

# CRF++ 使用

## 一、安装

1. CRF++ 是基于 C++ 开发，支持跨平台。安装时必须有 C++ 编译器。

安装命令：（首先进入到源码所在的目录）

```
./configure
make
sudo make install
```

2. 可以通过 `./configure --help` 命令来查看一些配置选项。常用的选项有：

```
-h, --help # 显示帮助信息
-V, --version # 显示版本信息
-q --quiet # 不打印 'checking ...' 信息
--prefix=PREFIX # 将系统无关的文件安装在 PREFIX 中，默认为 /user/local
--exec-prefix=EPREFIX # 将系统相关的文件安装在 EPREFIX 中，默认为 PREFIX
```

## 二、使用

### 2.1 数据文件格式

1. 如果要使用 CRF++，则训练文件、测试文件必须满足特定的格式：
  - 文件由很多 token 组成，每个 token 占据一行，包含固定数量的字段。
  - 所有 token 的字段数量相等，字段的数量没有限制，字段之间用空白分隔（空格符或者 tab 符）。
  - 每个字段通常表示某种含义。如：第一列表示 单词、第二列表示 词性、第三列表示 属性 ...。
  - 一个 sentence 由多个 token 表述，sentence 之间通过空行来区分边界。
  - 训练文件中，最后一个字段必须是标记，它将作为 CRF++ 训练的目标。

### 2.2 模板文件

1. CRF++ 使用模板文件来生成特征。模板文件需要用户编写，从而指定需要生成哪些特征。
2. 模板文件中，每一行都定义了一个特征模板。
3. 模板文件中，以 # 开头的行是注释行。

空行也会被认为是注释行而被剔除。
4. 有两种类型的特征模板，它们通过特征模板的第一个字符来区分。
  - Unigram 特征模板：模板的第一个字符串为 U，这种特征模板用于描述 unigram 特征。
  - Bigram 特征模板：模板的第一个字符串为 B，这种特征模板用于描述 bigram 特征。

#### 2.2.1 宏语句

1. 特征模板中，经常使用宏语句 `%x[row,col]`。其中：

- `%x` 是固定的，是宏语句的引导字符。
- `row` 是一个整数，指定了相对于当前的数据行的行数。
- `col` 是一个整数，指定了采用第几个字段（从 0 开始编号）。

注意：标记列不能作为特征，因此也就不能出现在特征模板中。

2. 假设输入数据为：

```
He      PRP  B-NP
reckons VBZ  B-VP
the      DT   B-NP << 当前行
current  JJ   I-NP
account  NN   I-NP
```

则下列特征模板为：

```
%x[0,0]      -->  the
%x[0,1]      -->  DT
%x[-1,0]     -->  reckons
%x[-2,1]     -->  PRP
%x[0,0]/%x[0,1] --> the/DT
ABC%x[0,1]123 --> ABCDT123
```

## 2.2.2 Unigram 特征模板

1. 给定一个 Unigram 特征模板 `U01:%x[0,1]`，它会生成  $M$  个特征函数，其中  $M$  为训练数据的行数（剔除空白行，因为空白行是 `sentence` 的分隔符）。

每个特征函数为：

```
func1 = if (output = LABEL1 and feature="U01:xx1") return 1 else return 0
func2 = if (output = LABEL2 and feature="U01:xx2") return 1 else return 0
func3 = if (output = LABEL3 and feature="U01:xx3") return 1 else return 0
....
funcM = if (output = LABELM and feature="U01:xxM") return 1 else return 0
```

其中：

- `LABEL1,...,LABELM` 就是训练文件中，每一行的标记。
- `feature="U01:xx1",...,feature="U01:xxM"` 就是训练文件中，每一行由 `U01:%x[0,1]` 指定的、从该行提取到的特征。

2. 事实上，上述生成的特征函数会有大量重复。

假设标记的种类一共有  $L$  个，由 `U01:%x[0,1]` 指定的、从该行提取到的特征的种类一共有  $N$  个，则特征函数的种类一共有  $L \times N$  个。

`CRF++` 会按照  $L$  种标记， $N$  种特征来自动生成  $L \times N$  个特征函数。

## 2.2.3 Bigram 特征模板

1. 给定一个 `Bigram` 特征模板 `B01:%x[0,1]`，它会生成  $M$  个特征函数，其中  $M$  为训练数据的行数（剔除空白行，因为空白行是 `sentence` 的分隔符）。

每个特征函数为：

```
func2 = if (output = LABEL2/LABEL1 and feature="U01:xx2") return 1 else return 0
func3 = if (output = LABEL3/LABEL2 and feature="U01:xx3") return 1 else return 0
func4 = if (output = LABEL4/LABEL3 and feature="U01:xx4") return 1 else return 0
....
funcM = if (output = LABELM/LABELM_1 and feature="U01:xxM") return 1 else return 0
```

其中：

- `LABEL1, ..., LABELM`、`feature="U01:xx1", ..., feature="U01:xxM"` 的意义与 `Unigram` 中的相同。
- 在 `Bigram` 中，特征函数中的 `output` 是当前的输出标记和前一个输出标记的联合，这也是它称作 `bigram` 的原因。

注意：它联合的是标记，而不是特征。特征的联合由宏语句来实现。

2. 上述生成的特征函数也会有大量重复。

假设标记的种类一共有  $L$  个，由 `U01:%x[0,1]` 指定的、从该行提取到的特征的种类一共有  $N$  个，则 `CRF++` 会按照  $L$  种标记， $N$  种特征自动生成  $L \times L \times N$  个特征函数。

3. 当标记的种类  $L$  较大时，`Bigram` 会生成非常多的特征函数，其中非常多的特征函数在样本中的返回值只有少量的 `1`。

这中情况下，模型的训练和测试将会非常低效。

4. 如果某一行的内容只有一个字符 `B`，则它表示：由当前的输出标记和前一个输出标记的联合生成的特征函数。

## 2.2.4 模板标识符

1. 在 `Unigram` 特征模板和 `Bigram` 特征模板中，在 `U` 或者 `B` 之后往往跟随一个数字作为标识符。
2. 标识符的作用是区分不同模板生成的特征。

例如：

|     |    |             |
|-----|----|-------------|
| The | DT | B-NP        |
| pen | NN | I-NP        |
| is  | VB | B-VP << 当前行 |
| a   | DT | B-NP        |

- 如果有标识符，则以下两个模板生成的特征函数为：
  - `U01:%x[-2,1]`：if (output = B-VP and feature="U01:DT") return 1 else return 0
  - `U02:%x[1,1]`：if (output = B-VP and feature="U02:DT") return 1 else return 0
- 如果没有标识符，则以下两个模板生成的特征函数为：
  - `U:%x[-2,1]`：if (output = B-VP and feature="U:DT") return 1 else return 0
  - `U:%x[1,1]`：if (output = B-VP and feature="U:DT") return 1 else return 0

可见这两个模板生成的特征函数无法区分。

3. 如果你需要使用 `Bag Of Words: BOW` 特征, 则你可以不使用模板标识符。

如果你需要考虑词序, 则必须使用模板标识符。

## 2.3 训练

1. 训练也称作 `encoding`, 是通过 `crf_learn` 程序来完成的。
2. 训练的命令为:

```
crf_learn template_file train_file model_file
```

其中:

- `template_file`: 人工编写的模板文件
- `train_file`: 人工标注的训练文件
- `model_file`: `CRF++` 生成的模型文件

3. 训练的输出内容如下:

```
CRF++: Yet Another CRF Tool Kit
Copyright(C) 2005 Taku Kudo, All rights reserved.

reading training data: 100.. 200.. 300.. 400.. 500.. 600.. 700.. 800..
Done! 1.94 s

Number of sentences: 823
Number of features: 1075862
Number of thread(s): 1
Freq:                1
eta:                  0.00010
C:                    1.00000
shrinking size:      20
Algorithm:            CRF

iter=0 terr=0.99103 serr=1.00000 obj=54318.36623 diff=1.00000
iter=1 terr=0.35260 serr=0.98177 obj=44996.53537 diff=0.17161
...
```

其中:

- `iter`: 表示迭代次数
- `terr`: 表示标记的训练错误率, 它等于  $\frac{\text{标记的训练错误数量}}{\text{标记的总数}}$ 。
- `serr`: 表示 `sentence` 的训练错误率, 它等于  $\frac{\text{sentence的训练错误数量}}{\text{sentence的总数}}$ 。
- `obj`: 当前的目标函数值。当目标函数值收敛到某个固定值时, `CRF++` 停止迭代。
- `diff`: 目标函数值的相对变化。它等于当前的目标函数值减去上一个目标函数值。

4. 常用训练参数:

- `-a CRF-L2` 或者 `-a CRF-L1`: 选择训练算法。  
`CRF-L2` 表示 `L2` 正则化的 `CRF`, 它也是 `CRF++` 的默认选择。`CRF-L1` 表示 `L1` 正则化的 `CRF`。
- `-c float`: 设置 `CRF` 的正则化项的系数  $C$ , `float` 是一个大于0的浮点数, 默认为 1.0。

如果  $C$  较大, 则 CRF++ 容易陷入过拟合。通过调整该参数, 模型可以在欠拟合和过拟合之间取得平衡。

- `-f NUM`: 设置特征的下限, `NUM` 是一个整数, 默认为 1。

如果某个特征 (由特征模板生成的) 发生的次数小于 `NUM`, 则该特征会被忽略。

当应用于大数据集时, 特征的种类可能到达上百万, 此时设置一个较大的 `NUM` 会过滤掉大部分低频特征, 提高模型的计算效率。

- `-p NUM`: 设置线程数量, `NUM` 是一个整数。

如果是多核 CPU, 则可以通过多线程来加速训练。`NUM` 表示线程的数量。

- `-t`: 同时生成文本格式的模型, 用于调试。
- `-e float`: 设置停止条件的阈值, `float` 是一个大于0的浮点数, 默认为 1.00.0001。
- `-v`: 显示版本并退出程序。
- `-m NUM`: 设置 LBFGS 的最大迭代次数, `NUM` 是一个整数, 默认为 10K。

5. 在 v0.45 以后的 CRF++ 版本中, 支持 single-best MIRA 训练算法。

Margin-infused relaxed algorithm:MIRA 是一种超保守在线算法, 在分类、排序、预测等应用领域取得不错成绩。

通过参数 `-a MIRA` 来选择 MIRA 算法。

- 输出:

```
CRF++: Yet Another CRF Tool Kit
Copyright(C) 2005 Taku Kudo, All rights reserved.

reading training data: 100.. 200.. 300.. 400.. 500.. 600.. 700.. 800..
Done! 1.92 s

Number of sentences: 823
Number of features: 1075862
Number of thread(s): 1
Freq: 1
eta: 0.00010
C: 1.00000
shrinking size: 20
Algorithm: MIRA

iter=0 terr=0.11381 serr=0.74605 act=823 uact=0 obj=24.13498 kkt=28.00000
iter=1 terr=0.04710 serr=0.49818 act=823 uact=0 obj=35.42289 kkt=7.60929
...
```

其中:

- `iter, terr, serr`: 意义与前面 CRF 相同
- `act`: `working set` 中, `active` 的样本的数量
- `uact`: 对偶参数达到软边界的上界  $C$  的样本的数量。

如果为 0, 则表明给定的训练样本是线性可分的。

- `obj`：当前的目标函数值  $\|\vec{w}\|^2$
- `kkt`：最大的 `kkt` 违反值。当它为 0.0 时，训练结束。

○ 参数：

- `-c float`：设置软边界的参数  $C$ ，`float` 是一个大于0的浮点数。

如果  $C$  较大，则 `CRF++` 容易陷入过拟合。通过调整该参数，模型可以在欠拟合和过拟合之间取得平衡。

- `-H NUM`：设置 `shrinking size`。

当一个训练 `sentence` 未能应用于更新参数向量 `NUM` 次时，认为该 `sentence` 不再对训练有用。此时 `CRF++` 会删除该 `sentence`。

当 `shrinking size` 较小时，会在早期发生收缩。这会大大减少训练时间。

但是不建议使用太小的 `shrinking size`，因为训练结束时，`MIRA` 会再次尝试所有的训练样本，以了解是否所有 `KKT` 条件得到满足。`shrinking size` 条小会增加重新检查的机会。

- `-f NUM`、`-e`、`-t`、`-p`、`-v`：意义与前面 `CRF` 相同

## 2.4 测试

1. 测试也称作 `decoding`，是通过 `crf_test` 程序来完成的。

2. 测试的命令为：

```
crf_test -m model_file test_file1 test_file2 ...
```

其中：

- `model_file`：由 `crf_learn` 生成的模型文件
- `test_file1, test_file2...`：多个测试文件。其格式与训练文件相同。

它将被 `crf_test` 添加一列（在所有列的最后）预测列

3. 常用参数：

```
crf_test -v0 -n 20 -m model test.data
```

- `v` 系列参数：指定输出的级别，默认为 `0` 级，即 `v0`。

级别越高，则输出的内容越多。其中：

- `-v0`：仅仅输出预测的标签。如：`B`。
- `-v1`：不仅输出预测的标签，还给出该标签的预测概率。如：`B/0.997`。
- `-v2`：给出每个候选标签的预测概率。如：`I/0.954883 B/0.00477976 I/0.954883 O/0.040337`。

注意：`v0` 也可以写作 `-v 0`。其它也类似。

- `-n NUM`：返回 `NUM` 个最佳的可能结果，结果按照 `CRF` 预测的条件概率来排序。

每个结果之前会给出一行输出：`# 结果序号 条件概率`。

## 三、Python接口

### 3.1 安装

1. 进入源码下的 `python` 目录，执行命令：

```
python3.6 setup.py build
python3.6 setup.py install
```

2. 如果希望安装到指定目录，则执行命令：

```
python3.6 setup.py install --prefix=PREFIX
```

### 3.2 使用

1. `CRF++` 并没有提供 `Python` 的训练结构，只提供了 `Python` 的测试接口。
2. `CRFPP.Tagger` 对象：调用解码器来解码。

```
CRFPP.Tagger("-m ../model -v 3 -n2")
```

创建对象，其中字符串中的内容就是 `crf_test` 程序执行的参数（不包含测试文件）。

- `.add('line')`：添加一行待解码的字段。
- `.clear()`：清除解码器的状态。
- `.parse()`：解码。它会修改解码器的状态。
- `.xsize()`：字段数量。
- `.size()`：样本行的数量。
- `.ysize()`：标记数量。

3. 使用示例：

```
import CRFPP
tagger = CRFPP.Tagger("-m ../model -v 3 -n2")
tagger.clear()
tagger.add("Confidence NN")
tagger.add("in IN")
tagger.add("the DT")
tagger.add("pound NN")
tagger.add("is VBZ")
tagger.add("widely RB")

print "column size: " , tagger.xsize()
print "token size: " , tagger.size()
print "tag size: " , tagger.ysize()

print "tagset information:"
ysize = tagger.ysize()
```

```

for i in range(0, ysize-1):
    print "tag " , i , " " , tagger.yname(i)

tagger.parse()
print "conditional prob=" , tagger.prob(), " log(Z)=" , tagger.Z()
size = tagger.size()
xsize = tagger.xsize()

for i in range(0, (size - 1)):
    for j in range(0, (xsize-1)):
        print tagger.x(i, j) , "\t",
        print tagger.y2(i) , "\t",
        print "Details",
        for j in range(0, (ysize-1)):
            print "\t" , tagger.yname(j) , "/prob=" , tagger.prob(i,j),"/alpha=" ,
            tagger.alpha(i, j),"/beta=" , tagger.beta(i, j),
            print "\n",

print "nbest outputs:"
for n in range(0, 9):
    if (not tagger.next()):
        continue
    print "nbest n=" , n , "\tconditional prob=" , tagger.prob()
    # you can access any information using tagger.y()...

```

## 四、常见错误

1. `crf++` 在 `linux` 上编译报错: `fatal error: winmain.h: No such file or directory`。

- 原因: `crf++` 考虑了跨平台, 而在 `linux` 上找不到该文件。
- 解决方案:

```

sed -i '/#include "winmain.h"/d' crf_test.cpp
sed -i '/#include "winmain.h"/d' crf_learn.cpp

```

2. 运行 `crf_learn` , 提示找不到 `libcrfpp.so.0`: `cannot open shared object file: No such file or directory`。

- 原因: 没有链接到库文件。
- 解决方案:

```

export LD_LIBRARY_PATH=/usr/local/lib/:$LD_LIBRARY_PATH

```

3. 运行 `crf_learn` , 提示: `inconsistent column size`。

- 原因: 语料库中, 出现了异常的标记行。
  - `crf++` 要求所有行的列数都相同。如果某些列出现了不同的列数, 则报错。
  - `crf++` 以 `\t` 或者空格分隔各列, 以空行来分隔 `sentence`。



4. 运行 `crf_learn` , 输出为: `reading training data: tagger.cpp(393) [feature_index_>buildFeatures(this)] 0.00 s` 。

- 原因: 模板文件编写不正确。

假设一共有 `N` 列, 则列编号必须为 `0~N-2` , 其中第 `N-1` 列为标签列, 不能进入模板中。

5. 编译 `python API` 时报错: `fatal error: Python.h: 没有那个文件或目录` 。

- 原因: 没有安装 `python3.6-dev`
- 解决方案:

```
sudo apt-get install python3.6-dev
```