基于TextCNN实现客服问题分类

1. 问题定义

1.1 项目背景

本人所在企业拥有庞大的客服团队，每天需要处理5000+客户的各种问题。如何提高客服团队的工作效率、提升客服的工作品质，为客户提供快速、及时、专业、到位的服务，成为公司提升客户服务体验，增进业务数据，树立市场口碑，建立行业壁垒的重要课题。

客服团队的传统管理哲学主要靠3大核心体系：**培训**、**激励**、**质检**。在互联网时代，尤其是人工智能时代，传统的管理思想已经明显落后且生产力低下，主要体现在：

- 培训需要投入大量培训资源，客服人员成熟时间周期长；
- 激励往往只画饼、定目标，很少提供达成目标的工具；
- 质检同样需要投入大量人力，且纠偏不及时，属于亡羊补牢。

现代客服团队管理强调的是--**标准化**。

通过将客服问题标准化，降低培训的难度和客服人员成熟的周期；标准化后，可以为客服人员提供更加便捷高效的客服工具，提升工作效能；有了标准化的客服话术，质检也可以实现自动化，更加实时有效，且节省人力。

因此，一套基于业务场景的客服问题知识图谱是提升企业客服品效的核心。公司希望利用人工智能为客服团队打造一套智能辅助系统，在理解客户问题的基础上，为客服人员推荐回应话术。

1.2 问题描述

实现智能客服的前提是首先理解客户的问题，然后根据客户所问的问题在知识库或知识图谱中匹配回应内容。所以如何准确理解客户问题是实现智能客服的核心。

公司的客服知识库是基于客户的问题类型进行分类组织的，经过2年的沉淀，已经形成了【知识类目和条数】。之前对客户问题的分类采用的是粗暴的关键字发现，从客户的问题中提取关键字，然后去匹配问题类型。（流程见图1）

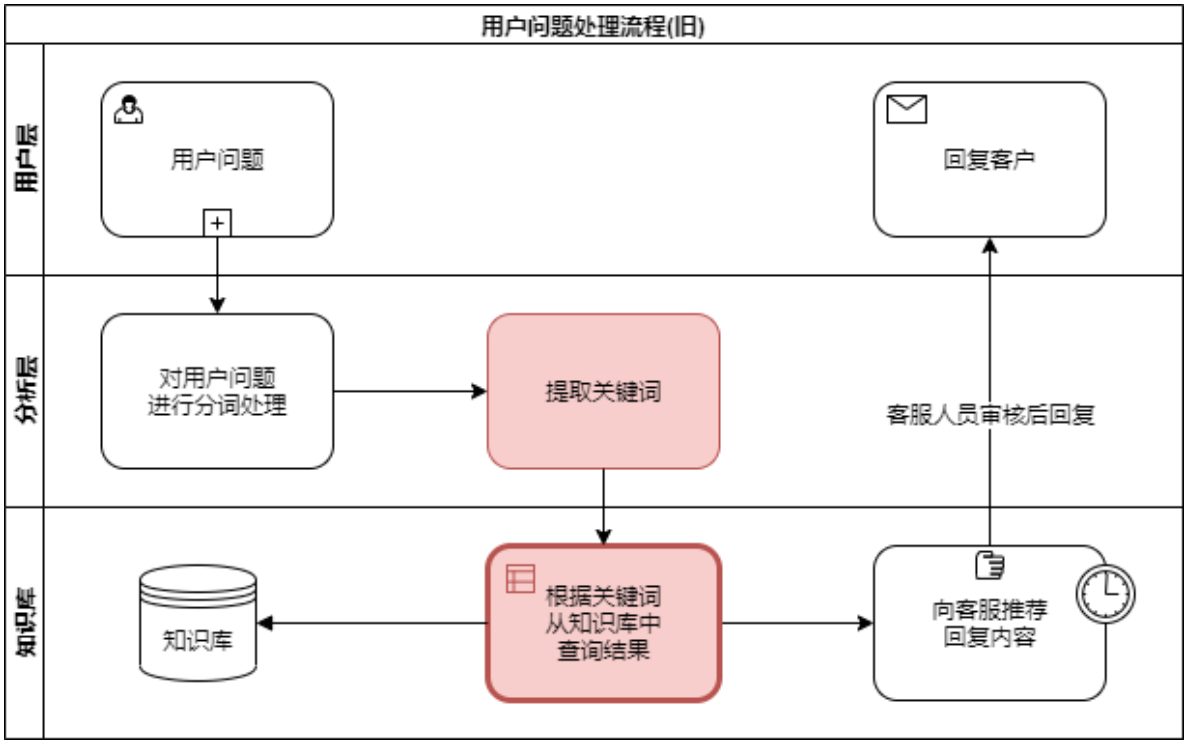

图1：基于关键词匹配的问题响应流程

上面流程最大的问题在红框标出的部分。

首先，关键词往往不能反映客户问题的本意。汉语博大精深，相同词汇组成的句子如果结构或标点符号不同，意思可能大相径庭。比如：

> 健康告知没有了，限制就没意义了。
>
> 健康告知没有了限制，就没意义了。

这两个句子里的文字完全一样，但标点符号不同，表达的意思截然相反。这种情况，通过关键字无法准确识别客户的真实意图，造成回复驴唇不对马嘴。

其次，客户的同一问题表达方式会多种多样，提取的关键词也会非常多变，系统不可能穷尽所有关键词情况。比如客户问：

> 我是美国人，我可以投保吗？
>
> 我是汤加人，我可以投保吗？

其实这两个问题是同一类问题，但是提取的关键字第一个是`美国`，第二个是`汤加`。如果系统没有对`汤加`做处理，则对第二个问题就没办法很好的回答。

最后，关键词匹配丢失了上下文，对于客户的连续问题无法很好的处理。比如：

> 我36岁，可以投保这个产品吗？
>
> 那我爸爸呢？

对第二个问题，只能提取到`爸爸`一个关键词，系统完全不知如何回应。

尽管可以通过很多算法和技巧优化上面的问题，但是整体在实际应用中，匹配准确率只有**47%**，无法满足商业应用的要求。理想的解决方案就是利用人工智能，让机器真的能够理解客户问题的语义，然后根据对问题语义进行分类，从而提高匹配的准确率。理想中的流程如下图：

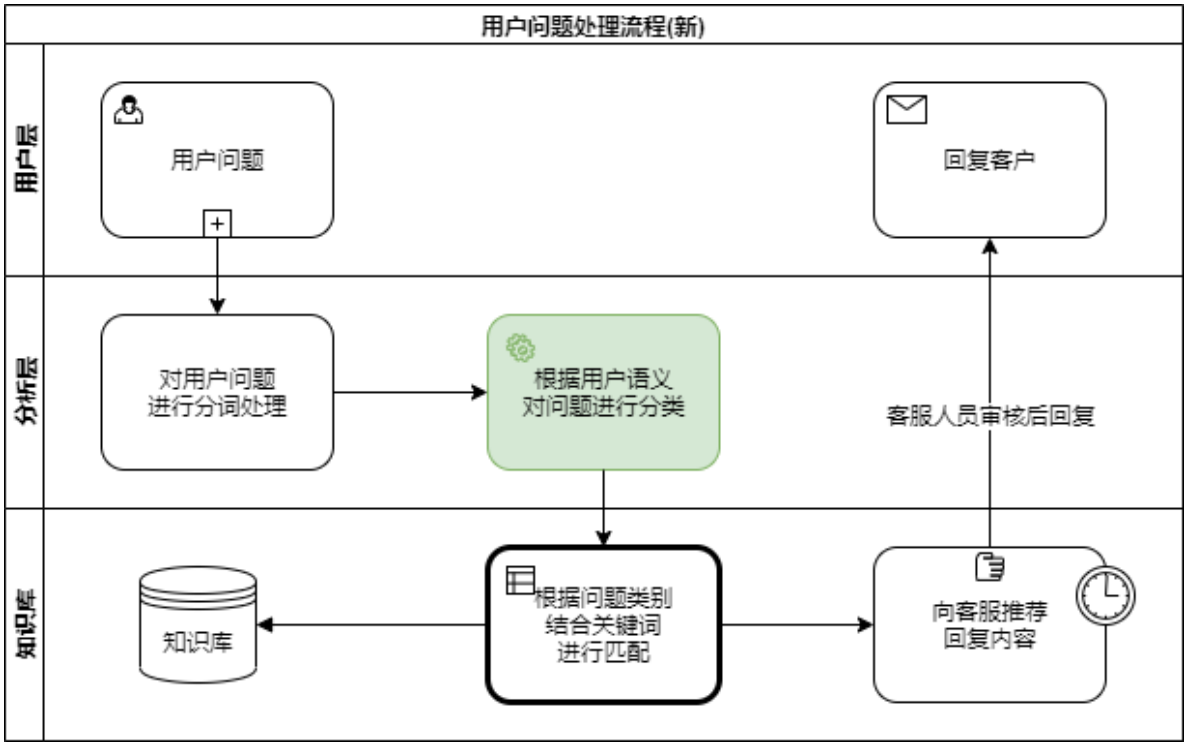

图2：基于语义理解的问题响应流程

新旧流程的区别就在绿框部分，需要训练一个模型能够理解用户输入文本的语义，并将其分类到一个问题分类上。本文下面的内容将围绕这个问题，利用公司已经标记好的客服问题数据集，采用深度学习的方法，训练一个客服问题分类模型。

1.3 评估指标

本项目的目的是将用户提问的问题准确的匹配到一个问题分类上，所以模型的评估指标是**分类的准确率**，即：分类正确的样本数/样本总数

$$Acc = \frac{N_{correct}}{N_{total}}$$

旧有的基于关键词匹配的准确率只有**47%**，为了满足商业应用需要，期望技术深度学习训练出的模型在测试集和验证集上的准确率都能在**90%**以上。

由于训练结果要用于商业使用上，所以还需要考虑性能和部署，希望训练好的模型对客户问题的识别耗时<1s；为了满足规模使用，应对客户问题峰值，整个解决方案需要支持迁移学习和分布式部署。

2. 问题分析

2.1 数据集获取

本文以公司积累的客服问题为研究对象。目前公司数据库中积累的客服问题有3万多条，剔除一些类似“你好”，“在吗？”等字数不超过5个字的问题，还剩下18000多条，从中取16890条作为数据集。

由于这些数据存储在公司服务器，为了便于数据获取和分析，经公司同意，将这批数据脱敏后下载到我个人服务器，保存在MongoDB数据库中。可以通入如下代码获取数据：

```
import pymongo

mongo_client = pymongo.MongoClient("mongodb://112.74.93.48:27017/")
db = mongo_client["InsuranceQA"]
col = db["question"]
```

> 注意：公司授权的使用范围仅限学术研究，不得用于商业目的。学期结束后本人将关闭数据库

2.2 数据探索

2.2.1 整体描述

将数据从MongoDB读入Pandas，对数据进行简单的探索。

输出结果：

```
RangeIndex: 16890 entries, 0 to 16889
Data columns (total 4 columns):
no          16890 non-null float64
question    16890 non-null object
type        16890 non-null object
dtypes: float64(1), object(3)
memory usage: 527.9+ KB
```

从输出可以看出数据集中共有**16888**条数据，每条数据有3个字段，字段描述如下：

字段名	类型	说明
no	int	问题的编号，从0开始，顺序编号
question	string	客服问题
type	string	问题类型

2.2.2 对`type`的统计描述

我们要解决的问题是希望对问题进行分类，即建立从`question`到`type`的映射，所以需要关注有多少种分类。

输出结果：

type	question	rate(%)
伤残险伤残险	949	5.618709
健康险	2002	11.853869
其他	36	0.213156
养老险	525	3.108532
医疗险	2301	13.624252
家财险	1751	10.367695
寿险	4371	25.880751
年金险	800	4.736811
租赁险	1086	6.430221
车险	2165	12.818995
退休计划	172	1.018414
重疾险	135	0.799337
长期护理险	596	3.528924

从输出结果中看出，所有问题分成了13类，各分类占比如下：

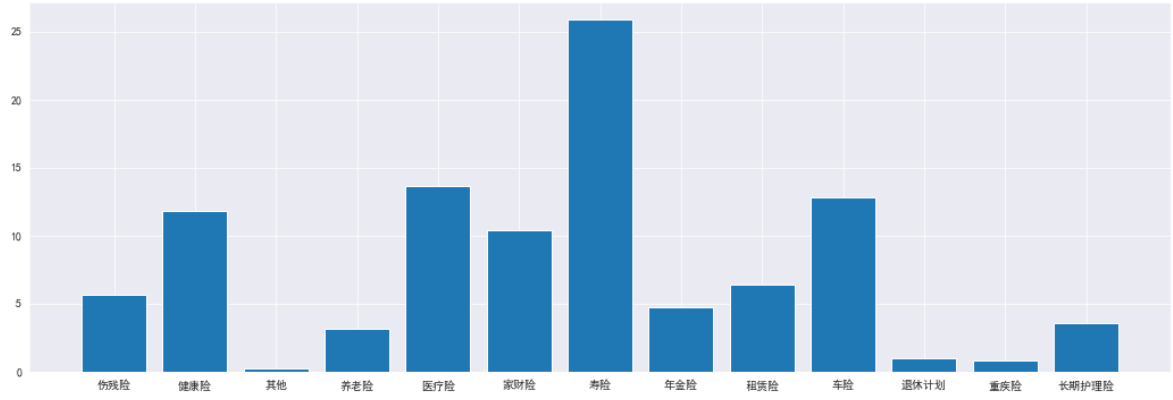

图3：各类型问题占比

从分类占比上可以看出，我们日常常见的寿险、健康险、医疗险和车险占比最大。

2.2.3 对`question`的统计描述

接下来看一下客户问题的属性。我们比较关注问题的长度，所以对问题字数做统计。

输出如下：

```
count    16889.000000
mean        13.041506
std          3.730499
min          2.000000
25%         11.000000
50%         13.000000
75%         15.000000
max         78.000000
Name: q_len, dtype: float64
```

从输出可以看出问题字数都不长，平均13个字（含标点）。最长问题78个字，最短的2个字。分布如下图：

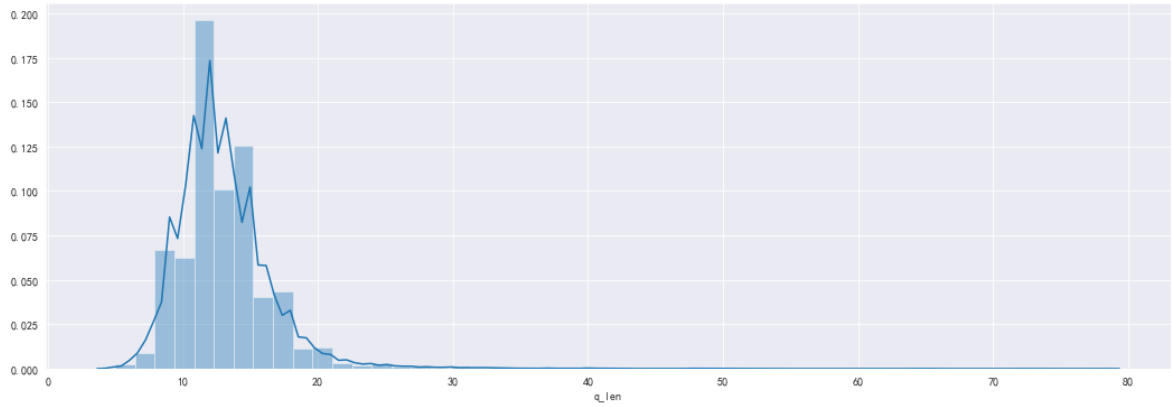

图4：问题字数分布

从分布上可以看出问题字数主要集中在8-18各字，其中以12个字的问题最多。文字字数少正向的一面是训练所需的时间会比较少，不利的一面是由于提供的信息有限，在分类任务中可能无法很好的泛化出有效的模型。所以算法选择就更为重要。

2.3 算法选型

文本分类有很多的实现算法，从大类划分有2大类：

- 基于传统机器学习的文本分类，如SVM、朴素贝叶斯等
- 基于深度学习的文本分类，如TextCNN, LSTM...

2.3.1 传统机器学习方法

传统机器学习方法是90年代后期，伴随着统计学习方法的发展和互联网的兴起，发展起来的经典机器学习方法。传统机器学习方法解决问题的思路是**人工特征工程+浅层的分类模型**。就文本分类问题而言，整个过程分解为特征提取和分类器构建两部分（见图3）。

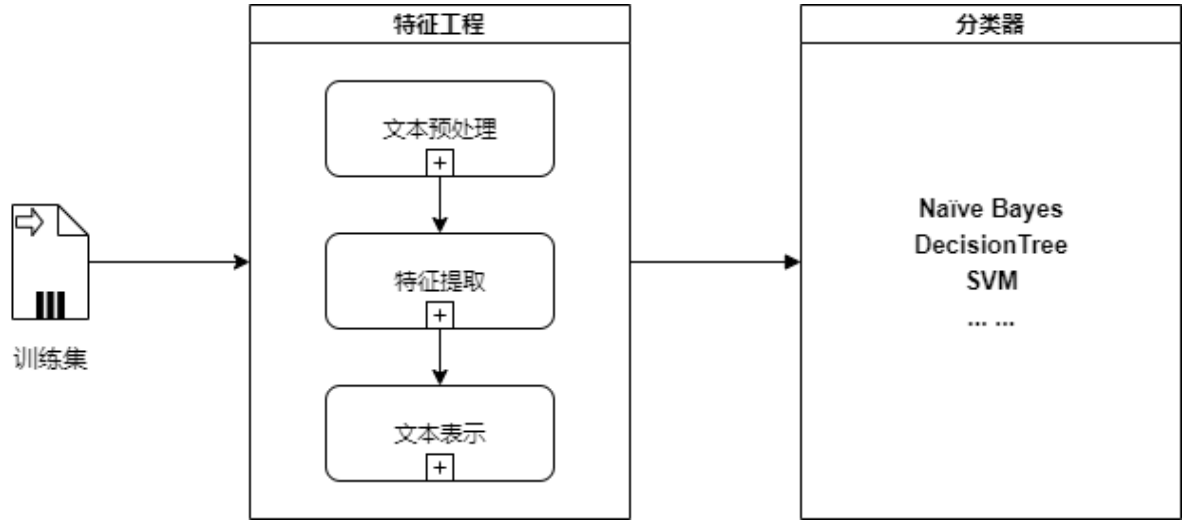

图5：传统机器学习方法解决文本分类流程

2.3.2 深度学习方法

传统机器学习方法主要问题的文本表示是高维度高稀疏的，特征表达能力很弱，而且神经网络很不擅长对此类数据的处理；此外人工特征工程需要人工介入，成本很高。近几年随着深度学习的发展，深度学习算法在文本表示和特征表达方面取得了巨大突破。2014年，Yoon Kim针对CNN的输入层做了一些变形，提出了文本分类模型TextCNN，利用CNN提取文本的特征，去掉繁杂的人工特征工程，最后根据特征进行分类，取得了良好的效果。

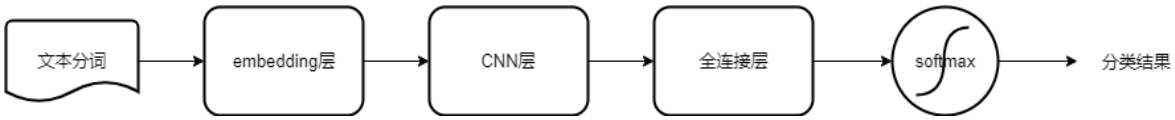

图6：深度学习方法解决文本分类流程

2.3.3 算法选择

通过上面的介绍，我们可以发现深度学习方法比传统机器学习方法具有明显优势。

首先，深度学习方法不需要人工手动的提取文本的特征，它可以自动的获取基础特征并组合为高级的特征，训练模型获得文本特征与目标分类之间的关系，省去了使用TF-IDF等提取句子的关键词构建特征工程的过程。

其次，由于我们的问题字数比较少，如果采用传统方法单纯通过词特征来分类可能无法很好的完成分类任务；相比传统模型而言，深度学习可以更好的利用词序的特征。

最后，考虑训练效率和最终上线部署实施，深度学习方法比传统方法具有优势。利用Tensorflow框架可以利用GPU加速训练过程，并且训练出的模型更容易分布式部署。

而TextCNN相对于其他网络有如下优势：

1. **网络结构简单**，却依旧有很不错的效果；
2. **参数数目少，计算量少，训练速度快**

综上，在模型算法上采用深度学习方法，利用TextCNN进行文本分类；实现上使用Tensorflow开发和部署。

2.4 基准指标

Yoon Kim的论文[2] Convolutional Neural Networks for Sentence Classification 给出的测试结果如下：

Model	MR	SST-1	SST-2	Subj	TREC	CR	MPQA
CNN-rand	76.1	45.0	82.7	89.6	91.2	79.8	83.4
CNN-static	81	45.5	86.8	93	92.8	84.7	**89.6**
CNN-non-static	**81.5**	48.0	87.2	93.4	93.6	84.3	89.5
CNN-multichannel	81.1	47.4	**88.1**	93.2	92.2	**85.0**	89.4

上表中MR，SST-2, SST-3, Subj, TREC, CR, MPQA是数据集名称，作者用了4各模型变种进行测试。

CNN-rand：基线模型，所有词随机初始化并在训练过程中修改；

CNN-static：采用预先训练好的词向量，所有词包括未知词随机初始化；

CNN-non-static：同CNN-static，只是预处理的词向量针对每个任务做了优化；

CNN-multichannel：采用2个预先训练好的词向量，每个词向量视为一个通道。

作者给出的结论是基线模型（CNN-rand）表现的不是很好，但是即便是简单的CNN-static模型，表现已经可圈可点，可以战胜大多数之前的成熟的深度学习模型，并且在**MPQA**数据集上取得了最高分--**89.6%**

Data	分类数	平均句子长度	数据量	词汇量	预训练词向量词汇数	测试集
MPQA	2	3	10606	6246	6083	10-fold CV

MPQA: Opinion polarity detection subtask of the MPQA dataset (Wiebe et al., 2005).

我接下来实现的模型会自己训练词向量，跟CNN-static类似，所以对标CNN-static比较合理。由于作者采用了多个数据集进行测试，MPQA跟我要解决的问题比较接近--句子数少，数据量相当，词汇量也差不多。刚好CNN-static又在MPQA数据集上表现最好，所以参照CNN-static在MPQA上的表现。

考虑到论文中的数据都是文章，句子多且长，而我的任务都是比较短的客户提问，基本都是一句话，平均字数也较少（平均13个字），所以预期我的模型分类准确率要高于**89.6%**，至少要达到**90%**以上，能够达到**99%**在工业应用上最好。

3. 模型实现

本模型的实现参考了**Yoon Kim**的论文Convolutional Neural Networks for Sentence Classification，论文中Yoon Kim的TextCNN结构可以描述为下图的网络：

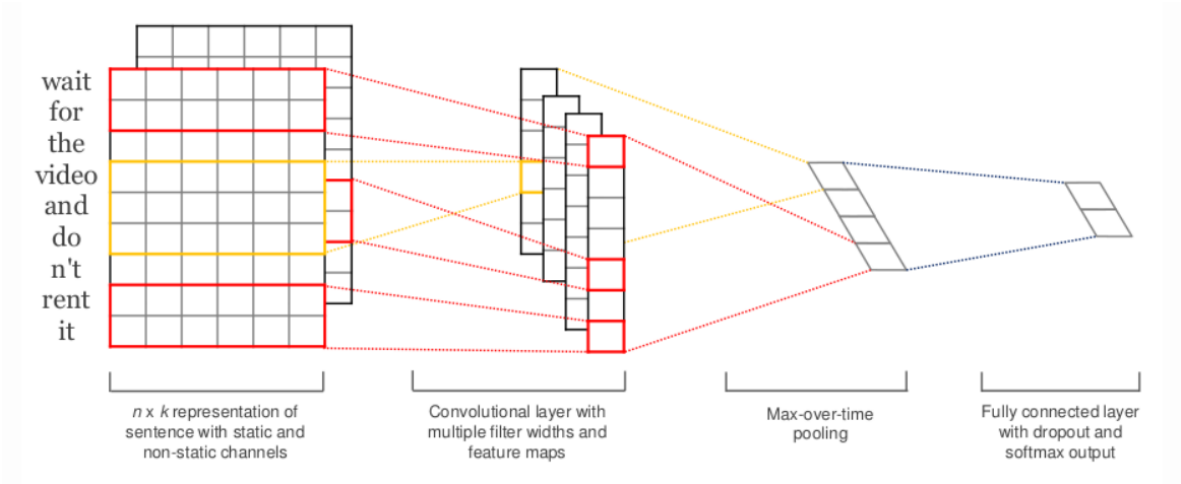

图7：TextCNN网络结构

TextCNN的实现过程可以分为图下4步：

- **Embedding**：第一层是图中最左边的句子矩阵，每行是词向量，这个可以类比为图像中的原始像素点。
- **Convolution**：然后经过一维卷积层，每个kernel_size 有两个输出 channel。
- **MaxPolling**：第三层是一个1-max pooling层，这样不同长度句子经过pooling层之后都能变成定长的表示。
- **FullConnection and Softmax**：最后接一层全连接的 softmax 层，输出每个类别的概率。

3.1 数据预处理

原始的文本数无法直接在CNN中使用，所以在训练模型前，首先需要对数据进行预处理并将其转化为CNN能用的格式。整个数据预处理分为如下几步：

图8：数据预处理步骤

3.1.1 分词并统计词频

分词采用结巴分词对中文进行分词。分词前先做去标点符号处理，然后再分词，接着去掉停用词，统计词频。分词结果保存到dataframe的`fenci`列，词频的统计保存到`word_dict`字典中，用于下一步构建字典。

数据结构

	no	question	type	q_len	fenci
0	0.0	法律要求残疾保险吗?	伤残险	10	法律 残疾 保险
1	1.0	债权人可以在死后人寿保险吗?	寿险	14	债权人 死 人寿保险
2	2.0	旅行者保险有租赁保险吗?	租赁险	12	旅行者 保险 租赁 保险
3	3.0	我可以开一辆没有保险的新车吗?	车险	15	开 一辆 保险 新车
4	4.0	人寿保险的现金转出价值是否应纳税?	寿险	17	人寿保险 现金 转 价值 应 纳税

3.1.2 生成词汇表

根据`word_dict`中的词汇，生成word-to-id的映射表。

考虑到数据输入到CNN时需要对齐，所以需要用0补齐，字典第一个词固定写死为`<PAD>`。

这里构建词汇表时采用了一个技巧，直接用dataframe的index作为id，所以构建词汇表很简单，只需要从python字典构建dataframe即可。

生成的数据格式如下：

	word	freq
		0
1	法律	27
2	残疾	912
3	保险	6561
4	债权人	43

其中用data frame的index作为id，word是词，freq是词频。

用同样的方法生成分类的映射表，

生成的数据格式如下：

	type	freq
0	伤残险	949
1	健康险	2002
2	其他	36
3	养老险	525
4	医疗险	2301

同样用data frame的index作为id，type是类型，freq是频率。

3.1.3 数据集划分

采用**8:1:1**的比例将数据集划分为训练集、测试集和验证集。

按照上面代码划分后：

- 训练集　13512
- 训练集　1689
- 验证集　1689

3.1.4 数据向量化

目前数据集中的数据还是分词后的文本，我们需要根据第2步构建的字典将问题和分类进行向量化处理。处理过程很简单，就是查字典的过程，需要注意的是由于问题长度不通过，分词出来的个数不同，为了CNN能够使用，需要做截断或补齐。根据数据探索部分对问题长度的统计，问题都不长，平均13个字（含标点），这里将分词个数设置为10个。最后将数据转化为numpy格式，这样数据准备工作就完成了。

处理后的数据结构如下

train_input	**[13512, 10]**	**train_output**	**[13512, 13]**
test_input	**[1689, 10]**	**test_output**	**[1689, 13]**
valid_input	**[1689, 10]**	**valid_output**	**[1689, 13]**

3.2 TextCNN模型实现

实际实现过程中采用了如下的结构:

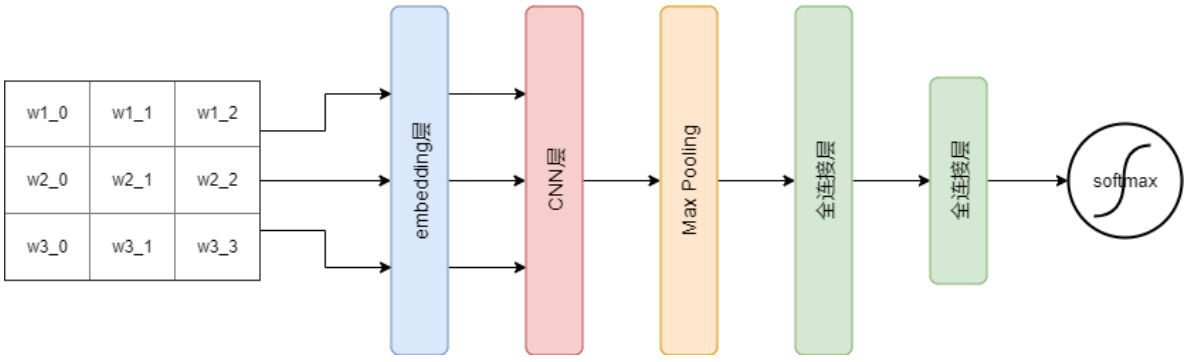

图9：实际实现的TextCNN网络结构

具体实现代码请参见`Insurance Question Classification with TextCNN.ipynb`

```
# 超参数
seq_length = 10                          # 序列长度
num_classes = type_df['type'].count()    # 类别数
vocab_size = word_df['word'].count()     # 词汇表大小
embedding_dim = 64                       # 词向量维度
num_filters = 256                        # 卷积核数目
kernel_size = 5                          # 卷积核尺寸
hidden_dim = 128                         # 全连接层神经元
```

训练过程采用设置了10轮迭代，训练过程输出如下：

```
Epoch: 1
Iter: 0, Train Loss:    2.6, Train Acc:   4.69%, Val Loss:    2.6, Val Acc:
7.46%
Iter: 100, Train Loss: 0.32, Train Acc:  95.31%, Val Loss:   0.33, Val Acc:
93.66%
Iter: 200, Train Loss: 0.34, Train Acc:  93.75%, Val Loss:   0.14, Val Acc:
96.39%
Epoch: 2
Iter: 300, Train Loss:0.055, Train Acc:  96.88%, Val Loss:   0.11, Val Acc:
97.10%
Iter: 400, Train Loss:0.087, Train Acc:  96.88%, Val Loss:   0.11, Val Acc:
97.04%
Epoch: 3
Iter: 500, Train Loss: 0.17, Train Acc:  96.88%, Val Loss:    0.1, Val Acc:
97.04%
Iter: 600, Train Loss:0.032, Train Acc:  98.44%, Val Loss:    0.1, Val Acc:
97.16%
Epoch: 4
Iter: 700, Train Loss:0.095, Train Acc:  96.88%, Val Loss:    0.1, Val Acc:
97.10%
Iter: 800, Train Loss:0.053, Train Acc:  96.88%, Val Loss:    0.1, Val Acc:
97.10%
Epoch: 5
Iter: 900, Train Loss: 0.16, Train Acc:  98.44%, Val Loss:   0.11, Val Acc:
97.16%
Iter: 1000, Train Loss:0.062, Train Acc:  98.44%, Val Loss:  0.11, Val Acc:
96.74%
Epoch: 6
Iter: 1100, Train Loss:0.023, Train Acc:  98.44%, Val Loss:  0.11, Val Acc:
96.86%
Iter: 1200, Train Loss:0.018, Train Acc:  98.44%, Val Loss:  0.11, Val Acc:
96.74%
Epoch: 7
Iter: 1300, Train Loss:0.039, Train Acc:  98.44%, Val Loss:  0.11, Val Acc:
96.80%
Iter: 1400, Train Loss:0.012, Train Acc: 100.00%, Val Loss:  0.12, Val Acc:
96.63%
Epoch: 8
Iter: 1500, Train Loss: 0.0088, Train Acc: 100.00%, Val Loss:0.11, Val Acc:
96.68%
Iter: 1600, Train Loss:  0.019, Train Acc:  98.44%, Val Loss:0.12, Val Acc:
96.74%
Epoch: 9
Iter: 1700, Train Loss: 0.0042, Train Acc: 100.00%, Val Loss:0.13, Val Acc:
96.68%
```

```
Iter: 1800, Train Loss:  0.013, Train Acc: 100.00%, Val Loss:0.14, Val Acc:
96.51%
Iter: 1900, Train Loss:  0.064, Train Acc:  95.31%, Val Loss:0.13, Val Acc:
96.68%
Epoch: 10
Iter: 2000, Train Loss:  0.013, Train Acc: 100.00%, Val Loss:0.13, Val Acc:
96.63%
Iter: 2100, Train Loss: 0.0031, Train Acc: 100.00%, Val Loss:0.13, Val Acc:
96.63%
```

从训练结果可以看出，100次迭代之后，验证集上的准确率就能达到90%，整个训练结束后，最好结果在验证集上准确率可以达到97.16%，效果还不错。

由于我们的数据集问题长度比较短，分次数比较少，所以尝试降低核数和过滤器再尝试训练一次。

```
embedding_dim = 32       # 词向量减少一半
num_filters = 128        # 卷积核数目减少一半
kernel_size = 3          # 卷积核尺寸
hidden_dim = 128         # 全连接层神经元不变
```

输出结果如下：

```
Epoch: 1
Iter:  0, Train Loss:    2.6, Train Acc:   6.25%, Val Loss:    2.6, Val
Acc:  10.01%
Iter:  100, Train Loss:   0.83, Train Acc:  70.31%, Val Loss: 0.71, Val
Acc:  76.49%
Iter:  200, Train Loss:   0.22, Train Acc:  90.62%, Val Loss:  0.2, Val
Acc:  95.20%
Epoch: 2
Iter:  300, Train Loss:   0.12, Train Acc:  95.31%, Val Loss: 0.15, Val
Acc:  96.09%
Iter:  400, Train Loss:  0.081, Train Acc:  96.88%, Val Loss: 0.12, Val
Acc:  97.04%
Epoch: 3
Iter:  500, Train Loss:  0.067, Train Acc:  98.44%, Val Loss: 0.11, Val
Acc:  97.10%
Iter:  600, Train Loss:   0.08, Train Acc:  96.88%, Val Loss: 0.11, Val
Acc:  97.04%
Epoch: 4
Iter:  700, Train Loss:  0.041, Train Acc:  98.44%, Val Loss: 0.11, Val
Acc:  96.92%
Iter:  800, Train Loss:  0.097, Train Acc:  95.31%, Val Loss:  0.1, Val
Acc:  97.22%
Epoch: 5
Iter:  900, Train Loss:   0.01, Train Acc: 100.00%, Val Loss:  0.1, Val
Acc:  97.16%
Iter: 1000, Train Loss:  0.079, Train Acc:  96.88%, Val Loss:  0.1, Val
Acc:  97.10%
Epoch: 6
Iter: 1100, Train Loss:  0.019, Train Acc: 100.00%, Val Loss:  0.1, Val
Acc:  96.98%
Iter: 1200, Train Loss:  0.014, Train Acc: 100.00%, Val Loss: 0.11, Val
Acc:  96.98%
Epoch: 7
```

```
Iter: 1300, Train Loss: 0.00028, Train Acc: 100.00%, Val Loss:  0.1, Val
Acc: 96.98%
Iter: 1400, Train Loss:  0.048, Train Acc:  98.44%, Val Loss: 0.11, Val
Acc:  96.80%
Epoch: 8
Iter: 1500, Train Loss:  0.012, Train Acc: 100.00%, Val Loss: 0.11, Val
Acc:  96.98%
Iter: 1600, Train Loss: 0.0079, Train Acc: 100.00%, Val Loss: 0.12, Val
Acc:  96.80%
Epoch: 9
Iter: 1700, Train Loss: 0.0032, Train Acc: 100.00%, Val Loss: 0.11, Val
Acc:  96.92%
Iter: 1800, Train Loss:  0.035, Train Acc:  98.44%, Val Loss: 0.11, Val
Acc:  96.98%
Iter: 1900, Train Loss: 0.0026, Train Acc: 100.00%, Val Loss: 0.11, Val
Acc:  96.98%
Epoch: 10
Iter: 2000, Train Loss:  0.027, Train Acc:  98.44%, Val Loss: 0.11, Val
Acc:  96.86%
Iter: 2100, Train Loss:  0.019, Train Acc: 100.00%, Val Loss: 0.12, Val
Acc:  96.74%
```

从验证集的效果看，没有明显的提升，并且比上一版稍差。所以换个方向，提高过滤器和核数，看一下效果。

```
embedding_dim = 64       # 词向量维度
num_filters = 512        # 卷积核数目
kernel_size = 7          # 卷积核尺寸
hidden_dim = 512         # 全连接层神经元
```

输出结果如下：

```
Epoch: 1
Iter:   0, Train Loss:    2.6, Train Acc:  26.56%, Val Loss:  2.60, Val
Acc:  18.65%
Iter:  100, Train Loss:   0.17, Train Acc:  96.88%, Val Loss: 0.26, Val
Acc:  94.02%
Iter:  200, Train Loss:   0.17, Train Acc:  92.19%, Val Loss: 0.14, Val
Acc:  96.45%
Epoch: 2
Iter:  300, Train Loss:  0.059, Train Acc:  98.44%, Val Loss: 0.12, Val
Acc:  96.68%
Iter:  400, Train Loss:  0.059, Train Acc:  98.44%, Val Loss: 0.11, Val
Acc:  96.86%
Epoch: 3
Iter:  500, Train Loss:  0.085, Train Acc:  96.88%, Val Loss: 0.11, Val
Acc:  96.86%
Iter:  600, Train Loss: 0.0094, Train Acc: 100.00%, Val Loss: 0.11, Val
Acc:  96.98%
Epoch: 4
Iter:  700, Train Loss:  0.019, Train Acc: 100.00%, Val Loss: 0.11, Val
Acc:  96.68%
Iter:  800, Train Loss:  0.019, Train Acc:  98.44%, Val Loss: 0.12, Val
Acc:  96.80%
Epoch: 5
Iter:  900, Train Loss:  0.011, Train Acc: 100.00%, Val Loss: 0.13, Val
Acc:  96.39%
```

```
Iter: 1000, Train Loss:  0.012, Train Acc: 100.00%, Val Loss: 0.13, Val
Acc:  96.80%
Epoch: 6
Iter: 1100, Train Loss: 0.00098, Train Acc: 100.00%, Val Loss: 0.14, Val
Acc: 96.92%
Iter: 1200, Train Loss: 0.0072, Train Acc: 100.00%, Val Loss: 0.14, Val
Acc:  96.45%
Epoch: 7
Iter: 1300, Train Loss:  0.041, Train Acc:  98.44%, Val Loss: 0.15, Val
Acc:  96.51%
Iter: 1400, Train Loss:  0.043, Train Acc:  98.44%, Val Loss: 0.16, Val
Acc:  96.74%
Epoch: 8
Iter: 1500, Train Loss:  0.079, Train Acc:  96.88%, Val Loss: 0.15, Val
Acc:  96.80%
Iter: 1600, Train Loss:  0.015, Train Acc: 100.00%, Val Loss: 0.16, Val
Acc:  96.39%
Epoch: 9
Iter: 1700, Train Loss:  0.052, Train Acc:  96.88%, Val Loss: 0.24, Val
Acc:  94.43%
Iter: 1800, Train Loss: 0.0068, Train Acc: 100.00%, Val Loss: 0.15, Val
Acc:  96.57%
Iter: 1900, Train Loss: 0.0051, Train Acc: 100.00%, Val Loss: 0.17, Val
Acc:  96.57%
Epoch: 10
Iter: 2000, Train Loss: 0.0011, Train Acc: 100.00%, Val Loss: 0.16, Val
Acc:  95.97%
Iter: 2100, Train Loss:   0.01, Train Acc: 100.00%, Val Loss: 0.16, Val
Acc:  95.86%
```

结果上来看，效果更差了。效果最好的还是最初版本的参数效果最好。

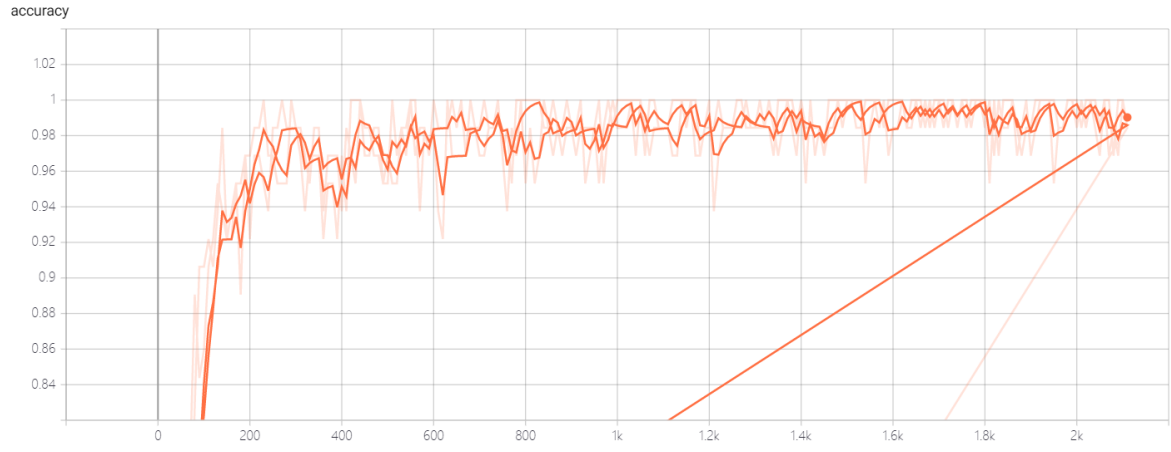

图10：初版参数准确率变化

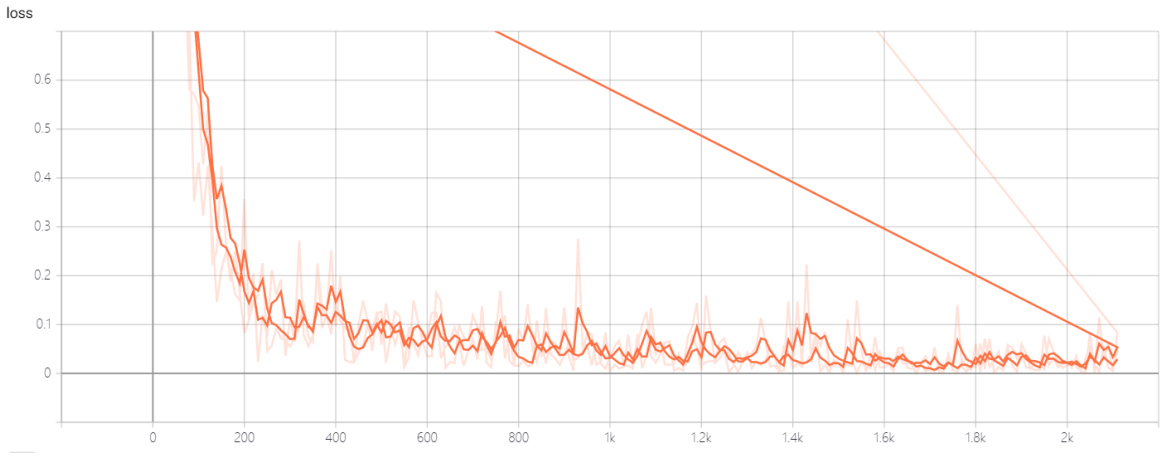

图11：初版参数loss变化

4. 模型效果

训练得到的最优模型保存在**./checkpoint/best**目录下，利用该模型在测试集上测试模型效果。

输出结果如下：

```
Test Loss:   0.11, Test Acc:  96.45%
              precision    recall  f1-score   support

          伤残险       0.98      0.99      0.98        89
          健康险       0.98      0.97      0.98       213
           其他       0.33      0.20      0.25         5
          养老险       0.68      0.97      0.80        62
          医疗险       0.99      0.97      0.98       208
          家财险       0.99      1.00      0.99       206
           寿险       0.99      0.99      0.99       406
          年金险       0.94      0.97      0.95        75
          租赁险       0.96      0.96      0.96       105
           车险       0.98      0.99      0.98       214
         退休计划       1.00      0.03      0.07        29
          重疾险       1.00      1.00      1.00        15
        长期护理险       1.00      1.00      1.00        62

    micro avg       0.96      0.96      0.96      1689
    macro avg       0.91      0.85      0.84      1689
 weighted avg       0.97      0.96      0.96      1689
```

从输出可以看出，模型在测试集上准确率达到**96.45%**，这个结果比基准**89.6%**要高不少。

从各分类的准确率、召回率和F1-score上看，其他和养老险这2各分类准确率很低，需要进一步优化。

输出一下混合矩阵

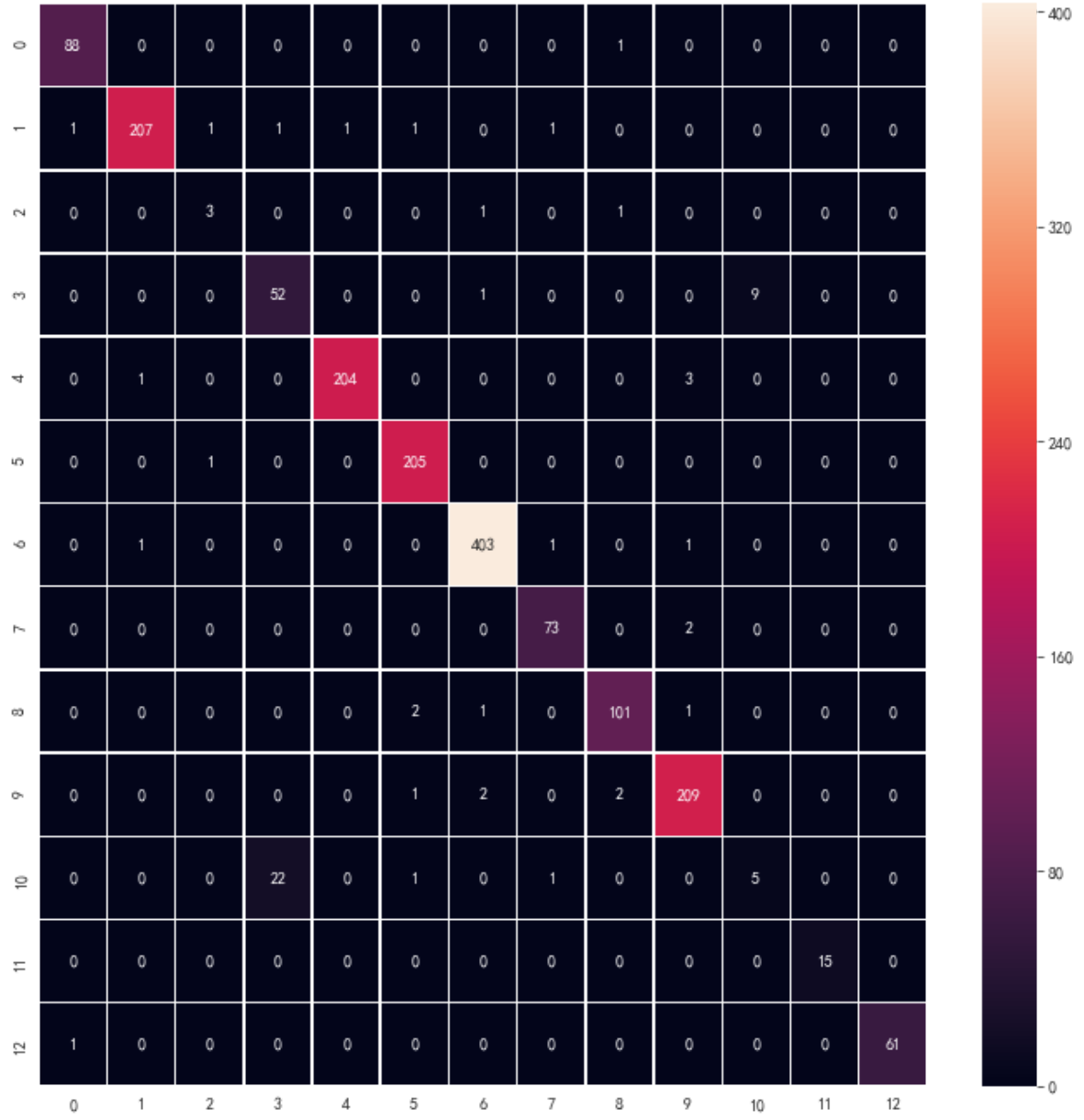

图12：测试集混合矩阵

可以发现：

- 分类2(其他)：测试数据很少，只有5条数据，有1条错误分类成了6(寿险)，1条错误分类成了8(租赁险)
- 分类10(退休计划): 有22条错误分类成了3(养老险)

只有这2项拉低了准确率，如果去除这2类，其余分类的准确率平均可以达到**99%**，这个结果相当不错。

对于其他和养老险，后续做专门针对性的优化。

5. 结论

从验证和测试结果来看分类效果非常理想，能够达到项目的评估指标。

但是具体到各分类，**其他**和**养老险**这两个类别准确很低，其中**其他**的准确率只有33%。稍微研究了一下这两个分类的数，**其他**准确率的原因主要是数据量太少，模型没有很好的提取出问题特征。而养老险和退休计划比较接近，因为退休和养老本身概念相关性很高，所以没有很好的泛化。

后面会针对这两个分类做更深入的研究：

1. 增加其他分类的数据样本；
2. 考虑将`退休计划`和`养老险`进行合并；

3. 采用RNN重写模型，验证RNN能否更好的解决养老险和退休计划无法有效提取特征的问题。

参考文献

[1] Minwei Feng, Bing Xiang, Michael R. Glass, Lidan Wang, Bowen Zhou. APPLYING DEEP LEARNING TO ANSWER SELECTION: A STUDY AND AN OPEN TASK[cs.CL]. 2015

[2] Yoon Kim. Convolutional Neural Networks for Sentence Classification[cs.CL]. 2014

[3] 文本分类算法TextCNN原理详解. https://www.cnblogs.com/ylaoda/p/11339925.html

[4] Tensorflow使用LSTM实现中文文本分类. https://blog.csdn.net/missyougoon/article/details/89414953