第 12 章
Kaldi 实践

Kaldi 是一个与 HTK 类似的开源语音识别工具包，其底层基于 C++编写，可以在 Linux 和 Windows 平台上编译。Kaldi 使用 Apache 2.0 作为开源协议发行，旨在提供自由的、易修改和易扩展的底层代码、脚本和完整的工程示例，供语音识别研究者自由使用。

Kaldi 工具有很多特色，包括：

- 在 C++代码级别整合了 OpenFst 库。
- 支持基于 BLAS、LAPACK、OpenBLAS 和 MKL 的线性代数运算加速库。
- 包含通用的语音识别算法、脚本和工程示例。
- 底层算法的实现更可靠，经过大量有效测试，代码规范易理解、易修改。
- 每个底层源命令均功能简单，容易理解，命令之间支持管道衔接，工作流程分工明确，整个任务由上层脚本联合众多底层命令来完成。
- 支持众多扩展工具，如 SRILM、sph2pipe 等。

虽然 Kaldi 相比 HTK 支持更多的特性，但 Kaldi 除了官网的介绍和开源的工程示例，没有一个类似于 HTKBook 的完整使用文档，这使得 Kaldi 的使用门槛更高，它要求使用者至少懂得 shell 脚本和一些语音知识。在本章中，我们将介绍基于 Linux 系统从零开始构建一个完整的 Kaldi 系统用于语音识别，并训练 GMM-HMM、DNN-HMM 和 Chain 模型，如图 12-1 所示。

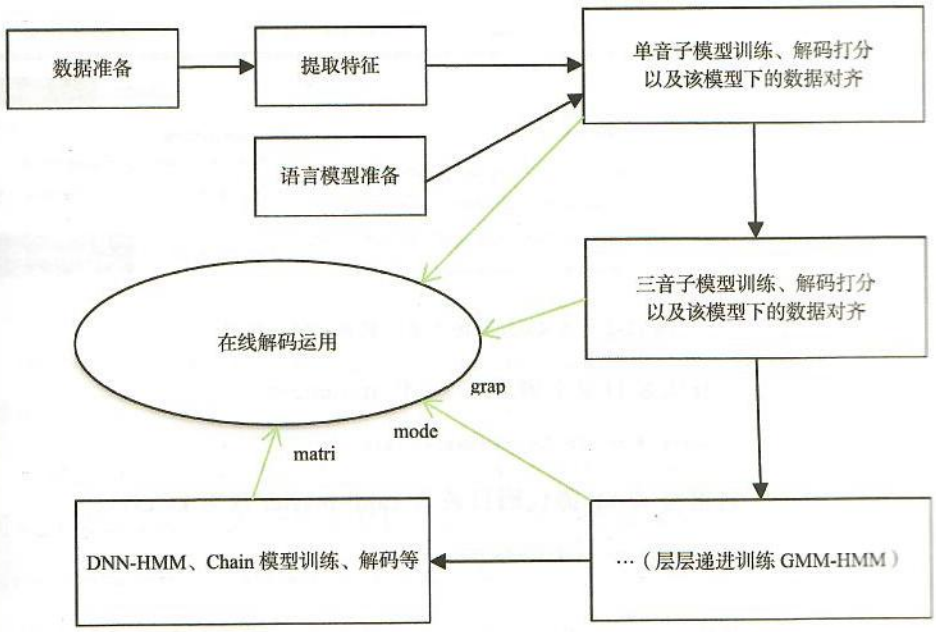

图 12-1　Kaldi 系统构建过程

12.1　下载与安装 Kaldi

我们可以从 Kaldi 的官网下载 Kaldi。

Kaldi 的安装非常简单，首先获取源代码，然后进行编译。这里将 Kaldi 安装在/work 目录下。

12.1.1　获取源代码

获取源代码有两种方式。

（1）直接在终端利用 git 命令从 Kaldi 的 GitHub 代码库克隆：

```
[root@localhost work]# git clone https://github.com/kaldi-asr/kaldi.git kaldi
--origin upstream
```

（2）从 Kaldi 开源地址（GitHub 上的 Kaldi 站点）下载，获得源代码压缩包 kaldi-master.zip，如图 12-2 所示。

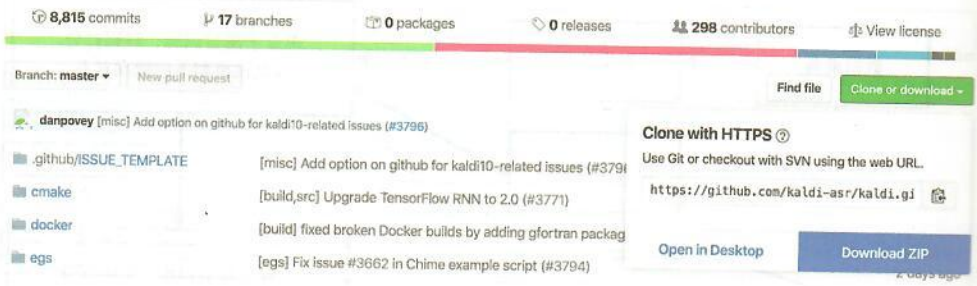

图 12-2　从 Kaldi 开源地址下载源代码压缩包

下载下来后，在安装目录下解压缩 kaldi-master.zip：

```
[root@localhost work]# unzip kaldi-master.zip
```

为了统一，这里将 Kaldi 源代码目录名 kaldi-master 改为 kaldi：

```
[root@localhost work]# mv kaldi-master kaldi
```

12.1.2　编译

编译可依次按 kaldi/INSTALL、kaldi/tools/INSTALL 和 kaldi/src/INSTALL 三个安装指示文件的说明进行。具体分两步：先编译依赖工具库 kaldi/tools，再编译 Kaldi 底层库 kaldi/src。

在编译 kaldi/tools 之前，检查依赖项：

```
[root@localhost work]# cd kaldi/tools
[root@localhost tools]# extras/check_dependencies.sh
extras/check_dependencies.sh: all OK.
```

在编译中可能因为系统环境不满足依赖项要求，如 gcc 版本问题等而导致编译失败。对于所有的编译问题，可通过查看 kaldi/INSTALL、kaldi/tools/INSTALL 和 kaldi/src/INSTALL 三个安装指示文件，以及安装失败时的报错信息来解决。通常，由于依赖工具未安装而导致的编译失败，在其报错信息中有相关依赖工具安装包的安装提示，可直接复制并进行安装，比如 Ubuntu 系统为 apt-get install **，CentOS 系统为 yum install **。具体的依赖工具列表，可通过查看 extras/check_dependencies.sh 脚本了解。

例如，报错如下：

```
[root@localhost tools]# extras/check_dependencies.sh
......
extras/check_dependencies.sh: Intel MKL is not installed. Download the installer
package for your
 ... system from: https://software.intel.com/mkl/choose-download.
 ... You can also use other matrix algebra libraries. For information, see:
 ...   http://kaldi-asr.org/doc/matrixwrap.html
extras/check_dependencies.sh: The following prerequisites are missing; install
them first:
   g++ zlib1g-dev automake autoconf sox gfortran libtool
```

此时应该先安装 g++、zlib1g-dev、automake、autoconf、sox、gfortran 和 libtool 依赖项。

如果希望使用 MKL 线性代数运算库加快计算速度(CPU 由 Intel 公司生产), 则可以在编译 kaldi/tools 之前先安装 MKL (否则，在编译时将默认安装 Atlas):

```
[root@localhost tools]# extras/install_mkl.sh
```

依赖检查通过后，使用多进程加速编译 kaldi/tools:

```
[root@localhost tools]# make -j 4
```

编译完 kaldi/tools 后，开始编译 kaldi/src 目录。在此之前，先执行配置脚本:

```
[root@localhost tools]# cd ../src/
[root@localhost src]# ./configure --shared
```

配置检查通过后，进行最后的编译:

```
[root@localhost src]# make depend -j 4
[root@localhost src]# make -j 4
```

最后,测试能否正常运行 yes/no 示例(需要联网,脚本会自动下载少量数据):

```
[root@localhost src]# cd ../egs/yesno/s5/
[root@localhost s5]# sh run.sh
......
%WER 0.00 [ 0 / 232, 0 ins, 0 del, 0 sub ] exp/mono0a/decode_test_yesno/wer_10_0.0
```

若运行成功，则最后会输出如上所示的 WER 指标。至此，Kaldi 安装完毕。

12.2 创建和配置基本的工程目录

在 Kaldi 中，为了统一环境变量配置，所有的工程目录均在 kaldi/egs 下创建。

具体需要创建一个如 kaldi/egs/name/version 这样的工程目录，即工程目录 version 应该是根目录 kaldi 的三级子目录。在正式通过 aishell 工程示例介绍语音识别训练系统之前，这里先介绍如何从零开始准备自己的工程目录。

创建一个工程目录：

```
[root@localhost work]# cd /work/kaldi/egs/
[root@localhost egs]# mkdir -p speech/s5
[root@localhost egs]# cd speech/s5/
[root@localhost s5]#
```

从官方 wsj 示例链接工具脚本集 utils 和训练脚本集 steps：

```
[root@localhost s5]# ln -s ../../wsj/s5/utils/ utils
[root@localhost s5]# ln -s ../../wsj/s5/steps/ steps
```

复制环境变量文件 path.sh，并根据之后构建训练总流程脚本 run.sh 的需要，复制并行配置文件 cmd.sh：

```
[root@localhost s5]# cp ../../wsj/s5/path.sh ./
[root@localhost s5]# cp ../../wsj/s5/cmd.sh ./
```

对于 path.sh 文件，在调用 utils 和 steps 中的脚本时会默认用到。cmd.sh 文件中保存了一些配置变量，供训练脚本用。比如 run.sh，全局使用；queue.pl，一般用于联机环境中，在单机情况下，并行处理脚本应使用 run.pl（原脚本在 utils/parallel/下）。

在创建了基本的工程目录并配置了基本环境后，就可以在工程中添加工程相关内容了。工程内容包括数据集映射目录、工程相关总流程脚本及其子脚本。对于前者，需要根据自己的数据存储结构，自行处理每个数据集并生成符合 Kaldi 映射规范的映射目录；而对于后者，根据工程目的，一方面可借鉴已有的公开示例中的处理或训练脚本来解决自己的工程任务，另一方面可根据自己的需要灵活修改任何脚本和 C++底层。接下来将结合 aishell 工程示例继续介绍与语音识别相关的工程内容要点。

12.3 aishell 语音识别工程

在 Kaldi 工程中，除了配置基本的工程目录，以使用 Kaldi 自身提供的底层

命令和脚本，在开展实验之前，我们还需要将数据集处理成符合 Kaldi 映射规范的映射目录。我们以 aishell 工程为例，来了解工程相关内容。

查看 aishell/s5 工程初始内容：

```
[root@localhost kaldi]# cd /work/kaldi/egs/aishell/s5/
[root@localhost s5]# ls
cmd.sh  conf  local  path.sh  RESULTS  run.sh  steps  utils
```

其中，cmd.sh、path.sh、steps 和 utils 是工程基本配置，其余部分则是 aishell 工程的具体内容。

在 Kaldi 的开源工程示例中，所有的示例均有一个名为 run.sh 的总流程脚本，通过运行该脚本，可直接跑完所有的步骤，包括数据准备、数据切分、特征提取、特征处理、模型训练、测试打分和结果收集等步骤。为了方便理解，接下来我们把 run.sh 包含的内容拆解成各个模块逐一介绍。

12.3.1 数据集映射目录准备

首先，由于开源工程示例的开源特性，数据准备通常包括从网上自动下载数据集并生成相应的数据集映射目录。在 aishell 工程中，也提供了这样的自动化脚本（注：如果在单机上运行本示例，则应在运行相关脚本前，先将 cmd.sh 中的所有 queue.pl 修改为 run.pl）。

修改 run.sh 中的数据存储路径变量，并通过在模块末尾添加终止命令执行各个模块。首先，自动下载数据并解压缩：

```
[root@localhost s5]# vi run.sh
>> run.sh
data= wavdata # 修改数据存储路径变量
data_url=www.openslr.org/resources/33
. ./cmd.sh
mkdir -p $data # 添加创建目录命令
local/download_and_untar.sh $data $data_url data_aishell || exit 1;
local/download_and_untar.sh $data $data_url resource_aishell || exit 1;
exit 1 # 添加终止命令，该模块执行完后，注释掉所有不相关的已执行代码并进入下一个模块，重复操作
```

执行第一个模块并查看下载的数据内容：

```
[root@localhost s5]# sh run.sh
[root@localhost s5]# ls wavdata/
```

```
data_aishell  README.txt  resource_aishell  resource_aishell.tgz  s5
```

自动准备映射目录：

```
>> run.sh
local/aishell_data_prep.sh $data/data_aishell/wav \
$data/data_aishell/transcript || exit 1; # 自动准备映射目录
[root@localhost s5]# ls data/
dev  local  test  train
[root@localhost s5]# ls data/train/
spk2utt  text  utt2spk  wav.scp
```

在 aishell 工程中包含了 3 个数据集：train、dev 和 test。每一个数据集都包含了 4 个映射文件：wav.scp、utt2spk、spk2utt 和 text。

在 Kaldi 中，每个数据集都由映射文件来描述，而在语音识别工程中，至少需要自行准备 wav.scp、utt2spk/spk2utt 和 text 这 3 个文件，其中，utt2spk 和 spk2utt 可使用 utils/spk2utt_to_utt2spk.pl 和 utils/utt2spk_to_spk2utt.pl 两个脚本相互生成，故只需要准备其中一个即可。映射文件每行格式如下：

wav.scp = [utt-id] + [wav-path]

```
[root@localhost s5]# head -n 3 data/train/wav.scp
BAC009S0002W0124 wavdata/data_aishell/wav/train/S0002/BAC009S0002W0124.wav
BAC009S0002W0125 wavdata/data_aishell/wav/train/S0002/BAC009S0002W0125.wav
BAC009S0002W0126 wavdata/data_aishell/wav/train/S0002/BAC009S0002W0126.wav
```

utt2spk = [utt-id] + [spk-id]

```
[root@localhost s5]# head -n 3 data/train/utt2spk
BAC009S0002W0124 S0002
BAC009S0002W0125 S0002
BAC009S0002W0126 S0002
```

spk2utt = [spk-id] + [utt-id-1 utt-id-2 …]

```
[root@localhost s5]# head -n 3 data/train/spk2utt
S0002 BAC009S0002W0122 BAC009S0002W0123 BAC009S0002W0124 …
S0003 BAC009S0003W0121 BAC009S0003W0122 BAC009S0003W0123 …
S0004 BAC009S0004W0121 BAC009S0004W0123 BAC009S0004W0124 …
```

text = [utt-id] + [transcript]

```
[root@localhost s5]# head -n 3 data/train/text
BAC009S0002W0124 自 六月 底 呼和浩特 市 率先 宣布 取消 限 购 后
BAC009S0002W0125 各地 政府 便 纷纷 跟进
```

```
BAC009S0002W0126 仅 一 个 多 月 的 时间 里
```

由于在语音识别工程中更重要的是文本标注 text 文件，因此实际上，在未用到说话人信息时，utt2spk 与 spk2utt 中的 utt-id 和 spk-id 可以总是一致的，即每句话都可以对应一个说话人，比如音频 wav10001 的 spk-id 可以直接被命名为 wav10001，这样就可以忽略有些数据集没有统计说话人信息的问题。另外，由于 Kaldi 一些排序算法的要求，utt-id 通常要包含 spk-id 字符串作为前缀。例如，使用 spk001-wav10001 字符串作为 wav10001 的 utt-id，否则当合并多个采用不同命名规范的数据集时，可能出现一些数据映射检查通不过的问题。

12.3.2 词典准备和 lang 目录生成

在 aishell 语音识别工程中，由于其基于中文的普通话音素级别建模，而我们的标注通常为汉字形式的字词级别，因此需要有一个中文到普通话音素的词典以建立映射。

自动准备词典：

```
>> run.sh
local/aishell_prepare_dict.sh $data/resource_aishell || exit 1;
```

词典每行格式如下：

lexicon.txt = [word] + [phone-sequence]

```
[root@localhost s5]# head -n 5 data/local/dict/lexicon.txt
SIL sil
<SPOKEN_NOISE> sil
啊 aa a1
啊 aa a2
啊 aa a4
[root@localhost s5]# tail -n 5 data/local/dict/lexicon.txt
坐诊 z uo4 zh en3
坐庄 z uo4 zh uang1
坐姿 z uo4 z iy1
座充 z uo4 ch ong1
座驾 z uo4 j ia4
```

aishell 工程使用的普通话音素和 kaldi/egs/thchs30/s5 工程使用的一致，其中音素集即静音音素“sil”与所有拼音的声母和韵母集合。声母包含真声母和虚拟声母（如 aa），虚拟声母使整体音节也可分解为两个音节，从而使得每个拼音的

音节个数均为 2——这样既方便处理，也有利于减少解码时音素映射到汉字时的边界混淆。例如，可认为“a o”是‘a’与‘o’或者“ao”，但“aa a1 oo o1”则显然有明确的含义。在英语等外语或者方言识别中，若基于音素级别建模，则同样需要准备适合该语种的音素集和词典。例如，英语识别工程 kaldi/egs/librispeech/s5/以单词或词组为词，音标作为音素。同时，词典中允许出现多音词和同音词，音素也可以按声调进一步划分，建模粒度非常自由。值得注意的是，与静音音素“sil”对应的词<SIL>和 <SPOKEN_NOISE>，即使被添加到词典中，但是如果语言模型中未添加这两个词，在解码时它们也始终不会出现在搜索路径中。另外，静音音素“sil”除了表示真正的静音，还起到一个特殊的作用，即作为任何音素之间的可选插入音素。这意味着，即使两个音素之间出现一段静音，也不会影响解码时把一个音素序列合成一个特定的词，此时的“sil”并非对应到<SIL>这个词，而是对应到有限状态转换器中的<eps>。比如在英文识别中，一个单词可能很长，在念该单词时中途即使出现停顿，也依然可以解码出这个单词。

在与词典相关的准备中，最终目标是生成与语法相关的 lang 目录，以使得 Kaldi 程序能够将词和音素等信息联系起来。实际上，只需要先准备好 data/local/dict，就可以通过相关脚本自动生成 lang 目录。而 data/local/dict 中除了包含上面所讲的词典文件 lexicon.txt 或 lexiconp.txt（包含概率的词典，准备两者中的一个即可），还需要包含另外 4 个文件。

查看 aishell 工程为生成 lang 目录准备的文件：

```
[root@localhost s5]# ls data/local/dict/
extra_questions.txt  lexiconp.txt  lexicon.txt  nonsilence_phones.txt
optional_silence.txt  silence_phones.txt
```

额外需要准备的 4 个文件如下。

- silence_phones.txt：包含所有静音音素，一般仅有“sil”。

```
[root@localhost s5]# cat data/local/dict/silence_phones.txt
sil
```

- nonsilence_phones.txt：包含所有非静音音素，在 aishell 工程中为所有声韵母。若音素出现在同一行，则在后面训练中生成决策树分支时，会依据该定义先将它们划分成一个小类，这相当于在基于数据驱动的决策树聚类中

加入了一些有明显联系的先验信息。显然，aishell 工程将声母作为一类，并将不同音调的韵母分为一类。如无特殊定义，每个音素可单独占一行。

```
[root@localhost s5]# cat data/local/dict/nonsilence_phones.txt
a1 a2 a3 a4 a5
aa
ai1 ai2 ai3 ai4 ai5
an1 an2 an3 an4 an5
ang1 ang2 ang3 ang4 ang5
……
```

- optional_silence.txt：可选音素，一般仅有“sil”。

```
[root@localhost s5]# cat data/local/dict/optional_silence.txt
sil
```

- extra_questions.txt：问题集，用于在产生决策树分支时，定义一些音素之间的其他联系。比如在 aishell 工程中，先将“sil”单独分为一类，然后将声母分为一类，最后将韵母按 5 种不同音调（后缀‘5’表示轻声）各自分为一类。如无特殊定义，该文件可置空。

```
[root@localhost s5]# ls data/local/dict/
```

在准备好 data/local/dict 后，自动生成 lang 目录：

```
>> run.sh
utils/prepare_lang.sh --position-dependent-phones false data/local/dict \
"<SPOKEN_NOISE>" data/local/lang data/lang || exit 1;
[root@localhost s5]# ls data/lang/
L_disambig.fst  L.fst  oov.int  oov.txt  phones  phones.txt  topo  words.txt
```

在 lang 目录中，phones.txt 和 words.txt 文件分别定义了音素和词的索引（0-based），以方便程序使用。而集外词文件 oov.txt 包含了生成 lang 目录时脚本参数指定的<SPOKEN_NOISE>词，其作用是在处理标注时，将词典中没有的词统一映射到自己的音素（此处为“sil”），以处理集外词的情况。然而，该词如前所述，通常在解码阶段不起作用，仅在训练阶段使用。

12.3.3 语言模型训练

在 Kaldi 中，语音识别解码用于构成静态搜索网络 HCLG 所用的语言模型为 *n*-gram 模型，而训练*n*-gram 语言模型分为 3 个步骤：收集并处理语料；使用相关工具训练*n*-gram 语言模型；将语言模型转换为有限状态转换器形式。

（1）收集并处理语料

关于语料，应该收集适用于语音识别模型应用场景的文本。例如，对于一般的用于日常识别的语音识别模型，可下载公开的《人民日报》语料或收集日常口语等作为语言模型训练语料。语料收集完成后，先进行文本清洗，去除词典中不存在的字符，并将文本语料分句，再使用分词算法按词典中已有的词进行分词，比如使用最大正向匹配分词算法。值得注意的是，对词典的准备，也可以在语料收集完成之后进行，并根据一些比较好的分词算法制作一个特殊的词典。总之，文本语料的分词应和词典保持一致。用于训练语言模型的语料应至少包含所有词典中的词，这样才能保证每个词在解码时都有概率出现。

（2）使用相关工具训练*n*-gram 语言模型

训练*n*-gram 语言模型的工具有很多。在 Kaldi 中，有两个工具比较好用：一个是经典的 SRILM 工具，另一个是 Kaldi 自带的语言模型训练工具 kaldi_lm。两者均需要在 kaldi/tools 中额外进行安装和编译，并需要将它们添加到相关的环境变量文件 kaldi/tools/env.sh 中（安装时一般自动添加，若未添加，则需要自行添加）。

安装 SRILM 工具：

```
[root@localhost s5]# cd /work/kaldi/tools
[root@localhost tools]# extras/install_srilm.sh
Installing libLBFGS library to support MaxEnt LMs
checking for a BSD-compatible install... /usr/bin/install -c
...
SRILM config is already in env.sh
Installation of SRILM finished successfully
```

安装 kaldi_lm 工具：

```
[root@localhost tools]# extras/install_kaldi_lm.sh
Installing kaldi_lm
...
Installation of kaldi_lm finished successfully
Please source tools/env.sh in your path.sh to enable it
```

在 aishell 工程中，默认使用 kaldi_lm 工具训练语言模型。安装完成后，开始训练语言模型：

```
>> run.sh
local/aishell_train_lms.sh || exit 1;
```

（3）将语言模型转换为有限状态转换器形式

只有将语言模型转换为有限状态转换器形式后，才能合成 HCLG。

```
>> run.sh
utils/format_lm.sh data/lang data/local/lm/3gram-mincount/lm_unpruned.gz \
  data/local/dict/lexicon.txt data/lang_test || exit 1;
```

将语言模型转换完成之后，再提取声学特征并训练声学模型。而在训练完任一声学模型（如单音子和三音子等声学模型）后，就可将其和已准备好的词典、语言模型等合成 HCLG，并通过开发集和测试集的文本解码来测试当前声学模型的性能。

12.3.4 声学特征提取与倒谱均值归一化

Kaldi 支持的声学特征有很多，如 FBank、MFCC、PLP、Spectrogram 和 Pitch 等特征。在语音识别中，通常使用 MFCC、FBank 和 PLP 以获得更好的性能。不同的声学特征适合不同的模型。例如，GMM-HMM 比较适合带差分的低维 MFCC 特征；而 DNN-HMM 适合有更多维度的 MFCC 或者 FBank 特征，甚至结合卷积网络可直接使用语谱图（Spectrogram）特征。由于在 Kaldi 中 DNN-HMM 的训练并不是端到端的，所以在训练时仍然需要帧级别的对齐标注。DNN-HMM 的训练属于有监督的判别式模型训练，为了保证 DNN-HMM 的效果，我们需要先额外训练几轮 GMM-HMM 以获得足够可靠的对齐标注。而如前所述，不同的模型采用不同的声学特征，因此，在整个训练流程中，我们需要为 GMM-HMM 和 DNN-HMM 各自提取一份声学特征。配置声学特征可使用很多属性，如是否计算能量、采样率、上下截止频率、mel 卷积核数等。

在 aishell 工程中可以查看声学特征配置文件：

```
[root@localhost s5]# ls conf/
decode.config  mfcc.conf  mfcc_hires.conf  online_cmvn.conf  online_pitch.conf
pitch.conf
```

用于 GMM-HMM 训练的 MFCC 特征配置如下：

```
[root@localhost s5]# cat conf/mfcc.conf
--use-energy=false   # only non-default option.
```

```
--sample-frequency=16000
```

用于 DNN-HMM 训练的高精度 MFCC 特征配置如下：

```
[root@localhost s5]# cat conf/mfcc_hires.conf
# config for high-resolution MFCC features, intended for neural network training
# Note: we keep all cepstra, so it has the same info as filterbank features,
# but MFCC is more easily compressible (because less correlated) which is why
# we prefer this method.
--use-energy=false   # use average of log energy, not energy.
--sample-frequency=16000 # Switchboard is sampled at 8kHz
--num-mel-bins=40     # similar to Google's setup.
--num-ceps=40     # there is no dimensionality reduction.
--low-freq=40    # low cutoff frequency for mel bins
--high-freq=-200 # high cutoff frequently, relative to Nyquist of 8000 (=3800)
```

MFCC 额外附带的 Pitch 特征配置如下：

```
[root@localhost s5]# cat conf/pitch.conf
--sample-frequency=16000
```

通常，Pitch 特征一共有 3 维，可附加在 MFCC、FBank 等主要声学特征后。Pitch 特征的采样率需要与主要声学特征的采样率一致。

提取用于 GMM-HMM 训练的 MFCC+Pitch 特征，以及计算相应的倒谱均值：

```
>> run.sh
mfccdir=mfcc
for x in train dev test; do
  steps/make_mfcc_pitch.sh --cmd "$train_cmd" --nj 10 data/$x exp/make_mfcc/$x
$mfccdir || exit 1;  # 提取 MFCC+Pitch 特征
  # 计算倒谱均值
  steps/compute_cmvn_stats.sh data/$x exp/make_mfcc/$x $mfccdir || exit 1;
  utils/fix_data_dir.sh data/$x || exit 1; # 检查映射目录
done
```

在提取特征和计算倒谱均值之后，在 3 个数据映射目录中会分别生成 feats.scp 和 cmvn.scp 文件，其中，每行第二列的数据地址均指向已存储在 mfcc 目录中的以 ark 为后缀的数据文件，该数据文件中存储了压缩二进制矩阵数据。需要注意的是，倒谱均值是单独存储的，声学特征仍然是最原始的声学特征，而不是被归一化后的，真正的归一化在训练时动态进行。

12.3.5 声学模型训练与强制对齐

在声学模型的训练中，最初的对齐标注是在训练单音子模型中进行强制对齐时产生的，但是单音子模型的训练并没有用对齐标注，而是采用了等距切分的方法，将一句话的总帧数平均分配给该句话对应的每个音素，然后通过不断的迭代训练以获得更精准的对齐。有了最初的标注后，再通过几轮对基于三音子的 GMM-HMM 的继续优化，可以生成更精准的对齐标注，最终将这些对齐标注用于 DNN-HMM 的训练。

1. 训练 GMM-HMM

训练单音子模型，并强制对齐训练集，生成对齐标注：

```
>> run.sh
# 单音子模型训练
steps/train_mono.sh --cmd "$train_cmd" --nj 10 \
  data/train data/lang exp/mono || exit 1;
# 强制对齐
steps/align_si.sh --cmd "$train_cmd" --nj 10 \
  data/train data/lang exp/mono exp/mono_ali || exit 1;
```

训练三音子模型，并强制对齐训练集，生成对齐标注：

```
# 三音子模型训练
steps/train_deltas.sh --cmd "$train_cmd" \
2500 20000 data/train data/lang exp/mono_ali exp/tri1 || exit 1;
steps/align_si.sh --cmd "$train_cmd" --nj 10 \
  data/train data/lang exp/tri1 exp/tri1_ali || exit 1; # 强制对齐
```

反复训练三音子模型，并强制对齐训练集，生成对齐标注：

```
# 反复训练三音子模型
steps/train_deltas.sh --cmd "$train_cmd" \
 2500 20000 data/train data/lang exp/tri1_ali exp/tri2 || exit 1;
# 强制对齐
steps/align_si.sh --cmd "$train_cmd" --nj 10 \
  data/train data/lang exp/tri2 exp/tri2_ali || exit 1;
# 训练 LDA+MLLT 三音子模型
steps/train_lda_mllt.sh --cmd "$train_cmd" \
 2500 20000 data/train data/lang exp/tri2_ali exp/tri3a || exit 1;
# 强制对齐
steps/align_fmllr.sh --cmd "$train_cmd" --nj 10 \
  data/train data/lang exp/tri3a exp/tri3a_ali || exit 1;
# 训练基于 SAT+fMLLR 的三音子模型
```

```
steps/train_sat.sh --cmd "$train_cmd" \
  2500 20000 data/train data/lang exp/tri3a_ali exp/tri4a || exit 1;
# 强制对齐
steps/align_fmllr.sh  --cmd "$train_cmd" --nj 10 \
  data/train data/lang exp/tri4a exp/tri4a_ali
# 用更多的数据集训练基于SAT+fMLLR的三音子模型
steps/train_sat.sh --cmd "$train_cmd" \
  3500 100000 data/train data/lang exp/tri4a_ali exp/tri5a || exit 1;
```

2. 训练 DNN-HMM

在训练完 GMM-HMM 之后，在训练 TDNN 模型之前，需要改用高精度的 MFCC 特征。此外，在 TDNN 模型的训练中，还可以做数据扩增来进一步提升效果。因此，我们首先对训练集做数据扩增。aishell 工程的 run.sh 脚本可以调用 local/nnet3/run_ivector_common.sh 来处理所有这些步骤。为了条理清晰，这里专门将相关脚本拿出来独立执行。

变速扩增数据：

```
[root@localhost s5]# utils/data/perturb_data_dir_speed_3way.sh data/train
data/train_sp
```

提取变速扩增数据的低精度声学特征并基于 GMM-HMM 生成对齐文件：

```
[root@localhost s5]# steps/align_fmllr.sh --nj 30 --cmd "run.pl" data/train_sp
data/lang exp/tri5a exp/tri5a_sp_ali
```

在变速扩增数据的基础上，额外加入音量扰动并提取高精度声学特征以用于训练：

```
[root@localhost s5]# utils/copy_data_dir.sh data/train_sp data/train_sp_hires
[root@localhost s5]# utils/data/perturb_data_dir_volume.sh data/train_sp_hires
[root@localhost s5]# steps/make_mfcc_pitch.sh --nj 10 --mfcc-config
conf/mfcc_hires.conf --cmd "run.pl" data/train_sp_hires exp/make_hires/
train_sp_hires mfcc
[root@localhost s5]# steps/compute_cmvn_stats.sh data/train_sp_hires
exp/make_hires/ train_sp_hires mfcc
```

神经网络的训练除需要准备应有的训练集、对齐标注和 Lattice 文件外，其余步骤均分为配置神经网络和训练模型两部分。有关模型的配置结构可查看 steps/libs/nnet3/xconfig/目录下关于神经网络层的 Python 定义，训练参数配置可查看 steps/nnet3/train*.py 和 steps/nnet3/chain/train*.py 系列脚本。

在 aishell 工程中训练 TDNN 模型和 Chain 模型：

```
>> run.sh
local/nnet3/run_tdnn.sh
local/chain/run_tdnn.sh # 以上步骤均包含在总脚本的调用中
```

需要注意的是，Chain 网络的训练基本同 TDNN 网络的训练，但需要额外使用 GMM-HMM 生成 Lattice 文件：

```
>> local/chain/run_tdnn.sh
steps/align_fmllr_lats.sh --nj $nj --cmd "$train_cmd" data/$train_set \
   data/lang exp/tri5a exp/tri5a_sp_lats
```

12.3.6 解码测试与指标计算

在 Kaldi 中，每个声学模型训练完成后，都可以进行解码测试。解码分为两步：生成 HCLG 和生成 Lattice 文件并解码。生成 HCLG 使用含语言模型的 data/lang_test 目录。

生成 HCLG：

```
>> run.sh
# GMM-HMM 和普通 TDNN 的 HCLG 生成方式，示例为单音子模型
utils/mkgraph.sh data/lang_test exp/mono exp/mono/graph
>> local/chain/run_tdnn.sh
#基于 Chain 模型生成 HCLG，要附带额外的参数
utils/mkgraph.sh --self-loop-scale 1.0 data/lang_test $dir $dir/graph
```

开发集和测试集解码：

```
>> run.sh
# 无 fMLLR 的 GMM-HMM 解码
steps/decode.sh --cmd "$decode_cmd" --config conf/decode.config --nj 10 \
  exp/mono/graph data/dev exp/mono/decode_dev
steps/decode.sh --cmd "$decode_cmd" --config conf/decode.config --nj 10 \
  exp/mono/graph data/test exp/mono/decode_test
# 基于 fMLLR 的 GMM-HMM 解码
steps/decode_fmllr.sh --cmd "$decode_cmd" --nj 10 --config conf/decode.config \
  exp/tri4a/graph data/dev exp/tri4a/decode_dev
steps/decode_fmllr.sh --cmd "$decode_cmd" --nj 10 --config conf/decode.config \
  exp/tri4a/graph data/test exp/tri4a/decode_test

>> local/nnet3/run_tdnn.sh
# 普通 DNN-HMM 的解码
steps/nnet3/decode.sh --nj $num_jobs --cmd "$decode_cmd" \
```

```
        --online-ivector-dir exp/nnet3/ivectors_${decode_set} \
        $graph_dir data/${decode_set}_hires $decode_dir || exit 1;
>> local/chain/run_tdnn.sh
# 基于 Chain 的 DNN-HMM 的解码，需要额外增加参数
steps/nnet3/decode.sh --acwt 1.0 --post-decode-acwt 10.0 \
     --nj 10 --cmd "$decode_cmd" \
     $graph_dir data/${test_set}_hires $dir/decode_${test_set} || exit 1;
```

解码完成后，通过匹配测试集的真实标注，可以计算出 WER 和 CER 指标，其中，前者通常用于英文，后者用于中文。

指标的计算通常由 decode*.sh 解码系列脚本调用 local/score.sh 来完成：

```
>> local/score.sh
# WER 指标的计算
local/score.sh $scoring_opts --cmd "$cmd" $data $graphdir $dir
# CER 指标的计算
steps/scoring/score_kaldi_cer.sh --stage 2 $scoring_opts --cmd "$cmd" $data
$graphdir $dir
```

从 scoring_kaldi 目录下的 best_wer 文件中，我们可以获得最优的 WER 指标，例如 Chain 模型的结果：

```
%WER 15.95 [ 10279 / 64428, 874 ins, 1704 del, 7701 sub ]
exp/chain/tdnn_1a_sp/decode_test/wer_12_0.5
```

从 scoring_kaldi 目录下的 best_cer 文件中，我们可以获得最优的 CER 指标，例如 Chain 模型的结果：

```
%WER 7.47 [ 7822 / 104765, 329 ins, 424 del, 7069 sub ]
exp/chain/tdnn_1a_sp/decode_test/cer_10_0.5
```

图 12-3 给出了 aishell-1（178 小时）的实验结果，同时对比了不同声学模型的 CER 指标。

图 12-3　aishell-1 实验结果

12.4　本章小结

本章详细介绍了 Kaldi 系统的构建过程，包括数据准备、特征提取、模型训练和解码过程。Kaldi 训练脚本众多，配置烦琐，入门比较困难，除了本章介绍的 aishell 例子，读者也可采用清华大学开源的 THCHS-30 训练脚本，逐步掌握 GMM-HMM、DNN-HMM 和 Chain 模型的训练流程。随着对 Kaldi 工程的深入学习，读者可再替换为 DNN 结构，采用 LSTM、GRU、TDNN-F 等配置，进一步对比不同网络的识别性能。