模式识别学习笔记

一、K-means算法

算法思想

无监督学习

数据集格式：$\{x^{(1)}, x^{(2)}, x^{(3)}, \cdots\cdots, x^{(m)}\}$

使用数据集：二维数据测试集 training_4k2_far.txt

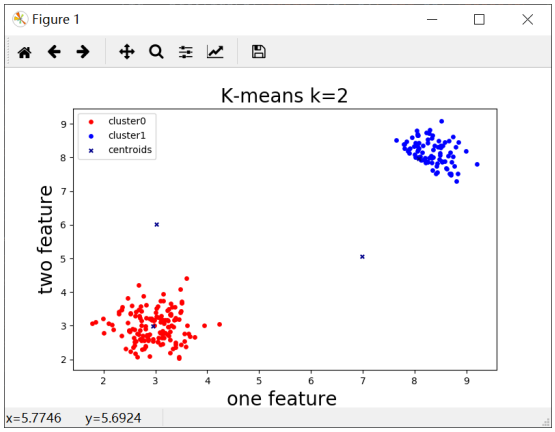
基于距离的聚类算法，采用距离作为相似性的评价指标，即认为两个对象的距离越近，其相似度就越大。K个初始聚类中心点的选取对聚类结果具有较大的影响，因为在该算法第一步中是随机地选取任意k个对象作为初始聚类中心，初始地代表一个簇。该算法在每次迭代中对数据集中剩余的每个对象，根据其与各个簇中心的距离赋给最近的簇。当考查完所有数据对象后，一次迭代运算完成，新的聚类中心被计算出来。计算步骤：

1. 在聚类中心初始化实现过程中采取在样本空间范围内随机生成K个聚类中心。
2. 对每个数据测量其到每个质心的距离，并把它归到最近的质心的类。
3. 重新计算已经得到的各个类的质心（所有数据到其质心的距离平均值作为新的质心）。
4. 迭代 2 ~ 3 步直至新的质心与原质心相等或小于指定阈值，算法结束。当所有样本所属的质心都不再变化时，算法收敛。

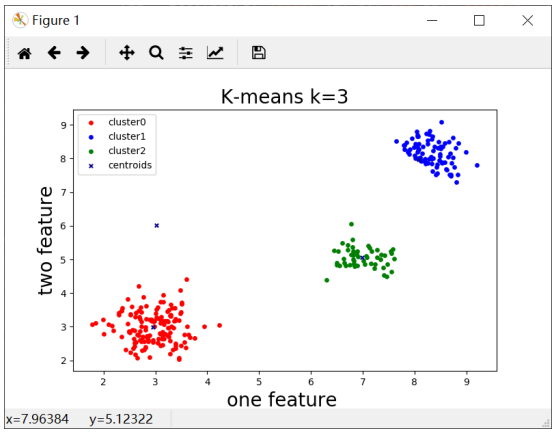
代码实现

```
def kmeans(dataSet, k, distMeans=distances, createCent=randCent):
    # 数据集有几个样本
    m = np.shape(dataSet)[0]
    # 初始化每一个样本的所属类别，那么就创建一个m×2的矩阵，第一列存储标签，后面一列存储距离的平方
    clusterAssment = np.mat(np.zeros((m, 2)))
    # 损失值
    # loss = np.mat(np.zeros((k,2))) 不用了这个该行删去
    # 创建 k 个随机质心
    centroids = createCent(dataSet, k)
    # 一个标志判断质心是否改变
    clusterChanged = True
    while clusterChanged:
        clusterChanged = False
        # 遍历每个样本
        for i in range(m):
            # 初始化最小距离为无群大、最小距离对应的索引为-1
            minDist = np.inf;
            minIndex = -1
            # 计算每一个样本和质心的距离
            for j in range(k):
                distJI = distMeans(centroids[j, :], dataSet[i, :])
                if distJI < minDist:
                    # 如果距离小于当前最小距离，则赋值，最小距离对应的索引为 j
                    minDist = distJI;
                    minIndex = j
            # 当前聚类结果中第 i 个样本的聚类结果发生变化，布尔值设为ture,继续聚类算法
            if clusterAssment[i, 0] != minIndex:
                clusterChanged = True
            clusterAssment[i, :] = minIndex, minDist ** 2
            loss = np.sum(clusterAssment[:, 1], axis=0)
        #    本来 i 是 minDist**2，用来记录平方误差
        print("分为 k 类，质心为————————————————\nk = ", k, centroids)
        print("每个样本的聚类标签[标签,损失]————————————\n", clusterAssment)
        print( "损失————————————\n", loss)
        # 遍历每一个质心
        for cent in range(k):
            # 将数据集合中所有属于当前质心类的样本通过条件过滤筛选出来
            ptsInClust = dataSet[np.nonzero(clusterAssment[:, 0].A == cent)[0]]
            # 计算这些数据的均值作为该类质心向量,更新质心
            centroids[cent, :] = np.mean(ptsInClust, axis=0)
            # lossIn = clusterAssment[np.nonzero(clusterAssment[:, 0].A == cent)[0]]
            # loss[cent,:] = np.sum(lossIn,axis= 0)
            # print "损失————————————", loss
    # 返回 k 个聚类，聚类的结果及误差
    return centroids, clusterAssment, loss
```

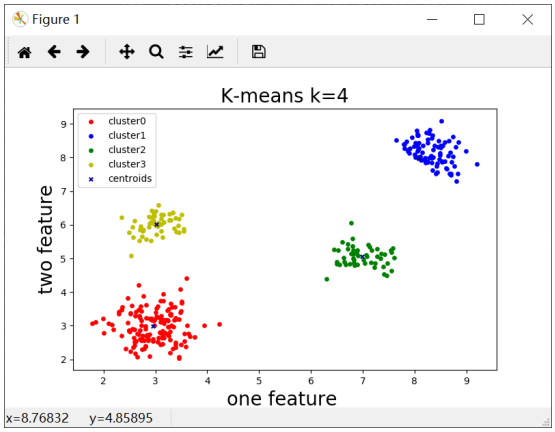
实现结果



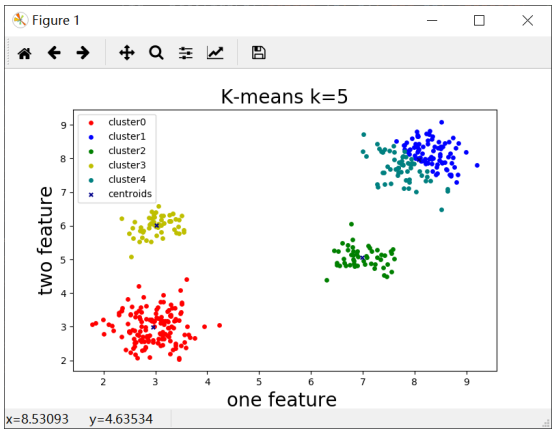
图一.聚两个类



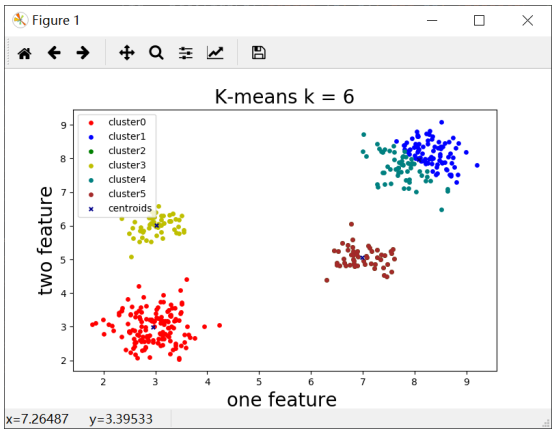
图二.聚三个类



图三.聚四个类



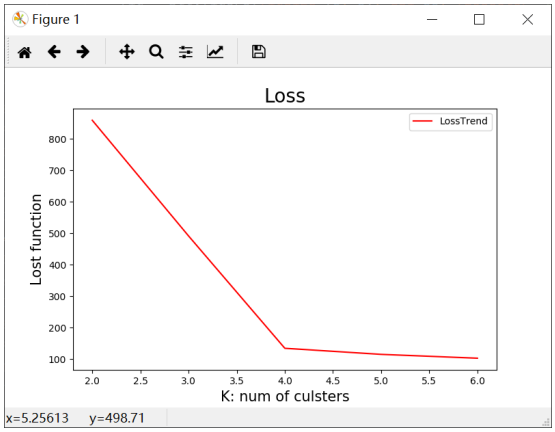
图四.聚五个类



图五.聚六个类

结果分析

损失函数：用来度量预测值和真实值之间误差大小的函数。有一种方法叫做"肘部法则"，分别计算在各种K值情况下，聚类算法最终的损失函数，绘制出随着K值变化损失函数变化的曲线：



图六.损失函数

例如图六，我们观察可以发现，在K=4时，损失函数急剧下降，然后在K=4之后损失函数逐渐趋于平稳，出现曲线类似于肘的形状，这时候我们就会选择K=4作为我们聚类的数目比较合适。

二、ISODATA算法

算法思想

1. 选择某些初始值。可选不同的参数指标，也可在迭代过程中人为修改，以将N个模式样本按指标分配到各个聚类中心中去。
2. 计算各类中诸样本的距离指标函数。
3. 按给定的要求，将前一次获得的聚类集进行分裂和合并处理，从而获得新的聚类中心。
4. 分裂处理。
5. 合并处理。
6. 重新进行迭代运算，计算各项指标，判断聚类结果是否符合要求。经过多次迭代后，若结果收敛，则运算结束。

算法过程

1. 输入N个模式样本：$\{x_i, i = 1, 2, \cdots, N\}$

```
/*第一步：输入N个模式样本{xi, i = 1, 2, …, N}
预选Nc个初始聚类中心*/
void isodata::init()
{
    clus.resize(K);
    set<int>aa;
    for (int i = 0; i < K; i++)
    {
        clus[i].center.resize(dim);
        int id = double(rand()) / RAND_MAX*dataset.size();
        while (aa.find(id) != aa.end())
        {
            id = double(rand()) / RAND_MAX*dataset.size();
        }
        aa.insert(id);
        for (int j = 0; j < dim; j++)
```

```
            clus[i].center[j] = dataset[id][j];
    }
}
```

2. 预选：N_c个初始聚类中心：$\{z_1, z_2, \ldots z_{N_c}\}$，它可以不等于所要求的聚类中心的数目，其初始位置可以从样本中任意选取。

   预选：K = 预期的聚类中心数目；

   θ_N = 每一聚类域中最少的样本数目，若少于此数即不作为一个独立的聚类；

   θ_S = 一个聚类域中样本距离分布的标准差；

   θ_c = 两个聚类中心间的最小距离，若小于此数，两个聚类需进行合并；

   L = 在一次迭代运算中可以合并的聚类中心的最多对数；

   I = 迭代运算的次数。

```
/*第二步：将N个模式样本分给最近的聚类Sj */
void isodata::assign()
{
    for (int i = 0; i < dataset.size(); i++)
    {
        double mindis = 100000000;
        int th = -1;
        for (int j = 0; j < clus.size(); j++)
        {
            double dis = distance(clus[j].center, i);
            if (dis < mindis)
            {
                mindis = dis;
                th = j;
            }
        }
        clus[th].clusterID.push_back(i);
    }
}
```

3. 将N个模式样本分给最近的聚类S_j，假若 $D_j = min\{||x - z_i||, i = 1, 2, \cdots N_c\}$ ，即$||x - z_j||$的距离最小，则$x \in S_j$。

```
/*第三步：如果Sj中的样本数目Sj<θN，
则取消该样本子集，此时Nc减去1*/
void isodata::check_thetaN()
{
    vector<int>toerase;
    for (int i = 0; i < clus.size(); i++)
    {
        if (clus[i].clusterID.size() < thetaN)
        {
            toerase.push_back(i);
            for (int j = 0; j < clus[i].clusterID.size(); j++)
            {
                double mindis = 10000000;
                int th = -1;
                for (int m = 0; m < clus.size(); m++)
                {
                    if (m == i)
                        continue;
                    double dis = distance(clus[m].center,
                        clus[i].clusterID[j]);
                    if (dis < mindis)
                    {
                        mindis = dis;
                        th = m;
                    }
                }
                clus[th].clusterID.push_back(
                    clus[i].clusterID[j]);
            }
            clus[i].clusterID.clear();
        }
    }
    for (vector<Cluster>::iterator it = clus.begin(); it != clus.end();)
    {
        if (it->clusterID.empty())
            it = clus.erase(it);
        else
            it++;
    }
}
```

```
void isodata::update_center(Cluster &aa)
{
    Centroid temp;
    temp.resize(dim);
    for (int j = 0; j < aa.clusterID.size(); j++)
    {
        for (int m = 0; m < dim; m++)
            temp[m] += dataset[aa.
            clusterID[j]][m];
    }
    for (int m = 0; m < dim; m++)
        temp[m] /= aa.clusterID.size();
    aa.center = temp;

}
```

4. 修正各聚类中心:

$$z_j = \frac{1}{N}\sum_{s\in S_i} x_i, \quad j = 1, 2, \ldots, N_c$$

```
/*第四步：修正各聚类中心*/
void isodata::update_centers()
{
    for (int i = 0; i < clus.size(); i++)
    {
        update_center(clus[i]);
    }
}
void isodata::update_sigma(Cluster&bb)
{
    bb.sigma.clear();
    bb.sigma.resize(dim);
    for (int j = 0; j < bb.clusterID.size(); j++)
        for (int m = 0; m < dim; m++)
            bb.sigma[m] += pow(bb.center[m] -
            dataset[bb.clusterID[j]][m], 2);
    for (int m = 0; m < dim; m++)
        bb.sigma[m] = sqrt(bb.sigma[m] /
        bb.clusterID.size());

}
```

5. 计算各聚类域Sj中模式样本与各聚类中心间的平均距离:

$$\overline{D_j} = \frac{1}{N}\sum_{s\in S_i} ||x - z_j||, \quad j = 1, 2, \ldots, N_c$$

6. 计算全部模式样本和其对应聚类中心的总平均距离

$$\overline{D} = \frac{1}{N}\sum_{j=1}^{N} N_j\overline{D_j}$$

```
/*五六步合并*/
void isodata::calmeandis()
{
    meandis = 0;
    for (int i = 0; i < clus.size(); i++)
    {
        double dis = 0;
        for (int j = 0; j < clus[i].
            clusterID.size(); j++)
        {
            dis += distance(clus[i].center,
                clus[i].clusterID[j]);
        }
        meandis += dis;
        clus[i].inner_meandis = dis /
            clus[i].clusterID.size();
    }
    meandis /= dataset.size();
}
```

7. 判别分裂、合并及迭代运算:

- 若迭代运算次数已达到I次，即最后一次迭代，则置$\theta_c = 0$，转至第十一步

- 若$N_c \leq \frac{K}{2}$,即聚类中心的数目小于或等于规定值的一半，则转至第八步，对已有聚类进行分裂处理。
- 若迭代运算的次数是偶数次，或$N_c \geq 2K$，不进行分裂处理，转至第十一步；否则（即既不是偶数次迭代，又不满足$N_c \geq 2K$），转至第八步，进行分裂处理。

```
/*第七步：判别下一步进行分裂或合并或迭代运算*/
void isodata::choose_nextstep()
{
    if (current_iter == maxiteration)
    {
        theta_c = 0;
        //goto step 11
        check_for_merge();
    }
    else if (clus.size() < K / 2)
    {
        check_for_split();
    }
    else if (current_iter % 2 == 0 ||
        clus.size() >= 2 * K)
    {
        //goto step 11
        check_for_merge();
    }
    else
    {
        check_for_split();
    }

}
```

8. 计算每个聚类中样本距离的标准差向量:

$$\sigma_j = (\sigma_1 j, \sigma_2 j, \ldots, \sigma_n j)^T$$

其中向量的各个分量为

$$\sigma_{ij} = \sqrt{\frac{1}{N_j}\sum_{k=1}^{N_j}(x_{ij} - z_{ij})^2}$$

式中，i = 1, 2, ..., n为样本特征向量的维数，j = 1, 2, ..., Nc为聚类数，Nj为Sj中的样本个数。

9. 求每一标准差向量$\{\sigma_j, j = 1, 2, \ldots, N_c\}$中的最大分量，以$\{\sigma_{jmax}, j = 1, 2, \ldots, N_c\}$代表。
10. 在任一最大分量集$\{\sigma_{jmax}, j = 1, 2, \ldots, N_c\}$中，若有$\sigma_{jmax} > \theta_S$，同时又满足如下两个条件之一:
    - $\overline{D_j} > \overline{D}$和$N_j > 2(\theta_N + 1)$，即$S_j$中样本总数超过规定值一倍以上;
    - $N_c \leq \frac{K}{2}$

    则将z_j分裂为两个新的聚类中心和，且N_c加1。 中对应于σ_{jmax}的分量加上$k\sigma_{jmax}$，其中；中对应于σ_{jmax}的分量减去$k\sigma_{jmax}$。
    如果本步骤完成了分裂运算，则转至第二步，否则继续。

```
/*第十步：分裂*/
void isodata::check_for_split()
{
    for (int i = 0; i < clus.size(); i++)
    {
        update_sigma(clus[i]);
    }
    while (true)
    {
        bool flag = false;
        for (int i = 0; i < clus.size(); i++)
        {
            for (int j = 0; j < dim; j++)
            {
                if (clus[i].sigma[j] > theta_s &&
                    (clus[i].inner_meandis>meandis&&
                    clus[i].clusterID.size()>
                    2 * (thetaN + 1) || clus.size() < K / 2))
                {
                    flag = true;
                    split(i);
                }
            }
        }
        if (!flag)
            break;
        else
            calmeandis();
    }
```

```
}
void isodata::split(const int kk)
{
    Cluster newcluster;
    newcluster.center.resize(dim);

    int th = -1;
    double maxval = 0;
    for (int i = 0; i < dim; i++)
    {
        if (clus[kk].sigma[i] > maxval)
        {
            maxval = clus[kk].sigma[i];
            th = i;
        }
    }
    for (int i = 0; i < dim; i++)
    {
        newcluster.center[i] = clus[kk].center[i];
    }
    newcluster.center[th] -= alpha*clus[kk].sigma[th];
    clus[kk].center[th] += alpha*clus[kk].sigma[th];
    for (int i = 0; i < clus[kk].clusterID.size(); i++)
    {
        double d1 = distance(clus[kk].center, clus[kk].clusterID[i]);
        double d2 = distance(newcluster.center, clus[kk].clusterID[i]);
        if (d2 < d1)
            newcluster.clusterID.push_back(clus[kk].clusterID[i]);
    }
    vector<int>cc; cc.reserve(clus[kk].clusterID.size());
    vector<int>aa;
    //insert_iterator<set<int, less<int> > >res_ins(aa, aa.begin());

    set_difference(clus[kk].clusterID.begin(), clus[kk].clusterID.end(),
        newcluster.clusterID.begin(), newcluster.clusterID.end(), inserter(aa, aa.begin()));//差集
    clus[kk].clusterID = aa;
    //应该更新meandis sigma。。。
    update_center(newcluster);
    update_sigma(newcluster);
    update_center(clus[kk]);
    update_sigma(clus[kk]);
    clus.push_back(newcluster);
}
```

11. 计算全部聚类中心的距离

$$D_{ij} = ||z_i - z_j||,\ i = 1, 2, \ldots, N_c - 1,\ j = i + 1, \ldots, N_c$$

12. 比较D_{ij}与θ_c的值，将$D_{ij} < \theta_c$的值按最小距离次序递增排列，即$\{D_{i1j1}, D_{i2j2}, \ldots, D_{iLjL}\}$，其中：
$D_{i1j1} < D_{i2j2} < \ldots < D_{iLjL}$
13. 将距离为D_{ikjk}的两个聚类中心Z_{ik}和Z_{jk}合并，得新的中心为：

$$z_k^* = \frac{1}{N_{ik} + N_{jk}}[N_{ik}z_{ik} + N_{jk}z_{jk}], k = 1, 2, \ldots, L$$

式中，被合并的两个聚类中心向量分别以其聚类域内的样本数加权，使Z_k^*为真正的平均向量。

```
/*第十一步：计算全部聚类中心的距离*/
/*第十二步：比较Dij 与θc 的值，将Dij <θc 的值按最小距离次序递增排列*/
/*第十三步：将距离为 的两个聚类中心 和 合并*/
void isodata::check_for_merge()
{
    vector<pair<pair<int, int>, double>>aa;
    for (int i = 0; i < clus.size(); i++)
    {
        for (int j = i + 1; j < clus.size(); j++)
        {
            double dis = distance(clus[i].center, clus[j].center);
            if (dis < theta_c)
            {
                pair<int, int>bb(i, j);
                aa.push_back(pair<pair<int, int>, double>(bb, dis));
            }
        }
    }
    // 利用函数对象实现升降排序
    struct CompNameEx
    {
        CompNameEx(bool asce) : asce_(asce)
        {}
```

```
        bool operator()(pair<pair<int, int>, double>const& pl, pair<pair<int, int>, double>const& pr)
        {
            return asce_ ? pl.second < pr.second : pr.second < pl.second; // 《Eff STL》条款21: 永远让比较函数对相等的值返回false
        }
    private:
        bool asce_;
    };
    sort(aa.begin(), aa.end(), CompNameEx(true));
    set<int>bb;
    int combinenus = 0;
    for (int i = 0; i < aa.size(); i++)
    {
        if (bb.find(aa[i].first.first) == bb.end()
            && bb.find(aa[i].first.second) == bb.end())
        {
            bb.insert(aa[i].first.first);
            bb.insert(aa[i].first.second);
            merge(aa[i].first.first, aa[i].first.second);
            combinenus++;
            if (combinenus >= maxcombine)
                break;
        }
    }
    for (vector<Cluster>::iterator it = clus.begin(); it != clus.end();)
    {
        if (it->clusterID.empty())
        {
            it = clus.erase(it);
        }
        else
            it++;
    }

}

void isodata::merge(const int k1, const int k2)//k1、k2顺序不能变
{
    for (int i = 0; i < dim; i++)
        clus[k1].center[i] = (clus[k1].center[i] * clus[k1].clusterID.size() +
        clus[k2].center[i] * clus[k2].clusterID.size()) /
        double(clus[k1].clusterID.size() + clus[k2].clusterID.size());
    //clus[k1].clusterID.insert(clus[k1].clusterID.end(),
    //  clus[k2].clusterID.begin(), clus[k2].clusterID.end());
    clus[k2].clusterID.clear();
}

int _tmain(int argc, _TCHAR* argv[])
{
    /*vector<int>aa;
    aa.push_back(1);
    aa.push_back(2);
    aa.push_back(3);
    aa.push_back(4);
    aa.push_back(5);
    for (vector<int>::iterator it = aa.begin(); it != aa.end(); )
    {
    cout << *it << endl;
    //it = aa.erase(it);
    //if (it == aa.end())
    //  break;
    if (*it > 3)
    {
    it = aa.insert(it+1, 2);
    cout << *it << endl;
    }
    else
    it++;
    }*/
    isodata iso;
    iso.generate_data();
    iso.set_paras();
    iso.apply();


    system("pause");
    return 0;
}
```

14. 如果是最后一次迭代运算，则算法结束；否则，若需要操作者改变输入参数，转至第一步；若输入参数不变，转至第二步。

实现结果

```
最终类别数 3 第一类 50 第二类 46 第三类 54
第 0 个聚类是
[array([5.1, 3.5, 1.4, 0.2]), array([4.9, 3. , 1.4, 0.2]), array([4.7, 3.2, 1.3, 0.2]), array([4.6, 3.1, 1.5,
0.2]), array([5. , 3.6, 1.4, 0.2]), array([5.4, 3.9, 1.7, 0.4]), array([4.6, 3.4, 1.4, 0.3]), array([5. , 3.4,
1.5, 0.2]), array([4.4, 2.9, 1.4, 0.2]), array([4.9, 3.1, 1.5, 0.1]), array([5.4, 3.7, 1.5, 0.2]), array([4.8,
3.4, 1.6, 0.2]), array([4.8, 3. , 1.4, 0.1]), array([4.3, 3. , 1.1, 0.1]), array([5.8, 4. , 1.2, 0.2]),
array([5.7, 4.4, 1.5, 0.4]), array([5.4, 3.9, 1.3, 0.4]), array([5.1, 3.5, 1.4, 0.3]), array([5.7, 3.8, 1.7,
0.3]), array([5.1, 3.8, 1.5, 0.3]), array([5.4, 3.4, 1.7, 0.2]), array([5.1, 3.7, 1.5, 0.4]), array([4.6, 3.6,
1. , 0.2]), array([5.1, 3.3, 1.7, 0.5]), array([4.8, 3.4, 1.9, 0.2]), array([5. , 3. , 1.6, 0.2]), array([5. ,
3.4, 1.6, 0.4]), array([5.2, 3.5, 1.5, 0.2]), array([5.2, 3.4, 1.4, 0.2]), array([4.7, 3.2, 1.6, 0.2]),
array([4.8, 3.1, 1.6, 0.2]), array([5.4, 3.4, 1.5, 0.4]), array([5.2, 4.1, 1.5, 0.1]), array([5.5, 4.2, 1.4,
0.2]), array([4.9, 3.1, 1.5, 0.1]), array([5. , 3.2, 1.2, 0.2]), array([5.5, 3.5, 1.3, 0.2]), array([4.9, 3.1,
1.5, 0.1]), array([4.4, 3. , 1.3, 0.2]), array([5.1, 3.4, 1.5, 0.2]), array([5. , 3.5, 1.3, 0.3]), array([4.5,
2.3, 1.3, 0.3]), array([4.4, 3.2, 1.3, 0.2]), array([5. , 3.5, 1.6, 0.6]), array([5.1, 3.8, 1.9, 0.4]),
array([4.8, 3. , 1.4, 0.3]), array([5.1, 3.8, 1.6, 0.2]), array([4.6, 3.2, 1.4, 0.2]), array([5.3, 3.7, 1.5,
0.2]), array([5. , 3.3, 1.4, 0.2])]
第 1 个聚类是
[array([6.4, 3.2, 4.5, 1.5]), array([5.5, 2.3, 4. , 1.3]), array([6.5, 2.8, 4.6, 1.5]), array([5.7, 2.8, 4.5,
1.3]), array([4.9, 2.4, 3.3, 1. ]), array([6.6, 2.9, 4.6, 1.3]), array([5.2, 2.7, 3.9, 1.4]), array([5. , 2. ,
3.5, 1. ]), array([5.9, 3. , 4.2, 1.5]), array([6. , 2.2, 4. , 1. ]), array([6.1, 2.9, 4.7, 1.4]), array([5.6,
2.9, 3.6, 1.3]), array([6.7, 3.1, 4.4, 1.4]), array([5.6, 3. , 4.5, 1.5]), array([5.8, 2.7, 4.1, 1. ]),
array([6.2, 2.2, 4.5, 1.5]), array([5.6, 2.5, 3.9, 1.1]), array([6.1, 2.8, 4. , 1.3]), array([6.3, 2.5, 4.9,
1.5]), array([6.1, 2.8, 4.7, 1.2]), array([6.4, 2.9, 4.3, 1.3]), array([6.6, 3. , 4.4, 1.4]), array([6. , 2.9,
4.5, 1.5]), array([5.7, 2.6, 3.5, 1. ]), array([5.5, 2.4, 3.8, 1.1]), array([5.5, 2.4, 3.7, 1. ]), array([5.8,
2.7, 3.9, 1.2]), array([5.4, 3. , 4.5, 1.5]), array([6. , 3.4, 4.5, 1.6]), array([6.3, 2.3, 4.4, 1.3]),
array([5.6, 3. , 4.1, 1.3]), array([5.5, 2.5, 4. , 1.3]), array([5.5, 2.6, 4.4, 1.2]), array([6.1, 3. , 4.6,
1.4]), array([5.8, 2.6, 4. , 1.2]), array([5. , 2.3, 3.3, 1. ]), array([5.6, 2.7, 4.2, 1.3]), array([5.7, 3. ,
4.2, 1.2]), array([5.7, 2.9, 4.2, 1.3]), array([6.2, 2.9, 4.3, 1.3]), array([5.1, 2.5, 3. , 1.1]), array([5.7,
2.8, 4.1, 1.3]), array([4.9, 2.5, 4.5, 1.7]), array([5.7, 2.5, 5. , 2. ]), array([6. , 2.2, 5. , 1.5]),
array([5.6, 2.8, 4.9, 2. ])]
第 2 个聚类是
[array([7. , 3.2, 4.7, 1.4]), array([6.9, 3.1, 4.9, 1.5]), array([6.3, 3.3, 4.7, 1.6]), array([5.9, 3.2, 4.8,
1.8]), array([6.8, 2.8, 4.8, 1.4]), array([6.7, 3. , 5. , 1.7]), array([6. , 2.7, 5.1, 1.6]), array([6.7, 3.1,
4.7, 1.5]), array([6.3, 3.3, 6. , 2.5]), array([5.8, 2.7, 5.1, 1.9]), array([7.1, 3. , 5.9, 2.1]), array([6.3,
2.9, 5.6, 1.8]), array([6.5, 3. , 5.8, 2.2]), array([7.6, 3. , 6.6, 2.1]), array([7.3, 2.9, 6.3, 1.8]),
array([6.7, 2.5, 5.8, 1.8]), array([7.2, 3.6, 6.1, 2.5]), array([6.5, 3.2, 5.1, 2. ]), array([6.4, 2.7, 5.3,
1.9]), array([6.8, 3. , 5.5, 2.1]), array([5.8, 2.8, 5.1, 2.4]), array([6.4, 3.2, 5.3, 2.3]), array([6.5, 3. ,
5.5, 1.8]), array([7.7, 3.8, 6.7, 2.2]), array([7.7, 2.6, 6.9, 2.3]), array([6.9, 3.2, 5.7, 2.3]), array([7.7,
2.8, 6.7, 2. ]), array([6.3, 2.7, 4.9, 1.8]), array([6.7, 3.3, 5.7, 2.1]), array([7.2, 3.2, 6. , 1.8]),
array([6.2, 2.8, 4.8, 1.8]), array([6.1, 3. , 4.9, 1.8]), array([6.4, 2.8, 5.6, 2.1]), array([7.2, 3. , 5.8,
1.6]), array([7.4, 2.8, 6.1, 1.9]), array([7.9, 3.8, 6.4, 2. ]), array([6.4, 2.8, 5.6, 2.2]), array([6.3, 2.8,
5.1, 1.5]), array([6.1, 2.6, 5.6, 1.4]), array([7.7, 3. , 6.1, 2.3]), array([6.3, 3.4, 5.6, 2.4]), array([6.4,
3.1, 5.5, 1.8]), array([6. , 3. , 4.8, 1.8]), array([6.9, 3.1, 5.4, 2.1]), array([6.7, 3.1, 5.6, 2.4]),
array([6.9, 3.1, 5.1, 2.3]), array([5.8, 2.7, 5.1, 1.9]), array([6.8, 3.2, 5.9, 2.3]), array([6.7, 3.3, 5.7,
2.5]), array([6.7, 3. , 5.2, 2.3]), array([6.3, 2.5, 5. , 1.9]), array([6.5, 3. , 5.2, 2. ]), array([6.2, 3.4,
5.4, 2.3]), array([5.9, 3. , 5.1, 1.8])]
```

结果分析

ISODATA算法是在k-均值算法的基础上，增加对聚类结果的“合并”和“分裂”两个操作，并设定算法运行控制参数的一种聚类算法。

三、GMM算法

算法模型

每个 GMM 由 K 个 Gaussian 分布组成，每个 Gaussian 称为一个“Component”，这些 Component 线性加成在一起就组成了 GMM 的概率密度函数：

$$
\begin{aligned}
p(x) &= \sum_{k=1}^{K} p(k)p(x|k) \\
&= \sum_{k=1}^{K} \pi_k N(x|\mu_k, \Sigma_k)
\end{aligned}
$$

根据上面的式子，如果我们要从 GMM 的分布中随机地取一个点的话，实际上可以分为两步：首先随机地在这 K个Gaussian Component 之中选一个，每个 Component 被选中的概率实际上就是它的系数 pi(k) ，选中了 Component 之后，再单独地考虑从这个 Component 的分布中选取一个点就可以了——这里已经回到了普通的 Gaussian 分布，转化为了已知的问题。
当有了数据，假定它们是由 GMM 生成出来的，那么我们只要根据数据推出 GMM 的概率分布来就可以了，然后 GMM 的 K 个 Component 实际上就对应了 K 个 cluster 了。根据数据来推算概率密度通常被称作 density estimation ，特别地，当我们在已知（或假定）了概率密度函数的形式，而要估计其中的参数的过程被称作“参数估计”。

算法步骤

1. 估计数据由每个 Component 生成的概率（并不是每个 Component 被选中的概率）：对于每个数据x_i 来说，它由第k个 Component 生成的概率为

$$\gamma(i,k)=\frac{\pi_k N(x_i|\mu_k,\Sigma_k)}{\sum_{j=1}^{K}\pi_j N(x_i|\mu_i,\Sigma_j)}$$

其中$N（xi|\mu k,\Sigma k）$就是后验概率

$$N(x|\mu,\Sigma)=\frac{1}{(2\pi)^{D/2}}\frac{1}{|\Sigma|^{1/2}}\exp\{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\}$$

2. 通过极大似然估计可以通过求到令参数=0得到参数p_μ，p_Σ的值

$$\mu_k=\frac{1}{N_k}\sum_{i=1}^{N}\gamma(i,k)(x_i-\mu_k)(x_i-\mu_k)^T$$

其中$N_k=\sum_N^{i=1}\gamma(i,k)$，并且$\pi_k$也顺理成章可估计为$\frac{N_k}{N}$

3. 重复迭代前面两步，直到似然函数的值收敛为止。

算法应用

- **问题描述**

  一片草地上有一群兔子，几个猎人在这里打猎，这时一声枪响过去，一只兔子倒下，估计这只兔子是哪个猎人打死的。

- **解决思路**

  1.传入的数据集（几只兔子），

  2.传出的数据集（谁杀死了哪几家兔子）

  3.要分成几个数据集（几个猎人）

  对于的程序如下：

  创建一个算法空间

  Ptrem_model = EM::create();

  设置有要分成的数据集合

  em_model->setClustersNumber(numCluster);

  设置算法类型

  ```
  em_model->setCovarianceMatrixType(EM::COV_MAT_SPHERICAL);
  ```

  设置退出循环的条件

  ```
  em_model->setTermCriteria(TermCriteria(TermCriteria::EPS + TermCriteria::COUNT, 100, 0.1));
  ```

  ermCriteria(TermCriteria::EPS + TermCriteria::COUNT, 100, 0.1);

  TermCriteria::EPS　　表示: 迭代到阈值终止

  TermCriteria::COUNT 表示: 最大迭代次数终止

  100：最大迭代次数

  0.1：结果的精确性

  typedef struct CvTermCriteria
  {
  int　type;　/* CV_TERMCRIT_ITER 和CV_TERMCRIT_EPS二值之一，或者二者的组合 /
  int　max_iter; / 最大迭代次数 /*
  double epsilon; / 结果的精确性 */*
  }

  放入数据进行计算

  em_model->trainEM(points, noArray(), Result, noArray());

  points:输入的数据集合（兔子）

  Result：结果，输出的集合

  最后画点显示数据

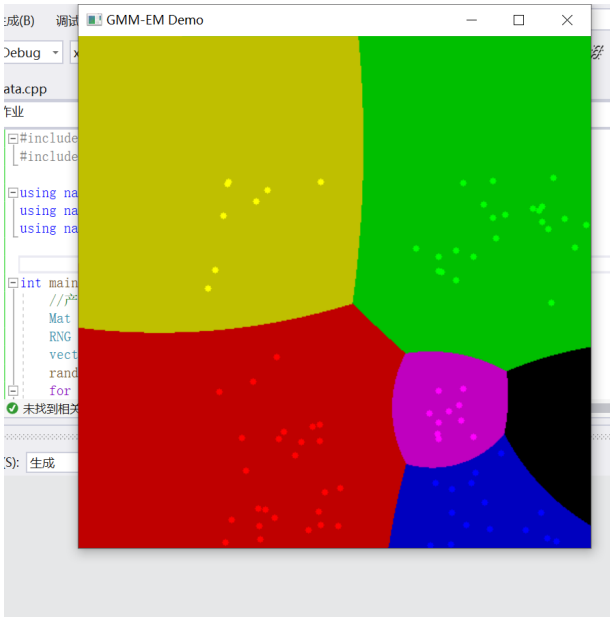
- **代码**

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace cv;
using namespace cv::ml;
using namespace std;


int main(int argc, char** argv) {
    //产生随机的点集
```

```
    Mat img(500, 500, CV_8UC3);
    RNG rng(12345);
    vector<int> vi = { 1,2,3,4,5,6 };
    randShuffle(vi, 1, &rng);
    for (size_t i = 0; i < vi.size(); i++)
    {
        cout << vi[i] << endl;
    }

    Scalar colorTab[] = {
        Scalar(0, 0, 255),
        Scalar(0, 255, 0),
        Scalar(255, 0, 0),
        Scalar(0, 255, 255),
        Scalar(255, 0, 255)
    };

    int numCluster = rng.uniform(2, 10);
    printf("number of clusters : %d\n", numCluster);

    int sampleCount = rng.uniform(5, 100);
    //1, Mat(row,col,type) 定义一个sampleCount行 , 2列的数据
    Mat points(sampleCount, 2, CV_32FC1, Scalar(10, 1));

    Mat labels;
    Mat centers;
    cout << points << endl;
    // 生成随机数
    for (int k = 0; k < numCluster; k++) {
        Point center;
        center.x = rng.uniform(0, img.cols);
        center.y = rng.uniform(0, img.rows);
        //为矩阵的指定行区间创建一个矩阵头 参数1,从0开始的行间距索引;  参数2,终止索引（把points的第n,到n+1行数据放到这里）
        Mat pointChunk = points.rowRange(k * sampleCount / numCluster,
            k == numCluster - 1 ? sampleCount : (k + 1) * sampleCount / numCluster);

        /*用随机数填充矩阵 ,
        InputOutputArray                    输入输出矩阵，最多支持4通道，超过4通道先用reshape()改变结构
        int distType                          UNIFORM 或 NORMAL，表示均匀分布和高斯分布
        InputArray a                        disType是UNIFORM,a表示为下限
        InputArray b                        disType是UNIFORM,b表示为上限
        bool saturateRange=false      只针对均匀分布有效。当为真的时候，会先把产生随机数的范围变换到数据类型的范围，再产生随机数;
                                            如果为假，会先产生随机数，再进行截断到数据类型的有效区间。请看以下fillM1和fillM2的例子并观察结果 */
        rng.fill(pointChunk, RNG::NORMAL, Scalar(center.x, center.y), Scalar(img.cols * 0.05, img.rows * 0.05));

    }

    randShuffle(points, 1, &rng);


    Ptr<EM> em_model = EM::create();
    em_model->setClustersNumber(numCluster);
    em_model->setCovarianceMatrixType(EM::COV_MAT_SPHERICAL);
    em_model->setTermCriteria(TermCriteria(TermCriteria::EPS + TermCriteria::COUNT, 100, 0.1));
    em_model->trainEM(points, noArray(), labels, noArray());
    //em_model 模型
    // classify every image pixels(像素)
    Mat sample(1, 2, CV_32FC1);
    for (int row = 0; row < img.rows; row++) {
        for (int col = 0; col < img.cols; col++) {
            sample.at<float>(0) = (float)col;
            sample.at<float>(1) = (float)row;
            //cvRound(): 四舍五入;
            // predict2: EM预言 sample:位置
            int response = cvRound(em_model->predict2(sample, noArray())[1]);
            /*typedef struct Scalar
            {
                double val[4];
            }Scalar*/
            Scalar c = colorTab[response];
            circle(img, Point(col, row), 1, c * 0.75, -1);
        }
    }

    // draw the clusters
    for (int i = 0; i < sampleCount; i++) {
        Point p(cvRound(points.at<float>(i, 0)), points.at<float>(i, 1));
        circle(img, p, 3, colorTab[labels.at<int>(i)], -1);
```

```
    }

    imshow("GMM-EM Demo", img);

    waitKey(0);
    return 0;
}
```

- **运行结果**



图七.猎人打兔子

点代表兔子，色块代表猎人，被色块覆盖的点代表被哪个猎人打死的兔子。

GM算法建立在最大似然估计的基础上， 最大似然估计，只是一种概率论在统计学的应用，它是参数估计的方法之一：已知某个参数能使这个样本出现的概率最大，我们当然不会再去选择其他小概率的样本，所以干脆就把这个参数作为估计的真实值。

四、基于高斯混合模型(GMM)的火焰检测算法

GMM模型

高斯分布：（Gaussian distribution）有时也被称为正态分布（normal distribution），是一种在自然界大量的存在的、最为常见的分布形式。

高斯分布密度函数：

$$f(x|\mu,\sigma^2)=\frac{1}{\sqrt{2\sigma^2\pi}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

我们通过观察采样的概率值和模型概率值的接近程度，来判断一个模型是否拟合良好。然后我们通过调整模型以让新模型更适配采样的概率值。反复迭代这个过程很多次，直到两个概率值非常接近时，我们停止更新并完成模型训练。

现在我们要将这个过程用算法来实现，所使用的方法是模型生成的数据来决定似然值，即通过模型来计算数据的期望值。通过更新参数μ和σ来让期望值最大化。这个过程可以不断迭代直到两次迭代中生成的参数变化非常小为止。在这里的高斯模型中，我们需要同时更新两个参数：分布的均值和标准差

高斯混合模型是对高斯模型进行简单的扩展，GMM使用多个高斯分布的组合来刻画数据分布。

高斯混合模型：

$$p(x)=\sum_{i=1}^{K}\phi_i\frac{1}{\sqrt{s\sigma_i^2\pi}}e^{-\frac{(x-\mu_i)^2}{2\sigma_i^2}}$$

问题描述

近几十年来，由于火灾造成的巨大损失，火灾 探测技术越来越受到人们的关注。为了减少伤害 和经济损失，通常要求火灾探测系统提供快速准确的警报。由于传统火灾监测技术的局限性， 基于视频的火灾探测方法越来越流行。与基于传感器的方法不同，基于计算机视觉的方法主 要利用从光学视频中提取的信息。现有火焰检测 技术除采用火焰的颜色和动态特性外，还采用纹理、形状和其他特征。这些特征与机器学习类 型的分类器一起广泛使用，从而实现高效的火灾检测。

根据颜色为每个像素分配一个概率，描述其成为火焰一部分的可能性。然后根据每个像素的光流大小得到显著性图。将显著性图与颜色模型的结果相结合，用两 个独立的实验阈值确定候选火焰像素。最后根据火焰的闪烁特性，对候选像素进行进一步处理，并 进行逐帧判定。

火焰颜色模型

DP 通过推断数据分配到簇的后验概率来估 计 GMM 分量的个数，假设存在无限个潜在簇，但 只有有限个簇用于生成观测数据。

- DP和中国餐馆模型

  Dirichlet过程：DP研究可变观测值问题 。 每个观测值被表示为 xi ，并且由参数 θ_i（x_i 和 θ_i 可 以是标量或矢量）的分布生成。不同的 θi是可交 换的，并且可能没有不同的值。参数 θi由先验分布G生成。因此，有如下模型:

  $$\begin{cases}\theta_i|G \sim G \\ xi|\theta_i \sim F(\theta_i)\end{cases}$$

  其中，$F(\theta_i)$是 xi给出的 θi分布。假定每个参数 θi 在给定分布G的条件下是独立的。

  给定可测空间和空间上的概率测度G0，DP被 定义为概率测度G在空间上的分布。它满足以下 条 件 ： 对 于 空 间 的 任 何 有 限 的 可 测 量 分 区 $(A_1, \ldots, A_r)$，$(G(A_1), \ldots, G(A_r))$遵循 Dirichlet 分布， 参数为$(\alpha_0 G_0(A_1), \ldots, \alpha_0 G_0(A_r))$，其中α0是正实参数，$(A_1), \ldots, G(A_r) \sim Dir(\alpha_0 G_0(A_1), \ldots, \alpha_0 G_0(A_r))$

  当 G 遵循 DP 时，将其表示为$G \sim DP(\alpha_0, G_0)$， 其参数为α_0，基本分布为G_0。

  中国餐馆过程：CRP是一个参数为α_0的整 数分区上的分布。这是 DP 的另一个视角。对 于G~DP(α_0, G_0)，CRP集中于G。 考虑一家无限大的中餐馆，有无限的桌子，每 一张桌子可以为无限多的顾客服务。顾客序列 θ1, θ2,...（一个从G中提取的可交换随机变量序列的隐喻）走进餐厅，选择餐桌坐下。第 i个客户 θi 既可以坐在现有的桌子上，也可以选择一个新的， 遵循下面式子给出的分布：

  $$p(c_i|c_1, \ldots, c_{i-1}) = \begin{cases}\frac{m_{k,-i}}{i-1+\alpha_0}, 在一张已有的餐桌k \\ \frac{\alpha_0}{i-1+\alpha_0}， 在一张新餐桌\end{cases}$$

  其中，ci是一个指示变量，指定客户θi位于哪个餐 桌上，$m_k, -i$是餐桌k中已有的客户数量（不包括θ）。i 在所有客户都坐好之后，得到了这些客户（变 量 θ1, θ2,...）的划分方案。当与 DP 相关时，同一 餐桌上的客户（随机变量）共享从基分布 G0 提取 的参数向量。餐桌的相关参数的离散值用 ϕ = { ϕ1,ϕ2,...}表示。因此，CRP 具有自然的聚类属 性，CRP 中的餐桌对应于集群。聚类数受浓度参 数α0 的影响，因为它决定了顾客相对于已经在餐 馆的顾客选择新餐桌的可能性。另外，由于 θ1, θ2,... 是可互换的，每个客户都可以被视为最 后一个客户。

- 基于DPGMM的火焰颜色模型

  训练一个 GMM 来模拟 RGB空间中火焰颜色 的分布。由于训练数据是从不同光照下的图像中 提取出来的，因此基于 DPGMM 的火焰颜色模型 对不同的光照条件具有较强的鲁棒性。表示火焰 像素 i 的颜色矢量作为$x_i = [R_i, G_i, B_i]^T$ 。那么，可以得到式子：

  $$p(x_i|\mu, \Sigma) = \sum_{k=1}^{K} \omega_k N(x_i|\mu_k, \Sigma_k)$$

  根据混合模型理论，每个 xi 是通过首先选 择由 ci 索引的成分，根据 $\omega = [\omega_1, \ldots, \omega_K]$分配的。 之后，观测到的xi是由选择的高斯分量生成的，参 数为$\theta_i = \phi_{c_i} \triangleq \mu_{c_i}, \sum_{c_i}$。然而，分布权重w并非仅对已知观测值有效，因此假设θi是根据DP分布的。 因此，生成模型:

  $$G \sim DP(\alpha_0, G_0)$$

  $$\theta_i|G \sim G$$

  $$x_i|\theta_i \sim N(\mu_{c_i}, \Sigma_{c_i})$$

  然而，在只有训练数据$X = x_1, \ldots, x_N$可用的 情况下，GMM参数和数据分配都是未知的。这里 采用折叠Gibbs抽样来获得数据X的赋值，并据 此估计其他参数。当 θ 按 G 分布时，CRP 诱导了 以$\{c_1, \ldots, c_{i-1}\}$为条件的 ci 分布。因此，后面部分如下式子所示：

  $$p(c_i = k|c_{-i}, X, \alpha_0, G_0) \propto p(c_i = k|c_{-i}, \alpha_0) \cdot p(x_i|X_{-i}, c_i = k, c_{-i}, G_0)$$

  其中，k ∈ {1,...,t, k* }和 t 表示被占用餐桌的数目， 而k* 表示选择一个新桌子。此外，X-i和c-i被称为 除xi之外的训练数据及其分配。

  为了根据其他观测值的赋值计算 ci = k 的条 件分布，根据可交换性，θi 被视为最后一个顾客。

  如果θi位于现有的桌子上，则上式可以表示成

  $$\begin{aligned} p(x_i|X_{-i}, c_i = k, c_{-i}, G_0) &= p(x_i|X_{k,-i}, G_0) \\ &= \frac{p(x_i, X_{k,-i}|G_0)}{p(X_{k,-i}|G_0)} \\ &= \frac{\int p(x_i|\theta_k)[\prod_{j \neq i, c_j = k} p(x_j|\theta_k)]G_0(\theta_k)d\theta_k}{\int[\prod_{j \neq i, c_j = k} p(x_j|\theta_k)]G_0(\theta_k)d\theta_k} \end{aligned}$$

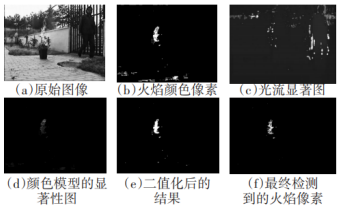
  其中，$X_k, -i = \{x \ : \ j \neq i, c_j = k\}$表示分配给餐桌k的 其他客户，不包括xi 。

  同样，如果第 i—th 个客户选择一张新桌子， 则：

  $$p(x_i|X_{-i,c_i} = k^*, c_{-i}, G_0) = p(x_i|G_0) = \int p(x_i|\theta)G_0(\theta)d\theta$$

  基于上式所描述的 CRP 的折叠 Gibbs 抽样，在收敛后得到分配方案。 通过训练的颜色模型，每个像素被分配一个 概率，根据颜色描述它是火焰的一部分的可能性 有多大。火焰像素将获得更高的概率，而非火焰 区域则可能具有较低的概率，并精确估计火焰颜 色分布。给定一个适当选择的阈值，得到几个候 选像素以供进一步处理。

实验结果



图八.火焰识别结果

结果分析

基于 DPGMM 的火焰颜色模型，结合显著性分析和时域小波变换进行火焰检测。颜色模型采用 GMM 来逼近火焰的颜色分布，GMM 的分量数由 DP 从训练数据中学习得到。该方法避免了经验高斯数不合适所引起的偏差，从而实现了对 GMM 其它参数的更精确估计，实验表明，该方法优于现有的颜色模型。 合显著性分析和基于小波变换的时间特征，该颜色模型使 TPR 和 TNR 的最终检测结果达到 95% 以上，优于现有的方法。

五、随机森林

算法思想

随机森林是一种新兴起的、高度灵活的一种机器学习算法，随机森林（Random Forest，简称RF）拥有广泛的应用前景，从市场营销到医疗保健保险，既可以用来做市场营销模拟的建模，统计客户来源，保留和流失，也可用来预测疾病的风险和病患者的易感性。

随机森林就是通过集成学习的思想将多棵树集成的一种算法，它的基本单元是决策树，而它的本质属于机器学习的一大分支——集成学习（Ensemble Learning）方法。随机森林的名称中有两个关键词，一个是“随机”，一个就是“森林”。“森林”我们很好理解，一棵叫做树，那么成百上千棵就可以叫做森林了其实这也是随机森林的主要思想--集成思想的体现。

从直观角度来解释，每棵决策树都是一个分类器（假设现在针对的是分类问题），那么对于一个输入样本，N棵树会有N个分类结果。而随机森林集成了所有的分类投票结果，将投票次数最多的类别指定为最终的输出，这就是一种最简单的 Bagging 思想。

随机森林特点

- 在当前所有算法中，具有极好的准确率/It is unexcelled in accuracy among current algorithms；
- 能够有效地运行在大数据集上；
- 能够处理具有高维特征的输入样本，而且不需要降；
- 能够评估各个特征在分类问题上的重要性；
- 在生成过程中，能够获取到内部生成误差的一种无偏估计；
- 对于缺省值问题也能够获得很好得结果。

随机森林相关概念

- **信息、熵以及信息增益**

  决策树中，如果带分类的事物集合可以划分为多个类别当中，则某个类（xi）的信息可以定义如下:

  $$I(X=x_i)=-log_2p(x_i)$$

  $I(x)$用来表示随机变量的信息，$p(x_i)$指是当x_i发生时的概率。

  熵是用来度量不确定性的，当熵越大，$X=x_i$的不确定性越大，反之越小。对于机器学习中的分类问题而言，熵越大即这个类别的不确定性更大，反之越小。

  信息增益在决策树算法中是用来选择特征的指标，信息增益越大，则这个特征的选择性越好。

- **决策树**

  决策树是一种树形结构，其中每个内部节点表示一个属性上的测试，每个分支代表一个测试输出，每个叶节点代表一种类别。常见的决策树算法有C4.5、ID3和CART。

- **集成学习**

  集成学习通过建立几个模型组合的来解决单一预测问题。它的工作原理是生成多个分类器/模型，各自独立地学习和作出预测。这些预测最后结合成单预测，因此优于任何一个单分类的做出预测。

随机森林是集成学习的一个子类，它依靠于决策树的投票选择来决定最后的分类结果。

随机森林的生成

随机森林中有许多的分类树。我们要将一个输入样本进行分类，我们需要将输入样本输入到每棵树中进行分类。打个形象的比喻：森林中召开会议，讨论某个动物到底是老鼠还是松鼠，每棵树都要独立地发表自己对这个问题的看法，也就是每棵树都要投票。该动物到底是老鼠还是松鼠，要依据投票情况来确定，获得票数最多的类别就是森林的分类结果。森林中的每棵树都是独立的，99.9%不相关的树做出的预测结果涵盖所有的情况，这些预测结果将会彼此抵消。少数优秀的树的预测结果将会超脱于芸芸“噪音”，做出一个好的预测。将若干个弱分类器的分类结果进行投票选择，从而组成一个强分类器，这就是随机森林bagging的思想（关于bagging的一个有必要提及的问题：bagging的代价是不用单棵决策树来做预测，具体哪个变量起到重要作用变得未知，所以bagging改进了预测准确率但损失了解释性。）。

生成规则：

1、如果训练集大小为N，对于每棵树而言，随机且有放回地从训练集中的抽取N个训练样本（这种采样方式称为bootstrap sample方法），作为该树的训练集；

2、如果每个样本的特征维度为M，指定一个常数m<<M，随机地从M个特征中选取m个特征子集，每次树进行分裂时，从这m个特征中选择最优的；

3、每棵树都尽最大程度的生长，并且没有剪枝过程。一开始我们提到的随机森林中的"随机"就是指的这里的两个随机性。两个随机性的引入对随机森林的分类性能至关重要。由于它们的引入，使得随机森林不容易陷入过拟合，并且具有很好得抗噪能力。

随机森林分类效果影响因素

1、森林中任意两棵树的相关性：相关性越大，错误率越大；

2、森林中每棵树的分类能力：每棵树的分类能力越强，整个森林的错误率越低。

袋外错误率：

构建随机森林的关键问题就是如何选择最优的m，要解决这个问题主要依据计算袋外错误率oob error（out-of-bag error）。随机森林有一个重要的优点就是，没有必要对它进行交叉验证或者用一个独立的测试集来获得误差的一个无偏估计。它可以在内部进行评估，也就是说在生成的过程中就可以对误差建立一个无偏估计。在构建每棵树时，我们对训练集使用了不同的bootstrap sample（随机且有放回地抽取）。所以对于每棵树而言（假设对于第k棵树），大约有1/3的训练实例没有参与第k棵树的生成，它们称为第k棵树的oob样本。

根据采样特点进行oob估计，计算方式如下：

（note：以样本为单位）

1）对每个样本，计算它作为oob样本的树对它的分类情况（约1/3的树）；

2）然后以简单多数投票作为该样本的分类结果；

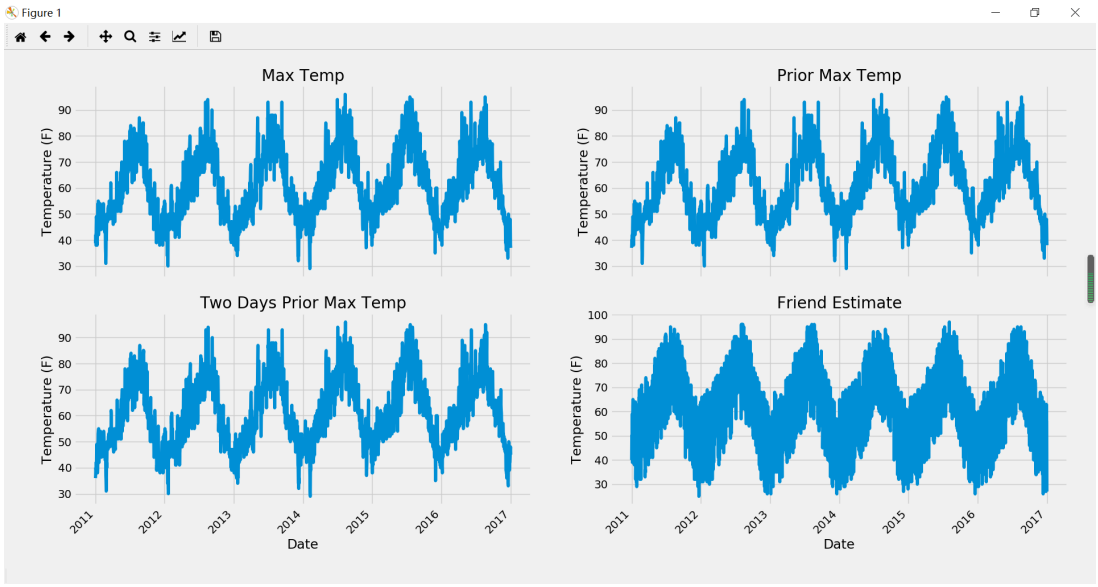
3）最后用误分个数占样本总数的比率作为随机森林的oob误分率。

使用随机森林的天气预测

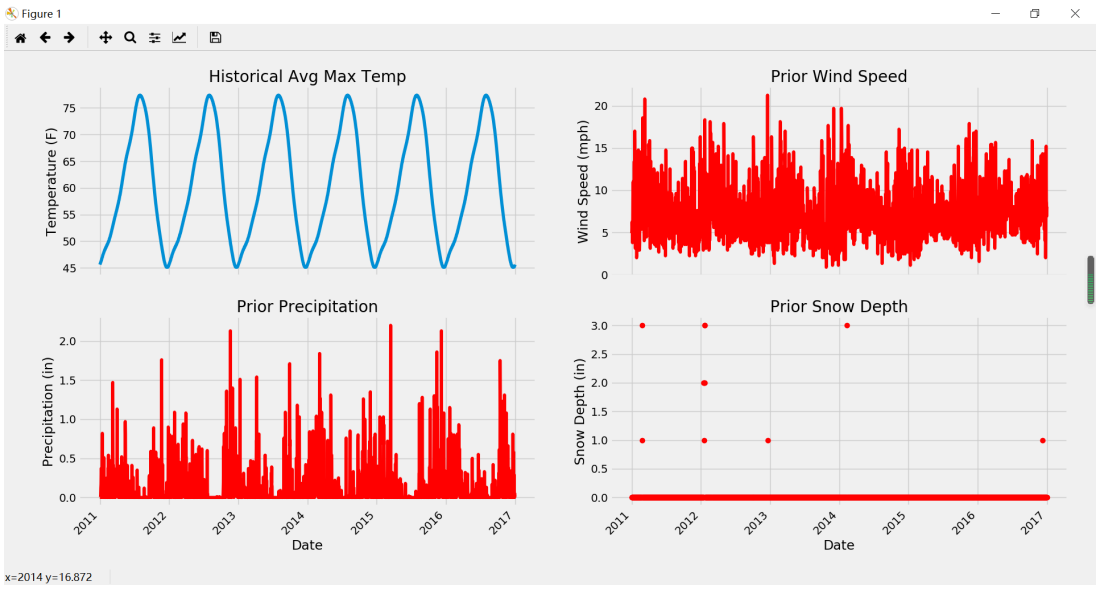
数据集：
$year$（年数）, $month$（月）, day（天）, $week$（周）, $temp_2$（温度2）, $temp_1$（温度）, $average$（平均气温）, $actual$（实际气温）, $friend$（天

根据天气状况预测天气友好度：

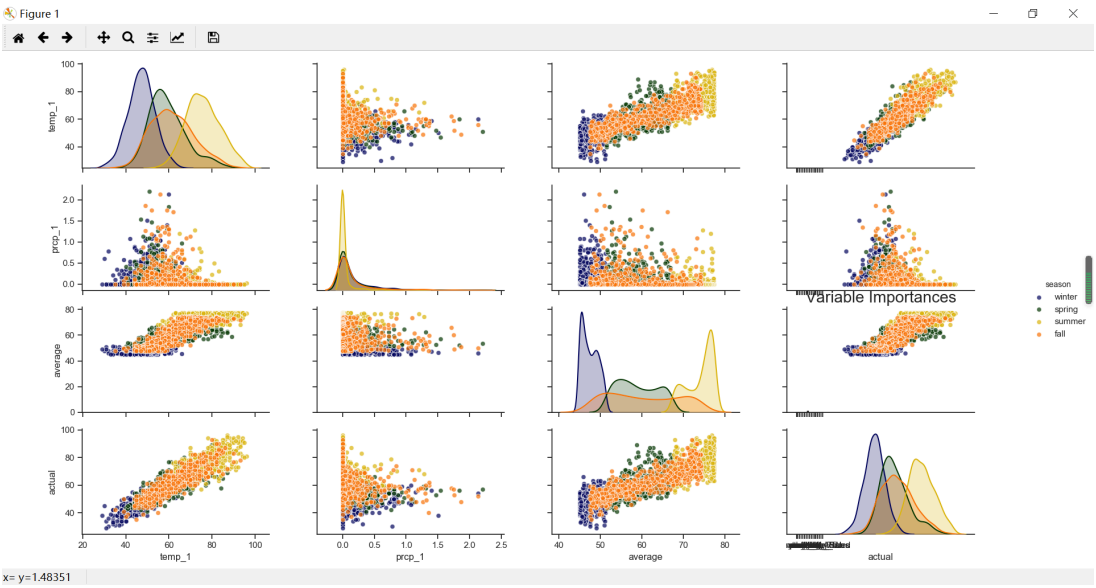
实验结果：



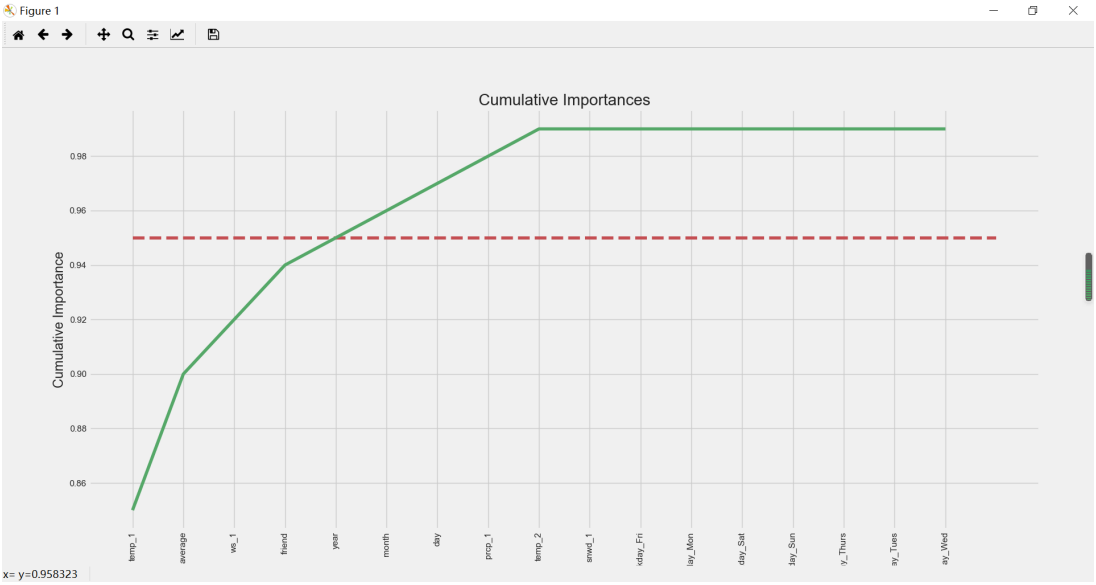
图九.气温数据



图十.预测数据



图十二.数据对比



图十三.结果累计

```
平均温度误差: 4.67 degrees.
Accuracy: 92.2 %.
平均温度误差: 4.2 degrees.
Accuracy: 93.12 %.
平均温度误差: 4.05 degrees.
特征增多后模型效果提升: 3.32 %.
Accuracy: 93.35 %.
Variable: temp_1               Importance: 0.85
Variable: average              Importance: 0.05
Variable: ws_1                 Importance: 0.02
Variable: friend               Importance: 0.02
Variable: year                 Importance: 0.01
Variable: month                Importance: 0.01
Variable: day                  Importance: 0.01
Variable: prcp_1               Importance: 0.01
Variable: temp_2               Importance: 0.01
Variable: snwd_1               Importance: 0.0
Variable: weekday_Fri          Importance: 0.0
Variable: weekday_Mon          Importance: 0.0
Variable: weekday_Sat          Importance: 0.0
Variable: weekday_Sun          Importance: 0.0
Variable: weekday_Thurs        Importance: 0.0
Variable: weekday_Tues         Importance: 0.0
Variable: weekday_Wed          Importance: 0.0
Number of features for 95% importance: 5
Important train features shape: (1643, 5)
Important test features shape: (548, 5)
```

```
平均温度误差: 4.12 degrees.
Accuracy: 93.28 %.
使用所有特征时建模与测试的平均时间消耗: 0.73 秒.
使用部分5个特征时建模与测试的平均时间消耗: 0.45 秒.
相对accuracy下降: 0.074 %.
相对时间效率提升: 38.998 %.
```

实现代码

```
# -*- encoding: utf-8 -*-
"""
@File    : predict.py
@Time    : 2020/5/24 9:29
@Author  : 王一宁
@Email   : wyn_365@163.com
庞大数据集的预测
"""

# 1.数据读取
import pandas as pd

features = pd.read_csv('D:\\pattern_recognation\\data\\temps_extended.csv')

print('数据规模',features.shape)

# 统计指标
round(features.describe(), 2)

# 2.转换成标准格式
import datetime

# 得到各种日期数据
years = features['year']
months = features['month']
days = features['day']

# 格式转换
dates = [str(int(year)) + '-' + str(int(month)) + '-' + str(int(day)) for year, month, day in zip(years,
months, days)]
dates = [datetime.datetime.strptime(date, '%Y-%m-%d') for date in dates]

# 绘图
import matplotlib.pyplot as plt

# 风格设置
plt.style.use('fivethirtyeight')

# Set up the plotting layout
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2, figsize = (15,10))
fig.autofmt_xdate(rotation = 45)

# Actual max temperature measurement
ax1.plot(dates, features['actual'])
ax1.set_xlabel(''); ax1.set_ylabel('Temperature (F)'); ax1.set_title('Max Temp')

# Temperature from 1 day ago
ax2.plot(dates, features['temp_1'])
ax2.set_xlabel(''); ax2.set_ylabel('Temperature (F)'); ax2.set_title('Prior Max Temp')

# Temperature from 2 days ago
ax3.plot(dates, features['temp_2'])
ax3.set_xlabel('Date'); ax3.set_ylabel('Temperature (F)'); ax3.set_title('Two Days Prior Max Temp')

# Friend Estimate
ax4.plot(dates, features['friend'])
ax4.set_xlabel('Date'); ax4.set_ylabel('Temperature (F)'); ax4.set_title('Friend Estimate')

plt.tight_layout(pad=2)

plt.show()

# 设置整体布局
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2, figsize = (15,10))
fig.autofmt_xdate(rotation = 45)

# 平均最高气温
ax1.plot(dates, features['average'])
ax1.set_xlabel(''); ax1.set_ylabel('Temperature (F)'); ax1.set_title('Historical Avg Max Temp')

# 风速
ax2.plot(dates, features['ws_1'], 'r-')
```

```
ax2.set_xlabel(''); ax2.set_ylabel('Wind Speed (mph)'); ax2.set_title('Prior Wind Speed')

# 降水
ax3.plot(dates, features['prcp_1'], 'r-')
ax3.set_xlabel('Date'); ax3.set_ylabel('Precipitation (in)'); ax3.set_title('Prior Precipitation')

# 积雪
ax4.plot(dates, features['snwd_1'], 'ro')
ax4.set_xlabel('Date'); ax4.set_ylabel('Snow Depth (in)'); ax4.set_title('Prior Snow Depth')

plt.tight_layout(pad=2)

plt.show()

# 3.Pairplots画图好看的图
# 创建一个季节变量
seasons = []

for month in features['month']:
    if month in [1, 2, 12]:
        seasons.append('winter')
    elif month in [3, 4, 5]:
        seasons.append('spring')
    elif month in [6, 7, 8]:
        seasons.append('summer')
    elif month in [9, 10, 11]:
        seasons.append('fall')

# 有了季节我们就可以分析更多东西了
reduced_features = features[['temp_1', 'prcp_1', 'average', 'actual']]
reduced_features['season'] = seasons

# 导入seaborn工具包
import seaborn as sns
sns.set(style="ticks", color_codes=True);

# 选择你喜欢的颜色模板
palette = sns.xkcd_palette(['dark blue', 'dark green', 'gold', 'orange'])

# 绘制pairplot
sns.pairplot(reduced_features, hue = 'season', diag_kind = 'kde', palette= palette, plot_kws=dict(alpha =
0.7),diag_kws=dict(shade=True));

# 4.数据预处理
# 独热编码
features = pd.get_dummies(features)

# 提取特征和标签
labels = features['actual']
features = features.drop('actual', axis = 1)

# 特征名字留着备用
feature_list = list(features.columns)

# 转换成所需格式
import numpy as np

features = np.array(features)
labels = np.array(labels)

# 数据集切分
from sklearn.model_selection import train_test_split

train_features, test_features, train_labels, test_labels = train_test_split(features, labels,
                                                                            test_size = 0.25, random_state =
0)
print('Training Features Shape:', train_features.shape)
print('Training Labels Shape:', train_labels.shape)
print('Testing Features Shape:', test_features.shape)
print('Testing Labels Shape:', test_labels.shape)

# 5.老数据的结果
# 工具包导入
import pandas as pd

# 为了剔除特征个数对结果的影响，这里特征统一只有老数据集中特征
original_feature_indices = [feature_list.index(feature) for feature in
                                    feature_list if feature not in
                                    ['ws_1', 'prcp_1', 'snwd_1']]

# 读取老数据集
original_features = pd.read_csv('D:\\pattern_recognation\\data\\temps.csv')
```

```
original_features = pd.get_dummies(original_features)

import numpy as np

# 数据和标签转换
original_labels = np.array(original_features['actual'])

original_features= original_features.drop('actual', axis = 1)

original_feature_list = list(original_features.columns)

original_features = np.array(original_features)

# 数据集切分
from sklearn.model_selection import train_test_split

original_train_features, original_test_features, original_train_labels, original_test_labels =
train_test_split(original_features, original_labels, test_size = 0.25, random_state = 42)

# 同样的树模型进行建模
from sklearn.ensemble import RandomForestRegressor

# 同样的参数与随机种子
rf = RandomForestRegressor(n_estimators= 100, random_state=0)

# 这里的训练集使用的是老数据集的
rf.fit(original_train_features, original_train_labels);

# 为了测试效果能够公平，统一使用一致的测试集，这里选择了刚刚我切分过的新数据集的测试集
predictions = rf.predict(test_features[:,original_feature_indices])

# 先计算温度平均误差
errors = abs(predictions - test_labels)

print('平均温度误差:', round(np.mean(errors), 2), 'degrees.')

# MAPE
mape = 100 * (errors / test_labels)

# 这里的Accuracy为了方便观察，我们就用100减去误差了，希望这个值能够越大越好
# 当我们把数据量增大之后，效果发生了一些提升，这也符合实际情况，
# 在机器学习任务中，我们都是希望数据量能够越大越好，这样可利用的信息就更多了。
# 下面我们要再对比一下特征数量对结果的影响，之前这两次比较还没有加入新的特征，

accuracy = 100 - np.mean(mape)
print('Accuracy:', round(accuracy, 2), '%.')

# 6.新数据集
# 这回我们把降水，风速，积雪3特征加入训练集中，看看效果又会怎样：
from sklearn.ensemble import RandomForestRegressor

# 剔除掉新的特征，保证数据特征是一致的
original_train_features = train_features[:,original_feature_indices]

original_test_features = test_features[:, original_feature_indices]

rf = RandomForestRegressor(n_estimators= 100 ,random_state=0)

rf.fit(original_train_features, train_labels);

# 预测
baseline_predictions = rf.predict(original_test_features)

# 结果
baseline_errors = abs(baseline_predictions - test_labels)

print('平均温度误差:', round(np.mean(baseline_errors), 2), 'degrees.')

# (MAPE)
baseline_mape = 100 * np.mean((baseline_errors / test_labels))

# accuracy
baseline_accuracy = 100 - baseline_mape
print('Accuracy:', round(baseline_accuracy, 2), '%.')

# 7.加入新特征
# 准备加入新的特征
from sklearn.ensemble import RandomForestRegressor

rf_exp = RandomForestRegressor(n_estimators= 100, random_state=0)
rf_exp.fit(train_features, train_labels)
```

```
# 同样的测试集
predictions = rf_exp.predict(test_features)

# 评估
errors = abs(predictions - test_labels)

print('平均温度误差:', round(np.mean(errors), 2), 'degrees.')

# (MAPE)
mape = np.mean(100 * (errors / test_labels))

# 看一下提升了多少
improvement_baseline = 100 * abs(mape - baseline_mape) / baseline_mape
print('特征增多后模型效果提升:', round(improvement_baseline, 2), '%.')

# accuracy
accuracy = 100 - mape
print('Accuracy:', round(accuracy, 2), '%.')

# 8.特征重要性
# 特征名字
importances = list(rf_exp.feature_importances_)

# 名字，数值组合在一起
feature_importances = [(feature, round(importance, 2)) for feature, importance in zip(feature_list,
importances)]

# 排序
feature_importances = sorted(feature_importances, key = lambda x: x[1], reverse = True)

# 打印出来
[print('Variable: {:20} Importance: {}'.format(*pair)) for pair in feature_importances];

# 指定风格
plt.style.use('fivethirtyeight')

# 指定位置
x_values = list(range(len(importances)))

# 绘图
plt.bar(x_values, importances, orientation = 'vertical', color = 'r', edgecolor = 'k', linewidth = 1.2)

# x轴名字得竖着写
plt.xticks(x_values, feature_list, rotation='vertical')

# 图名
plt.ylabel('Importance'); plt.xlabel('Variable'); plt.title('Variable Importances');
plt.show()

# 对特征进行排序 0.95那 累加特征重要性
sorted_importances = [importance[1] for importance in feature_importances]
sorted_features = [importance[0] for importance in feature_importances]

# 累计重要性
cumulative_importances = np.cumsum(sorted_importances)

# 绘制折线图
plt.plot(x_values, cumulative_importances, 'g-')

# 画一条红色虚线，0.95那 累加特征重要性
plt.hlines(y = 0.95, xmin=0, xmax=len(sorted_importances), color = 'r', linestyles = 'dashed')

# X轴
plt.xticks(x_values, sorted_features, rotation = 'vertical')

# Y轴和名字
plt.xlabel('Variable'); plt.ylabel('Cumulative Importance'); plt.title('Cumulative Importances');
plt.show()

# 看看有几个特征
print('Number of features for 95% importance:', np.where(cumulative_importances > 0.95)[0][0] + 1)

# 9.效率对比分析
# 选择这些特征
important_feature_names = [feature[0] for feature in feature_importances[0:5]]
# 找到它们的名字
important_indices = [feature_list.index(feature) for feature in important_feature_names]

# 重新创建训练集
important_train_features = train_features[:, important_indices]
important_test_features = test_features[:, important_indices]
```

```
# 数据维度
print('Important train features shape:', important_train_features.shape)
print('Important test features shape:', important_test_features.shape)

# 再训练模型
rf_exp.fit(important_train_features, train_labels);

# 同样的测试集
predictions = rf_exp.predict(important_test_features)

# 评估结果
errors = abs(predictions - test_labels)

print('平均温度误差:', round(np.mean(errors), 2), 'degrees.')

mape = 100 * (errors / test_labels)

# accuracy
accuracy = 100 - np.mean(mape)
print('Accuracy:', round(accuracy, 2), '%.')

# 10.看起来奇迹并没有出现，本以为效果反而会更好，其实还有一点点下降，
# 这里可能由于是树模型本身具有特征选择的被动技能了。虽然模型没有提升，
# 我们还可以再看看在时间效率的层面上有没有进步呢：
# 要计算时间了
import time

# 10.1这次是用所有特征
all_features_time = []

# 算一次可能不太准，来10次取个平均
for _ in range(10):
    start_time = time.time()
    rf_exp.fit(train_features, train_labels)
    all_features_predictions = rf_exp.predict(test_features)
    end_time = time.time()
    all_features_time.append(end_time - start_time)

all_features_time = np.mean(all_features_time)
print('使用所有特征时建模与测试的平均时间消耗:', round(all_features_time, 2), '秒.')

# 10.2 这次是用部分5个重要的特征
reduced_features_time = []

# 算一次可能不太准，来10次取个平均
for _ in range(10):
    start_time = time.time()
    rf_exp.fit(important_train_features, train_labels)
    reduced_features_predictions = rf_exp.predict(important_test_features)
    end_time = time.time()
    reduced_features_time.append(end_time - start_time)

reduced_features_time = np.mean(reduced_features_time)
print('使用部分5个特征时建模与测试的平均时间消耗:', round(reduced_features_time, 2), '秒.')

# 11.用分别的预测值来计算评估结果
all_accuracy =  100 * (1- np.mean(abs(all_features_predictions - test_labels) / test_labels))
reduced_accuracy = 100 * (1- np.mean(abs(reduced_features_predictions - test_labels) / test_labels))

#创建一个df来保存结果
comparison = pd.DataFrame({'features': ['all (17)', 'reduced (5)'],
                           'run_time': [round(all_features_time, 2), round(reduced_features_time, 2)],
                           'accuracy': [round(all_accuracy, 2), round(reduced_accuracy, 2)]})

comparison[['features', 'accuracy', 'run_time']]

relative_accuracy_decrease = 100 * (all_accuracy - reduced_accuracy) / all_accuracy
print('相对accuracy下降:', round(relative_accuracy_decrease, 3), '%.')

relative_runtime_decrease = 100 * (all_features_time - reduced_features_time) / all_features_time
print('相对时间效率提升:', round(relative_runtime_decrease, 3), '%.')

# 12. 通常我们买东西都会考虑性价比，这里同样也是这个问题，时间效率的提升相对更大一些，
# 而且基本保证了模型效果是差不多的。
# 最后让我们把所有的实验结果汇总到一起来进行对比吧：
# Pandas is used for data manipulation
import pandas as pd

# Read in data as pandas dataframe and display first 5 rows
original_features = pd.read_csv('\\data\\temps.csv')
original_features = pd.get_dummies(original_features)
```

```
# Use numpy to convert to arrays
import numpy as np

# Labels are the values we want to predict
original_labels = np.array(original_features['actual'])

# Remove the labels from the features
# axis 1 refers to the columns
original_features= original_features.drop('actual', axis = 1)

# Saving feature names for later use
original_feature_list = list(original_features.columns)

# Convert to numpy array
original_features = np.array(original_features)

# Using Skicit-learn to split data into training and testing sets
from sklearn.model_selection import train_test_split

# Split the data into training and testing sets
original_train_features, original_test_features, original_train_labels, original_test_labels =
train_test_split(original_features, original_labels, test_size = 0.25, random_state = 42)

# Find the original feature indices
original_feature_indices = [feature_list.index(feature) for feature in
                            feature_list if feature not in
                            ['ws_1', 'prcp_1', 'snwd_1']]

# Create a test set of the original features
original_test_features = test_features[:, original_feature_indices]

# Time to train on original data set (1 year)
original_features_time = []

# Do 10 iterations and take average for all features
for _ in range(10):
    start_time = time.time()
    rf.fit(original_train_features, original_train_labels)
    original_features_predictions = rf.predict(original_test_features)
    end_time = time.time()
    original_features_time.append(end_time - start_time)

original_features_time = np.mean(original_features_time)

# Calculate mean absolute error for each model
original_mae = np.mean(abs(original_features_predictions - test_labels))
exp_all_mae = np.mean(abs(all_features_predictions - test_labels))
exp_reduced_mae = np.mean(abs(reduced_features_predictions - test_labels))

# Calculate accuracy for model trained on 1 year of data
original_accuracy = 100 * (1 - np.mean(abs(original_features_predictions - test_labels) / test_labels))

# Create a dataframe for comparison
model_comparison = pd.DataFrame({'model': ['original', 'exp_all', 'exp_reduced'],
                                 'error (degrees)':  [original_mae, exp_all_mae, exp_reduced_mae],
                                 'accuracy': [original_accuracy, all_accuracy, reduced_accuracy],
                                 'run_time (s)': [original_features_time, all_features_time,
reduced_features_time]})

# Order the dataframe
model_comparison = model_comparison[['model', 'error (degrees)', 'accuracy', 'run_time (s)']]

model_comparison

# 绘图来总结把
# 设置总体布局，还是一整行看起来好一些
fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3, figsize = (16,5), sharex = True)

# X轴
x_values = [0, 1, 2]
labels = list(model_comparison['model'])
plt.xticks(x_values, labels)

# 字体大小
fontdict = {'fontsize': 18}
fontdict_yaxis = {'fontsize': 14}

# 预测温度和真实温度差异对比
ax1.bar(x_values, model_comparison['error (degrees)'], color = ['b', 'r', 'g'], edgecolor = 'k', linewidth =
1.5)
ax1.set_ylim(bottom = 3.5, top = 4.5)
```

```
ax1.set_ylabel('Error (degrees) (F)', fontdict = fontdict_yaxis);
ax1.set_title('Model Error Comparison', fontdict= fontdict)

# Accuracy 对比
ax2.bar(x_values, model_comparison['accuracy'], color = ['b', 'r', 'g'], edgecolor = 'k', linewidth = 1.5)
ax2.set_ylim(bottom = 92, top = 94)
ax2.set_ylabel('Accuracy (%)', fontdict = fontdict_yaxis);
ax2.set_title('Model Accuracy Comparison', fontdict= fontdict)

# 时间效率对比
ax3.bar(x_values, model_comparison['run_time (s)'], color = ['b', 'r', 'g'], edgecolor = 'k', linewidth = 1.5)
ax3.set_ylim(bottom = 0, top = 1)
ax3.set_ylabel('Run Time (sec)', fontdict = fontdict_yaxis);
ax3.set_title('Model Run-Time Comparison', fontdict= fontdict);
plt.show()
print("original代表是我们的老数据，也就是量少特征少的那份；exp_all代表我们的完整新数据；exp_reduced代表我们按照95%阈值选择的部分重要特征数据集。结果也是很明显的，数据量和特征越多，效果会提升一些，但是时间效率也会有所下降。")
```