人工智能lab1实验报告

学号：17338233　　专业：计科　　姓名：郑戈涵

TF-IDF

1.算法原理

句子词频归一化后的概率表示(term frequency)，简称tf，能够体现句子的特征。

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

n_i, d代表第i个单词在第d个文档中出现的频数。

但是某些词如果在多个句子中均有出现，则说明其重要性较低，不能很好的体现句子的特征，因此需要计算逆向文档频率(inverse document frequency)。

$$idf_i = log\frac{|C|}{|C_i|}$$

C代表文档数量， C_i代表在i篇文档中出现。由于有些单词可能完全不出现，该公式会出现除0错，因此可以修改为：

$$idf_i = log\frac{|C|}{|C_i|+1}$$

结合两者，可以得到TF-IDF矩阵，公式如下：

$$tf-idf_{i,j} = \frac{n_{i,j}}{\sum_{k=1}^{V} n_{i,k}} \times log\frac{|C|}{|C_j|+1}$$

2.伪代码

```
function TF-IDF(Array)
input:经过处理后只保留文本内容的字符串列表
output:TF-IDF矩阵
n <- num of different word in Array
C <- num of sentence in Array
TF-IDF <- C*n Matrix
wholeWordFrequency
wordList<-all words in Array
for sentence in Array
    wordFrequency <- empty map<string,float>
    for word in wordList
        sentence.wordFrequency[word] <- 0
        if word is in sentence
            sentence.wordFrequency[word] <- sentence.wordFrequency[word] + 1
            wholeWordFrequency[word] <- wholeWordFrequency[word] + 1
 for word in wordList
    idf[word]=log(C/(wholeWordFrequency[word]+1))
 for sentence in Array
    for word in wordList
```

```
        TF-IDF[sentence_index]
[word_index]=idf[word]*sentence.wordFrequency[word]
 return TF-IDF
```

3.代码截图

类设计

为了计算TF-IDF矩阵，我设计了两个类，sentence负责存储一个文档和它的词频，dataSet存储所有sentence。

```
class sentence
{
private:
    int id;
    int wordCnt = 0;
    string str;
    void load(const string& s);
    map<string, float> termFreq;
    set<string>word;
public:
    sentence() = default;
    explicit sentence(const string& s) :str(getStr(s)) {
        load(s);
    }
    ~sentence() = default;
    friend bool operator<(const sentence& s1, const sentence& s2) { return
s1.id < s2.id; }
};

class dataSet
{
private:
    void load(ifstream& infile);
    int sentenceCnt = 0;
    int wordCnt = 0;
    vector<sentence> data;
    //IDF
    vector < float > InvDocFreq;
    //词语集合
    vector<string> word;
    //词频表
    vector<int> wordFreq;
    vector<vector<float>> TF_IDF;
    bool TF_IDF_inited = false;
    void data_init(ifstream& infile);
    void TF_IDF_init() ;
public:
    dataSet() = default;
    dataSet(ifstream& infile);
    ~dataSet() = default;
};
```

sentence类

sentence类的load函数负责处理其中出现的单词的词频，重载<运算符用于map的排序功能，可以使其中的键值对按照字符串的字典序排序。

```
void load(const string& s) {
    string tmpStr;
    stringstream ss(str);
    while (ss >> tmpStr) {
        wordCnt++;
        word.insert(tmpStr);
        termFreq[tmpStr]++;
    }
    for (auto& str : termFreq) {
        str.second /= wordCnt;
    }
}
friend bool operator<(const sentence& s1, const sentence& s2) { return
s1.str < s2.str; }
```

dataSet类

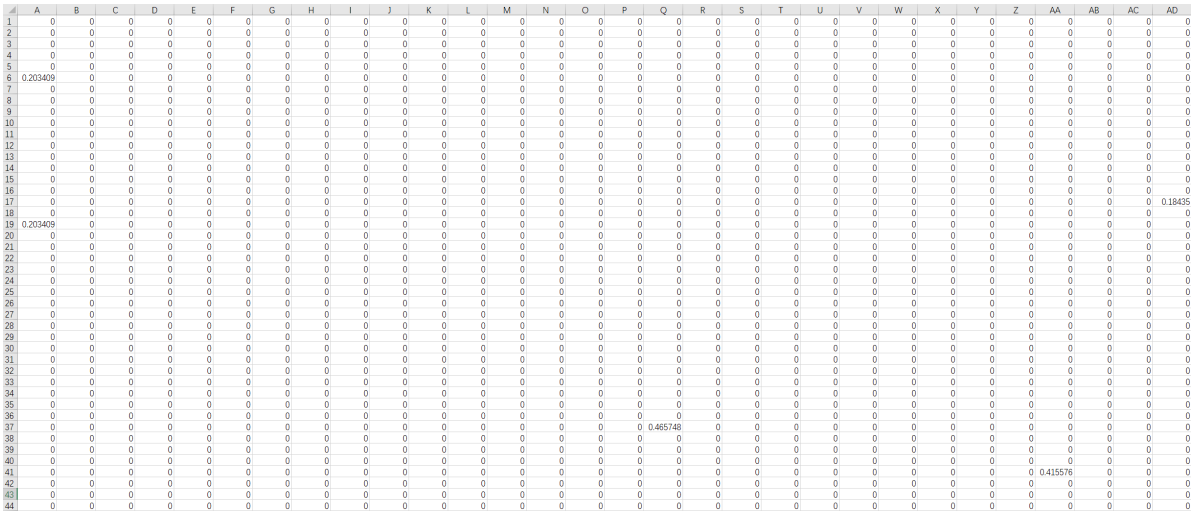
dataSet类初始化时需要处理所有的句子，统计整体的词频和句子个数，用于计算IDF向量。使用map可以将句子按照字典序排好，用transform函数导出到vector中方便索引。此函数需要保证所有vector内的数据是有序的，以后计算TF-IDF矩阵可以直接通过索引获得数据，效率较高。

```
void data_init(ifstream& infile) {
    int line_no = 0;
    multiset<sentence>sentenceSet;
    map<string, int> wordFreqMap;
    string s;
    getline(infile, s);
    while (getline(infile, s))
    {
        sentence sent(s);
        sentenceSet.insert(sent);
        auto TF = sent.getTF();
        for (auto& t : TF) {
            wordFreqMap[t.first]++;
        }
        sentenceCnt++;
    }
    wordCnt = wordFreqMap.size();
    word.resize(wordCnt);
    wordFreq.resize(wordCnt);
    data.resize(sentenceCnt);
    //将词频map转换为两个数组
    transform(wordFreqMap.begin(), wordFreqMap.end(), word.begin(), []
(pair<string, int> p) { return p.first; });
    transform(wordFreqMap.begin(), wordFreqMap.end(), wordFreq.begin(), []
(pair<string, int> p) { return p.second; });
    transform(sentenceSet.begin(), sentenceSet.end(), data.begin(), []
(sentence s) { return s; });
    //此后data有序
}
```

dataSet类经过初始化后，TF_IDF_init函数负责计算TF-IDF矩阵，检查是否计算了各个句子的词频后，会根据公式计算矩阵中每个元素的值

```
void TF_IDF_init() {
    if (sentenceCnt == 0 || wordCnt == 0)
        throw invalid_argument("Data hasn't been initialized!");
    TF_IDF = vector<vector<float>>(sentenceCnt, vector<float>(wordCnt, 0));
    int i = 0;
    string currTerm;
    for (int i = 0; i < sentenceCnt; i++)
    {
        for (int j = 0; j < wordCnt; j++)
        {
            currTerm = word[j];
            const auto& TF = data[i].getTF();
            if (TF.find(currTerm) != TF.end())
                //计算公式
                TF_IDF.at(i).at(j) = TF.at(currTerm) * InvDocFreq[j];
            else TF_IDF.at(i).at(j) = 0;
        }
    }
    TF_IDF_inited = true;
}
```

4.实验结果以及分析



从图片中可以发现，大部分位置都是0，因为一句话中只有不到十个单词，所以大部分词的词频为0。经手动计算比对部分数据，该矩阵与公式是一致的。

KNN分类

1.算法原理

在文档进行编码（one hot或TF-IDF）了前提下，可以选定一种距离（比如欧式距离），并计算目标文档与训练集中各个文档的距离，就可以根据K-邻近算法用训练集的文档的属性，计算目标文档的属性。

计算规则为，目标文档的情感是距离最小的前k个文档的众数的情感。

2.伪代码

```
function classify(sentence,p,k)
    input:想要分类的句子，距离计算参数，k
    output:情感
    for train_sentence in train_set //train_set：数据集
        dist[train_sentence]<-0
        for word in word_set //word_set:整个数据集的单词集合
            //encode函数计算新句子的编码，TF_IDF为矩阵
            dist[train_sentence] += (TF_IDF[train_sentence][word]-
encode(sentence)[word])^p
        dist[train_sentence]=dist[train_sentence]^(1/p)
    sort dist by distance between train_sentence and sentence
    list[type_of_emotion]<-count emotion of first k dist element
    return emotion correspond to max count of list
```

3.代码截图

emotion枚举

使用枚举类可以使代码更清晰，我设计了emotion的枚举。

```
enum class emotion :int
{
    anger, disgust, fear, joy, sad, surprise, unknown
};
const string emoStr[] = {
    "anger", "disgust", "fear", "joy", "sad", "surprise"
};
```

在第一题的基础上，sentence类和dataSet类都需要做修改，sentence类需要增加emotion的域，dataSet需要能够计算给定文档与其中各个文档的距离。

编码：TF-IDF表示

这次可以使用两种编码，因此我设计了编码的枚举

```
enum class codeType :int
{
    one_hot, tf_idf
};
```

按照算法，首先为目标文档编码，由于要求使用TF_IDF表示，而其中的逆向文档频率会受到所有文档的共同影响。为了不让新文档影响整个数据集的编码，我只为目标文档进行单独编码。也就是说，其他文档的TF-IDF矩阵不受影响，新文档的TF-IDF表示使用原有矩阵的逆向文档频率。**要注意的是，新文档中可能有原词表中不存在的词，我这里直接当做不存在。因为对所有训练集内的文档，这些词的距离贡献是一样的，是否计算不影响分类结果。**

```
vector<float> dataSet::encode(const sentence& s) {
    vector<float> code(wordCnt, 0);//编码用于计算距离
    auto& wordset = s.getWordSet();
    int i;//找到句子的单词在word数组中的位置
    for (auto w : wordset) {
        auto pos = find(word.begin(), word.end(), w);
```

```
        if (pos != word.end()) {
            i = pos - word.begin();
            code[i] = s.getWordFreq(w) * InvDocFreq[i];
        }
    }

    return code;
}
```

距离计算

计算距离是本算法的核心之一，由于要多次使用，需要考虑性能问题。本代码除类的初始化时使用map外，其他部分均使用vector进行索引。课上讲的距离可以分为两类，第一类是Lp距离，第二类是余弦相似度

Lp距离

本函数假定已知新文档的编码，只需将两个向量的Lp距离计算出来，顺带句子的信息一并放入vector即可。公式如下：

$$L_p = (\sum_{l=1}^{n} |v_1^{(l)} - v_2^{(l)}|^p)^{\frac{1}{p}}$$

计算距离前要根据编码得到向量。13~16行获得one-hot编码，根据传入的编码参数决定如何计算距离。

```
vector<pair<sentence*, float>> dataSet::distLP(const sentence& s, codeType st, float p = 1) {
    vector<float>code = encode(s);
    if (p <= 0)throw invalid_argument("line 185: p must be positive!");
    vector<pair<sentence*, float>> res(sentenceCnt);
    for (int j = 0; j < sentenceCnt; j++)
    {
        const auto& v1 = TF_IDF[j];
        res[j].first = &data[j];
        res[j].second = 0;
        float lhs, rhs;
        for (int k = 0; k < wordCnt; k++)
        {
            if (st == codeType::one_hot) {
                const auto& sent = data[j];
                //one-hot编码并没有保存，需要临时从词频map和单词计数等信息中计算得到
                lhs = v1[k] == 0 ? 0 : sent.getWordRelFreq(word[k]) * sent.getWordCnt();
                rhs = code[k] == 0 ? 0 : s.getWordCnt() * s.getWordRelFreq(word[k]);
            }
            else//TF-IDF表示
                lhs = v1[k], rhs = code[k];
            if (lhs != 0 || rhs != 0) {//避免多余的计算，提高训练速度
                res[j].second += pow(abs(lhs - rhs), p);
            }
        }
        res[j].second = pow(res[j].second, 1 / p);
    }
    return res;
}
```

余弦相似度

代码基本一样，公式如下：

$$similarity = \frac{\sum_{i=1}^{n} v_1^{(i)} \cdot v_2^{(i)}}{\sqrt{\sum_{i=1}^{n} (v_1^{(i)})^2} \sqrt{\sum_{i=1}^{n} (v_2^{(i)})^2}}$$

需要注意的是25行的余弦距离=1-余弦相似度

同样的，在12~15行，需要获得one-hot编码，根据传入的编码参数决定如何计算距离。

```
1  vector<pair<sentence*, float>> dataSet::distCosine(const sentence& s,
   codeType st) {
2      vector<float>code = encode(s);
3      vector<pair<sentence*, float>> res(sentenceCnt);
4      for (int j = 0; j < sentenceCnt; j++)
5      {
6          const auto& v1 = TF_IDF[j];
7          res[j].first = &data[j];
8          res[j].second = 0;
9          float normlhs = 0, normrhs = 0, lhs, rhs;
10         for (int k = 0; k < wordCnt; k++)
11         {
12             if (st == codeType::one_hot) {
13                 const auto& sent = data[j];
14                 //one-hot编码并没有保存，需要临时从词频map和单词计数等信息中计算得到
15                 lhs = v1[k] == 0 ? 0 : sent.getWordRelFreq(word[k]) *
   sent.getWordCnt();
16                 rhs = code[k] == 0 ? 0 : s.getWordCnt() *
   s.getWordRelFreq(word[k]);
17             }
18             else//TF-IDF表示
19                 lhs = v1[k], rhs = code[k];
20             if (lhs != 0 || rhs != 0) {//避免多余的计算，提高训练速度
21                 res[j].second += lhs * rhs;
22                 normlhs += lhs * lhs;
23                 normrhs += rhs * rhs;
24             }
25         }
26         res[j].second = 1 - res[j].second / sqrt(normlhs * normrhs);
27     }
28     return res;
29 }
```

KNN类

我设计了继承dataSet的KNN类，负责调整参数和输出目标文档的结果

```
1  class KNN :public dataSet
2  {
3  public:
4      KNN() = default;
5      KNN(ifstream& infile) :dataSet(infile), k(1), p(1) {}
6      ~KNN() = default;
7      KNN& setK(int _k) { k = _k; return *this; }
8      KNN& setP(int _p) { p = _p; return *this; }
9      emotion classify(const sentence& target);
```

```
10
11  private:
12      int k;
13      int p;
14      codeType ct = codeType::one_hot;//默认使用one-hot编码
15  };
```

分类

分类函数是本算法的第二个核心，由于计算距离时已经得到了所有训练集文档的距离，排序后选出前k个，并计算众数最大的输出即可。

```
1   emotion classify(const sentence& target) {
2       vector<pair<sentence*, float>> vec;
3       if (p == 0) {//p=0时当做余弦距离
4           vec = distCosine(target, ct);
5       }
6       else//LP距离
7           vec = distLP(target, ct, p);
8       sort(vec.begin(), vec.end(), [](const pair<sentence*, float>& p1, const
    pair<sentence*, float>& p2) {return p1.second < p2.second; });
9       map<emotion, int>kElem;
10      for (int i = 0; i < k; i++)
11      {
12          kElem[vec[i].first->getEmotion()]++;
13      }
14      auto res = std::max_element
15      (
16          kElem.begin(), kElem.end(),
17          [](const pair<emotion, int>& p1, const pair<emotion, int>& p2) {
18              return p1.second < p2.second;
19          }
20      );
21      return res->first;
22  }
```

由于余弦距离和Lp距离无法统一，我将余弦距离设为p=0，在分类时根据p判断使用哪种距离计算函数。

10~13行统计了每种情感的文档个数，然后使用<algorithm>中的max_element函数找出第一大的元素并返回其emotion。

4.实验结果以及分析

（1）结果展示和分析

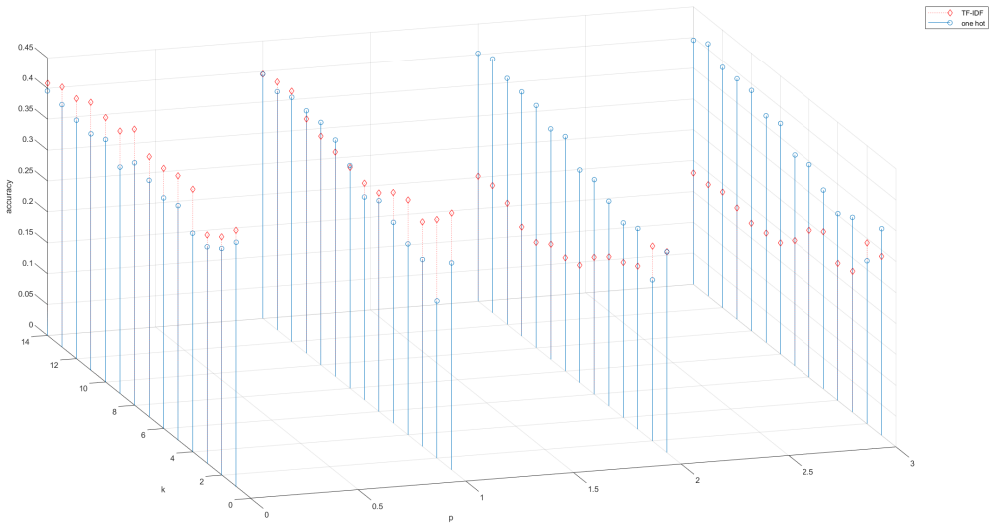
使用如下代码，计算k从1~15，p从1~4的结果，通过正确率评估模型的好坏。

```
1   KNN_CLASSIFICATION::KNN KNN_CLASSIFICATION::train(int max_k, int max_p)
2   {
3       cout << "KNN_CLASSIFICATION:training..." << endl;
4       string s;
5       ifstream train_set("lab1_data\\classification_dataset\\train_set.csv");
6       KNN k(train_set);
7       train_set.close();
8       //记录最优K和P
9       int bestK = 1, bestP = 1;
```

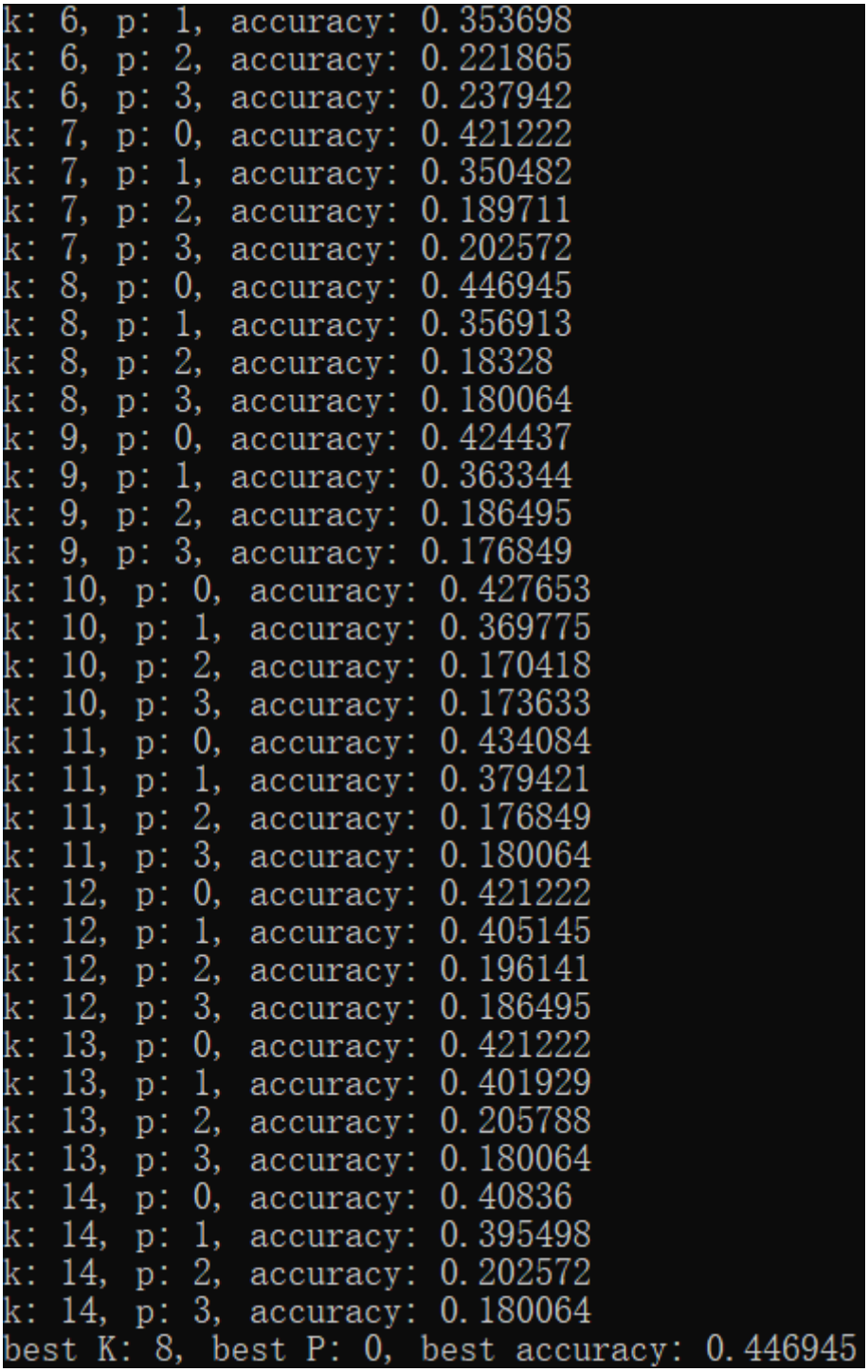
```
    ofstream train_res("classifi_train_res.csv");
    train_res << "k,p,accuracy" << endl;
    //记录最优准确率
    float bestCR = 0;
    k.setCodeType(codeType::tf_idf);
    for (int i = 1; i < max_k; i++)
    {
        k.setK(i);

        for (int j = 0; j < max_p; j++)
        {
            k.setP(j);
            ifstream validation_set("lab1_data\\classification_dataset\\validation_set.csv");
            int correct = 0, total = 0;
            getline(validation_set, s);
            while (getline(validation_set, s)) {
                auto sent = sentence(s);
                if (sent.getEmotion() == k.classify(sent)) {
                    correct++;
                }
                total++;
            }
            cout << "k = " << i << ", p = " << j << ", accuracy = " << (float)correct / total << endl;
            train_res << i << ',' << j << ',' << (float)correct / total << endl;
            validation_set.close();
            if (bestCR < (float)correct / total) {
                bestK = i, bestP = j, bestCR = (float)correct / total;
            }
        }
    }
    cout << "best K = " << bestK << "best P = " << bestP << ", best accuracy =" << bestCR << endl;
    k.setK(bestK).setP(bestP);
    return k;
}
```

1	senator carl krueger thinks ipods can kill you	sad
2	who is prince frederic von anhalt	joy
3	prestige has magic touch	joy
4	study female seals picky about mates	joy
5	no e book for harry potter vii	sad
6	blair apologises over friendly fire inquest	sad
7	vegetables may boost brain power in older adults	surprise
8	afghan forces retake town that was overrun by taliban	sad
9	skip the showers male sweat turns women on study says	surprise
10	made in china irks some burberry shoppers	joy

根据内容大致可以判断答案比较合理。



上图是准确率分别关于p，k和编码的stem图，可以看出，p=2（欧氏距离）时准确率整体较低，p=3时也比较低。



上图是训练的结果，最高准确率约为44.7%，相较于猜测的准确率20%有很大的提升，但是不到一半的准确率说明数据集的特征不够明显或是规模不够大，又或者算法不足以挖掘其中的特征。

（2）模型性能展示和分析

一开始我使用的是欧氏距离和one hot编码，效果不算太差，最好的准确率约为40%上下。课上新规定只能使用tf-idf，因此我添加了tf-idf编码方式，然而结果非常差，最高约为25%，经过调试，是距离公式上有错误，更正后结果也比one hot差不少，而且随k增加下降的速度非常快。然而和同学讨论后，我得知曼哈顿距离效果可能更好，尝试后确实如此，之后又尝试了余弦距离，效果更好。

	余弦距离	曼哈顿距离	欧氏距离	one hot	TF-IDF	最优K	准确率
初始			1	1		13	0.411576
优化一			1		1	1	0.324759
优化二		1			1	1	0.414791
优化三	1				1	8	0.446945
最优结果	1				1	8	0.446945

	A	B	C	D
1	k	p	codeType	accuracy
2	8	0	tf-idf	0.446945
3	11	0	tf-idf	0.434084
4	5	0	tf-idf	0.427653
5	10	0	tf-idf	0.427653
6	4	0	tf-idf	0.424437
7	9	0	tf-idf	0.424437
8	6	0	tf-idf	0.421222
9	7	0	tf-idf	0.421222
10	12	0	tf-idf	0.421222
11	13	0	tf-idf	0.421222
12	1	0	tf-idf	0.414791
13	1	1	tf-idf	0.414791
14	13	2	one-hot	0.411576
15	13	3	one-hot	0.40836
16	14	0	tf-idf	0.40836
17	12	1	tf-idf	0.405145
18	14	2	one-hot	0.401929
19	13	1	tf-idf	0.401929
20	12	2	one-hot	0.398714

观察上表可以发现，余弦距离配合tf-idf表示的效果非常好，可见对于当前数据集，余弦相似度是发掘特征的一个重要指标。

93	5	2	tf-idf	0.241158
94	6	3	tf-idf	0.237942
95	3	3	tf-idf	0.228296
96	4	3	tf-idf	0.221865
97	6	2	tf-idf	0.221865
98	13	2	tf-idf	0.205788
99	7	3	tf-idf	0.202572
100	14	2	tf-idf	0.202572
101	12	2	tf-idf	0.196141
102	7	2	tf-idf	0.189711
103	9	2	tf-idf	0.186495
104	12	3	tf-idf	0.186495
105	8	2	tf-idf	0.18328
106	8	3	tf-idf	0.180064
107	11	3	tf-idf	0.180064
108	13	3	tf-idf	0.180064
109	14	3	tf-idf	0.180064
110	9	3	tf-idf	0.176849
111	11	2	tf-idf	0.176849
112	10	3	tf-idf	0.173633
113	10	2	tf-idf	0.170418

然而最差的准确率也来自tf-idf表示，主要是p=2和p=3的情况。说明设计模型时不能想当然的使用欧氏距离，要多尝试其他的。

模型的训练速度比较快，3分钟左右能将各种模式的情况训练完成。

KNN回归

1.算法原理

KNN回归算法和KNN分类处理文档的方式是一样的，区别在于KNN回归针对的文档有多个属性。所以找到K个近邻后，获得结果的方式也不一样，KNN回归通过距离的倒数作为权值，将K个近邻的A属性的概率加权得到目标文档的A属性的概率。

比如计算目标文档的happy概率的公式如下：

$$P(test1\ is\ happy) = \frac{train1\ probility}{d(train1, test1)} + \frac{train2\ probility}{d(train2, test1)} + \frac{train3\ probility}{d(train3, test1)} + \cdots$$

要注意的是，这里因为权值之和不为1，得到的概率不满足和为1的性质，需要归一化，整个表达式除以权值之和即可。修改后的公式为：

$$P_{test1}(happy) = \frac{\sum_{i=1}^{n} \frac{p^i(happy)}{d(train_i, test_i)}}{\sum_{i=1}^{n} \frac{1}{d(train_i, test_i)}}$$

因为KNN回归的结果为几个概率值，无法直接判断是否准确，一般来说使用皮尔逊相关系数，公式如下：

$$COR(X,Y) = \frac{Cov(X,Y)}{\sigma_X \sigma_Y} = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2 \sum_{i=1}^{n}(Y_i - \bar{Y})^2}}$$

疑惑与思考

但是由于有六个情感维度，验证集和预测结果都是二维的矩阵，行为文档编号，列为情感维度。因此使用相关系数也有两种选择，一种是计算列之间的相关系数，取行平均，也就是ppt上的做法，一种是计算行之间的相关系数，取列平均，我一开始使用的是第一种，后来与同学讨论后改为了第二种。两种做法数值结果有较大差别，但是不同策略对其影响都相似。我认为先计算行相关系数是优先比较情感概率的相关程度，先计算列相关系数是优先比较文档结果的相关程度，有各自的道理，虽然我最后使用了ppt的做法，但是我依然觉得比较文档的相关度更为合理。

2.伪代码

```
function classify(sentence,p,k)
    input:想要分类的句子，距离计算参数，k
    output:包含每种情感概率的Array
    for train_sentence in train_set //train_set：数据集
        init dist[train_sentence]//dist的每个元素记录文档属性和距离
        for word in word_set //word_set:整个数据集的单词集合
            //encode函数计算新句子的编码，TF_IDF为矩阵
            dist[train_sentence] <- dist[train_sentence]+
(TF_IDF[train_sentence][word]-encode(sentence)[word])^p
        end for
        dist[train_sentence]=dist[train_sentence]^(1/p)
    end for
    sort dist by distance between train_sentence and sentence
    for emotion_prob in Array
        emotion_prob <- 0
        inv_dist_sum <- 0
        i<-1
        while i <= k
            emotion_prob <- emotion_prob + dist[i].emotion/dist[i].distance
            inv_dist_sum<- 1/dist[i].distance + inv_dist_sum
            i<-i+1
        end while
        emotion_prob/=inv_dist_sum
    end for
    return Array
```

3.代码截图

KNN回归与KNN分类的代码大致相同，基本只有读取和回归步骤需要修改。

sentence类

需要增加数组用于记录情感概率，还有相关系数的计算函数。

```
class sentence
{
private:
    array<float, type_of_emotion>emoPercent;//情感概率
    int wordCnt = 0;
    string str;
    map<string, float> termFreq;
    set<string>word;
    void load(const string& s);
    template<size_t N>
```

```
    float COR(const array<float, N>& X, const array<float, N>& Y);//计算相关系数

public:
    sentence() = default;
    explicit sentence(const string& s) {
        load(s);
    }
    ~sentence() = default;
    float correlate(const array<float, type_of_emotion>& res);//计算相关系数
    friend bool operator<(const sentence& s1, const sentence& s2) { return s1.str < s2.str; }
};
```

读取文档

读取文档中使用sscanf匹配所有概率。

```
void sentence::load(const string& s) {
    auto pos_of_comma = s.find_first_of(',');
    str = s.substr(0, pos_of_comma);
    const auto& dataStr = s.substr(pos_of_comma + 1);
    stringstream ss(str);
    string tmpStr;
    while (ss >> tmpStr) {
        wordCnt++;
        word.insert(tmpStr);
        termFreq[tmpStr]++;
    }
    sscanf_s(dataStr.c_str(), "%f,%f,%f,%f,%f,%f", &emoPercent[0], &emoPercent[1], &emoPercent[2], &emoPercent[3], &emoPercent[4], &emoPercent[5]);
    for (auto& str : termFreq) {
        str.second /= wordCnt;
    }
}
```

计算情感概率

通过和KNN分类一样的距离函数获得与所有训练样本之间的距离（7,10行），排序后，如果有完全匹配项，概率直接取前k个完全匹配项的均值，否则按照公式将距离的倒数作为权值对概率进行加权，最后归一化（38行）。

```
array<float, type_of_emotion> classify(const sentence& target) {
    array<float, type_of_emotion>res;
    vector<pair<sentence*, float>> vec;
    float inv_dist_sum = 0;
    res.fill(0);
    if (p == 0) {
        vec = distCosine(target, ct);
    }
    else
        vec = distLP(target, ct, p);
    sort(vec.begin(), vec.end(), [](const pair<sentence*, float>& p1, const pair<sentence*, float>& p2) {return p1.second < p2.second; });
    //完全匹配时需单独处理，否则权重为正无穷会出现异常
```

```
    bool perfect_match = (vec[0].second < 1e-4);
    if (perfect_match) {
        int zeroCnt = 0;
        for (int i = 0; i < k; i++)
        {
            if (vec[i].second != 0) continue;
            zeroCnt++;
            //取前k个（如果没有k个就取所有）完全匹配的结果的均值
            for (emotion e = emotion::anger; e < emotion::unknown; e = emotion((int)e + 1))
                res[(int)e] += vec[i].first->get_prob(e);
        }
        transform(res.begin(), res.end(), res.begin(), [zeroCnt](float f){return f / zeroCnt; });
    }
    else {
        for (int i = 0; i < k; i++)
            inv_dist_sum += 1 / vec[i].second;
        for (emotion e = emotion::anger; e < emotion::unknown; e = emotion(int(e) + 1))
        {
            for (int i = 0; i < k; i++)
                res[(int)e] += (vec[i].first->get_prob(e) / vec[i].second);
            res[(int)e] /= inv_dist_sum;
        }
    }
    return res;
}
```

计算相关系数

代入公式即可。

```
template<size_t N>
float COR(const array<float, N>& X, const array<float, N>& Y) {
    float sum = accumulate(X.begin(), X.end(), 0.0);
    float Xmean = sum / N;
    sum = accumulate(Y.begin(), Y.end(), 0.0);
    float Ymean = sum / N;
    float cov = 0, squareOfSigmaX = 0, squareOfSigmaY = 0;
    for (int i = 0; i < N; i++)
    {
        cov += (X[i] - Xmean) * (Y[i] - Ymean);
        squareOfSigmaX += pow(X[i] - Xmean, 2);
        squareOfSigmaY += pow(Y[i] - Ymean, 2);
    }
    if (squareOfSigmaX < 0 || squareOfSigmaY < 0)throw overflow_error("line 45:sqrt of minus number!");
    return cov / sqrt(squareOfSigmaX * squareOfSigmaY);
}
```

训练

训练分为三个层次，分别为编码方式，p和k，代码如下：

```
KNN_REGRESSION::KNN KNN_REGRESSION::train(int max_k, int max_p)
{
    cout << "*****************************************" << endl
        << "KNN_CLASSIFICATION:training..." << endl;
    string s;
    ifstream train_set("lab1_data\\regression_dataset\\train_set.csv");
    ofstream train_res("regress_train_res.csv");
    KNN_REGRESSION::KNN k(train_set);
    train_set.close();
    int K, P;
    float CR;
    codeType CT;
    for (int i = 0; i < 2; i++)
    {
        //用强制转换得到当前训练的编码类型
        codeType bestCt, currCT = (codeType)i;
        cout << "*****************************************" << endl
            << "KNN_REGRESSION:mode:" << codeTypeStr[(int)currCT] << endl;
        train_res << "k,p,codeType,accuracy" << endl;
        int bestK = 1, bestP = 0;
        float bestCR = 0;
        k.setCodeType(currCT);
        for (int i = 1; i < 30; i++)
        {
            k.setK(i);
            for (int j = 0; j < max_p; j++) {
                vector<array<float, 6>>vec1, vec2;
                k.setP(j);
                ifstream
validation_set("lab1_data\\regression_dataset\\validation_set.csv");
                getline(validation_set, s);
                while (getline(validation_set, s)) {
                    auto sent = KNN_REGRESSION::sentence(s);
                    vec1.push_back(sent.getEmotion());
                    vec2.push_back(k.classify(sent));
                }
                float corrRate = fullCOR(vec1, vec2);
                train_res << i << ',' << j << ',' <<
codeTypeStr[(int)currCT] << ',' << (float)corrRate << endl;
                cout << "k: " << i << ", p: " << j << ", accuracy: " <<
(float)corrRate << endl;
                validation_set.close();
                if (bestCR < corrRate) {
                    bestK = i, bestP = j, bestCR = (float)corrRate, bestCt =
currCT;
                    K = bestK, P = bestP, CT = bestCt, CR = bestCR;
                }
            }
        }
        cout << "best K: " << bestK << ",best P: " << bestP << ", best
corrRate: " << bestCR << endl;
    }
    cout << "*****************************************" << endl
```

```
        << "KNN_REGRESSION::train over. "
        << "best K: " << K << ",best P: " << P << ", best corrRate: " << CR
<< ", best codeType: " << codeTypeStr[(int)CT] << endl;
    k.setK(K).setP(P).setCodeType(CT);
    train_res.close();
    return k;
}
```

4.实验结果以及分析

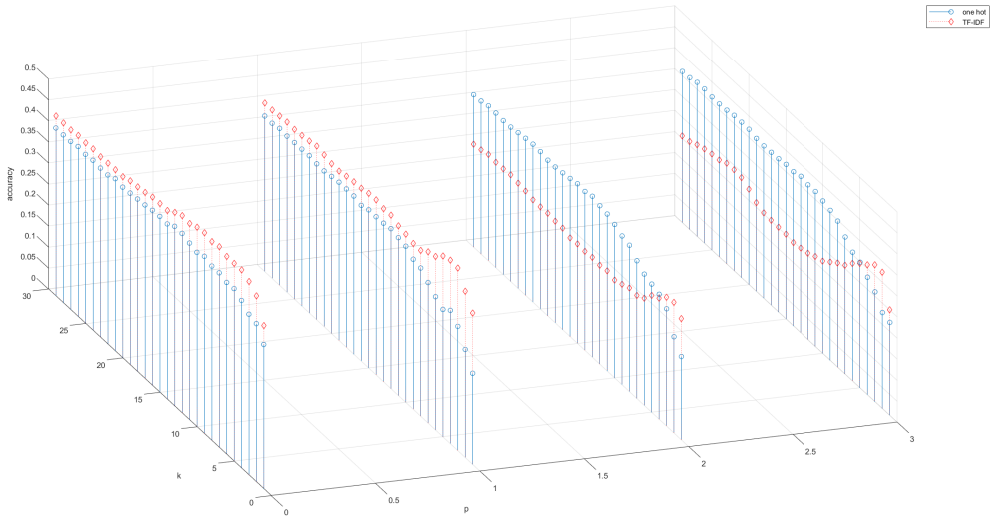
（1）结果展示和分析

	A	B	C	D	E	F	G	H
1	textid	anger	disgust	fear	joy	sad	surprise	total
2	1	0.113478	0.141474	0.063981	0.155305	0.225465	0.30027	0.999973
3	2	0.100936	0.131197	0.062886	0.232547	0.24034	0.232068	0.999974
4	3	0	0	0.037997	0.413885	0.184739	0.363408	1.000029
5	4	0.042409	0.068322	0.073468	0.384447	0.088188	0.343167	1
6	5	0.104148	0.088482	0.070066	0.21728	0.267018	0.253047	1.000041
7	6	0.109574	0.038051	0.266515	0.005556	0.491522	0.08878	0.999999
8	7	0	0	0.0926	0.4444	0	0.463	1
9	8	0.082915	0.006302	0.353648	0.052473	0.40859	0.096071	1
10	9	0.023979	0.13949	0.0268	0.123632	0.027261	0.658853	1.000015
11	10	0.094221	0.092516	0.154532	0.310706	0.079202	0.268796	0.999973
12	11	0.125702	0.057598	0.279339	0.22126	0.184077	0.131993	0.999969
13	12	0.088547	0.035952	0.100586	0.336004	0.12135	0.317535	0.999973
14	13	0.089481	0.052316	0.269943	0.22671	0.157237	0.204314	1.000001
15	14	0.115077	0.076997	0.182903	0.194546	0.205225	0.225255	1.000003
16	15	0.181494	0.138889	0.416987	0.013805	0.201347	0.047449	0.999971
17	16	0.144367	0.099121	0.357618	0.119509	0.169746	0.109612	0.999973
18	17	0.046649	0.090413	0.096319	0.443135	0.138982	0.184488	0.999986
19	18	0.099451	0.012552	0.181428	0.084696	0.245931	0.375942	1
20	19	0.062435	0.00951	0.157972	0.296556	0.230053	0.243489	1.000015
21	20	0.116985	0.094595	0.195131	0.307864	0.137898	0.147541	1.000014
22	21	0.152746	0.041546	0.235129	0.241525	0.154183	0.174871	1
23	22	0.077149	0.030067	0.10258	0.345925	0.08481	0.359458	0.999988
24	23	0.129356	0.022451	0.220147	0.194583	0.260556	0.172923	1.000016
25	24	0.1176	0	0.0588	0	0.1176	0.7059	0.9999
26	25	0.07481	0.053902	0.232923	0.331921	0.140694	0.165744	0.999995
27	26	0.054438	0.057444	0.195069	0.129442	0.309523	0.254084	1
28	27	0.111001	0.085507	0.093455	0.288158	0.203031	0.218862	1.000014
29	28	0.03198	0.047664	0.167587	0.19332	0.105543	0.453907	1
30	29	0.175262	0.129045	0.22931	0.10331	0.184913	0.178186	1.000026
31	30	0.087137	0.097787	0.170429	0.267305	0.246502	0.130811	0.99997
32	31	0.025844	0.039333	0.072165	0.526327	0.108211	0.228121	1
33	32	0.09739	0.089053	0.217055	0.224309	0.194379	0.177801	0.999986
34	33	0.049292	0.040981	0.092389	0.361216	0.185771	0.270365	1.000013
35	34	0.090362	0.050754	0.266351	0.293655	0.116625	0.182306	1.000053
36	35	0.028216	0.06064	0.055224	0.549199	0.07666	0.230085	1.000025
37	36	0.041921	0	0.03954	0.66342	0.120661	0.134459	1
38	37	0.086978	0.029768	0.328739	0.025301	0.223622	0.305523	0.999931

图片最右一列统计了各个概率之和，由于浮点数有误差，可以认为每一行的和为1，说明归一化正确。

```
k: 20, p: 0, accuracy: 0.358584
k: 20, p: 1, accuracy: 0.323979
k: 20, p: 2, accuracy: 0.254725
k: 20, p: 3, accuracy: 0.237356
k: 21, p: 0, accuracy: 0.355967
k: 21, p: 1, accuracy: 0.326657
k: 21, p: 2, accuracy: 0.247794
k: 21, p: 3, accuracy: 0.236179
k: 22, p: 0, accuracy: 0.351504
k: 22, p: 1, accuracy: 0.329698
k: 22, p: 2, accuracy: 0.2499
k: 22, p: 3, accuracy: 0.232435
k: 23, p: 0, accuracy: 0.348772
k: 23, p: 1, accuracy: 0.325475
k: 23, p: 2, accuracy: 0.25127
k: 23, p: 3, accuracy: 0.2277
k: 24, p: 0, accuracy: 0.353332
k: 24, p: 1, accuracy: 0.321761
k: 24, p: 2, accuracy: 0.253912
k: 24, p: 3, accuracy: 0.220402
k: 25, p: 0, accuracy: 0.349052
k: 25, p: 1, accuracy: 0.321898
k: 25, p: 2, accuracy: 0.252896
k: 25, p: 3, accuracy: 0.224139
k: 26, p: 0, accuracy: 0.348746
k: 26, p: 1, accuracy: 0.321049
k: 26, p: 2, accuracy: 0.249761
k: 26, p: 3, accuracy: 0.220981
k: 27, p: 0, accuracy: 0.34318
k: 27, p: 1, accuracy: 0.319043
k: 27, p: 2, accuracy: 0.252884
k: 27, p: 3, accuracy: 0.215759
k: 28, p: 0, accuracy: 0.346085
k: 28, p: 1, accuracy: 0.316776
k: 28, p: 2, accuracy: 0.244501
k: 28, p: 3, accuracy: 0.207867
k: 29, p: 0, accuracy: 0.342854
k: 29, p: 1, accuracy: 0.316459
k: 29, p: 2, accuracy: 0.238935
k: 29, p: 3, accuracy: 0.211089
best K: 5,best P: 0, best corrRate: 0.40819
KNN_CLASSIFICATION::train over. best K: 5,best P: 0, best corrRate: 0.40819, best codeType: tf-idf
```

上图是训练结果，仍然是余弦距离效果最好。



上图是相关系数分别关于p，k和编码的stem图，可以看出，p=2（欧氏距离）时相关系数整体较低，p=3时也比较低。

（2）模型性能展示和分析

回归和分类我是一起写的，因此模型的策略先后是一致的。令人惊讶的是tf-idf和欧氏距离一起使用时相关系数依然很低，和KNN分类的结果一致，优化的结果也基本一致。

	余弦距离	曼哈顿距离	欧氏距离	one hot	TF-IDF	最优K	相关系数
初始			1	1		12	0.34469
优化一			1		1	2	0.269407
优化二		1			1	5	0.34619
优化三	1				1	5	0.40819
最优结果	1				1	5	0.40819

根据KNN分类和回归的结果，可以得出结论，TF-IDF与欧氏距离的组合很难得到文档的特征，TF-IDF和余弦距离的组合则很好。

模型的训练速度比较快，3分钟左右能将各种模式的情况训练完成。我认为是因为C++偏底层，所以性能较好。