一：作业完成情况

- 采用已有的英文名字，训练一个RNN，实现一个起名字的计算机程序，当输入名字的第一个或前几个字母时，程序自动生成后续的字母，直到生成一个名字的结束符。
- 采用可视化技术，绘制出模型为每个时刻预测的前5个最可能的候选字母。

> 也就是未完成附加题（划去）

二：目录说明

```
│ 报告.md
│
├─codes
│ │ EasyRnn_model.py # 包含了两个模型，一个失败版一个成功版
│ │ generate_name.py # 调用模型的函数
│ │ generate_name_plus.py # 相比上一个函数加了候选字母可视化
│ │ load_data.py # 加载数据用的
│ │ name_filter_8000.py # 规整数据用的
│ │ train.py # 训练函数
│ │
│ └─__pycache__
│       EasyRnn_model.cpython-311.pyc
│       load_data.cpython-311.pyc
│
├─data
│     AT_8000_M&F.csv # 澳大利亚八千人名
│     CN_8000_M&F.csv # 中国
│     description.txt # 数据集描述
│     JP_8000_M&F.csv # 日本
│     TR_8000_M&F.csv # 土耳其
│
├─imgs
│     Heatmap_of_easy_Rnn.png # 澳大利亚-名字生成热力图
│     Heatmap_of_easy_Rnn_CN.png # 中国
│     Heatmap_of_easy_Rnn_JP.png # 日本
│     Heatmap_of_easy_Rnn_TR.png # 土耳其
│
│
└─models
  │ name_generation_rnn2_AT_doublename.png
  │ name_generation_rnn2_AT_doublename.pth # 后面会提到

  │ name_generation_rnn2_AT_doublenamekongge.png
  │ name_generation_rnn2_AT_doublenamekongge.pth

  │ name_generation_rnn2_AT_signalname_doubleend.pth
  │ name_generation_rnn2_CN_doublenamekongge.png
```

```
│  name_generation_rnn2_CN_doublenamekongge.pth
│  name_generation_rnn2_JP_doublenamekongge.png

│  name_generation_rnn2_JP_doublenamekongge.pth
│  name_generation_rnn2_TR_doublenamekongge.png

│  name_generation_rnn2_TR_doublenamekongge.pth
│  training_curve_rnn2_AT_signalname_doubleend.png
│
└─default_model  # 没有加hidden隐藏层的版本
        name_generation_rnn_AT.pth
        name_generation_rnn_AT_100epoch.pth
        name_generation_rnn_AT_signalname.pth
        name_generation_rnn_AT_signalname_doubleend.pth
        training_curve_rnn_AT.png
        training_curve_rnn_AT_100epoch.png
        training_curve_rnn_AT_signalname.png
        training_curve_rnn_AT_signalname_doubleend.png
```

三：运行说明

- 需要修改代码文件里的所有目录，因为我用的都是绝对路径。
- 配置完毕后，运行对应的文件结构即可。

四：数据集

数据集来源

philipperemy/name-dataset: The Python library for names.

数据集描述

该数据集包含来自 **106** 个国家/地区的 **491,655,925** 条记录。未压缩版本在磁盘上占用约 **10GB** 的空间。

每个国家/地区的数据都存储在一个单独的 CSV 文件中。**FR** 代表法国，**DE** 代表德国，以此类推。

这里我用了四个国家的数据，分别是：

- 澳大利亚（**AT**）
- 中国（**CN**）
- 土耳其（**TR**）
- 日本（**JP**）

数据集剪切

因为题目里说的用 **8000** 个英文名字，外加我们这个任务可能也不需要很多的数据，所以我从他的 10GB 的数据里，分别从四个国家提取了 **8000** 个英文名字（有中文拼音）。

他的 **male** 和 **female** 有的是缺失的，比如 **TR** 整个数据集都没有 **male** 和 **female**，不过影响不大。

他这个数据集感觉有点乱乱的，有的名字不是英文的而是特殊字符（比如阿拉伯或者中文），需要先给他提取一下：

```
# 过滤掉 last name 为空或者不是英文
df_filtered = df[df.iloc[:, 1].notna() & df.iloc[:, 1].apply(is_english_name)]

# 过滤掉 first name 为空或者不是英文
df_filtered = df_filtered[df_filtered.iloc[:, 0].notna() & df_filtered.iloc[:, 0].apply(is_english_name)]
```

过滤条件为：

1. **last name** 为空或者不是英文（第二列）
2. **first name** 为空或者不是英文（第一列）

不过就算提取了之后，数据集看起来不是很好。比如中文名：

669	Bobo	Bobo	M	CN

感觉不会有中国人叫bobobobo?

9547	Oliver	Twist	M	CN

这不是查尔斯狄更斯的小说吗.....

然后数据集的声明为：

- Each record is a real person.
- Lists of names are not copyrightable, generally speaking, but if you want to be completely sure you should talk to a lawyer.

我对此持怀疑态度，不太明白他的名字是怎么收集到的。

五：字典配置与数据加载

字典配置

我们使用 **26个小写字母**、**26个大写字母**、**空格** 和 **终止符**（`<`）来构建字符字典：

```
# 创建一个字符映射字典，包括小写字母、大写字母、空格和结束符 '<'
special_end_symbol = '<'

char_to_index = {char: idx for idx, char in enumerate(
    string.ascii_lowercase + string.ascii_uppercase + ' ' + special_end_symbol)}

index_to_char = {idx: char for char, idx in char_to_index.items()}

input_size = len(char_to_index)  # 26小写 + 26大写 + 1空格 + 1结束符 = 54
```

下面是加载数据，空格和终止符由代码写入。

```
def load():
    df = pd.read_csv(r'C:\Users\49452\Desktop\神经网络 作业\刘逸-2022212054-A7\AT_8000_M&F.csv')  # 跳过文件的第一行，第二行是列名
    result = []

    for i, row in df.iterrows():
        name1 = row['Tc'] + ' ' + row['Kaplan'] + '<'  # 合并名字并添加结束符 '<'
        result.append({"number": i, "name": name1})

    return result
```

然后我尝试丢给dataset然后直接训练，但是 `stack` 函数在尝试将张量堆叠时遇到了尺寸不匹配的问题。这通常发生在 `DataLoader` 中批处理不同长度的序列时。名字的长度不同，因此无法直接将它们堆叠成一个统一的张量。所以我又写了个函数，在加载的时候选择这个批次长度最大数据作为这批数据的长度，没到达长度的终止符填充。

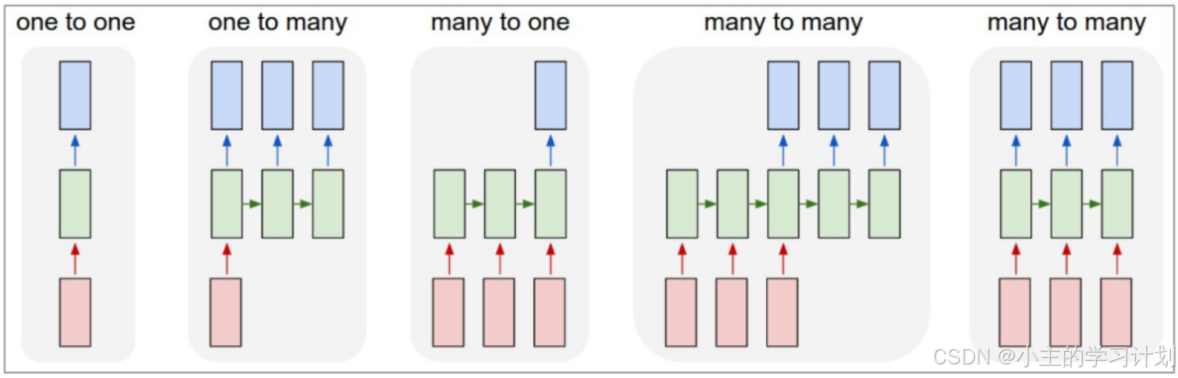
现在，对于一个名字他会输出两个串，一个是输入张量，一个是目标张量。

最后再加个随机种子和tqml。

RNN模型结构设计

1.失败案例：

RNN有几种模式：



我们要用第二种RNN，多输入多输出。

RNN除了接受每一步的输入x(t)，同时还会连接输入上一步的反馈信息——隐藏状态h(t-1)。

你可能觉得我的RNN从这里讲起有点突兀，但是很悲惨的是，第一次我的问题就出在这里，我想的是，我们先用最简单的全连接网络做RNN，网络的代码如下：

```
```
class NameGenerationRNNModel(nn.Module):

    def __init__(self, input_size, hidden_size, output_size):
        super(NameGenerationRNNModel, self).__init__()

        self.input_size = input_size # 输入大小（字符集的大小）
        self.hidden_size = hidden_size # 隐藏层大小
        self.output_size = output_size # 输出大小（字符集的大小）

        # 定义 RNN 层
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
```
```

```
        # 定义全连接层
        self.fc = nn.Linear(hidden_size, output_size)

        # Note: Removed Softmax because CrossEntropyLoss applies it internally

    def forward(self, x):
        out, _ = self.rnn(x)  # out: (batch_size, seq_len, hidden_size)
        out = self.fc(out)   # (batch_size, seq_len, output_size)
        return out
```

很标准的输入-隐层-输出的顺序，看着没有问题。

但是我在调用的时候，忘了当时是怎么写的了，导致代码的训练逻辑和生成逻辑是：

输入一个字符，然后预测下一个字符，再把这个预测的字符输入RNN，再预测一个字符，也就是，hidden每次都只携带了一个字符，他也只根据这一个字符来预测，而没有之前的时间步信息。

这就导致：

1.模型重复生成

比如，

`Jenininininini.....`

所以我的代码使用了 `random.choice` 从概率分布的前 `top_k` 个字符中随机选择一个字符，而不是选择最可能的字符。

2.不生成终止符

这一度让我十分苦恼，比如我输出过终止符的概率：

```
Probabilities for next character:
Character: a, Probability: 0.3419
Character: b, Probability: 0.0012
Character: c, Probability: 0.0105
Character: d, Probability: 0.0003
.......
Character: R, Probability: 0.0000
Character: S, Probability: 0.0000
Character: T, Probability: 0.0000
Character: U, Probability: 0.0000
Character: V, Probability: 0.0000
Character: W, Probability: 0.0000
Character: X, Probability: 0.0000
Character: Y, Probability: 0.0000
```

```
Character: Z, Probability: 0.0000
Character:  , Probability: 0.0000
Character: <, Probability: 0.0001
```

<的概率也太小了,然后我想过很多歪门邪道:

比如我在结尾的时候加两个结束符:

`name1 = row['Tc'] + ' ' + row['Kaplan'] + '<<'`

或者这样:

`name1 = row['Tc'] + '<<' + row['Kaplan'] + '<<'`

去掉字典里的空格，用终止符替代，输入是26个小写字母，26个大写字母加空格，也就是53个输出。
然后生成的逻辑变为：生成两个<后终止，代码如下:

```
# 如果生成结束符 '<'，停止生成
if index_to_char[chosen_index] == special_end_symbol:
    count += 1
    count_for_end = 0
    if count == 2:
        count = 0
        break
```

或者:

如果前五个字符有<，那么就选择<。

不过还是不行。所以你看我的default_mode里面有一大堆失败的模型。

所以我想着要不要在生成名字的时候增加输出<的概率:

```
termination_weight = 1 + count * 50  # 可以根据需要调整这个权重（例如每输出一个字符，<的权重增加5%）

probabilities[0, char_to_index['<']] *= termination_weight
```

然后我还测过不同概率叠加的效果:

- 50 概率

  Judevalorilaleri<Xeradolisilalemirirudol<
- 100概率

  Joledilulolevo<Vevu<
- 150概率

  Jolo<Xaronis<
- 200概率

  Jo<Memi<

我写到这里都想交作业了，不过我还是想从模型本身解决这个问题。检查了很久，终于找到了原因。

2.加入hidden:

模型rnn2结构如下:

```
class NameGenerationRNNModel2(nn.Module):
```

```
  def __init__(self, input_size, hidden_size, output_size):

    super(NameGenerationRNNModel2, self).__init__()

    self.input_size = input_size

    self.hidden_size = hidden_size

    self.output_size = output_size

    # 定义 RNN 层
    self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)

    # 定义全连接层，用于生成输出
    self.fc = nn.Linear(hidden_size, output_size)

    # Dropout 层用于防止过拟合
    self.dropout = nn.Dropout(0.1)

    # softmax 层用于输出概率分布
    self.softmax = nn.LogSoftmax(dim=2)  # 应用于 output_size 维度

  def forward(self, input, hidden):
    # RNN 层前向传播
    rnn_out, hidden = self.rnn(input, hidden)

    # 全连接层生成输出
    output = self.fc(rnn_out)

    # 应用 dropout 防止过拟合
    output = self.dropout(output)

    # 使用 softmax 得到每个字符的概率分布
    output = self.softmax(output)

    return output, hidden

  def initHidden(self, batch_size):
    # 初始化隐藏状态
    return torch.zeros(1, batch_size, self.hidden_size).to(device)
```

这样就没问题了。

1.尝试

中间我还尝试过很多种形式，比如去掉空格，这么load名字然后训练：

```
name1 = row['Tc'] + '<' + row['Kaplan'] + '<'
```

生成文件name_generation_rnn2_AT_doublename.pth

但是生成的时候，发现<后面输出的下一个符号一定是<，他就会生成一个名字类似于：

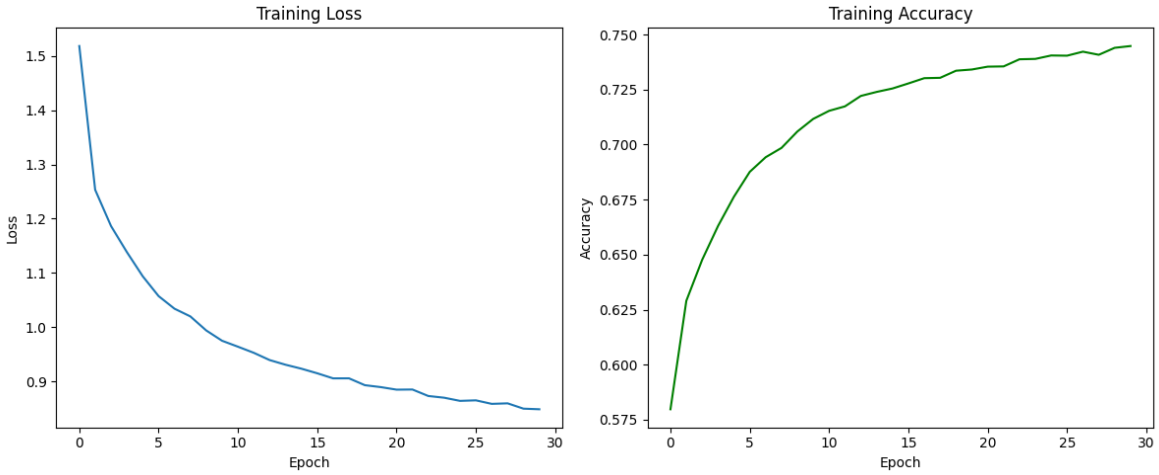
`Jarnia<<`

以及：文件name_generation_rnn2_AT_signalname_doubleend.pth的来源是：

这么load数据：

```
name1 = row['Tc'] + '<'
name2 = row['Kaplan'] + '<'
```

虽然生成还是有只能生成单个名字的问题，不过训练的两个数据（loss，训练集准确率）很好，可能是序列短的原因。

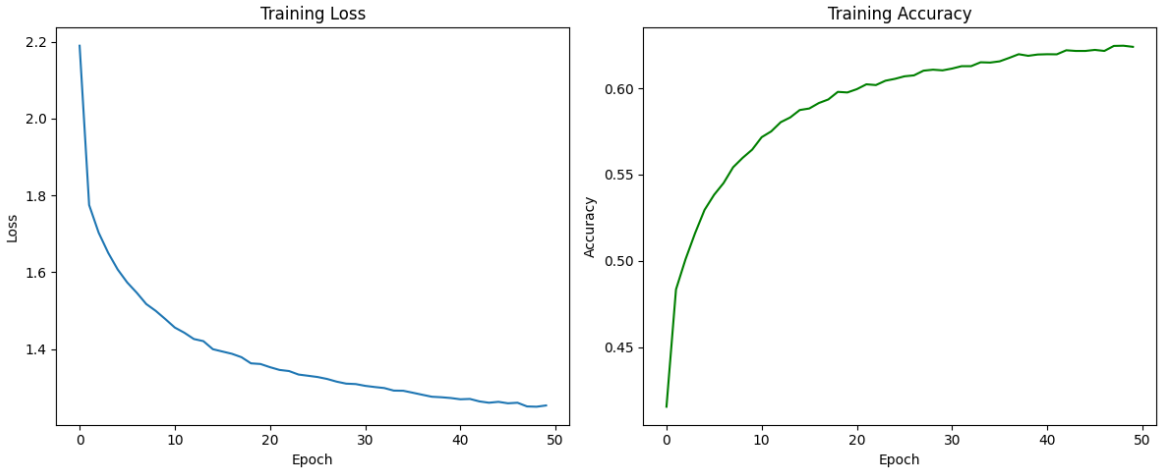


2.实验结果-训练

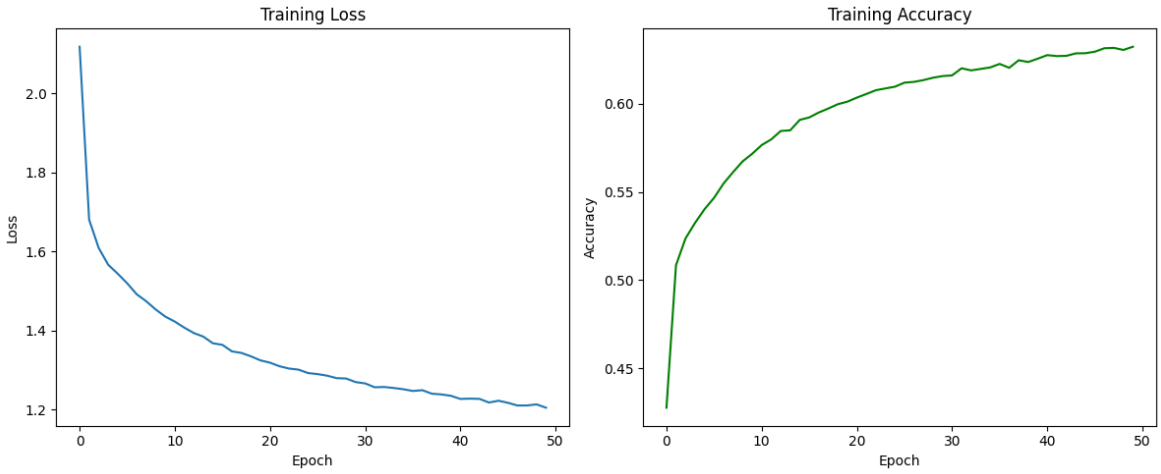
所以还是老老实实把中间的终止符变成了空格。

损失函数为交叉熵，轮数见图，batch为16。

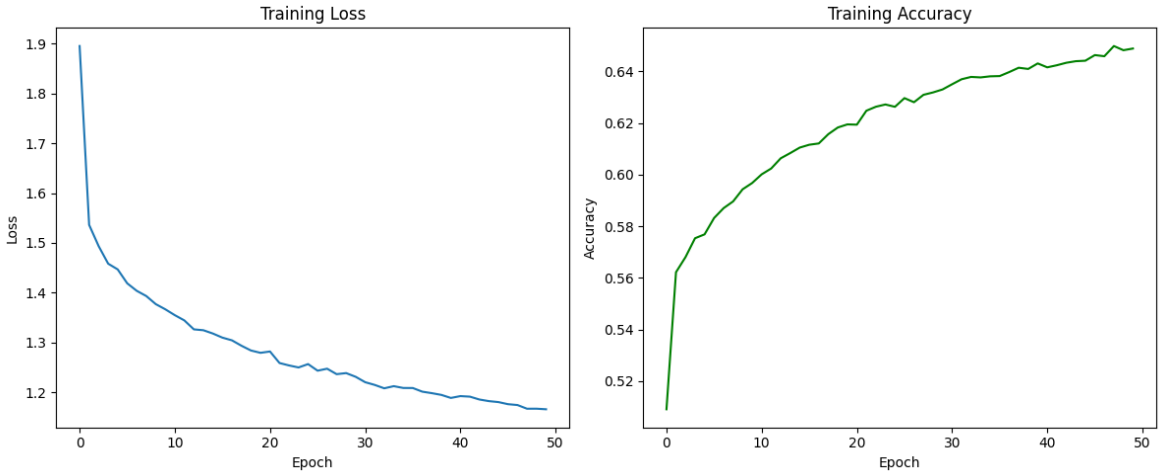
澳大利亚-训练数据（loss，训练集准确率，下同）



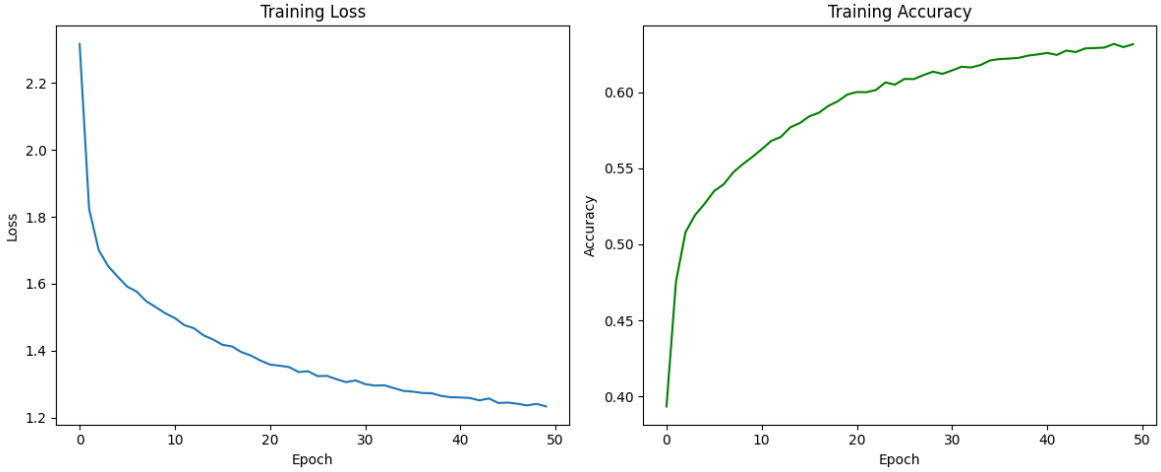
日本-训练数据



中国-训练数据



土耳其-训练数据



3.实验结果-生成

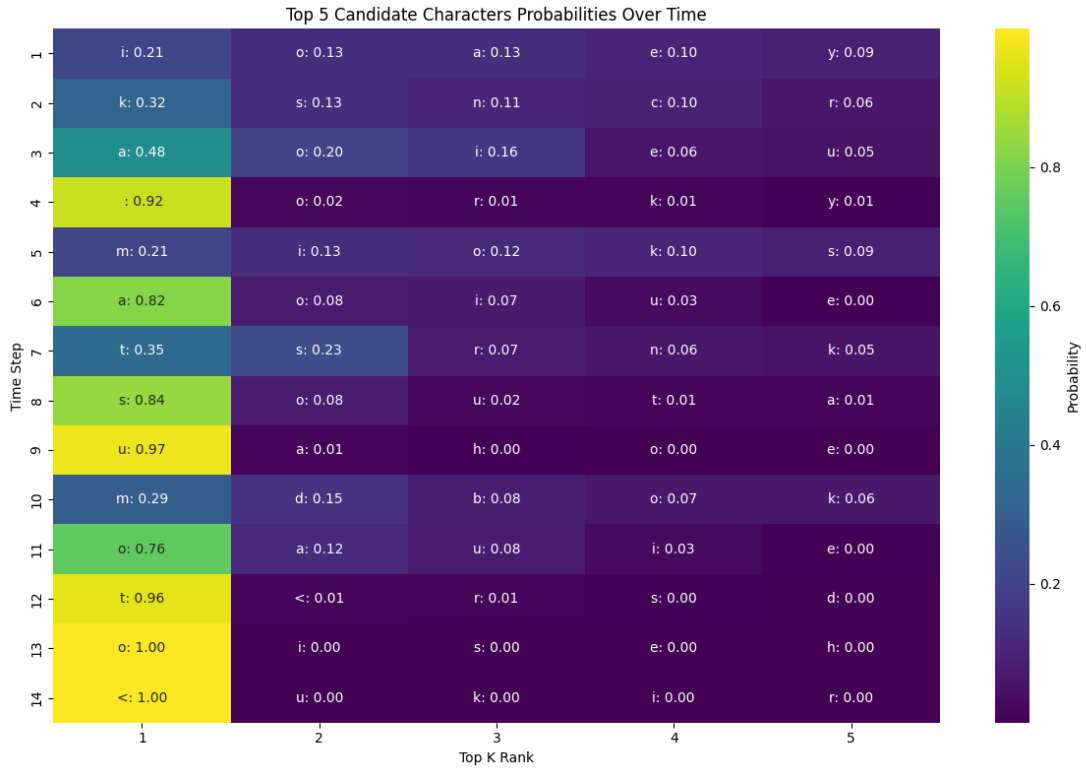
在 `generate_name` 函数中添加代码记录前5个候选字母及其概率，然后可以使用一个列表来存储这些信息。

接下来，我们使用 Matplotlib 来可视化 `prediction_history` 中的数据。这里我们选择使用热力图（Heatmap）来展示每个时间步前5个候选字母的概率。

日本-提示词Y

生成结果：Yika matsumoto<

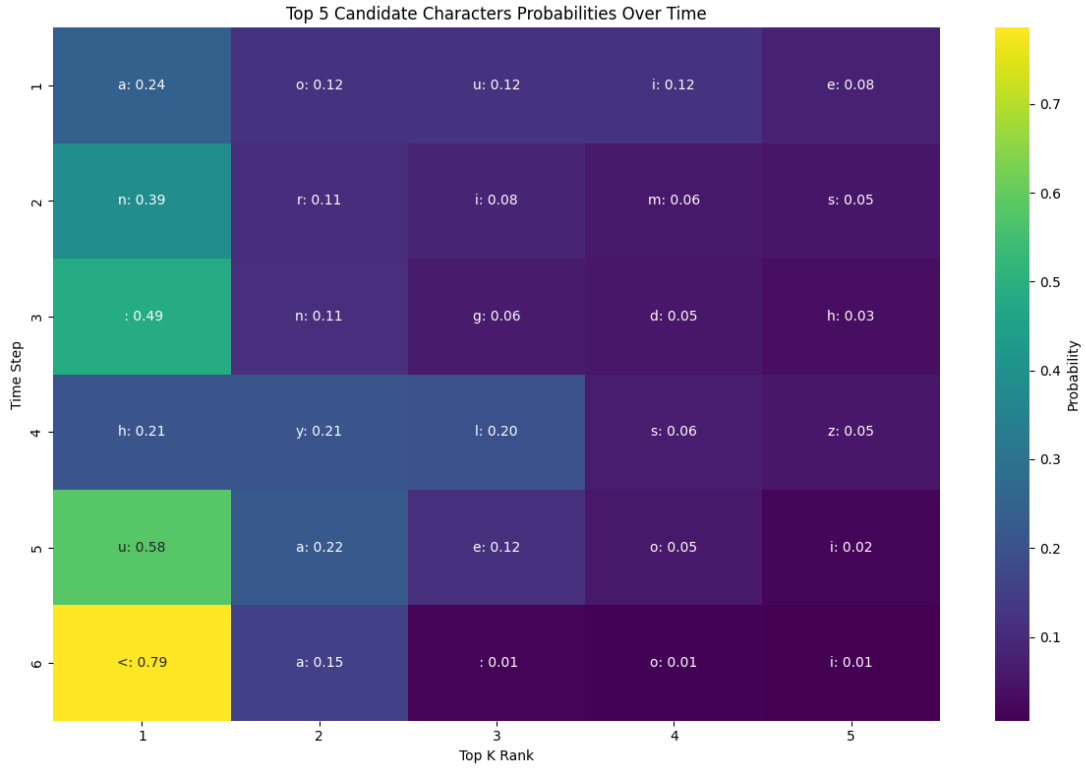
热力图：



中国-提示词Z

生成结果：Zan hu<

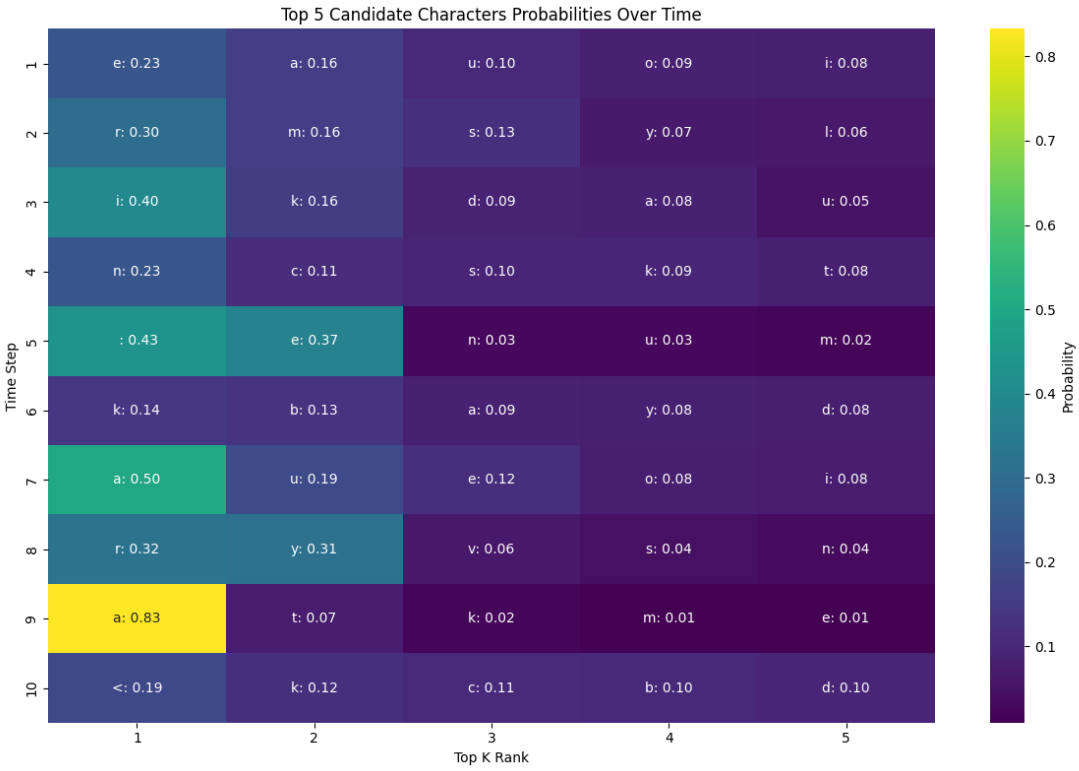
热力图：



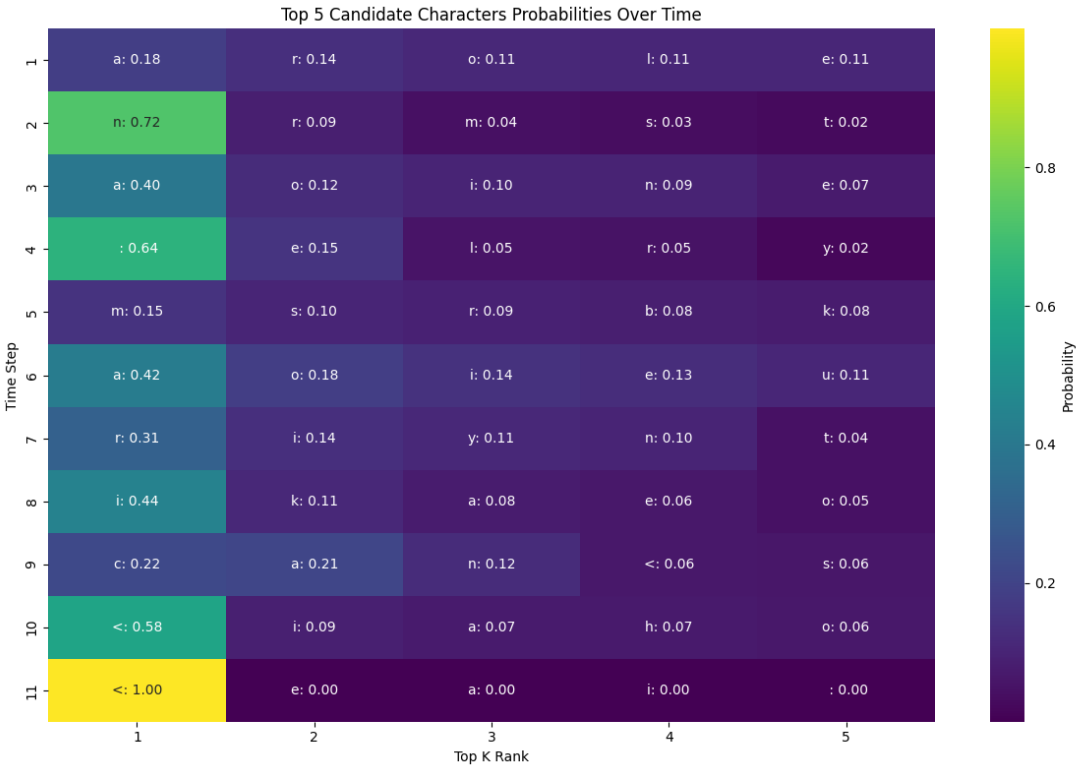
土耳其-提示词F

生成结果：Ferin kara<

热力图：



澳大利亚-热力图



总体看着还不错，除了不太了解土耳其的名字特色，看不太出来，但是日本和中国都是比较有当地的名字特色的。