

# 中山大学移动信息工程学院本科生实验报告

## (2015 年秋季学期)

课程名称: Artificial Intelligence

任课教师: 饶洋辉

|    |             |         |                         |
|----|-------------|---------|-------------------------|
| 年级 | 13 级        | 专业 (方向) | 移动信息工程                  |
| 学号 | 13354485    | 姓名      | 朱琳                      |
| 电话 | 13726231932 | Email   | <u>280273861@qq.com</u> |

### 一. 实验方法

#### 1. 阐述你的思路。

##### (1) kNN 算法

- ①首先我需要读取文本文件和每个训练文本的情感值。由于每个文本都有一些固定的属性，所以我需要一个数据结构将他们聚合起来，以文本为单位。
- ②得到不同的单词（这个在读文本的时候可以直接判断）以及计算每个文本的向量，并将他们归一化。
- ③选用 stopwords，用来筛选掉一些停用词，可以有效防止一些无关紧要的介词，助词等的影响。
- ④计算每个测试文本与每个训练文本之间的距离（可以是欧式距离，也可以是 city-block 距离或者余弦距离）
- ⑤根据 kNN 算法，需要手动输入 k。然后我们对于每个测试文本需要找到与之距离最小的 k 个训练文本（余弦距离需要找最大的 k 个文本）。
- ⑥ 将这 k 个文本距离进行取倒数（余弦不需要取倒数），然后将这些权重归一化。最后将这些权重乘以他们的情感值即为测试文本情感值。
- ⑦输出情感值到指定文本。

##### (2) NB 算法

- ①②③与 (1) 相同
- ④在②中进行对**训练文本**向量归一化的时候，假设每个单词都事先出现一遍。这样就解决了乘法因子为零的问题。
- ⑤ NB 算法：对于每个测试文本，将其每个不为 0 的维度对应的训练文本进行相乘。在将每个训练文本得到的值进行累加。
- ⑥对每个测试文本的六种情感进行归一化。
- ⑦输出得到的上述情感值到指定的文本

#### 2. 截图你的关键代码（注明使用的是什么语言）

【说明】使用 c++语言；

##### (1) kNN

- ①关于文本的数据结构。k\_th 为距离最近的 k 个训练文本的一些参数。

```

struct Files { //每个训练文本
    string words[100]; //文本的所包含的单词
    float vector[5000]; //与不相同的单词对应生成的向量
    float feel[10];
    float dis[2000]; //每个测试文本与各个训练文本的距离。训练文本不需要这个。
    float num;
};
struct k_th {
    float dis;
    float probability;
    float weight;
};

```

②读取 stopwords 以及读取数据集的词并筛掉 stopwords。均用 map 来实现。Stopwords 中以每个单词为关键字，返回的是 bool 类型。而读取数据集是用单词做关键字，返回 int 类型，有利于计数，来记录每个单词在文本中出现的顺序。

```

void read_stopwords() {
    ifstream fin;
    fin.open(stopwords);
    string words;
    while (fin >> words) {
        check_stop_words[words] = true;
    }
}

void read_words() { //读取文本单词，并生成不重复的单词
    ifstream fin;
    fin.open(Dataset_words);
    string words;
    int col = 0, flag = 0; //用于区分trains和test
    getline(fin, words); //忽略掉第一行标题行
    while (fin >> words) {
        string tmp = words + "xxxxx"; //防止判断单词长度不够导致bug
        if (tmp.substr(0, 5) == "train" && tmp.substr(5, 1) <= "9"
            && tmp.substr(5, 1) > "0") {
            flag = 0;
            num_of_trains++;
            col = 0;
        } else if (tmp.substr(0, 4) == "test" && tmp.substr(4, 1) <= "9"
            && tmp.substr(4, 1) > "0") {
            flag = 1;
            num_of_tests++;
            col = 0;
        } else if (check_stop_words[words] == false) {
            if (!flag) {
                trains[num_of_trains].words[col++] = words;
            }
            if (check_map[words] == 0) {
                check_map[words] = 1;
                unique_words[num_of_unique++] = words;
            }
        }
    }
}

```

```

    }
    map_train[num_of_trains][words]++;

    } else {
        tests[num_of_trains].words[col++] = words;
        map_test[num_of_tests][words]++;
        if (check_map[words] == 0) {
            check_map[words] = 1;
            unique_words[num_of_unique++] = words;
        }
    }
}
}
num_of_trains++;
num_of_tests++;
cout << "num_of_trains=" << num_of_trains << endl;
cout << "num_of_tests=" << num_of_tests << endl;
cout << "num_of_unique=" << num_of_unique << endl;
fin.close();
}

```

### ③计算向量。以及向量的归一化

```

void get_vector() {
    for (int row = 0; row < num_of_trains; row++) { //对于每个训练文本
        trains[row].num = 0;
        for (int col = 0; col < num_of_unique; col++) {
            if (map_train[row][(unique_words[col])]) {
                trains[row].vector[col] = 1;
                trains[row].num++;
            } else {
                trains[row].vector[col] = 0;
            }
        }
    }
    for (int row = 0; row < num_of_tests; row++) { //对于每个测试文本
        tests[row].num = 0;
        for (int col = 0; col < num_of_unique; col++) {
            if (map_test[row][(unique_words[col])]) {
                tests[row].vector[col] = 1;
                tests[row].num++;
            } else {
                tests[row].vector[col] = 0;
            }
        }
    }
}

void vector_to_one() {
    for (int row = 0; row < num_of_trains; row++) {
        for (int col = 0; col < num_of_unique; col++) {
            trains[row].vector[col] /= trains[row].num;
        }
    }

    for (int row = 0; row < num_of_tests; row++) {
        for (int col = 0; col < num_of_unique; col++) {
            tests[row].vector[col] /= tests[row].num;
        }
    }
}
}

```

④计算距离。最终我选用的是余弦距离，这个跑出来的效果最理想。

#### A 余弦距离

```
void get_dis() {
    for (int t_row = 0; t_row < num_of_tests; t_row++) { //每个测试文本
        for (int train_row = 0; train_row < num_of_trains; train_row++) { //每个训练文本
            tests[t_row].dis[train_row] = 0;
            float distance = 0;
            for (int k = 0; k < num_of_unique; k++) {
                distance += abs(trains[train_row].vector[k] * tests[t_row].vector[k]);
            }
            tests[t_row].dis[train_row] = distance;
        }
    }
}
```

#### B 曼哈顿距离

```
void get_dis() {
    for (int t_row = 0; t_row < num_of_tests; t_row++) { //每个测试文本
        for (int train_row = 0; train_row < num_of_trains; train_row++) { //每个训练文本
            tests[t_row].dis[train_row] = 0;
            float distance = 0;
            for (int k = 0; k < num_of_unique; k++) {
                distance += abs(trains[train_row].vector[k] - tests[t_row].vector[k]);
            }
            tests[t_row].dis[train_row] = distance;
        }
    }
}
```

#### C 欧氏距离

```
void get_dis() {
    for (int t_row = 0; t_row < num_of_tests; t_row++) { //每个测试文本
        for (int train_row = 0; train_row < num_of_trains; train_row++) { //每个训练文本
            tests[t_row].dis[train_row] = 0;
            float distance = 0;
            for (int k = 0; k < num_of_unique; k++) {
                distance +=
                    (trains[train_row].vector[k] - tests[t_row].vector[k])
                    * (trains[train_row].vector[k]
                     - tests[t_row].vector[k]);
            }
            tests[t_row].dis[train_row] = sqrt(distance);
        }
    }
}
```

④kNN。这里计算 k 个最大值是采用了类似于冒泡的方法，循环 k 次，每次都找到最大值并且将其置为最小值不再参与比较，最后再将其数据恢复。这种方法避免了直接全排序，在 k 值比较小的情况下大大降低了时间复杂度。——以余弦距离来进行说明

```
void kNN(int k) {
    for (int f = 0; f < num_of_feel; f++) { //对于每种情感。0=anger, 1=disgust...见enum
        for (int t_row = 0; t_row < num_of_tests; t_row++) { //对于每个测试文本
            //得到距离此test最远的k个向量。
            k_th v[300]; //距离此test最远的文本。
            int number[300];
            for (int count = 0; count < k; count++) { //计算距离测试文本最远的k个文本;
                float max = 0;
                for (int train_row = 0; train_row < num_of_trains;
                     train_row++) {
                    if (tests[t_row].dis[train_row] > max) {
                        max = tests[t_row].dis[train_row];
                        number[count] = train_row; //第trains_row个训练文本
                    }
                } //循环找到最大值。
                v[count].dis = max;
                v[count].probability = trains[(number[count])].feel[f];
                tests[t_row].dis[(number[count])] = -1; //将此轮找到的那个最小距离置为最小，不再参与比较。共有k轮比较
            }
            for (int count = 0; count < k; count++) {
                tests[t_row].dis[(number[count])] = v[count].dis; //数据恢复
            }
        }
    }
}
```

```

        //至此已得到距离此test最远的k个向量。
        caculate(k, v, f, tests[t_row]);
    }
}

```

Calculate 函数：用来计算权重和文本预测的情感值

```

void caculate(int k, k_th v[], int f, Files &test) { //算权重，权重归一化，计算anger等值
    test.feel[f] = 0;
    float num = 0;
    for (int i = 0; i < k; i++) {
        v[i].weight = v[i].dis;
        num += v[i].weight;
    }
    for (int i = 0; i < k; i++) {
        v[i].weight /= num;
        test.feel[f] += v[i].weight * v[i].probability;
    }
}

```

Calculate 函数 2：这个是实现欧式距离或者街区距离的关于权重的计算，需要用距离的倒数的三次方来进一步稀疏画得到的距离，有利于使得最接近的文本产生更好的权重

```

void caculate(int k, k_th v[], int f, Files &test) { //取倒数，算权重，权重归一化，计算anger等值
    // float num_dis = 0;
    test.feel[f] = 0;
    // for (int i = 0; i < k; i++) {
    //     num_dis += v[i].dis;
    // }
    float num = 0;
    for (int i = 0; i < k; i++) {
        //v[i].dis /= num_dis; //先进行向量的归一化
        v[i].weight = pow(1.0 / v[i].dis, 3);
        num += v[i].weight;
    }
    for (int i = 0; i < k; i++) {
        v[i].weight /= num;
        test.feel[f] += v[i].weight * v[i].probability;
    }
}

```

## (2)NB

①NB 算法的数据结构，他们作用均写在声明后面

```

struct Files { //每个训练文本
    string words[100]; //文本的所包含的单词
    float vector[5000]; //与不相同的单词对应生成的向量
    float feel[10]; //保存各个情感值的概率
    float num; //用于Vector的归一化。vector的all维度向量之和。
    int Dim[10000]; //记录file内的每个words所在的维度。
    int numDim; //一个test的词总量。
};

```

②关于文本的读取和 kNN 相同，不过，在 **vector** 进行归一化的时候，先要设定每个词都预先出现过一次

```
void vector_to_one() {
    for (int row = 0; row < num_of_trains; row++) {
        trains[row].num += num_of_unique;
        for (int col = 0; col < num_of_unique; col++) {
            trains[row].vector[col] += 1.0;
            trains[row].vector[col] /= trains[row].num; //假设每个维度的词都事先出现过一次。
        }
    }

    for (int row = 0; row < num_of_tests; row++) {
        int num_dim = 0;
        for (int col = 0; col < num_of_unique; col++) {
            if (tests[row].vector[col] == 1) {
                tests[row].vector[col] /= tests[row].num;
                tests[row].Dim[num_dim++] = col;
            }
        }
        tests[row].numDim = num_dim;
    }
}
```

③NB 关键代码实现，以及 6 种情感的归一化


```
void NB() { //对于情感feel的NB
    for (int feel = 0; feel < num_of_feel; feel++) {
        for (int test_row = 0; test_row < num_of_tests; test_row++) { //对于每个测试文本。
            int nDim = tests[test_row].numDim; //此tests共有dim个有效维度。
            tests[test_row].feel[feel] = 0;
            for (int train_row = 0; train_row < num_of_trains; train_row++) { //对于每个训练文本
                float tmp = trains[train_row].feel[feel];
                for (int d = 0; d < nDim; d++) {
                    int Dim = tests[test_row].Dim[d]; //test在第Dim个维度上的词。
                    tmp *= trains[train_row].vector[Dim];
                }
                tests[test_row].feel[feel] += tmp;
            }
        }
    }
    feel_to_one(); //每个test文本进行六种情感归一化
}
```

```
void feel_to_one() {
    for (int t_row = 0; t_row < num_of_tests; t_row++) {
        float num_feel = 0;
        for (int feel = 0; feel < 6; feel++) {
            num_feel += tests[t_row].feel[feel];
        }
        for (int feel = 0; feel < 6; feel++) {
            tests[t_row].feel[feel] /= num_feel;
        }
    }
}
```

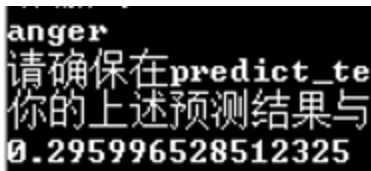


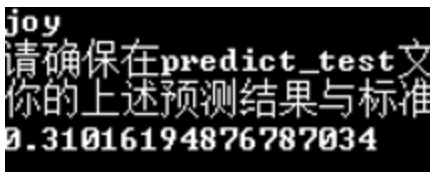
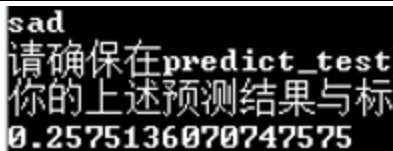
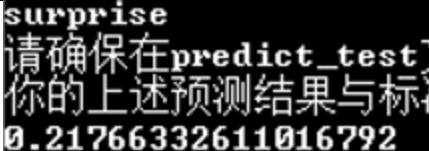
## 二. 实验结果

(1) kNN

A 余弦距离

| ①anger  | ②disgust  | ③fear   |
|---|---|---|
|  |  |  |
| ④joy  | ⑤sad  | ⑥surprise   |
|  |  |  |
| 平均值   | 0.300 (k=10)  |   |

## B city-clock 距离

| ①anger  | ②disgust  | ③fear   |
|---|---|---|
|  |  |  |
| ④joy  | ⑤sad  | ⑥surprise   |
|  |  |  |
| 平均值   | 0.280 (k=10) 效果不如余弦距离好  |   |

## (2)NB

| ①anger  | ②disgust  | ③fear   |
|---|---|---|
|   |   |   |
| ④joy  | ⑤sad  | ⑥surprise   |
|  |  |  |
| 平均值   | 0.280   |   |

## 【实验分析】

①kNN 算法中，我共前后选用了 city-block 距离，欧式距离，余弦距离三种，欧氏距离由于数据比较密集，不太好处理，所以实现的效果也就在 0.20 左右，街区距离的效果比较好一点，尤其是添加了筛掉停用词这一步，6 种情感相关度的平均值可以达到 0.28，后来我尝试用余弦距离来做，从理论上讲余弦距离解决了前两种距离求差的一些误差，能与文本完全不相关的训练文本剔除。

②在实现选取 k 个距离最大值作为权重的时候，我采取的方式是冒泡形式，只冒到第 k 个值就停止，避免了直接排序，减低了复杂度。

③ NB 算法，除了上述方式我还尝试了使用一个很小的值代替 0 的情况（比如 0.01），但是实现的效果均在 0.25 左右，并不能很好的实现。后来我又用 1 代替 0，即忽略掉那个单词，这个的效果更差一些，大概平均值只有 0.18 的样子，所以最终还是选择了效果最好的一组。

④ 上次的实验所交的材料中，实现的效果不是很理想，当时是选用街区距离进行计算的，后来加了停用词进行筛选之后，效果得到了显著提升。