基于果蝇嗅觉系统的相似性查找算法

易凯 西安交通大学

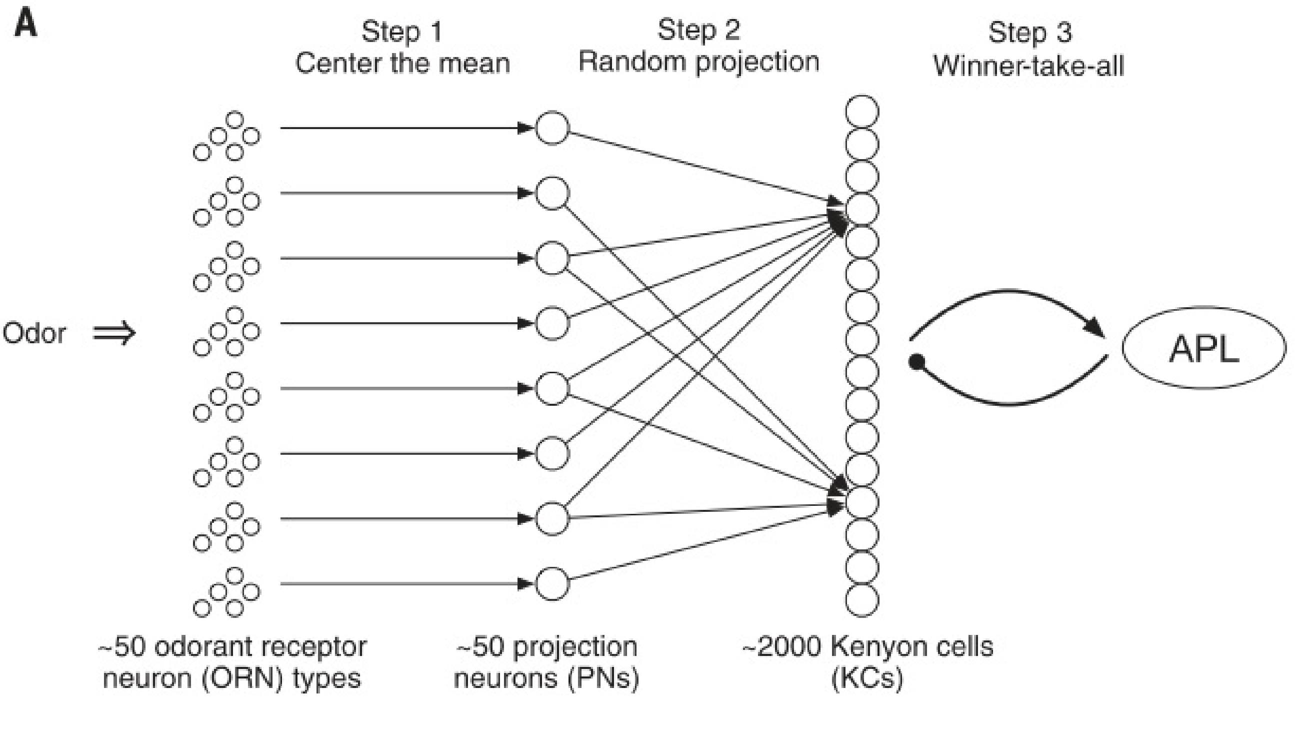
论文作者： Sanjoy Dasgupta, Charles F. Stevens, Saket Navlakha

概要介绍

相似性查找(Similarity Search)在搜索领域有着巨大的作用，例如通过一副大象的图片搜索出数据库中相关的大象的图片。本文的主要研究对象是果蝇(fruit fly), 其分析的问题是，如果果蝇嗅觉系统发现A气味是可以吃的食物发出来的，选择靠近，下次闻到了气味B，通过与A气味的相似度分析，这时果蝇是如何选择去靠近还是远离的。

果蝇算法工作流

果蝇算法的实现分为三个阶段(分别为center the mean, random projection, winner-take-all)，如下图：



Center the mean

对于任何一个气味来说，ORNs是遵循指数分布的，它有不同的均值以及极值；而对于PNs来说，其也遵循指数分布，但是有基本相同的均值和极值。因而为了不至于混淆气味的类型需要进行"center the mean"操作，它在计算机中常常叫做divisive normalization.

center the mean 的具体实现如下：

```
def standardize_data(D,do_norm):
    """ Performs several standardizations on the data.
            1) Makes sure all values are non-negative.
            2) Sets the mean of example to SET_MEAN.
            3) Applies normalization if desired.
    """

    # 1. Add the most negative number per column (ORN) to
make all values >= 0.
    for col in xrange(DIM):
        D[:,col] += abs(min(D[:,col]))

    # 2. Set the mean of each row (odor) to be SET_MEAN.
    for row in xrange(N):

        # Multiply by: SET_MEAN / current mean. Keeps
proportions the same.
        D[row,:] = D[row,:] * ((SET_MEAN /
np.mean(D[row,:])))
        D[row,:] = map(int,D[row,:])

        assert abs(np.mean(D[row,:]) - SET_MEAN) <= 1

    # 3. Applies normalization.
    if do_norm: # := v / np.linalg.norm(v)
        D = D.astype(np.float64)
        D = normalize(D)

    # Make sure all values (firing rates) are >= 0.
    for row in xrange(N):
        for col in xrange(DIM):
            assert D[row,col] >= 0

    return D
```

其分为如下三个步骤：

- 确保每列的数字都是非负数，通过D[:, col] += abs(min(D[:, col]))，也就是将同一列的所有元素都加上最小值的绝对值来实现
- 将每一行所有元素的均值设置为SET_MEAN, 通过D[row,:] = D[row,:] * ((SET_MEAN / np.mean(D[row,:]))) 实现，将可能产生的浮点数全部转换为整数，同时assert转换后均值与SET_MEAN差距的绝对值小于1
- 进行归一化。首先通过D = D.astype(np.float64)将所有元素转换为FP64, 然后调用normalize()函数实现归一化。

Random projection

random projection阶段就是将PNs映射到KCs, 其与传统的locality-sense projection一个显著不同是，其将PNs映射到更高维度的KCs. 在果蝇的实例中，PNs约为50个神经元，而KCs约为2000个神经元。而且其采用的是随机化二值映射。理论分析表明，随机投影可以有效保证数据的局部性。

随机稀疏二值矩阵的产生如下：

```
def create_rand_proj_matrix():
    """ Creates a random projection matrix of size
NUM_KENYON by NUM_ORNS. """

    # Create a sparse, binary random projection matrix.
    if PROJECTION.startswith("SB"):

        num_sample = int(PROJECTION[2:]) # "SB6" -> 6
        assert num_sample <= DIM
        # Each row (KC) samples from the glomeruli: every
row has num_sample
        # random 1s, and 0s everywhere else.
        M = np.zeros((NUM_KENYON,DIM))
        for row in xrange(NUM_KENYON):

            # Sample NUM_SAMPLE random indices, set these
to 1.
            for idx in
random.sample(xrange(DIM),num_sample):
                M[row,idx] = 1

            # Make sure I didn't screw anything up!
            assert sum(M[row,:]) == num_sample

    # Create a dense, Gaussian random projection matrix.
    elif PROJECTION == "DG":
        M = np.random.randn(NUM_KENYON,DIM)

    else: assert False

    return M
```

其实现代码进行了sparse-binary projection 以及 dense-gaussian projection的对比。

其实现的具体矩阵操作在后文pipeline中进行说明。

对该矩阵，首先选定一行row, 然后在每一列的DIM维中随机选择num_sample个元素进行初始化。对于致密高斯随机映射，直接调用np.random.randn()函数即可生成。

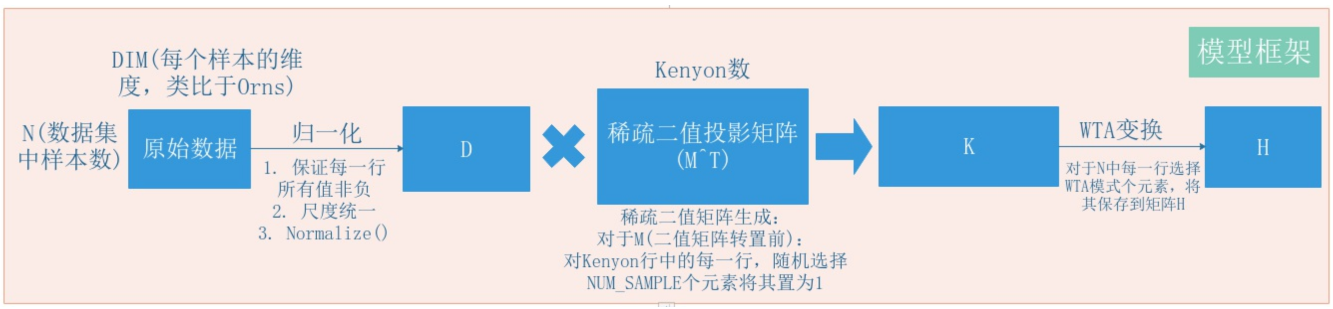
Winner-take-all(WTA)

对于每一个神经元，其所有的连接中选择spiking firing rate最高的5%来表征对这一气味的tag.

```
elif WTA == "top":
                indices = np.argpartition(K[i,:],-
HASH_LENGTH)[-HASH_LENGTH:]
```

其WTA的思想也就是选取比率最高的指定个数个。

通过对其源代码的解析得到如下流程图：



果蝇算法与传统LSH算法的差异

果蝇算法与传统LSH算法存在着如下的三个差异：

	Fly Algorithm	**LSH Algorithm**
映射方法	sparse, binary random projections	dense, Gaussian random projections
映射机制	升维(d << m)	降维(d >> m)
结果表示	WTA mechanism	dense representation

Note: Dense and Gaussian random projections are more computational than sparse and binary random projections.

算法mAP评估

算法评估的原始代码如下：

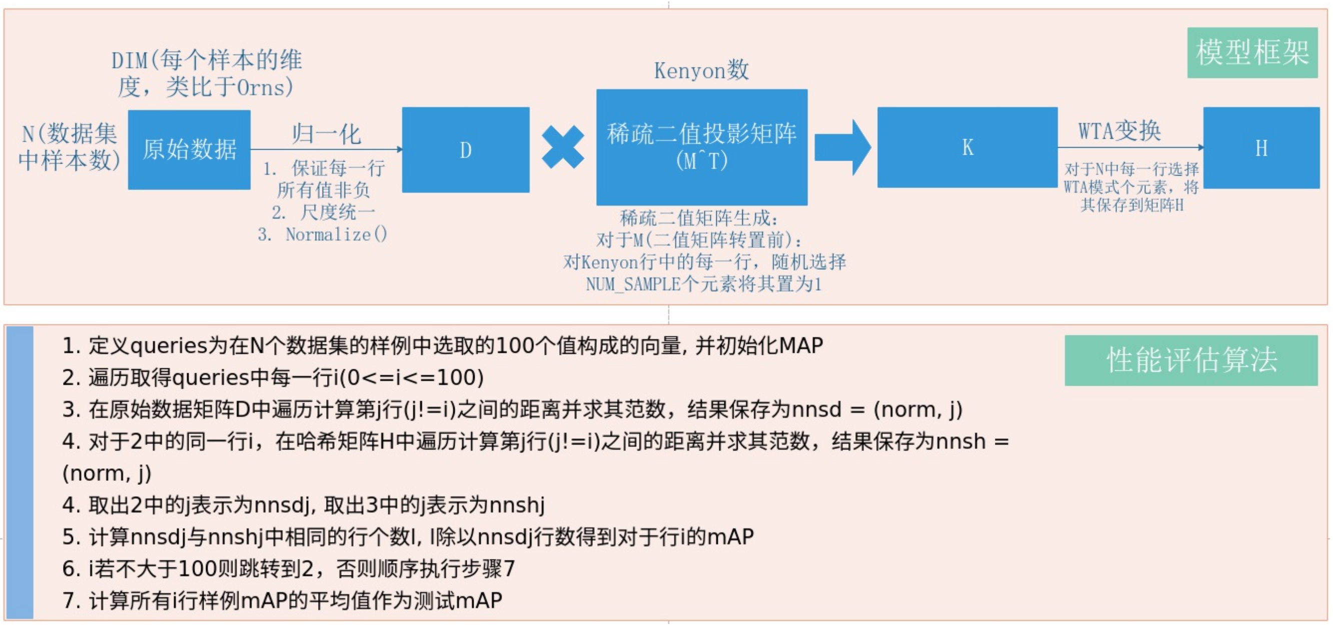
```
def tesht_map_dist(D,H):
    """ Computes mean average precision (MAP) and
distortion between true nearest-neighbors
        in input space and approximate nearest-neighbors
in hash space.
    """


    queries = random.sample(xrange(N),100)

    MAP       = [] # [list of MAP values for each query]
    for i in queries:

        temp_i = [] # list of (dist input space,odor) from
i.
        temp_h = [] # list of (dist hash space ,odor) from
i.
        for j in xrange(N):
            if i == j: continue

            # Distance between i and j in input space.
            dij_orig = dist(D[i,:],D[j,:])
            if dij_orig <= 0: continue # i and j are
duplicates, e.g. corel: i=1022,j=2435.
            temp_i.append( (dij_orig,j) )

            # Distance between i and j in hash space.
            dij_hash = dist(H[i,:],H[j,:])
            temp_h.append( (dij_hash,j) )

        assert len(temp_i) == len(temp_h) # == N-1 # not
the last part bc of duplicates.


        # Create a set of the true NUM_NNS nearest
neighbors.
        #true_nns = sorted(temp_i)[0:NUM_NNS]      # true
NUM_NNS tuples.
        true_nns = heapq.nsmallest(NUM_NNS,temp_i) # true
```

```
NUM_NNS tuples. (faster than above)
        true_nns = set([vals[1] for vals in true_nns]) #
true NUM_NNS examples.

        # Go through predicted nearest neighbors and
compute the MAP.
        #pred_nns = sorted(temp_h)[0:NUM_NNS]      # pred
NUM_NNS tuples.
        pred_nns = heapq.nsmallest(NUM_NNS,temp_h) # pred
NUM_NNS tuples. (faster than above)
        pred_nns = [vals[1] for vals in pred_nns] # pred
NUM_NNS examples.

        assert len(true_nns) == len(pred_nns)

        num_correct_thus_far = 0
        map_temp = []
        for idx,nghbr in enumerate(pred_nns):

            if nghbr in true_nns:
                num_correct_thus_far += 1

map_temp.append((num_correct_thus_far)/(idx+1))

        map_temp = np.mean(map_temp) if len(map_temp) > 0
else 0
        assert 0.0 <= map_temp <= 1.0

        MAP.append(map_temp)

    # Store overall performance for these queries.
    x_map = np.mean(MAP)

    return x_map
```

该评估部分的核心代码可用伪代码表示为：

```
1. 定义queries为在N个数据集的样例中选取的100个值构成的向量，并初始化MAP

2. 遍历取得queries中每一行i(0<=i<=100)

3. 在原始数据矩阵D中遍历计算第j行(j!=i)之间的距离并求其范数，结果保存为nnsd = (norm, j)

4. 对于2中的同一行i，在哈希矩阵H中遍历计算第j行(j!=i)之间的距离并求其范数，结果保存为nnsh = (norm, j)

4. 取出2中的j表示为nnsdj，取出3中的j表示为nnshj

5. 计算nnsdj与nnshj中相同的行个数l，l除以nnsdj行数得到对于行i的mAP

6. i若不大于100则跳转到2，否则顺序执行步骤7

7. 计算所有i行样例mAP的平均值作为测试mAP
```

最后得到的main pipeline为如下所示：



1. 定义queries为在N个数据集的样例中选取的100个值构成的向量，并初始化MAP
2. 遍历取得queries中每一行i(0<=i<=100)
3. 在原始数据矩阵D中遍历计算第j行(j!=i)之间的距离并求其范数，结果保存为nnsd = (norm, j)
4. 对于2中的同一行i，在哈希矩阵H中遍历计算第j行(j!=i)之间的距离并求其范数，结果保存为nnsh = (norm, j)
4. 取出2中的j表示为nnsdj，取出3中的j表示为nnshj
5. 计算nnsdj与nnshj中相同的行个数l，l除以nnsdj行数得到对于行i的mAP
6. i若不大于100则跳转到2，否则顺序执行步骤7
7. 计算所有i行样例mAP的平均值作为测试mAP

参考文献

1. Sanjoy D et al. A neural algorithm for a fundamental computing problem. Science. 10 Nov 2017. pp. 793-796.
2. Makarand Tapaswi et al. Intuition behind Average Precision and MAP.
3. Sanjoy D et al. Supplementary Materials and source code for A neural algorithm for a fundamental computing problem. Science. 10 Nov 2017. pp. 793-796.