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$\mathrm{T}=\{(\mathrm{x}_1,\mathrm{y}_1),(x_2,y_2),\ldots,(\mathrm{x}_\mathrm{N},\mathrm{y}_\mathrm{N})\}$，其中，$\mathrm{x}_\mathrm{i}\in X\subseteq R^n(\mathrm{i}=1,2,\ldots,\mathrm{N})$为实例的特征向量，$\mathrm{y}_\mathrm{i}\in Y=\{c_1,c_2,\ldots,c_k\}(\mathrm{i}=1,2,\ldots,\mathrm{N})$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$\mathrm{N_k}(x)$，在$\mathrm{N_k}(x)$中根据多数表决的方式决定 x 的类别 y:

$$\mathrm{y} = \arg\max_{c_j} \sum_{x_i \in N_k(x)} I(y_i = c_j), i = 1,2,\dots,N; j = 1,2,\dots,K$$

其中，I 为指示函数，当$\mathrm{y_i} = c_j$时 I 为 1，否则 I 为 0。

2) 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$\mathrm{P(A|B)} = \frac{P(AB)}{P(B)}$，贝叶斯定理通过 P(A|B)来求 P(B|A)：$\mathrm{P(B|A)} = \frac{P(A|B)P(B)}{P(A)}$，其中 P(A)由全概率公式可分解为：$\mathrm{P(A)} = \sum_{i=1}^{n} P(B_i)P(A|B_i)$。

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$\mathrm{x} = (\mathrm{x_1}, \mathrm{x_2}, \dots, \mathrm{x_n})$，类标记集合含有 k 中类别，即$\mathrm{y} = (\mathrm{y_1}, y_2, \dots, y_n)$。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$\mathrm{P(y_1}|x), P(y_2|x), \dots, P(y_k|x)$中最大的项，即求出后验概率最大的输出：$\arg\max_{y_k} P(y_k|x)$。由贝叶斯定理可知：$\mathrm{P(y_k}|x) = \frac{P(x|y_k)P(y_k)}{\sum_{k=1}^{n} P(x|y_k)P(y_k)}$。

分子中的$\mathrm{P(y_k)}$是先验概率，可直接根据训练集数据计算得出，而条件概率$\mathrm{P(x|y_k)}$有指数级数量的参数，假设第 j 维特征$\mathrm{x_j}$可取值有$\mathrm{S_j}$个，$\mathrm{j} = 1,2,3,\dots,\mathrm{n}$，y 可取值有 K 个，那么参数个数为$\mathrm{K}\prod_{j=1}^{n} S_j$。

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$\mathrm{x_1}, \mathrm{x_2}, \dots, x_n$相互独立，在这个假设下，条件概率：$\mathrm{P(x|y_k)} = \mathrm{P}(\mathrm{x_1}, x_2, \dots, x_n|y_k) = \prod_{i=1}^{n} P(x_i|y_k)$，如此，参数规模降为$\sum_{i=1}^{n} S_i K$，那么$\mathrm{P(y_k}|x) = \frac{P(y_k)\prod_{i=1}^{n} P(x_i|y_k)}{\sum_k P(y_k)\prod_{i=1}^{n} P(x_i|y_k)}$，于是朴素贝叶斯分类器可表示为

$$y = f(x) = \arg\max_{y_k} P(y_k|x) = arg\max_{y_k} \frac{P(y_k)\prod_{i=1}^{n} P(x_i|y_k)}{\sum_k P(y_k)\prod_{i=1}^{n} P(x_i|y_k)}$$

在计算先验概率和条件概率时，需要做平滑处理：

$$P(y_k) = \frac{N_{y_k} + a}{N + ka}$$
$$P(x_i|y_k) = \frac{N_{y_k,x_i} + a}{N_{y_k} + na}$$

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

在实际实现的过程中，考虑到$P(y_k|x)$中分母都为P(x)，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：

$$y = f(x) = arg\max_{y_k} P(y_k|x) = arg\max_{y_k}(\log P(y_k) + \sum_{i=1}^{n} \log P(x_i|y_i))$$

3) 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：

$$P(Y = 1|x) = \frac{\exp(w \cdot x + b)}{1 + \exp(w \cdot x + b)}$$
$$P(Y = 0|x) = \frac{1}{1 + \exp(w \cdot x + b)}$$

其中，$x \in R^n$是输入，$Y \in \{0,1\}$是输出，$w \in R^n$和$b \in R$是参数，w 称为权值向量，b 称为偏置，$w \cdot x$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$P(Y = 1|x)$和$P(Y = 0|x)$，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$t = w \cdot x$，其值域为实数域，通过逻辑回归模型可将线性函数$t = w \cdot x$转化为概率：$P(Y = 1|x) = \frac{\exp(w \cdot x + b)}{1 + \exp(w \cdot x + b)}$。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：$P(Y = 1|x) = h(x)$，　　$P(x = 0|x) = 1 - h(x)$

似然函数为：$\prod_{i=1}^{n}(h(x_i))^{y_i}(1-h(x_i))^{1-y_i}$

对数似然函数为：$\mathrm{L(w)}=\sum_{i=1}^{n}(y_i\log h(x_i)+(1-y_i)\log(1-h(x_i)))$

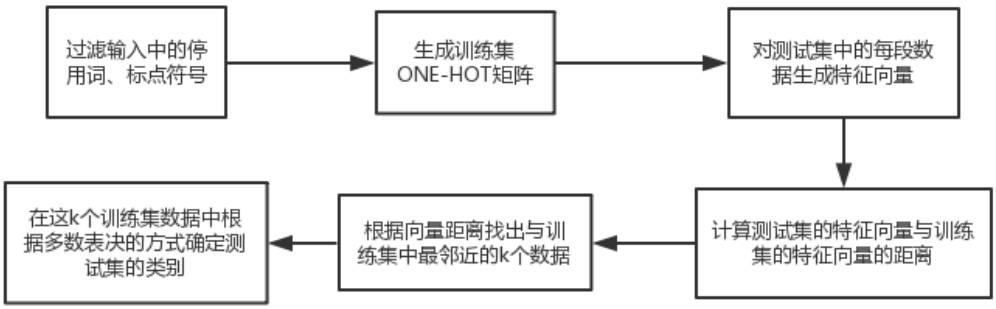
对 L(w)求极大值可得到 w 的估计值，问题就转变成以对数似然函数为目标函数的最优化的问题。令$\mathrm{C(w)}=-\mathrm{L(w)}=-\sum_{i=1}^{n}(y_i\log h(x_i)+(1-y_i)\log(1-h(x_i)))$，可使用梯度下降法求解 C(w)的极小值。由梯度下降法可知 w 的更新过程：

$$\mathrm{w_j}=w_j-\eta\frac{\partial C(w)}{\partial w_j}=w_j-\eta\sum_{i=1}^{n}(h(x_i)-y_i)x_j,(j=1,2,\dots,m)$$

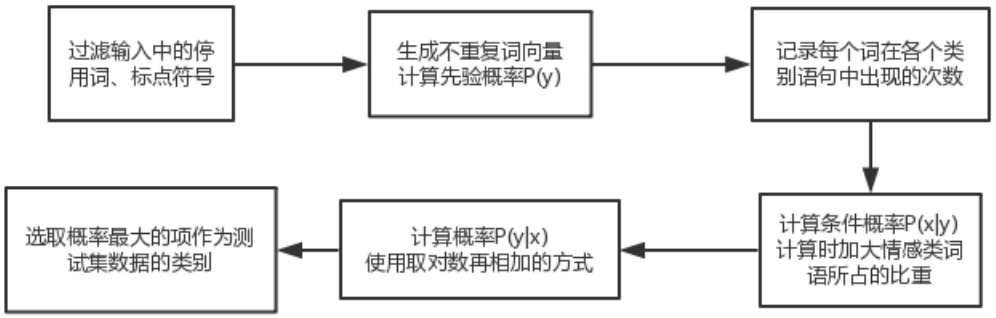
不断迭代直至 w 收敛，这样就可以得到 w 的估计值。

· 流程图&伪代码

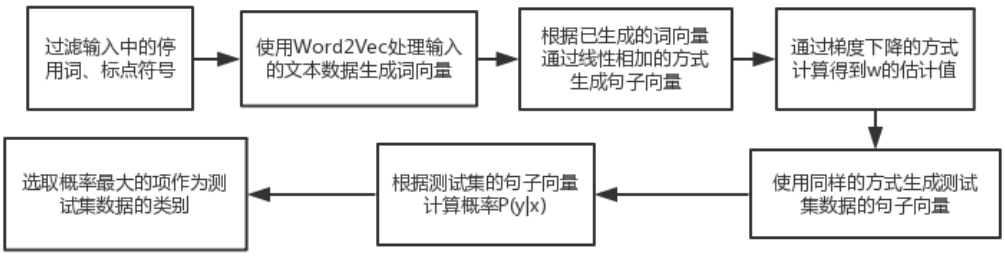
1) KNN



2) Naïve Bayes



3) 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
```

2) 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
```

3) 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	二分类	
2	SYSUfello	91.2
3	太吾绘卷	90.96667
4	(*·ω·`)???	90.46667
4	某不知名	90.3
6	Artificial Id	89.5
7	来个名字	89.23333
8	智商二五	88.45
9	队伍名字	88.4
10	脱单率25	88.2
11	快乐码男	88.03333
12	拜冥坤尼	86.95
13	tee	86.93333
14	红鲤鱼与	86.11667
15	佚名	85.86667
16	[Y]	83.88333
17	化肥挥发	83.3
18	rank第一	82.51667
19	wolfberry	66.56667
20	小熊维尼	61.25
21	小猪佩皮	60.83333

2) 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:

1	二分类	
2	(*·ω·*)???	91.9
3	SYSUfello	91.48333
4	某不知名	91.45
5	太吾绘卷	90.96667
6	来个名字	90.66667
7	快乐码男	90.4
8	佚名	89.85
9	至强531	89.83333
10	Artificial Id	89.56667
11	鸽子固定	89.56667
12	拜冥坤尼	89.31667
13	太真实了	88.8
14	智商二五	88.58333
15	脱单率25	88.55
16	队伍名字	88.55
17	tee	86.93333
18	rank第一	86.45
19	化肥挥发	86.21667
20	红鲤鱼与	86.11667
21	小猪佩皮	86.06667

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。

3) 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：

```
验证集准确率： 60.0133333333333
```

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。

2) 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：

```
1 验证集准确率： 0.85833333333
2 验证集准确率： 0.85866666666
3 验证集准确率： 0.85756666666
4 验证集准确率： 0.85812222222
```

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$P(x_i|y_k)=\frac{N_{y_k x_i}+a}{N_{y_k}+\lambda}$时，如果$x_i$为情感类词语，那么将取较大的 a 值；如果$x_i$为非情感类词语，那么将取$a=1$。取不同的 a 效果不同：

```
5 验证集准确率： 0.8608333333333333
10 验证集准确率： 0.8611666666666666
15 验证集准确率： 0.8611666666666666
20 验证集准确率： 0.8608333333333333
25 验证集准确率： 0.8606666666666667
30 验证集准确率： 0.8605
35 验证集准确率： 0.8593333333333333
40 验证集准确率： 0.8591666666666666
45 验证集准确率： 0.859
```

在图中可以看出，准确率有所提升。

3) 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 $w=(1,1,1,\dots,1)$，迭代 500 次，效果如下：

```
学习率： 1e-06 验证集准确率： 0.7846666666666666
学习率： 1e-05 验证集准确率： 0.8251666666666667
学习率： 0.0001 验证集准确率： 0.7353333333333333
学习率： 0.001 验证集准确率： 0.7558333333333334
```

从图中可以看出，当学习率$\eta=0.00001$时，准确率最高，但效果还是没有朴素贝叶斯分类器好。