从他们的名字里区分，这个代码是非常容易理解的。想象一下一个 Event (type EventType)当做一个向量的集合，而且变量的名字和值都为整数。也有一个类型 EventAnswerType，和他们一样也是 int32，它是由 EventMap 返回的类型；

事实上它是一个 pdf identifier (声学状态索引)。需要 EventType 的函数期望你可以首先对它进行排序(e.g. by calling std::sort(e.begin(),e.end());)

EventMap 类的用意可以通过一个例子很好的阐述。假设我们的上下文音素是 a/b/c；假设我们有一个标准的三音素拓扑结构；和假设在这个上下文中我们想问音素 "b"的中心状态的 pdf 对应的索引是多少。这个问题里这个状态将是状态1，因为我们是用从0开始的索引。当我们说到"state"；我们将跳过一些细节；更多的细节你可以看 Pdf-classes。假设音素 a, b and c 的整数索引分别为10, 11 and 12 。"event"对应的映射为：

```
(0  -> 10), (1  -> 11), (2  -> 12), (-1 -> 1)
```

在3音素窗"a/b/c"的0, 1 and 2 是位置，和-1是我们用来编码这个 state-id(c.f. 常量 kPdfClass = -1)的一个特殊的索引。用这种方式的话，我们的一个排好序的向量对是：

```
// caution: not valid C++.
EventType e = { {-1,1}, {0,10}, {1,11}, {2,12} };
```

假设相对应的声学状态的这个声学状态索引(pdf-id) 是1000。然后如果我们有一个表示 tree 的 EventMap "emap"，然后我们期望接下来的设置不会失效：

```
EventAnswerTypeans;
boolret = emap.Map(e, &ans); // emap is of type EventMap; e is EventType
KALDI_ASSERT(ret == true && ans == 1000);
```

因此当我们指出 EventMap 是从 EventType 映射到 EventAnswerType，你可以认为这大概是一个从上下文音素到整数索引的一个映射。音素上下文是由(key-value)对的集合表示，和原则上我们可以添加新的 keys 和投入更多的信息。注意函数 EventMap::Map()返回 bool。这是因为它可能对于某些 events 来说，不映射到其他任何的答案(比如：考虑一个无效的音素，或者想象一下，当 EventType 不包含所有的 EventMap 查询的 keys)。

EventMap 是一个很被动的对象。它没有任何学习决策树的能力，它仅仅就是存储决策树。可以把它看成表示一个从 EventType 到 EventAnswerType 的函数的一个结构。EventMap 是一个多态，纯虚类(i.e. 它不能被实例化，因为它具有不实施虚拟功能)。实现 EventMap 接口需要三个具体的类：

- ConstantEventMap: 认为它是一个决策树叶子结点。这个类存储类型 EventAnswerType 的一个整数和它的映射函数常常返回这个值。
- SplitEventMap: 认为它是一个决策树的非叶子结点，它查询一个特定的 key，和根据问题的答案去"yes" or "no" 孩子结点。它的映射函数调用合适的孩子结点的映射函数。它存储对应"yes"孩子结点的类型 kAnswerType 的整数集(everything else goes to "no").
- TableEventMap: 这个在一个特定的 key 上做一个完整的分裂。一个典型的例子就是：你也许首先完全分裂中心音素，然后你对这个音素的每一值有一个分离的决策树。内部地它存储 EventMap* 指针的一个向量。它查找对应分裂的 key 的值，和调用向量对应位置的 EventMap 的映射函数。

除了从 EventType 映射到 EventAnswerType.，EventMap 的确不能够做很多事情。它的接口不能提供允许你遍历树的函数，无论是向上还是向下。这里仅

仅有一个函数允许你修改 EventMap，和这个就是 EventMap::Copy()函数，这样的声明(作为一个类成员)：

```
virtualEventMap *Copy(const std::vector<EventMap*> &new_leaves) const;
```

这个与函数组合有一样的功能。如果你用一个空的向量"new_leaves"调用 Copy()，它将仅仅返回整个对象的一个深度拷贝，拷贝它沿着树往下的所有的指针。然而，如果 new_leaves 不为空，每次函数 Copy()到达一个叶子，如果这个叶子1是在(0, new_leaves.size()-1)范围内的和 new_leaves[l]不为空， 函数 Copy()将返回调用 new_leaves[l]->Copy()的结果，而不是 Copy()函数的叶子本身(将返回一个新的 ConstantEventMap)。一个典型的例子就是当你决定对一个特定的叶子来分裂，比如叶子852。你将创建一个 vector<EventMap*>类型的一个对象，它的非空数字是在位置852那里。它包含一个指针指向一个真正类型为 SplitEventMap 的对象，和"yes"和 "no" 指针是有叶子值为852和1234的 ConstantEventMap 类型。(我们为新的叶子重新私用旧的叶子结点)。真正的树建立代码不是这么低效的。

Statistics for building the tree

用来建立音素决策树的统计量如下：

```
typedefstd::vector<std::pair<EventType, Clusterable*> > BuildTreeStatsType;
```

这种类型的一个对象的参考被传递到所有决策树建立过程中。这些统计量被期望包含相同的 EventType 成员的 no duplicates，比如它们从概念上表示一个从 EventType 到 Clusterable 的映射。 在我们目前的代码中 Clusterable 对象是一个真正的类型 GaussClusterable，但是树建立代码不知道这个，积累这些统计量的程序是 acc-tree-stats.cc。

Classes and functions involved in tree-building

Questions (config class)

类 Questions 是一个与树建立代码相交互的类，行为有点像类"configuration"。它是一个从"key" 值(类型 EventKeyType)到一个类型 QuestionsForKey 的 configuration 类的真正映射。

QuestionsForKey 类有一个"questions"集，每一个是类型 EventValueType 的整数集他们大对数对应于音素集，或者如果 key 是-1，他们就是对应 HMM 状态索引集的一个典型例子，比如{0,1,2}的子集。QuestionsForKey 包含一个类型 RefineClustersOptions 的一个 configuration 类。它在树建立代码中控制树建立的行为将试图在分裂的两边迭代的移动这些值(e.g. phones)来最大化似然比(as in K-means with K=2)。然而，这些可以通过设定"refining clusters"的迭代数目为0来关闭，相当于仅仅从一个固定列表的问题集中选择。这看起来效果要好点。

Lowest-level functions

一个完整的列表，可以看 Low-level functions for manipulating statistics and event-maps；我们这里将总结一些重要的。

这些函数大多是涉及到类型 BuildTreeStatsType 的一些操作，作为我们下面看到的一个(EventType,Clusterable*)对的一个向量。最简单的一个是 DeleteBuildTreeStats(), WriteBuildTreeStats()和 ReadBuildTreeStats().

函数 PossibleValues()寻找在一个统计量集中一个特定的 key 对应的值(和通知用户这个值是否经常被定义)；SplitStatsByKey()将通过一个特定的 key 对应的值来分裂类型 BuildTreeStatsType 的一个对象为一个向量 <BuildTreeStatsType>(比如：在中心音素上分裂)；SplitStatsByMap()做同样的事情，但是索引不是这个 key 的值，但是由 EventMap 返回的答案被提供给这个函数。SumStats()在一个 BuildTreeStatsType 对象里求统计量(i.e 对象 Clusterable)的和返回对应的 Clusterable*对象。SumStatsVec()采用一个类型 vector<BuildTreeStatsType>的对象和输出为 vector<Clusterable*>类型的量，比如它像 SumStats()但是是一个向量；处理 SplitStatsByKey() andSplitStatsByMap()的输出是非常有用的。

给定一些统计量和 EventMap，ObjfGivenMap()评估这个目标函数：它在每一个聚类中对所有的统计量求和(相对应 EventMap::Map()函数的每一个不同的答案)，通过这些聚类来添加目标函数，和返回整个。

FindAllKeys() 将找到所有定义在统计量集合里的值，和根据不同的参数，它可以找到被定义为所有的 event 的所有 keys，或者为其他一些 event 定义的所有的 keys (比如采用定义的 keys 的交集或者并集)。

Intermediate-level functions

涉及到树建立的下一组函数被列在 here 和对应树建立的不同阶段。我们现在将提一些有代表的。

首先，我们指出许多这些函数有一个参数 int32 *num_leaves。这个指向的整数作为一个计数器来分配新的叶子。在树建立的开始阶段，这个 caller 被设定为0。每次一个新节点被需要时，树建立的代码采用当前的数字来指向作为新的 leaf-id，和然后增加它。

一个重要的函数是 GetStubMap()。这个函数返回一个没有分裂的树，比如 pdfs 不根据上下文。这个函数的输入控制所有的音素要不是不同的要不他们的一些是共享决策树的根节点，和或者在一个特定的音素里所有的状态共享相同的决策树结点。

SplitDecisionTree()函数采用输入一个 EventMap，对应有 GetStubMap()创建的类型的一个分裂的"stub"决策树，和做决策树分裂，知道无论是叶子的最大数目达到了，还是分裂一个叶子的增益比特定的门限小。

函数 ClusterEventMap()采用一个 EventMap 和一个门限，和只要这样做的成本小于阈值，合并 EventMap 的叶子。在 SplitDecisionTree()后，这个函数正常

的被调用。这里有一些操作限制的函数的不同版本 (e.g.从不同的音素里去避免合并叶子)。

Top-level tree-building functions

高层次树建立的函数被列在 here。这些函数可以从命令行里直接调用。最重要的一个就是 BuildTree()。它被赋予了一个 Questionsconfiguration 类，和音素集的一些信息，这些音素他们的根节点是共享的和(对于音素的每一个集合)无论是在单决策树的根节点共享还是对每一个 pdf-class 有一个分离的根节点。它去掉关于音素的长度的信息，加上不同的门限。它建立树和返回一个用来构建 ContextDependency 对象 EventMap 对象。

在这组里有一个重要的事是 AutomaticallyObtainQuestions()，它用来获得通过自动对音素聚类的问题。它把音素聚类成树，和对树的每一个节点，从这个节点的所有叶子组成一个问题(问题等价于音素集)。这个(从 cluster-phones.cc 调用) 使我们的脚本独立于人产生的音素集。

给定特征和 transition-ids 的一个序列的对齐，函数 AccumulateTreeStats()为训练树来积累统计量(看 Integer identifiers used by TransitionModel)。这个函数被定义在一个与树建立相关函数的不同的目录(hmm/ not tree/) ，它依赖于更多的代码(e.g.it knows about the theTransitionModel class),和我们更愿意保持对核心的树建立代码的依赖性更小。

4.11 HMM 拓扑结构和转移模型

HMM topology and transition modeling

介绍

在这里我们将介绍在 kaldi 用如何表示 HMM topologies 和我们如何让建模和训练 HMM 转移概率的。我们将简要的说下它是如何跟决策树联系的；决策树你可以在 How decision trees are used in Kaldi 和 Decision tree internals 这些地方看到更详细的； 对于这个里面的类和函数，你可以看 Classes and functions related to HMM topology and transition modeling。

HMM topologies

HmmTopology 类是为用户指定到 HMM 音素的拓扑结构。在平常的样例里，脚本创建一个 HmmTopology 对象的文本格式的文件，然后给命令行使用。为了说明这个对象包含什么，接下来的就是一个 HmmTopology 对象的文本格式的文件(the 3-state Bakis model)：

```
 <Topology>
<TopologyEntry>
```

```
  <ForPhones> 1 2 3 4 5 6 7 8 </ForPhones>
  <State> 0 <PdfClass> 0
<Transition> 0 0.5
<Transition> 1 0.5
</State>
  <State> 1 <PdfClass> 1
<Transition> 1 0.5
<Transition> 2 0.5
  </State>
<State> 2 <PdfClass> 2
  <Transition> 2 0.5
<Transition> 3 0.5
  </State>
<State> 3
  </State>
</TopologyEntry>
  </Topology>
```

这里在这个特定的 HmmTopology 对象里有一个 TopologyEntry，它覆盖音素1到8(所以在这个例子里有8个音素和他们共享相同的 topology)。他们有3个发射态；每一个有个自环和一个转移到下一个状态的。这里也有一个第四个非发射态，状态3 (对于它没有<PdfClass> entry) 没有任何的转移。这是一个这些 topology entries 的标准特征；Kaldi 把第一个状态(state zero) 看做开始状态。最后一个状态，一般都是非发射状态和没有任何的转移，有最终的概率的那一个。在一个 HMM 中，你可以把转移概率看做最后一个状态的最终概率。在这个特定的例子里所有的发射状态他们有不同的 pdf's (因为 PdfClass 数目是不同的)。我们可以强制<PdfClass>数目一样。在对象 HmmTopology 里概率是用来初始化训练的；这个训练概率对于上下文相关的 HMM 是特定的和存储在 TransitionModel 对象。TransitionModel 用一个类名来存储 HmmTopology 对象，但是要注意，在初始化 TransitionModel 后，一般不使用在 HmmTopology 对象的转移概率。这里有个特例，对于那些不是最终的非发射状态(比如：他们有转移但是没有<PdfClass> entry)，Kaldi 不训练这些转移概率，而是用 HmmTopology 对象里给的概率。对于不支持非发射状态的转移概率的训练，简化了我们的训练机制，和由于非转移状态有转移是不正常的，我们就觉得这些没有很大的损失。

Pdf-classes

pdf-class 是一个与 HmmTopology 对象有关的概念。HmmTopology 对象为每个音素指定了一个 prototype HMM。"prototype HMM"的每一个状态数有一个"pdf_class"变量。如果两个状态有相同的 pdf_class 变量，他们将共享相同的概率分布函数(p.d.f.)，如果他们在相同的音素上下文。这是因为决策树代码不能够直接“看到”HMM 状态，它仅仅可以看到 pdf-class。在正常的情况下，pdf-class 和 HMM 状态的索引(e.g. 0, 1 or 2)一样，但是 pdf classes 为用户提供一种强制共享的方式。如果你想丰富转移模型但是想离开声学模型，这个是非常有用的，要不然就变成一样的。pdf-class 的一个功能就是指定非发射状态。如果一些 HMM 状态的 pdf-class 设定常量 kNoPdf = -1，然后 HMM 状态是非发射的(它与 pdf 没有联系)。这个可以通过删除<PdfClass>标签和相联系的数目来简化对象的文本格式。

一个特定的 prototype HMM 的 pdf-classes 的设置期望是从0开始和邻近的(e.g. 0, 1, 2)。这些是为了图建立代码的方便和没有导致任何的损失。

Transition models (the TransitionModel object)

TransitionModel 对象存储转移概率和关于 HMM topologies 的信息 (它包括一个 HmmTopology 对象)。图建立代码根据 TransitionModel 对象来获得 topology 和转移概率(它也需要一个 ContextDependencyInterface 对象来获得与特定音素上下文的 pdf-ids)。

How we model transition probabilities in Kaldi

The decision that underlies a lot of the transition-modeling code is as follows: 我们很据下面的4件事情决定一个上下文相关的 HMM 状态的转移概率(你可以把他们看做4-tuple):

- 音素(HMM 里的)
- 源 HMM-state (我们解释成 HmmTopology 对象, i.e. normally 0, 1 or 2)
- pdf-id (i.e. 与状态相关的 pdf 索引)
- 在 HmmTopology 对象的转移的索引

这4项可以看做在 HmmTopology 对象里的目标 HMM 状态的编码。根据这些事情来求转移概率的原因是，这是最细粒度（the most fine-grained way）的方式，我们在不增加编译解码图的大小来对转移建模。对于训练转移概率也是非常方便的。事实上，这与用传统的方法来对转移尽可能精确的建模可能没有任何区别，但比在单音素层面上的共享转移的 HTK 方式可能更加有效。

The reason for transition-ids etc.

TransitionModel 对象当被初始化时，就被设置成许多整数的映射，也被其他部分的代码用作这些映射。除了上面提到的数量，也有转移标识符(transition-ids), 转移索引(这个与转移标识符不是一样的), 和转移状态。之所以要介绍这些标识符和相联系的映射，是因为我们可以用一个完整的基本 FST 的训练脚本。最自然的基本 FST 的设置是输入标签是 pdf-ids。然而，考虑到我们的树建立的方法，它不是总可能从 pdf-id 唯一的映射到一个音素，所以那样很难从输入的标签序列映射到音素序列，和这不是很方便的很多原因；一般很难仅仅用 FST 的信息来训练转移概率。基于这个原因，我们把转移标识符 transition-ids 作为 FST 的输入标签，和把这些映射到 pdf-id，也可以映射到音素，也可以映射到一个 prototype HMM (在 HmmTopology 对象里)具体的转移概率。

Integer identifiers used by TransitionModel

以下类型的标识符都用于 TransitionModel 接口，它们所有的都用 int32类型。注意其中的一些量是从1开始索引的，也有有的是从0开始索引的。我们尽可能

的避免从1索引，因为与 C++数组索引不兼容，但是因为 OpenFst 把0看做特定的情况(代表特殊符号 epsilon), 我们决定把那些频繁在 FST 中用作输入符号的从1开始索引，我们就允许从1开始。更重要的是，transition-ids 是从1开始的。因为我们没有想到作为 FST 的标签，pdf-ids 出现很频繁，和因为我们经常使用他们作为 C++数组索引，我们决定使他们从0开始，但当 pdf-ids 作为 FST 的输入标签，我们就给 pdf-ids 加1。当我们阅读 TransitionModel 代码时，我们应该对索引从1开始的量，如果是这种情况我们就减去1，如果不是就不用减去。我们记录这些声明的成员变量。在任何情况，这些代码不是公开接口的，因为那会引起很多困惑。在 TransitionModel 中用的许多整数量如下：

音素(从1开始): 标识符的类型贯穿整个工具箱；用 OpenFst 符号表可以把 pdf-ids 转换成一个音素。这类的 id 不一定是邻近的(工具箱允许跳过音素索引)。

- Hmm 状态(从0开始)：这是 HmmTopology::TopologyEntry 类型的索引，一般取值为{0, 1, 2}里的一个。
- Pdf 或者 pdf-id (从0开始)：这是 p.d.f.的索引，由决策树的聚类来初始化(看 PDF identifiers)。在一个系统里通常由几千个 pdf-ids。
- 转移状态，或者 trans_state (从0开始)：这是由 TransitionModel 自己定义的。每个可能的(phone, hmm-state, pdf) 映射到一个自己的转移状态。转移状态可以认为是 HMM 状态最细化的分割尺度，转移概率都是分别评估的。
- 转移索引或者 trans_index (从0开始)：这是 HmmTopology::HmmState.类型中的转移数组的索引，这个索引记录了转移出某个转移状态的次数。
- 转移标识符或者 trans_id (从0开始)：其中的每一个都对应转移模型中的唯一的一个转移概率。用从(transition-state, transition-index) 到 transition-id 的映射，反之亦然。

在转移模型代码中也参考如下的概念：

- A triple 意思就是 a triple (phone, hmm-state, pdf) 从转移状态中映射。
- A pair 意思就是 a pair (transition-state, transition-index) 从转移 id 中映射。

Training the transition model

转移模型的训练步骤是非常简单的，我们创建的 FST(训练和测试中) 将 transition-ids 作为他们的输入标签。在训练阶段，我们做维特比解码，生成输入标签的序列，就是每一个 transition-ids (每个特征向量一个，意思也就是对应每帧输出有限个 transition-id 的序列)的序列。训练转移概率时我们积累的统计量就是数每个 transition-id 在训练阶段出现了多少次(代码本身用浮点数来数，但在正常的训练时我们仅仅用整数)。函数 Transition::Update() 对每一个 transition-state 做最大似然更新。这个明显是非常有用的。这里也有一些小的和概率下线有关的困难，如果一个特定的 transition-state 不知道，我们怎么做？更多的细节，请看代码。

Alignments in Kaldi

这里我们将介绍对齐的概念。说到对齐，我们意思就是<int32>类型的向量，含有 transition-ids 的序列，(c.f. Integer identifiers used by TransitionModel)它的长度跟需要对齐的那句话一样长。transition-ids 序列可以通过对输入标签序列解码获得。对齐通常用在训练阶段的维特比训练和在测试时间的自适应阶段。因为 transition-ids 编码了音素的信息，所以从对齐中计算出音素的序列是可行的 (c.f. SplitToPhones() and ali-to-phones.cc)。

我们经常需要处理以句子 id 做索引的一系列对齐。为了方便，我们用表来读和写“对齐”，更多的信息可以看 I/O with alignments 。

函数 ConvertAlignment() (c.f.命令行 convert-ali)把对齐从一个转移模型转换成其他的。这种典型的情况就是你使用一个转移模型(由特定的决策树来创建)来做对齐，和希望将其转换为不同的树的其他的转移模型。最好的就是从最原始的音素映射到新的音素集；这个特征通常不是一定要的，但是我们使用它来处理一些由减少的音素集的简化的模型。

在对齐的程序里通常使用一个后缀"-ali"。

State-level posteriors

状态级的后验概率是“对齐”这个概念的扩展，区别于每帧只有一个 transition-ids，这里每帧可以有任意多的 transition-ids，每个 transition-ids 都有权重，存储在下面的结构体里：

```
typedef std::vector<std::vector<std::pair<int32, BaseFloat> > > Posterior;
```

如果我们有个 Posterior 类型的对象"post"，post.size()就等于一句话的帧数，和 post[i] 是一串 pair (用向量存储)，每一个 pair 有一个 (transition-id, posterior) 构成。

在当前的程序里，有两个方式创建后验概率：

- 用程序 ali-to-post 来讲对齐转换为后验，但是生成的 Posterior 对象很细小，每帧都有一个单元后验的 transition-id
- 用程序 weight-silence-post 来修改后验，通常用于减小静音的权重。

未来，当加入 lattice generation，我们将会从 lattice 生成后验。

与"-ali"相比,读入的后验程序没有后缀。这是为了简洁；读取状态层的后验要考虑对齐信息的这类程序的默认值。

Gaussian-level posteriors

一个句子的高斯级别的后验概率用以下的形式存储：

```
typedef std::vector<std::vector<std::pair<int32, Vector<BaseFloat> > > > GauPost;
```

这个和状态级的后验概率结构体非常相似，除了原来是一个浮点数，现在是一个浮点数向量(代表状态的后验)，每个值就是状态里的每个高斯分量。向量的

大小跟 pdf 里的高斯分量的数目一样，pdf 对应于转移标识符 transition-id ，也就是每个 pair 的第一个元素。

程序 post-to-gpost 把后验概率的结构体转换为高斯后验的结构体，用到模型和特征来计算高斯级别的后验概率。当我们需要用不同的模型或特征计算高斯级别的后验时，这个就有用了。读取高斯后验程序有一个后缀"-gpost"。

Functions for converting HMMs to FSTs

把 HMMs 转换为 FSTs 的整个函数和类的列表可以在 here 这里找到。

GetHTransducer()

最重要的函数就是 GetHTranducer(), 声明如下：

```
fst::VectorFst<fst::StdArc>*
GetHTransducer(const std::vector<std::vector<int32> > &ilabel_info,
constContextDependencyInterface &ctx_dep,
constTransitionModel &trans_model,
constHTransducerConfig &config,
std::vector<int32> *disambig_syms_left);
```

如果没有介绍 ilabel_info 对象，ContextDependencyInterface 接口，和 fst 在语音识别里的一些应用基础，这个函数的很多方面大家会很难理解。这个函数返回一个 FST，输入标签是 transition-ids ，输出标签是代表上下文相关的音素(他们有对象 ilabel_info 的索引)。FST 返回一个既有初始状态又有最终状态，所有的转移都有输出标签(CLG 合理利用)。每一个转移都是一个3状态的 HMM 结构，和循环返回最初状态。FST 返回 GetHTransducer()仅仅对表示 ilabel_info 的上下文相关音素有效，我们称为特定。这是非常有用的，因为对于宽的上下文系统，有很多数量的上下文，大多是没有被使用。ilabel_info 对象可以从 ContextFst (代表 C)对象获得，在合并它和一些东西，和它含有已经使用过的上下文。我们提供相同的 ilabel_info 对象给 GetHTransducer()来获得覆盖我们需要的一个 H 转换器。

注意 GetHTransducer()函数不包括自环。这必须通过函数 AddSelfLoops()来添加；当所有的图解码优化后，仅仅添加自环是很方便的。

The HTransducerConfig configuration class

The HTransducerConfig configuration class 控制了 GetHTransducer 行为。

- 变量 trans_prob_scale 是一个转移概率尺度。当转移概率包含在图里，他们就包含了这个尺度。命令行就是–transition-scale。对尺度的合理使用和讨论，可以看 Scaling of transition and acoustic probabilities 。
- 一个变量 reverse 和二个其他变量是可以修改的，如果"reverse"选项是 true。"reverse" 选项是为后向解码创建一个 time-reversed FSTs。为了使这个有用，我们需要添加功能到 Arpa 语言模型里，我们将在以后去做。

The function GetHmmAsFst()

函数 GetHmmAsFst()需要一个音素上下文窗和返回一个用 transition-ids 作为符号的对应的有限状态接受器。这个在 GetHTransducer()使用。函数 GetHmmAsFstSimple()有很少的选择被提供，是为了表示在原理上是怎么工作的。

AddSelfLoops()

AddSelfLoops()函数把自环添加到一个没有自环的图中。一个典型的就是创建一个没有自环的 H 转换器，和 CLG 一起，做确定性和最小化，和然后添加自环。这样会使确定性和最小化更有效。AddSelfLoops()函数哟选项来重新对转移排序，更多的细节可以看 Reordering transitions。它也有一个转移概率尺度，"self_loop_scale",这个跟转移概率尺度不是一样的，可以看 Scaling of transition and acoustic probabilities。

Adding transition probabilities to FSTs

AddTransitionProbs()函数添加转移概率到 FST。这样做时有用的，原因是我们创造了一个没有转移概率的图(i.e. without the component of the weights that arises from the HMM transitions), 和他们将在以后被添加；这样就使得训练模型的不同的迭代使用相同的图成为可能，和保持图中的转移概率更新。创造一个没有转移概率的图是使用 trans_prob_scale (command-line option: –transition-scale) 为0来实现的。在训练的时候，我们的脚本开始存储没有转移概率的图，然后每次我们重新调整时，我们就增加当前有效的转移概率。

Reordering transitions

AddSelfLoops()函数有一个布尔选项"reorder" ，这个选项表明重新对转移概率排序，the self-loop comes after the transition out of the state。当应用变成一个布尔命令行，比如，你可以用 –reorder=true 在创建图时重新排序。这个选项使得解码更加简单，更快和更有效(看 Decoders used in the Kaldi toolkit)，尽管它与 kaldi 的解码不兼容。

重排序的想法是，我们切换自环弧与所有其他状态出来的弧，所以自环维语每一个互相连接的弧的目标状态（The idea of reordering is that we switch the order of the self-loop arcs with all the other arcs that come out of a state, so the self-loop is located at the destination state of each of the other arcs）。为了这个，我们必须保证 FST 有某些特性，即一个特定状态的所有弧必须引导相同的自环(也，一个有自环的状态不能够有一个静音的输入弧，或者是开始的状态)。AddSelfLoops()函数修改图，以确保他们有这个特性。一个相同的特性也是需要的，即使"reorder"选项设置为 false。创建一个有"reorder"选项的图准确的说就相对于你解码一个句子得到的声学和转移模型概率而言，是一个非重新排序的图 。得到的对齐的 transition-ids 是一个不同的顺序，但是这个不影响你使用这些对齐。

Scaling of transition and acoustic probabilities

这里有三种在 kaldi 中使用的尺度类型：

Name in code	Name in command-line arguments	Example value (train)	Example value (test)
acoustic_scale	–acoustic-scale=?	0.1	0.08333
self_loop_scale	–self-loop-scale=?	0.1	0.1
transition_scale	–transition-scale=?	1.0	1.0

你也许注意到这里没有用语言模型的尺度；相对于语言模型来说，任何事情都是尺度。我们不支持插入一个词的惩罚，一般来说(景观 kaldi 的解码不支持这个)。语言模型表示的真正的概率，一切相对于他们的尺度都是有意义的人。在训练阶段的尺度是我们在通过 Viterbi 对齐解码得到的。一般而言，我们用0.1，当一个参数不是很关键和期望它是小的。当测试是很重要的和这个任务是调整的，声学尺度将被使用。现在我们来解释这三个尺度：

- 声学尺度是应用到声学上的尺度(比如：给定一个声学状态的一帧的 log 似然比).
- 转移尺度是转移概率上的尺度，但是这个仅仅适合多个转换的 HMM 状态，它应用到这些转移中的相对的权重。它对典型的没有影响。
- 自环尺度是那些应用到自环的尺度。特别的是，当我们添加自环，让给定自环的概率为 p，而剩下的为(1-p)。我们添加一个自环，对数概率是 self_loop_scale * log(p), 和对所有其他不在这个状态的对数转移概率添加 (self_loop_scale * log(1-p))。在典型的 topologies，自环尺度仅仅是那个有关系的尺度。

我们觉得对自环应用一个不同的概率尺度，而不是正常的转移尺度有意义的原因是我们认为他们可以决定不同时间上事件的概率。稍微更加微妙的说法是，所有的转移概率可以被看成真正的概率(相对于 LM 概率), 因为声学概率的相关性的问题与转移没有关系。然而，一个问题出现了，因为我们在测试时使用 Viterbi 算法(在我们的例子里，训练也是)。转移概率仅仅表示当累加起来的真正的概率，在前向后向算法里。我们期望这个是自环的问题，而不是由于 HMM 的完全不同的路径的概率。因为后面的情况里，声学分布通常不是连接的，在前向后向算法和 Viterbi 算法里的差别很小。

4.12 kaldi 里的聚类机制

kaldi 里的聚类机制

这里讲阐述在 kaldi 里的聚类机制和接口。

可以看 Classes and functions related to clustering 来了解涉及到的类和函数列表。这里不包括音素决策树聚类(看 Decision tree internals 和 How decision trees are used in Kaldi)，尽管这里介绍的类和函数是在音素聚类的代码的底层使用。

The Clusterable interface

TheClusterable 类是一个从类 GaussClusterable 继承来的一个纯虚拟类(GaussClusterable 表示高斯统计量)。未来我们可以添加一些其他类型的从 Clusterable 继承的 clusterable。用 Clusterable 类的原因就是它允许我们使用通用的聚类算法。

Clusterable 接口的核心概念就是把统计量添加到一起，和用目标函数来衡量。二个 Clusterable 对象距离的概念是从分别衡量这两个对象的目标函数得来的，然后把他们添加到一起，和衡量他们的目标函数；目标函数的降低来定义距离。

我们想加进去的 Clusterable 类的例子在一定程度上包含从一个固定的，共享的混合高斯模型里得带的混合高斯的统计量，和离散观测量的计数(目标函数等于这个分布的否定熵，计算的数量)。

得到类型 Clusterable*指针(事实上是 GaussClusterable 类型) 的一个例子如下：

```
Vector<BaseFloat> x_stats(10), x2_stats(10);
BaseFloatcount = 100.0, var_floor = 0.01;
// initialize x_stats and x2_stats e.g. as
// x_stats = 100 * mu_i, x2_stats = 100 * (mu_i*mu_i + sigma^2_i)
Clusterable *cl = new GaussClusterable(x_stats, x2_stats, var_floor, count);
```

Clustering algorithms

我们已经实现了许多通用的聚类算法，这些都列在 Algorithms for clustering。在这些算法用的最多的数据结构就是 Clusterable 接口类的一个 vector 指针：

```
std::vector<Clusterable*> to_be_clustered;
```

vector 的索引是聚类的指针的索引。

K-means and algorithms with similar interfaces

调用聚类代码的一个例子如下：

```
std::vector<Clusterable*> to_be_clustered;
// initialize "to_be_clustered" somehow ...
std::vector<Clusterable*> clusters;
int32num_clust = 10; // requesting 10 clusters
ClusterKMeansOptions opts; // all default.
std::vector<int32> assignments;
ClusterKMeans(to_be_clustered, num_clust, &clusters, &assignments, opts);
```

当聚类代码被调用时，"assignments" 将告诉你每一个 item 在 "to_be_clustered"，每一个 cluster 被分配。ClusterKMeans()算法即使对大数据的 points 也是很有效的；更多的细节可以点击这个函数看看。

有2个算法和 ClusterKMeans()有相同的接口：即 ClusterBottomUp()和 ClusterTopDown()。也许最有用的一个是 ClusterTopDown()，当聚类数量很大时，它比 ClusterKMeans()更有效。(它做一个二次分裂，和然后又在叶子上做二次分裂等等)。它被叫做 TreeCluster()。

Tree clustering algorithm

TreeCluster()函数把 points 聚类成一个二叉树(每个叶子不是仅仅含有一个 point，你可以特定叶子的最大数目)。这个函数是非常有用的，举例来说，当为自适应建立一个聚类树。输出格式的更多细节可以看函数的文档。快速浏览的是，在首先是叶子和最后是根结点的拓扑顺序的叶子和非叶子结点，和输出一个告诉你每一个结点的父节点是什么的 vector。

4.13 Decoding-graph creation recipe (training time)

Decoding-graph creation recipe (training time)

在训练阶段的图建立脚本是比在测试阶段的简单，主要是因为我们没有不需要消歧符号。

我们假设你已经读过了测试阶段的 recipe。

在训练阶段我们用测试阶段相同的 HCLG 形式，除了 G 是相当于训练脚本构成的一个线性接受器(当然，这个建立是很容易扩展到在那些脚本的不确定)。

Command-line programs involved in decoding-graph creation

这里我们将解释在训练脚本时发生了什么；下面我们在看程序里发生了什么。假设我们已经建立了一个树和一个 model。接下里的命令建立一个包含对应每个训练 transcripts 的图 HCLG(c.f. The Table concept)。

```
compile-train-graphs $dir/tree $dir/1.mdl data/L.fst ark:data/train.tra \
  ark:$dir/graphs.fsts
```

输入文件 train.tra 是一个包含训练 transcripts 的整数版本，比如典型的命令行就是：

```
011c0201 110906 96419 79214 110906 52026 55810 82385 79214 51250 106907
111943 99519 79220
```

第一个 token 就是 utterance id。程序的输出是一个 archive graphs.fsts；它包含在 train.tra 的每一个句子的一个 FST (in binary form)。对应 HCLG 的这个 FST, 除了没有转移概率(默认的, compile-train-graphs 有–self-loop-scale=0 和–transition-scale=0)。这是因为这些图是用来多阶段训练和转移概率将被改变，所以我们在后面加入他们。但是 FSTs 将有从静音概率(这些概率被编码在 L.fst), 上升的概率，和如果我们使用发音概率，这些也将被显示出来。

一个读取这些 archives 和对训练数据解码的命令是，接下来将产生 state-level alignments (c.f. Alignments in Kaldi)；我们将简单的回顾这个命令，尽管这页我们的焦点是图建立本身。

```
gmm-align-compiled \
  --transition-scale=1.0 --acoustic-scale=0.1 --self-loop-scale=0.1 \
    --beam=8 --retry-beam=40 \      $dir/$x.mdl ark:$dir/graphs.fsts \
    "ark:add-deltas --print-args=false scp:data/train.scp ark:- |" \
      ark:$dir/cur.ali
```

前三个参数就是概率尺度(c.f. Scaling of transition and acoustic probabilities)。transition-scale 和 self-loop-scale 选择在这里的原因是在解码之前，程序需要添加转移概率到图上(c.f. Adding transition probabilities to FSTs)。接下来的选项是 beams；我们使用一个初始化的 beam，和然后如果对齐没能达到最后的状态，我们使用其他的 beams。因为我们使用一个声学的尺度0.1，相对于声学似然比，我们将不得不对这些 beams 乘以10来得到 figures。程序需要 model；通过迭代，$x.mdl 可以扩展为1.mdl 或者2.mdl。它通过 archive (graphs.fsts)来读图。在引号里的参数被认为是一个管道(minus the "ark:" and "|")，和被解释成 utterance-id 的 archive 索引的，包含特征。输出就是 cur.ali，和如果写成 txt 格式，它看来像是上面描述的.tra 文件，尽管整数现在对应的不是 word-ids 而是 transition-ids (c.f. Integer identifiers used by TransitionModel)。

我们注意到这两个阶段(graph creation and decoding)可以通过一个命令行程序完成：gmm-align，可以编译你想要的图。因为图建立需要花费相对长的时间，当把他们写到磁盘里，任何图都需要不止一次写入。

Internals of graph creation

图建立都是在训练时间里完成的，无论是程序 compile-train-graphs 还是 gmm-align，都是有相同的代码来完成的： 类 TrainingGraphCompiler。当一个接着一个编译图，他们的行为如下：

在初始化：

- 添加子序列自环(c.f. Making the context transducer) 到词典 L.fst，确保按输出标签排序和存储。

当编译一个 transcript, 它执行以下的步骤：

- 生成一个相对于词序列的线性接收器
- 和词典一起编译它 (using TableCompose) 得到一个含有输入是音素和输出是词的 FST (LG) ；这个 FST 包含发音和静音的选项。

- Create a new context FST "C" that will be expanded on demand.
- Compose C with LG to get CLG; the composition uses a special Matcher that expands C on demand.
- Call the function GetHTransducer to get the transducer H (this covers just the context-dependent phones that were seen).
- Compose H with CLG to get HCLG (this will have transition-ids on the input and words on the output, but no self-loops)
- Determinize HCLG with the function DeterminizeStarInLog; this casts to the log semiring (to preserve stochasticity) before determinizing with epsilon removal.
- Minimize HCLG with MinimizeEncoded (this does transducer minimization without weight-pushing, to preserve stochasticity).
- 添加自环。

TrainingGraphCompiler 类有一个函数 CompileGraphs() ，这个函数是在一个 batch 里联合一些图。这个被用在工具 compile-train-graphs 来加速图的编译。主要的原因是可以帮助在一个 batch 里，当创建 H 转换器时，我们仅仅需要处理每一个看到的上下文窗，即使它被用在许多的 CLG 例子里。

没用第一个添加消歧符号做确定性的原因是，在这个情况下 HCLG 是函数(因为任何一个输入标签序列转换为同一个 string)和非周期的；任何一个非周期的 FST 有两个特性，和任何函数性的有两个特性的 FST 是确定性的。

4.14 Decoding-graph creation recipe (test time)

Decoding-graph creation recipe (test time)

这里我们将解释怎样一步一步的建立我们正常的图模型，可能会涉及到一些数据准备阶段的知识。

这个方法的许多细节没有被编码到我们的工具里；我们仅仅解释现在我们是怎么做的。如果这部分你感到迷惑，最好的办法就是去读 Mohri 写的"Speech Recognition with Weighted Finite-State Transducers" 。警告你的是，这篇文章非常长，如果你对 FST 不熟悉可能需要花费几个小时。还有一个更好的资料就是 OpenFst website ，可以提供像符号表一样的更多的介绍。

Preparing the initial symbol tables

我们现在准备 OpenFst 符号表 words.txt 和 phones.txt。在我们的系统里，他们用整数 id 来分配所有的词和音素。注意 OpenFst 为静音保留符号0。对于 WSJ 任务的一个符号表的例子：

```
## head words.txt
<eps> 0
!SIL 1
<s> 2
</s> 3
<SPOKEN_NOISE> 4
<UNK> 5
```

```
<NOISE> 6
!EXCLAMATION-POINT 7
"CLOSE-QUOTE 8
## tail -2 words.txt
}RIGHT-BRACE 123683
#0 123684
## head data/phones.txt
<eps> 0
SIL 1
SPN 2
NSN 3
AA 4
AA_B 5
```

words.txt 文件包含单个消歧符号"#0" (在 G.fst 的输入里用于静音)。在我们的脚本里这是最后数字的词。如果你的词典含有一个词"#0"，你就要小心点。phones.txt 文件不包含消歧符号，但是当我们创建 L.fst 后，我们将创建一个含有消歧符号的 phones_disambig.txt 文件(在调试时这是非常有用的)。

Preparing the lexicon L

首先我们创建一个文本格式的词典，初始化时没有消歧符号。我们的 C++ 工具将不会跟它交互，它仅仅在创建词典 FST 是被使用。我们 WSJ 词典的一小部分是：

```
## head data/lexicon.txt
!SIL SIL
<s>
</s>
<SPOKEN_NOISE> SPN
<UNK> SPN
<NOISE> NSN
!EXCLAMATION-POINT EH2_B K S K L AH0 M EY1 SH AH0 N P OY2 N
T_E
"CLOSE-QUOTE K_B L OW1 Z K W OW1 T_E
```

音素的开始，结束和应急标记物(e.g. T_E, or AH0) 在我们的 WSJ 脚本中是特定的，和至于我们的工具箱，它们被视为不同的音素(然而，对于这种设置，我们在树建立时特定的处理；可以看在 The tree building process 里的根文件)。

注意我们允许含有空音素表示的词。在训练时词典被用来创建 L.fst(不含消歧符号)。我们也创建一个含消歧符号的词典，在图解码创建的时候使用。这个文件的一部分在这里：

```
# [from data/lexicon_disambig.txt]
!SIL        SIL
<s> #1
</s>        #2
<SPOKEN_NOISE>    SPN #3
<UNK>      SPN #4
<NOISE> NSN
...
```

```
{BRACE    B  B R EY1 S  E #4
{LEFT-BRACE L_B EH1 F T B R EY1 S_E #4
```

这个文件通过一个脚本创建；脚本的输出就是我们需要添加的消歧符号的数目，和这个用来创建符号表 phones_disambig.txt。这个 phones.txt 一样，但是它也包含消歧符号#0, #1, #2等等的整数 id (#0是从 G.fst 得到的一个特殊的消歧符号，但是将在 L.fst 由自环过滤掉)。 phones_disambig.txt 文件中间的一部分是：

```
ZH_E 338
ZH_S 339
#0    340
#1    341
#2    342
#3    343
```

这个数字是非常大，因为在 WSJ 脚本中我们为音素添加 stress and position 信息。注意用在空词的消歧符号(i.e. <s> and </s>)与用在正常词的消歧符号是不一样的，所以在这个例子里正常消歧符号是从#3开始的。

把没有消歧符号的词典转换为 FST 的命令是：

```
scripts/make_lexicon_fst.pl data/lexicon.txt 0.5 SIL | \
    fstcompile --isymbols=data/phones.txt --osymbols=data/words.txt \
    --keep_isymbols=false --keep_osymbols=false | \
     fstarcsort --sort_type=olabel > data/L.fst
```

这里，脚本 make_lexicon_fst.pl 创建一个文本格式的 FST。这个0.5是在句子的开头的静音概率(比如：在句子的开头和在每个词后，我们输出一个概率为0.5的静音；所以分配给没有静音的概率就是1.0 - 0.5 = 0.5。这个例子里命令的其他部分与把 FST 转换为编译的形式有关；fstarcsort 是很有必要的，因为我们稍后将组合它们。

词典的结构大约跟我们期待的一样。有一个最终的状态(环状态) 。存在具有二个转移成环状态的起始状态，一个是静音，一个没有静音。从环状态存在对应每个词的一个转移，词是在转移时的输出符号；输入符号是这个词的第一个音素。对于有效的组合和最小化来说，最重要的就是输出符号尽可能的早(i.e. 在词的开始而不是末尾)。在每个词的末尾，来处理可选的静音，对于最后音素对应的转移有二种形式：一种对环状态和一种它具有与环状态转移的 "silence state"。在静音词后，我们不必烦扰放置可选的静音，我们将定义这是一个静音音素的词。

创建一个带有消歧符号的词典是有点复杂的。问题就是我们不得不添加自环到词典里，以至于从 G.fst 的消歧符号#0 可以被词典过滤。我们将通过程序 fstaddselfloops 来做(参考 Adding and removing disambiguation symbols)，尽管我们可以很容易通过脚本 make_lexicon_fst.pl 自动完成。

```
phone_disambig_symbol=`grep \#0 data/phones_disambig.txt | awk '{print $2}'`
```

```
word_disambig_symbol=`grep \#0 data/words.txt | awk '{print $2}'`

scripts/make_lexicon_fst.pl data/lexicon_disambig.txt 0.5 SIL  | \
    fstcompile --isymbols=data/phones_disambig.txt --osymbols=data/words.txt \
    --keep_isymbols=false --keep_osymbols=false |    \
    fstaddselfloops  "echo $phone_disambig_symbol |" "echo
$word_disambig_symbol |" | \
    fstarcsort --sort_type=olabel > data/L_disambig.fst
```

程序 fstaddselfloops 不是原始的 OpenFst 的命令行工具，是我们自己的工具(我们有许多这样的程序)。

Preparing the grammar G

语法 G 是把词作为它的符号的接收器最重要的部分(i.e.输入和输出符号是在每个弧相同)。特殊的就是消歧符号#0 仅仅出现在输入这边。假设我们的输入是一个 Arpa 文件，我们用 kaldi 程序 arpa2fst 来转为为 FST。程序 arpa2fst 的输出是一个有嵌入符号的 FST。在 kaldi 中，我们一般使用没有嵌入符号的 FSTs (i.e. 我们用分离的符号表)。除了仅仅运行 arpa2fst，我们还需要做的步骤如下：

- 我们必须从 FST 中移除嵌入符号(和他们依赖于磁盘里的符号表)。
- 我们必须确保语言模型里没有词典以外的词
- 我们必须移除句子开始和结束读好的不合法的序列，比如 <s> followed by </s>,因为这些导致 L o G 不是确定性。
- 我们必须在输入时用特殊的符号#0代替静音。

做这个工作的实际脚本的稍微简化版本如下：

```
gunzip -c data_prep/lm.arpa.gz | \
    scripts/find_arpa_oovs.pl data/words.txt  > data/oovs.txt

gunzip -c data_prep/lm.arpa.gz | \
   grep -v '<s> <s>' | \
   grep -v '<s> </s>' | \
   grep -v '</s> </s>' | \
    ../src/lm/arpa2fst - | fstprint | \
   scripts/remove_oovs.pl data/oovs.txt | \
   scripts/eps2disambig.pl | \
    fstcompile --isymbols=data/words.txt --osymbols=data/words.txt \
    --keep_isymbols=false --keep_osymbols=false > data/G.fst
fstisstochastic data/G.fst
```

最后一个命令(fstisstochastic)是一个诊断步骤(see Preserving stochasticity and testing it)。在一个典型的例子里，它将打印这些数字：

```
9.14233e-05 -0.259833
```

第一个数字是非常小的，所以它肯定在这个弧里没有一个状态的概率加上最后一个状态的概率小于1。第二次数字是有意义的，它意思就是存在有很大

概率的状态(在 FST 里的权重的数值可以解释为 log 概率)。对于一个有反馈的语言模型的 FST 的表示，有大概率的一些状态是正常的。在后来的图建立步骤，我们将确认这些非随机性并没有比开始的时候变得更糟。

最后结果的 FST G.fst 当然仅仅用在测试阶段。在训练阶段，我们使用从训练词序列得到的线性 FST，但是这是在 kaldi 的内部做的，不是脚本做的。

Preparing LG

当组合 L 和 G，我们坚持一个相对标准的脚本，比如我们计算 min(det(L o G))。命令行如下：

```
fsttablecompose data/L  disambig.fst data/G.fst | \
    fstdeterminizestar --use-log=true | \
    fstminimizeencoded   > somedir/LG.fst
```

这个与 OpenFst's 算法有些不一样。我们使用一个更加高效的组合算法(see Composition)的命令行工具"fsttablecompose"。我们的确定性也移除了静音的算法，有命令行 fstdeterminizestar 实现。选项–use-log=true 是询问程序 是否先把 FST 投掷到 log semiring；它保证随机性(在 log semiring)；可以看 Preserving stochasticity and testing it。

我们通过程序"fstminimizeencoded"做最小化。这个大部分跟运用到有权重的接收器的 OpenFst's 的最小化算法一样。仅仅的改变就是避免 pushing weights，因为保证随机性(更多的看 Minimization)。

Preparing CLG

为了得到一个输入是上下文相关的音素的转换器，我们需要准备一个叫 CLG 的 FST，它等于 C o L o G，这里的 L 和 G 是词典和语法，C 表示音素上下文。对于一个三音素系统，C 的输入符号是 a/b/c 形式的(i.e. 音素的三元组)，和输出符号是一个单音素(e.g. a or b or c)。关于音素上下文窗可以看 Phonetic context windows，和如何用到不同的上下文大小里。首先，我们描述如何来创建上下文 FST C，如果我们去通过自身来做和正常的组合(为了有效和可扩展性，我们的脚本事实上不那么去做)。

Making the context transducer

这部分我们来解释如何让获得作为一个独自的 FST C。

C 的基本结构是它由所有可能音素窗大小 N-1的状态(c.f. Phonetic context windows; N=3在三音素的情况里)。在第一个状态，意味着句子的开始，仅仅对应 N-1静音。每一个状态有每个音素的转移(现在让我们忘记自环)。作为一个通用的例子，状态 a/b 在输出时有一个 c 的转移和输入是 a/b/c，到状态 b/c。在句子的输入和输出是特殊的情况。

在句子的开始，假设状态是<eps>/<eps> 和输出符号是 a。通常，输入符号是<eps>/<eps>/a。但是这个不代表一个音素(假设 P = 1)，中心元素是<eps> ，它不是一个音素。在这种情况下，我们让弧的输入符号是一个特殊的符号#-1 ，这是我们介绍的主要原因。(作为标准的脚本，我们不使用静音这里，当有空词时会导致 nondeterminizability)。

句子的结尾是有点复杂的。上下文 FST 在右边(它的输出边)会有一个在句子末尾的特殊的符号$。考虑三音素的情况。在句子的末尾，看完所有的符号后，我们需要过滤掉最后一个三音素(e.g. a/b/<eps>, where <eps> represents undefined context)。很自然的方式就是在输入是 a/b/<eps> 和输出是<eps>之间有个传递，从状态 a/b 到最后一个状态(e.g. b/<eps>或者一个特定的最终状态)。但是对于组合这不是有效的，因为如果它不是句子的末尾，我们不得不在删掉这些之前发现这些转移。我们在句子的末尾用符号$ 代替，和保证它出现在 LG 的每一条路径的末尾。然后我们在 C 的输出用$代填<eps>。一般而言，$的重复数目是 N-P-1。为了避免计算出有多少后续的符号添加到 LG 的麻烦，我们仅仅允许它在句子的末尾接受任何数量这样的符号。可以通过函数 AddSubsequentialLoop()和命令行程序 fstaddsubsequentialloop 得到。

如果我们想要它自己的 C，我们首先需要消歧符号的列表；和我们需要计算一个没有使用的符号 id，这些符号我们将用在后续的符号里，如下：

```
grep '#' data/phones_disambig.txt | awk '{print $2}' > $dir/disambig_phones.list
subseq_sym=`tail -1 data/phones_disambig.txt | awk '{print $2+1;}'`
```

We could then create C with the following command:

```
fstmakecontextfst --read-disambig-syms=$dir/disambig_phones.list \
--write-disambig-syms=$dir/disambig_ilabels.list data/phones.txt $subseq_sym \
  $dir/ilabels | fstarcsort --sort_type=olabel > $dir/C.fst
```

程序 fstmakecontextfst 需要音素的一个列表，消歧符号的一个列表和后续符号的标识。除了 C.fst，它写出了解释 C.fst 左边的符号的"ilabels"文件(看 The ilabel_info object)。LG 的组合可以按下面的做：

```
fstaddsubsequentialloop $subseq_sym $dir/LG.fst | \
  fsttablecompose $dir/C.fst - > $dir/CLG.fst
```

用于打印 C.fst 和使用相同的索引为"ilabels"的符号，我们可以使用下面的命令来做一个合适的符号表：

```
fstmakecontextsyms data/phones.txt $dir/ilabels > $dir/context_syms.txt
```

这个命令要知道"ilabels"格式(The ilabel_info object)。CLG fst 的随机一条路径(for Resource Management)，打印它的符号表，接下来是：

```
## fstrandgen --select=log_prob $dir/CLG.fst | \
    fstprint --isymbols=$dir/context_syms.txt --osymbols=data/words.txt -
0    1    #-1 <eps>
1    2    <eps>/s/ax    SUPPLIES
```

```
2     3     s/ax/p    <eps>
3     4     ax/p/l    <eps>
4     5     p/l/ay    <eps>
5     6     l/ay/z    <eps>
6     7     ay/z/sil        <eps>
7     8     z/sil/<eps> <eps>
8
```

Composing with C dynamically

在正常的图建立脚本中，我们使用程序 fstcomposecontext ，可以动态的创建需要的状态和 C 的弧，而不不需要浪费的建立。命令行是：

```
fstcomposecontext    --read-disambig-syms=$dir/disambig_phones.list \
                              --write-disambig-syms=$dir/disambig_ilabels.list \
                              $dir/ilabels < $dir/LG.fst >$dir/CLG.fst
```

如果我们有不同的上下文参数 N 和 P ，相当于默认的(3 and 1)。我们将对这个程序使用其他的选项。这个程序写入文件"ilabels" (看 The ilabel_info object) ，可以解释为 CLG.fst 的输入符号。一个 rm 数据库的 ilabels 文件的开始几行是：

```
65028 [ ]
[ 0 ]
[ -49 ]
[ -50 ]
[ -51 ]
[ 0 1 0 ]
[ 0 1 1 ]
[ 0 1 2 ]
...
```

65028是文件里元素的个数。像[-49] 行是为了消歧符号；像[0 1 2] 行代表声学上下文窗；一开始的2个 entries 是为了静音(从来不使用)的[ ]和 and 特定的消歧符号[0] ，它的打印格式为#-1 ，这是在 C 的开始，为了替代静音，为了确保确定性。

Reducing the number of context-dependent input symbols

当创建 CLG.fst 后，就有一个减小其大小的图建立选项。,我们使用程序 make-ilabel-transducer，它可以形成决策树和得到 HMM 拓扑信息，上下文相关音素的子集相对于相同的图编译和将合并(我们选择每个自己的任意元素和把所有的上下文窗转为为这个上下文窗)。这个跟 HTK's logical-to-physical mapping 相似。命令行是：

```
    make-ilabel-transducer
--write-disambig-syms=$dir/disambig_ilabels_remapped.list \
      $dir/ilabels $tree $model $dir/ilabels.remapped > $dir/ilabel_map.fst
```

这个程序需要 tree 和 model；它的输入是一个新的 ilabel_info 对象，称为"ilabels.remapped"；这个跟原始的"ilabels"文件有相同的格式，但是有很少的行。FST "ilabel_map.fst" 是由 CLG.fst 构成的和重新映射标签的。当我们做了这个后，我们将确定性和最小化，所以我们可以马上实现任意大小的减少：

```
fstcompose $dir/ilabel_map.fst $dir/CLG.fst | \
  fstdeterminizestar --use-log=true | \
  fstminimizeencoded > $dir/CLG2.fst
```

这个阶段为了典型的建立事实上我们不会减少图大小很多(一般减少5% to 20%)，和在任何情况下，它是中间图建立阶段的大小，我们通过这个机制来减少。但是这个减少对宽上下文的系统是有意义的。

Making the H transducer

在传统的 FST 脚本里，H 转换器是有它的上下相关音素的输出和输入是代表声学状态的符号。在我们的情况下，H (or HCLG)的输入的符号不是声学状态(在我们的术语里是 pdf-id)，但是我们用 transition-id 来代替(see Integer identifiers used by TransitionModel)。transition-id 是对 pdf-id 加上包含音素的一些其他的信息的编码。每一个 transition-id 可以映射一个 pdf-id。我们创建的 H 转换器没有把自环编码进来。他们稍后将通过一个单独的程序添加进来。H 转换器 具有开始和最终的状态，和从这个状态的每个 entry 有一个转移除了在 ilabel_info 对象(the ilabels file, see above)里的第0个。上下文相关音素的转移到相对应的 HMM(缺少自环)的结构中，和然后到开始的状态。对于正常的拓扑结构，HMM 的这些结构仅仅是三个弧的线性序列。每一个消歧符号(#-1, #0, #1, #2, #3 等等)的开始状态，H 都有自环。

脚本的这个部分是生成 H 转换器(我们称为 Ha 因为这里缺少自环)是：

```
make-h-transducer --disambig-syms-out=$dir/disambig_tstate.list \
  --transition-scale=1.0 $dir/ilabels.remapped \
  $tree $model > $dir/Ha.fst
```

这是一个设置转移尺度的选项；在我们现在训练的脚本里，这个尺度是1.0。这个尺度仅仅影响那些与自环概率无关的转移部分。和正常的拓扑(Bakis model)没有任何影响；更多的解释可以看 Scaling of transition and acoustic probabilities。除了 FST，程序也写消歧符号的列表，这些符号稍后将被删除。

Making HCLG

生成最后一个图 HCLG 的第一部就是生成缺少自环的 HCLG。命令行是：

```
fsttablecompose $dir/Ha.fst $dir/CLG2.fst | \
  fstdeterminizestar --use-log=true | \
  fstrmsymbols $dir/disambig_tstate.list | \
  fstrmepslocal | fstminimizeencoded > $dir/HCLGa.fst
```

这里，CLG2.fst 是一个减少符号集版本的 CLG ("logical" triphones, in HTK terminology)。我们删除了消歧符号和任何容易删除的静音(看 Removing epsilons)，在最小化之前；我们最小化算法是避免 pushing symbols and weights (因此保证随机性)，和接受非确定行的输入(看 Minimization)。

Adding self-loops to HCLG

添加自环到 HCLG 是用下面的命令完成的：

```
add-self-loops --self-loop-scale=0.1 \
  --reorder=true $model < $dir/HCLGa.fst > $dir/HCLG.fst
```

怎么把 self-loop-scale 为0.1使用上的更多细节可以看 Scaling of transition and acoustic probabilities(注意这会影响 non-self-loop 概率)。"reorder"选项的更多解释，可以看 Reordering transitions；"reorder"选项增加了解码的速度，但是它与 kaldi 的解码不兼容。add-self-loops 程序不是仅仅添加自环，它也许会复制状态和添加静音转移来确保自环可以以一致的方式加入。这个事情的更多细节可以看 Reordering transitions。这是图建立中唯一一步不需要保证随机性；它不需要保证它，是因为 self-loop-scale 不为1。所以程序 fstisstochastic 可以给所有的 G.fst, LG.fst, CLG.fst 和 HCLGa.fst 相同的输出，但是不能是 HCLG.fst。在 add-self-loops 之后，我们不需要再做确定性；这个会无效，因为我们已经移除了消歧符号。所以，这个会很慢，和我们相信没有什么可以从确定性和最小化那里得到。

4.15 kaldi tutorial

以下是我看 kaldi 教程记的些笔记，希望能对你有所帮助（你可以把这个文档当成 kaldi tutorial 的简要翻译）命令行我都加了下划线。数据准备

这部分基本略过了，比较简单。

从 data/lang 说起。 data/lang 是由 prepare_lang.sh 生成的。

首先生成的是 words.txt 和 phones.txt，这是 openfst 格式的 symbol tables(后来我就喜欢直接写英文了，与其费心思翻译为合适的中文，不如直接用原词来得原汁原味)，它们是字符串（字符串是指标注文本的单位，）到整数的映射，下面这个是日语识别中的部分 words.txt：

```
<eps> 0
a 1
b 2
ch 3
d 4
e 5
f 6
g 7
h 8
i 9
j 10
k 11
m 12
n 13
```

再看 phones 文件夹下以.csl 为后缀的文件，分别有 disambig.csl nonsilence.csl optional_silence.csl silence.csl，它们的内容都是以冒号作为分割。他们是后续命令行中偶尔会用到的选项。

看 phones.txt，在 data/lang/下。这个是 phone symbol table，也包含了标准

FST 脚本中的“消歧符号”，这些符号就是 1，2，3，。。。。我们在此加了个 0 符号，代替了语言模型中的 epsilon transitions。

L.fst 文件是编译后的 FST 格式的词典（lexicon）。想看里面有什么信息，输入：(from s5/)

fstprint --isymbols=data/lang/phones.txt --osymbols=data/lang/words.txt data/lang/L.fst | head

然后下面是对应的输出：

```
0	1	<eps>	<eps>	0.693147182
0	1	sil	<eps>	0.693147182
1	2	a	a	0.693147182
1	1	a	a	0.693147182
1	1	b	b	0.693147182
1	2	b	b	0.693147182
1	1	ch	ch	0.693147182
1	2	ch	ch	0.693147182
1	1	d	d	0.693147182
1	2	d	d	0.693147182
```

G.fst 是个描述这种语言语法结构的 FST，下面是这个 G.fst 的内容：

```
0	2	21	21	0.000315946876
0	1	29	0	8.03088665
1	29	25	25	12.1202555
1	28	6	6	8.34519005
1	27	15	15	6.83906889
1	26	28	28	5.35383272
1	25	19	19	3.59180236
1	24	16	16	5.82719803
1	23	26	26	4.43024254
1	22	17	17	4.63756752
```

我觉得这应该就是对应词、或 phone 出现次数统计后算的概率。

提取特征

这一段是提取训练特征，首先跑对应的命令，然后来看生成的文件。

看看 exp/make_mfcc/train/make_mfcc.1.log ，首先给出的跑过的命令行，（kaldi 总是在 log 最上面显示命令行，）

split_scp.pl 就是把 scp 分成几个小的 scp(.ark 和.scp 是 kaldi 中两种记录数据的格式, .ark 是数据（二进制文件），scp 是记录对应 ark 的路径）

.ark 文件一般都是很大的（因为他们里面是真正的数据），可以通过下面这条命令来看：

copy-feats ark:mfcc/raw_mfcc_train.1.ark ark,t:- | head

以下是对应的输出：

```
NF089001  [
 53.54222 -31.82449 -9.899872 -0.02364012 -5.681367 2.072489 -19.41396
-15.6856 14.83652 25.04876 -11.34208 1.64803 9.309975
 49.06616 -28.41237 1.188962 -0.5514585 -14.60496 -6.065259 -12.19813
-17.75549 -9.185356 4.032361 -9.320414 1.339788 12.23572
 48.10678 -24.78042 8.86155 -4.958602 -4.843619 1.443337 -8.813286 0.4328361
-3.807028 0.8784758 9.743609 7.107668 9.02508
 63.17915 -21.53388 -22.33113 5.595533 -12.11316 -4.990936 -14.4953 -10.58425
2.666025 -0.3021607 -11.49867 -1.502062 3.861568
 70.48519 -19.16981 -25.84126 10.23085 -15.72831 -5.344745 -22.62867 -12.71542
0.8277165 -4.167449 -19.62204 -5.533485 2.644755
 52.99891 -16.45959 0.7519462 -4.386663 3.804989 -1.37611 -24.83507 5.490471
-3.33739 -8.404724 -17.6997 -0.2677126 5.236793
 54.01795 -19.39126 -3.082492 -1.624617 -8.421985 -11.15252 -18.0968 -11.92423
-6.684193 -11.88862 -8.570399 -3.803415 5.675081
 55.33753 -18.44497 -9.369541 -7.717715 -8.041488 -11.45842 -19.81938 -12.43418
```

-1.97697 -4.627994 -7.774594 4.451687 7.557387
55.72844 -20.32559 -12.32121 -9.614379 -2.77022 -8.572324 -14.91047 -6.382179 -7.155323 -7.767553 -17.01464 1.11917 -2.572359

同名的 archive（.ark） 和 script(.scp) 文件代表的同一部分数据，注意，这些命令行都有前缀"scp:" 或 "ark:",kaldi 不会自己判断这到底是个 script 还是 archive 形式，这需要我们加前缀告诉 kaldi 这是什么格式的文件。对于 code 而言，这两种格式对它来说都是一样的。

这两种格式都是‘表（table）’的概念。一个‘表’就是一组有序的事物，前面是识别字符串（如句子的 id），一个‘表’不是一个 c++的对象，因为对应不同的需求（写入、迭代、随机读入）我们分别有 c++对象来读入数据。

.scp 格式是 text-only 的格式，每行是个 key（一般是句子的标识符（id））后接空格，接这个句子特征数据的路径 。

.ark 格式可以是 text 或 binary 格式，（你可以写为 text 格式，命令行要加‘t’，binary 是默认的格式）文件里面数据的格式是：key（如句子的 id）空格后接数据。

下面是关于 script 和 archive 的几点说明：

用于说明如何读表的字符串叫做“rspecifier”,如 ark:gunzip -c my/dir/foo.ark.gz|

用于说明如何写入表的字符串叫做“wspecifier”,如 ark,t:foo.ark

archive 可以合并成大的 archive，仍然有效

code 可以读入这两种格式通过顺序读入或随机读入。用户级的 code 只知道它是在迭代还是查找，并不知道它接触的数据格式是什么（是 script 还是 archive）

通过随机读入（random access）来读取 archive 数据，内存使用效率较低，要想高效的随机读入 archive，那么生成时就生成对应的 ark 和 scp 文件，读入时通过 scp 文件读入。

这部分更多信息在官网（google 输入 kaldi）Kaldi I/O mechanisms 中。

训练单音子系统（monophone）

输入：

gmm-copy --binary=false exp/mono/0.mdl - | less

下面是对应的输出：

<TransitionModel>
<Topology>
<TopologyEntry>
<ForPhones>
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
</ForPhones>
<State> 0 <PdfClass> 0 <Transition> 0 0.75 <Transition> 1 0.25 </State>
<State> 1 <PdfClass> 1 <Transition> 1 0.75 <Transition> 2 0.25 </State>
<State> 2 <PdfClass> 2 <Transition> 2 0.75 <Transition> 3 0.25 </State>
<State> 3 </State>
</TopologyEntry>
<TopologyEntry>
<ForPhones>
1
</ForPhones>
<State> 0 <PdfClass> 0 <Transition> 0 0.5 <Transition> 1 0.5 </State>
<State> 1 <PdfClass> 1 <Transition> 1 0.5 <Transition> 2 0.5 </State>
<State> 2 <PdfClass> 2 <Transition> 2 0.75 <Transition> 3 0.25 </State>
<State> 3 </State>
</TopologyEntry>
</Topology>
<Triples> 84
1 0 0
1 1 1
1 2 2
2 0 3

2 1 4
2 2 5
3 0 6
3 1 7
3 2 8
4 0 9
4 1 10
4 2 11
5 0 12
5 1 13
5 2 14
6 0 15
6 1 16
6 2 17
7 0 18
7 1 19
7 2 20
8 0 21
8 1 22
8 2 23
9 0 24
9 1 25
9 2 26
10 0 27
10 1 28
10 2 29
11 0 30
11 1 31
11 2 32
12 0 33
12 1 34
12 2 35
13 0 36
13 1 37
13 2 38
14 0 39
14 1 40
14 2 41
15 0 42
15 1 43
15 2 44
16 0 45
16 1 46
16 2 47
17 0 48
17 1 49
17 2 50
18 0 51
18 1 52
18 2 53
19 0 54
19 1 55
19 2 56
20 0 57
20 1 58
20 2 59
21 0 60
21 1 61
21 2 62
22 0 63
22 1 64
22 2 65
23 0 66

23 1 67
23 2 68
24 0 69
24 1 70
24 2 71
25 0 72
25 1 73
25 2 74
26 0 75
26 1 76
26 2 77
27 0 78
27 1 79
27 2 80
28 0 81
28 1 82
28 2 83
</Triples>
<LogProbs>
[0 -0.6931472 -0.6931472 -0.6931472 -0.6931472 -0.2876821 -1.386294 -0.2876821
-1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294
-0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821
-1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294
-0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821
-1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294
-0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821
-1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294
-0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821
-1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294
-0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821
-1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294
-0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821
-1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294
-0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821
-1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294
-0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821
-1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294
-0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821
-1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294
-0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821
-1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294
-0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821
-1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294 -0.2876821 -1.386294]
</LogProbs>
</TransitionModel>
<DIMENSION> 39 <NUMPDFS> 84 <DiagGMM>
<GCONSTS> [-94.49178]
<WEIGHTS> [1]
<MEANS_INVVARS> [
-0.005838452 -0.0107621 0.007483369 0.002269829 0.01010145 0.001220717
-0.002948278 0.004102771 -0.009732663 0.005548568 -0.00846673 0.003018271
0.002561719 -0.001072273 -0.0003676935 0.0009567018 0.0004904701
0.001004559 0.0006702438 0.002065411 0.001736847 -0.0004884294
-0.0001839283 0.000573744 -6.096664e-06 0.0008038587 0.000548786
0.0005939789 -0.001607142 -0.0008620437 0.0002163016 -0.0002253224
0.0009042169 0.0007718542 0.0001247094 -0.0003084296 -0.001637235
0.0004870822 0.002509772]
<INV_VARS> [
0.002302196 0.004163655 0.005391662 0.002574274 0.003217114 0.002863562
0.005842444 0.004294651 0.003149447 0.005013018 0.004209091 0.00450796
0.006656926 0.06917789 0.07472826 0.08657464 0.05778877 0.05123304
0.05053843 0.08058306 0.07275081 0.0605317 0.07780149 0.07870268 0.08504011
0.1153897 0.4811986 0.4609138 0.5185907 0.3938675 0.3216299 0.3013106

0.4305585 0.4167711 0.3496516 0.4075994 0.4405422 0.4886224 0.6218032]
</DiagGMM>
<DiagGMM>
<GCONSTS> [-94.49178]
<WEIGHTS> [1]
<MEANS_INVVARS> [
 -0.005838452 -0.0107621 0.007483369 0.002269829 0.01010145 0.001220717
-0.002948278 0.004102771 -0.009732663 0.005548568 -0.00846673 0.003018271
0.002561719 -0.001072273 -0.0003676935 0.0009567018 0.0004904701
0.001004559 0.0006702438 0.002065411 0.001736847 -0.0004884294
-0.0001839283 0.000573744 -6.096664e-06 0.0008038587 0.000548786
0.0005939789 -0.001607142 -0.0008620437 0.0002163016 -0.0002253224
0.0009042169 0.0007718542 0.0001247094 -0.0003084296 -0.001637235
0.0004870822 0.002509772]
<INV_VARS> [
 0.002302196 0.004163655 0.005391662 0.002574274 0.003217114 0.002863562
0.005842444 0.004294651 0.003149447 0.005013018 0.004209091 0.00450796
0.006656926 0.06917789 0.07472826 0.08657464 0.05778877 0.05123304
0.05053843 0.08058306 0.07275081 0.0605317 0.07780149 0.07870268 0.08504011
0.1153897 0.4811986 0.4609138 0.5185907 0.3938675 0.3216299 0.3013106
0.4305585 0.4167711 0.3496516 0.4075994 0.4405422 0.4886224 0.6218032]
</DiagGMM>
<DiagGMM>
<GCONSTS> [-94.49178]
<WEIGHTS> [1]

首先给出的 topo 的信息，有一个 phone 和其他 phone 的 topology 不同，通过对比 phones.txt,可知这个不同的 phone 是 sil（代表 silence），topo 文件的惯例是第一个状态是初始状态（概率为 1），最后一个状态是结束状态（概率为 1）在这里面，明显，-1 是初始状态，0，1，2 是 HMM 中间的转移状态，3 是结束状态。

.mdl 文件的惯例是包含两部分信息：一部分的类型是 Transition Model（转换模型），包含拓扑信息（topo），作为 HMMtopology 的一个成员变量，另一部分是相关模型类型(叫 AmGmm),这种类型的文件不是‘表’,写入是 binary or text，取决于命令行选项 --binary=true or --binary=false，‘表’就是指 script 和 archive。

看上面个数据，会发现，0.mdl 是初始化模型，所以参数都是初始化的，这个模型训练 40 次，所以 40.mdl 中的概率是最终的参数。

以上信息更多见官网 HMM topology and transition modeling。

再提重要的一点：在 kaldi 中 p.d.f.'s 使用数字标识符表示的，从 0 开始（这些数字我们叫做 pdf-ids），在 HTK 中他们没有名字。.mdl 文件没有足够的信息能在 context-dependent phones 和 pdf-ids 间建立映射，为看这个，看 tree 文件，输入：

copy-tree --binary=false exp/mono/tree - | less

以下是输出：

ContextDependency 1 0 ToPdf TE 0 29 (NULL TE -1 3 (CE 0 CE 1 CE 2)
TE -1 3 (CE 3 CE 4 CE 5)
TE -1 3 (CE 6 CE 7 CE 8)
TE -1 3 (CE 9 CE 10 CE 11)
TE -1 3 (CE 12 CE 13 CE 14)
TE -1 3 (CE 15 CE 16 CE 17)
TE -1 3 (CE 18 CE 19 CE 20)
TE -1 3 (CE 21 CE 22 CE 23)
TE -1 3 (CE 24 CE 25 CE 26)
TE -1 3 (CE 27 CE 28 CE 29)
TE -1 3 (CE 30 CE 31 CE 32)
TE -1 3 (CE 33 CE 34 CE 35)
TE -1 3 (CE 36 CE 37 CE 38)
TE -1 3 (CE 39 CE 40 CE 41)
TE -1 3 (CE 42 CE 43 CE 44)
TE -1 3 (CE 45 CE 46 CE 47)
TE -1 3 (CE 48 CE 49 CE 50)

TE -1 3 (CE 51 CE 52 CE 53)
TE -1 3 (CE 54 CE 55 CE 56)
TE -1 3 (CE 57 CE 58 CE 59)
TE -1 3 (CE 60 CE 61 CE 62)
TE -1 3 (CE 63 CE 64 CE 65)
TE -1 3 (CE 66 CE 67 CE 68)
TE -1 3 (CE 69 CE 70 CE 71)
TE -1 3 (CE 72 CE 73 CE 74)
TE -1 3 (CE 75 CE 76 CE 77)
TE -1 3 (CE 78 CE 79 CE 80)
TE -1 3 (CE 81 CE 82 CE 83)
)
EndContextDependency

这是个 monophone 的 tree，所以非常 trivial，因为它没有任何分支，CE 是 constant eventmap(代表树的叶子们)，TE 指 table eventmap,(代表查询表之类的东东)，这里没有 SE，指 split eventmap(代表树的分支)因为这是个 monophone。“TE 0 29”是一个 table eventmap 从 key 0 开始分裂的开始（key 0 指在长度为 1（因为是 monophone）的 phone-context 向量的第 0 个音子位置）。接着，在括号内，有 29 个 event map。第一个是 NULL，代表一个指向 eventmap 的 0 指针，因为 phone-id0 是为‘epsilon’保留的。“TE -1 3 (CE 75 CE 76 CE 77)”这个字符串代表一个 table eventmap 从 key-1 开始分裂，这个 key 代表在 topo 文件中说过的 pdfclass，在这里就是 HMM 状态。这个 phone 有 3 个状态，所以分配给这个 key 的值可以取 0，1，2。在括号内是三个 constant eventmap，每个代表树的一个叶子。

现在看 exp/mono/ali.1.gz，输入：
copy-int-vector "ark:gunzip -c exp/mono/ali.1.gz|" ark,t:- | head -n 2

以下是对应的输出：
NF089001 2 1 1 1 4 3 3 6 5 5 2 1 4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 6 5 5 5 5 5 62 61 61 61 64 63 63 63 66 65 65 56 58 57 57 60 14 13 13 16 15 18 17 17 17 140 142 141 141 141 141 141 141 144 80 79 82 84 80 79 79 79 79 82 81 81 81 84 83 86 85 85 88 87 90 89 89 89 89 56 55 55 55 55 55 55 58 57 57 57 60 59 68 67 67 70 72 32 31 34 33 33 36 35 35 35 35 35 80 79 79 79 79 79 79 79 79 79 79 79 82 84 50 49 52 51 51 54 53 53 53 8 7 7 7 7 7 10 9 12 11 11 11 11 2 4 3 3 3 6 5 5 5 56 55 55 55 55 55 55 55 55 55 58 60 32 34 36 35 35 35 35 35 35 35 35 80 79 82 81 81 84 83 8 7 7 7 7 7 7 7 10 9 9 9 9 12 11 56 55 55 55 58 57 57 57 60 59 59 2 4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 6
NF089002 2 4 3 6 5 5 5 5 5 5 2 4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 6 5 5 5 5 62 61 61 61 61 61 61 64 63 66 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 140 142 144 140 142 144 143 44 43 46 45 48 47 86 85 88 87 87 90 89 89 89 89 80 82 81 84 83 56 55 55 55 58 57 57 60 59 59 59 59 20 19 19 19 22 21 24 23 23 23 56 55 55 55 55 55 55 55 55 55 58 57 57 60 59 59 59 122 121 121 121 121 121 121 121 121 121 121 124 123 123 126 125 56 58 60 59 20 19 19 19 19 22 21 21 24 23 23 56 55 55 55 55 55 58 60 62 61 61 61 61 61 61 64 66 65 65 65 65 56 55 55 55 55 55 58 57 57 57 57 60 59 59 2 1 1 1 1 1 1 4 3 3 3 3 6 5 128 130 132 131 131 131 86 85 85 85 88 87 87 87 87 87 87 87 87 87 90 89 140 142 144 68 67 67 67 67 67 67 70 72 158 157 160 162 161 161 86 85 88 87 87 87 87 87 87 87 87 87 87 90 140 142 144 44 43 43 43 46 48 47 32 34 33 36 35 35 35 35 35 35 35 35 56 55 55 58 60 59 62 61 61 61 61 61 64 66 65 140 142 144 134 133 133 133 136 135 135 135 135 138 137 137 137 140 142 144 143 143 143 143 143 143 143 143 143 2 4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 6

这是对训练数据做的维特比 alignement（对齐）。每句话是一行，再看看上面 exp/mono/tree 文件，p.d.f. id 数值最大的是 83，而这里的数值远大于这个，是因为，alignment 文件用的不是
p.d.f. id，这里用的是更细分的标识符（identifier），称作“transition-id”，这些 id 将 phone 和它们在拓扑原型结构中的转移概率也编码进来了。若想知道“transition-id”是什么，输入：
show-transitions data/lang/phones.txt exp/mono/0.mdl

以下是对应的输出：
Transition-state 1: phone = sil hmm-state = 0 pdf = 0
Transition-id = 1 p = 0.5 [self-loop]
Transition-id = 2 p = 0.5 [0 -> 1]

Transition-state 2: phone = sil hmm-state = 1 pdf = 1
Transition-id = 3 p = 0.5 [self-loop]
Transition-id = 4 p = 0.5 [1 -> 2]
Transition-state 3: phone = sil hmm-state = 2 pdf = 2
Transition-id = 5 p = 0.75 [self-loop]
Transition-id = 6 p = 0.25 [2 -> 3]
Transition-state 4: phone = a hmm-state = 0 pdf = 3
Transition-id = 7 p = 0.75 [self-loop]
Transition-id = 8 p = 0.25 [0 -> 1]
Transition-state 5: phone = a hmm-state = 1 pdf = 4
Transition-id = 9 p = 0.75 [self-loop]
Transition-id = 10 p = 0.25 [1 -> 2]
Transition-state 6: phone = a hmm-state = 2 pdf = 5
Transition-id = 11 p = 0.75 [self-loop]
Transition-id = 12 p = 0.25 [2 -> 3]
Transition-state 7: phone = b hmm-state = 0 pdf = 6
Transition-id = 13 p = 0.75 [self-loop]
Transition-id = 14 p = 0.25 [0 -> 1]
Transition-state 8: phone = b hmm-state = 1 pdf = 7
Transition-id = 15 p = 0.75 [self-loop]
Transition-id = 16 p = 0.25 [1 -> 2]
Transition-state 9: phone = b hmm-state = 2 pdf = 8
Transition-id = 17 p = 0.75 [self-loop]
Transition-id = 18 p = 0.25 [2 -> 3]
Transition-state 10: phone = ch hmm-state = 0 pdf = 9
Transition-id = 19 p = 0.75 [self-loop]
Transition-id = 20 p = 0.25 [0 -> 1]
Transition-state 11: phone = ch hmm-state = 1 pdf = 10
Transition-id = 21 p = 0.75 [self-loop]
Transition-id = 22 p = 0.25 [1 -> 2]
Transition-state 12: phone = ch hmm-state = 2 pdf = 11
Transition-id = 23 p = 0.75 [self-loop]
Transition-id = 24 p = 0.25 [2 -> 3]

.....

Transition-state 82: phone = z hmm-state = 0 pdf = 81
Transition-id = 163 p = 0.75 [self-loop]
Transition-id = 164 p = 0.25 [0 -> 1]
Transition-state 83: phone = z hmm-state = 1 pdf = 82
Transition-id = 165 p = 0.75 [self-loop]
Transition-id = 166 p = 0.25 [1 -> 2]
Transition-state 84: phone = z hmm-state = 2 pdf = 83
Transition-id = 167 p = 0.75 [self-loop]
Transition-id = 168 p = 0.25 [2 -> 3]

显然，上面这个是训练前的初始状态
为了增加可读性，输入：
show-alignments data/lang/phones.txt exp/mono/40.mdl exp/mono/40.occs | less
（.occs 文件是指 occupation counts）
以下是对应的输出：
Transition-state 1: phone = sil hmm-state = 0 pdf = 0
Transition-id = 1 p = 0.934807 count of pdf = 1.13866e+06 [self-loop]
Transition-id = 2 p = 0.0651934 count of pdf = 1.13866e+06 [0 -> 1]
Transition-state 2: phone = sil hmm-state = 1 pdf = 1
Transition-id = 3 p = 0.889584 count of pdf = 672302 [self-loop]
Transition-id = 4 p = 0.110416 count of pdf = 672302 [1 -> 2]
Transition-state 3: phone = sil hmm-state = 2 pdf = 2
Transition-id = 5 p = 0.7137 count of pdf = 259284 [self-loop]
Transition-id = 6 p = 0.2863 count of pdf = 259284 [2 -> 3]
Transition-state 4: phone = a hmm-state = 0 pdf = 3
Transition-id = 7 p = 0.713307 count of pdf = 390711 [self-loop]
Transition-id = 8 p = 0.286693 count of pdf = 390711 [0 -> 1]

Transition-state 5: phone = a hmm-state = 1 pdf = 4
Transition-id = 9 p = 0.594051 count of pdf = 275931 [self-loop]
Transition-id = 10 p = 0.405949 count of pdf = 275931 [1 -> 2]
Transition-state 6: phone = a hmm-state = 2 pdf = 5
Transition-id = 11 p = 0.594987 count of pdf = 276569 [self-loop]
Transition-id = 12 p = 0.405013 count of pdf = 276569 [2 -> 3]
Transition-state 7: phone = b hmm-state = 0 pdf = 6
Transition-id = 13 p = 0.590539 count of pdf = 19660 [self-loop]
Transition-id = 14 p = 0.409461 count of pdf = 19660 [0 -> 1]
Transition-state 8: phone = b hmm-state = 1 pdf = 7
Transition-id = 15 p = 0.417553 count of pdf = 13821 [self-loop]

这个用的是 40.mdl，得到的概率都是模型最后用的转移概率
要想了解更多关于 HMM 拓扑结构，转移标识符（transition-ids），转移模型之类的，看官网 HMM topology and transition modeling 这部分。

接下来看看训练是如何进行的，输入：
grep Overall exp/mono/log/acc.{?.? ,?.??,??.?,??.??}.log

以下是输出的最后一部分：
exp/mono/log/acc.35.10.log:LOG (gmm-acc-stats-ali:main():gmm-acc-stats-ali.cc:115)
exp/mono/log/acc.37.12.log:LOG (gmm-acc-stats-ali:main():gmm-acc-stats-ali.cc:115)
Overall avg like per frame (Gaussian only) = -99.1242 over 595815 frames.
exp/mono/log/acc.38.10.log:LOG (gmm-acc-stats-ali:main():gmm-acc-stats-ali.cc:115)
Overall avg like per frame (Gaussian only) = -99.0045 over 666753 frames.
exp/mono/log/acc.38.11.log:LOG (gmm-acc-stats-ali:main():gmm-acc-stats-ali.cc:115)
Overall avg like per frame (Gaussian only) = -95.769 over 793715 frames.
exp/mono/log/acc.38.12.log:LOG (gmm-acc-stats-ali:main():gmm-acc-stats-ali.cc:115)
Overall avg like per frame (Gaussian only) = -99.0953 over 595815 frames.
exp/mono/log/acc.39.10.log:LOG (gmm-acc-stats-ali:main():gmm-acc-stats-ali.cc:115)
Overall avg like per frame (Gaussian only) = -98.9901 over 666753 frames.
exp/mono/log/acc.39.11.log:LOG (gmm-acc-stats-ali:main():gmm-acc-stats-ali.cc:115)
Overall avg like per frame (Gaussian only) = -95.7472 over 793715 frames.
exp/mono/log/acc.39.12.log:LOG (gmm-acc-stats-ali:main():gmm-acc-stats-ali.cc:115)
Overall avg like per frame (Gaussian only) = -99.0786 over 595815 frames.

你可以看到每次迭代的声学似然概率。
就写到这儿吧。
hope it is helpful (well, it is to me),hope you like it.

4.14

Kaldi 中的 I/O 机制

本章概述 Kaldi 中的输入输出机制。

本部分文档主要面向代码级别的 I/O 机制；关于命令行级别的 I/O 机制可参见 Kaldi I/O from a command-line perspective.

The input/output style of Kaldi classes

Kaldi 中 I/O 类有一个统一的接口。标准的接口如下：

```
class SomeKaldiClass {
public:
void Read(std::istream&is, bool binary);
void Write(std::ostream&os, bool binary) const;
};
```

注意上述函数返回值为 void；错误通过异常来表示（参见 Kaldi logging and error-reporting）。参数“binary”指明对像在读写时采用二进制或文本形式。调用这些函数的代码必须知道我们是否希望以二进制或文本形式来读写对像（关于代码如何实现这些功能可参见 How Kaldi objects are stored in files）。注意在 windows 平台下打开文件时，“binary”变量没有必要设置为与二进制或文本模式完全一样；更详细的解释可参见 How the binary/text mode relates to the file open mode。

上述 Read/Write 函数还可以有一些附加可选参数。关于 Read 函数的一种常见形式为：

```
class SomeKaldiClass {
public:
void Read(std::istream&is, bool binary, bool add = false);
};
```

若 add 为真，并且当前类为非空，该 Read 函数会添加一些存储在硬盘上的信息（例如“统计量”）到当前类中。

Input/output mechanisms for fundamental types and STL types

专门介绍此部分函数的资料可参见 Low-level I/O functions 。我们提供这些函数使得读写基础类型更为方便；这些函数大多被 Kaldi 类中的 Read 和 Write 函数调用。在 Kaldi 类中，并没有强制要求使用这些函数，只要保证 Read 函数可以访问 Write 函数产生的数据即可。

这部分中最重要的函数是 ReadBasicType() 和 WriteBasicType()；它们是一种模板函数，包括了 bool, float, double, integer 多种数据类型。在 Read 和 Write 函数中使用这两个函数的例子如下：

```
// we suppose that class_member_ is of type int32.
voidSomeKaldiClass::Read(std::istream&is, bool binary) {
ReadBasicType(binary, &class_member_);
}
voidSomeKaldiClass::Write(std::ostream&is, bool binary)const{
WriteBasicType(binary, class_member_);
}
```

上面例子中，我们假设了 class_member_是 int32 类型，它是一种字节数已知的类型。在上面函数中采用类似 int 型的变量是不安全的。二进制模式下，这些函数输出的字符已包含数据的长度和符号，若数据格式不匹配，在读取时会出错。我们曾决定尝试自动转换数据格式，但最终没有实现此功能；目前，在 I/O 访问

中，用户可以使用长度确定的整型（推荐 int32）。另外浮点型是自动转换的。这主要是为了调试，在编译时你可以使用“-DKALDI_DOUBLE_PRECISION”选项，并仍然可以读取那些没有采用此选项进行输出的文件。我们的 I/O 线程中没有字节交换功能（译者注：大小端模式相关），若这对你来说使用不方便，可以使用文本格式。

此部分还有 WriteIntegerVector()and ReadIntegerVector()模板函数。它们与 WriteBasicType() and ReadBasicType()函数具有类似的形式，但它们主要用于 std::vector<I>，其中 I 是某种整型（再次强调，I 的字节长度在编译时必须是已知的，例如 int32）

一些更底层的 I/O 函数，例如：

```
voidReadToken(std::istream&is, bool binary, std::string *token);
voidWriteToken(std::ostream&os, bool binary, conststd::string & token);
```

Token 参数必须是不包含空格的非空字符串，如类似 XML 的串 "<SomeKaldiClass>" or "<SomeClassMemberName>" or "</SomeKaldiClass>"。这些函数的功能从命名上可以很直观地看出。为了便捷，我们还提供了 ExpectToken()函数，此函数类似 ReadToken() ，不同之处是：对于 ExpectToken 函数，你可以输入一个期望的字符串（当其未获取时，它会抛出一个异常）。典型的代码如下：

```
// in writing code:
WriteToken(os, binary, "<MyClassName>");
// in reading code:
ExpectToken(is, binary, "<MyClassName>");
// or, if a class has multiple forms:
std::string token;
ReadToken(is, binary, &token);
if(token == "<OptionA>") { ... }
elseif(token == "<OptionB>") { ... }
...
```

还有 WritePretty()和 ExpectPretty()函数。这两个函数比较少使用，它们的功能类似相应的 Token 函数，但它们只能在文本模式下进行读写，并且能接受任意的字符串（即允许有空格）；ReadPretty 函数还接受与期望空格有区别的那些输入。Kaldi 类中的 Read 函数从不检查文件末尾，但期望一直读取数据直到 Write 函数输出的末尾（文本模式下，留下的一些空格不读取并无关紧要）。这正是多个 Kaldi 对像可以存储在同一个文件中的原由，也是允许存档概念（The Kaldi archive format）能起作用的原因。

How Kaldi objects are stored in files

前面我们已看到，Kaldi 中的读代码需要事先知道是文本或二进制形式，但我们并不想用户必须一直跟踪文件是文本或二进制形式。为此，包含 Kaldi 对像的文件需要事先申明它是二进制或文本数据。二进制文件会以“\0B”开头；因为文本文件不可能包含“\0”，文本文件不需要头信息。若你采用标准 C++机制打开一个文件（通常不会这样做，参见 How to open files in Kaldi），在做任何操作之前，你必须小心头信息。你可以用 InitKaldiOutputStream()（也会设置文件流精度）和 InitKaldiInputStream()函数完成上述操作。

How to open files in Kaldi

假设你想加载/保存一个 Kaldi 对像，并假设该对像为语音模型（而不是经常使用的，例如语音特征；参见 The Table concept）。你可以使用 Input 类 /Output 类，示例如下：

```
{ // input.
Boolbinary_in;
Input ki(some_rxfilename, &binary_in);
my_object.Read(ki.Stream(), binary_in);
// you can have more than one object in a file:
my_other_object.Read(ki.Stream(), binary_in);
}
// output. note, "binary" is probably a command-line option.
{
Output ko(some_wxfilename, binary);
my_object.Write(ko.Stream(), binary);
}
```

上述示例中，大括号的目的是：一旦程序执行完，确保 Input 和 Output 对象离开作用域，则文件会被立即关闭。这看起来有点无意义（为什么不采用常规的 C++流？）。原因是这样我们可以支持不同类型的文件扩展名。这样做也可更容易处理错误（Input 和 Output 类会打印一个错误信息，并抛出一个异常）。注意：文件名有“rxfilename”和“wxfilename”两种类型。我们经常使用这两种文件名，它们也被用于提示编译器这些是扩展文件名。下一节将详细描述这两种扩展文件名。

Input 和 Output 类有着比上面示例中更为丰富的接口。你可以用 Open 函数打开这两个类，当然你也可以调用 Close 函数来关闭它们，而不只是使它们超过作用域来关闭。在程序出错时，这些函数返回布尔型状态，而不是像构造函数和析构函数期望的抛出异常。当不处理 Kaldi 二进制头时，Open 函数（和构造函数）也可以被调用，以防止你需要读写的是非 Kaldi 对象。你可能不需要任何额外的功能。

关于 Input and Output 两个类及相应的函数请参见"Classes for opening streams"，“rxfilenames”和“wxfilenames”见下一节。

Extended filenames: rxfilenames and wxfilenames

“rxfilename”与“wxfilename”并不是两个类；它们只是经常使用的变量名，它们的说明如下：

- “rxfilename”是一个字符串，它作为一个扩展名可以被 Input 类在读数据时解析
- “wxfilename”也是一个字符串，它作为一个扩展名可以被 Output 类在写数据时解析

“rxfilename”串可以有如下形式：

- “-”或者“”表示标准的输入；
- “some command |”表示一种输入管道命令，即，我们可以去掉管道符“|”，并通过 popen 函数将剩余的命令字符提交给 shell；
- “/some/filename:12345”表示文件中偏移，即，打开文件，并定位到文件中 12345 地址处；
- “/some/filename”与上面三种情况不符的其它任何类型均看作一个正常的文件名（当然，在打开文件之前，会识别一些很明显的错误）。

用户可以通过 ClassifyRxfilename()找到“rxfilename”使用的是什么类型，但这通常没有必要。

“wxfilename”串可以有如下形式：

- “-”或者“”表示标准的输入；
- “| some command”表示一种输出管道命令，即，我们可以去掉管道符“|”，并通过 popen 函数将剩余的命令字符提交给 shell；
- “/some/filename”与上面两种情况不符的其它任何类型均当作一个正常的文件名（也会排除一些很明显的错误）。

相应地，ClassifyWxfilename() 可以告诉我们文件名的类型。

The Table concept

Table 是一个概念，而不是实际的 C++类。Table 是由一些已知类型的对象组合成的集合，它可以通过字符串来索引。这些字符串必须是令牌形式（一个令牌定义为不包括空格的非空字符串）。典型的 Table 例子有：

- 由特征文件（格式为 Matrix<float>）组成的集合，采用 utterance id 索引；

- 由转写文本（格式为 std::vector<int32>）组成的集合，采用 utterance id 索引；
- 由约束的 MLLR 变换矩阵（格式为 Matrix<float>）组成的集合，采用 speaker id 索引。

在 Types of data that we write as tables 小节，我们将更详细地介绍这些不同类别的表；这里，我们只阐述其基本原理和内部机制。一个表存储在硬盘（也可以在管道）上有两种可能的格式：一种是 script 文件，另一种是 archive 文件（可参见 The Kaldi script-file format 和 The Kaldi archive format）。关于表的函数及类型，可参见类及相关的 "Table types and related functions"。

Table 有三种访问方式：采用 TableWriter，或者 SequentialTableReader 或者 RandomAccessTableReader（还可以通过 RandomAccessTableReaderMapped 访问，这是一种很特殊的访问方式，在后面会详细介绍）。这些都是类模板；但它们不是表中对象的模板，而是一种 Holder 类型（Holders as helpers to Table classes），这种 Holder 告诉操作 Table 的代码如何读写其中的对象。为了打开一个 Table，你必需提供一个名为“wspecifier”或“rspecifier”（Specifying Table formats: wspecifiers and rspecifiers）的字符串，这个字符串告诉操作 Table 的代码表是如何存储在硬盘上的，以及赋予代码一些不同的其它指令。我们使用下述代码来说明这情情况，代码首先读取特征，再把特征经过线性变换，最后输出。

```
    std::string feature_rspecifier = "scp:/tmp/my_orig_features.scp",
    transform_rspecifier = "ark:/tmp/transforms.ark",
    feature_wspecifier = "ark,t:/tmp/new_features.ark";
    // there are actually more convenient typedefs for the types below,
    // e.g. BaseFloatMatrixWriter, SequentialBaseFloatMatrixReader, etc.
    TableWriter<BaseFloatMatrixHolder>feature_writer(feature_wspecifier);
    SequentialTableReader<BaseFloatMatrixHolder>feature_reader(feature_rspecifie
r);
    RandomAccessTableReader<BaseFloatMatrixHolder>transform_reader(transform
_rspecifier);
    for(; !feature_reader.Done(); feature_reader.Next()) {
    std::stringutt = feature_reader.Key();
    if(transform_reader.HasKey(utt)) {
    Matrix<BaseFloat>new_feats(feature_reader.Value());
    ApplyFmllrTransform(new_feats, transform_reader.Value(utt));
    feature_writer.Write(utt, new_feats);
    }
    }
```

这样做的好处是：访问表的代码可将表看作通用的映射或者列表。数据的格式以及读过程中的其它方面（例如误差容忍度）可通过“rspecifiers”和“wspecifiers”中的选项来控制，而不必在调用代码中处理；在上述代码示例中，选项“t”指明以文本形式进行写数据。

Table 的柏拉图式设计是将字符串映射为对象的一种映射。但是，若我们并不是进行随机访问一个特定的表，并且该表包含重复的实体串时，这种代码设计并不好。

关于 Table 类型中读写 specific types 中信息可参见"Specific Table types"。

The Kaldi script-file format

Script 文件（也许命名不太准确）是一种文本文件，其中每一行通常包括如下的内容：

```
some_string_identifier /some/filename
```

Script 文件中行还有另外一种有效的表示：

```
utt_id_01001gunzip -c /usr/data/file_010001.wav.gz |
```

当读取 Script 文件中某一行时，Kaldi 会去掉前后的空格，同时也会根据第一个有效空格将文本进行拆分成两部分。第一部分是访问表的 key（例如，utterance id，上述示例中为“utt_id_01001”）。第二部分属于“xfilename”（即“wxfilename”或“rxfilename”，上述示例中为“gunzip -c /usr/data/file_010001.wav.gz |”）。Script 中不允许有空行或者空的“xfilename”。Script 文件可读、可写、或同时读写，这主要决定于“xfilename”是“rxfilenames”、“wxfilenames”、或 both。

若 Script 文件用于读，并且其中包括 Kaldi 类中一些对象，那么我们通常可能读取 Script 文件中每一行上的“xfilename”，并用 Input 来打开它（How to open files in Kaldi）。若在二进制模式下，数据流会包括二进制模式头信息“\0B”（即使在需要读取的对象位于存档文件中间）。

The Kaldi archive format

Kaldi 中 archive 格式文件十分简单。回忆一下令牌（token）定义为无空格的非空字符串。Archive 文件格式如下：

```
token1 [something]token2 [something]token3 [something] ....
```

可以描述为零个或多个重复的部分：（令牌；空格；调用 Holder 中 Write 函数获取的结果）。注意 Holder 也是一个对象，它告诉 Table 代码如何进行读写数据。

写 Kaldi 对象时，上述示例中的“[something]”部分包括二进制头信息（文本模式下没有此信息）和由调用 Write 函数生成的结果。写非 Kaldi 对象也很简单（例如 int32、float 或者 vector<int32>），在文本模式下，我们常用的 Holder 类会确保“[something]”是一个以换行符结尾的字符串。因此，Archive 文件与 Script 文件类似，每一行表示独立的实体，例如：

```
utt_id_1 5
utt_id_2 7
...
```

这是文本模式下的 archive 文件，该示中存储的是整型数据。

多个 archive 格式文件可以合并成为一个 archive，并且合并后原数据仍然有效（前提是它们是相同类型的数据）。Archive 格式也支持管道，即当你把 archive 数据放于管道之中时，在程序数据之前，程序不必等到管道结束就可以读取数据。为了有效地随机访问 archives，在写 archive 文本到硬盘时，可能也同时写入了一个包含 archive 偏移量的 script 文件。对于这一点，请参见下一节。

Specifying Table formats: wspecifiers and rspecifiers

Table 类需要一个字符串传递给它的构造函数或 Open 方法。在使用 TableWriter 时，此字符串就是“wspecifier”；而在使用 RandomAccessTableReader 或 SequentialTableReader 时，此字符串为“rspecifier”。有效的“rspecifiers”和“wspecifiers”如下：

```
std::string rspecifier1 = "scp:data/train.scp"; // script file.
std::string rspecifier2 = "ark:-"; // archive read from stdin.
// write to a gzipped text archive.
std::string wspecifier1 = "ark,t:| gzip -c > /some/dir/foo.ark.gz";
std::string wspecifier2 = "ark,scp:data/my.ark,data/my.scp";
```

一般情况下，“rspecifiers”或“wspecifiers”包括逗号分隔符，无序的选项（一个或两个字母），以及字符串“ark”或“scp”，接着是一个冒号，然后是一个“rxfilename”或者“wxfilename”。冒号之前的选项的顺序无关紧要。

Writing an archive and a script file simultaneously

“wspecifiers”有一种特殊情况，即冒号之前“ark,scp”可同时存在，冒号之后，有一个写 archive 的“wxfilename”，接着一个逗号，然后是一个写 script 文件的“wxfilename”。例如

```
"ark,scp:/some/dir/foo.ark,/some/dir/foo.scp"
```

上述示例中，会输出一个 archive 文件及一个 script 文件，其中 script 中每行形式如“utt_id /somedir/foo.ark:1234”，指出了 archive 文件中数据的偏移量，这使得随机访问时更高效。用户可以有针对性地处理 script 文件，也可以将 script 拆分为多个片段，各片段的功能与其它 script 文件并无差别。注意：虽然冒号之

前选项的顺序通常情况下无关紧要，但“ark,scp”这种情况下，“ark”必须在“scp”之前，以防止混淆冒号之后的两个“wxfilenames”（一般 archive 在前）。Archive 的“wxfilename”应该是正常的文件名，否则生成的 script 文件可能不能被 Kaldi 正常读取，但代码不能保证这一点（译者注：未检查文件名）。

Valid options for wspecifiers

关于“wspecifier”，有效的选项有：

- “b”（binary）表示以二进制格式进行写数据（默认选项）；
- “t”（text）表示以文本格式进行写数据；
- “f”（flush）表示每次写操作后清空 stream；
- “nf”（no-flush）表示每次写操作后不清空 stream（当前无意义的，可调用代码更改默认选项）
- “p”（permissive）表示许可模式，对“scp：”的“wspecifiers”有作用，在 script 文件中缺失某些实体时，选项“p”使得程序对于这些文件不输出任何信息，也不会输出错误。

“wspecifiers”示例如下：

```
"ark,t,f:data/my.ark"
"ark,scp,t,f:data/my.ark,|gzip -c > data/my.scp.gz"
```

Valid options for rspecifiers

在看下面的选项时，记住程序在读取 archives 时不会执行 seek 操作，以防 archive 实际上是一个管道（这种情况经常使用）。若 RandomAccessTableReader 正在读取 archive 文件，程序可能要在内存中存储许多对象以防止后续会重复使用它们，或者当某个 key 并不存在 archive 文件之中时，程序也可能要搜索到 archive 文件末尾。下面的一些选项可用于避免这种情况：

关于“rspecifier”，较为重要的选项有：

- “o”（once）：是用户使用 RandomAccessTableReader 代码的一种方式，其中每个 key 将只被查询一次。这个策略将阻止已经读取的对象一直保存在内存中（以防后续会再次使用）。
- “p”（permissive）表示程序会忽略错误，并只提供有效的数据；无效的数据被看作不存在。在 Script 文件中，该选项意味着对于 HasKey()的查询会强制加载相应的文件，因此若文件有问题，程序是可以知道返回错误的。在 Archive 文件中，若 archive 被破坏或不完整，该选项也会阻止此类异常（程度只是停止读取有异常的文件点）。

- “s”（sorted）指明读取 archive 中的 keys 是有序的。对于 RandomAccessTableReader，当 HashKey 函数被调用并用于判断某个 key 是否存在 archive 中时，一量程序遇到一个更大的 key，就会返回错误，而不会一直读到文件末尾。
- “cs”（called-sorted）指明调用 HasKey() 和 Value()的代码是有序的。因此，若这些函数被调用，读取代码可以丢弃那些更小的 keys 对应的对象。这可以节约内存。在效率方面，“cs”代表了用户的思维，即一些程轮流使用的 archive 必须是有序列的。

若用户提供了任何错误的选项，例如：对于一个未排序的 archive 的文件使用了“s”选项，RandomAccessTableReader 代码会尽最大努力去检测这种错误或意外。

我们也为“rspecifier”提供了类似“wspecifiers”的选项，但目前这些选项用处并不大。

- “no”（not-once）与选项“o”功能相反（当前，此选项无效）；
- “np”（not-permissive）与选项“p”功能相反（当前，此选项无效）；
- “ns”（not-sorted）与选项“s”功能相反（当前，此选项无效）；
- “ncs”（not-called-sorted) 与选项“cs”功能相反（当前，此选项无效）；
- “b”（binary）并不做任何事情，只是为了脚本方便；
- “t”（text）并不做任何事情，只是为了脚本方便。

“rspecifiers”常用的例子如：

```
"ark:o,s,cs:-"
"scp,p:data/my.scp"
```

Holders as helpers to Table classes

正如前面提到，Table 类，即 TableWriter、RandomAccessTableReader 和 SequentialTableReader，它们可表示为同一个 Holder。Holder 并不是实际的类或基类，但它可用于描述同一类别的类，并以 Holder 后缀来命名，例如：TokenHolder 或 KaldiObjectHolder（这是一个通用的 Holder，可作为许多类的模板，只要这些类满足在 The input/output style of Kaldi classes 小节描述的 Kaldi I/O 类型。我们实现了类模板 GenericHolder，但并不打算使用它，主要是为了说明 Holder 类必须满足的基本属性。

The type of the class "held" by the Holder class is a typedef Holder::T（其中 Holder 是实际问题中 Holder 类的名字）。可用的 Holder 类型列表可参见"Holder types"。

How the binary/text mode relates to the file open mode

此节内容只与 Windows 平台相关。一般的规则是：在写数据时，文件模式需要与 Write 函数中的“binary”参数匹配；而在读二进制数据时，文件本身通常是二进制格式的，但当我们读文本数据时，文件模式可能是二进制或文本形式（因此文本模式下的读函数需要接受 Window 插入的额外“\r”字符）。我们并不总是知道文件是文本或二进制格式，直至我们打开该文件；所以当不确定时，我们采用二进制模式。

Avoiding memory bloat when reading archives in random-access mode

通过 Table 代码随机读取较大的 archives 时，可能会造成内存溢出。这种潜在的风险可能发生在以 RandomAccessTableReader<SomeHolder>读取 archive 时。Table 代码如此实现主要是为了首先保证正确性，所以当随机读取 archive 时，除非特别说明（后面会讨论），否则是不会将读进内存的对象抛弃的（以防止后续需要使用）。这里有一个明显的问题：为什么 Table 代码不只跟踪对象在文件中的位置，并只在需要的时候利用 fseek()定位然后再读取它呢？我们没有实现这个功能，主要原因是：fseek()只能用于实际文件的读取时的定位（不能用于管道命令或者标准输入）。若 archive 是磁盘上的真实文件，用户在生成该文件时，通常会生成一个附加的 script 文件（使用类似“ark,scp:”的命令行），该项 script 包含了对象在文件中的偏移量（参见 Writing an archive and a script file simultaneously），当程序需要读取 archive 时，只需要利用 script 文件即可进行随机访问。这是最直接的高效的读取方式，因为读取 script 的代码足够智能地避免了重新打开文件时不必要的 fseek 操作。因此，archives 作为一种特殊情况，缓存偏移量到内存中解决不了任何问题。

随机访问模式读取 archive 时还有另外两个问题；当只使用“ark:”而不附带任何其它选时，这两个问题可能同时发生。

- 若在 archive 中查询一个不存在的 key，代码会强制地读取到文件末尾，以确定该 key 确实不存在。
- 每读取一个对象时，都会保存在内存之中，以防止后续再次使用。
-

关于第一个问题（必须读到文件末尾），可以通过申明 archive 是有序的来避免（采用“C”语言中正常的字符串排序方式即可，若使用了“export LC_ALL=C”则“sort”有效）。在 Kaldi 中，在读取 archive 时，可用“s”选项来达到上述目的，例如：rspecifier“ark,s:-”表明代码读入标准输入作为 archive，并期望它是有序的。Table 代码会检查用户申请的规则是否正确，若不正确则程序会崩溃。当然，你必须以这样一种方式（archive 是以某个 key 排序的）配置你的脚本（通常这在特征提取阶段完成）。

关于第二个问题（将读取的数据一直保存在内存中，以防后续再次使用），有两种解决办法：

- 第一种方法是采用“once”选项，但这种方法比较脆弱；例如，repecifier“ark,o:-”从标注输入中读对象，并申明对每个读取的对象只使用一次。若这样申明，你应该了解程序的工作流程（译者注：如在何时需要这些数据），并且有必要了解其它的该程序读取的其它 Table 中不包含重复的 keys（Table 若只在顺序模式下访问，是可以有重复的 key 的）。

  若采用了“o”选项，则 Table 在被访问之后就可以被释放。然而，这通常只能在 archive 是完全同步或者不缺失任何数据的情况下才能正常执行。例如，假设你执行如下命令：

  ```
  some-program ark:somedir/some.ark "ark,o:some command|"
  ```

  “some-program”首先迭代地顺序访问“somedir/some.ark”，对于接下来遇到的每一个 key，均通过随机访问的形式访问第二个 archive。注意：命令行中参数的顺序不是任意的，我们采用的惯例是：顺序访问的 rspecifiers 在前，而随机访问的在后。

  假设两个 archives 几乎是同步的，但可能存在 gaps（例如：特征提取或是数据对齐过程造成的 key 缺失）。当任何时刻，第一个 archive 中存在 gap 时，程序会从第二个 archive 中缓存相应的对象，因为我们不确定后续该对象是否会被再次使用（it can only throw away an object once you have read it）。第二个 archive 中的 Gaps 会带来更严重的后果，当即使是有一个 element gap，程序读取其相应 key 时，会一直查找直到第二个 archives 末尾，查找过程会缓存所有读取的对象。

- 第二种方法更为稳健，采用“called-sorted”（cs）选项。这个申明表示需要的数据对象是的有序列排列的，当然也需要熟悉程序工作机制，以及顺序访问的 archives 必须是有序的。“cs”选项通常与“s”选项联合使用。假设执行如下命令：

  ```
  some-program ark:somedir/some.ark "ark,s,cs:some command|"
  ```

  假设两个 archives 均是有序的，程序按顺序访问第一个 archive，而随机访问第二个。这样对 gaps 较为稳健。想象一下当第一个 archive 中有 gaps 时（例如 keys 为 001, 002, 003, 081, 082, ...），此时 key 003 后则是 key 081，读取第二个 archive 的程序会遇到 keys 004、005 等，但是它可以丢弃这些相关的对象，因为它知道在 key 081 之前的 key 不会再被访问（得益于“cs”选项）。若第二个 archive 中有 gap，可以申明该 archive 是有序的来防止搜索到文件末尾（这是“s”选项的任务） 。

为了提炼一种特定的反复出现在许多程序中的代码模式，我们使用了模板 RandomAccessTableReaderMapped。与 RandomAccessTableReader 模板不同，这需要两个初始的参数，即：

```
std::string rspecifier, utt2spk_map_rspecifier; // get these from somewhere.
RandomAccessTableReaderMapped<BaseFloatMatrixHolder>transform_reader(r
specifier, utt2spk_map_rspecifier);
```

若“utt2spk_map_rspecifier”为空，则其功能与标准的 RandomAccessTableReader 类似。 若其非空，例如：“ark:data/train/utt2spk”，则会从相应的地址中读取“utterance-to-speaker”映射。每当查询“utt1”时，将使用此映射表将 utterance-id 转换为 speaker-id（如 spk1），并从 rspecifier 指定的表中查找 speaker-id 相应的数据。“utterance-to-speaker”映射也是 archive 形式，故 Table 代码是读取该映射最为便捷的方式。

4.15 Kaldi 中的 I/O 可视化

本章借助命令行工具从用户可视化角度介绍 Kaldi 中的 I/O 机制，更多代码级别的概述可参见 Kaldi I/O mechanisms。

Non-table I/O

我们首先描述“non-table”接口，这是指包含一个或两个文件或数据流（例如声学模型文件，变换矩阵），并不是指那些由字符串索引的对象集合。

- Kaldi 中的文件格式默认是二进制模式，但可以使用“-binary=false”来输出非二进制文件。
- Kaldi 许多对象均有相应的“copy”函数，例如“copy-matrix”、“gmm-copy”，均可以使用“-binary=false”来将数据转换成文本形式，例如“copy-matrix --binary=false foo.mat -”。
- 通常上，磁盘上的文件和内存中的 C++对象是一一对应的，如：float 型矩阵；但是一些文件也可能包含多个对象（例如声学模型文件，它包含“TransitionModel”对象和声学模型对象）。
- Kaldi 程序通常事先知道需要读取哪一类型对象，而不是从数据流中计算出数据类型。
- 与“perl”类似，文件名可以被“-”替代（“-”表示标准输入输出）；也可以被字符串（如“|gzip -c >foo.gz" or "gunzip -c foo.gz|”）来替代。
- 读文件时，我们支持命令如“foo:1045”，表示从 foo 文件中偏移 1045 的地址开始读取。
- 为了与扩展名概念相对应，我们一般用特定术语“rxfilename”（字符串）来表示将要读取的数据流（例如，一个文件、数据流、标准输入），对于输出流，使用字符串“wxfilename”。更详细的介绍可参见 Extended filenames: rxfilenames and wxfilenames。

为进一步阐述上述概念，在设置好“$KALDI_ROOT/src/bin”路径后，可输入以下命令来直观地了解：

```
echo '[ 0 1 ]' | copy-matrix - -
```

会输出一个 log 信息，以及矩阵中的二进制数据信息。

```
echo '[ 0 1 ]' | copy-matrix --binary=false - -
```

输出为如下形式：

```
# copy-matrix --binary=false - -
copy-matrix --binary=false - -
 [
  0 1 ]
LOG (copy-matrix:main():copy-matrix.cc:68) Copied matrix to -
```

虽然矩阵数据和 log 信息是混在一起的，但 log 信息属于标准错误，并不会传递到管道中。用户可以在命令行中加入“2>/dev/null”（将 stderr 指向为/dev/null）来避免输出 log 信息。

Kaldi 程序可以通过管道连接，也可以通过 Kaldi I/O 接口中的流文件机制连接。管道连接示例：

```
echo '[ 0 1 ]' | copy-matrix - - | copy-matrix --binary=false - -
```

这会输出一个文本形式的矩阵（第一个“copy-matrix”命令将数据转换成二进制，第二个则转换成文本形式）。当然，你也可以通过如下命令来完成同样的操作：

```
copy-matrix 'echo [ 0 1 ]|' '|copy-matrix --binary=false - -'
```

这里，没有理由这样做，但是某些情况下十分有用，例如：“stdin”“stdout”均被占用，并且程序需要多个输入输出时。这对于表的访问特别有用（请参见下一节）。

Table I/O

Kaldi 中有特殊的 I/O 机制来处理由对象组成的集合，其中对象由字符串索引。这种例子如：由 utterance-ids 索引的特征矩阵，或者由 speaker-ids 索引的说话人自适应变换矩阵。用作索引的字符串必须是非空的，且不包含空格的。更深入的讨论可参见 The Table concept。

Table 有两种形式：“archive”和“script”文件。它们之间的差别在于：archive 包含实际的数据，而 script 文件则指出数据的具体位置。

从 Tabla 中读数据的程序需要一个“rspecifier”输入，该字符串指明了如何读取索引数据；而写数据到 Table 中的程序需要一个“wspecifier”输入，该字符串指示如何写数据。这两个字符串指出是否需要 scritpt 或 archive 文件，文件位置，以及不同的选项。常用的“rspecifiers”有“ark:-”，表示从标准的输入中读取 archive 数据；而“scp:foo.scp”表示 script 文件“foo.scp”会告诉我们从哪里去读取数据。下面几点需要牢记：

- 冒号后面的部分被解释为“wxfilename”或“rxfilename”（与 Non-table I/O 中一样），这意味着管道和标准的输入/输出均可支持。
- 一个 Table 通常只包含一种类型的对象（例如浮点型矩阵）
- 你可以查看“rspecifiers”和“wspecifiers”中的选项，主要有：
    - 在“rspecifiers”中，“ark,s,cs:-”表示当我们读数据时，我们希望 keys 是有序的（s），以及我们会按顺序访问数据（cs），若这些条件不满足时，程序会崩溃。这些保证了 Kaldi 中高效的随机访问，而不占用大量的内存。
    - 对于规模较少并且不太方便进行排序列的数据（例如：说话人自适应变换矩阵），若省略掉 s, cs 参数，几乎没有损害。
    - 当 Kaldi 程序有多个“rspecifiers”输入时，程序会轮流访问相应的对象；其中第一个使用顺序访问，而后面的采用随机访问，因此，第一个“rspecifier”不需要“s, cs”选项。

    - 在访问 script 文件时，例如“p:foo.scp”，其中“p”意味着当我们需要的文件不存在时，程序不会崩溃（对于 archive，当期望的文件被破坏或被截断时，“p”也地阻止程序崩溃）。
    - 写数据时，选项“t”表示文本模式，例如：“ark,t:-”。“-binary”命令行选项对 archives 无效。
- Script 文件每一行的格式为：“<key><rspecifier|wspecifier>”，例如：“utt1 /foo/bar/utt1.mat”。“rspecifier”和“wspecifier”中是允许包含空格的，例如：“utt1 gunzip -c /foo/bar/utt1.mat.gz|”。
- Archive 文件格式为：
  <key1><object1><newline><key2><object2><newline> ...
- 多个 archives 可以拼接在一起并仍然有效，但此时需要注意拼接的顺序，例如：若需要有序的拼接时，尽量避免类似“cat a/b/*.ark”的命令。
- Script 文件也可用于输出（这个功能并不常用），例如：当“wspecifier”为“scp:foo.scp”，程序先尝试写入关键字段“utt1”，并在“foo.scp”查找类似“utt1 some_file”的行，然后将数据写入“some_file”中。若未查找到相应的“utt1 some_file”，则程序会崩溃。
- 可以同时输出 archive 和 script 文件，例如：“ark,scp:foo.ark,foo.scp”。输出的 script 文件中每一行的格式类似“utt1 foo.ark:1016”（数字 1016 表示偏移量）。这种设置有利于随机访问，因为我们并不想生成许多小文件。
- Archive 机制中处理单一文件有许多技巧，例如：

```
echo '[ 0 1 ]' | copy-matrix 'scp:echo foo -|' 'scp,t:echo foo -|'
```

这值得进一步解释。首先，“scp:echo foo -|”等价于“scp:bar.scp”（bar.scp 文件中只包含一行“foo -”数据），这表示从标准输入中根据“foo”读取对象。相似地，对于“scp,t:echo foo -|”，将“foo”相关的数据写入标准输出。这一技巧也不能滥用。上述示例中，就没有必要这样使用，因为“copy-matrix”程序支持“non-table”读写，可以直接写成“copy-matrix --”。若你想经常使用这种技巧，最好修改相关的代码。

- 若想从 archive 提取某一个数据，可以在“wspecifier”的“scp:”中使用“p”选项，这只会输出用户想要的数据，并忽略“scp:”中的其它数据。例如：你想要的数据关键字段为“foo_bar”，并且“some_archive.ark”文件中包含该关键字段的相关数据，你可以这样提取：

```
copy-matrix 'ark:some_archive.ark' 'scp,t,p:echofoo_bar -|'
```

- 在某些特定情况下，与 archive 读数据相关的代码允许有限的数据类型之间的转换，例如 float 和 double 型矩阵，Lattice 和 CompactLattice 型词图之间可相互转换。

Utterance-to-speaker and speaker-to-utterance maps.

Kaldi 中许多程序使用“utterance-to-speaker”和“speaker-to-utterances”映射，这类文件命名为“utt2spk”和“spk2utt”。它们通常由命令选项“–utt2spk”和“–spk2utt”指定。“utt2spk”映射文件的格式如下：

```
utt1 spk_of_utt1
utt2 spk_of_utt2
...
```

而“spk2utt”映射的格式为：

```
spk1 utt1_of_spk1 utt2_of_spk1 utt3_of_spk1
spk2 utt1_of_spk2 utt2_of_spk2
...
```

这些文件可用于说话人自适应，例如：对于一句“utterance”，可以找到对应的“speaker”。出于以下两个原由：1）Kaldi 中示例脚本的配置；2）我们将数据划分为多个片段，所以保证“utterance-to-speaker”映射中的 speakers 是有序的是非常重要的（参见 Data preparation）。无论如何，这些文件实际上可看作为 archives，因此你会经常看到如下的一些命令行“–utt2spk=ark:data/train/utt2spk”。你也会发现这些文件符合通用的 archive 格式：“<key1><data><newline><key2><data><newline>”，只是在上述示例中数据是文本形式。在代码级别，“utt2spk”文件被当作一个包括一个字符串的表；而“spk2utt”文件则被看成包含字符串列表的表。

5 附录

5.1 kaldi 上搭建 TIMIT 基线系统

因工作需要，要搭建几个基于 kaldi 的语音识别系统，也正在学习 kaldi，把过程和经验同大家分享，希望互相帮助共同学习，欢迎与我联系，若需转载请注明出处。

注：kaldi 自带的示例提供了实验流程，但是其 monophone 的 HMM-GMM 基线是 39 phones 的，而后的 DNN 基线是 triphone 的，和我们目前需要的 61 monophone 不一致，故需要重新搭建，本文档的基线系统包括 61 monophone 的 HMM-GMM 系统和 61 monophone，183 states 的 DNN 基线，包括 fMLLR 等改进技术可以参见其它相关示例程序加入。

一. Kaldi 下搭建 TIMIT 数据集的 HMM-GMM 基线系统

1. 复制 egs/timit/s5 到同级目录下并改名为 61monophone，方便后面直接在脚本里面改动，可以在 run.sh 里面修改或者新建一个 61monophone_run.sh。
2.
```
. ./cmd.sh
```
本机运行选择 run it locally… 注释掉其余部分
```
. ./path.sh
```
3. Data & Lexicon & Language Preparation
```
timit=/home/sfxue/data/TIMIT
local/timit_data_prep.sh $ timit
```
将会新产生两个目录/data/local/data 和/data/local/nist_lm 后者用不到不去管它，这个脚本主要是对数据集的结构、数据集完整性进行检查然后生成一些列表和文件，示例本身是要将 61 phone 的标注列表转成 39 phone 的，而我们需要的就是 61 phone 的，所以要对其中的那条转换命令进行修改，修改后的命令为：
```
cat ${x}.trans | $local/timit_norm_trans.pl -i - -m $conf/phones.61-61-61.map -to 61 | sort > $x.txt || exit 1;
```
phones.61-61-61.map 文件是 phones.60-48-39.map 修改得来，我们的目的只是将 h#转换成 sil，而 q 保留，同时 timit_norm_trans.pl 也需要稍作修改来让命令可以执行。打开/data/local/data 里面的文件看看，应该可以理解它们的功能。
```
local/timit_prepare_dict.sh
```
示例中的这段注释说的的很明显了，打开这几个文件看看就行了：
```
# The parts of the output of this that will be needed are
# [in data/local/dict/ ]
# lexicon.txt
# extra_questions.txt
# nonsilence_phones.txt
# optional_silence.txt
# silence_phones.txt

utils/prepare_lang.sh --position-dependent-phones false --num-sil-states 3 data/local/dict "sil" data/local/lang_tmp data/lang
local/timit_format_data.sh
```
4. MFCC Feature Extraction & CMVN
```
# Now make MFCC features.
mfccdir=mfcc
use_pitch=false
use_ffv=false
for x in train dev test; do
steps/make_mfcc.sh --cmd "$train_cmd" --nj 10 data/$x exp/make_mfcc/$x $mfccdir
    || exit 1;
steps/compute_cmvn_stats.sh data/$x exp/make_mfcc/$x $mfccdir
done
```
生成 mfcc 特征的命令是 compute-mfcc-feats，默认 chanel 是 13，特征在此处没有做差分，后面在每个实验中可以选择差分操作。--nj 10 是并行任务数，后面

三个参数分别是数据目录，exp 目录和特征存储目录，默认压缩存储所以占用空间不大。CMVN 是句子级的。

4. Monophone Training

steps/train_mono.sh --nj "$train_nj" --cmd "$train_cmd" data/train data/lang exp/mono || exit 1;

utils/mkgraph.sh --mono data/lang_test_bg exp/mono exp/mono/graph || exit 1;

该步进行 MLE 的训练，首先要弄清楚模型结构，可以用 gmm-copy –binary=false exp/mono/0.mdl 0.model.txt 来讲模型文件转成可阅读的 txt 文件，浏览里面的结构可以看到和 HTK 的差异，具体可以看网站上的指南文档。

Kaldi 的 MLE 训练中每个状态的混合高斯数目是不一样的，在 train_mono.sh 中有几个比较关键的参数设定：

num_iters= # Number of iterations of training

max_iter_inc= # Last iter to increase #Gauss on. (最后涨高斯的一次迭代)

totgauss= # Target #Gaussians. (训练最终希望达到的高斯总数)

realign_iters= (re-alignment 分别在哪些迭代进行)

6.decode

steps/decode.sh --nj "$decode_nj" --cmd "$decode_cmd" exp/mono/graph data/test exp/mono/decode_test || exit 1;

解码本身没有什么需要修改，只是在 score 的时候需要注意我们要将 61 monophone 映射到 39 monophone 进行。在 sore.sh 文件里修改两处：

phonemap="conf/decodephones.61-48-39.map"

cat $data/text | sed 's:::g' | sed 's:::g' | local/decodetimit_norm_trans.pl -i - -m conf/decodephones.61-48-39.map -to 39 | sort > $dir/scoring/test_filt.txt

decodephones.61-48-39.map 是修改自 phones.60-48-39.map，只是将 h#换成 sil。decodetimit_norm_trans.pl 修改自 timit_norm_trans.pl 只是保留了丢掉 q 的操作。PER 识别结果 28.11。

二． Kaldi 下搭建 TIMIT 数据集的 DNN 基线

流程主要包括三个部分 force alignment，pre-train 和 fine-tune。

1. Force Alignment

traindata_dir=data/train

language_dir=data/lang

src_dir=exp/mono

ali_dir=exp/mono_ali

#steps/align_si.sh --nj 10 --cmd $train_cmd $traindata_dir $language_dir $src_dir $ali_dir

2. pre-train

Pre-train the DBN

dir=exp/pretrain-dbn

(tail --pid=$$ -F $dir/_pretrain_dbn.log 2>/dev/null)&

$cuda_cmd $dir/_pretrain_dbn.log \

steps/pretrain_dbn.sh --nn_depth 6 --delta_order 2 --hid-dim 1024 --rbm-iter 15 data/train $dir

kaldi 的 pre-train 使用一个初始学习速率，然后随着迭代递减，同时 momentum 从 0.5 涨到 0.9，我目前使用默认的 B-B 0.4，G-B 0.01 参数，效果还可以。迭代次数 G-B 是 B-B 的两倍，也就是 30,15,15,……。--delta_order 2 表示使用二阶差分，kaldi 提供的脚本里有个 bug，不能够使用差分选项，否则 CMVN 时会出错，找到出错的地方仿照 train_net.sh 修改即可。

3. fine-tune

```
dir=data/train
# split the data : 90% train 10% cross-validation (held-out)
 utils/subset_data_dir_tr_cv.sh $dir ${dir}_tr90 ${dir}_cv10

# Train the MLP
dir=exp/pretrain-dbn_dnn
ali=exp/mono_ali
feature_transform=exp/pretrain-dbn/final.feature_transform
dbn=exp/pretrain-dbn/6.dbn
#with pretrain
(tail --pid=$$ -F $dir/_train_nnet.log 2>/dev/null)&
$cuda_cmd $dir/_train_nnet.log \
steps/train_nnet.sh --feature-transform $feature_transform --delta_order 2 --dbn
$dbn --hid-layers 0 --learn-rate 0.008 \
data/train_tr90 data/train_cv10 data/lang $ali $ali $dir || exit 1;
## decode (reuse HCLG graph)
steps/decode_nnet.sh --nj 10 --cmd $decode_cmd --config conf/decode_dnn.config
--acwt 0.1 \
exp/mono/graph data/test $dir/decode || exit 1;
```

--delta_order 2 表示使用二阶差分， --dbn $dbn 表示使用之前的 dbn 初始化 DNN 除最后一层的网络，--hid-layers 0 要特别注意指的是增加一个隐层数为 0，也即输入输出共两层的网络，也就是我们一般加在 dbn 之后的随机层，如果没有之前的--dbn $dbn 选项的话要训练一个 6 隐层的网络这里也需要设成 6，个人感觉 kaldi 在这个参数的设计上不够明智，很容易让人出错。--learn-rate 0.008 表示初始学习速率而 0.008。

最终的识别结果：22.95

5.2 kaldi 里的 voxforge

Some weeks ago there was a question on the Kaldi's mailing list about the possibility of creating a Kaldi recipe using VoxForge's data. For those not familiar with it, VoxForge is a project, which has the goal of collecting speech data for various languages, that can be used for training acoustic models for automatic speech recognition. The project is founded and maintained, to the best of my knowledge, by Ken MacLean and thrives thanks to the generous contributions of great number of

volunteers, who record sample utterances using the Java applet available on the website, or submit pre-recorded data. As far as I know this is the largest body of free(in both of the usual senses of the word) speech data, readily available for acoustic model training. It seemed like a good idea to develop a Kaldi recipe, that can be used by people who want to try the toolkit, but don't have access to the commercial corpora. My previous recipe, based on freely available features for a subset of RM data can be also used for that purpose, but it has somewhat limited functionality. This post describes the data preparation steps, specific to VoxForge's data.

Prerequisites

As usual the following instructions assume you are using Linux operating system. The scripts try to install some of the external tools needed for their work, but also assume you have some tools and libraries already installed. These are things that should be either already available on your system or can be easily installed through its package manager. I will try to enumerate the main dependencies when describing the respective part of the recipe making use of them.

If you don't have Kaldi installed you need to download and install it by following the steps in the documentation. If you already have it installed you need to make sure you have a recent version, which includes this recipe. It can be found in egs/voxforge/s5 under Kaldi's installation directory. From this point on all paths in this blog entry will be relative to this directory, unless otherwise noted.

Downloading the data

Before doing anything else you should set DATA_ROOT variable in ./path.sh to point to a directory residing on a drive, which has enough free space to store several tens of gigabytes of data (about 25GB should be enough in the default recipe config at the time of writing).

You can use the ./getdata.sh to download VoxForge's data. As many other parts of the recipe it hasn't been extensively tested, but hopefully will work. It assumes you have wget installed on your system and downloads the 16KHz versions of the data archives to ${DATA_ROOT}/tgz and extracts them to ${DATA_ROOT}/extracted. If you want to free several gigabytes by deleting the archives after the extraction finishes, you can add a "--deltgz true" parameter:

view sourceprint?

```
./getdata.sh --deltgz true
```

Configuration and starting the recipe

It's usually recommended to run Kaldi recipes by hand by copy/pasting the commands in the respective run.sh scripts. If you do this be sure to source path.sh first in order to set the paths and LC_ALL=C variable. If this environment variable is not set you may run into sometimes hard to diagnose problems, due to the different sorted orders used in other locales. Or if you are like me and prefer to just start /bin/bash run.sh to execute all steps automatically, you can do this too but you may want to modify several variables first. If you happen to have a cluster of machines with Sun Grid

Engine installed you may want to modify the train and decode commands in cmd.sh to match your config. A related parameter in run.sh is njobs, which defines the maximum number of parallel processes to be executed. The current default is 2 which is perhaps too low even for a relative new home desktop machine.
There are several parameters toward the start of run.sh script, that I will explain when discussing the parts of the recipe they affect.
The recipe is structured according to the currently recommended("s5") Kaldi script style.
The main sub-directories used are:

- local/ - hosts scripts that are specific to each recipe. This is mostly data normalization stuff, which is taking the information from the particular speech database and is transforming it into the files/data structures that the subsequent steps expect. The work of using new data with Kaldi, including this recipe, involves writing and modifying scripts that fall into this category.
- conf/ - small configuration files, specifying things like e.g. feature extraction parameters and decoding beams.
- steps/ - scripts implementing various acoustic model training methods, mostly through calling Kaldi's binary tools and the scripts in "utils/"
- utils/ - scripts performing small, low-level task, like adding disambiguation symbols to lexicons, converting between symbolic and integer representation of words etc.
- data/ - This is where various metadata, produced at the recipe run time is stored. Most of it is result from the work of the scripts in local/
- exp/ - The most important output of the recipe goes there. This includes acoustic models and recognition results.
- tools/ - A non-standard directory, specific to this recipe, that I am using to put various external tools into(more about these later).

Note: Keep in mind that steps/ and utils/ are shared between all "s5" scripts and are just symlinks pointing to egs/wsj/s5. That means you should be careful if you make changes there, as this may affect the other "s5" recipes.

Data subset selection
The recipe has the option to train and test on a subset of the VoxForge's data. For (almost) every submission directory there is a etc/README file, with meta-information like pronunciation dialect and gender of the speaker. For English there are many pronunciation dialects, but from an (admittedly very limited) sample of the data I've got the impression that some of the recordings with dialect set to say "European English" and "Indian English" sound distinctively non-native. So you have the option to select a subset of the dialects if you like. In my limited experience it seems, that the speech tagged with "American English" has relative lower percentage of non-native speech intermixed. The default for the recipe is currently set to produce an acoustic model with good coverage over (presumably) native English speakers:

view sourceprint?

dialects="((American)|(British)|(Australia)|(Zealand))"

The local/voxforge_select.sh script creates symbolic links to the matching submission subdirectories in ${DATA_ROOT}/selected and the subsequent steps work just on this data.

If you want to select all of VoxForge's English speech you should perhaps set this to:

view sourceprint?

dialects="English"

I don't have experience with the non-English audio at VoxForge, so I can't comment how the scripts should be modified to work with it.

Mapping anonymous speakers to unique IDs

VoxForge allows for anonymous speaker registration and speech submission. This is not ideal from Kaldi's scripts viewpoint, because they perform various speaker-dependent transforms. The "anonymous" speech is recorded under different environment/channel conditions (microphones used, background noise etc), by speakers that may be both males and females and have different accents. So instead of lumping all this data together I decided to give each speaker unique identity. The script local/voxforge_fix_data.sh renames all "anonymous" speakers to "anonDDDD"(D is a decimal digit), based on the submission date. This is not entirely precise of course because it may give two different IDs to the same speaker who made recordings on two different dates, and also give the same ID to two or more different "anonymous" speakers who happened to submit speech on the same date.

Train/test set splitting and normalizing the metadata

The next steps is to split the data into train and test sets and to produce the relevant transcription and speaker-dependent information. These steps are performed by a rather convoluted and not particularly efficient script called local/voxforge_data_prep.sh. The number of speakers to be assigned to the test set is defined in run.sh. The actual test set speakers are chosen at random. This is probably not an ideal arrangement, because the speakers will be different each time voxforge_data_prep.sh is started and probably the WER for the test set will be slightly different too. I think this is not very important in this case, however because VoxForge still doesn't have predefined high-quality train and test sets and test time language model. It has something at here, but there are speakers, for which there are utterances in both sets and I wanted the sets to be disjoint.

The script assumes that you have FLAC codec installed on your machine, because some of the files are encoded in this lossless compression format. One solution is to convert the files beforehand into WAV format, but the recipe is using a nifty feature of Kaldi called extended filenames to convert the audio on-demand. For example data/local/train_wav.scp, which contains a list of files to be used for feature extraction, there are lines like:

benkay_20090111-ar_ar-09 flac -c -d --silent /media/secondary/voxforge/selected/benkay-20090111-ar/flac/ar-09.flac |

This means in effect, that when ar-09.flac needs to be converted into MFCC features flac is invoked first to decode the file, and the decoded .wav-format stream is passed to compute-mfcc-feats using Unix pipes.
It seems there are missing or not properly formatted transcripts for some speakers and these are ignored by the data preparation script. You can see these errors in exp/data_prep/make_trans.log file.

Building the language model
I decided to use MITLM toolkit to estimate the test-time language model. IRSTLM is installed by default under $ KALDI_ROOT/tools by Kaldi installation scripts, but I had a mixed experience with this toolkit before and decided to try a different tool this time. A script called local/voxforge_prepare_lm.sh installs MITLM in tools/mitlm-svn and then trains a language model on the train set. The installation of MITLM assumes you have svn, GNU autotools, C++ and Fortran compilers, as well as Boost C++ libraries installed. The order of the LM is given by a variable named lm_order in run.sh.

Preparing the dictionary
The script, used for this task is called local/voxforge_prepare_dict.sh. It downloads the CMU's pronunciation dictionary first, and prepares a list of the words that are found in the train set, but not in cmudict. Pronunciations for these words are automatically generated using Sequitur G2P, which is installed under tools/g2p. The installation assumes you have NumPy, SWIG and C++ compiler on your system. Because the training of Sequitur models takes a lot of time this script is downloading and using a pre-built model trained on cmudict instead.

Decoding
These were the most important steps specific to this recipe. Most of the rest is just borrowed from WSJ and RM scripts.
The decoding results for some of the steps are as follows:

view sourceprint?

```
exp/mono/decode/wer_10
%WER 64.96 [ 5718 / 8803, 320 ins, 1615 del, 3783 sub ]
%SER 96.29 [ 935 / 971 ]

exp/tri2a/decode/wer_13
%WER 44.63 [ 3929 / 8803, 412 ins, 824 del, 2693 sub ]
%SER 87.95 [ 854 / 971 ]

exp/tri2b_mmi/decode_it4/wer_12
%WER 38.94 [ 3428 / 8803, 401 ins, 753 del, 2274 sub ]
%SER 81.77 [ 794 / 971 ]
```

These are obtained using a monophone(mono), a maximum likelihood trained triphone(tri2a) and a discriminatevely trained(tri2b_mmi) triphone acoustic models. Now, I know the results are not very inspiring and may look even somewhat disheartening, but keep in mind they were produced using a very poor language model - a bigram estimated just on the quite small corpus of training set transcripts. Another factor contributing to the relatively poor results is that a typically about 2-3% of the words(depending on the random test set selection) are found just in test set, and thus considered out-of-vocabulary. Not only they don't have any chance to be recognized themselves, but are also reducing the chance for the words surrounding them to be correctly decoded.
One thing that I haven't had the time to try yet is to change the number of states and Gaussians in the acoustic model. As I've mentioned the training and decoding commands in run.sh were copied from RM recipe. My guess is that if the states and PDFs are increased somewhat that will lower the WER by 2-3% at least. This may be done by tweaking the relevant lines in run.sh, e.g.

view sourceprint?

```
# train tri2a [delta+delta-deltas]
steps/train_deltas.sh --cmd "$train_cmd" 1800 9000 \
data/train data/lang exp/tri1_ali exp/tri2a || exit 1;
```

In the above 1800 is the number of PDFs and 9000 is the total number of Gaussians in the system.
There is also something else worth mentioning here. As you may have noticed already I am using relatively small number of speakers(40) for testing. This is mostly because the set of prompts used in audio submission applet is limited and so there are many utterances duplicated across speakers. Because the test utterances are excluded when training the LM, we don't want too many test set speakers, because this would also mean less text for training language model and thus even worse performance.
Just to check everything is OK with the acoustic model I ran some tests using language models trained on both the train and the test set (WARNING: this is considered to be "cheating" and a very bad practice - please don't do this at home!). The results were about 3% WER with a trigram and about 17% with a bigram model with a discriminatevely trained AM.

5.3 在 vs 2013 中编译 kaldi

@王立军在 cnblog 上发表。

在 visual studio 2013 中编译 kaldi 的主要困难的地方在于正确编译 kaldi 的依赖库，主要是：openfst，ATLAS, pthreadVC2，其中又属 ATLAS 最难编译，因此，为求整个编译逻辑清晰简洁，在此将忽略 kaldi 所有依赖库的编译，如果有时间，我会直接提供其所依赖的库。

编译环境：

1.操作系统：windows8 （推荐）

2.编译工具：visual studio2013

3.其他：要安装完整的 CygWin 环境

编译步骤：

1.新建解决方案：打开 visual studio 2013，新建一个 Win32 项目，不妨将解决方案名称名为 kalid_vs13。可以先简单写个 HelloWorld 程序，用来测试项目是否成功建立

2.新建项目：在解决方案 kalid_vs13 中用 DLL 项目模板添加一个新的项目 kalid-lib

3.添加源文件：把 kaldi-trunk/src/下的的*.cc 文件加入项目 kalid-lib 中，注意这里不要把一些*bin 目录里的测试文件以及其他的一些文件名中含有关键字 test 的*.cc 文件加入 kalid-lib 中

4.配置 kalid-lib 的相关属性，包括：1）配置头文件的引用目录，2）预处理器定义的宏(HAVE_ATLAS)，3）依赖库的引用(openfst.lib,atlas.lib,pthreadVC2.lib）

5.其他配置：因为 nnet1 和 nnet2 目录下有相同的文件名：nnet-nnet.cc 和 nnet-componet.cc，所以需要配置他们各自的“输出文件”中的”对象文件名“，比如可以将 nnet1 中的 nnet-nnet.cc 的对象文件名设为$(IntDir)nnet1\，而将 nnet2 中的 nnet-nnet.cc 的对象文件名设为$(IntDir)nnet2\，其他重复文件类推

备注：

1. 要想顺利编译，还需要修改一些 kaldi 源文件，有时间我会向官方提交一些 patch，这样大家就能顺利编译 kaldi 了

2.以上的编译环境是我自己的，其他的环境是否能行我就不确定了

3. 这篇随笔主要叙述 kaldi 的整体编译思路，暂时不会关注编译细节！

5.4 kaldi 学习联盟群第一次讨论记录

2015 年 1 月 16 日，@神牛在 20 点到 21 点在 kaldi 学习联盟群回答各位群友的问题，感谢@神牛的分享和各位群友的提问。下面是一些问题的记录：

Q1：

@Sin-String :那小语种做语音识别，数据准备有什么特别注意的嘛？

因为 kws 的识别部分，数据准备是比较麻烦的，如果具体到小语种，那些词不像英语，错了还知道，所以想问一下，具体的注意事项

@神牛：

数据肯定是越大越好　可以考虑 参考 multilanguage 方面的论文

DNN 有一些 multilanguage 方面的论文： 基本思路是 共享隐藏层

每一种语言 独立拥有一个 softmax

@书包哥：

low-resource 情况下~multilingual 显然没有 crosslingual 好~

Q2：

@尘 :你好，我我能问个比较简单的问题么？最近在跑 kaldi 中的例子，结果跑出来了。但是其中有些细节我不是很理解。请问 timit 中的 DNN 例子中输入层和输出层的神经元数目分别是多少啊？是 40*11 和 61*3 么？谢谢啦

@神牛

DNN 的输入 是　前后拼接帧+当前帧的特征向量 输出 就是　tied-stated 数目

高斯混合数目

DNN 的输入： 比如输入前后 5 帧 那么输入数目就是 （5+5+1）*特征维数

输出数目就是 高斯混合数目　每一个输出节点的概率值 等于【要经过贝叶斯转换】一个 GMM 的概率值

Q3：

@可乐乐:我的数据源可能和别人不一样，我的数据是人体声道的三维描述特征，也有一帧一帧的信息，请问我该如何准备我的数据，以及后面做识别有什么要注意的地方？

@神牛

不同来源的特征 可以拼接在一起　在机器学习中 应该是叫 特征组合

那就拼接到 对应时间帧后面　完全可以特征组合 ASR 不是也使用 pitch 做特征吗？

不同的特征组合后面的 HMM 结构和单个特征一样吗？

那多流 HMM 是指啥？不是对应多个特征的么？@神牛

注意维数的上升~如果用同样多数据，hmm 的自由度可能需要调整~拼接完一般 hmm 自由度需要调整

多流：多个 特征流

Q4：

提 pitch 应该用什么工具呢？就 Kaldi 嘛？还是 praat 什么的

@神牛:

pitch Kaldi 中有相应地代码

kaldi 里的 pitch 是改进的 nccf 方法，具体可以看下代码 可以看下 povey 的论文

tone 语音 如 汉语 pitch 特征 很有效 情感你可以试着加一些长时韵律特征

Q5:

说话人识别目前 state of the art 的方法是什么呢？是 ivector+什么？

@书包哥：

ivector+plda

Q6: ASR 最近研究的热点都有啥？Kaldi 在这方面有啥优势？

1.Kaldi 的文档 和 脚本 都很清晰；Shell 和 C++ 也都是大众语言；

2. 多核 or 集群 计算环境的加速

3.DNN 的支持 【含 CUDA】

高配置单机比如 6 核 12 线程 4GPU 的配置 完全可以在单周内完成 一套 ASr 的训练 数千小时的训练数据

相比 HTK，除了 DNN 支持、扩展方面等，算法方面还有什么优势吗？

算法方面 就是 WFST 的集成了

Kaldi 完全基于 WFST， 训练、识别 都使用 WFST

嗯，明白了，感觉研究用的数据库好像没有那么大；DNN 和 WFST 还是比较前沿的 WFST 不算前沿了 上个实际的东西

Q7：

@Amnesia :请教一下，关于中文语音识别。之前都看了英文的实验，主要是 timit,别的数据库我也没有。英文把单词用音素表示，然后对音素建模。那中文怎么办呢 就用普通的拼音可以做吗 还是有类似的音素词典？多谢解答

@神牛:

中文 使用声韵母 比较常见　英语语料库 有很多 开放的

这个可以看下 cmu 的 dict 也可以使用 其他的方式，可以结合 语音学相关的知识 设计发音词典 或者使用 数据驱动的方式

具体可以参考：《中文连续语音识别系统音素建模单元集的构建》

都是声韵母，增加或减少些吧

Q8：:

@书包哥：请问拿到一个新语种~怎么构建状态绑定的问题集？数据驱动的问题集怎么构建呢~比如日语

@神牛：

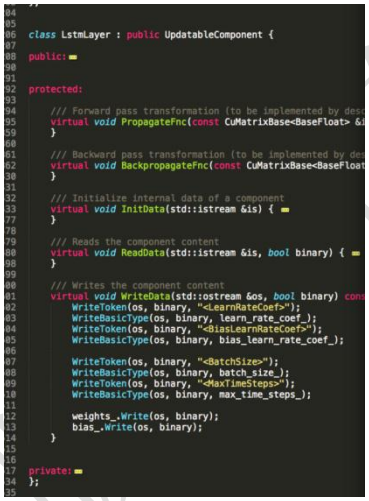
现在 Kaldi 是使用 数据驱动的方式 构建 决策树的　聚类

可以参考 Kaldi 构建决策树的方式　完全数据驱动~就是不要问题集吗

问题集 是数据驱动构建的

Q9：

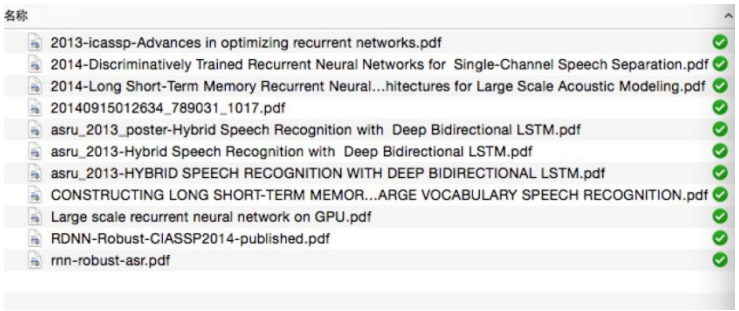
比如双向 RNN、LSTM 等在语音识别上是否值得探索？



```
class LstmLayer : public UpdatableComponent {

public:

protected:

    /// Forward pass transformation (to be implemented by des
    virtual void PropagateFnc(const CuMatrixBase<BaseFloat> &
    }

    /// Backward pass transformation (to be implemented by de
    virtual void BackpropagateFnc(const CuMatrixBase<BaseFloat
    }

    /// Initialize internal data of a component
    virtual void InitData(std::istream &is) {
    }

    /// Reads the component content
    virtual void ReadData(std::istream &is, bool binary) {
    }

    /// Writes the component content
    virtual void WriteData(std::ostream &os, bool binary) con
        WriteToken(os, binary, "<LearnRateCoef>");
        WriteBasicType(os, binary, learn_rate_coef_);
        WriteToken(os, binary, "<BiasLearnRateCoef>");
        WriteBasicType(os, binary, bias_learn_rate_coef_);

        WriteToken(os, binary, "<BatchSize>");
        WriteBasicType(os, binary, batch_size_);
        WriteToken(os, binary, "<MaxTimeSteps>");
        WriteBasicType(os, binary, max_time_steps_);

        weights_.Write(os, binary);
        bias_.Write(os, binary);
    }

private:
};
```

这个类 目前实现了 Deep RNN，　总体来说　能求导 完成这个 并不困难

RNN 是语音的下一个热点



名称

- 2013-icassp-Advances in optimizing recurrent networks.pdf
- 2014-Discriminatively Trained Recurrent Neural Networks for Single-Channel Speech Separation.pdf
- 2014-Long Short-Term Memory Recurrent Neural...hitectures for Large Scale Acoustic Modeling.pdf
- 20140915012634_789031_1017.pdf
- asru_2013_poster-Hybrid Speech Recognition with Deep Bidirectional LSTM.pdf
- asru_2013-Hybrid Speech Recognition with Deep Bidirectional LSTM.pdf
- asru_2013-HYBRID SPEECH RECOGNITION WITH DEEP BIDIRECTIONAL LSTM.pdf
- CONSTRUCTING LONG SHORT-TERM MEMOR...ARGE VOCABULARY SPEECH RECOGNITION.pdf
- Large scale recurrent neural network on GPU.pdf
- RDNN-Robust-CIASSP2014-published.pdf
- rnn-robust-asr.pdf

Q10：想问一个 cmn 方面的问题，假设待测试语句两端包含静音 计算谱均值的时候，要不要考虑静音部分

假设经过 vad，可以确定语音的起始与结束点 不考虑静音的话，识别结果会变差

@神牛：

与你训练的时候 保持一致

因为 vad 可以确定语音的起止点，所以静音部分就没送进解码器

可以认为静音部分 是背景噪音 -背景噪声的 mean 相当于 除噪

确切地说就是测试语音进行 cmn 的时候考虑 mean（测试语料的背景噪声）是 mean（训练语料的噪声）

Q11：有过 ReLU 和 Dropout 的调参经验吗

@神牛：

加过 dropout BP 几轮完事 去掉 dropout 再 BP 几轮

调整网络结构（svd dropout 缩减节点数目）后 都需要 BP 几轮 不然效果会很差

relu 没搞过。

5.5 kaldi 在线中文识别

最近研究了下 kaldi，也看了好多文章，感觉这方面的有用文章太少！ 大多停留在编译和深层理论方面。对于我们这种没有基础的人，除了看完编译，就只能很茫然的看哪些高高在上的理论了。

本文章，适合那些。刚刚编辑完 kaldi。想试试手，找找感觉的人来看的。如果你还没有编辑过，可以查查相关文章。有好多。

声明一下，本文章，只适合在 windows 下装虚拟机 UBuntu 的机器上玩。你的虚拟机内存要大些（4g--8g 最好）。不然很痛苦。

找感觉一定要从中文开始，我们直接拿那个清华大学的中文例子（thchs30）开练，

1 首先进入 github 下载的源码中 egs 目录下找到这个例子，然后要看一遍它的介绍。在 readme 里面有语料库的下载方式。把它下完（3 个压缩包全下）。4 个多 G 比较大。

2 考到对应的文件夹下（与 s5 同级别即可），取名叫 thchs30-openslr，将所有压缩包解压到这下面

3 打开 s5 目录，编辑 cmd.sh. 如下：()

export train_cmd=run.pl

export decode_cmd=run.pl

export mkgraph_cmd=run.pl

export cuda_cmd=run.pl

4 打开 run.sh，看到#data preparation 这句，在它之后就全是 shell 的命令。建议一条一条的跑。不然中间会有莫名奇妙的断档和错误。如何一条条跑呢？ 使用注释：

<<!EOF! !EOF! 这两句相当于 c 语言的/* */. 但愿这个你能看懂。在此谢谢小凡妹妹的在线指导。

5 按照上面的一句一句的来。它大概有几个过程：数据准备，monophone 单音素训练， tri1 三因素训练， trib2 进行 lda_mllt 特征变换，trib3 进行 sat 自然语言适应，trib4 做 quick（这个我也不懂），后面就是 dnn 了

6 当运行到 dnn 时候会报错，因为默认 dnn 都是用 GPU 来跑的。它会检查一下，发现只在 CPU 下，就终止了。这里建议不要跑 dnn 了。因为我试过，改成 CPU 之后跑了 7，8 天，才迭代 17，18 次。太慢了。而一次训练怎么的也得 20 多次。还要训练好几回。所以，想跑 dnn 的话还是找 GPU 吧。

7.下面说说前面的几个步骤干啥用。其实每一次都会有个模型，这些模型都可以用了，你可以看它的 exp 目录，所有的步骤都在这里面输出。我们先简单看一个，tri1，打开后，有个 decode_test_word,里面有个 scoring_kaldi，best_wer 就是它的错误率，36.15%.好回到 tri1 下，看到 final.mdl 了吗，这个就是有用的东西，学出来的模型。另外还得到 graph_word 下面，找到 words.txt,和 HCLG.fst，一个是字典，一个是有限状态机。有这 3 个文件，就可以来使用你的识别功能了。

8.这步是比较好玩的了。我们用这个模型来识别自己的语言。（是非常的不准，只是感受一下而已），回到源码的 src 下。make ext 编译扩展程序。（在这之前确定你的 tools 文件夹下的 portaudio 已经装好）之后，会看到 onlinebin 文件夹。里面有两个程序，online-wav-gmm-decode-faster 用来回放 wav 文件来识别的，online-gmm-decode-faster 用来从麦克风输入声音来识别的。

8.1 这里开个小差：portaudio 装好后，有可能收不到声音，可以装个 audio recoder（用 apt-get）,之后用它录音试试，测测是否有声音，只要能录音，portaudio 就没问题，一般装完就好了，不行就再重启一下。不知道为啥。

8.2 介绍下 online。很鄙视 kaldi 在这块的设计，居然搞出来两个版本。online 是不再维护了。但是非常好用，online2 是新的。参数 n 多。也不支持从麦克风采集。特别不人性化，如果有那个朋友把 online2 研究好了，请把经验也分享一下。

9.我们找一个例子吧：去 egs 下，打开 voxforge，里面有个 online_demo，直接考到 thchs30 下。在 online_demo 里面建 2 个文件夹 online-data work,在 online-data 下建两个文件夹 audio 和 models，audio 下放你要回放的 wav，models 建个文件夹 tri1，把 s5 下的 exp 下的 tri1 下的 final.mdl 和 35.mdl（final.mdl 是快捷方式）考过去。把 s5 下的 exp 下的 tri1 下的 graph_word 里面的 words.txt,和 HCLG.fst，考到 models 的 tri1 下。

10。打开 online_demo 的 run.sh

a)将下面这段注释掉：（这段是 voxforge 例子中下载现网的测试语料和识别模型的。我们测试语料自己准备，模型就是 tri1 了）

```
if [ ! -s ${data_file}.tar.bz2 ]; then
    echo "Downloading test models and data ..."
    wget -T 10 -t 3 $data_url;



    if [ ! -s ${data_file}.tar.bz2 ]; then
        echo "Download of $data_file has failed!"
        exit 1
    fi
fi
```

b) 然后再找到如下这句，将其路径改成 tri1

```
 # Change this to "tri2a" if you like to test using a ML-trained model
ac_model_type=tri2b_mmi

# Alignments and decoding results
```

改成：

```
 # Change this to "tri2a" if you like to test using a ML-trained model
ac_model_type=tri1
```

c）

```
online-wav-gmm-decode-faster --verbose=1 --rt-min=0.8 --rt-max=0.85\
            --max-active=4000 --beam=12.0 --acoustic-scale=0.0769 \
            scp:$decode_dir/input.scp $ac_model/model
```

改成：online-wav-gmm-decode-faster --verbose=1 --rt-min=0.8 --rt-max=0.85\

--max-active=4000 --beam=12.0 --acoustic-scale=0.0769 \

scp:$decode_dir/input.scp $ac_model/final.mdl

11。直接./run.sh 吧，就是开始回放识别了。里面有提示，./run.sh --test-mode live 命令就是从麦克风识别。

12.升华部分在这里。我们试完 tri1 的模型后，一定很想试试 tri2 或 3.但当你操作时，会遇到如下的问题：

ERROR (online-wav-gmm-decode-faster:LogLikelihoods():diag-gmm.cc:533)

DiagGmm::ComponentLogLikelihood, dimension mismatch 39vs. 40

怎么解决？ 仔细看看这个源文件，它是 dieta 的。如果要是 ldp 还得加 matrix 参数（拿 tri2b 举例）。

于是修改 run.sh 成如下这个样子 ：（就是把 final.mat 考过来，引入命令中）

```
if [ -s $ac_model/matrix ]; then
    trans_matrix=$ac_model/12.mat
fi
```

同时把把 s5 下的 exp 下的 tri2b 下的 12.mat 考到 models 的 tri2b 下。

13 再次修改 run.sh 成如下这个样子（添加 2 个参数--left-context=3 --right-context=3）

```
 online-wav-gmm-decode-faster --verbose=1 --rt-min=0.8 --rt-max=0.85 \
            --max-active=4000 --beam=12.0 --acoustic-scale=0.0769 --left-context=3
--right-context=3\
            scp:$decode_dir/input.scp $ac_model/final.mdl $ac_model/HCLG.fst \
            $ac_model/words.txt '1:2:3:4:5' ark,t:$decode_dir/trans.txt \
            ark,t:$decode_dir/ali.txt $trans_matrix;;
```

14 运行./run.sh，结果如下。怎么样，有点酷不？ 如果想使用 tri2 等模型做麦克风在线的，也同理修改就可以了。

```
online-wav-gmm-decode-faster --verbose=1 --rt-min=0.8 --rt-max=0.85 --max-active=4000
--beam=12.0 --acoustic-scale=0.0769 --left-context=3 --right-context=3 scp:./work/input.scp
online-data/models/tri2b/final.mdl online-data/models/tri2b/HCLG.fst
online-data/models/tri2b/words.txt 1:2:3:4:5 ark,t:./work/trans.txt ark,t:./work/ali.txt
online-data/models/tri2b/12.mat
File: D4_750
```

苏北 军礼 下跪 将 是 马 湛 杀人 里 杜 唐 据 五 苏 并 案 但 甜美 但 也 分析 抗战

6 其他资料和资源

1.kaldi 主页：http://kaldi.sourceforge.net/

2.Povey 主页：http://www.danielpovey.com/

3.Kaldi+pdnn：http://www.cs.cmu.edu/~ymiao/kaldipdnn.html

4.@马源的有关 kaldi 的介绍：
http://blog.csdn.net/sytxsybm/article/category/1867123

5.@神牛的 csdn 博客里关于 kaldi 的内容：
http://blog.csdn.net/lifeitengup/article/category/2095113
以及新的博客：http://ftli.farbox.com/

6.@天大的同学的博客：
http://blog.csdn.net/jojozhangju/article/category/2121691

7.wbglearn 的博客（我的博客）：http://blog.csdn.net/wbgxx333

欢迎大家在这里添加资料和资源，如果你觉得有哪些比较好的，欢迎和我联系，我将逐渐的完善这部分。

7 版本更新日志

2014.7.14	V0.1	kaldi 全部资料第一版发布
2014.8.16	V0.2	更新：1.添加新的翻译：4.9 2.kaldi 在 vs 上的调试建议
2014.8.20	V0.3	更新：添加新的翻译：（有 3 个）4.10.4.11.4.12 4.13
2014.9.10	V0.4	更新：添加 kaldi 里数据库的介绍 3.2 添加新的翻译
2014.12.9	V0.5	修改一些细节和安装的一些细节问题
2015.1.21	V0.6	更新：添加新的章节：3.7 3.8 3.9 5.4
2016.9.5	V0.7	更新：添加一些新的章节