思源 翻译

2017/11/14 13:37

详解可视化利器 t-SNE 算法：数无形时少直觉

本文介绍了 T 分布随机近邻嵌入算法，即一种十分强大的高维数据降维方法。我们将先简介该算法的基本概念与直观性理解，再从详细分析与实现该降维方法，最后我们会介绍使用该算法执行可视化的结果。

T 分布随机近邻嵌入（T-Distribution Stochastic Neighbour Embedding）是一种用于降维的机器学习方法，它能帮我们识别相关联的模式。t-SNE 主要的优势就是保持局部结构的能力。这意味着高维数据空间中距离相近的点投影到低维中仍然相近。t-SNE 同样能生成漂亮的可视化。

当构建一个预测模型时，第一步一般都需要理解数据。虽然搜索原始数据并计算一些基本的统计学数字特征有助于理解它，但没有什么是可以和图表可视化展示更为直观的。然而将高维数据拟合到一张简单的图表（降维）通常是非常困难的，这就正是 t-SNE 发挥作用的地方。

在本文中，我们将探讨 t-SNE 的原理，以及 t-SNE 将如何有助于我们可视化数据。

t-SNE 算法概念

这篇文章主要是介绍如何使用 t-SNE 进行可视化。虽然我们可以跳过这一章节而生成出漂亮的可视化，但我们还是需要讨论 t-SNE 算法的基本原理。

t-SNE 算法对每个数据点近邻的分布进行建模，其中近邻是指相互靠近数据点的集合。在原始高维空间中，我们将高维空间建模为高斯分布，而在二维输出空间中，我们可以将其建模为 t 分布。该过程

的目标是找到将高维空间映射到二维空间的变换，并且最小化所有点在这两个分布之间的差距。与高斯分布相比 t 分布有较长的尾部，这有助于数据点在二维空间中更均匀地分布。

控制拟合的主要参数为困惑度（Perplexity）。困惑度大致等价于在匹配每个点的原始和拟合分布时考虑的最近邻数，较低的困惑度意味着我们在匹配原分布并拟合每一个数据点到目标分布时只考虑最近的几个最近邻，而较高的困惑度意味着拥有较大的「全局观」。

因为分布是基于距离的，所以所有的数据必须是数值型。我们应该将类别变量通过二值编码或相似的方法转化为数值型变量，并且归一化数据也是也十分有效，因为归一化数据后就不会出现变量的取值范围相差过大。

T 分布随机近邻嵌入算法（**t-SNE**）

Jake Hoare 的博客并没有详细解释 t-SNE 的具体原理和推导过程，因此下面我们将基于 Geoffrey Hinton 在 2008 年提出的论文和 liam schoneveld 的推导与实现详细介绍 t-SNE 算法。如果读者对这一章节不感兴趣，也可以直接阅读下一章节 Jake Hoare 在实践中使用 t-SNE 进行数据可视化。

- liam schoneveld 推导与实现地址：https://nlml.github.io/in-raw-numpy/in-raw-numpy-t-sne/
- 论文地址：http://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf

因为 t-SNE 是基于随机近邻嵌入而实现的，所以首先我们需要理解随机近邻嵌入算法。

随机近邻嵌入（**SNE**）

假设我们有数据集 X，它共有 N 个数据点。每一个数据点 x_i 的维度为 D，我们希望降低为 d 维。在一般用于可视化的条件下，d 的取值为 2，即在平面上表示出所有数据。

SNE p_j|i

通过将数据点间的欧几里德距离转化为条件概率而表征相似性（下文用　　表示）：

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

如果以数据点在 x_i 为中心的高斯分布所占的概率密度为标准选择近邻，那么 p_j|i 就代表 x_i 将选择 x_j 作为它的近邻。对于相近的数据点，条件概率 p_j|i 是相对较高的，然而对于分离的数据点，p_j|i 几乎是无穷小量（若高斯分布的方差σ_i 选择合理）。

其中σ_i 是以数据点 x_i 为均值的高斯分布标准差，决定σ_i 值的方法将在本章后一部分讨论。因为我们只对成对相似性的建模感兴趣，所以可以令 p_i|i 的值为零。

现在引入矩阵 Y，Y 是 N*2 阶矩阵，即输入矩阵 X 的 2 维表征。基于矩阵 Y，我们可以构建一个分布 q，其形式与 p 类似。

对于高维数据点 x_i 和 x_j 在低维空间中的映射点 y_i 和 y_j，计算一个相似的条件概率 q_j|i 是可以实现的。我们将计算条件概率 q_i|j 中用到的高斯分布的方差设置为 1/2。因此我们可以对映射的低维数据点 y_j 和 y_i 之间的相似度进行建模：

$$q_{j|i} = \frac{\exp\left(-\|y_i - y_j\|^2\right)}{\sum_{k \neq i} \exp\left(-\|y_i - y_k\|^2\right)}.$$

我们的总体目标是选择 Y 中的一个数据点，然后其令条件概率分布 q 近似于 p。这一步可以通过最小化两个分布之间的 KL 散度（损失函数）而实现，这一过程可以定义为：

$$C = \sum_i KL(P_i||Q_i) = \sum_i \sum_j p_{j|i} l$$

因为我们希望能最小化该损失函数，所以我们可以使用梯度下降进行迭代更新，我们可能对如何迭代感兴趣，但我们会在后文讨论与实现。

使用 **NumPy** 构建欧几里德距离矩阵

计算 p_ij 和 q_ij 的公式都存在负的欧几里德距离平方，即-||x_i - x_j||^2，下面可以使用代码实现这一部分：

```
def neg_squared_euc_dists(X):
```

```
    """Compute matrix containing negative squared euclidean
    distance for all pairs of points in input matrix X

    # Arguments:
        X: matrix of size NxD
    # Returns:
        NxN matrix D, with entry D_ij = negative squared
        euclidean distance between rows X_i and X_j
    """
    # Math? See https://stackoverflow.com/questions/37009647
    sum_X = np.sum(np.square(X), 1)
    D = np.add(np.add(-2 * np.dot(X, X.T), sum_X).T, sum_X)
    return -D
```

为了更高的计算效率，该函数使用矩阵运算的方式定义，该函数将返回一个 N 阶方阵，其中第 i 行第 j 列个元素为输入点 x_i 和 x_j 之间的负欧几里德距离平方。

使用过神经网络的读者可能熟悉 exp(·)/∑exp(·) 这样的表达形式，它是一种 softmax 函数，所以我们定义一个 softmax 函数：

```
def softmax(X, diag_zero=True):
    """Take softmax of each row of matrix X."""

    # Subtract max for numerical stability
    e_x = np.exp(X - np.max(X, axis=1).reshape([-1, 1]))

    # We usually want diagonal probailities to be 0.
    if diag_zero:
        np.fill_diagonal(e_x, 0.)

    # Add a tiny constant for stability of log we take later
    e_x = e_x + 1e-8  # numerical stability
```

```
    return e_x / e_x.sum(axis=1).reshape([-1, 1])
```

注意我们需要考虑 p_i|i=0 这个条件，所以我们可以替换指数负距离矩阵的对角元素为 0，即使用 np.fill_diagonal(e_x, 0.) 方法将 e_x 的对角线填充为 0。

将这两个函数放在一起后，我们能构建一个函数给出矩阵 P，且元素 P(i,j) 为上式定义的 p_i|j：

```
def calc_prob_matrix(distances, sigmas=None):
    """Convert a distances matrix to a matrix of probabilities."""
    if sigmas is not None:
        two_sig_sq = 2. * np.square(sigmas.reshape((-1, 1)))
        return softmax(distances / two_sig_sq)
    else:
        return softmax(distances)
```

困惑度

在上面的代码段中，Sigmas 参数必须是长度为 N 的向量，且包含了每一个σ_i 的值，那么我们如何取得这些σ_i 呢？这就是困惑度（perplexity）在 SNE 中的作用。条件概率矩阵 P 任意行的困惑度可以定义为：

$$Perp(P_i) = 2^{H(P_i)},$$

其中 H(P_i) 为 P_i 的香农熵，即表达式如下：

$$H(P_i) = -\sum_j p_{j|i} \log_2 p_{j|i}.$$

在 SNE 和 t-SNE 中，困惑度是我们设置的参数（通常为 5 到 50 间）。我们可以为矩阵 P 的每行设置一个σ_i，而该行的困惑度就等于我们设置的这个参数。直观来说，如果概率分布的熵较大，那么其分布的形状就相对平坦，该分布中每个元素的概率就更相近一些。

困惑度随着熵增而变大，因此如果我们希望有更高的困惑度，那么所有的 p_j|i（对于给定的 i）就会彼此更相近一些。换而言之，如果我们希望概率分布 P_i 更加平坦，那么我们就可以增大σ_i。我们配置的σ_i 越大，概率分布中所有元素的出现概率就越接近于 1/N。实际上增大σ_i 会增加每个点的近邻数，这就是为什么我们常将困惑度参数大致等同于所需要的近邻数量。

搜索**σ_i**

为了确保矩阵 P 每一行的困惑度 Perp(P_i) 就等于我们所希望的值，我们可以简单地执行一个二元搜索以确定σ_i 能得到我们所希望的困惑度。这一搜索十分简单，因为困惑度 Perp(P_i) 是随σ_i 增加而递增的函数，下面是基本的二元搜索函数：

```
def binary_search(eval_fn, target, tol=1e-10, max_iter=10000,
                  lower=1e-20, upper=1000.):
    """Perform a binary search over input values to eval_fn.

    # Arguments
        eval_fn: Function that we are optimising over.
        target: Target value we want the function to output.
        tol: Float, once our guess is this close to target, stop.
        max_iter: Integer, maximum num. iterations to search for.
        lower: Float, lower bound of search range.
```

```
        upper: Float, upper bound of search range.
    # Returns:
        Float, best input value to function found during search.
    """
    for i in range(max_iter):
        guess = (lower + upper) / 2.
        val = eval_fn(guess)
        if val > target:
            upper = guess
        else:
            lower = guess
        if np.abs(val - target) <= tol:
            break
    return guess
```

为了找到期望的σ_i，我们需要将 eval_fn 传递到 binary_search 函数，并且将σ_i 作为它的参数而返回 P_i 的困惑度。

以下的 find_optimal_sigmas 函数确实是这样做的以搜索所有的σ_i，该函数需要采用负欧几里德距离矩阵和目标困惑度作为输入。距离矩阵的每一行对所有可能的σ_i 都会执行一个二元搜索以找到能产生目标困惑度的最优σ。该函数最后将返回包含所有最优σ_i 的 NumPy 向量。

```
def calc_perplexity(prob_matrix):
    """Calculate the perplexity of each row
    of a matrix of probabilities."""
    entropy = -np.sum(prob_matrix * np.log2(prob_matrix), 1)
    perplexity = 2 ** entropy
    return perplexity


def perplexity(distances, sigmas):
    """Wrapper function for quick calculation of
    perplexity over a distance matrix."""
```

```
    return calc_perplexity(calc_prob_matrix(distances, sigmas))


def find_optimal_sigmas(distances, target_perplexity):
    """For each row of distances matrix, find sigma that results
    in target perplexity for that role."""
    sigmas = []
    # For each row of the matrix (each point in our dataset)
    for i in range(distances.shape[0]):
        # Make fn that returns perplexity of this row given sigma
        eval_fn = lambda sigma: \
            perplexity(distances[i:i+1, :], np.array(sigma))
        # Binary search over sigmas to achieve target perplexity
        correct_sigma = binary_search(eval_fn, target_perplexity)
        # Append the resulting sigma to our output array
        sigmas.append(correct_sigma)
    return np.array(sigmas)
```

对称 **SNE**

现在估计 SNE 的所有条件都已经声明了，我们能通过降低成本 C 对 Y 的梯度而收敛到一个良好的二维表征 Y。因为 SNE 的梯度实现起来比较难，所以我们可以使用对称 SNE，对称 SNE 是 t-SNE 论文中一种替代方法。

在对称 SNE 中，我们最小化 p_ij 和 q_ij 的联合概率分布与 p_i|j 和 q_i|j 的条件概率之间的 KL 散度，我们定义的联合概率分布 q_ij 为：

$$q_{ij} = \frac{\exp\left(-\|y_i - y_j\|^2\right)}{\sum_{k \neq l} \exp\left(-\|y_k - y_l\|^2\right)},$$

该表达式就如同我们前面定义的 softmax 函数，只不过分母中的求和是对整个矩阵进行的，而不是当前的行。为了避免涉及到 x 点的异常值，我们不是令 p_ij 服从相似的分布，而是简单地令 p_ij=(p_i|j+p_j|i)/2N。

我们可以简单地编写这些联合概率分布 q 和 p：

```
def q_joint(Y):
    """Given low-dimensional representations Y, compute
    matrix of joint probabilities with entries q_ij."""
    # Get the distances from every point to every other
    distances = neg_squared_euc_dists(Y)
    # Take the elementwise exponent
    exp_distances = np.exp(distances)
    # Fill diagonal with zeroes so q_ii = 0
    np.fill_diagonal(exp_distances, 0.)
    # Divide by the sum of the entire exponentiated matrix
    return exp_distances / np.sum(exp_distances), None


def p_conditional_to_joint(P):
    """Given conditional probabilities matrix P, return
    approximation of joint distribution probabilities."""
    return (P + P.T) / (2. * P.shape[0])
```

同样可以定义 p_joint 函数输入数据矩阵 X 并返回联合概率 P 的矩阵，此外我们还能一同估计要求的σ_i 和条件概率矩阵：

```
def p_joint(X, target_perplexity):
    """Given a data matrix X, gives joint probabilities matrix.

    # Arguments
        X: Input data matrix.
    # Returns:
        P: Matrix with entries p_ij = joint probabilities.
    """
    # Get the negative euclidian distances matrix for our data
    distances = neg_squared_euc_dists(X)
    # Find optimal sigma for each row of this distances matrix
    sigmas = find_optimal_sigmas(distances, target_perplexity)
    # Calculate the probabilities based on these optimal sigmas
    p_conditional = calc_prob_matrix(distances, sigmas)
    # Go from conditional to joint probabilities matrix
    P = p_conditional_to_joint(p_conditional)
    return P
```

所以现在已经定义了联合概率分布 p 与 q，若我们计算了这两个联合分布，那么我们能使用以下梯度更新低维表征 Y 的第 i 行：

$$\frac{\delta C}{\delta y_i} = 4\sum_j (p_{ij} - q_{ij})(y_i - y_j).$$

在 Python 中，我们能使用以下函数估计梯度，即给定联合概率矩阵 P、Q 和当前低维表征 Y 估计梯度：

```
def symmetric_sne_grad(P, Q, Y, _):
    """Estimate the gradient of the cost with respect to Y"""
    pq_diff = P - Q  # NxN matrix
    pq_expanded = np.expand_dims(pq_diff, 2)  #NxNx1
```

```
    y_diffs = np.expand_dims(Y, 1) - np.expand_dims(Y, 0)  #NxNx2
    grad = 4. * (pq_expanded * y_diffs).sum(1)  #Nx2
    return grad
```

为了向量化变量，np.expand_dims 方法将十分有用，该函数最后返回的 grad 为 N*2 阶矩阵，其中第 i 行为 dC/dy_i。一旦我们计算完梯度，那么我们就能利用它执行梯度下降，即通过梯度下降迭代式更新 y_i。

估计对称 **SNE**

前面已经定义了所有的估计对称 SNE 所需要的函数，下面的训练函数将使用梯度下降算法迭代地计算与更新权重。

```
def estimate_sne(X, y, P, rng, num_iters, q_fn, grad_fn, learning_rate,
                 momentum, plot):
    """Estimates a SNE model.

    # Arguments
        X: Input data matrix.
        y: Class labels for that matrix.
        P: Matrix of joint probabilities.
        rng: np.random.RandomState().
        num_iters: Iterations to train for.
        q_fn: Function that takes Y and gives Q prob matrix.
        plot: How many times to plot during training.
    # Returns:
        Y: Matrix, low-dimensional representation of X.
    """

    # Initialise our 2D representation
    Y = rng.normal(0., 0.0001, [X.shape[0], 2])
```

```
    # Initialise past values (used for momentum)
    if momentum:
        Y_m2 = Y.copy()
        Y_m1 = Y.copy()

    # Start gradient descent loop
    for i in range(num_iters):

        # Get Q and distances (distances only used for t-SNE)
        Q, distances = q_fn(Y)
        # Estimate gradients with respect to Y
        grads = grad_fn(P, Q, Y, distances)

        # Update Y
        Y = Y - learning_rate * grads
        if momentum:  # Add momentum
            Y += momentum * (Y_m1 - Y_m2)
            # Update previous Y's for momentum
            Y_m2 = Y_m1.copy()
            Y_m1 = Y.copy()

        # Plot sometimes
        if plot and i % (num_iters / plot) == 0:
            categorical_scatter_2d(Y, y, alpha=1.0, ms=6,
                                   show=True, figsize=(9, 6))

    return Y
```

为了简化表达，我们将使用 MNIST 数据集中标签为 0、1 和 8 的 200 个数据点，该过程定义在 main() 函数中：

```
# Set global parameters
```

```
NUM_POINTS = 200            # Number of samples from MNIST
CLASSES_TO_USE = [0, 1, 8]  # MNIST classes to use
PERPLEXITY = 20
SEED = 1                    # Random seed
MOMENTUM = 0.9
LEARNING_RATE = 10.
NUM_ITERS = 500             # Num iterations to train for
TSNE = False                # If False, Symmetric SNE
NUM_PLOTS = 5               # Num. times to plot in training



def main():
    # numpy RandomState for reproducibility
    rng = np.random.RandomState(SEED)

    # Load the first NUM_POINTS 0's, 1's and 8's from MNIST
    X, y = load_mnist('datasets/',
                      digits_to_keep=CLASSES_TO_USE,
                      N=NUM_POINTS)

    # Obtain matrix of joint probabilities p_ij
    P = p_joint(X, PERPLEXITY)

    # Fit SNE or t-SNE
    Y = estimate_sne(X, y, P, rng,
             num_iters=NUM_ITERS,
             q_fn=q_tsne if TSNE else q_joint,
             grad_fn=tsne_grad if TSNE else symmetric_sne_grad,
             learning_rate=LEARNING_RATE,
             momentum=MOMENTUM,
             plot=NUM_PLOTS)
```

构建 **t-SNE**

前面我们已经分析了很多关于随机近邻嵌入的方法与概念，并推导出了对称 SNE，不过幸运的是对称 SNE 扩展到 t-SNE 是非常简单的。真正的区别仅仅是我们定义联合概率分布矩阵 Q 的方式，在 t-SNE 中，我们 q_ij 的定义方法可以变化为：

$$q_{ij} = \frac{\exp\left(-\|y_i - y_j\|^2\right)}{\sum_{k \neq l} \exp\left(-\|y_k - y_l\|^2\right)},$$

上式通过假设 q_ij 服从自由度为 1 的学生 T 分布（Student t-distribution）而推导出来。Van der Maaten 和 Hinton 注意到该分布有非常好的一个属性，即计数器（numerator）对于较大距离在低维空间中具有反平方变化规律。本质上，这意味着该算法对于低维映射的一般尺度具有不变性。因此，最优化对于相距较远的点和相距较近的点都有相同的执行方式。

这就解决了所谓的「拥挤问题」，即当我们试图将一个高维数据集表征为 2 或 3 个维度时，很难将邻近的数据点与中等距离的数据点区分开来，因为这些数据点都聚集在一块区域。

我们能使用以下函数计算新的 q_ij：

```
def q_tsne(Y):
    """t-SNE: Given low-dimensional representations Y, compute
    matrix of joint probabilities with entries q_ij."""
    distances = neg_squared_euc_dists(Y)
    inv_distances = np.power(1. - distances, -1)
    np.fill_diagonal(inv_distances, 0.)
    return inv_distances / np.sum(inv_distances), inv_distances
```

注意我们使用 1. - distances 代替 1. + distances，该距离函数将返回一个负的距离。现在剩下的就是重

新估计损失函数对 Y 的梯度，t-SNE 论文中推导该梯度的表达式为：

$$\frac{\delta C}{\delta y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j).$$

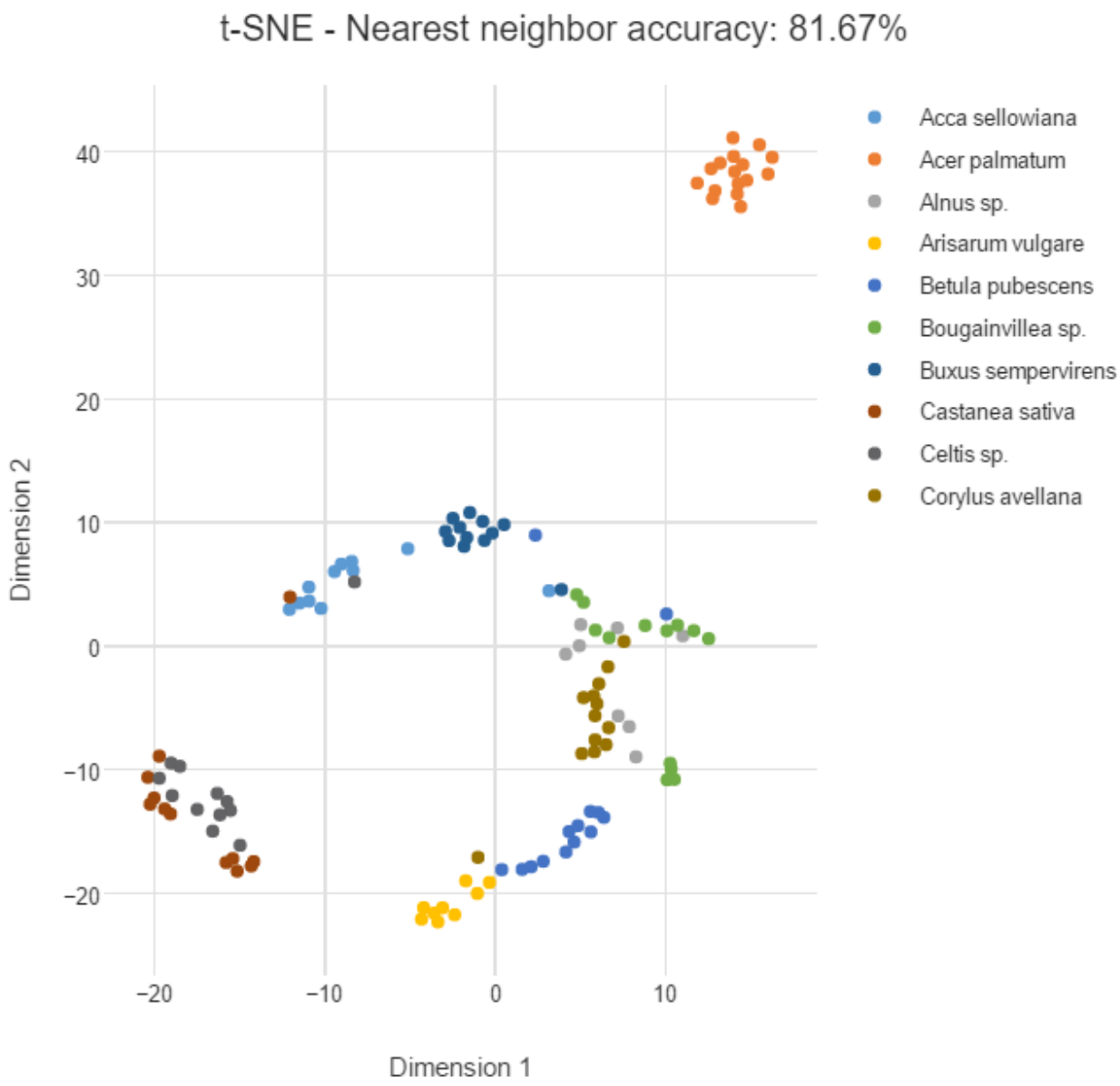
同样，我们很容易按照计算对称 SNE 梯度的方式构建 t-SNE 的梯度计算方式：

```
def tsne_grad(P, Q, Y, inv_distances):
    """Estimate the gradient of t-SNE cost with respect to Y."""
    pq_diff = P - Q
    pq_expanded = np.expand_dims(pq_diff, 2)
    y_diffs = np.expand_dims(Y, 1) - np.expand_dims(Y, 0)

    # Expand our inv_distances matrix so can multiply by y_diffs
    distances_expanded = np.expand_dims(inv_distances, 2)

    # Multiply this by inverse distances matrix
    y_diffs_wt = y_diffs * distances_expanded

    # Multiply then sum over j's
    grad = 4. * (pq_expanded * y_diffs_wt).sum(1)
    return grad
```

以上我们就完成了 t-SNE 的具体理解与实现，那么该算法在具体数据集中的可视化效果是怎样的呢？Jake Hoare 给出了实现可视化的效果与对比。

t-SNE 可视化

下面，我们将要展示 t-SNE 可视化高维数据的结果，第一个数据集是基于物理特征分类的 10 种不同叶片。这种情况下，t-SNE 需要使用 14 个数值变量作为输入，其中就包括叶片的生长率和长宽比等。下图展示了 2 维可视化输出，植物的种类（标签）使用不同的颜色表达。

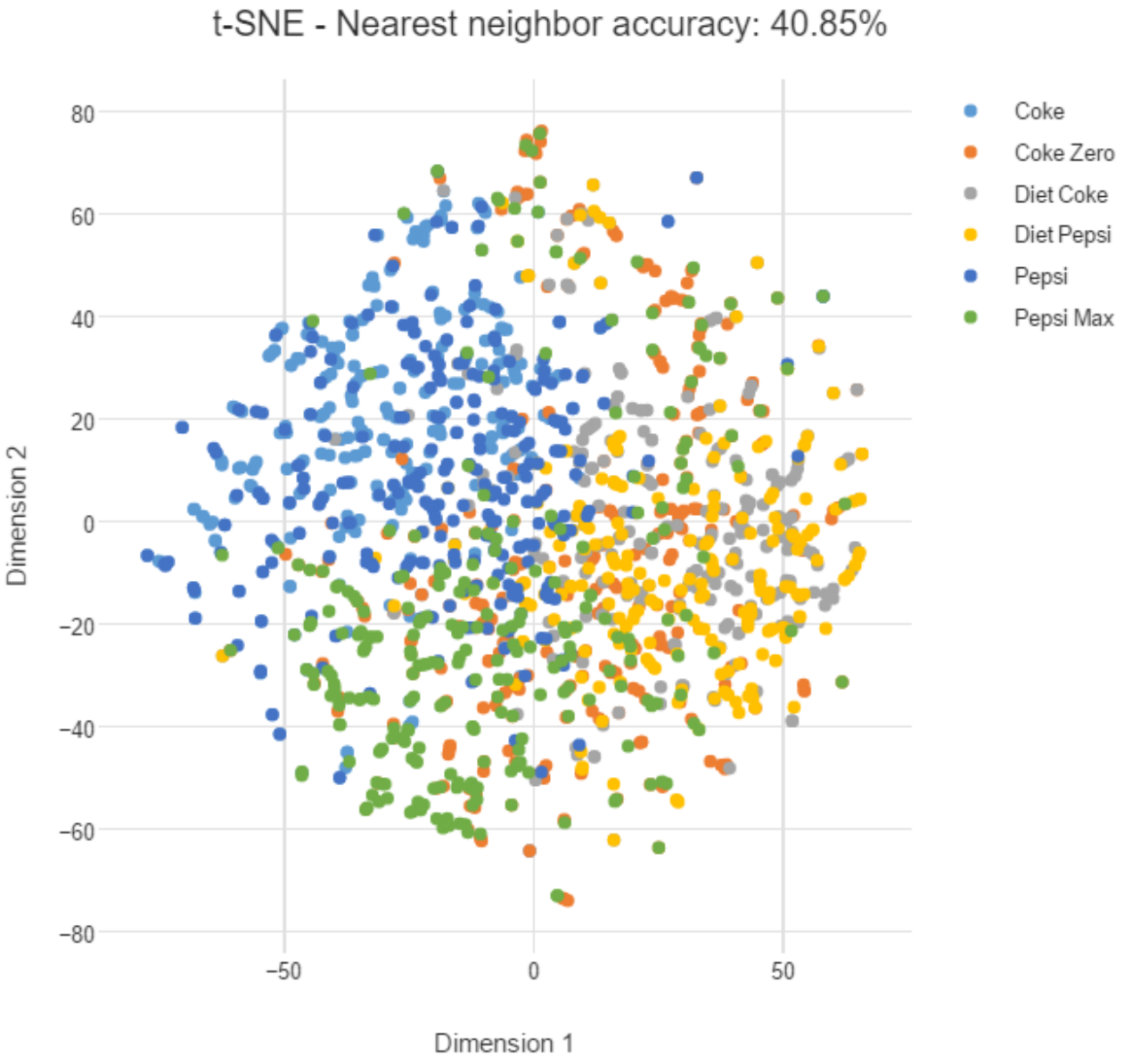


物种 Acer palmatum 的数据点在右上角形成了一个橙色集群，这表明它的叶子和其他物种有很大的不同。该示例中类别通常会有很好的分组，相同物种的叶子（同一颜色的数据点）趋向于彼此靠紧聚集在一起。左下角有两种颜色的数据点靠近在一起，说明这两个物种的叶子形状十分相似。

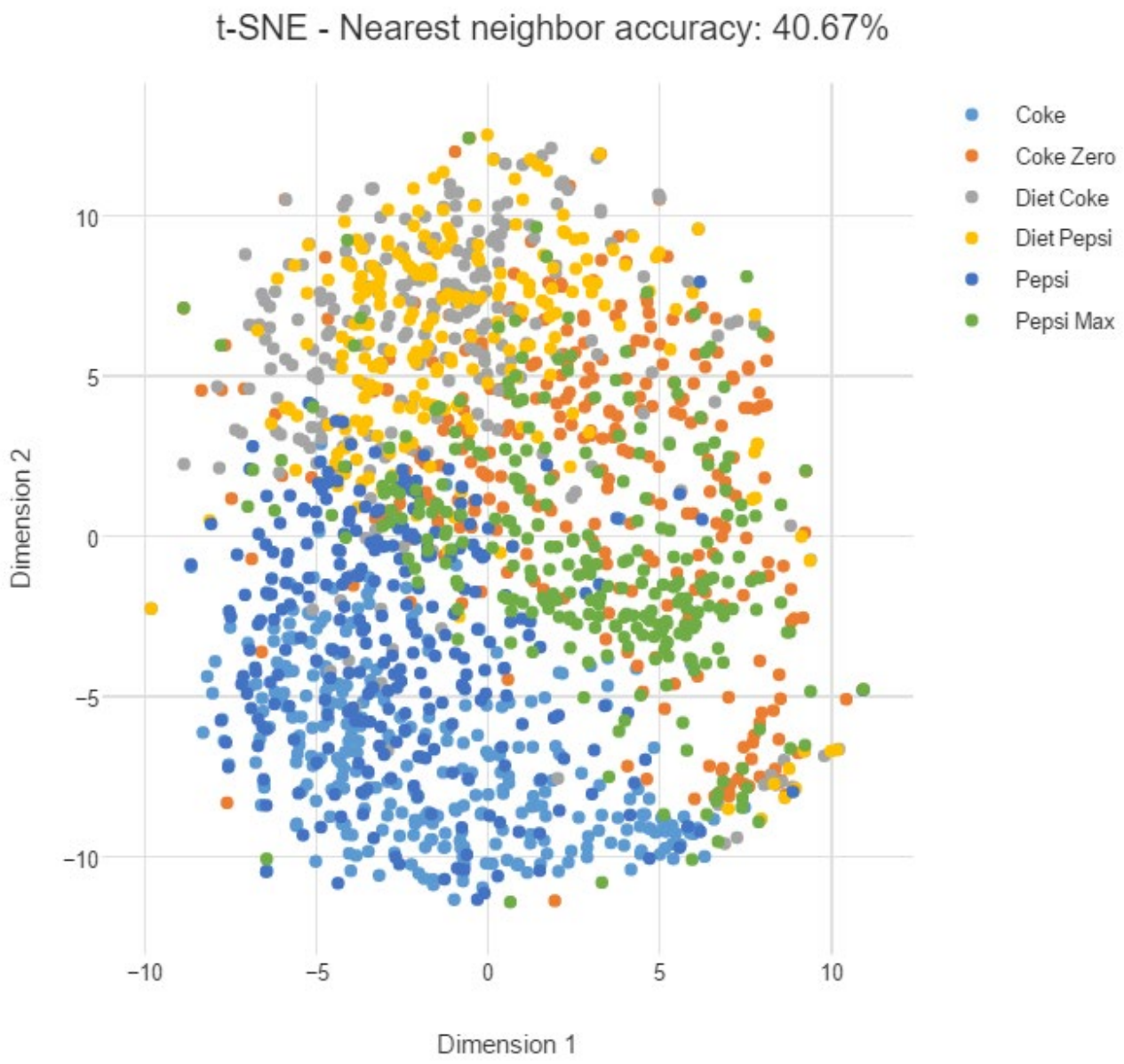
最近邻准确度表明给定两个随机点，它们是相同物种的概率是多少。如果这些数据点完美地根据不同物种而分类，那么准确度就会非常接近 100%，高准确度意味着数据能被干净地分为不同的集群。

调整困惑度

下面，我们对可乐品牌做了类似的分析。为了演示困惑度（perplexity）的影响，我们首先需要将困惑度设置为较低的值 2，每个数据点的映射只考虑最近邻。如下，我们将看到许多离散的小集群，并且每一个集群只有少量的数据点。



下面我们将 t-SNE 的困惑度设置为 100，我们可以看到数据点变得更加扩散，并且同一类之间的联系变弱。



在该案例中，可乐本身就要比树叶更难分割，即使一类数据点某个品牌要更集中一些，但仍然没有明确的边界。

在实践中，困惑度并没以一个绝对的标准，不过一般选择 5 到 50 之间会有比较好的结果。并且在这个范围内，t-SNE 一般是比较鲁棒的。

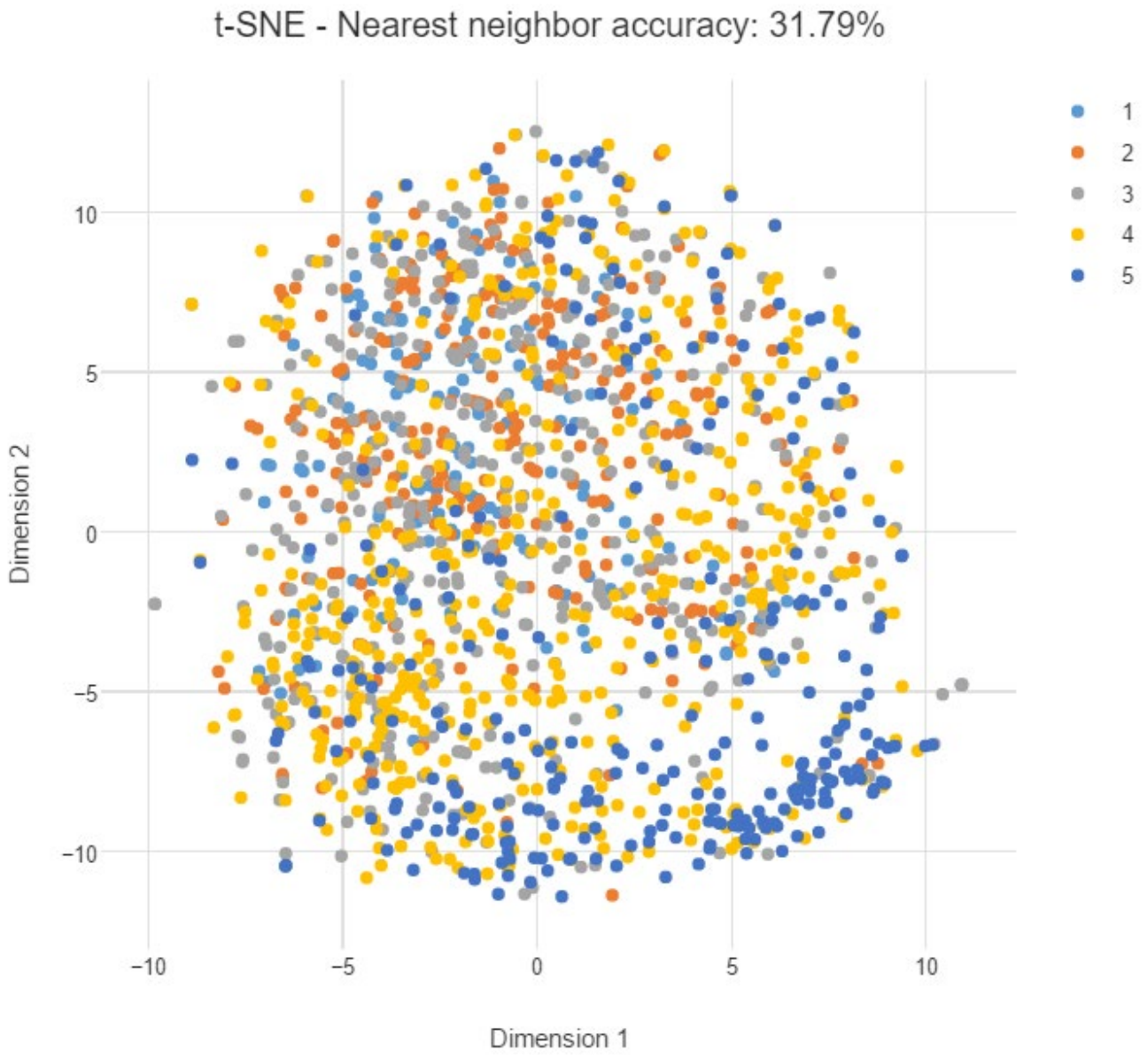
预测的解释

度量数据点之间的角度或距离并不能推断出任何数据上的具体或定量的信息，所以 t-SNE 的预测更多

的是用于可视化数据。

在模型搭建前期直观地挖掘数据模式有助于指导数据科学下一步进程。如果 t-SNE 能很好地分割数据点，那么机器学习同样也能找到一种将未知新数据投影到相应类别的好方法。给定一种预测算法，我们就能实现很高的准确度。

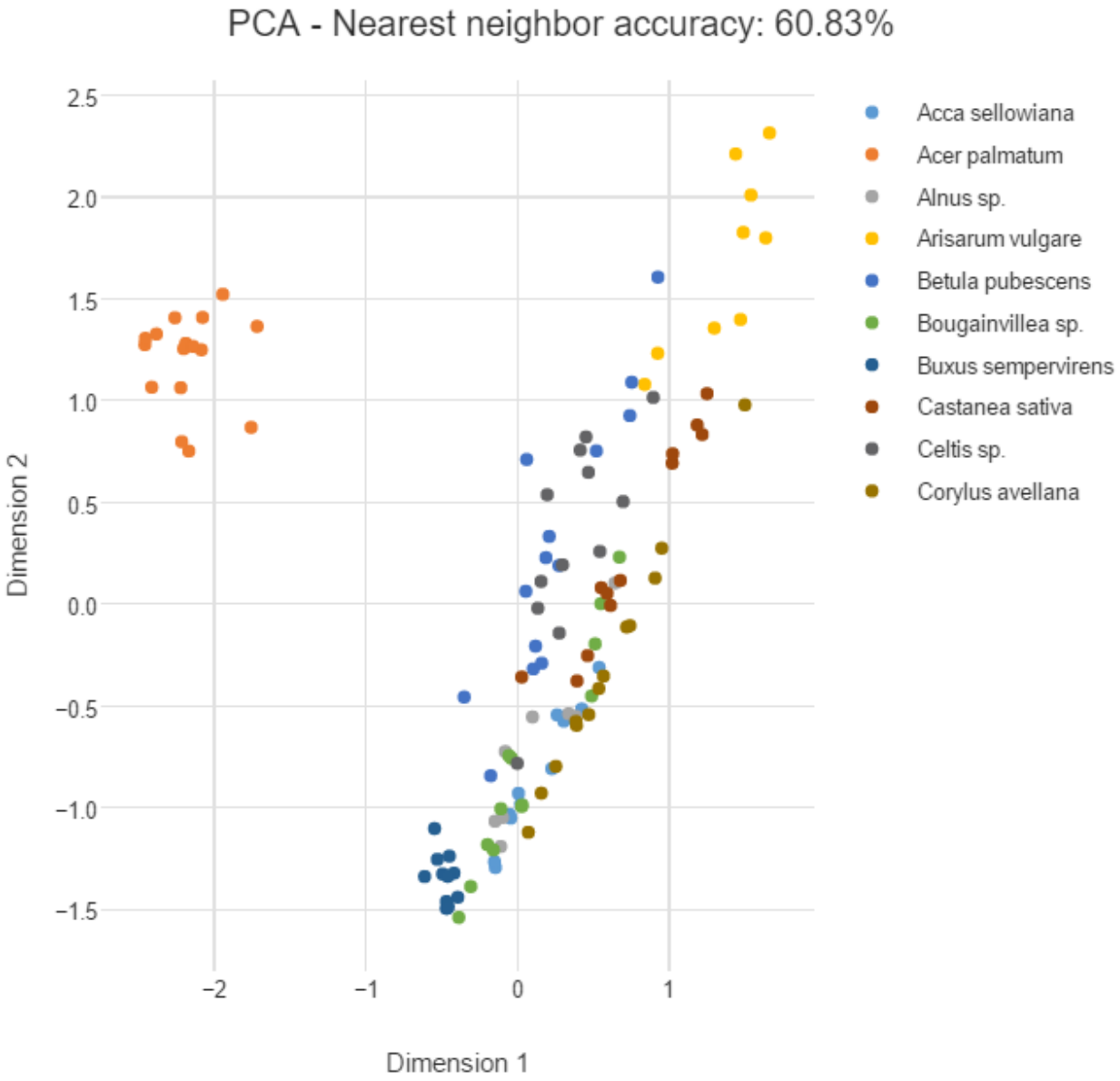
上例中每一个类别都是孤立分类的，因此简单的机器学习模型就能将该类别与其他类别分离开。但如果类别重叠，我们可能就要构建更精细的模型做出预测。如下所示，如果我们按照某个品牌的偏好从 1 到 5 进行分类，那么类别可能更加离散、更难以预测，最近邻精度也会相对较低。



对比 PCA

很自然我们就希望将 t-SNE 和其他降维算法做比较。降维算法中比较流行的是主成分分析法（PCA），PCA 会寻找能尽可能保留数据方差的新维度。有较大的方差的数据保留的信息就越多，因为彼此不同的数据可以提供不同的信息，所以我们最好是保留彼此分离的数据点，因为它们提供了较大的方差。

下面是采用 PCA 算法对上文的树叶类别进行降维的结果，我们看到虽然左侧的橙色是分离的，但其它类别都有点混合在一起。这是因为 PCA 并不关心局部的最近邻关系，此外 PCA 是一种线性方法，所以它表征非线性关系可能效果并不是很好。不过 PCA 算法在压缩数据为更小的特征向量而投入到预测算法中有很好地表现，这一点它要比 t-SNE 效果更好。



结语

t-SNE 是一种可视化高维数据的优秀算法，它经常要比其它降维算法生成更具特点的可视化结果。在数据分析中，获得数据的先验知识总是很重要的，正如华罗庚先生说过：数无形时少直觉，形少数时难入微，我们只有先理解了数据的大概分布，然后再能选择具体的算法对这些数据进一步分析。数形结合百般好，隔离分家万事休，也许高维数据的可视化与机器学习算法的结合才是数据分析的正确打开方式。

产业 工程 可视化 机器学习 降维算法

2 2

登录 后评论

zsh_o Pro

t分布 公式写错了

wuli

可以求第一个叶片的数据集吗?