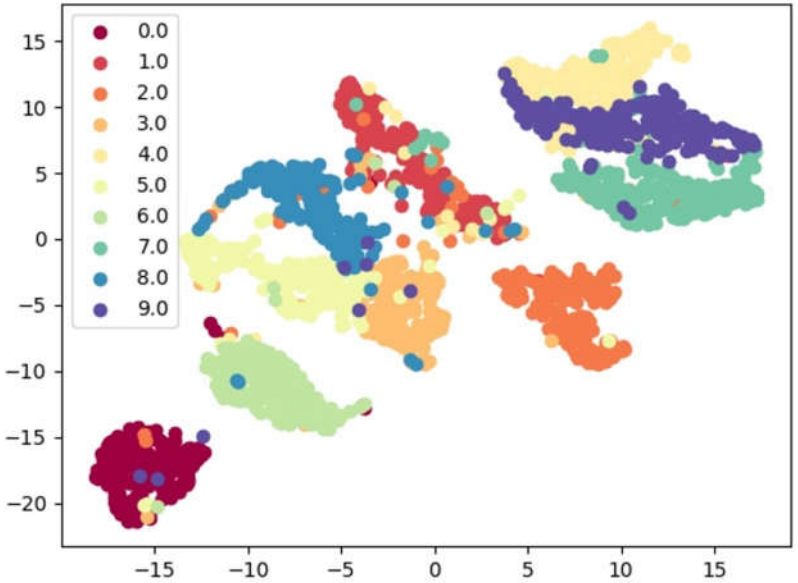
Python编程与人工智能实践

算法篇：数据降维-UMAP
（Uniform Manifold Approximation and Projection）

均匀流形逼近与投影



于泓
鲁东大学
信息与电气工程学院
2021.10.15

UMAP（Uniform Manifold Approximation and Projection）

UMAP 是新近提出的一种新型的数据降维（数据可视化）方法。其算法原理以及降维效果与t-sne类似，但有以下改进：

（1）比t-sne 可以得到更好的数据聚拢效果，能能够表达更好的局部结构。

（2）运算效率以及运算速度比t-sne好的多，可以适用于大规模数据降维

（3）可以实现任意维度的降维

参考文献：[1] Mcinnes L , Healy J . UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction[J]. The Journal of Open Source Software, 2018, 3(29):861.

论文代码：https://github.com/lmcinnes/umap

参考文案：[译]理解 UMAP(1)：UMAP是如何工作的 & UMAP 与 tSNE的原理对比 - 知乎 (zhihu.com)
https://zhuanlan.zhihu.com/p/150788883

- UMAP算法本身的理论推导比较复杂，涉及到黎曼几何、拓扑代数等复杂的数学理论，但具体实现过程和t-sne算法类似：
- （1）设计一个函数（**概率**）来构建高维样本点，两两之间的关系（联合概率）
- （2）构建另一个函数（**概率**）来构建低维样本点两两之间的关系
- （3）构造一个损失函数，通过学习的方法（梯度下降）令高维样本点之间的关系和低维样本点之前的关系尽可能相似。

(1) 高维点之间的关系

固定值（k）

$$k = 2^{\sum_i p_{ij}}$$

条件概率 $$p_{i|j} = e^{-\frac{d(x_i, x_j) - \rho_i}{\sigma_i}}$$

调节参数

只考虑k个近邻点

联合概率 $$p_{ij} = p_{i|j} + p_{j|i} - p_{i|j} p_{j|i}$$ 保证对称性

(2) 低维点之间的关系

$$q_{ij} = \left(1 + a(y_i - y_j)^{2b}\right)^{-1}$$

通过调节 a,b 可以调整映射后低维数据的聚拢情况

(3) 损失函数 二进制交叉熵

$$CE(X, Y) = \sum_i \sum_j \left[p_{ij}(X) \log\left(\frac{p_{ij}(X)}{q_{ij}(Y)}\right) + (1 - p_{ij}(X)) \log\left(\frac{1 - p_{ij}(X)}{1 - q_{ij}(Y)}\right) \right]$$

与t-sen的比较

关键改进

（1）高维点关系：

t-sne

$$p_{j|i} = \frac{\exp\left(-\|x_i - x_j\|^2 / 2\sigma_i^2\right)}{\sum_{k \neq i} \exp\left(-\|x_i - x_k\|^2 / 2\sigma_i^2\right)}$$

UMAP

$$p_{i|j} = e^{-\frac{d\left(x_i, x_j\right) - \rho_i}{\sigma_i}}$$

固定的松弛系数

$$\log(\mathrm{Per}) = -\sum_j p_{j|i} \log(p_{j|i})$$

固定值（k）

$$k = 2^{\sum_i p_{ij}}$$

在具体应用中 $d\left(x_i, x_j\right)$ 主要决定了 $p_{i|j}$ 的输出

t-sne： 如果有一个点i**距离所有的点都很远，那么所有的** $p_{i|j}$ **都接近0，会造成图的不连接**

UMAP：引入 ρ_i（距离点i最近的点的距离）保证至少有一个 $p_{i|j}$ =1，**保证图的连通性**

去掉正则项，减少计算复杂度

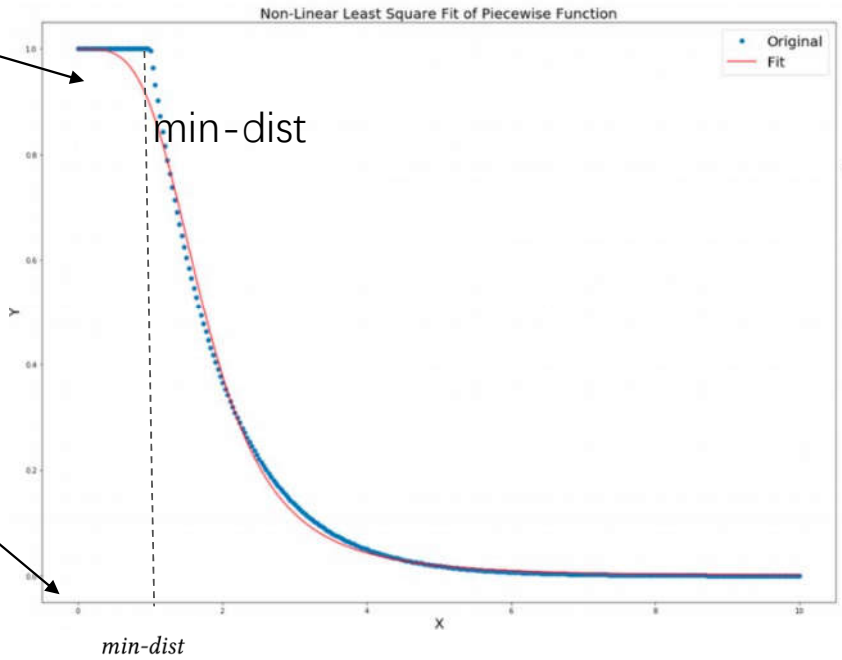
（2）低维点的关系

t-sne $$q_{ij} = \frac{\left(1+\|y_i - y_j\|^2\right)^{-1}}{\sum_{k\neq l}\left(1+\|y_k - y_l\|^2\right)^{-1}}.$$

UMAP $$q_{ij} = \left(1 + a(y_i - y_j)^{2b}\right)^{-1}$$

去掉正则项，增加了超参数a,b

高维空间较接近的点，
p_{ij}较大
映射后 q_{ij} 较大



点点间的距离
<min-dist

a,b的设置的目的是为了拟合分段函数:

$$\Psi(\mathbf{x},\mathbf{y}) = \begin{cases} 1 & \textit{if } \|\mathbf{x}-\mathbf{y}\|_2 \leq \textit{min-dist} \\ \exp(-(\|\mathbf{x}-\mathbf{y}\|_2 - \textit{min-dist})) & \textit{otherwise} \end{cases}$$

min-dist 越小，投影后相似的点越聚集

min-dist 越大，投影后相似的点越稀疏

在实际应用中 min-dist 为超参数
a,b通过曲线拟合的方式获取

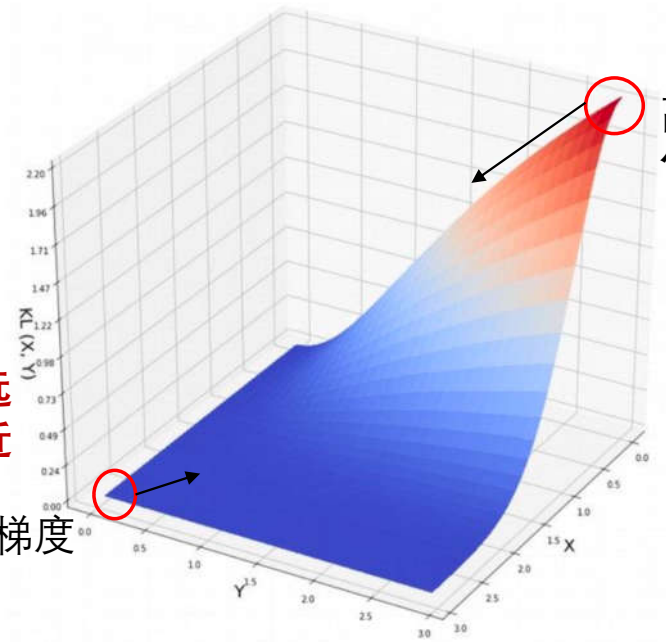
（3） 损失函数方面

X 高维点间距离　　Y 低维点间距离

t-sne $$C = KL(P||Q) = \sum_i \sum_i p_{ij} \log \frac{p_{ij}}{q_{ij}}.$$

UMAP $$CE(X,Y) = \sum_i \sum_j \left[p_{ij}(X) \log\left(\frac{p_{ij}(X)}{q_{ij}(Y)}\right) + (1 - p_{ij}(X)) \log\left(\frac{1 - p_{ij}(X)}{1 - q_{ij}(Y)}\right)\right]$$

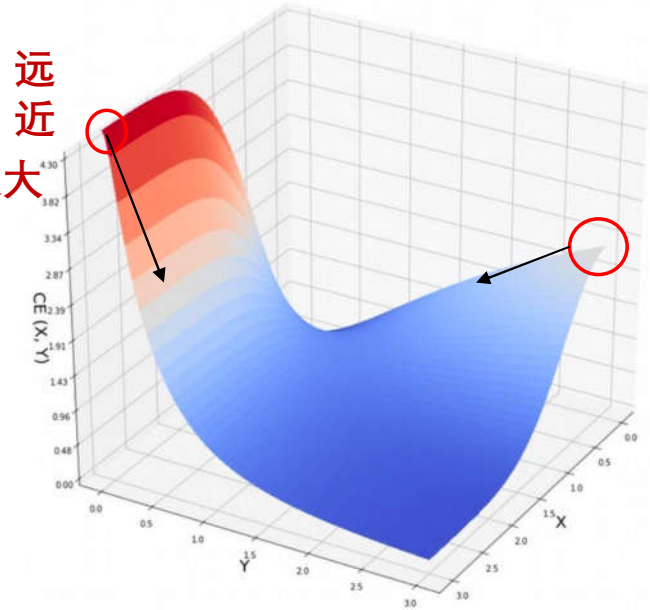
t-sne



高维X，**近**
低维Y，远
产生**较大**梯度
减少**Y**

高维X，**远**
低维Y，近
产生**较小**梯度
增大**Y**

高维X，**远**
低维Y，近
产生**较大**
梯度
增大**Y**



高维X，**近**
低维Y，远
产生**较大**梯度
减少**Y**

UMAP比**tsne** 更适合令高维空间较远的点在低维空间也远

UMAP 算法流程:

(1) 给定 **min-dist ，利用曲线拟合的方法求超参数 a,b**

(2) 给定高维数据 X， $[N,D]$,对每个样本点计算k个近邻点

求每个点的参数 ρ_i ： 距离点i最近的点的距离

σ_i 利用 $k = 2^{\sum_i p_{ij}}$ 通过二值搜索得到

计算条件概率 $p_{i|j} = e^{-\frac{d(x_i,x_j)-\rho_i}{\sigma_i}}$ 并得到联合概率 $p_{ij} = p_{i|j} + p_{j|i} - p_{i|j}p_{j|i}$

(3) 低维映射部分

将 p_{ij} 看做一个图，采用图割的方法，利用谱聚类，实现低维数据初始化

正则化拉普拉斯矩阵 L = I - D * graph * D

对L进行特征值分解，取特征值较小的d个特征值所对应的特征向量

训练部分:

$$CE(X,Y)=\sum_i\sum_j\left[p_{ij}(X)\log\left(\frac{p_{ij}(X)}{q_{ij}(Y)}\right)+(1-p_{ij}(X))\log\left(\frac{1-p_{ij}(X)}{1-q_{ij}(Y)}\right)\right]$$

$$\begin{aligned}C((A,\mu),(A,\nu))&=\sum_{a\in A}\mu(a)\log\left(\frac{\mu(a)}{\nu(a)}\right)+(1-\mu(a))\log\left(\frac{1-\mu(a)}{1-\nu(a)}\right)\\&=\sum_{a\in A}(\mu(a)\log(\mu(a))+(1-\mu(a))\log(1-\mu(a)))\\&\qquad-\sum_{a\in A}(\mu(a)\log(\nu(a))+(1-\mu(a))\log(1-\nu(a)))\,.\end{aligned}$$

与y相关

最小化: $$-\sum_{a\in A}(\mu(a)\log(\nu(a))+(1-\mu(a))\log(1-\nu(a)))\,.$$

$$-\sum_{a\in A}\left(\mu(a)\log(\nu(a))+(1-\mu(a))\log(1-\nu(a))\right).$$

导数

导数

$$\frac{-2ab\|\mathbf{y_i}-\mathbf{y_j}\|_2^{2(b-1)}}{1+\|\mathbf{y_i}-\mathbf{y_j}\|_2^2}w((x_i,x_j))\left(\mathbf{y_i}-\mathbf{y_j}\right)$$

$$\frac{2b}{(\epsilon+\|\mathbf{y_i}-\mathbf{y_j}\|_2^2)\left(1+a\|\mathbf{y_i}-\mathbf{y_j}\|_2^{2b}\right)}\left(1-w((x_i,x_j))\right)\left(\mathbf{y_i}-\mathbf{y_j}\right).$$

正样本

点i，j 出现的概率

本来是 0，为了防止分母过小加入一个较小值

负样本

在实际的应用中采用采样的方式，进行参数更新

概率大，多更新，概率小，少更新

function OPTIMIZEEMBEDDING(top-rep, Y, min-dist, n-epochs)

$\alpha \leftarrow 1.0$

Fit Φ from Ψ defined by min-dist

for $e \leftarrow 1, \ldots,$ n-epochs **do**

for all $([a, b], p) \in$ top-rep$_1$ **do**

if RANDOM( ) $\leq p$ **then** *# Sample simplex with probability p*

$y_a \leftarrow y_a + \alpha \cdot \nabla(\log(\Phi))(y_a, y_b)$

for $i \leftarrow 1, \ldots,$ n-neg-samples **do**

$c \leftarrow$ random sample from Y

$y_a \leftarrow y_a + \alpha \cdot \nabla(\log(1 - \Phi))(y_a, y_c)$

$\alpha \leftarrow 1.0 - e/$n-epochs

return Y

隔多久更新一次，
概率越大，间隔越小

总 epoch

代码实现:

第一步 计算高维空间点之间的距离（概率）

```python
def get_graph_Inputs(X,n_neighbors,local_connectivity=1):
    n_samples = X.shape[0]

    # 计算每个样本点的N个临近点的位置和距离
    NN_index, NN_dists = get_n_neighbors(X,n_neighbors)

    # 计算每个样本的 sigm 与 rho 为后边的图计算提供参数
    sigmas,rhos = compute_sigmas_and_rhos(NN_dists,n_neighbors,local_connectivity)

    # 计算两点间的 连接强度 即计算条件概率 Pj|i
    rows, cols, vals = compute_membership_strengths(NN_index,NN_dists,sigmas,rhos)

    # 构造稀疏矩阵
    result = scipy.sparse.coo_matrix((vals, (rows, cols)), shape=(X.shape[0], X.shape[0]))
    result.eliminate_zeros() # 去掉0

    # 计算联合概率 Pij
    transpose = result.transpose()
    prod_matrix = result.multiply(transpose)
    # Pij  =  Pj|i + Pi|j - Pj|i* Pi|j
    result = result + transpose - prod_matrix
    return result
```

```python
# x 维度 [N,D]
def cal_pairwise_dist(x):
    print("compute distance")
    N,D = np.shape(x)

    dist = np.zeros([N,N])

    for i in tqdm(range(N)):
        for j in range(N):
            dist[i,j] = np.sqrt(np.dot((x[i]-x[j]),(x[i]-x[j]).T))

    #返回任意两个点之间距离
    return dist
```

计算K近邻

```python
# 获取每个样本点的 n_neighbors个临近点的位置以及距离
def get_n_neighbors(data, n_neighbors = 15):
    dist = cal_pairwise_dist(data)
    dist[dist < 0] = 0
    N = dist.shape[0]
    NN_index = np.argsort(dist,axis=1)[:,0:n_neighbors]
    NN_dist = np.sort(dist,axis=1)[:,0:n_neighbors]
    return NN_index,NN_dist
```

求每个点的参数 ρ_i ： 距离点i最近的点的距离

σ_i 利用 $k = 2^{\sum_i p_{ij}}$ 通过二值搜索得到

```
# 计算每个样本点的参数 sigmas 以及 rhos
def compute_sigmas_and_rhos(distances,k,
                            local_connectivity=1,n_iter=64,
                            tol = 1.0e-5,min_k_dis_scale=1e-3):
    print("computing sigmas and rhos")
    # 获取样本数目
    N= distances.shape[0]

    # 定义变量存储每个样本的 sigma 和 rho
    rhos = np.zeros(N, dtype=np.float32)
    sigmas = np.zeros(N, dtype=np.float32)

    mean_distances = np.mean(distances)

    target = np.log2(k)

    for i in tqdm(range(N)):
        lo = 0.0
        hi = np.inf
        mid = 1.0

        # rho_i 为距离第i个样本最近的第local_connectivity个距离
        ith_distances = distances[i]
        non_zero_dists = ith_distances[ith_distances > 0.0]
        rhos[i] = non_zero_dists[local_connectivity - 1]
```

```
    # 通过2值搜索的方法计算sigma_i
    for n in range(n_iter):

        psum = 0.0
        for j in range(1, distances.shape[1]):
            d = distances[i, j] - rhos[i]
            if d > 0:
                psum += np.exp(-(d / mid))
            else:
                psum += 1.0

        if np.fabs(psum - target) < tol:
            break

        if psum > target:
            hi = mid
            mid = (lo + hi) / 2.0
        else:
            lo = mid
            if hi == np.inf:
                mid *= 2
            else:
                mid = (lo + hi) / 2.0

    sigmas[i] = mid

    # 进一步处理 防止 sigma_i 过小
    if rhos[i] > 0.0:
        mean_ith_distances = np.mean(ith_distances)
        if sigmas[i] < min_k_dis_scale * mean_ith_distances:
            sigmas[i] = min_k_dis_scale * mean_ith_distances
    # rhos[i]<=0 N个近邻点距离过近
    else:
        if sigmas[i] < min_k_dis_scale * mean_distances:
            sigmas[i] = min_k_dis_scale * mean_distances

return sigmas,rhos
```

```python
# 计算两点间的连接强度
def compute_membership_strengths(NN_index,NN_dists,sigmas,rhos):

    print("compute membership strengths")
    n_samples, n_neighbors =  np.shape(NN_index)

    rows = np.zeros(n_samples*n_neighbors, dtype=np.int32)
    cols = np.zeros(n_samples*n_neighbors, dtype=np.int32)
    vals = np.zeros(n_samples*n_neighbors, dtype=np.float32)

    for i in tqdm(range(n_samples)):
       for j in range(n_neighbors):

            if NN_index[i, j] == i:
                val = 0.0
            elif NN_dists[i, j] - rhos[i] <= 0.0 or sigmas[i] == 0.0:
                val = 1.0
            else:
                val = np.exp(-((NN_dists[i, j] - rhos[i]) / (sigmas[i])))

            rows[i * n_neighbors + j] = i
            cols[i * n_neighbors + j] = NN_index[i, j]
            vals[i * n_neighbors + j] = val

    return rows, cols, vals
```

$$p_{i|j} = e^{-\frac{d\left(x_i, x_j\right) - \rho_i}{\sigma_i}}$$

```python
def get_graph_Inputs(X,n_neighbors,local_connectivity=1):
    n_samples = X.shape[0]

    # 计算每个样本点的N个临近点的位置和距离
    NN_index, NN_dists = get_n_neighbors(X,n_neighbors)

    # 计算每个样本的 sigm 与 rho 为后边的图计算提供参数
    sigmas,rhos = compute_sigmas_and_rhos(NN_dists,n_neighbors,local_connectivity)

    # 计算两点间的 连接强度 即计算条件概率 Pj|i
    rows, cols, vals = compute_membership_strengths(NN_index,NN_dists,sigmas,rhos)

    # 构造稀疏矩阵
    result = scipy.sparse.coo_matrix((vals, (rows, cols)), shape=(X.shape[0], X.shape[0]))
    result.eliminate_zeros() # 去掉0

    # 计算联合概率 Pij
    transpose = result.transpose()
    prod_matrix = result.multiply(transpose)
    # Pij  =  Pj|i + Pi|j - Pj|i* Pi|j
    result = result + transpose - prod_matrix
    return result
```

构造稀疏矩阵
返回联合概率

```python
def get_embedding(graph,
                  dim,a,b,
                  negative_sample_rate,
                  n_epochs=None,
                  initial_alpha=1.0):

    # 行列交换
    graph = graph.tocoo()
    graph.sum_duplicates()
    # 顶点数目
    n_vertices = graph.shape[1]

    # 计算迭代轮次 数据越少迭代轮次越多
    if n_epochs is None:
        if graph.shape[0] <= 10000:
            n_epochs = 500
        else:
            n_epochs = 200

    # 边的权重过低，无法采样，将权重设置为0
    if n_epochs > 10:
        graph.data[graph.data < (graph.data.max() / float(n_epochs))] = 0.0

    graph.eliminate_zeros()
```

```python
    # 利用谱分析的方法，借助graph，对低维数据进行初始化
    initialisation = init_embedding_spectral(graph,dim)

    # 加入一些随机数据增加随机性
    expansion = 10.0 / np.abs(initialisation).max()
    embedding = (initialisation * expansion).astype(
        np.float32
    ) + np.random.normal(
        scale=0.0001, size=[graph.shape[0], dim]
    ).astype(
        np.float32
    )

    # 计算图中每条边，每隔多少个epoch 更新一次
    epochs_per_sample = make_epochs_per_sample(graph.data, n_epochs)
    # 负样本，每隔多少个epoch 更新一次
    epochs_per_negative_sample = epochs_per_sample/negative_sample_rate

    # 开始进行训练，获取 embedding
    head = graph.row
    tail = graph.col

    # 训练获取降维数据
    embedding =train_embedding(embedding,embedding,
                               head,tail,
                               epochs_per_sample,epochs_per_negative_sample,
                               a,b,init_alpha,n_epochs,n_vertices)
    return embedding
```

```python
# 谱分析法进行初始化
def init_embedding_spectral(graph,dim):
    n_samples = graph.shape[0]
    k = dim
    diag_data = np.asarray(graph.sum(axis=0))

    # Normalized Laplacian
    I = scipy.sparse.identity(graph.shape[0], dtype=np.float64)
    D = scipy.sparse.spdiags(
        1.0 / np.sqrt(diag_data), 0, graph.shape[0], graph.shape[0]
    )
    L = I - D * graph * D
    num_lanczos_vectors = max(2 * k + 1, int(np.sqrt(graph.shape[0])))
    try:
        if L.shape[0] < 2000000:
            eigenvalues, eigenvectors = scipy.sparse.linalg.eigsh(
                L,
                k,
                which="SM",
                ncv=num_lanczos_vectors,
                tol=1e-4,
                v0=np.ones(L.shape[0]),
                maxiter=graph.shape[0] * 5,
            )
        else:
            print("---------------eigenvalues-------------------")
            eigenvalues, eigenvectors = scipy.sparse.linalg.lobpcg(
                L, np.random.normal(size=(L.shape[0], k)), largest=False, t
            )
        order = np.argsort(eigenvalues)[1:k]
        return eigenvectors[:, order]
```

```python
except scipy.sparse.linalg.ArpackError:
    warn(
        "WARNING: spectral initialisation failed! The eigenvector solver\n"
        "failed. This is likely due to too small an eigengap. Consider\n"
        "adding some noise or jitter to your data.\n\n"
        "Falling back to random initialisation!"
    )
    return np.random.uniform(low=-10.0, high=10.0, size=(graph.shape[0], dim))
```

最小特征值是0

图割的方法，较小特征值映射权重较小的边

```python
def make_epochs_per_sample(weights, n_epochs):
    result = -1.0 * np.ones(weights.shape[0], dtype=np.float64)
    # 边的权重越大在整个训练过程中更新的次数越多，更新间隔越小
    n_samples = n_epochs * (weights / weights.max())
    result[n_samples > 0] = float(n_epochs) / n_samples[n_samples > 0] # 更新间隔
    return result
```

所有边的数目

每条边更新多少次

每隔多久更新一次

隔多久更新一次，
概率越大，间隔越小

总 epoch

```python
# 通过训练获得embedding
def train_embedding(head_embedding, #头结点 向量
                    tail_embedding, #尾结点 向量
                    head, # 头结点 编号
                    tail, # 尾结点 编号
                    epochs_per_sample, # 正样本采样控制
                    epochs_per_negative_sample, # 负样本采样控制
                    a,b, #
                    initial_alpha, #  初始化学习率
                    n_epochs, # 训练轮次
                    n_vertices # 顶点数目
                    ):
    dim = head_embedding.shape[1]
    alpha = initial_alpha

    epoch_of_next_negative_sample = epochs_per_negative_sample.copy()
    epoch_of_next_sample = epochs_per_sample.copy()

    optimize_fn = numba.njit(train_one_epoch, fastmath=True, parallel=False)
```

下次更新的epoch

利用numba 提高
运行速度

```python
for n in tqdm(range(n_epochs)):

    # 进行1轮更新
    optimize_fn(head_embedding,
                tail_embedding,
                head,
                tail,
                n_vertices,
                epochs_per_sample,
                epochs_per_negative_sample,
                epoch_of_next_sample,
                epoch_of_next_negative_sample,
                a,
                b,
                alpha,
                n,
                dim)

    # 更新学习率
    alpha = initial_alpha * (1.0 - (float(n) / float(n_epochs)))

return head_embedding
```

```python
def train_one_epoch(head_embedding,
                    tail_embedding,
                    head,
                    tail,
                    n_vertices,
                    epochs_per_sample,
                    epochs_per_negative_sample,
                    epoch_of_next_sample,
                    epoch_of_next_negative_sample,
                    a,
                    b,
                    alpha,
                    n,
                    dim):

    for i in range(epochs_per_sample.shape[0]):
        #对正样本进行采样
        if epoch_of_next_sample[i] <= n:

            j = head[i]
            k = tail[i]

            current = head_embedding[j]
            other = tail_embedding[k]

            # 计算两点间距离
            dist_squared = np.dot((current-other),(current-other))

            # 计算正样本梯度
            if dist_squared > 0.0:
                grad_coeff = -2.0 * a * b * pow(dist_squared, b - 1.0)
                grad_coeff /= a * pow(dist_squared, b) + 1.0
            else:
                grad_coeff = 0.0
```

```python
# 进行更新
for d in range(dim):
    # 梯度裁剪
    grad_d = clip(grad_coeff * (current[d] - other[d]))
    # 梯度
    current[d] += grad_d * alpha

# 下次更新的轮次
epoch_of_next_sample[i] += epochs_per_sample[i]
```

$$\frac{-2ab\|\mathbf{y_i}-\mathbf{y_j}\|_2^{2(b-1)}}{1+\|\mathbf{y_i}-\mathbf{y_j}\|_2^2}w((x_i,x_j))\,(\mathbf{y_i}-\mathbf{y_j})$$

正样本更新

```python
# 计算负样本的数目
n_neg_samples = int(
    (n - epoch_of_next_negative_sample[i]) / epochs_per_negative_sample[i]
)

# 进行负样本采样
for p in range(n_neg_samples):
    k = np.random.randint(n_vertices)

    other = tail_embedding[k]

    dist_squared = np.dot((current-other),(current-other))

    if dist_squared > 0.0:
        grad_coeff = 2.0 * b
        grad_coeff /= (0.001 + dist_squared) * (
            a * pow(dist_squared, b) + 1
        )
    elif j == k:
        continue
    else:
        grad_coeff = 0.0

    for d in range(dim):
        if grad_coeff > 0.0:
            grad_d = clip(grad_coeff * (current[d] - other[d]))
        else:
            grad_d = 4.0
        current[d] += grad_d * alpha


    # 计算下次负样本更新轮次
    epoch_of_next_negative_sample[i] += (
        n_neg_samples * epochs_per_negative_sample[i]
    )
```

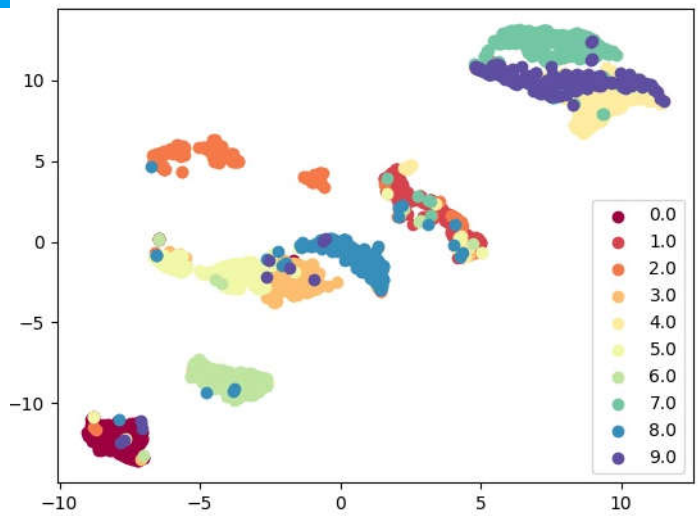
$$\frac{2b}{\left(\epsilon + \|\mathbf{y_i} - \mathbf{y_j}\|_2^2\right)\left(1 + a\|\mathbf{y_i} - \mathbf{y_j}\|_2^{2b}\right)}\left(1 - w((x_i, x_j))\right)\left(\mathbf{y_i} - \mathbf{y_j}\right).$$

```python
def UAMP(X,
        dim=2, # 降维后的维度
        n_neighbors=15, # N近邻
        min_dist = 0.1, # 控制投影后，相似点的聚拢程度
        spread = 1,
        negative_sample_rate=5, # 负样本采样是正样本采样的多少倍
        n_epochs=None, # 训练轮次
        initial_alpha= 1.0 # 初始化学习率
        ):

    # 估算参数 a,b
    a,b = find_ab_params(min_dist,spread)

    # 根据高维数据 计算点与点之间的连接关系
    graph = get_graph_Inputs(X,n_neighbors,local_connectivity=1)
    print(graph)
    #
    embedding = get_embedding(graph,dim,a,b,negative_sample_rate,n_epochs,initial_alpha)
    return embedding
```

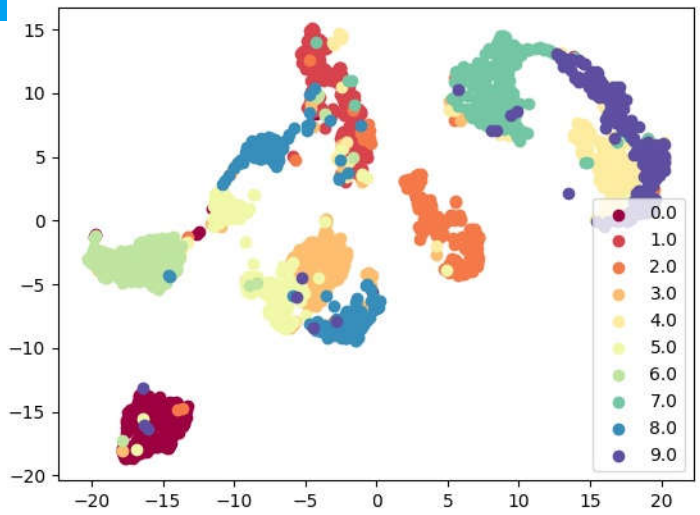
```python
def find_ab_params(min_dist,spread):
    def curve(x, a, b):
        return 1.0 / (1.0 + a * x ** (2 * b))

    xv = np.linspace(0, spread * 3, 300)
    yv = np.zeros(xv.shape)
    yv[xv < min_dist] = 1.0
    yv[xv >= min_dist] = np.exp(-(xv[xv >= min_dist] - min_dist) / spread)
    params, covar = curve_fit(curve, xv, yv)
    return params[0], params[1]
```

$$\Psi(\mathbf{x},\mathbf{y}) = \begin{cases} 1 & \textit{if } \|\mathbf{x}-\mathbf{y}\|_2 \le \textit{min-dist} \\ \exp(-(\|\mathbf{x}-\mathbf{y}\|_2 - \textit{min-dist})) & \textit{otherwise} \end{cases}$$

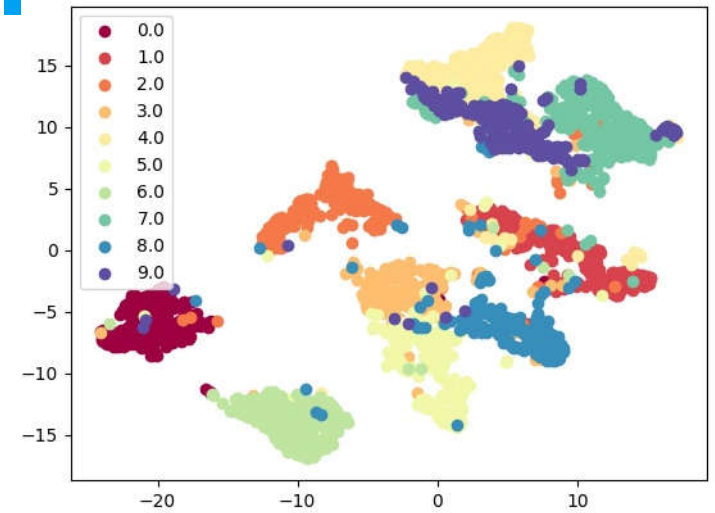
```python
def draw_pic(datas,labs,name = '1.jpg'):
    plt.cla()
    unque_labs = np.unique(labs)
    colors = [plt.cm.Spectral(each)
      for each in np.linspace(0, 1,len(unque_labs))]
    p=[]
    legends = []
    for i in range(len(unque_labs)):
        index = np.where(labs==unque_labs[i])
        pi = plt.scatter(datas[index, 0], datas[index, 1], c =[colors[i]] )
        p.append(pi)
        legends.append(unque_labs[i])

    plt.legend(p, legends)
    plt.savefig(name)


if __name__ == "__main__":
    mnist_datas = np.loadtxt("mnist2500_X.txt")
    mnist_labs = np.loadtxt("mnist2500_labels.txt")
    print(mnist_datas.shape)
    # mnist_datas = mnist_datas[:500,:]
    embedding = UAMP(mnist_datas,dim=2,min_dist=0.3,spread=2)
    print(embedding.shape)

    draw_pic(embedding,mnist_labs,name = "final-d0.01.jpg")
```



min_dist = 0.01



min_dist = 0. 1



min_dist = 0. 3