中山大学数据科学与计算机学院

计算机科学与技术专业-人工智能

本科生实验报告

（2018-2019学年秋季学期）

课程名称：**Artificial Intelligence**

|  |  |  |  |
| --- | --- | --- | --- |
| 教学班级 | 计科2班 | 专业（方向） | 计算机科学与技术 |
| 学号 | 16337341 | 姓名 | 朱志儒 |

## 实验题目

**Project 二元分类**

## 实验内容

* **算法原理**

1. **KNN**

KNN算法简单而直观，给定一个训练数据集，对新的输入实例，在训练数据集中找到与该实例最邻近的k个实例，这个k个实例的多数属于某类，就把输入实例分为这个类。

KNN算法的输入是训练集，其中，为实例的特征向量，为实例的类别。

输出是实例x所属的类别y。

具体内容是根据给定的距离向量，在训练集T中找出与x最邻近的k个点，涵盖这k个点的x的邻域记为，在中根据多数表决的方式决定x的类别y：

其中，I为指示函数，当时I为1，否则I为0。

1. **Naïve Bayes**

Naïve Bayes，即朴素贝叶斯分类器，有坚实的理论基础——贝叶斯定理。贝叶斯定理基于条件概率，条件概率P(A|B)表示在事件B已经发生的前提下，事件A发生的概率，即，贝叶斯定理通过P(A|B)来求P(B|A)：，其中P(A)由全概率公式可分解为：。

假设给定训练数据集(X, Y)，其中每个样本x都包括n维特征，即，类标记集合含有k中类别，即。对于测试集样本x，为判断其类别，从概率的角度来看，就是x属于k个类别中哪个概率最大，问题就变成找出中最大的项，即求出后验概率最大的输出：。由贝叶斯定理可知：。

分子中的是先验概率，可直接根据训练集数据计算得出，而条件概率有指数级数量的参数，假设第j维特征可取值有个，，y可取值有K个，那么参数个数为。

朴素贝叶斯对条件概率作了条件独立性假设，即各个维度的特征相互独立，在这个假设下，条件概率：，如此，参数规模降为，那么，于是朴素贝叶斯分类器可表示为

在计算先验概率和条件概率时，需要做平滑处理：

其中，N为总样本个数，k为总类别个数，是类别为的样本个数，a为平滑值，n为特征的维数，是类别为的样本中，第i维特征的值是的样本个数。

在实际实现的过程中，考虑到中分母都为，所以在比较时可以忽略分母而只考虑分子。考虑到大量的概率浮点数乘法运算，为避免floating-point underflow问题，将乘法转化为取log再相加的运算：

1. **Logistic Regression**

逻辑回归与线性回归的原理相似，但它实际上是一种分类方法。二项逻辑回归模型的条件分布：

其中，是输入，是输出，和是参数，w称为权值向量，b称为偏置，为w和x的内积。

对于给定的输入实例x，由上述两式可以求出和，比较这两个条件概率的大小，将实例x分到概率值较大的一类。

对输入x进行分类的线性函数，其值域为实数域，通过逻辑回归模型可将线性函数转化为概率：。此时，t的值越接近正无穷，概率值就越接近1；t的值越接近负无穷，概率值就越接近0。

在逻辑回归模型学习时，可应用极大似然估计法估计模型参数，从而得到逻辑回归模型。

设：

似然函数为：

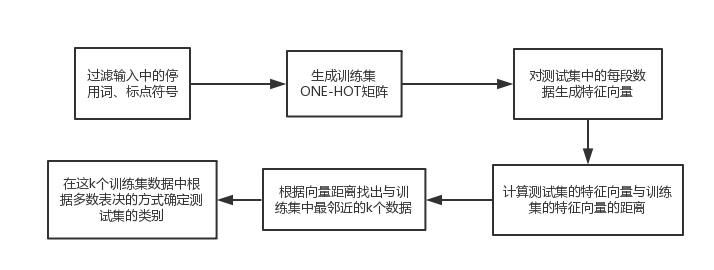
对数似然函数为：

对L(w)求极大值可得到w的估计值，问题就转变成以对数似然函数为目标函数的最优化的问题。令，可使用梯度下降法求解C(w)的极小值。由梯度下降法可知w的更新过程：

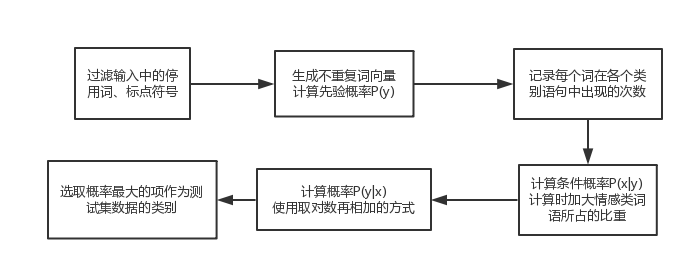
不断迭代直至w收敛，这样就可以得到w的估计值。

* **流程图&伪代码**

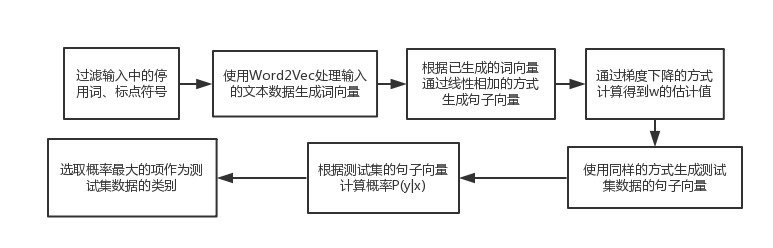
1. **KNN**



1. **Naïve Bayes**



1. **Logistic Regression**



* **关键代码**

**过滤输入中的停用词、标点符号和数字：**

def filter\_data(filename):

all\_data = []

#停用词表

common\_words = ['have', 'had', 'has', 'are', 'was', 'were', 'the', 'this', 'that', 'and', 'etc', 'they', 'them', 'their', 'theirs', 'our', 'ours', 'you', 'your', 'yours', 'its', 'she', 'her', 'hers', 'him', 'his', 'it´s', 'thats', 'haven', 'there', 'theres', 'maybe', 'hey', 'anyway', 'doesn', 'think', 'really', 'been']

#控制字符表

ctlword = '\x97\x91\x96\x08\x83\x8e\x9e\x84'

with open(filename, 'r', encoding='utf-8') as file:

for line in file:

line = line.strip().replace('<br />', ' ').replace("n't", " not")

#去除所有的标点符号、数字、控制字符

table = str.maketrans(string.punctuation + string.digits + ctlword,

' ' \* (len(string.punctuation) + len(string.digits) + len(ctlword)))

line = line.translate(table).split()

row = []

for item in line:

#去除长度小于2的词和停用词

if len(item) > 2 and item.lower() not in common\_words:

row.append(item.lower())

all\_data.append(row)

return all\_data

1. **KNN**

**生成不重复词向量和生成ONE-HOT矩阵：**

def \_\_init\_\_(self, train\_set, train\_set\_label):

self.train\_row\_num = len(train\_set)

self.train\_label = train\_set\_label

self.words\_vector = []

#生成不重复词向量

for row in train\_set:

self.words\_vector += row

self.words\_vector = list(set(self.words\_vector))

#生成ONE-HOT矩阵

self.xmatrix = np.zeros((self.train\_row\_num, len(self.words\_vector)))

for i in range(len(train\_set)):

for word in train\_set[i]:

self.xmatrix[i][self.words\_vector.index(word)] += 1

print('已处理', i, '行')

**根据训练集数据将测试集数据分类：**

def classification\_KNN(self, test\_set, k):

labels = []

for line in test\_set:

#生成测试集的特征向量

test\_vector = np.zeros(len(self.words\_vector))

for word in line:

if word in self.words\_vector:

test\_vector[self.words\_vector.index(word)] += 1

#计算测试集向量与训练集向量的距离

test\_matrix = np.abs((np.tile(test\_vector, (self.train\_row\_num, 1)) - self.xmatrix))

distance = np.sum(test\_matrix, axis=1).T.tolist()

maps = {}

for i in range(len(distance)):

if distance[i] not in maps.keys():

maps[distance[i]] = [self.train\_label[i]]

else:

maps[distance[i]].append(self.train\_label[i])

distance.sort()

count = 0

mood = []

for index in range(len(distance)):

i = distance[index]

if len(maps[i]) + count >= k:

mood += maps[i]

break

else:

mood += maps[i]

count += len(maps[i])

#根据多数表决的规则得到测试集的类别

top = Counter(mood).most\_common(1)[0][0]

labels.append(top)

print('已生成', len(labels), '个')

return labels

1. **Naïve Bayes**

**生成不重复词向量并计算先验概率P(y)**

def \_\_init\_\_(self, train\_set, train\_labels):

self.train\_set = train\_set

self.trian\_labels = train\_labels

self.pos\_words\_vector = {}

self.neg\_word\_vector = {}

# 生成不重复词向量

self.words\_vector = []

for word in train\_set:

self.words\_vector += word

self.words\_vector = list(set(self.words\_vector))

for word in self.words\_vector:

self.pos\_words\_vector[word] = 0

self.neg\_word\_vector[word] = 0

self.pos\_num = 0

for label in self.trian\_labels:

if label == 1:

self.pos\_num += 1

self.neg\_num = len(self.train\_set) - self.pos\_num

# 计算先验概率P(y)

self.neg\_prob = (self.neg\_num + 1) / (len(self.trian\_labels) + 2)

self.pos\_prob = (self.pos\_num + 1) / (len(self.trian\_labels) + 2)

**将测试集数据分类**

def classification(self, test\_set):

# 使用Word2vec获取情感类词语集合

model = word2vec.Word2Vec.load('all3000.model')

important = ['sweet', 'great', 'recommend', 'happy', 'like', 'fun', 'good', 'bad', 'hate', 'boring', 'sad', 'love', 'satirical', 'romantic', 'whimsical']

for i in range(20):

for item in model.most\_similar(positive=important, topn=10):

important.append(item[0])

important = list(set(important))

important.append('not')

print(len(important), important)

for i in range(len(self.train\_set)):

print('计算第', i, '行')

tmp = []

for word in self.train\_set[i]:

if word not in tmp:

if self.trian\_labels[i] == 1:

# 记录每个词在类别为1的语句中出现的次数

self.pos\_words\_vector[word] += 1

else:

# 记录每个词在类别为0的语句中出现的次数

self.neg\_word\_vector[word] += 1

tmp.append(tmp)

for key in self.words\_vector:

# 计算条件概率P(x|y)

if key not in important:

self.pos\_words\_vector[key] = (self.pos\_words\_vector[key] + 1) / (self.pos\_num + 2)

self.neg\_word\_vector[key] = (self.neg\_word\_vector[key] + 1) / (self.neg\_num + 2)

else:

# 加大情感类词语所占的比重

self.pos\_words\_vector[key] = (self.pos\_words\_vector[key] + 5) / (self.pos\_num + 2)

self.neg\_word\_vector[key] = (self.neg\_word\_vector[key] + 5) / (self.neg\_num + 2)

labels = []

for line in test\_set:

# 计算概率P(y|x)时使用取对数相加的方式

is\_pos = math.log(self.pos\_prob)

is\_neg = math.log(self.neg\_prob)

for word in line:

if word in self.words\_vector:

is\_pos += math.log(self.pos\_words\_vector[word])

is\_neg += math.log(self.neg\_word\_vector[word])

# 选取概率最大的项作为测试集数据的类别

if is\_pos > is\_neg:

labels.append(1)

else:

labels.append(0)

print('已预测', len(labels), '行')

return labels

1. **Logistic Regression**

**对训练集数据进行处理**

def \_\_init\_\_(self, train\_set, train\_labels, size, step):

self.train\_set = train\_set

self.train\_labels = train\_labels

self.step = step

self.weights = np.ones((size, 1))

# 使用Word2Vec生成词向量

self.model = word2vec.Word2Vec.load('train\_all\_5000.module')

self.word\_vector = list(self.model.wv.vocab.keys())

# 将每句文本中的词向量线性相加生成句子向量，组合训练集所有句子向量生成矩阵

self.matrix = []

for line in train\_set:

sentence\_vector = np.zeros(size)

for word in line:

if word in self.word\_vector:

sentence\_vector += self.model[word]

self.matrix.append(sentence\_vector)

print('已处理', len(self.matrix), '行')

**实现sigmod函数**

def sigmod(self, x):

# sigmod函数

return 1 / (1 + np.exp(-x))

**使用梯度下降法更新w的值**

def grandascent(self, datain, labels, numiter):

# 使用梯度下降法更新w的值

datain = np.mat(datain)

m, n = np.shape(datain)

weights = self.weights

step = self.step

for j in range(numiter):

print('迭代', j, '次', end=' ')

for i in range(m):

output = self.sigmod(datain[i, :] \* weights)

err = labels[i] - output

weights = weights + step \* datain[i, :].transpose() \* err

print(weights[0][0], weights[1][0], weights[2][0], weights[3][0])

return weights

**对测试集数据进行分类**

def classification(self, test\_set, size, numiter, module\_file\_name):

# 使用Word2Vec生成测试集数据的词向量

model = word2vec.Word2Vec.load(module\_file\_name)

test\_word\_vector = list(model.wv.vocab.keys())

# 使用梯度下降法迭代出w的估计值

labels = []

self.weights = self.grandascent(np.array(self.matrix), self.train\_labels, 500)

self.step = 0.01

self.weights = self.grandascent(np.array(self.matrix), self.train\_labels, numiter - 500)

# 生成测试集数据的句子向量，再组合成矩阵

test\_x = []

for line in test\_set:

sentence = np.zeros(size)

for word in line:

if word in test\_word\_vector:

sentence += model[word]

test\_x.append(sentence)

test\_x = np.mat(test\_x)

# 计算P(y|x)，选择概率最大的项作为测试集数据的类别

for i in range(len(test\_x)):

labels.append(self.sigmod(test\_x[i, :] \* self.weights)[0, 0] > 0.5)

return labels

## 实验结果及分析

* **实验结果展示**

1. **KNN**

虽说KNN效果并不是很理想，但我还是选择它预测一次测试集数据，然后进行一次rank测试。在本次预测中，我选择前6000行数据作为训练集数据，预测测试集数据，效果如图：



1. **Naïve Bayes**

使用不同的a值运行朴素贝叶斯分类器预测测试集数据类别效果比KNN算法效果好，rank排名如图所示：



虽然Naïve Bayes方法比KNN算法的准确率提升了26个百分点，但是排名并没有什么变化。

1. **Logistic Regression**

由于使用逻辑回归预测验证集的准确率并没有比使用朴素贝叶斯高，所以在跑rank时并没有使用逻辑回归这种方法，这可能是我没有找到最优的参数导致的吧。

* **评测指标展示**

1. **KNN**

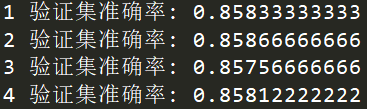
由于KNN算法效率过低，在进行高维的矩阵运算时时间复杂度过高，等待的时间过长，所以没有进行交叉验证，只选取前5000行数据作为训练集，预测随机选取的3000行验证集，效果如图所示：



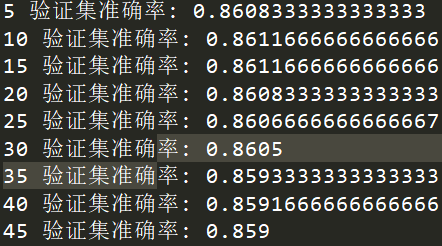
验证集的准确率有60%，效果并不是特别理想，所以在project中并没有将KNN算法作为主要考虑对象。

1. **Naïve Bayes**

使用朴素贝叶斯分类器预测验证集时，我采用了交叉验证的方法。将整个训练集数据分为4部分，每6000行数据作为一部分，每次验证时使用其中的一部分，其余3部分作为训练集，效果如下：



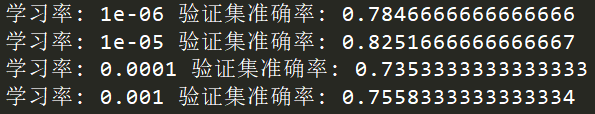
考虑到二元分类是“影评预测正负”，那么情感类词语对影评正负的影响会比较高，所以在计算时，如果为情感类词语，那么将取较大的a值；如果为非情感类词语，那么将取。取不同的a效果不同：



在图中可以看出，准确率有所提升。

1. **Logistic Regression**

使用逻辑回归预测验证集时，我使用Word2Vec利用训练集数据生成5000维的词向量，取前18000条语句作为训练集，后6000条作为验证集，使用不同的的学习率，初始 ，迭代500次，效果如下：



从图中可以看出，当学习率时，准确率最高，但效果还是没有朴素贝叶斯分类器好。