

# 单 词 预 测 项 目 报 告

项目名称 下一个单词预测器

组员姓名 黄宇威、王润鹏、李爽、熊腾

报告时间 2022年8月20日

# 基于自然语言处理的单词预测器

黄宇威

2022 年 8 月 20 日

## 摘要

手机上的键盘如何知道您接下来要输入什么？本系统旨在设计一个单词预测器，根据输入的单词或词组给出下一个单词的建议，达到节省时间的目的。信息时代的不断发展使计算机人工智能的开发与应用越来越广泛。本系统基于自然语言处理中的分词技术，采用三种语言模型，最终实现单词预测的功能。

## 1 项目介绍

单词预测是一种自然语言处理 - NLP 应用程序，通过大量的短句或者词组训练模型，来实现单词预测，本文将介绍如何开发一个单词预测器，首先最重要的是选择语言模型，这里我们将采用三种模型用于开发：N-gram 自然语言模型、LSTM（长短期记忆人工神经网络）、NNLM神经网络语言模型。目的是为了对比不同方法的预测结果差异变化。

## 2 数据介绍

本文训练数据来源于github，数据采用txt文件格式存储，包含上万个英文短句以及词组，每个单词之间以空格符隔开，

短句用行的形式区分。部分数据示例如下图：

```
how a silencer works In U.S. law, the terms "firearm muffler" and "firearm silencer" are synonymous. 0
how a silencer works Suppressors were regularly used by agents of the United States Office of Strategic Services, who favored the newly designed
how a silencer works OSS Director William Joseph "Wild Bill" Donovan demonstrated the pistol for President Franklin D. Roosevelt at the White
how a silencer works According to OSS research chief Stanley Lovell, "Donovan, an old and trusted friend of the President, was waved into the
how a silencer works While Roosevelt finished his message, Donovan turned his back and fired ten shots into a hallway he had brought with him.
how a silencer works The British SAS "Special Operations Executive" Weirid assassination party, with an integral suppressor was also used by t
how long is taken blood vengeance movie is a 2011 Japanese 3D computer-animated film based on the Tekken video game series, produced by Digital
how long is taken blood vengeance movie The film was released in North America by Shochu Entertainment on July 24, 2011, and in Australia on 24
how long is taken blood vengeance movie It was released on September 3, 2011 in Japan. 0
how long is taken blood vengeance movie The film was released on Blu-ray Disc on November 23, 2011 for USA, and December 3, 2011 for Japan.
how long is taken blood vengeance movie This film was also released on DVD on November 23, 2011 (2D Version only). 0
how long is taken blood vengeance movie The film was additionally included with the PlayStation 3 game, Tekken Hybrid, and the Nintendo 3DS t
how can bus collect unpaid fees on property for a discussion of nonprofit, voluntary neighborhood advocacy groups, see neighborhood association.
how can bus collect unpaid fees on property in the United States, a homeowners association is a corporation formed by a real estate developer for t
how can bus collect unpaid fees on property it grants the developer privileged voting rights in governing the association, while allowing the deves
how can bus collect unpaid fees on property Membership in the homeowners association by a residential buyer is typically a condition of purchase.;
```

## 3 算法细节

### 3.1 LSTM 神经网络

#### 3.1.1 LSTM 算法原理

LSTM 神经网络是 RNN 网络的衍生物，是特殊的循环神经网络，能够克服 RNN 网络的缺点，用来处理长时依赖问题，经过大量的研究证明 LSTM 神经网络在时间序列预测问题上获得了更进一步的成功。LSTM 网络的结构与 RNN 相似，处理模块 A 更加复杂，标准 LSTM 结构如下图所示：

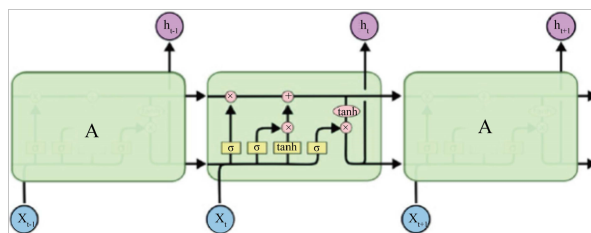


图 3.1: LSTM 网络标准结构

LSTM 网络结构细节解析:

1. 遗忘门(Forget Gate), 是 LSTM 的忘记阶段, 对上一节点的输入信息选择性忘记, 由一个被称为“遗忘门限层”的 sigmoid 层决定。根据上一时刻的输出和当前输入为单元状态的每一个数字计算 0 到 1 之间的数字, 0 表示“完全抛弃”, 1 表示“完全保留”。
2. 输入门(Input Gate), 是 LSTM 的记忆阶段, 随着时间不断更新状态值, 留下有用的状态值。过程分为两个部分, 首先 sigmoid 层决定需要更新的数值, 其次 tanh 层创建向量, 再结合二者创建一个状态更新。
3. 通过简单的线性交互, 更新旧单元状态, 输入到新单元状态, 即来完成对信息的更新。前两个部分已经决定需要哪些信息, 结合遗忘门和输入门的信息得到更新后的单元状态。
4. 输出门(Output Gate), 是 LSTM 的输出阶段, 建立在单元状态的基础上, 经遗忘门和输入门对信息的不断筛选过滤, 最终决定当前状态的输出。

### 3.1.2 LSTM 算法实现步骤

1. 预处理数据-读取文本, 使用 word\_tokenize 函数将句子分为单个单词。

代码如下:

```
1 # 读取文件
2 with open(path, encoding='utf-8') as f:
3     sents = f.read()
```

```
4 cleaned = re.sub(r'\W+', ' ', sents).lower()
5 # 将短句分为单词
6 tokens = word_tokenize(cleaned)
```

2. 使用 fit\_on\_texts 函数分词后, 在用 texts\_to\_sequences 函数对序列进行重新编码

```
1 tokenizer = Tokenizer()
2 # 实现分词
3 tokenizer.fit_on_texts(text_sequences)
4 # 输出向量序列
5 sequences = tokenizer.texts_to_sequences(text_sequences)
```

3. 获得编码形式的序列后, 通过将序列拆分为输入和输出标签来定义使用前三个词作为输入, 最后一个词作为模型要预测的标签

```
1 n_sequences = np.empty([len(sequences), train_len], dtype='int32')
2 for i in range(len(sequences)):
3     n_sequences[i] = sequences[i]
4 train_inputs = n_sequences[:, :-1]
5 train_targets = n_sequences[:, -1]
```

4. 将目标标签序列通过 to\_categorical 函数转换为 one-hot 向量, 即 0 和 1 的组合

```

1  # 将类别向量转换为二进制
2  train_targets =
    to_categorical(
        train_targets,
        num_classes=
            vocabulary_size)

```

5. 训练LSTM模型，一共有5层的Sequential模型：一个Embedding层、两个LSTM层和两个Dense层。在我们模型的输入层（即嵌入层）中，输入长度设置为序列的大小，在本例中为3

```

1  #模型创建或加载
2  def train_model(
        vocabulary_size,
        train_inputs,
        train_targets, seq_len):
3  if not os.path.exists("
        mymodel.h5"):
4  # 创建模型
5  model = Sequential()
6  model.add(Embedding(
        vocabulary_size,
        seq_len, input_length=
            seq_len))
7  model.add(LSTM(50,
        return_sequences=True))
8  model.add(LSTM(50))
9  model.add(Dense(50,
        activation='relu'))
10 model.add(Dense(
        vocabulary_size,
        activation='softmax'))
11 # compile network
12 model.compile(loss='

```

```

        categorical_crossentropy
        ', optimizer='adam',
        metrics=['accuracy'])
13 model.fit(train_inputs,
        train_targets, epochs
            =500, verbose=1)
14 model.save("mymodel.h5")
15 print('模型创建完成')
16 else:
17 # 加载模型
18 print('加载模型')
19 model = load_model("
        mymodel.h5")
20 return model

```

6. 输入测试数据，并从softmax函数中获取三个最可能的单词

```

1 input_text = input().strip().lower()
2 encoded_text = tokenizer.
    texts_to_sequences([
        input_text])[0]
3 # 对上面生成的不定长序列进行补全
4 pad_encoded = pad_sequences
    ([encoded_text], maxlen=
        seq_len, truncating='pre'
    )
5 print("输入:", input_text)
6 l = len(tokenizer.index_word
    )
7 # 开始预测
8 for i in (model.predict(
        pad_encoded)[0]).argsort
    ()[-3:][::-1]:
9     pred_word = tokenizer.
        index_word[i]

```

```
10 print("候选词:", pred_word)
```

## 3.2 N-gram 自然语言模型

### 3.2.1 N-gram 算法原理

N-gram是指给定的文本或语音序列中包含N个最小分割单元的连续序列。最小分割单元可以是音素、音节、字母、字或者是一些根据具体应用而自定义的基本对(BasicPairs)。

N-gram实际上是N-1阶马尔可夫语言模型的表示。假设一系列随机变量 $S_1, S_2, \dots, S_m$ 中, 如果其中任何一个随机变量 $S_i$ 发生的概率只与其前面的N-1个变量 $S_{i-1}, S_{i-2}, \dots, S_{i-n+1}$ 有关, 即:

$$P(S_i | S_{i-n+1} S_{i-n+2} \dots S_{i-2} S_{i-1}) = P(S_i | S_1 S_2 \dots S_{i-2} S_{i-1})$$

则称之为N-1阶马尔可夫过程。N-gram模型就是将所有连续可重叠的N个词作为一个单元, 并将其假设为一个N-1阶马尔可夫过程。这种假设的意义在于, 第N个词的出现只与前N-1个词相关, 而其他任何词都不相关。整个句子的概率是各个词出现概率的乘积, 而这些单个词的概率可由语料库中统计N个词同时出现的次数得到。

N-gram理论在信息检索研究中主要应用于检索预处理、索引、语种识别等先导性工作, 包括语音和文本分析领域。在语音分析领域, Torres-Carrasquillo等使用音素标记化结合N-gram将语言模型化来进行语言识别, 得到稳定的结果。

### 3.2.2 N-gram 算法实现步骤

1. 读取数据, 为每个唯一单词创建对,

每个唯一的单词作为键, 其后面的单词列表作为值被添加到我们的字典lookup\_dict中

代码如下:

```
1 def add_document(self,
2     string):
3     self.lookup_dict.clear()
4     with open(string, encoding=
5         'utf-8') as f:
6         sents = f.read()
7         preprocessed_list = self.
8             _preprocess(sents)
9         pairs = self.
10             __generate_tuple_keys(
11                 preprocessed_list)
12         #将生成的词组加入字典中
13         for pair in pairs:
14             self.lookup_dict[pair
15                 [0]].append(pair[1])
16         pairs2 = self.
17             __generate_2tuple_keys(
18                 preprocessed_list)
19         for pair in pairs2:
20             self.lookup_dict[tuple([
21                 pair[0], pair[1]])].
22                 append(pair[2])
23         pairs3 = self.
24             __generate_3tuple_keys(
25                 preprocessed_list)
26         for pair in pairs3:
27             self.lookup_dict[tuple([
28                 pair[0], pair[1],
29                 pair[2]])].append(
30                 pair[3])
31
32 def _preprocess(self, string
```

```

):
18 #将短句分为单词
19 cleaned = re.sub(r'\W+', ' ', string).lower()
20 tokenized = word_tokenize(cleaned)
21 return tokenized

```

2. 判断输入的字符是几个单词,然后从字典中查找下一个单词

```

1 # 根据空格符分隔词组
2 tokens = string.split(" ")
3 if len(tokens) == 1:
4 # 根据一个单词查找
5 txt = self.oneword(string)
6 elif len(tokens) == 2:
7 # 根据两个单词查找
8 txt = self.twowords(
    string.split(" "))
9 elif len(tokens) == 3:
10 # 根据三个单词查找
11 txt = self.threewords(
    string.split(" "))
12 elif len(tokens) > 3:
13 txt = self.morewords(
    string.split(" "))
14
15
16 #根据一个单词查找字典中的候选词
17 def oneword(self, string):
18 #统计出现次数最多的候选词
19 return Counter(self.
    lookup_dict[string]).
    most_common()[ :10]
20 #根据两个单词查找字典中的候选词
21 def twowords(self, string):

```

```

22 suggest = Counter(self.
    lookup_dict[tuple(
    string)]).most_common(
    )[:10]
23 if len(suggest) == 0:
24 return self.oneword(
    string[-1])
25 return suggest

```

### 3.3 NNLM神经网络语言模型

#### 3.3.1 NNLM 算法原理

NNLM是通过第 $t$ 个词前的 $n-1$ 个词,预测每个词在第 $t$ 个位置出现的概率,即:

$$f(w_t, \dots, w_{t-n+1}) = \hat{P}(w_t | w_1^{t-1})$$

其中 $f > 0$ , 即预测出的结果都是大于0的, 且预测结果的和为1, NNLM包含一个三层的神经网络, 具体模型结构如下:

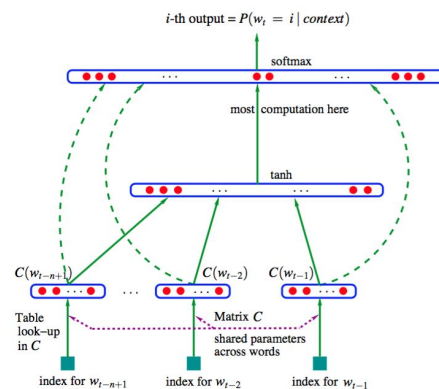


图 3.2: NNLM 模型结构

模型一共三层, 第一层是映射层, 将 $n$ 个单词映射为对应word embeddings的拼接, 其实这一层就是MLP的输入层; 第二层是隐藏层, 激活函数用tanh; 第三层是输出层, 因为是语言模型, 需要根据前 $n$ 个单词预测下一个单词, 所以是一个多分类器, 用softmax。

1. 读取数据, 添加到词典word\_list中, 并为每个单词创建下标索引

```

1  #按照空格分词,
2  #统计 的分词的个数 sentences
3  word_list = " ".join(
    sentences).split()
4  #去重 统计词典个数
5  word_list = list(set(
    word_list))
6  word_dict = {w: i for i, w
    in enumerate(word_list)}
7  number_dict = {i: w for i, w
    in enumerate(word_list)}

```

2. 创建NNLM模型

```

1  # Model
2  class NNLM(nn.Module):
3  def __init__(self):
4  super(NNLM, self).
    __init__()
5  # 矩阵Q (V x m) V
6  #表示的字典大小 word,
7  # m 表示词向量的维度
8  self.C = nn.Embedding(
    n_class, m)
9  self.H = nn.Linear(n_step
    * m, n_hidden, bias=
    False) ###
10 self.d = nn.Parameter(
    torch.ones(n_hidden))
11 self.U = nn.Linear(
    n_hidden, n_class,
    bias=False)
12 self.W = nn.Linear(n_step
    * m, n_class, bias=
    False)

```

```

13 self.b = nn.Parameter(
    torch.ones(n_class))
14
15 def forward(self, X):
16 X = self.C(X) # X : [
    batch_size, n_step, m]
17 X = X.view(-1, n_step * m
    ) # [batch_size,
    n_step * m]
18 tanh = torch.tanh(self.d
    + self.H(X)) # [
    batch_size, n_hidden]
19 output = self.b + self.W(
    X) + self.U(tanh) # [
    batch_size, n_class]
20 return output

```

3. 构建输入数据和目标标签

```

1 def make_batch():
2 input_batch = []
3 target_batch = []
4 for sen in sentences:
5 word = sen.split() #space
    tokenizer
6 input = [word_dict[n] for
    n in word[:-1]]
7 target = word_dict[word
    [-1]]
8 input_batch.append(input)
9 target_batch.append(
    target)
10 return input_batch,
    target_batch

```

4. 将构建的输入数据和目标标签转成tensor形式



```

1  # 训练模型迭代次    5000
2  for epoch in range(5000):
3      #梯度归零
4      optimizer.zero_grad()
5      output = model(
6          input_batch)
7      # output : [batch_size,
8          n_class], target_batch :
9          [batch_size]
10     loss = criterion(output,
11         target_batch)
12     if (epoch + 1) % 1000 ==
13         0:
14         print( 'Epoch: ', '%04d '
15             % (epoch + 1), 'cost
16             =', '{:.6f}'.format
17             (loss))
18     # 反向传播计算每个参数的梯度值
19     loss.backward()
20     # 每一个参数的梯度值更新
21     optimizer.step()
22     # 预测
23     predict = model(input_batch
24         ).data.max(1, keepdim=
25         True)[1]

```

#### 4.1 LSTM神经网络算法结果

```
['how', 'many', 'people', 'live', 'in', 'atlanta', 'georgia', 'atlanta',
```

分词后: [['how', 'many', 'people', 'live'], ['many', 'people', 'live', 'in'], ['people', 'live', 'in', 'atlanta'],  
分词后向量序列: [[6, 13, 22, 15], [13, 22, 15, 8], [22, 15, 8, 16], [15, 8, 16, 23], [8, 16, 23, 16], [16, 23, 16,

```
拆分前 [[ 6 13 22 15]]
拆分后输入目标标签 [[ 6 13 22]]
拆分后输出(预测)目标标签 [15]
```

[illegible]

```
Epoch 1/500
6/6 [=====] - 7s 9ms/step - loss: 4.3808 - accuracy: 0.0491

Epoch 2/500
6/6 [=====] - 0s 5ms/step - loss: 4.3746 - accuracy: 0.0552

Epoch 3/500
6/6 [=====] - 0s 5ms/step - loss: 4.3676 - accuracy: 0.0920

Epoch 4/500
6/6 [=====] - 0s 5ms/step - loss: 4.3579 - accuracy: 0.0982

Epoch 5/500
6/6 [=====] - 0s 5ms/step - loss: 4.3429 - accuracy: 0.0982

Epoch 6/500
6/6 [=====] - 0s 5ms/step - loss: 4.3196 - accuracy: 0.0982

Epoch 7/500
6/6 [=====] - 0s 5ms/step - loss: 4.2808 - accuracy: 0.0982

Epoch 8/500
6/6 [=====] - 0s 5ms/step - loss: 4.2189 - accuracy: 0.0982

Epoch 9/500
6/6 [=====] - 0s 5ms/step - loss: 4.1045 - accuracy: 0.0982
```



## 6. 预测结果

```

输入: how many people
1/1 [=====] - 1s 1s/step
候选词: died
候选词: visit
候选词: live

```

4. 上述测试数据how many输入了两次, 再次输入how时, many次数多了两次

```

输入: how
候选词: [('many', 1577),

```

## 4.2 N-gram自然语言模型算法结果

1. 预处理数据-读取文本, 使用word.tokenize函数将句子分为单个单词。

```
['how', 'many', 'people', 'live', 'in', 'atlanta', 'georgia', 'atlanta',
```

2. 创建键值对, 添加进字典lookup\_dict中

```

{ 'how': ['are', 'many', 'are'], 'are': ['you', 'your'],
  'you': ['how'], 'many': ['days'], 'days': ['since'],
  'since': ['we'], 'we': ['last'], 'last': ['met'], 'met': ['how'],
  'your': ['parents']
}

{
  ('how', 'are'): ['you', 'your'],
  ('how', 'many'): ['days'],
  ('many', 'days'): ['since'],
  ('how', 'are', 'you'): ['how'],
  ('how', 'many', 'days'): ['since'],_
}

```

3. 根据输入的单词预测, 测试输入一个单词, 两个单词, 三个单词的结果, 候选词的顺序以训练次数排序

```

输入: how
候选词: [('many', 1575), ('die', 459), ('door', 369), ('much', 369), ('is', 272), ('old', 193), ('do', 173), ('long', 168), ('are', 122), ('big', 89)]

输入: how many
候选词: [('people', 46), ('countries', 31), ('episodes', 30), ('seasons', 30), ('many', 30), ('live', 30), ('die', 30), ('are', 30), ('in', 30), ('visit', 30), ('have', 25), ('were', 18), ('was', 14), ('is', 14), ('die', 9)]

输入: how many people
候选词: [('died', 34), ('live', 27), ('visit', 26), ('have', 25), ('were', 18), ('are', 14), ('in', 14), ('die', 9)]

```

## 4.3 NNLM神经语言模型算法结果

1. 读取数据, 添加到词典word\_list中, 并为每个单词创建下标索引

```

word_list: ['countries', 'states', 'episodes', 'people', 'many', 'seasons', 'how']
word_dict: {'countries': 0, 'states': 1, 'episodes': 2, 'people': 3, 'many': 4, 'seasons': 5, 'how': 6}
number_dict: {0: 'countries', 1: 'states', 2: 'episodes', 3: 'people', 4: 'many', 5: 'seasons', 6: 'how'}

```

2. 构建输入数据和目标标签

```

input_batch [[6, 4], [6, 4], [6, 4], [6, 4], [6, 4]]
target_batch [3, 0, 2, 5, 1]

```

3. 输入数据和目标标签转成tensor形式

```

input_batch tensor([[[6, 4],
  [6, 4],
  [6, 4],
  [6, 4]]])
target_batch tensor([3, 0, 2, 5, 1])

```

4. 训练数据集以及每1000次的CrossEntropyLoss数据

```

Epoch: 1000 cost = 0.142009
Epoch: 2000 cost = 0.020039
Epoch: 3000 cost = 0.006503
Epoch: 4000 cost = 0.002758
Epoch: 5000 cost = 0.001322

```

5. 根据输入单词进行预测

```

输入: [['how', 'many']]
候选词: ['people', 'countries', 'episodes', 'seasons', 'states']

```

## 参考文献

- [1] 刘甲, 孙德山. 基于注意力机制和LSTM网络的股价预测[J]. 应用数学进展, 2021, 10(12): 4379-4385.
- [2] 王昊, 李思舒, 邓三鸿. 基于N-Gram的文本语种识别研究[J]. 现代图书情报技术, 2013, (4): 54-61. Wang Hao, Li Sishu, Deng Sanhong. Study on Text Language Recognition Based on N-Gram. New Technology of Library and Information Service, 2013, (4): 54-61.
- [3] 段宇锋, 鞠菲. 基于N-Gram的专业领域中文新词识别研究[J]. 现代图书情报技术, 2012, 28(2): 41-47. Duan Yufeng, Ju Fei. Research on Chinese New Word Recognition in Specialized Field Based on N-Gram. New Technology of Library and Information Service, 2012, 28(2): 41-47.