

---

# 在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.html>[http://kaldi.sourceforge.net/tutorial\\_prereqs.html](http://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 and 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 **linux** 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](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，假设您像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

假设你想根据你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 系统必须用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夹 `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: <utteranceID><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: <utteranceID><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: <utteranceID><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](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` 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` 中找到类似的例子。

### 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 **SRILM** 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/featbin/:$KALDI_ROOT/src/lm/:$KALDI_ROOT/src/sgmmbin/:$KALDI_ROOT/src/sgmm2bin/:$KALDI_ROOT/src/fgmmbin/:$KALDI_ROOT/src/latbin/:$PWD:$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.scf data/train/feats.scf
data/train/split1 data/test/spk2utt data/test/cmvn.scf data/test/feats.scf
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     [<utteranceID><full_path_to_audio_file>]

# text        [<utteranceID><text_transcription>]

# utt2spk     [<utteranceID><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

nggram-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 | fstmrepsilon | 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](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](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。