人工智能期中项目

幽默文本检测

中山大学计算机学院　计算机科学与技术

施天予　　孙奥远

19335174　　19335177

目录

一、概述

本次期中项目的内容是幽默文本检测，包括幽默文本的分类与回归问题。在文本数据处理方面，我们尝试了 TFIDF、Word2Vec、Doc2Vec 的方法。在分类与回归算法上，我们尝试了朴素贝叶斯、神经网络、随机森林、基于 KNN 的集成学习的方法，并对各种方法进行了分析比较。

二、分工

施天予：

- 文本数据预处理和文本降维
- 随机森林的分类与回归

孙奥远：

- 朴素贝叶斯分类
- 神经网络的分类与回归
- 基于 KNN 的集成学习

实验报告的对应部分由每个人自己完成

三、实验原理

1. TFIDF

TF-IDF 是一种评估一个字词对于一个文件的重要程度，如果一个词在一篇文档中出现的频率高，并且在语料库中其他文档中出现的次数较少，则这个词对于该篇文档很重要，即具有很好的类别区分能力。

TF: 是词频归一化后的概率表示，即一个词语在某篇文档中出现的频率，一个词出现的越频繁，它的重要性就越高。公式如下, 单词 i 在文档 d 的 tf 值，表示为单词 i 在文档 d 中出现的次数除以文档 d 中单词的总数。

$$tf_{i,d} = \frac{n_{i,d}}{\sum_v n_{v,d}}$$

逆文档频率 IDF: 仅仅用 TF 来表示一个词的重要性是不够的，某些词出现的频率很高，但是对于我们判断一个文档的类别没有什么用，比如停用词等，因此我们需要把罕见词的权值提高，把常见词的权值降低，IDF 的计算公式如下，文档的总数 |D| 除以包含单词 i 的文档的数目。

$$idf_i = log\frac{|D|}{|\{j : t_i \in d_j\}|}$$

但是以上数学表达式存在一定的问题，若一个单词 i 没有被任何文档包含，则会出现分母

为 0 的情况，为了防止除以 0 现象的出现，我们将分母 +1，变为下面的形式

$$idf_i = log\frac{|D|}{|\{j : t_i \in d_j\}| + 1}$$

单词 i 的 TF-IDF 的值就是将 TF 和 IDF 进行相乘，某个词对于文章的重要性越高，它的 TF-IDF 值也越大，计算公式如下

$$tfidf_{i,j} = tf_{i,j} \times idf_i$$

2. Word2Vec

Word2Vec，是一个做 Word Embedding 的模型，也就是说将单词从原始的数据空间映射到低维空间转为向量形式的模型。Word2Vec 是用一个一层的神经网络，把每个单词的稀疏词向量映射成为一个 n 维的稠密向量的过程，以达到文本降维的效果，可以用在许多文本分析任务的数据处理上。Word2Vec 里面有两个重要的模型，CBOW 模型 (Continuous Bag-of-Words Model) 与 Skip-Gram 模型，CBOW 模型是给定上下文预测 input 单词，Skip-Gram 模型是给定 input 单词预测上下文。在 Word2Vec 中，还可以使用负采样 (Negative Sampling)，霍夫曼 (Hierarchical) 等方法完成加速。总而言之，对 Word2Vec 模型属于一个文档，它的输出是文档中每个单词的 n 维向量。

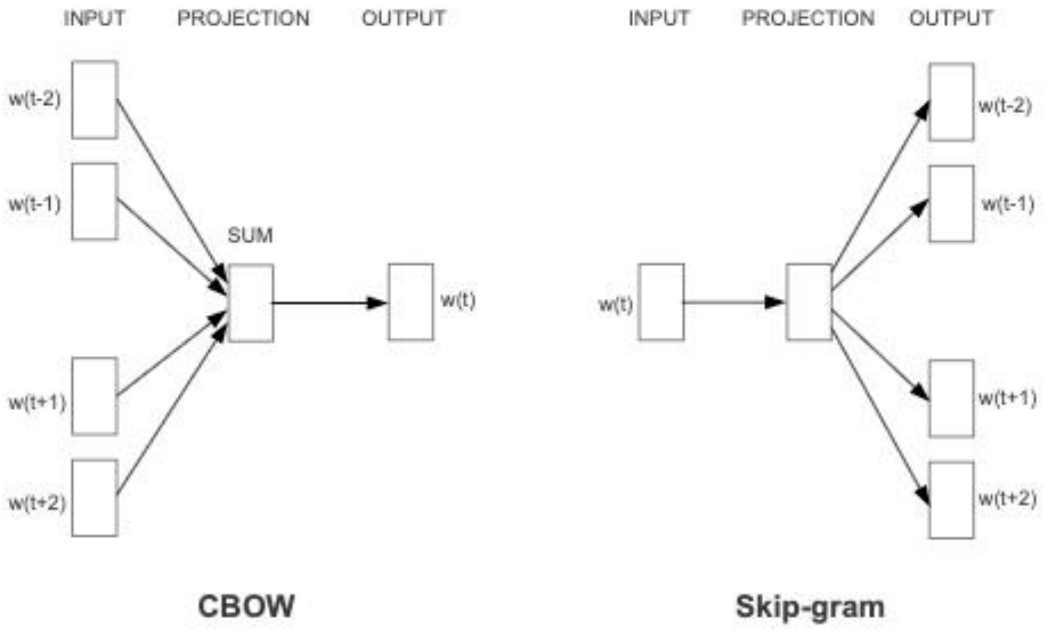


图 1: Word2Vec 模型

3. Doc2Vec

Word2Vec 模型能获得每个单词的词向量,但是如何表示一个句子的词向量呢?即使 Word2Vec 模型对一个句子中的所有单词词向量加起来进行平均处理，我们仍然忽略了单词之间的排列顺序对情感分析的影响。即 word2vec 只是基于词的维度进行“语义分析”的，而并不具有上下文的“语义分析”能力。

Doc2Vec 作为一个处理可变长度文本的总结性方法,除了增加一个段落向量以外,这个方法几乎等同于 Word2Vec。和 Word2Vec 一样，该模型也存在两种方法：Distributed Memory(DM) 和 Distributed Bag of Words(DBOW)。DM 试图在给定上下文和段落向量的情况下预测单词的概率。在一个句子或者文档的训练过程中，段落 ID 保持不变，共享着同一个段落向量。DBOW 则在仅给定段落向量的情况下预测段落中一组随机单词的概率。

- 在训练阶段，新增了 paragraph id，即每个句子的唯一 id。先将 paragraph id 映射成一个向量，即 paragraph vector。在之后的计算里，paragraph vector 和 word vector 累加或者连接起来，作为输出层 softmax 的输入。在一个句子或者文档的训练过程中，paragraph id 保持不变，共享着同一个 paragraph vector，相当于每次在预测单词的概率时，都利用了整个句子的语义。
- 在预测阶段，给待预测的句子新分配一个 paragraph id，词向量和输出层 softmax 的参数保持训练阶段得到的参数不变，重新利用梯度下降训练待预测的句子。待收敛后，即得到待预测句子的 paragraph vector。

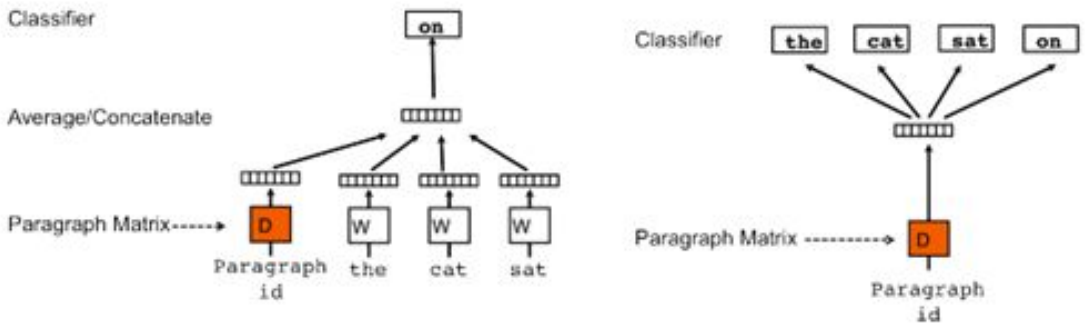


图 2: Doc2Vec 模型

4. 朴素贝叶斯

朴素贝叶斯分类是一种简单的分类算法，朴素贝叶斯的基本思想为：对于给出的待分类项，求解在此项出现的条件下每一个类别的概率，哪个概率大就认为该待分类项属于哪个类别。

设类别集合为 $C = \{y_1, y_2, \ldots, y_n\}$，待分类项为 $X = \{a_1, a_2, \ldots, a_m\}$，我们需要计算

$P(y_1|X), P(y_2|X), P(y_3|X), \ldots, P(y_n|X)$。从中选择值最大的 $P(y_k|X)$, 则预测 X 属于类别 y_k。

根据贝叶斯定理得知 $P(A|B) = \frac{P(AB)}{P(B)}$, 因此可以得到计算上面给定 X 的条件下类别出现的概率

$$P(y_i|X) = \frac{P(X|y_i) * P(y_i)}{P(X)} \quad (1)$$

朴素贝叶斯之所以称为朴素的原因在于,它假设各个特征属性之间是条件独立的,即 $P(X|y_i) = P(a_1|y_i) * P(a_2|y_i) * P(a_3|y_i) \cdots * P(a_n|y_i)$,$P(X) = P(a_1) * P(a_2) * \cdots * P(a_n)$。那么 (1) 式可以写成

$$P(y_i|X) = \frac{P(a_1|y_i) * P(a_2|y_i) * P(a_3|y_i) * \cdots * P(a_n|y_i) * P(y_i)}{P(a_1) * P(a_2) * \cdots * P(a_n)}$$

由于对于每个类别来说分母均是相同的，因此仅考虑分子即可。

所以朴素贝叶斯分类算法要做的就是找到使 $P(a_1|y_i) * P(a_2|y_i) * P(a_3|y_i) * \cdots * P(a_n|y_i) * P(y_i)$ 取最大值的 y_k, 预测 X 属于类别 y_k。

公式中的几种概率的计算方法

$P(y_i)$ 是类别为 y_i 的项的先验概率，一般情况下，在利用朴素贝叶斯分类时，都有一个带类别的数据集—训练集。$P(y_i)$ 可以利用训练集中类别为 y_i 的项出现的频率进行近似。$P(a_j|y_i)$ 是在类别为 y_i 的条件下，第 j 个特征值取值为 a_j 的条件概率，设第 j 个特征有 S_j 个取值，从训练集中找到类别为 y_i 的项 (数目为 n_1)，再从这些项中得到第 j 个特征取值为 a_j 的项的数目 n_2, 则可以利用 $\frac{n_2}{n_1}$ 近似 $P(a_j|y_i)$

防止下溢

在对 $P(a_1|y_i) * P(a_2|y_i) * P(a_3|y_i) * \cdots * P(a_n|y_i) * P(y_i)$ 求值时，由于其中的每一个乘积项均小于 <1，因此越乘越小，会造成下溢。为了解决这个问题，要对该表达式进行取对数，即计算 $\sum_{j=1}^{n} log(P(a_j|y_i)) + log(P(y_i))$。而且由于对数函数是单调递增函数，对一个函数取对数，不会影响其变化趋势，因此取对数不会影响最终的分类结果。

零概率问题

假设现在有一个待分类样本 $x = (a_1, a_2, \ldots, a_n)$，根据上面的朴素贝叶斯的计算公式使 $P(a_1|y_i) * P(a_2|y_i) * P(a_3|y_i) * \cdots * P(a_n|y_i) * P(y_i)$ 取最大值的 y_k, 预测 x 属于类别 y_k。若由于训练集太小，导致某个类别中没有出现某个特征的某个取值的项，那么估计的 $P(a_j|y_i)$ 就为 0，导致预测属于类别 y_j 的概率为 0。为了避免这个问题，可以在计算 $P(a_j|y_i)$ 时，在分子多加一个 1，对应的在分母加上该特征可能取得值得数目。假定在训练集很大的情况下，分子加 1 不会对最终的 $P(a_j|y_i)$ 的估计造成较大的影响，但是避免了 0 概率问题。

5. 神经网络

神经网络其实就是一个输入 X 到输出 Y 的一个映射函数 $f(X) = Y$, 给定一个输入 x_i，经过神经网络的一系列计算得到最终的输出 y_i。一个三层的神经网络的结构如图3所示

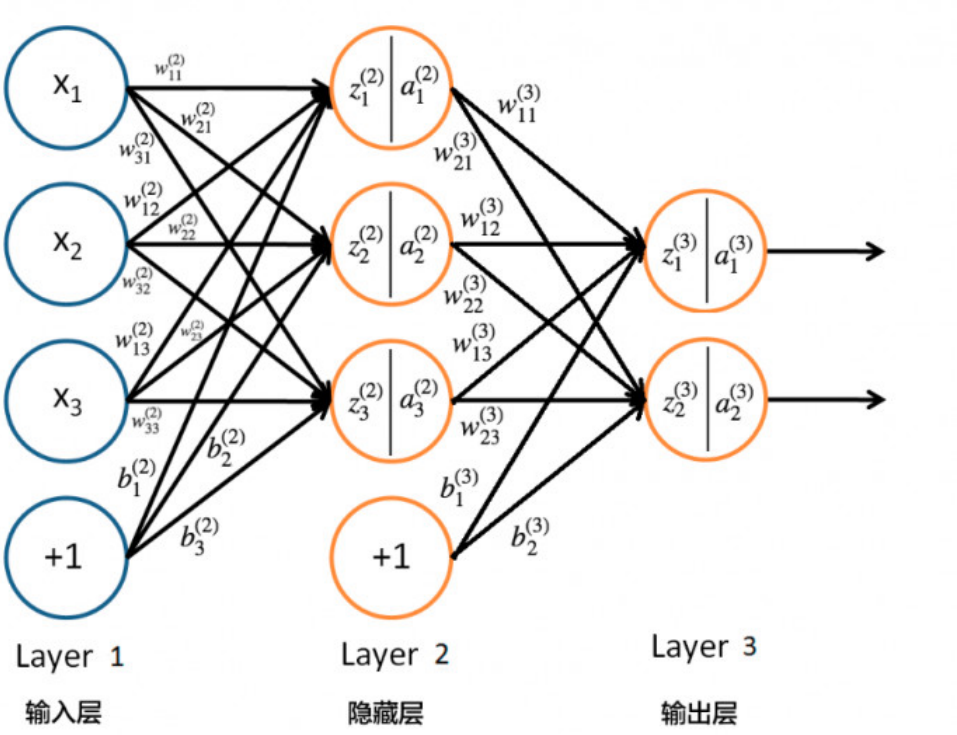


图 3: 三层的神经网络

我们可以看到神经网络是一个分层结构，一个神经网络由输入层，隐藏层，输出层组成。每一层均是由若干节点组成，每一个节点可以看作一个处理单元，其接收上一层节点的输出作为输入，经过激活函数处理之后产生一个输出。可以知道，若一个层有 N 个节点，那么该层的所有节点的输出产生一个 N 维向量。

前向传播

前向传播是指输入数据从输入层开始，经过每一层节点的处理，最终到达输出层的过程。在前向传播的过程中，每一个节点需要做的有两件事，一是对上层节点的输出做加权求和，二是进行激活函数处理。

- 加权求和

  在图3中，变量 z_i^j 代表的是第 j 层网络的第 i 个神经元的输入，变量 a_i^j 代表的是第 j 层神经网络的第 i 个神经元的输出。以上面的 $z_1^{(2)}$ 的计算为例，$z_1^{(2)} = w_{11}^{(2)} * x_1 + w_{12}^{(2)} * x_2 + w_{13}^{(2)} * x_3 + b_1^{(2)}$，其中超参数 w 即为权重，超参数 b 为偏置。

- 激活函数处理

  激活函数的存在是神经网络的核心，激活函数 (一般为非线性函数) 的意义在于为神经网络提供非线性，使得神经网络能够学习到数据中的复杂模式，若激活函数为线性函数或者不存在，那么神经网络也仅仅相当于一个线性变换。常用的激活函数有 Relu、sigmod、LeakyRelu、tanh 等。

反向传播

上面已经提到，神经网络可以看作一个从输入到输出的带有超参数的函数。要使神经网络的输出尽可能的与真实结果类似，需要对超参数进行调节，而反向传播就是对神经网络中的超参数进行更新的过程。

反向传播的核心在于，选择一个代价函数 Cost(w,b)，其是关于超参数 w 和 b 的函数。反向传播过程就是寻找一个 w,b 的解使得代价函数的取值达到最小。而由于 Cost 函数基本上是非凸的，因此利用梯度下降法进行求解最小化 Cost 的问题，神经网络中利用链式法则求 Cost 对超参数的梯度，利用复合函数求导的链式法则，从输出层到输入层更新参数，反向传播算法由此得名。

6. 随机森林

随机森林是多个决策树的集合，是一种集成学习的算法，分类和回归都能使用。集成学习通过某种策略将多个单一弱模型集成起来，通过群体决策来提高决策准确率，主要有 Bagging 和 Boosting 两种方法。

Bagging，也就是 Bootstrap Aggregating，意思是自助抽样集成。Bootstrap 自助抽样，即从样本自身中再生成很多可用的同等规模的新样本，有放回地抽取，构成新的数据集（可能有重复样本）。Bagging 方法在每个新训练集上构建一个模型，各自不相干，最后预测时我们将每个模型的结果进行融合，得到最终结果。对于不同的模型进行不同的融合，即可得到比单一模型表现要好的融合模型。

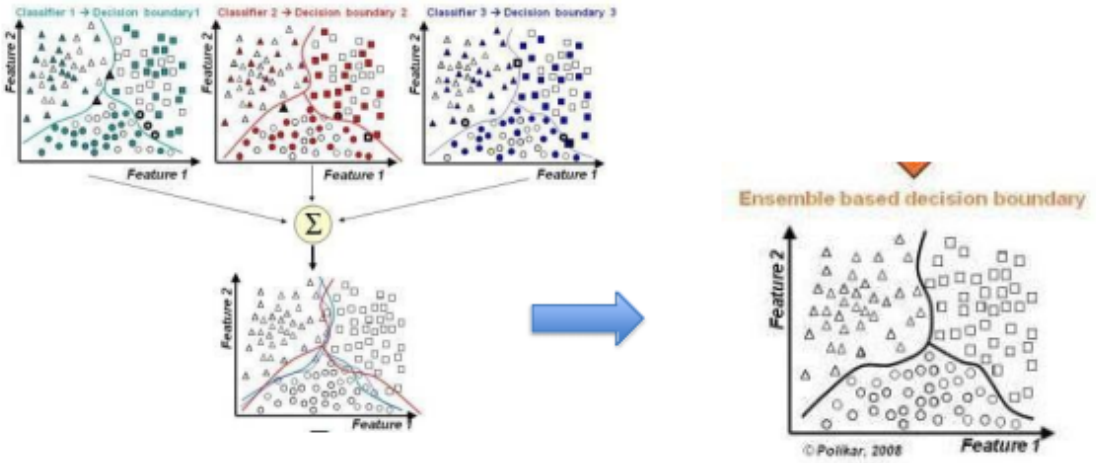


图 4: 随机森林

随机森林实际上是一种特殊的 bagging 方法，它将决策树用作 bagging 中的模型。首先，用 bootstrap 方法生成 m 个训练集，然后，对于每个训练集，构造一颗决策树，在节点找特征进行分裂的时候，并不是对所有特征找到能使得指标最大的，而是在特征中随机抽取一部分特征，

在抽到的特征中间找到最优解，应用于节点，进行分裂。随机森林的方法由于有了 bagging，也就是集成的思想在，实际上相当于对于样本和特征都进行了采样，所以可以避免过拟合。

为什么要有放回地抽样？我认为，如果不是有放回的抽样，那么每棵树的训练样本都是不同的，都是没有交集的，这样每棵树都是 “有偏的”，可能会相对 “片面的”，也就是说每棵树训练出来都是有很大的差异的。而随机森林最后分类取决于多棵树（弱分类器）的投票表决，这种表决应该是 “求同”，因此使用完全不同的训练集来训练每棵树这样对最终分类结果可能是没有帮助的。

随机森林在分类和回归的模型上基本一致，主要有三点不同。

方法	分类	回归
融合多个模型的方法	多数投票	均值
选择最优分裂特征的指标	基尼系数	平方误差
评价模型的指标	准确率	均方根误差

总而言之，随机森林作为一种集成学习模型，有以下特点：

- 优点
    1. 在广泛的数据集上表现良好，相比其他算法有很大的优势
    2. 能够处理高维数据，不用特征选择
    3. 泛化能力强，抗干扰能力强，不容易过拟合
    4. 如果有很大一部分的特征遗失，仍可以维持准确度
- 缺点
    1. 运行速度慢
    2. 在解决回归问题时没有像在分类中表现好，因为它并不能给出一个连续的输出，且不能够做出超越训练集数据范围的预测
    3. 在某些噪音较大的分类或者回归问题上会过拟合
    4. 对于小数据或者低维数据（特征较少的数据），可能不能产生很好的效果

7. 基于 KNN 的集成学习

KNN

K 近邻算法可用于分类和回归问题中，在分类问题中，基于某一种距离度量方式在训练集中选择与目标最近的 k 个数据，用多数投票法确定目标所属的类别。在回归问题中，基于某一种距离度量方式在训练集中选择与目标最近的 k 个数据，利用某种加权方式对这 k 个数据的得分进行求和，作为目标的得分。常用的距离度量方式有欧氏距离，曼哈顿距离，余弦相似度等。KNN 算法基于这样一个原理，与目标距离越接近的数据，越与目标近似。

Bagging 集成

对于给定的训练样本，每轮从训练样本中采用有放回抽样 (Booststraping) 的方式抽取 M 个样本，共进行 n 轮，这样我们就得到了 n 个相互独立的大小为 M 的样本集合。得到 n 个样本集合之后，对于每一个样本集合都训练一个预测模型，可以得到 n 个预测模型。综合利用这 n 个预测模型求解我们的问题。在分类算法中，通常利用多数投票法从这 n 个模型的输出中选择类别数最大的类别进行输出；在回归问题中，对着 n 个模型的输出进行某种加权求和，得到最终的结果。

基于 Bagging 的 KNN 集成学习

为了构建一个有效的集成 KNN 模型，每个子模型本身具有很高的准确性的同时，每个子模型之间的差异性要明显。仅仅利用单一的 booststraping 取样法抽取训练样本无法得到具有明显差异性的 KNN 模型，因此需要采取额外的方法。这里采用对特征进行采样的方法，即对于每一个 KNN 子模型，它的距离计算没有用到所有的特征属性，而是从这些特征属性中随机选择一些进行计算，每一个 KNN 子模型选择特征均不同。这样每一个 KNN 子模型不仅仅在选择的数据集上是相互独立的，而且在距离计算所需的特征上也是相互独立的，使得各个 KNN 子模型的差异更加明显。

四、方法

1. 流程图

本次实验的整体流程图如图5所示，后面会有对每个部分的详细介绍。

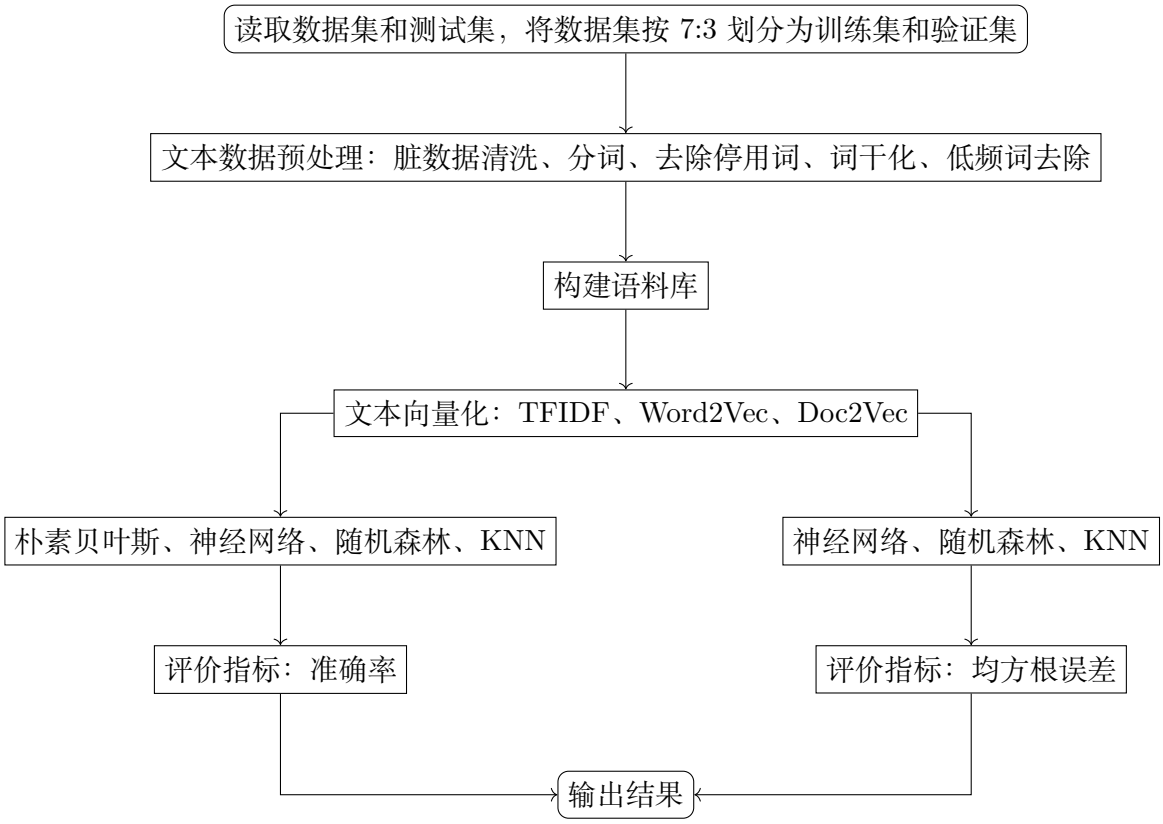


图 5: 整体流程图

2. 文本数据预处理

1. **脏数据清洗**: 在训练集中有一些乱码，因此我写了一个函数对于分词后的每个词语判断是否是英文字母，以此来去除脏数据。

```
def judge_pure_english(keyword):
    return all(((ord(c)>64 and ord(c)<91) or (ord(c)>96 and ord(c)<128)) for
        ↪ c in keyword)
```

2. **分词**: 需要对每个句子分割成英文单词，才能做 Embedding。在 TFIDF 和 Word2Vec 中，我使用了 jieba 库来进行分词，在 Doc2Vec 中我使用了其自带的分词工具，这些后面文本向量化部分会有具体的介绍。
3. **去除停用词**: 本次实验我使用了 nltk.corpus 中自带的 stopwords 作为停用词来去除。

```
from nltk.corpus import stopwords
import nltk
```

```
nltk.download('stopwords')
stop = set(stopwords.words('english'))
```

具体的 stopwords 如图6所示。

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\DELL\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
{'him', "aren't", "shan't", 'than', 'of', 'shouldn', 'off', 'isn', "isn't", 'do', 'those',
"shouldn't", "wasn't", 'not', 'we', 'just', 'once', 'his', 'here', 'hasn', 'he', "it's", 'i
', 'how', 'will', 'they', "needn't", 'you', 'o', 'yours', 'is', 'their', 'y', "you'll", "wo
uldn't", 'over', 'didn', 'but', "mightn't", 'wasn', "hadn't", 'now', 'or', "she's", 'were',
 'few', 'into', 'mightn', "couldn't", 'who', 'both', 'won', 'll', 'being', "you've", "haven
't", 'ma', "weren't", 'shan', 'been', 'a', 'ours', 'under', 'there', 'some', 'at', 'it', 'o
ther', 'so', 'before', 'doesn', 'further', 'me', "don't", 'as', 'an', 'in', 'above', 'own',
 'needn', 'what', "mustn't", 'have', 'with', "should've", 'weren', 'theirs', 'are', 'during
', 'that', 'through', 'be', 'hadn', 'having', 'ourselves', 'wouldn', 'can', 'your', 'then',
 'nor', 'below', 'has', "won't", 'most', 'haven', 'its', 'such', 'whom', 'yourselves', 'and
', 'she', 'any', 're', 's', "didn't", 'ain', 'by', 'couldn', 'them', "hasn't", 'herself', "
doesn't", 't', 'himself', 'too', 'yourself', 'her', "you'd", 'between', 'my', 'am', 'very',
 've', 'from', 'about', 'd', 'again', 'where', 'mustn', 'up', 'until', 'all', 'while', 'aga
inst', 'these', 'out', 'each', 'this', 'hers', 'should', 'after', 'only', 'itself', "you're
", 'why', 'which', 'more', 'because', 'the', 'themselves', 'don', 'on', 'myself', 'aren', '
was', 'm', 'doing', 'did', 'to', 'our', 'had', 'if', 'does', 'no', "that'll", 'when', 'down
', 'same', 'for'}
```

图 6: 停用词展示

在分词后，需要遍历每个句子的每个词语进行停用词的去除，之后有具体的代码解释。

4. **词干化**：英文单词可能会以各种形式出现，比如复数、现在时、过去式、过去分词等等。那么为了方便文本语义的分析，我们就需要对单词做词干化 stemming 处理。python 中有自带的 stemming 工具 PorterStemmer。

```
from nltk.stem.porter import PorterStemmer
porter = PorterStemmer()
```

每个单词的词干化也需遍历每个句子，可与去除停用词一起进行。

5. **低频词去除**：文本中许多词语出现次数极少，其实对文本语义分析没有作用，需要去除。在 TFIDF、Word2Vec 和 Doc2Vec 中，都可以设置参数进行低频词去除。

3. 文本向量化

(i) TFIDF

在实验一中我们已经实现过 TFIDF 矩阵的相关代码，但本次实验为了方便低频词去除和加快程序运行速度，我就调库实现了。

首先要对数据集预处理，对于每个句子，先使用 jieba 分词转为一个词语的 list，然后进行脏数据清洗、词干化、去除停用词。因为实现 TFIDF 的库输入的还是句子，所以最后要把处理

后的词语 list 还原成句子。

```
import jieba
def build(data):
    sentences = []
    for content in data:
        tokens = list(jieba.cut(content, cut_all=False)) # jieba分词
        words = []
        for word in tokens:
            if judge_pure_english(word): # 脏数据清洗
                temp = porter.stem(word) # 词干化
                if temp not in stop:    # 去除停用词
                    words.append(temp)
        sentence = " " .join(words)
        sentences.append(sentence)     # 还原成句子
    return sentences
```

通过上面的函数，构建训练集、验证集、测试集的语料库。

```
train_corpus = build(rd.train_data)
valid_corpus = build(rd.valid_data)
test_corpus = build(rd.test_data)
```

使用 CountVectorizer 将语料库中的每个句子转为稀疏计数矩阵，去除词频小于 2 的单词。

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer(stop_words='english', min_df=2)
# 先fit训练传入的文本数据，然后对文本数据进行标记并转换为稀疏计数矩阵
train_counts = vectorizer.fit_transform(train_corpus)
valid_counts = vectorizer.transform(valid_corpus)
test_counts = vectorizer.transform(test_corpus)
```

使用 TfidfTransformer 将计数矩阵转为 TFIDF 矩阵。

```
from sklearn.feature_extraction.text import TfidfTransformer
transform = TfidfTransformer()
# 使用TF-IDF（词频、逆文档频率）应用于稀疏矩阵
train_vectors = np.array(transform.fit_transform(train_counts).toarray())
valid_vectors = np.array(transform.transform(valid_counts).toarray())
test_vectors = np.array(transform.transform(test_counts).toarray())
```

最后得到训练集、验证集、测试集的 TFIDF 矩阵。

(ii) Word2Vec

文本数据预处理与 TFIDF 大同小异，只不过 Word2Vec 的 build 函数返回的是每个句子的词语 list，不用还原成句子。

```
import jieba
def build(data):
    sentences = []
    for content in data:
        tokens = list(jieba.cut(content, cut_all=False)) # jieba分词
        sentence = []
        for word in tokens:
            if judge_pure_english(word): # 脏数据清洗
                temp = porter.stem(word) # 词干化
                if temp not in stop:    # 去除停用词
                    sentence.append(temp)
        sentences.append(sentence)
    return sentences
```

建立训练集、验证集、测试集的语料库

```
train_corpus = build(rd.train_data)
valid_corpus = build(rd.valid_data)
test_corpus = build(rd.test_data)
```

接下来使用 Word2Vec 进行文本降维，设置向量维度 100，最小词频 2，4 线程并行处理，词语上下文窗口为 2，迭代 40 次。

```
from gensim.models import Word2Vec
model = Word2Vec(sentences=train_corpus,vector_size=100, min_count=2, workers = 4,
    ↪ window = 2)
model.save("word2vec.model")
model.train(train_corpus, total_examples=model.corpus_count, epochs=40)
```

得到的是每个词语的 100 维向量，转为句子的向量需要将句子中每个单词的向量加起来取均值，具体如make_sentence函数所示。(后来在验收时经助教提醒发现可以用 attention 的方法对每个词按权重转为句向量，但因时间实在有限就没有实现)

```
def make_sentence(corpus, bow):
    vectors = []
    for line in corpus:
        cnt = 0
        s = np.zeros(100)
        for word in line:
            if word in bow:
                cnt += 1
                vec = np.array(model.wv[word])
                s = s + np.array(vec) # 将每个句子中的词语向量加起来
        if cnt!=0: # 防止除数为0
            s = s / cnt
```

```
        s = list(s)
        vectors.append(s)
    return np.array(vectors)
```

最后得到训练集、验证集、测试集的 100 维 Word2Vec 矩阵。

```
train_vectors = make_sentence(train_corpus, model.wv.index_to_key)
valid_vectors = make_sentence(valid_corpus, model.wv.index_to_key)
test_vectors = make_sentence(test_corpus, model.wv.index_to_key)
```

(iii) Doc2Vec

分类

在 Doc2Vec 中我使用了其自带的分词器，接下来去除停用词和脏数据，并进行词干化处理。与 Word2Vec 不同的是，训练集在构建语料库时可以加入 label 进行训练。

```
def build_corpus(data_set, label_set, tokens_only=False):
    for i, line in enumerate(data_set):
        tokens = gensim.utils.simple_preprocess(line) # 分词
        sentence = []
        for word in tokens:
            if word not in stop and judge_pure_english(word): # 去除停用词和脏数据
                temp = porter.stem(word) # 词干化
                sentence.append(temp)
        if tokens_only:
            yield sentence
        else:
            yield gensim.models.doc2vec.TaggedDocument(sentence, [label_set[i]])
```

建立训练集、验证集、测试集的语料库（验证机和测试集不能加入 label）

```
train_corpus = list(build_corpus(rd.train_data, rd.train_labels))
valid_corpus = list(build_corpus(rd.valid_data, rd.valid_labels, True))
test_corpus = list(build_corpus(rd.test_data, rd.valid_labels, True))
```

使用 Doc2Vec 进行文本降维，设置向量维度 100，最小词频 2，4 线程并行处理，词语上下文窗口为 2，迭代 40 次。

```
import gensim
model = gensim.models.doc2vec.Doc2Vec(vector_size=100, min_count=2, epochs = 40,
    ↪ workers = 4, window = 2)
model.build_vocab(train_corpus)
model.train(train_corpus, total_examples=model.corpus_count, epochs=model.epochs)
```

最后将数据集中的每个句子映射为向量

```
train_vectors = []
```

```
for i in range(len(train_corpus)):
    inferred_vector = model.infer_vector(train_corpus[i].words)
    train_vectors.append(inferred_vector)
valid_vectors = []
for i in range(len(valid_corpus)):
    inferred_vector = model.infer_vector(valid_corpus[i])
    valid_vectors.append(inferred_vector)
test_vectors = []
for i in range(len(test_corpus)):
    inferred_vector = model.infer_vector(test_corpus[i])
    test_vectors.append(inferred_vector)
```

回归

在回归问题中，Doc2Vec 构建语料库需要加入 rating 进行训练而不是 label，其他部分与之前完全相同。

```
def build_corpus(data_set, rating_set=None, tokens_only=True):
    print(len(data_set))
    for i, line in enumerate(data_set):
        tokens = gensim.utils.simple_preprocess(line) # 分词
        sentence = []
        for word in tokens:
            if word not in stop and judge_pure_english(word): # 去除停用词和脏数据
                temp = porter.stem(word) # 词干化
                sentence.append(temp)
        if tokens_only:
            yield sentence
        else:
            yield gensim.models.doc2vec.TaggedDocument(sentence, [rating_set[i]])
```

4. 分类算法

(i) 朴素贝叶斯

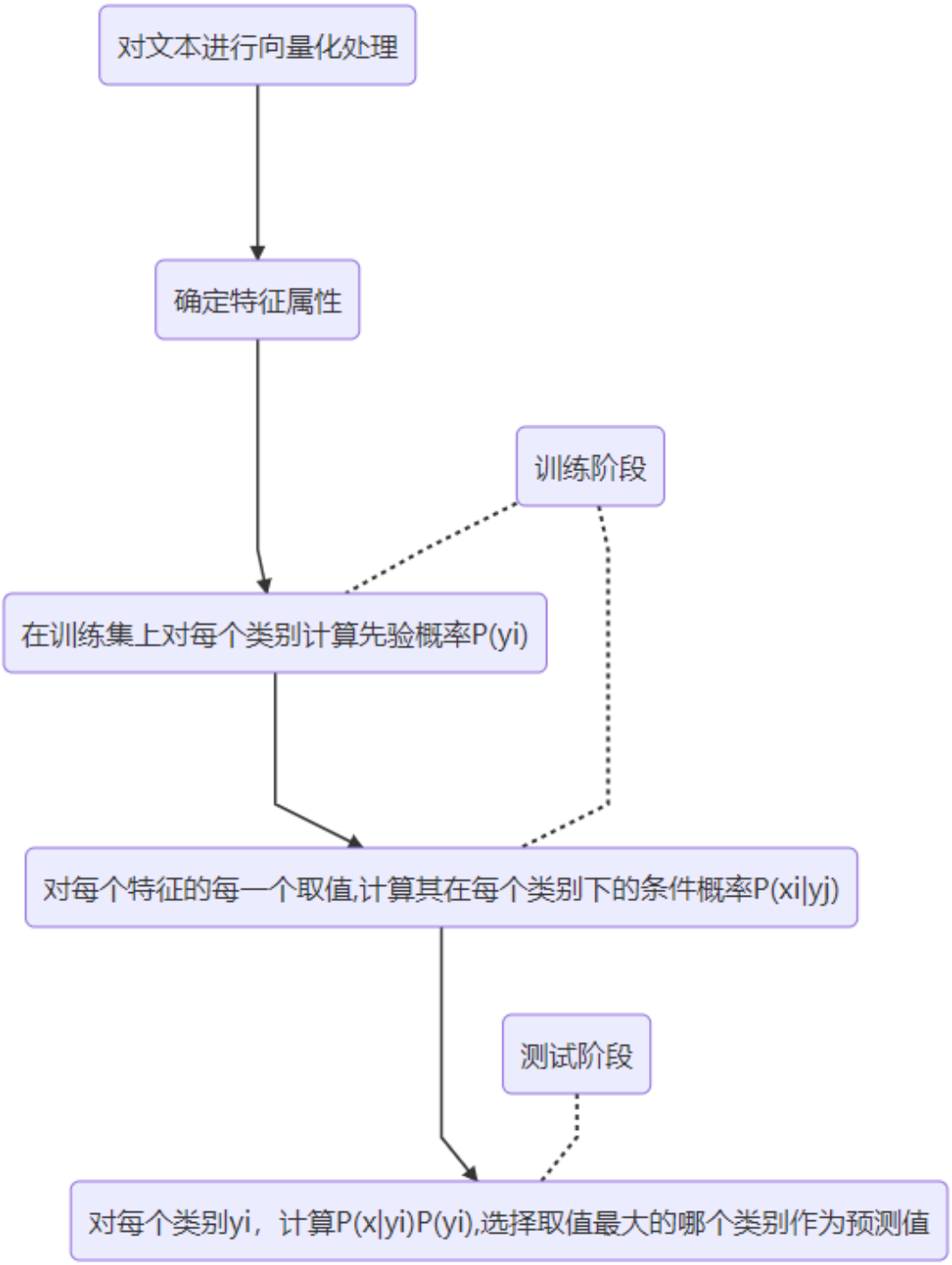


图 7: 朴素贝叶斯流程图

关键代码

先计算每一个类别出现的先验概率 $P(y_i)$，利用大数定理可知，$P(y_i)$ 可以利用训练集中类别 y_i 的文档出现的频率近似，$P(humor) = \frac{\text{训练集中幽默文本数目}}{\text{训练集中文本总数}}$。

```
numTrainDocs = len(trainVectors)                #计算训练的文档数目
numWords = len(trainVectors[0])                 #计算每篇文档的词条数
PriorHumor = sum(trainLabels)/float(numTrainDocs) #文档属于幽默类的概率
```

接下来计算条件概率，在英文文本的分类问题中，通常对英文文本进行向量化表示。利用朴素贝叶斯的条件独立性的假设得到 $P(text|y_i) = P(word_1|y_i) * P(word_2) * \cdots * P(word_n|y_i)$，因此需要做的就是统计各个单词在类别为 y_i 的文档中出现的频率，然后将文本 $text$ 中含有的单词的条件概率相乘，即得到 $text$ 在类别 y_i 的条件概率。对于 TFIDF 向量表示，可以将 TFIDF 向量中的值看作单词出现的频数。为了防止因为训练集过小导致一些词语并没有出现在训练集中，导致 0 概率出现，这里进行拉普拉斯平滑，将每一个单词出现的次数额外加一。

```
WordNumHumor = np.ones(numWords)
WordNumNotHumor = np.ones(numWords)             #创建numpy.ones数组,单词出现数初始化
    ↪ 为1，拉普拉斯平滑
for i in range(numTrainDocs):
    if trainLabels[i] == 1:                      #P(w0|1),P(w1|1),P(w2|1)
        WordNumHumor += trainVectors[i]
        HumorWordSum += sum(trainVectors[i])
    else:                                        #P(w0|0),P(w1|0),P(w2|0)
        WordNumNotHumor += trainVectors[i]
        NotHumorWordSum += sum(trainVectors[i])
HumorCondP = np.log(WordNumHumor/HumorWordSum)    #取对数，防止下溢出
NotHumorCondP = np.log(WordNumNotHumor/NotHumorWordSum)
```

已知一个文档的向量表示，下面计算其属于幽默文本，不幽默文本的后验概率的对数

$$\sum_{j=1}^{n} log(P(a_j|y_i)) + log(P(y_i))$$

比较这两个数的大小，选择数值较大的输出。前面乘以 testVector 的原因在于，对 $log(P(a_j|y_i))$ 进行加权求和，若 testVector 在某一个单词的取值较大，说明这个单词对于决定该文本的类别就越重要，因此加大其所占的比重。

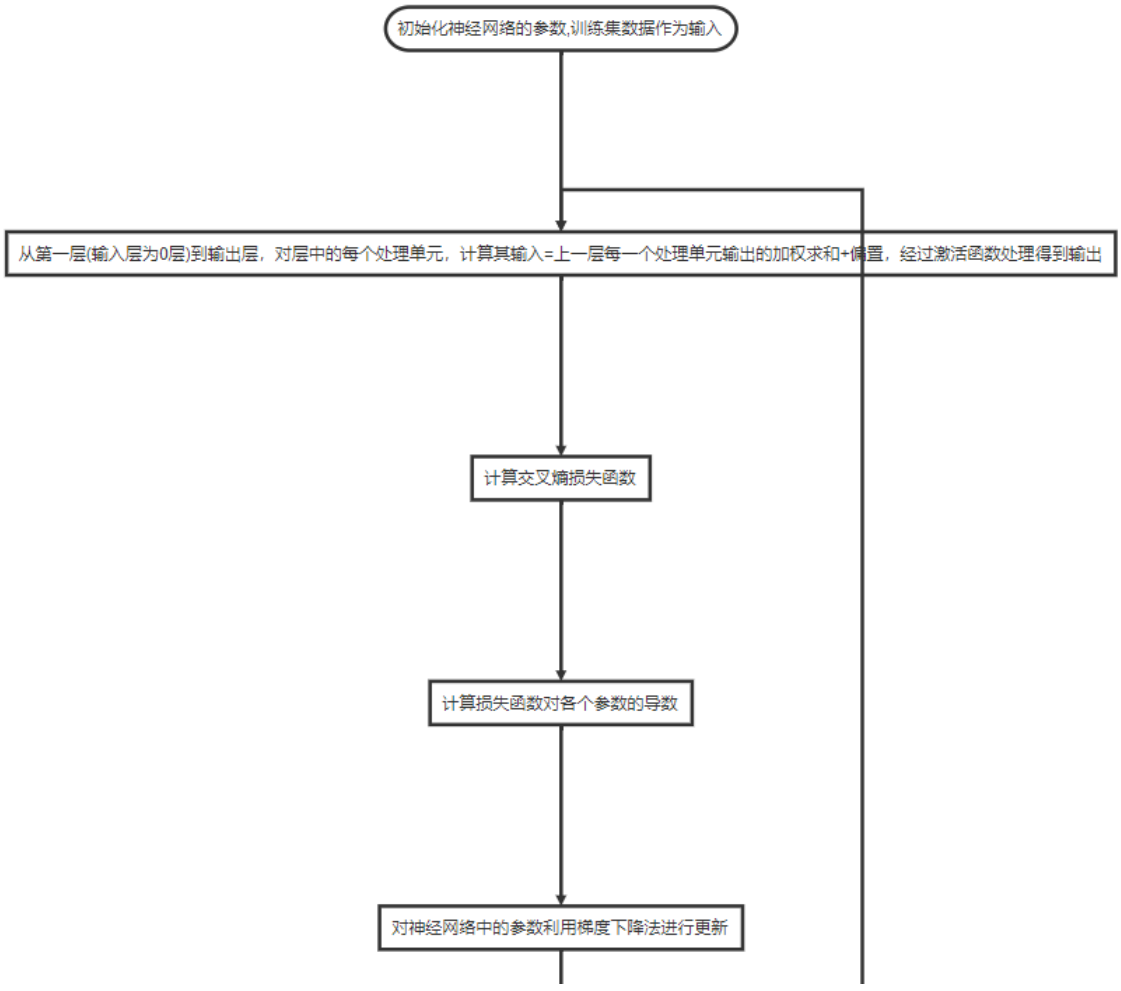
```
Humor = sum(testVector*HumorCondP) + np.log(PriorHumor)
NotHumor = sum(testVector*NotHumorCondP) + np.log(1.0 - PriorHumor)
if Humor > NotHumor:
    return 1
else:
```

```
    return 0
```

创新点及优化

$logP(text|y_i) = \sum_{j=1}^{n} log(P(a_j|y_i)) + log(P(y_i))$ 利用 $\sum_{j=1}^{N} Vec[j] * log(P(a_j|y_i)) + log(P(y_i))$ 的计算代替。计算 $P(text|y_i)$ 时，对于每一个在 text 中出现的单词 w，将其在 y_i 类的文档中频率累乘。若是 one-hot 或者词袋表示，很容易判断一个单词是否在文档 text 中出现，但是对于 TFIDF 向量表示，由于进行了拉普拉斯平滑，导致 TFIDF 中没有元素为 0，不容易判断一个词是否在 text 中出现，而考虑到 TFIDF 向量表示一个词对于判断一篇文档特征的重要性程度，于是想到在 $log(P(w_i|y_i))$ 前乘以 TFIDF 向量。

(ii) 神经网络



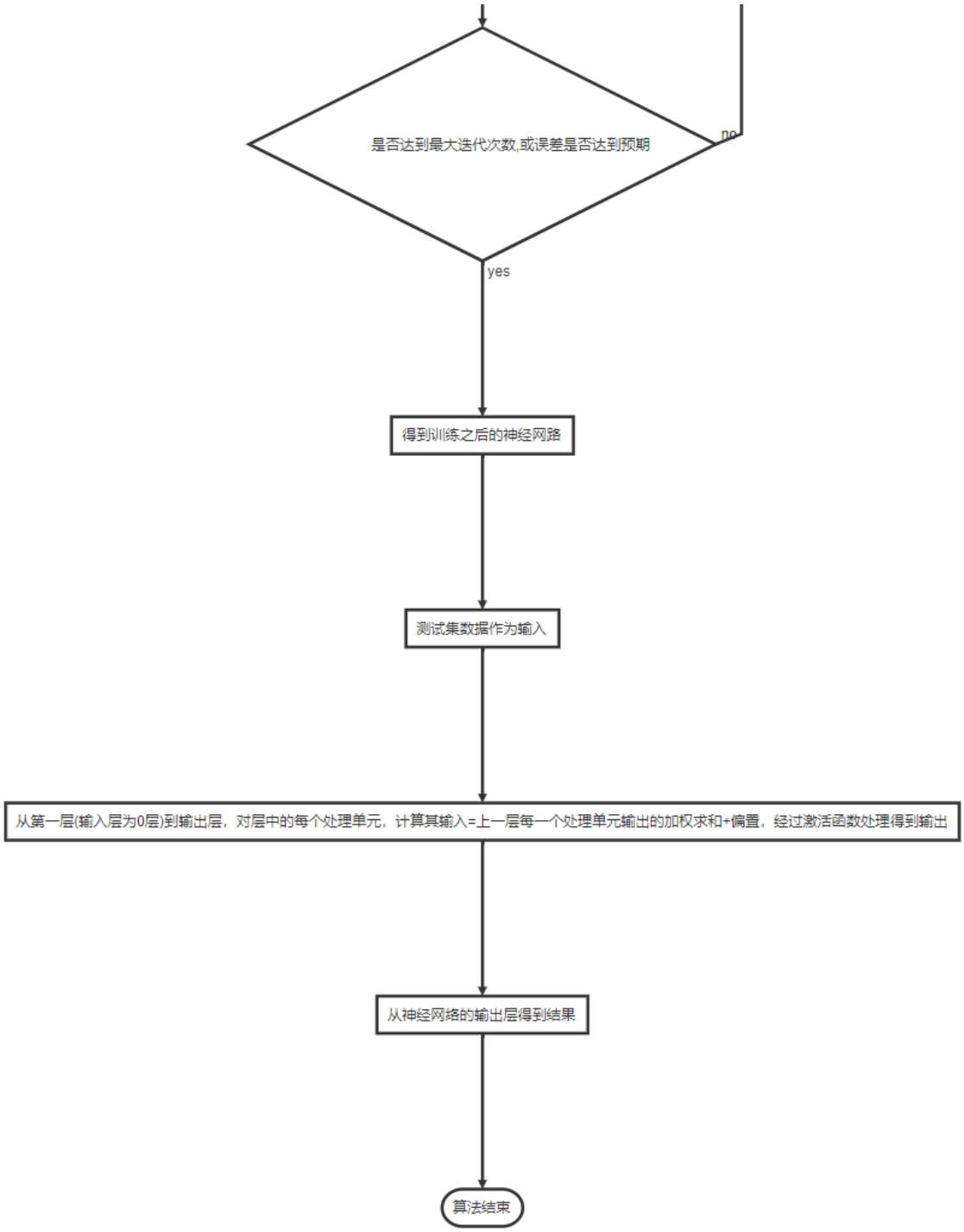


图 8: 神经网络流程图

神经网络前向后向传播公式推导

说明: 神经网络共有 L 层 (加上输入层)，输入层为第 0 层，隐藏层为第 1，2，...，L-2 层，输出层为第 L-1 层。

神经网络输入为英文文本向量矩阵，矩阵的每一列代表一个英文文本向量表示，矩阵的行数为文本向量维度 (也为输入层节点数目)，矩阵的列数为文本数目。神经网络输出层只有一个节点，代表文本是否幽默，神经网络输出为一个文本类别向量。符号说明:

- $layers\_dismension[i]$: 第 i 层神经网络的节点数目 (0<=i<=L-1)
- $Input[i]$: 代表第 i 层神经网络的输入 (0<=i<=L-1)。其中第 0 层 (输入层) 认为没有输入。其是一个 $layers\_dismension[i] * m$ 的向量，其值为上一层节点的加权求和，作为本层激活函数的输入
- $Output[i]$: 第 i 层神经网络的激活函数的输出
- $W[i]$: 代表第 i-1 层和第 i 层之间的连接上的权重参数
- $b[i]$: 第 i 层神经网络的偏置参数
- $loss$: 分类问题中，神经网络的损失函数，用交叉熵函数。

推导公式

- 前向传播推导

$$Output[0] = X$$

$$Input[i] = W[i] * Output[i-1] + b[i]$$

$$Output[i] = \sigma(Input[i]) \quad i \in [1, L-1)$$

- 反向传播推导

$$chain \quad rule:$$

$$loss = -\frac{1}{m}[log(Output[L-1]) * Y + log(1 - Output[L-1]) * (1 - Y)]$$

$$dOutput[L-1] = \frac{\partial loss}{\partial Output[L-1]} = -\frac{1}{m}[\frac{Y}{Output[L-1]} - \frac{1-Y}{1-Output[L-1]}]$$

$$dInput[i] = \frac{\partial loss}{\partial Output[i]} * \frac{\partial Output[i]}{\partial Input[i]} = dOutput[i] * \frac{\partial \sigma(Input[i])}{\partial Input[i]}$$

$$dW[i] = \frac{\partial loss}{\partial Output[i]} * \frac{\partial Output[i]}{\partial Input[i]} * \frac{\partial Input[i]}{\partial W[i]} = dInput[i] * Output[i-1]$$

$$db[i] = \frac{\partial loss}{\partial Output[i]} * \frac{\partial Output[i]}{\partial Input[i]} * \frac{\partial Input[i]}{\partial b[i]} = dInput[i] * \frac{\partial Input[i]}{\partial b[i]} = dInput[i]$$

$$dOutput[i-1] = dInput[i] * \frac{\partial Input[i]}{\partial Output[i-1]} = dInput[i] * W[i]$$

$$Gradient \quad descent:$$

$$W[i] = W[i] - \eta * dW[i]$$

$$b[i] = b[i] - \eta * db[j]$$

激活函数

1. sigmod 激活函数

   sigmod 函数将一个实数映射到 (0,1) 的区间，可以用来做二分类

$$f(x) = \frac{1}{1+e^{-x}}$$

   导数

$$f(x) = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{f(x)}{1-f(x)}$$

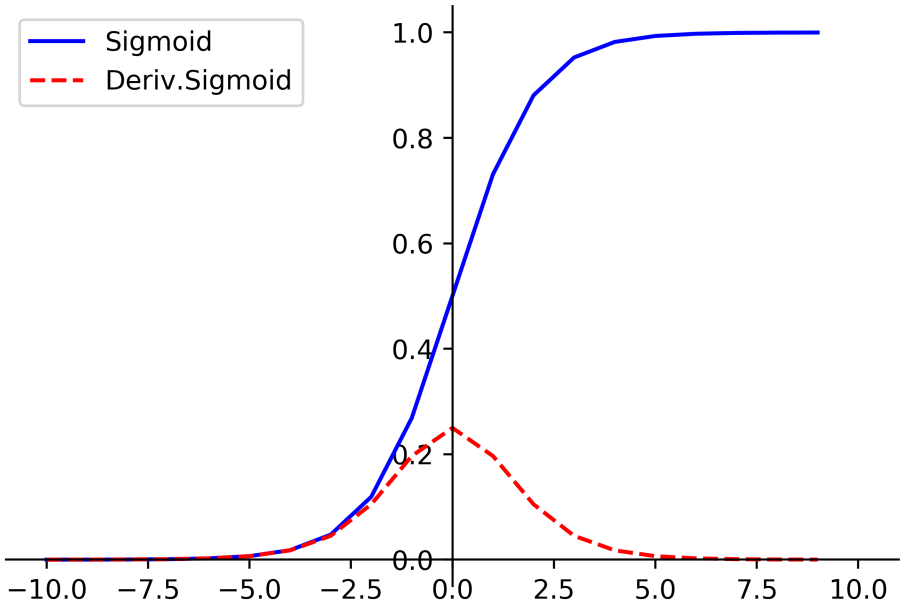


图 9: sigmod 激活函数和导数

特点: 当 x 超出 [-6,6] 的范围之后，函数值基本上没有什么变化，在应用中一般不考虑

优点:

(a) 将很大范围内的输入特征值压缩在 (0,1) 之间，使得在深层网络中可以保持数据幅度不会出现较大变化

(b) 由于其输出范围为 (0,1)，因此适用于将预测概率作为输出的模型

缺点:

(a) 当输入非常大或者非常小时，输出基本为常数，导致梯度会接近于 0，导致出现梯度

消失的现象，从而使得收敛速度较慢

(b) 幂运算相对耗时

(c) 输出不是 0 均值，随着网络的加深，会改变原始数据的分布趋势

2. Relu 激活函数

$$ReLU(x) = \begin{cases} x & x > 0 \\ 0 & x <= 0 \end{cases}$$

导数

$$\frac{dRelu(x)}{dx} = \begin{cases} 1 & x > 0 \\ 0 & x <= 0 \end{cases}$$

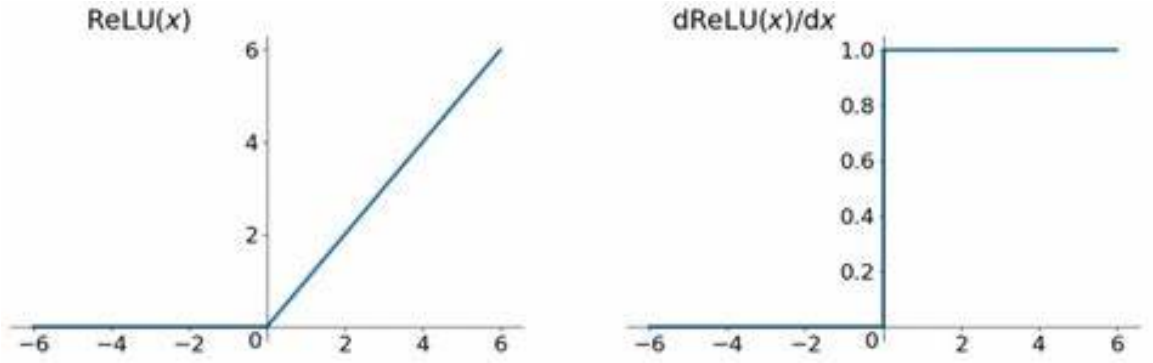


图 10: Relu 激活函数和导数

优点:

(a) 相对于 sigmoid 函数来说，在输入为正时，Relu 函数不存在梯度消失的问题，使得深层神经网络可以训练

(b) 计算速度很快，收敛速度很快

(c) Relu 输出会使得一部分神经元为 0 值，使得网络更加稀疏，一定程度上缓解了过拟合的问题

缺点:

(a) Relu 的输出不是以 0 为均值的函数

(b) 在训练的时候很脆弱，会出现神经元坏死的现象，由于 Relu 的导数在 x<0 时取值为 0，因此会出现某些神经元的梯度永远为 0 的情况，这些神经元可能永远不会被激活

3. Tanh 激活函数

$$tanh(x) = \frac{e^x - e^{-x}}{e^{-x} + e^x}$$

导数

$$(tanhx)' = sech^2x = 1 - tanh^2x$$

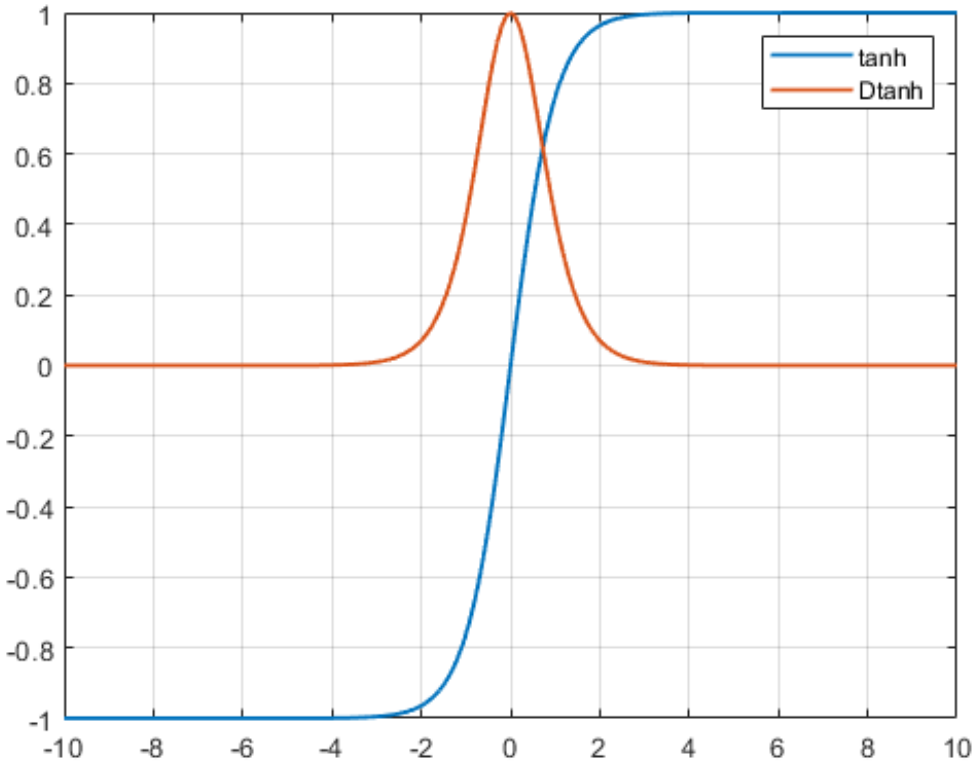


图 11: Tanh 激活函数和导数

优点:

(a) 输出为 0 均值

(b) Tanh 导数的取值范围在 0-1，比 sigmod 的导数范围更广，一定程度上减缓了梯度消失的问题

缺点:

(a) 梯度消失问题仍然存在

(b) 仍然存在幂运算

4. LeakyRelu 函数

$$LeakyRelu(x) = \begin{cases} x & x >= 0 \\ 0.01 * x & x < 0 \end{cases}$$

导数

$$\frac{dLeakyRelu(x)}{dx} = \begin{cases} 1 & x >= 0 \\ 0.01 & x < 0 \end{cases}$$

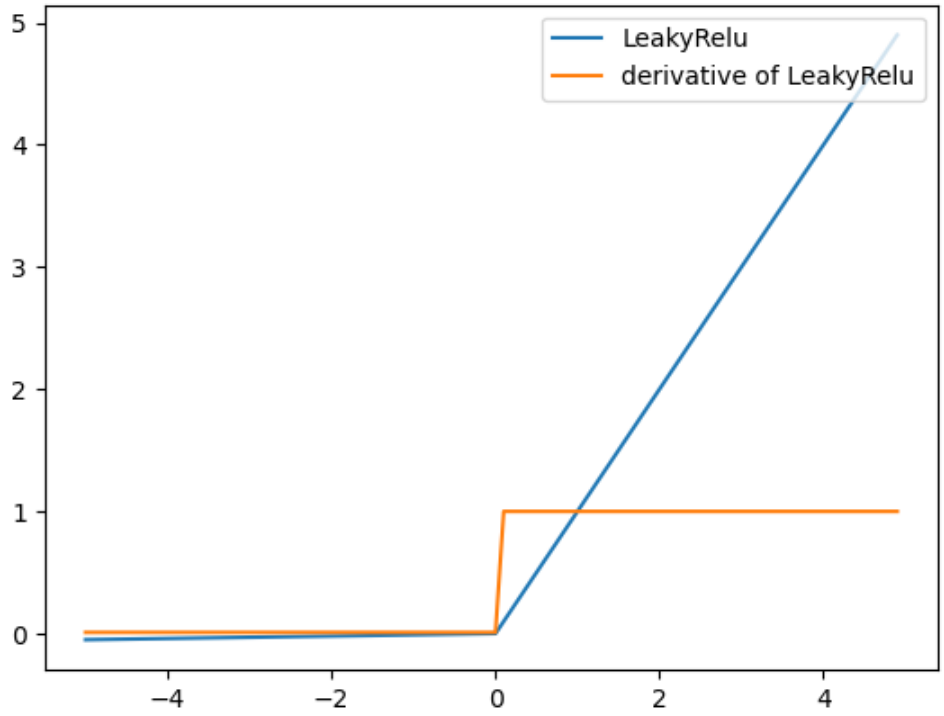


图 12: LeakyRelu 函数激活函数和导数

优点:

(a) 针对 Relu 函数中存在的 Dead Relu 问题，当输入为负值时，给予输入一个很小的斜率，解决了 relu 在负输入的情况下的梯度为 0 的情况，缓解了某些神经元不能激活的问题

(b) 取值范围负无穷到正无穷

缺点:

(a) 虽然理论上效果应当比 relu 函数要好，但在实际应用过程中，效果不太好

神经网络优化

- L2 正则化

  神经网络中会出现过拟合的问题，通过正则化损失函数能够避免过拟合现象的发生，L2 正则化也叫权重衰减，在原本的损失函数上加一个惩罚项。

$$loss_{L2} = loss + \frac{\lambda}{2 * n} \sum w^2$$

  w 是神经网络中所有权重，λ 是正则化参数 ($\lambda > 0$)。上述正则化损失函数综合考虑了模型效果和模型的复杂度，若 λ 很小，则更加考虑模型效果，若 λ 很大，则更倾向于惩罚模型的复杂度。通过在损失函数中加入 L2 正则化，要使 loss 函数较小，L2 的项的取值就应该减小，使得一部分权重趋于 0，这样就减少了神经网络的参数数量，使得模型复杂度降低，避免了过拟合的问题。

  加入 L2 正则化之后，对权重 W 的更新公式也发生了变化。

$$\frac{\partial loss_{L2}}{\partial W} = \frac{\partial loss}{\partial W} + \frac{\lambda}{n} * W$$
$$W \leftarrow W - \eta(\frac{\partial loss}{\partial W} + \frac{\lambda}{n} * W)$$

  其中 $\frac{\partial loss}{\partial W}$ 已在上面推导。

- mini-batch 梯度下降

  mini-batch 梯度下降，适用于数据集比较大的情况下。在数据集比较小的情况下，可以利用全数据集的形式对参数进行更新，由全数据集确定的方向能够更好的代表样本总体，使得参数的更新准确的向极值所在的方向进行更新。但是在数据集较大的情况下，一次性载入所有数据变得很困难，而且训练起来速度较慢。可以将训练集分割成数份，一次只学习一份数据集，利用这一个数据集对神经网络中的参数进行更新，当数据集比较充分时，利用一部分的数据就可以代替整体的数据集，这样使得网络收敛的速度加快，解决了内存不足的问题，batch 的大小没有什么特别的标准，但是这个参数会影响神经网络的训练效果，一般设置为 2 的整数幂。

  “洗牌”：每一次从数据集中选择”batch” 份数据进行训练都是随机选取的，每一次神经网络学习的数据均是不同的，为其中增加了一些随机的因素，在实践中证明此举会优化神经网络。

关键代码

神经网络的参数初始化，神经网络的参数初始化对神经网络的训练效果有着非常大的影响，

如果对神经网络中的参数进行随机初始化，可能会导致梯度下降和梯度消失。对于不同的激活函数，参数的初始化是不同的，在 relu 激活函数下，一般对权重参数执行 MSRA 初始化，MSRA 初始化是一个均值为 0 方差为 2/n 的高斯分布。

```
for i in range(1, L):
    if i < L-1:
        W[i] = np.random.randn(layers_dismension[i], layers_dismension[i-1])* np.
            ↪ sqrt(2 / layers_dims[i - 1])#MSRA初始化
    else:
        W[i] = np.random.randn(layers_dismension[i],layers_dismension[i-1])*0.01
    b[i] = np.zeros((layers_dismension[i], m))
```

前向传播过程，即从神经网络的输入层经过各个处理节点的激活以及神经元之间链接的加权求和处理，得到最终输出的过程，Input[j] 为第 j 层的激活函数输入，Output[j] 为第 j 层的激活函数的输出.

```
for j in range(1, L):
    Input[j] = np.dot(W[j], Output[j-1])+b[j]
    if j < L-1:
        Output[j] = LeakyRelu(Input[j])
    else:
        Output[j] = sigmoid(Input[j])
```

计算神经网络的误差损失函数，分类问题中一般采用交叉熵损失函数

```
loss = -1/m * (np.dot(np.log(Output[L-1]), Y.T) + np.dot(np.log(1-Output[L-1]), 1-
    ↪ Y.T))#交叉熵
```

加入 L2 正则化，即对所有权重参数进行求和

```
    regular_cost = 0
    for j in range(1,L):
        regular_cost += np.sum(np.square(W[j]))*lamd/(2*m) #正则化
    loss += regular_cost
```

反向传播过程对参数进行更新，更新的公式已在上面推导出，这里可以改变各个层使用的激活函数

```
    dOutput[L-1] = -1/m*(np.divide(Y,Output[L-1]) - np.divide(1-Y,1-Output[L-1]))
    for j in range(L-1, 0, -1):
        if j == L-1: # sigmoid
            dInput[j] = dOutput[L-1]*der_sigmoid(Input[L-1])
        else: # leakyRelu
            dInput[j] = dOutput[j]*der_LeakyRelu(Input[j])
        dW[j] = np.dot(dInput[j], Output[j-1].T)
        db[j] = dInput[j]
```

```
        dOutput[j-1] = np.dot(W[j].T, dInput[j])
        W[j] = W[j]-learning_rate*dW[j]
        b[j] = b[j] - learning_rate*db[j]
```

mini-batch梯度下降代码

mini-batch梯度下降,首先利用 numpy 中的函数 choice 从所有的文本向量中选择batch_size个向量下标，作为训练神经网络的数据，replace = False 表明这些下标中没有重复

```
batch_mask = np.random.choice(X.shape[1], batch_size,replace=False)
```

前向传播，和前面的前向传播类似，由于这里只考虑下标在batch_mask中的文本向量。因此利用切片操作，从 Input,Ouput 中选择对应的列。

```
for j in range(1, L):
    Input[j][:, batch_mask] = np.dot(W[j], Output[j-1][:, batch_mask])+b[j]
    if j < L-1:
        Output[j][:, batch_mask] = LeakyRelu(Input[j][:, batch_mask])
    else:
        Output[j][:, batch_mask] = sigmoid(Input[j][:, batch_mask])
```

反向传播，和全数据集的更新操作类似，同样需要利用切片操作，选择在batch_mask中的文本向量。正则化中的 m 要改为batch_size。

```
        dOutput[L-1][:, batch_mask] = -1/batch_size*(np.divide(Y[batch_mask],Output
            ↪ [L-1][:, batch_mask]) - np.divide(1-Y[batch_mask],1-Output[L-1][:,
            ↪ batch_mask]))
        for j in range(L-1, 0, -1):
            if j == L-1: # sigmoid
                dInput[j][:, batch_mask] =dOutput[L-][:,batch_mask]*der_sigmoid(
                    ↪ Input[L-1][:, batch_mask])
            else: # Relu
                dInput[j][:, batch_mask] =dOutput[j[:,batch_mask]*der_LeakyRelu(
                    ↪ Input[j][:, batch_mask])
            dW[j] = np.dot(dInput[j][:, batch_mask], Output[j-1][:, batch_mask].T)+
                ↪ lamd/batch_size*W[j]
            db[j] = 1/batch_size* np.sum(dInput[j][:, batch_mask], axis=1, keepdims=
                ↪ True)
            dOutput[j-1][:, batch_mask] = np.dot(W[j].T, dInput[j][:, batch_mask])
            W[j] = W[j]-learning_rate*dW[j]
            b[j] = b[j] - learning_rate*db[j]
```

(iii) 随机森林

随机森林由多课决策树构成，首先需要定义决策树。cal_predict_value函数用于验证集和测试集的预测。通过数据集对应特征的值大于还是小于决策树每个分裂点的阈值，不断递归找到叶子结点的值。

```
class Tree(object):
    def __init__(self):
        self.split_feature = None # 分裂的特征
        self.split_value = None # 分裂的特征值
        self.leaf_value = None  # 叶子节点的值
        self.tree_left = None   # 左子树
        self.tree_right = None  # 右子树

    # 通过递归决策树找到样本所属叶子节点（用于预测）
    def cal_predict_value(self, dataset):
        if self.leaf_value is not None:
            return self.leaf_value
        elif dataset[self.split_feature] <= self.split_value:
            return self.tree_left.cal_predict_value(dataset)
        else:
            return self.tree_right.cal_predict_value(dataset)
```

这次我写的随机森林共有 6 个参数。tree_num表示树的数量；max_depth表示树的最大深度；min_samples_split是节点分裂所需的最小样本数量，小于该值节点终止分裂；col_samples_rate是列采样设置，默认是随机选择 sqrt(n_features) 个特征；row_samples_rate表示行采样比例；random_state是随机种子，确保实验可重复。

```
class RandomForestClassifier(object):
    def __init__(self, tree_num, max_depth, min_samples_split, col_samples_rate,
        ↪ row_samples_rate, random_state):
        self.tree_num = tree_num
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.col_samples_rate = col_samples_rate
        self.row_samples_rate = row_samples_rate
        self.random_state = random_state
        self.trees = None
```

fit函数是模型的训练入口，用于传入训练集。随机种子可以使每次建树的采样都是随机的。设置列采样的方法，每次取平方根数目的特征。并且利用所有线程并行建树，加快运行速度。

```
    def fit(self, dataset, targets):
        if self.random_state:
            random.seed(self.random_state)
        random_state_stages = random.sample(range(self.tree_num), self.tree_num)

        # 列采样
        if self.col_samples_rate == "sqrt":
            self.col_samples_rate = int(math.sqrt(len(dataset.columns)))
```

```
        else:
            self.col_samples_rate = len(dataset.columns)

        # 并行建立多棵决策树
        self.trees = Parallel(n_jobs=-1, backend="threading")(
            delayed(self.parallel_build_trees)(dataset, targets, random_state)
                for random_state in random_state_stages)
```

parallel_build_trees函数用于随机采样建树，每次采样都是有放回的。先随机抽取一定数目的特征，然后将训练集和标签按行采样比例随机采样，采样完成后准备建树。

```
    def parallel_build_trees(self, dataset, targets, random_state):
        feature_samples = random.sample(list(dataset.columns), self.
            ↪ col_samples_rate)
        dataset_stage = dataset.loc[:, feature_samples]
        dataset_stage = dataset_stage.sample(n=int(self.row_samples_rate*len(
            ↪ dataset)), replace=True, random_state=random_state).reset_index(drop
            ↪ =True)
        targets_stage = targets.sample(n=int(self.row_samples_rate*len(dataset)),
            ↪ replace=True, random_state=random_state).reset_index(drop=True)
        tree = self.build_single_tree(dataset_stage, targets_stage, depth=0)
        return tree
```

build_single_tree函数用于递归建立决策树。如果分裂后的节点类别全部一样，或者小于分裂所需最小样本数量，则终止分裂，采取多数投票计算叶子结点的值。建树至最大深度前，需要不断选取最优特征，划分数据集，然后将树分为左子树和右子树；否则，就要用多数投票计算叶子节点的值。

```
    def build_single_tree(self, dataset, targets, depth):
        # 如果该节点的类别全都一样/样本小于分裂所需最小样本数量，则终止分裂
        if len(np.unique(targets['label'])) <= 1 or len(dataset) <= self.
            ↪ min_samples_split:
            tree = Tree()
            tree.leaf_value = self.cal_leaf_value(targets['label'])
            return tree

        if depth < self.max_depth:
            best_split_feature, best_split_value = self.choose_best_feature(dataset,
                ↪  targets)
            left_dataset, right_dataset, left_targets, right_targets = self.
                ↪ split_dataset(dataset, targets, best_split_feature,
                ↪ best_split_value)

            tree = Tree()
            tree.split_feature = best_split_feature
```

```
            tree.split_value = best_split_value
            tree.tree_left = self.build_single_tree(left_dataset, left_targets,
                ↪ depth+1)
            tree.tree_right = self.build_single_tree(right_dataset, right_targets,
                ↪ depth+1)
            return tree
        # 如果树的深度超过预设值，则终止分裂
        else:
            tree = Tree()
            tree.leaf_value = self.cal_leaf_value(targets['label'])
            return tree
```

choose_best_feature函数用于选取最优分裂特征。如果采样的数据集中某个维度的值过多，那么就需要选择 100 个百分位值作为待选分裂阈值。接下来对每个分裂阈值求基尼指数，选取基尼指数最小的作为最优分裂特征。

```
    def choose_best_feature(self, dataset, targets):
        best_split_gain = float("inf")
        best_split_feature = None
        best_split_value = None

        for feature in dataset.columns:
            if len(np.unique(dataset[feature])) <= 100:
                unique_values = np.unique(dataset[feature])
            # 如果该维度特征取值太多，则选择100个百分位值作为待选分裂阈值
            else: # unique去除数组中的重复数字并进行排序之后输出 percentile 求%0到%100
                ↪ 的100个点
                unique_values = np.unique([np.percentile(dataset[feature], x)for x
                    ↪ in np.linspace(0, 100, 100)])

            # 对可能的分裂阈值求分裂增益
            for split_value in unique_values:
                left_targets = targets[dataset[feature] <= split_value]
                right_targets = targets[dataset[feature] > split_value]
                split_gain = self.cal_gini(left_targets['label'], right_targets['
                    ↪ label'])

                if split_gain < best_split_gain:
                    best_split_feature = feature
                    best_split_value = split_value
                    best_split_gain = split_gain
        return best_split_feature, best_split_value
```

cal_leaf_value函数用于多数投票计算叶子节点的值。

```
    def cal_leaf_value(self, targets):
        label_counts = collections.Counter(targets)
        if (label_counts[0] > label_counts[1]):
            return 0
        else:
            return 1
```

cal_gini函数是基尼指数的计算方法。

```
    def cal_gini(self, left_targets, right_targets):
        gain = 0
        for targets in [left_targets, right_targets]:
            gini = 1
            # 统计每个类别有多少样本，然后计算gini
            label_counts = collections.Counter(targets)
            for key in label_counts:
                prob = label_counts[key]*1.0/len(targets)
                gini -= prob**2
            gain += len(targets)*1.0/(len(left_targets)+len(right_targets))*gini
        return gain
```

split_dataset函数根据最优特征的阈值将样本划分成左右两份，左边小于等于阈值，右边大于阈值。

```
    def split_dataset(self, dataset, targets, split_feature, split_value):
        left_dataset = dataset[dataset[split_feature] <= split_value]
        left_targets = targets[dataset[split_feature] <= split_value]
        right_dataset = dataset[dataset[split_feature] > split_value]
        right_targets = targets[dataset[split_feature] > split_value]
        return left_dataset, right_dataset, left_targets, right_targets
```

最后设置随机森林的参数，开始训练。分别计算模型在训练集和验证集的准确率，最后输出测试集上的结果。

```
RF = RandomForestClassifier(tree_num=5, max_depth=5, min_samples_split=400,
    ↪ col_samples_rate='sqrt',row_samples_rate=0.4,random_state=77)
RF.fit(train_vectors, train_labels)
print(metrics.accuracy_score(train_labels, RF.predict(train_vectors)))
print(metrics.accuracy_score(valid_labels, RF.predict(valid_vectors)))
test_predict = RF.predict(test_vectors)
test_output = pd.DataFrame({'id': np.arange(len(test_predict))+8001, 'is_humor':
    ↪ test_predict})
test_output.to_csv('output1.csv', index=None, encoding='utf8')
```

(iv) 基于 KNN 的集成学习

图 13: 基于 KNN 的集成学习分类流程图

关键代码

利用多个 KNN, 每一个 KNN 基于不同的距离度量方式 (欧式距离，曼哈顿距离等)，对于每一个 KNN 模型，分别随机选择特征计算距离，训练每个 KNN 得到效果最好的 k 值，此外，在训练时，对于每个 KNN，训练数据的选择也是从总的数据中随机选择的，即每个 KNN 在训练数据上是相互独立的，在计算距离选择的特征上也是相互独立的。

首先定义一个集成 KNN 类，定义如下，类中的几个变量表征集成 KNN 模型

```
class EnsembleKNN(object):
    def __init__(self):
        self.KSet = None # 对应每个不同距离度量方式knn选择的不同的k
        self.dis_type_Set = None # 对应选择的距离度量方式 欧式距离 曼哈顿距离
        self.col_choice_Set = None # 每个KNN选择的不同的特征
        self.row_choice_Set = None # 每个KNN选择的不同的数据
```

建立单个 KNN 子模型，在某种距离度量方式下，已经为该 KNN 子模型选择好特征和数据集，KNNSubModel 就是在选择的特征下利用验证集得到该 KNN 子模型对应的最好的 K, 我们的集成 KNN 模型就是通过 KNNSubModel 训练出不同的 KNN 子模型，然后将这些子模型进行集成。

```
    '''
    KNNSubModel:
    SampleTrainDataset:经过采样后的训练集
    SampleTrainLabel:经过采样后的训练集标签集合
```

```
    SampleValidDataset:经过采样后的验证集
    SampleValidLabel:经过采样后的验证集标签集合
    dis_type:该KNN子模型选择的距离度量方式
    返回值:该KNN子模型对应的最优的k
    '''
    def KNNSubModel(self, SampleTrainDataset, SampleTrainLabel, SampleValidDataset,
        ↪ SampleValidLabel, dis_type):
        k_to_accuracy = {}
        for k in range(3, 80):
            predict_res = []
            for line in SampleValidDataset:
                dis_list = []
                for ID, record in enumerate(SampleTrainDataset):
                    dis = self.ComputeDis(line, record, dis_type)
                    l = []
                    l.append(dis)
                    l.append(SampleTrainLabel[ID]) # 训练集中编号为ID的文档对应的标签
                    dis_list.append(l)
                dis_list = sorted(dis_list, key=(lambda x: x[0])) # 按距离升序排序
                dict1 = {}
                j = 0
                while j < k:
                    if dis_list[j][1] not in dict1:
                        dict1[dis_list[j][1]] = 1 # 统计前k篇文档中每个类别各有多少文章
                    else:
                        dict1[dis_list[j][1]] += 1
                    j += 1
                predict_res.append(max(dict1, key=lambda x: dict1[x])) # 类别取众数
            accuracy = np.sum(np.array(predict_res) == np.array(
                SampleValidLabel))/len(predict_res)
            k_to_accuracy[k] = accuracy

        return max(k_to_accuracy, key=k_to_accuracy.get)#选择最高的准确率对应的K
```

下面的代码为为每一个 KNN 子模型进行数据集和特征的随机选择，然后利用这些选择出的数据进行训练 KNN 子模型 (即找到最优 K)。

```
    '''
    train_dataset:全部的训练集数据
    train_label:全部的训练集标签
    valid_dataset:全部的验证集数据
    valid_label:全部的验证集标签
    dis_type:指定的距离度量方式列表,将根据不同的距离度量方式,训练出len(dis_type)个knn
        ↪ 子模型进行集成
    ColSampleNum:采样的特征得数目
```

```
    RowSampleNum:采样得数据的数目
    random_state_row/random_state_col:随机种子,由于数据和特征均要用到random.sample进行随机采样,为了保证实验的可重复性,每一列,每一行的选择均设置随机种子
    '''
    def EnsembleModel(self, train_dataset, train_label, valid_dataset, valid_label, dis_type, ColSampleNum, RowSampleNum, random_state_row, random_state_col):
        random.seed(random_state_col)
        SampleColIndex = random.sample(
            range(len(train_dataset[0])), int(ColSampleNum))#随机采样特征
        random.seed(random_state_row)
        SampleRowIndex = random.sample(
            range(len(train_dataset)), int(RowSampleNum))#随机采样数据
        SampleTrainDataset = np.array(train_dataset)[SampleRowIndex, :]#得到采样后的训练集
        SampleTrainDataset = SampleTrainDataset[:, SampleColIndex]
        SampleTrainLabel = np.array(train_label)[SampleRowIndex]#得到采样后的训练集标签
        SampleValidDataset = np.array(valid_dataset)[:, SampleColIndex]#只对验证集进行特征采样
        k = self.KNNSubModel(SampleTrainDataset.tolist(), SampleTrainLabel.tolist(
        ), SampleValidDataset.tolist(), valid_label, dis_type)
        return k, dis_type, SampleColIndex, SampleRowIndex#该KNN子模型由该4个数据进行描述
```

类 EnsembleKNN 的 train() 函数利用训练集和验证集得到集成 KNN 模型，参数dis_type指定集成 KNN 由几个 KNN 子模型组成，以及每个 KNN 子模型所选择的距离度量方式是什么

```
    '''
    训练
    通过定义一个EnsembleKNN实例，然后调用train()，得到EnsembleKNN模型。train_dataset,train_label:训练数据集
    valid_dataset,valid_label:验证集
    dis_type:字符串列表,每一个字符串代表一个距离类型,len(dis_type)表示总的子KNN模型。
    '''
    def train(self, train_dataset, train_label, valid_dataset, valid_label, dis_type, random_state=None):
        if random_state:
            random.seed(random_state)
        random_seed_list = []
        for i in range(2*len(dis_type)):
            random_seed_list.append(i+random_state)#设置随机种子
        ColSampleNum = 0.8*len(train_dataset[0])#决定为每个子KNN模型选择的特征的数目
        RowSampleNum = 0.8*len(train_dataset)#决定为每个子KNN模型选择的数据数目
```

```
        self.KSet, self.dis_type_Set, self.col_choice_Set, self.row_choice_Set =
            ↪ zip(*Parallel(n_jobs=-1, verbose=0, backend="threading")(
        delayed(self.EnsembleModel)(train_dataset, train_label, valid_dataset,
            ↪ valid_label, dis_type[i], ColSampleNum, RowSampleNum,
            ↪ random_seed_list[2*i], random_seed_list[2*i+1]) for i in range(
            ↪ len(dis_type))))
```

集成 KNN 模型在训练时得到数个 KNN 子模型，其集成表现在预测分类结果上，集成 KNN 模型在分类上采用多数投票法，由于在训练每个 KNN 子模型时进行随机的特征选择和数据选择，因此各个 KNN 的差异性比较大，因此在分类时，能够利用不同的特征。

```
    '''
    测试
    对于集成KNN中的每个KNN子模型,在测试集上得到预测分类结果,然后利用投票法得到最终的分
        ↪ 类
    '''
    def test(self, train_dataset, train_label, test_dataset):
        res = np.zeros(len(test_dataset))
        for i in range(len(self.KSet)): # 第i个knn子模型
            k = self.KSet[i]
            dis_type = np.array(self.dis_type_Set[i])
            ColChoice = np.array(self.col_choice_Set[i])
            RowChoice = np.array(self.row_choice_Set[i])

            SampleTrainDataset = np.array(train_dataset)[RowChoice, :]
            SampleTrainDataset = np.array(SampleTrainDataset)[:, ColChoice]
            SampleTrainLabel = np.array(train_label)[RowChoice]
            SampleTestDataset = np.array(test_dataset)[:, ColChoice]
            temp = []
            for line in SampleTestDataset:
                dis_list = []
                for ID, record in enumerate(SampleTrainDataset):
                    dis = self.ComputeDis(line, record, dis_type)
                    l = []
                    l.append(dis)
                    l.append(SampleTrainLabel[ID]) # 训练集中编号为ID的数据对应的标签
                    dis_list.append(l)
                dis_list = sorted(dis_list, key=(lambda x: x[0])) # 按距离升序排序
                dict1 = {}
                j = 0
                while j < k:
                    if dis_list[j][1] not in dict1:
                        dict1[dis_list[j][1]] = 1 # 统计前k篇文档中每个类别各有多少文章
                    else:
                        dict1[dis_list[j][1]] += 1
```

```
                j += 1
            temp.append(max(dict1, key=lambda x: dict1[x])) # 类别取众数
        print(res)
        res = res + np.array(temp)
    ans = []
    for i in range(len(res)):
        if res[i] >= len(self.KSet)/2:
            ans.append(1)
        else:
            ans.append(0)
    return np.array(ans)
```

5. 回归算法

(i) 神经网络

回归算法和分类算法基本一致，下面主要说明其不同的地方

- 损失函数的选择

  分类算法中，损失函数选择交叉熵函数，而当神经网络用于回归问题时，损失函数通常选择均方差函数

$$MSE = \frac{1}{2N}\sum_{i=1}^{N}(y_i - \hat{y_i}^2)$$

  前面的 $\frac{1}{2N}$ 是为了更好的求导。这就导致损失函数对神经网络的输出的求导变为

$$\frac{\partial loss}{\partial Output[L-1]} = \frac{1}{N}(Output[L-1] - Y)$$

- 输出层的激活函数的选择

  由于本问题中是计算属于幽默文本的概率，且其取值范围为 [0,4] 之间，这提示我们利用 sigmod 函数，只要将 sigmod 函数的输出 $*4$ 即可。另一种方式是在输出层使用 relu 函数，由于 relu 函数的取值范围为 >=0，符合概率的需要满足的条件。

(ii) 随机森林

随机森林的回归与分类大体一致，就是在计算叶子节点的值和选择最优分裂特征的方法上有所不同。

随机森林的回归用所有样本的均值作为叶子节点取值。

```
def cal_leaf_value(self, targets):
    return targets.mean()
```

并且使用平方误差作为指标选择最优分裂点。

```
def cal_r2(self, left_targets, right_targets):
```

```
        gain = 0
        mean = left_targets.mean()
        for temp in left_targets:
            gain += (temp - mean) ** 2
        mean = right_targets.mean()
        for temp in right_targets:
            gain += (temp - mean) ** 2
        return gain
```

(iii) 基于 KNN 的集成学习

图 14: 基于 KNN 的集成学习回归流程图

回归的代码与分类的代码类似，主要不同体现在以下几个方面

1. 单个 KNN 子模型的评价标准

   在分类中，一个 KNN 子模型的预测结果是 K 个邻居的类别众数，评价一个 KNN 子模型的标准是其分类的准确率；在回归模型中，一个 KNN 预测的结果是 K 个邻居的加权求和，其评价标准为预测结果和真实结果的均方误差

```
    def _build_single_KNN(self,SampleTrainDataset,SampleTrainRating,
        ↪ SampleTargetDataset,SampleTragetRating,dis_type):
        k_to_error = {}#
        for k in range(3, 80):
            predict_res = []
            for line in SampleTargetDataset:
                dis_list = []
                for ID,record in enumerate(SampleTrainDataset):
```

```
                dis = self.ComputeDis(line,record,dis_type)
                l = []
                l.append(dis)
                l.append(SampleTrainRating[ID])#训练集中编号为ID的文档对应的
                    ↪ 标签
                dis_list.append(l)
            dis_list = sorted(dis_list,key=(lambda x:x[0]))#按距离升序排序
            normalized = 0
            predict_rating = 0
            for i in range(k):
                normalized += 1/dis_list[i][0]
                predict_rating += dis_list[i][1]/dis_list[i][0]
            predict_res.append(predict_rating/normalized)
        error = np.power(np.sum(np.square(np.array(predict_res)-np.array(
            ↪ SampleTragetRating)))/len(predict_res),0.5)
        k_to_error[k] = error

    return min(k_to_error,key=k_to_error.get)
```

2. 在集成多个 KNN 子回归模型时，使用的是对所有结果取平均的方式

```
def predict(self,train_dataset,train_rating,test_dataset):
    res = np.zeros((1,len(test_dataset)))
    for i in range(len(self.KSet)):#第i个knn子模型
        k = self.KSet[i]
        dis_type = np.array(self.dis_type_Set[i])
        ColChoice = np.array(self.col_choice_Set[i])
        RowChoice = np.array(self.row_choice_Set[i])

        SampleTrainDataset = np.array(train_dataset)[RowChoice,:]#选择
        SampleTrainDataset = np.array(SampleTrainDataset)[:,ColChoice]
        SampleTrainRating = np.array(train_rating)[RowChoice]
        SampleTestDataset = np.array(test_dataset)[:,ColChoice]
        temp = []
        for line in SampleTestDataset:
            dis_list = []
            for ID,record in enumerate(SampleTrainDataset):
                dis = self.ComputeDis(line,record,dis_type)
                l = []
                l.append(dis)
                l.append(SampleTrainRating[ID])#训练集中编号为ID的文档对应的
                    ↪ 分数
                dis_list.append(l)
            dis_list = sorted(dis_list,key=(lambda x:x[0]))#按距离升序排序
            normalized = 0#归一化,距离越近,占的比重越大
            predict_rating = 0
```

```
            for i in range(k):
                normalized += 1/dis_list[i][0]
                predict_rating += dis_list[i][1]/dis_list[i][0]
            temp.append(predict_rating/normalized)
        res = res + np.array(temp)
    res = res / len(self.KSet)#集成所有knn子模型的结果取平均
    return res
```

五、实验结果与分析

在最后的 Private Leaderboard 中，我们在分类问题上使用 TFIDF 和朴素贝叶斯得到了最好的结果，准确率在测试集达到 0.8500。在回归问题上使用 Doc2Vec 和随机森林得到了最好的结果，均方根误差达到 0.55854。十分遗憾的是，回归问题其实我们在 Private Leaderboard 最好的结果均方根误差达到 0.54676。但其在 Public Leaderboard 中结果不是那么好，所以并没有最为最后 Private Leaderboard 的结果。在本地测试中，我们将数据集按 7：3 划分为训练集和验证集，进行各项实验结果分析。

1. 分类结果

(i) 朴素贝叶斯

TFIDF 向量表示

验证集准确率	测试集准确率	运行速度	过拟合	输出结果稳定性
0.845833	0.85	6.6783309s	没有过拟合	多次测试，结果稳定

Doc2Vec 向量表示

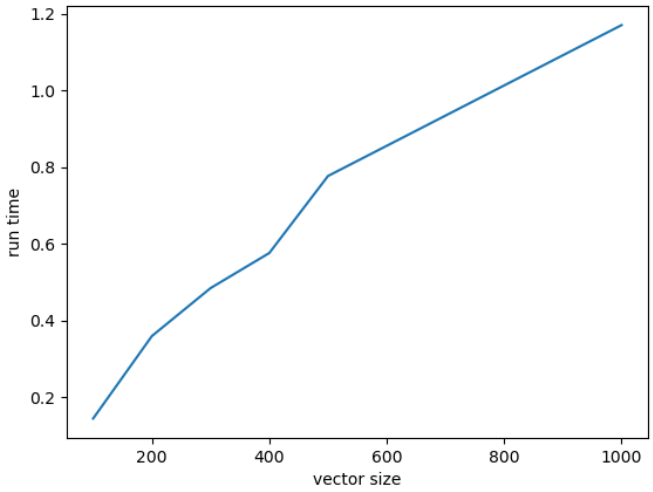


图 15: 朴素贝叶斯的运行时间与 Doc2Vec 向量大小的关系

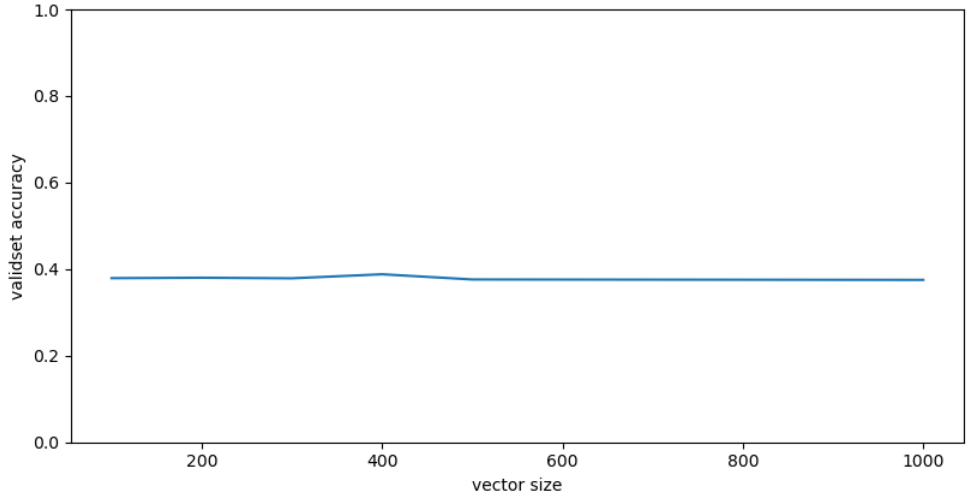


图 16: 朴素贝叶斯的准确率与 Doc2Vec 向量大小的关系

在输入向量一定的情况下，多次测试，发现在验证集上的准确率变化不大，说明在 doc2vec 向量表示下，朴素贝叶斯算法在稳定性方面较好，但是在准确率方面较差。

TFIDF 和 doc2vec 的对比分析

经过以上测试，发现朴素贝叶斯在 TFIDF 上的效果明显好于 doc2vec，原因也不难理解。在朴素贝叶斯中，假设 $p(text|class) = p(word_1|class)p(word_2|class)\dots p(word_n|class)$，计算 $p(word_n|class)$ 时，是利用单词 $word_n$ 在类别为 $class$ 的频率所代替，由于 TFIDF 得到的一个文档的向量表示中，每一个元素代表一个词的重要性程度，类似于一个词在一个文档中出现的次数，因此可以得到 $p(word_n|class)$ 的很好的近似，而 doc2vec 得到文档的向量表示中每一个元素的意义不明确，不能表示一个单词在一个文档中出现的频率，因此效果很差。

(ii) 神经网络

主要从不同的激活函数，是否添加 L2 正则化，是否使用 minibatch 以及不同的向量表示进行分析，选择迭代次数为 5000 次。

1. 不同的激活函数

   参数：神经网络为三层神经网络，由于是二分类问题，因此输出层选择 sigmoid 激活函数，改变隐藏层的激活函数，隐藏层节点数目为 50，词向量选择 doc2vec，长度为 100。

隐藏层激活函数	sigmoid	tanh	relu	leakyrelu
验证集准确率	0.62697	0.805000	0.803333	0.802917
过拟合	否	否	否	否
稳定性	不稳定	不稳定	不稳定	不稳定

当隐藏层为 sigmoid 激活函数时，增加迭代次数，发现验证集上的准确率提高，这说明了激活函数选择 sigmoid 时，神经网络的收敛速度会减慢。

2. 是否添加 L2 正则化

参数：神经网络结构："输入层-隐藏层-输出层"，隐藏层节点数目为 50，词向量为 doc2vec，长度为 100。隐藏层为 LeakyRelu 激活函数，输出层为 sigmoid 激活函数。

图17为正则化系数分别取 0,0.01,0.1,1,10 时，loss 函数的变化趋势，当 λ 取值很小时，loss 函数和不加 L2 正则化时的 loss 函数相差无几。当 λ 取值很大时，这时惩罚项在损失函数的占比增大，使得神经网络的中的权重参数减小甚至为 0，这样就不大会出现过拟合现象，由图中曲线得知，当 lambda 取值为 10 时，loss 函数在 3500 次迭代时已经收敛，减少了训练模型时的迭代次数。而且加了正则化之后，得到的模型在验证集上的准确率没有太明显的变化。

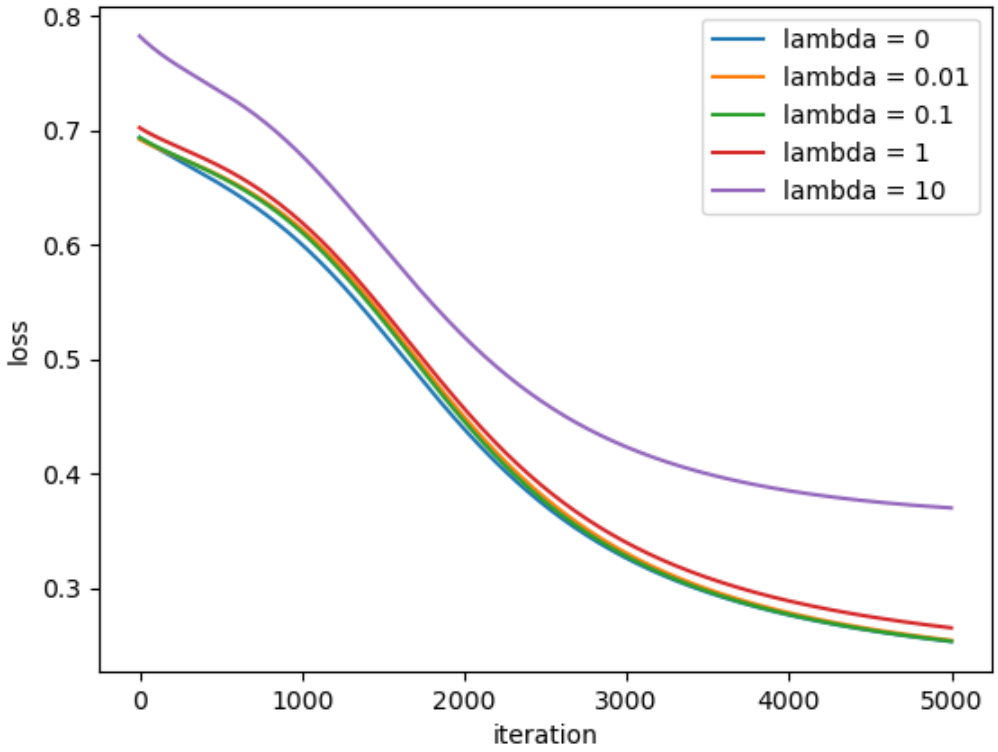


图 17: 不同正则化系数对 loss 函数的影响

lambda	验证集准确率
0	0.809583
0.01	0.804583
0.1	0.806667
1	0.805833
10	0.809583

3. minibatch 的 batchsize

参数：隐藏层选择 leakyrelu 激活函数，输出层选择 sigmoid 函数，网络结构为“输入层–隐藏层–输出层”，隐藏层节点数目为 50，词向量为 doc2vec，长度为 100。

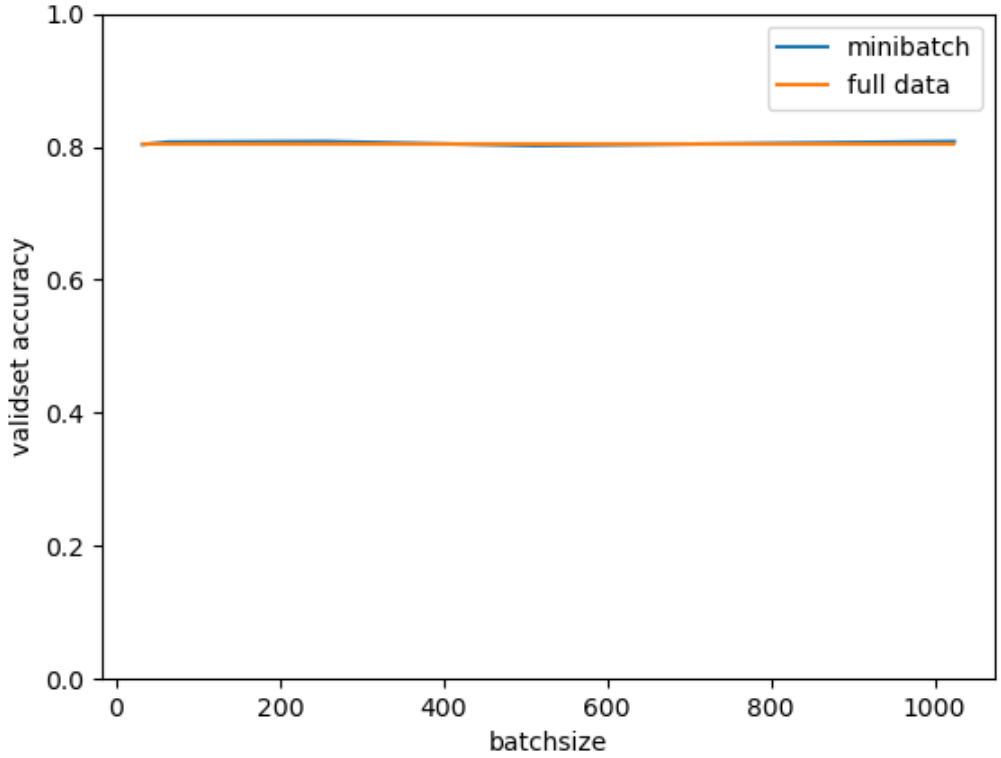


图 18: 不同 batchsize 对 loss 函数的影响

由图18可以得到，使用小批量梯度下降，在不同的 batchsize 下，其在验证集上的准确率和使用全数据集进行训练得到的效果相差无几。而且经过验证，在测试集上，小批量梯度下降也能达到使用全数据集进行训练的神经网络的正确率，有时甚至超过。

图19为使用小批量梯度下降和全数据进行训练时，损失函数的值的变化 (迭代在 5000 次时已经收敛)。可以看到使用批量梯度下降算法，loss 函数的值会一直减小直到模型收敛，

使用小批量梯度下降法，由于每一次更新只用到一部分数据，因此参数的更新对于另一些数据可能不是最优的，因此 loss 函数值震荡的比较明显。

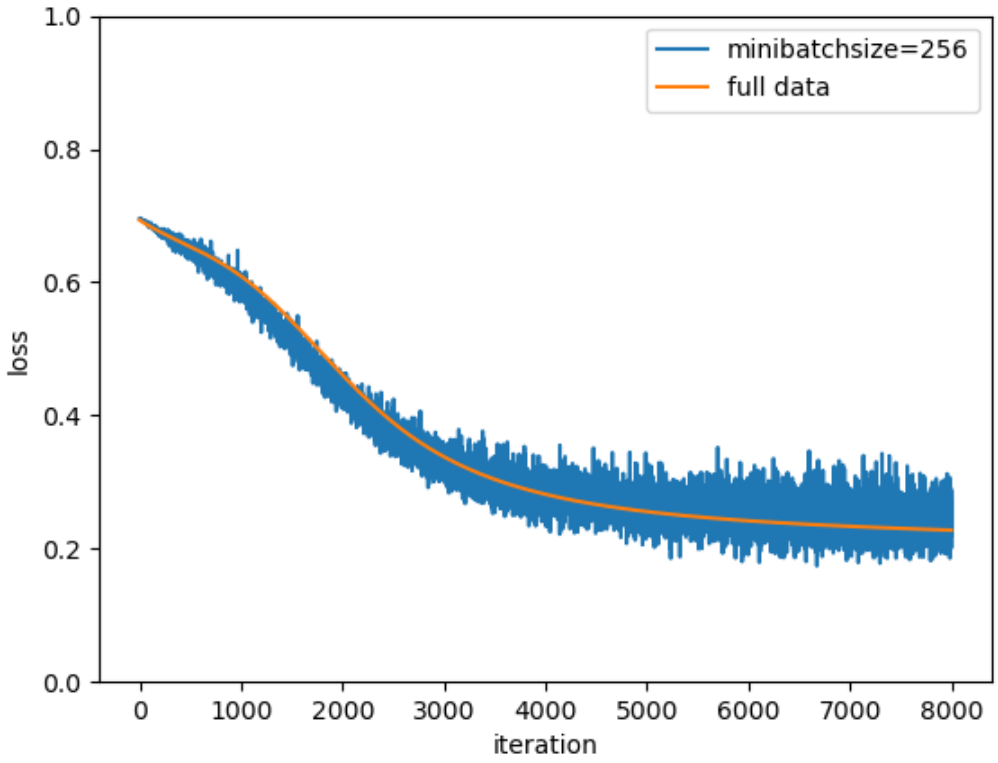


图 19: 小批量梯度下降和批量梯度下降 loss 函数值的变化

4. 不同的向量表示

参数：3 层神经网络，隐藏层选择 leakyrelu 激活函数，输出层选择 sigmoid 激活函数，未使用小批量梯度下降，隐藏层节点数目为 50，学习率为 0.01。

词向量类型	doc2vec	doc2vec	doc2vec	doc2vec	tfidf
词向量长度	100	500	1000	2000	8217
运行时间	2min	5min	6min	15min	35min
准确率	0.805417	0.800833	0.805417	0.807083	0.62
过拟合	否	否	否	否	欠拟合

由上表发现，doc2vec 由于其向量维度较小，因此神经网络的训练时间较短。不同的向量长度在神经网络上的表现类似。而 TFIDF 由于其向量的维度太大，神经网络需要学习的特征增加，需要更多的训练次数，才能得到一个较好的模型。我尝试将 doc2vec 的向量长度增加到 8000 维，再输入到神经网络中，发现其验证集准确率达到 0.8 这说明 tfidf 在神

经网络上的效果确实不好。原因在于 TFIDF 是一种稀疏的特征向量，特征过于稀疏会导致整个网络的收敛非常慢，每一个样本的学习只有极少数的权重会得到更新，会使得模型不收敛。

(iii) 随机森林

我们的随机森林可以调整的参数主要有树的数量`tree_num`、树的最大深度`max_depth`、最小分裂样本数`min_samples_split` 以及行采样的比例`row_samples_rate`。列采样`col_samples_rate` 通过取特征数量的平方根，没有改变。通过固定其它参数调整一个参数的方式，逐步比较分析。

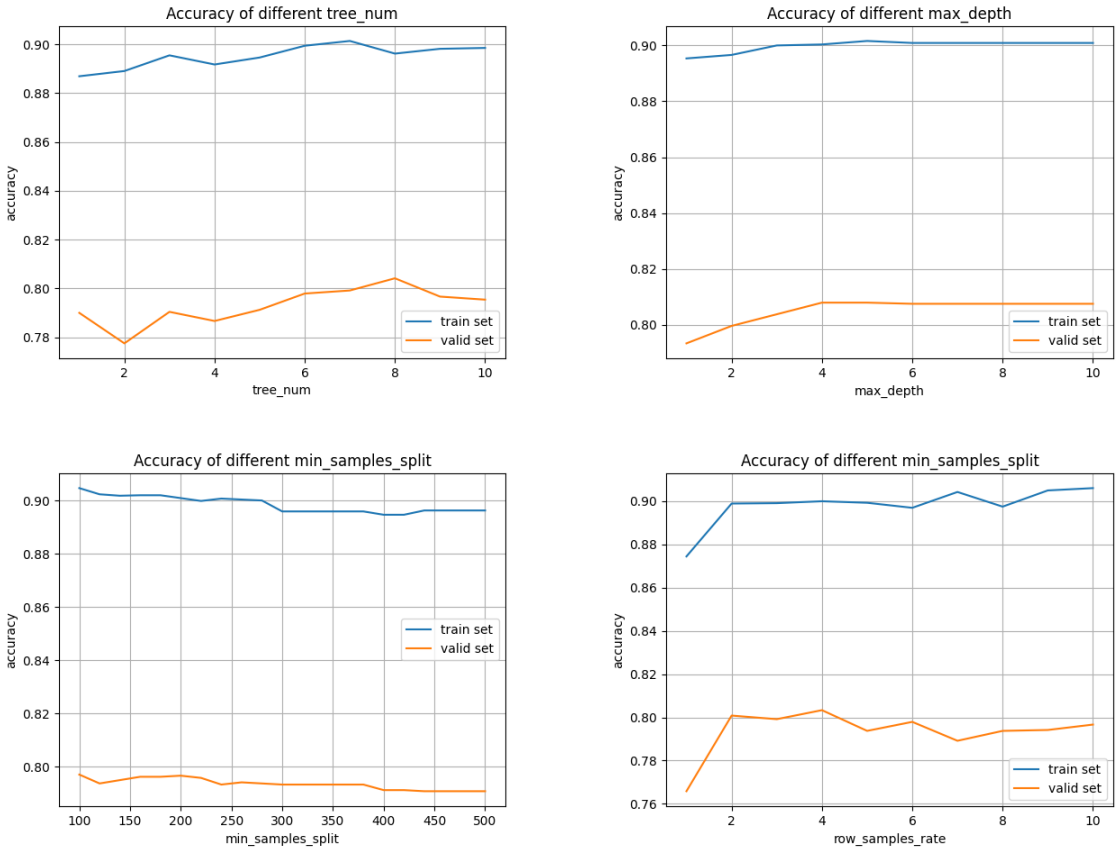


图 20: 随机森林的各项参数与准确率

最后发现得到的最好超参数如下，在验证集可以达到 0.8075 的准确率。

- `tree_num` = 8
- `max_depth` = 5
- `min_samples_split` = 200
- `row_samples_rate` = 0.4
- `col_samples_rate` = ”sqrt”

通过分析结果可以发现

- 随机森林作为一种集成学习的模型，在广泛的试验中都能得到相对不错的结果，但是在本次数据集中还是不如朴素贝叶斯的分类结果。
- 训练时间长，通常每次需要几分钟，在树的数量多、深度大、最小分裂样本数小的时候甚至达到几个小时。
- 不容易过拟合，训练集和验证集的准确率误差维持在10%左右，属于正常范围
- 输出结果稳定，因为我设置了一个random_state参数固定，可以保证复现相同的实验结果。如果不设置那么输出结果每次都随机，就不稳定了。

(iv) 集成 KNN

参数说明：使用词向量为 Doc2Vec，词向量维度为 100。

KNN 子模型距离度量方式选择为闵可夫斯基距离

模型数量为 1 时: 选择欧式距离 (曼哈顿距离结果类似)

模型数量为 3 时，选择 p=1，p=2，p=3 的闵可夫斯基距离，以此类推。

子模型数量	1	3	5	7
验证集准确率	0.81	0.812916	0.81417	0.81417
运行时间	数个小时	数个小时	数个小时	数个小时
过拟合	否	否	否	否
稳定性	稳定	稳定	稳定	稳定

下面是使用基于欧氏距离的单个 KNN 在全部数据上进行训练得到的不同的 k 和验证集上准确率误差的关系。

在 k 为 28 时，验证集最高准确率为 81.625%。可见和传统 KNN 相比，集成 KNN 的效果没有取得极大的进步，在文献 [6] 中其提出了一种基于邻近成分分析 (NCA) 的距离学习集成 KNN 分类器，取得了不错的结果，但是由于时间有限，未能实现。

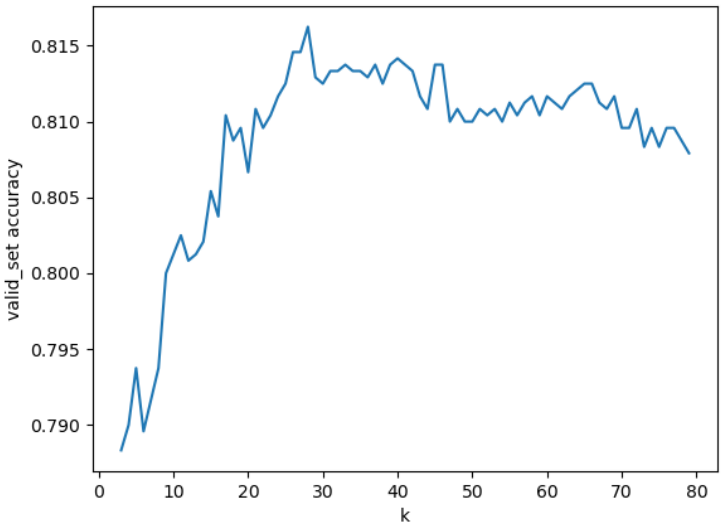


图 21: 基于欧式距离的传统 KNN 在训练集上的准确率

2. 回归结果

(i) 神经网络

1. 是否使用 minibatch

   以下神经网络的结构为: “输入层-隐藏层-隐藏层-输出层”。隐藏层和输出层均选择 relu 激活函数，隐藏层节点数目分别为 50,7, 学习率为 0.01。

   - 不使用 minibatch

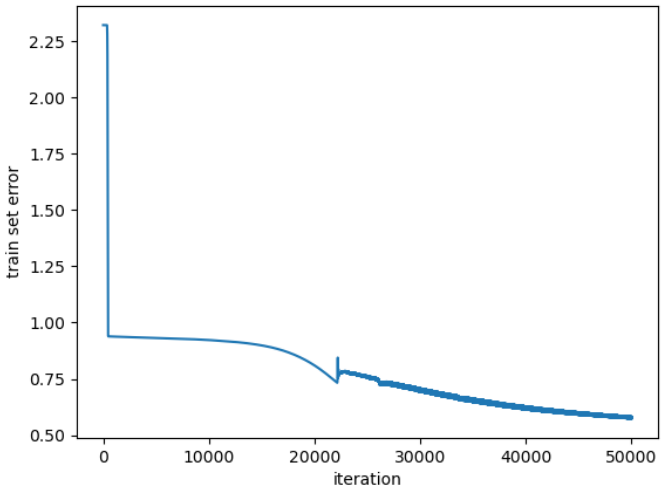


图 22: 不使用 minibatch 时训练集误差随迭代次数的变化

图22为不使用 minibatch 得到的训练集上的误差随迭代次数的变化规律，发现迭代 50000 次训练集上的误差在 0.58 左右，而在验证集上的误差竟达到了 0.63。

- 使用 minibatch

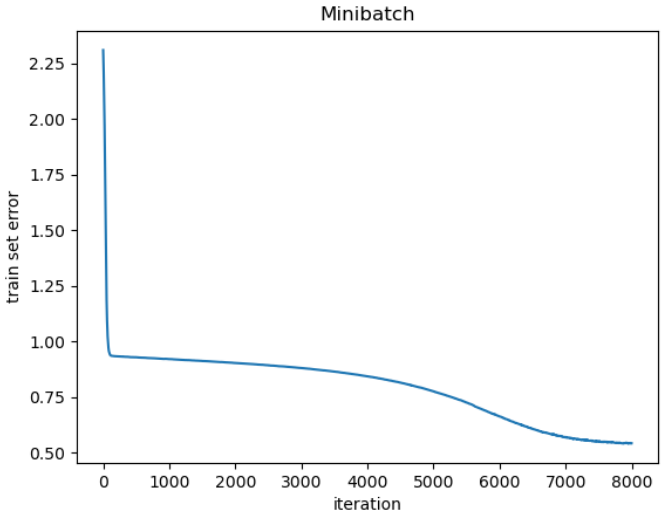


图 23: 使用 minibatch 时训练集误差随迭代次数的变化

由图23可知，使用 minibatch, 在迭代次数仅为 8000 次时，在训练集上的误差已经达到了比较好的取值 0.55 左右，在验证集上的误差为 0.58，因此，下面的神经网络均使用 minibatch。

2. 不同的神经网络层数

参数：隐藏层和输出层均使用 relu 激活函数。四层神经网络隐藏层节点数目分别为 50,7。三层神经网络隐藏层节点数目为 50。

下图为三层神经网络和四层神经网络在训练集上的 RMSE，三层神经网络的效果较差，在训练集和验证集上的 RMSE 均为 0.8 左右，而 minibatch 在验证集和训练集上的 RMSE 均能达到 0.57 左右。

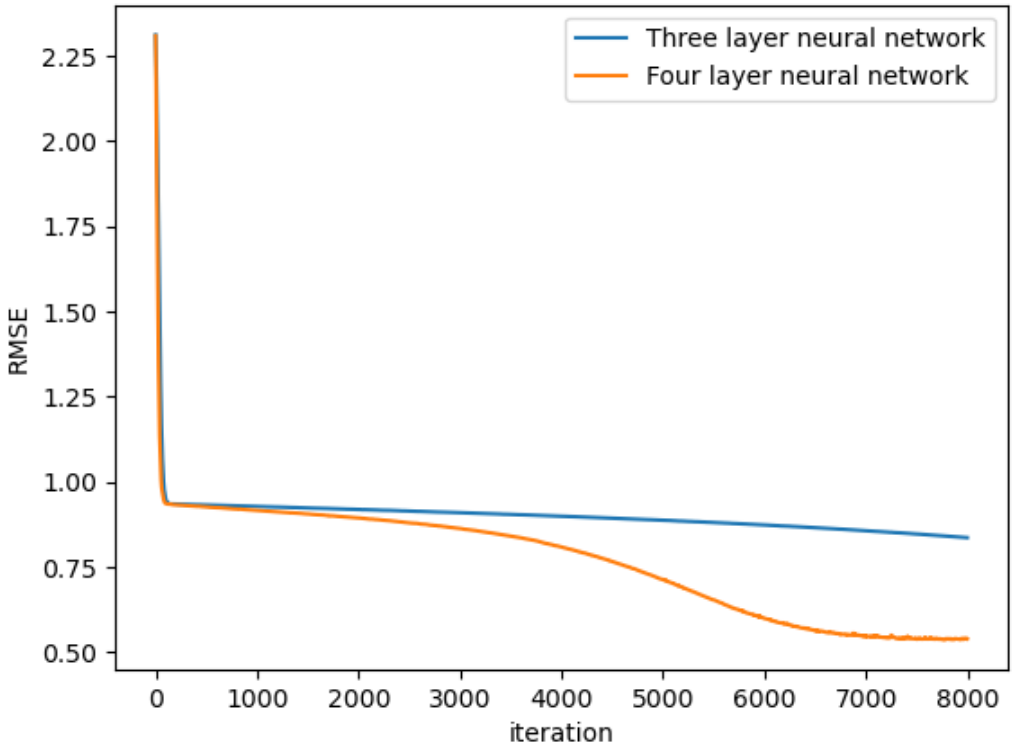


图 24: 三层和四层神经网络在训练集上的 RMSE

	三层神经网络	四层神经网络
验证集准确率	0.88323531766153	0.5657514686200872

3. 不同的激活函数

对于回归问题来说，神经网络的输出层的激活函数的选择至关重要，下面的测试在隐藏层均为 relu 激活函数的神经网络上进行。由于回归的输出范围为 [0,4]，因此输出层选择 sigmoid 激活函数或者 relu 激活函数。当选择 sigmoid 激活函数时，要将 sigmoid 激活函数的输出范围由 [0,1] 映射到 [0,4]。

输出层激活函数	sigmoid	relu
验证集 RMSE	0.571378	0.57074
过拟合	否	否

两种激活函数在训练集上的 RMSE 随着迭代的变化如下，使用 sigmoid 函数作为输出层激活函数几乎能够迅速收敛，在验证集上的误差和使用 relu 函数类似。

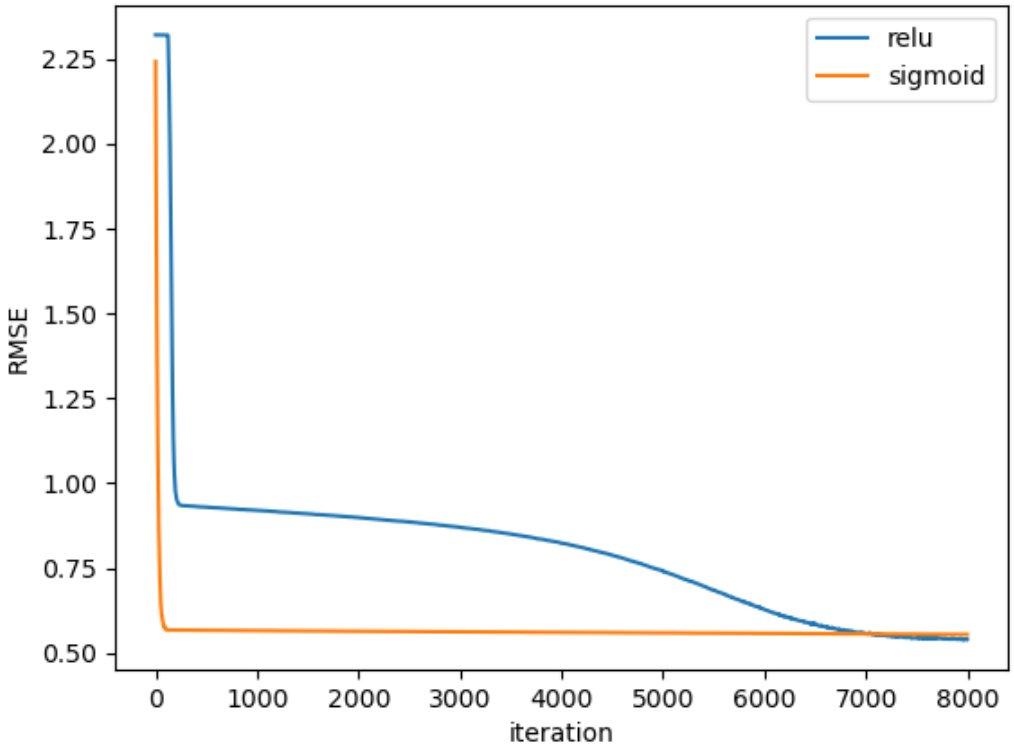


图 25: sigmoid 和 relu 在训练集上的 RMSE 随迭代的变化

4. 神经网络的稳定性

   当隐藏层和输出层选择的激活函数均为 relu 激活函数时，输出结果会出现不稳定的现象，下面分别为两次运行同一个程序在训练集上的误差，其中上面一幅图对应的验证集上的 RMSE 为 2.3527。第二幅图对应的验证集上的 RMSE 为 0.57807。可见，relu 存在神经元坏死现象，满足一定的条件时，这种现象就会出现。由于 Relu 的导数在 x<0 时取值为 0，因此会出现某些神经元的梯度永远为 0 的情况，这些神经元可能永远不会被激活，神经网络学习不到数据的特征。

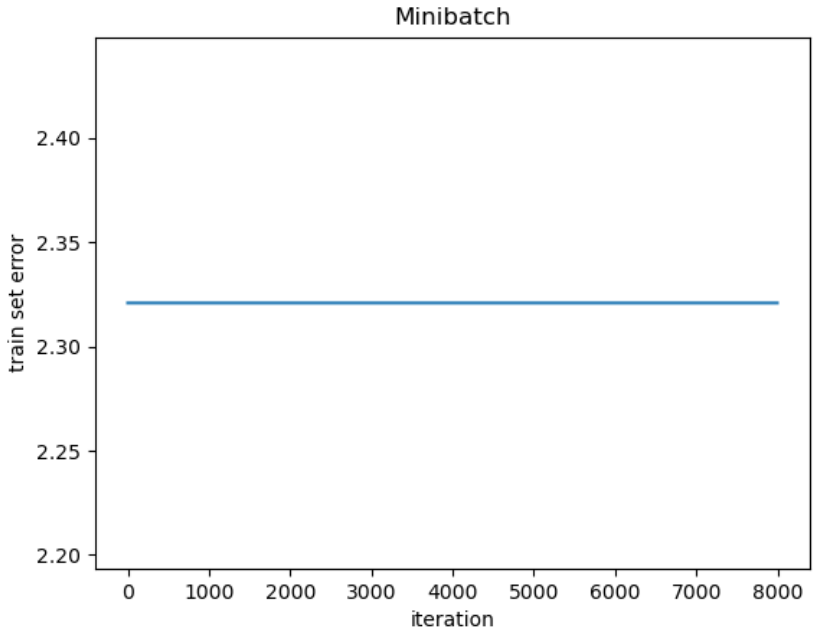


图 26: 发生 relu dead 现象

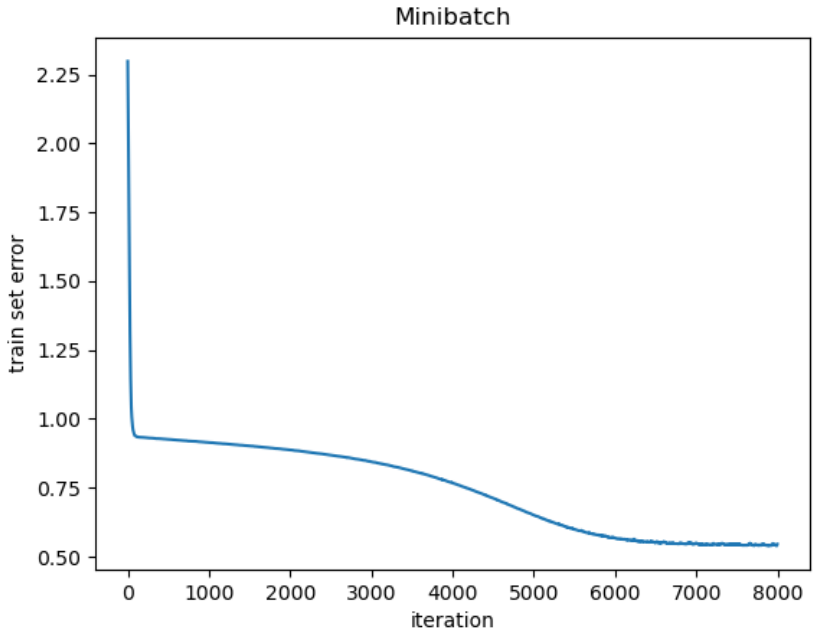


图 27: relu 正常

(ii) 随机森林

随机森林的回归与分类大体一致，只是将评测指标改为均方根误差 RMSE。通过固定其它参数调整一个参数的方式，逐步比较分析。

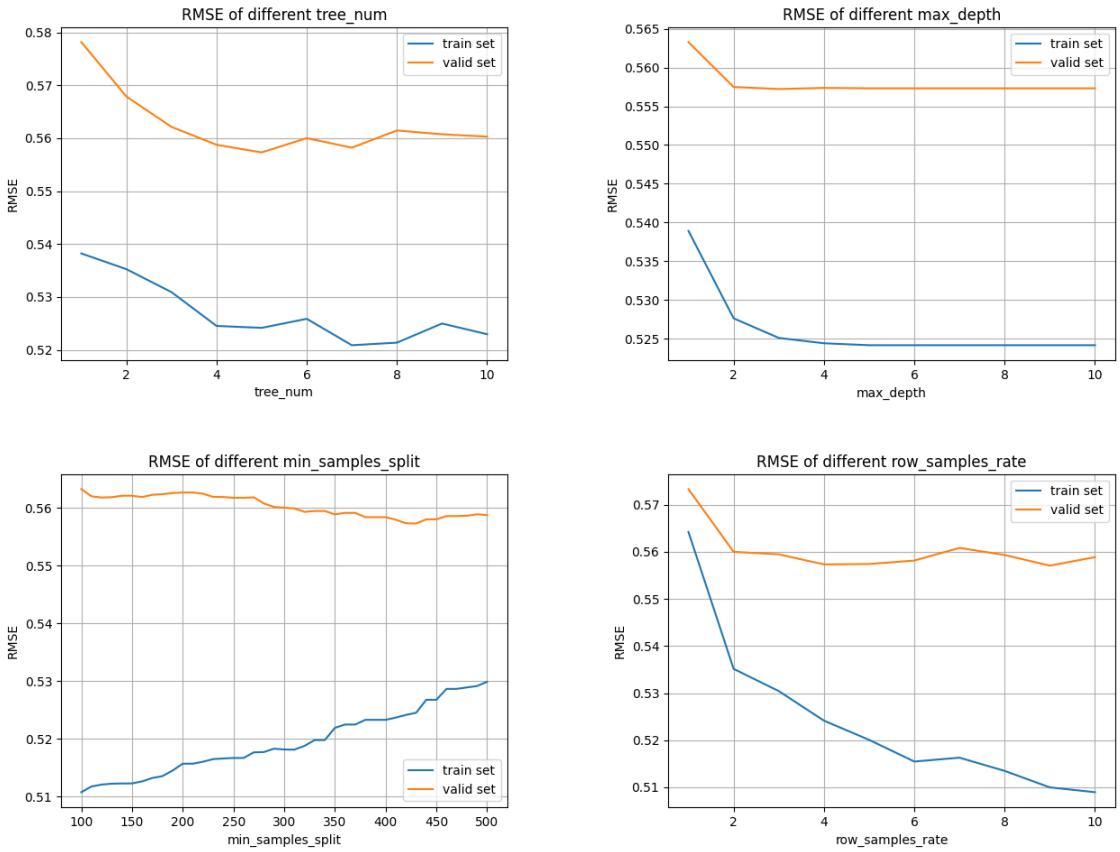


图 28: 随机森林的各项参数与 RMSE

最后发现得到的最好超参数如下，在验证集的 RMSE 可以达到 0.5573。

- `tree_num` = 5
- `max_depth` = 5
- `min_samples_split` = 420
- `row_samples_rate` = 0.4
- `col_samples_rate` = "sqrt"

通过分析结果可以发现

- 随机森林的回归也能得到相对不错的结果，但因为它并不能给出一个连续的输出，所以不如分类。
- 训练时间长。

- 不容易过拟合，除非树的深度特别大、最小分裂样本数特别小。
- 输出结果稳定，设置`random_state`参数固定，可以保证复现相同的实验结果。

(iii) 集成 KNN

参数说明：使用词向量为 Doc2Vec，词向量维度为 100。

KNN 子模型距离度量方式选择为闵可夫斯基距离

模型数量为 1 时: 选择欧式距离 (曼哈顿距离结果类似)

模型数量为 2 时，选择 p=1,p=2 的闵可夫斯基距离

模型数量为 3 时，选择 p=1,p=2,p=3 的闵可夫斯基距离，以此类推。

子模型数量	1	2	3
验证集 RMSE	0.56544	0.5640	0.56399
运行时间	数个小时	数个小时	数个小时
过拟合	否	否	否
稳定性	稳定	稳定	稳定

子模型数量为 1

最优 k 值 57

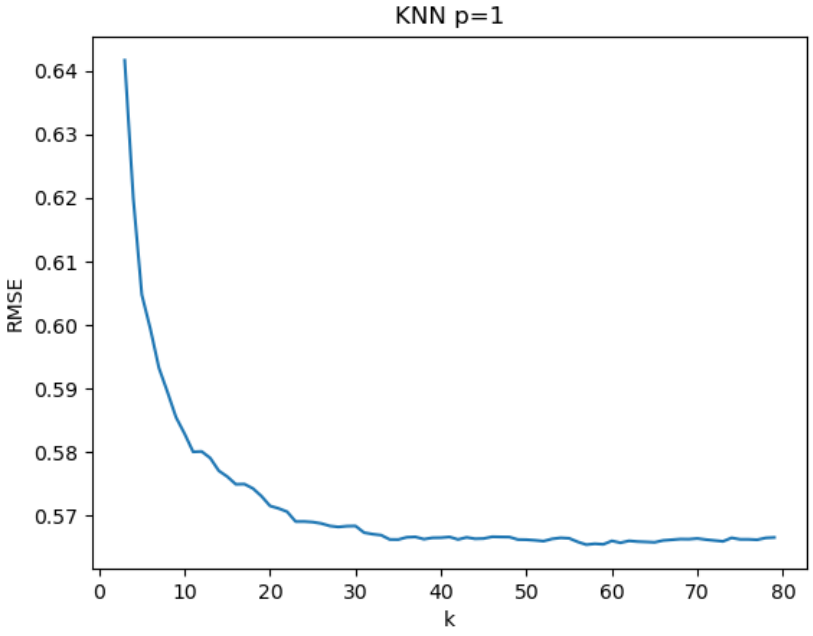


图 29: p=1 闵可夫斯基距离的子 KNN 模型在训练集上的误差

子模型数量为 2

对应最优 k 值为 57,31

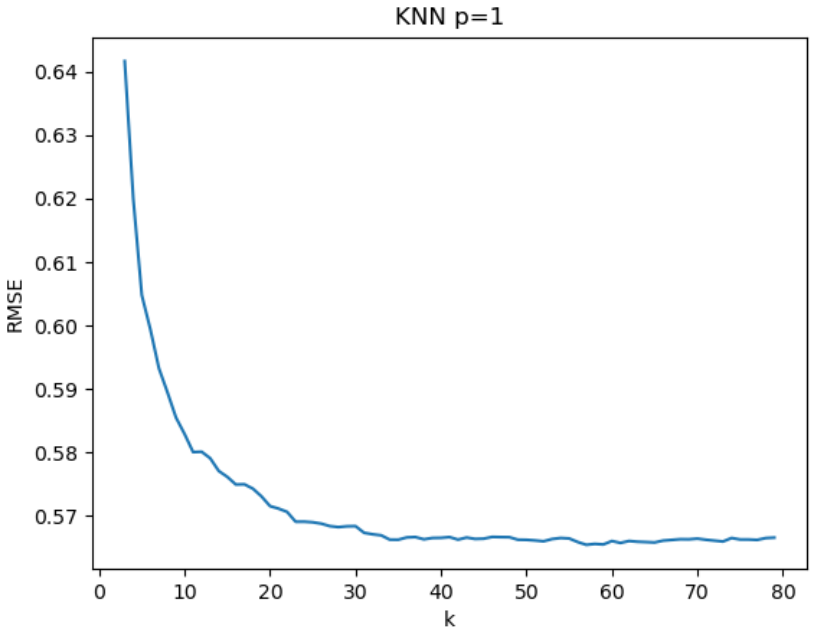


图 30: p=1 闵可夫斯基距离的子 KNN 模型在训练集上的误差

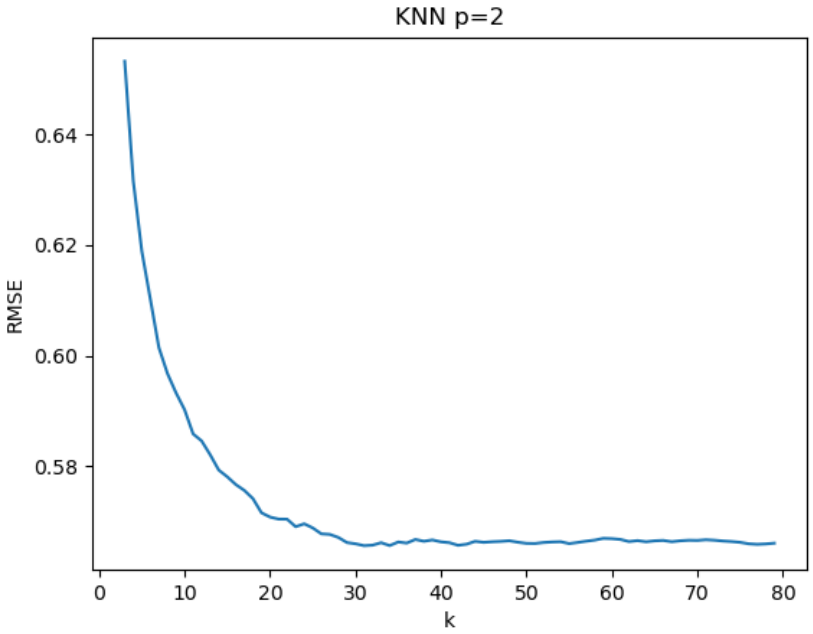


图 31: p=2 闵可夫斯基距离的子 KNN 模型在训练集上的误差

子模型数量为 3

3 个子模型对应的最优 k 值分别为 57,50,33

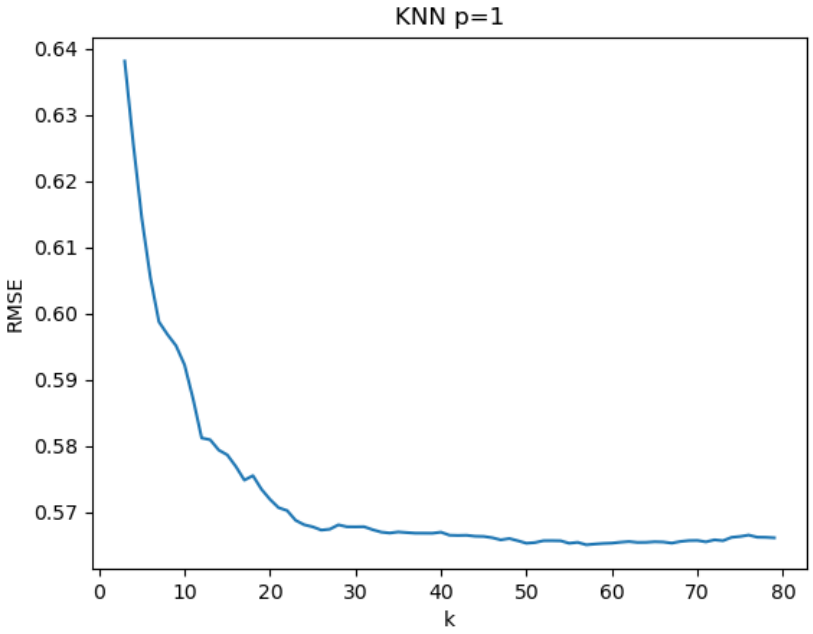


图 32: p=1 闵可夫斯基距离的子 KNN 模型在训练集上的误差

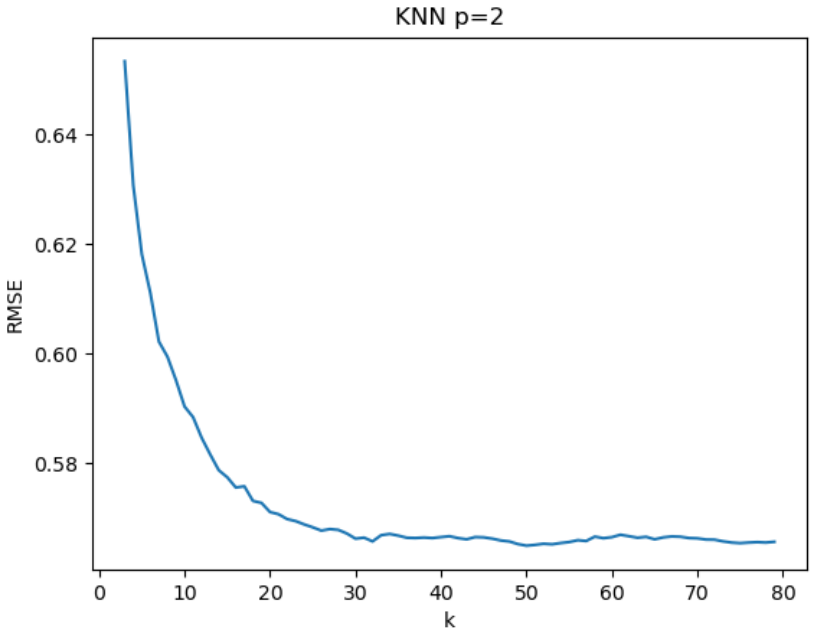


图 33: p=2 闵可夫斯基距离的子 KNN 模型在训练集上的误差

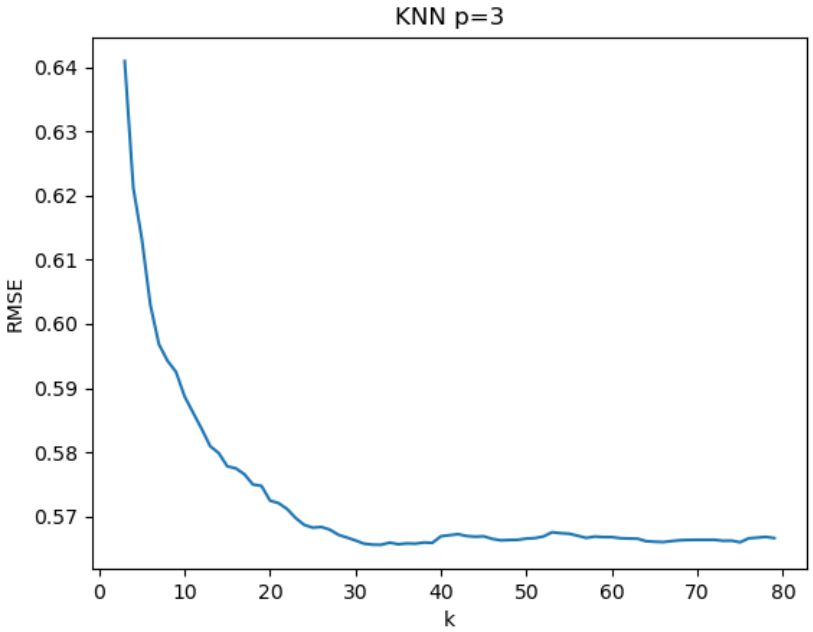


图 34: p=3 闵可夫斯基距离的子 KNN 模型在训练集上的误差

3. 所有算法的结果比较与分析

以下所有算法都以验证集上的最佳结果进行比较。

分类

方法	运行速度	抗过拟合能力	输出结果稳定性	准确率
朴素贝叶斯	7s	强	稳定	0.8458
神经网络	15min	弱	不稳定	0.8070
集成 KNN	数小时	强	稳定	0.8147
随机森林	271.6510s	强	稳定	0.8075

回归

方法	运行速度	抗过拟合能力	输出结果稳定性	RMSE
神经网络	2-3min	弱	不稳定	0.57074
集成 KNN	数小时	强	稳定	0.56399
随机森林	69.3416s	强	稳定	0.5573

抗过拟合能力和输出结果稳定性分析

- 朴素贝叶斯

  朴素贝叶斯具有较强的抗过拟合能力，朴素贝叶斯的一个较强的假设为特征之间是相互

独立的，这个假设在现实中几乎是不可能存在的，也使得其无法对训练数据进行过拟合。由于该模型中没有什么随机的参数，因此它的输出结果是稳定的，在非常少量的训练数据下表现得较好。

- 神经网络

  对于神经网络来说，当网络结构过于复杂，单层节点数目较多时很容易造成过拟合现象。此外当样本中数据噪声较大时，模型过分记住了噪声得特征，而忽略了真实的输入和输出之间的关系，使得在训练集上表现良好，而在测试集上表现很差。在本次实验中，也遇到了神经网络出现过拟合的现象，由于设置的最大迭代次数较大，使得模型过分学习训练集上的特征，导致模型泛化能力降低。

  为缓解神经网络的过拟合现象，有以下几种方法可供选择

  1. 增加数据集规模
  2. 使模型简单化
  3. 正则化

  在输出稳定性方面，由于神经网络的参数初始化是随机初始化，使得模型的稳定性较差。在相同的网络结构下，某些初始化会导致模型不收敛，而另一些初始化会使得模型收敛得很快。在实验中，当激活函数均为 relu 时，会出现神经元坏死的现象。

- 随机森林

  随机森林不容易发生过拟合，因为在建立每一颗树的过程中，需要做到随机采样和完全分裂。每一棵树的输入均不是全部样本，因此不容易出现过拟合问题。

  树越多，随机森林的表现越稳定，理论上可以证明，当随机森林产生的树的数目趋近于无穷时，其训练误差和测试误差是收敛到一起的。

- 集成 KNN

  对于单个 KNN，当选择较小的 K 值时，只有与测试实例相近的训练数据才会对预测结果起作用，较小的 k 值使模型变得更加复杂，容易发生过拟合。当选择较大的 k 值时，与输入实例较远的训练数据也会对预测结果产生影响，使预测错误，K 值得增大使得模型变得简单，不容易发生过拟合，但是容易发生欠拟合。

  由于在构建集成 KNN 时，使用了随机选择特征和随机选择训练数据和 bagging，因此集成 KNN 不容易发生过拟合问题。而且当构成集成 KNN 的 KNN 子模型越多时，其稳定性越好。

运行时间和准确率/RMSE 分析

在分类问题上，朴素贝叶斯分类器的分类效果明显好于其他分类模型，这也体现了它在简单数据集上的优势，另一个原因在于，相对于其他模型朴素贝叶斯分类器需要调节的参数较少，

因此能够得到较好的结果；而其他模型需要调节的参数较多，而且我们对调节参数没有什么经验，因此导致其他模型的表现要逊色不少。运行时间方面，朴素贝叶斯需要计算的数据较少，模型较简单，因此时间最短。而集成 KNN 需要训练出几个 KNN 子模型，然后基于这几个 KNN 子模型进行预测结果，需要的计算量最大，也最慢。

在回归问题上，在验证集上，随机森林的表现较好，但是实际在测试集上时，还是神经网络更胜一筹。但是，这与两个模型的优劣并没有什么关系，关键还在于调参。运行时间方面，由于 KNN 本身的缺陷，其运行时间最长。

4. 算法优缺点表格

方法	优点	缺点
朴素贝叶斯	所需估计参数少。 对缺失数据不敏感。 有坚实的数学基础以及稳定的分类效率。	需要假设属性之间相互独立 需要知道先验概率。 分类决策存在错误率。
KNN	核心思路简单。 训练时间复杂度 $O(n)$。 对数据没有假设，对离群值不敏感。	计算量太大。 样本分类不均会产生误判。 需要大量内存，输出可解释性不强。
神经网络	分类回归效果良好。 并行处理能力强，分布式存储学习能力强。 鲁棒性较强，不易受噪声影响。	需要大量参数。 结果难以解释。 训练时间长。
随机森林	分类回归效果良好。 泛化能力强，不容易过拟合。 部分特征遗失仍可维持准确度。	对小数据或低维数据效果不太好。 在噪声较大问题上会过拟合 训练时间长。

六、总结

人工智能的期中项目终于告一段落了，这次实验让我们充分掌握了文本处理和降维的各种方法，以及机器学习分类与回归的各种算法，自己实操。虽然花费了我们大量的时间，但也收获了很多。

在这次实验中我们尝试了许多方法，也做了一定的创新，不得不感叹人工智能 “浩如烟海” 的知识宝库有许多许多的内容值得我们探究。在整个实验过程中，常常写代码写了一天，兴致勃勃地交到 Kaggle 上去，发现结果太糟糕了，内心直接崩溃。于是第二天疯狂调参，又想尝试别的方法，争取在 Leaderboard 上击败别人。并且在运行自己的代码时经常要跑几十分钟甚至几个小时，得不到好的结果，然后又得思考是模型本身的问题，还是参数没调好，亦或是自己写了一堆 bug。虽然现在人人都在炼丹，人人都在调包调参，但是想真正当好一个 “炼丹师” 和 “调参侠” 也不是一件容易的事情。Private Leaderboard 公布的时刻是最刺激的，可谓是 “几家欢喜几家愁”，有的组排名狂升，有的组排名狂掉。我们的分类结果幸运地升到并列第一，可惜

回归最好的结果并没有作为最后 Private Leaderboard 的结果。

总而言之，这次实验也还是十分有趣的，不过也好在期中我们有一周休息的时间才能尝试那么多的方法。期待下一次实验!

参考文献

[1] Doc2Vec(v4.0.0).https://radimrehurek.com/gensim/apiref.html,2021.

[2] 随机森林.https://blog.csdn.net/edogawachia/article/details/79357844,2020.

[3] Feature Selection.https://scikit-learn.org/stable/modules/classes.html#module-sklearn.feature_extraction.text,2021.

[4] 神经网络中的参数初始化问题.https://blog.csdn.net/weixin_44441131/article/details/106547116,2020.

[5] 带你理解朴素贝叶斯分类算法.https://zhuanlan.zhihu.com/p/26262151,2017.

[6] 于飞. 基于距离学习的集成 KNN 分类器的研究 [D]. 大连理工大学,2009.