INFO 7390 Advances data science/Architecture

Leaf recognition

Name: Weiyi Lan, Yiqun Xu, Yang Zong

1.Introduction:

There are estimated to be nearly half a million species of plant all over the world. Here comes a problem that how can we classify these leaves, in other words, how can we get to know the name of leaves quickly and easily? Maybe we can use the knowledge in machine learning to do that. Machine learning has been a trending research and experimentation topic recently. But It is still a challenge when it comes to computer version.

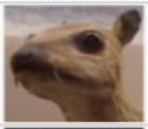
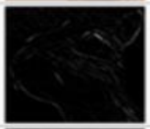
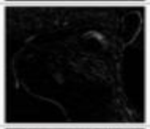
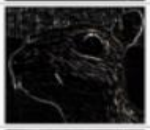
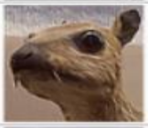
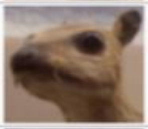
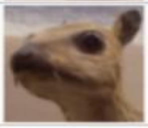
Automating plant recognition might have many applications, including: Species population tracking and preservation, Plant-based medicinal research, Crop and food supply management and so on. If people can easily recognize various kinds of leaves, all the people can get full use of leaves very well.

So, we use neural network, which is a system of computer software that is patterned after the working of neurons in the human being. Deep learning refers to a subdivision of machine learning.one of the most popular deep learning technique is a convolutional neural network(CNN).it is commonly used for solving problems related to computer version. Our project is using CNN to recognize which kinds of leaves they are. And then when we get high enough accuracy, we will cut our dataset, in other words, we will do some works to reduce the dimension of our dataset and then test the accuracy of classifying the leaves, in that case we can make sure whether we can cut down some parameter and keep the accuracy as high as before. How to make the process above works, we will explain below.

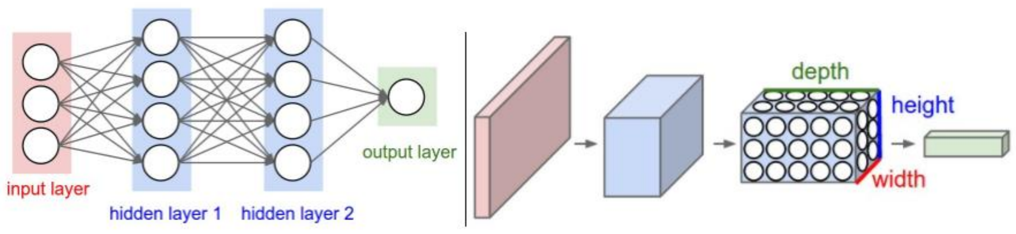
2.Background Research of Related Work:

1. **Convolutional Neural Networks (CNNs / ConvNets) :**

    (1.) The picture below shows the effects of convolution of the above image with different filters. As we can perform operations such as Edge Detection, Sharpen and Blur just by changing the numeric values of our filter matrix before the convolution operation – this means that different filters can detect different features from an image, for example edges, curves, etc.

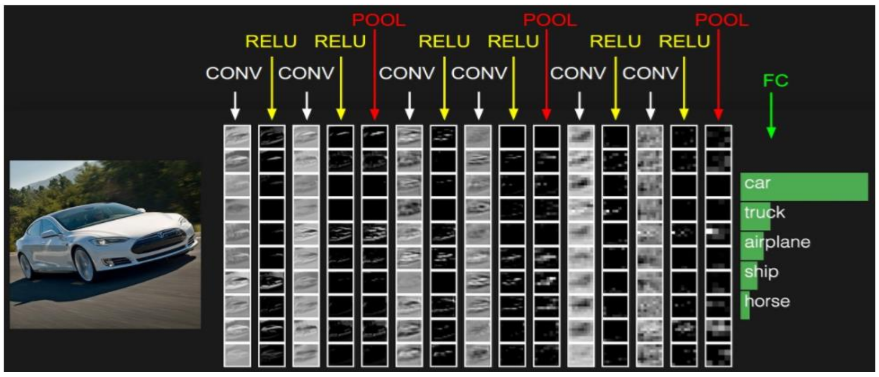
Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

(2.)CNN are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. It makes the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the number of parameters in the network.

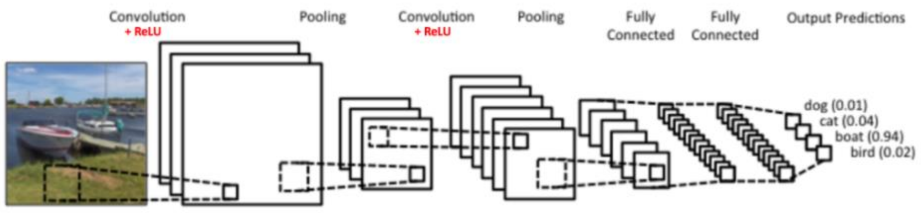


Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations.

(2). We will stack three layers, **Convolutional Layer**, **Pooling Layer**, and **Fully-Connected Layer** to form a full ConvNet **architecture**.



The initial volume stores the raw image pixels (left) and the last volume stores the class scores (right). Each volume of activations along the processing path is shown as a column.



(3). The Conv Layer:

Accepts a volume of size W1×H1×D1W1×H1×D1

Requires four hyperparameters:

- Number of filters KK,
- their spatial extent FF,
- the stride SS,
- the amount of zero padding PP.

Produces a volume of size W2×H2×D2W2×H2×D2 where:

- W2=(W1−F+2P)/S+1W2=(W1−F+2P)/S+1
- H2=(H1−F+2P)/S+1H2=(H1−F+2P)/S+1 (i.e. width and height are computed equally by symmetry)
- D2=KD2=K

With parameter sharing, it introduces F⋅F⋅D1F⋅F⋅D1 weights per filter, for a total of (F⋅F⋅D1)⋅K(F⋅F⋅D1)⋅K weights and KK biases.

In the output volume, the dd-th depth slice (of size W2×H2W2×H2) is the result of performing a valid convolution of the dd-th filter over the input volume with a stride of SS, and then offset by dd-th bias.

A common setting of the hyperparameters is F=3,S=1,P=1F=3,S=1,P=1. However, there are common conventions and rules of thumb that motivate these hyperparameters

(4).Pooling Layer:

Accepts a volume of size W1×H1×D1W1×H1×D1

Requires two hyperparameters:

- their spatial extent FF,
- the stride SS,

Produces a volume of size W2×H2×D2W2×H2×D2 where:

- W2=(W1−F)/S+1W2=(W1−F)/S+1
- H2=(H1−F)/S+1H2=(H1−F)/S+1
- D2=D1D2=D1

Note that it is not common to use zero-padding for Pooling layers

It is worth noting that there are only two commonly seen variations of the max pooling layer found in practice: A pooling layer with F=3,S=2F=3,S=2, and more commonly F=2,S=2F=2,S=2. Pooling sizes with larger receptive fields are too destructive.

(4) **Introduction of how CNN works:**

Step 1: Break the image into overlapping image tiles

Step 2: Feed each image tile into a small neural network

Step 3: Save the results from each tile into a new array

Step 4: Down sampling

Last step: Make a prediction

2. TensorFlow s an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers from the Google Brain team within Google's AI organization, it comes with dedicated support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains.

3. Gaussian Model

(1). What is a Gaussian process?

A time continuous stochastic process is Gaussian if and only if for every finite set of indices in the index set is a multivariate Gaussian random variable. That is the same as saying every linear combination of has a univariate normal (or Gaussian) distribution. Using characteristic functions of random variables, the Gaussian property can be formulated as follows: is Gaussian if and only if, for every finite set of indices , there are real-valued , with such that the following equality holds for all where denotes the imaginary number .

The numbers and can be shown to be the covariances and means of the variables in the process.

$$\mathrm{E}\left(\exp\left(i\sum_{\ell=1}^{k} s_\ell\,\mathbf{X}_{t_\ell}\right)\right)=\exp\left(-\frac{1}{2}\sum_{\ell,j}\sigma_{\ell j}s_\ell s_j+i\sum_{\ell}\mu_\ell s_\ell\right).$$

• Continuous stochastic process — random functions — a set of random variables indexed by a continuous variable: f(x)
• Set of ‘inputs’ X = {x1, x2, . . . , xN}; corresponding set of random function variables f = {f1, f2, . . . , fN}
• GP: Any set of function variables {fn} N n=1 has joint (zero mean) Gaussian distribution: p(f|X) = N (0, K)

• Conditional model - density of inputs not modeled
• Consistency: p(f1) = Z df2 p(f1, f2)

(2). Constructing new covariances from old

• Sum: K(x, x0) = K1(x, x0) + K2(x, x0) addition of independent processes
• Product: K(x, x0) = K1(x, x0)K2(x, x0) product of independent processes

• Convolution: K(x, x0) = Z dz dz0 h(x, z)K(z, z0)h(x 0 , z0) blurring of process with kernel h

(3). Relationship to neural networks

Neural net with one hidden layer of NH units:
F(x) = b + X NH j=1 vjh(x; uj)
h — bounded hidden layer transfer function(e.g. h(x; u) = erf(u >x))

• If v’s and b zero mean independent, and weights uj iid, then CLT implies NN → GP as NH → ∞ [Neal, 1996]
• NN covariance function depends on transfer function h, but is in general non-stationary.

4.PCA

Principal Component Analysis (PCA) is a dimension-reduction tool that can be used to reduce a large set of variables to a small set that still contains most of the information in the large set.

Algorithm:

Suppose that we have a random vector **X**, then get population variance-covariance matrix And consider the linear combinations. Y_i is a function of our random data, and so is also random. Therefore, it has a population variance. Y_i and Y_j have population covariance. Collect the coefficients e_{ij} into the vector. When it comes to PCA, i^{th} *Principal Component* (*PCAi*): Y_i We select $\boldsymbol{e}_{i1}$, $\boldsymbol{e}_{i2}$, ..., $\boldsymbol{e}_{ip}$ to maximize. We subject to the constraint that the sums of squared coefficients add up to one...along with the additional constraint that this new component is uncorrelated with all the previously defined components.

$$\mathbf{X} = \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_p \end{pmatrix}$$

$$\operatorname{var}(\mathbf{X}) = \Sigma = \begin{pmatrix} \sigma_1^2 & \sigma_{12} & \cdots & \sigma_{1p} \\ \sigma_{21} & \sigma_2^2 & \cdots & \sigma_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{p1} & \sigma_{p2} & \cdots & \sigma_p^2 \end{pmatrix}$$

$$\begin{aligned} Y_1 &= e_{11}X_1 + e_{12}X_2 + \cdots + e_{1p}X_p \\ Y_2 &= e_{21}X_1 + e_{22}X_2 + \cdots + e_{2p}X_p \\ &\vdots \\ Y_p &= e_{p1}X_1 + e_{p2}X_2 + \cdots + e_{pp}X_p \end{aligned}$$

$$\operatorname{var}(Y_i) = \sum_{k=1}^{p}\sum_{l=1}^{p} e_{ik}e_{il}\sigma_{kl} = \mathbf{e}_i'\Sigma\mathbf{e}_i$$

$$\operatorname{var}(Y_1) = \sum_{k=1}^{p}\sum_{l=1}^{p} e_{1k}e_{1l}\sigma_{kl} = \mathbf{e}_1'\Sigma\mathbf{e}_1$$

$$\operatorname{cov}(Y_i, Y_j) = \sum_{k=1}^{p}\sum_{l=1}^{p} e_{ik}e_{jl}\sigma_{kl} = \mathbf{e}_i'\Sigma\mathbf{e}_j$$

$$\mathbf{e}_i = \begin{pmatrix} e_{i1} \\ e_{i2} \\ \vdots \\ e_{ip} \end{pmatrix}$$

$$\mathbf{e}_1'\mathbf{e}_1 = \sum_{j=1}^{p} e_{1j}^2 = 1$$

$$\operatorname{var}(Y_i) = \sum_{k=1}^{p}\sum_{l=1}^{p} e_{ik}e_{il}\sigma_{kl} = \mathbf{e}_i'\Sigma\mathbf{e}_i$$

$$\mathbf{e}_i'\mathbf{e}_i = \textstyle\sum_{j=1}^{p} e_{ij}^2 = 1$$

$$\begin{aligned} \operatorname{cov}(Y_1, Y_i) &= \textstyle\sum_{k=1}^{p}\sum_{l=1}^{p} e_{1k}e_{il}\sigma_{kl} = \mathbf{e}_1'\Sigma\mathbf{e}_i = 0, \\ \operatorname{cov}(Y_2, Y_i) &= \textstyle\sum_{k=1}^{p}\sum_{l=1}^{p} e_{2k}e_{il}\sigma_{kl} = \mathbf{e}_2'\Sigma\mathbf{e}_i = 0, \\ &\vdots \\ \operatorname{cov}(Y_{i-1}, Y_i) &= \textstyle\sum_{k=1}^{p}\sum_{l=1}^{p} e_{i-1,k}e_{il}\sigma_{kl} = \mathbf{e}_{i-1}'\Sigma\mathbf{e}_i = 0 \end{aligned}$$

5.k-means :

K-Means is one of the most popular "clustering" algorithms. K-means stores kk centroids that it uses to define clusters. A point is considered to be in a particular cluster if it is closer to that cluster's centroid than any other centroid.

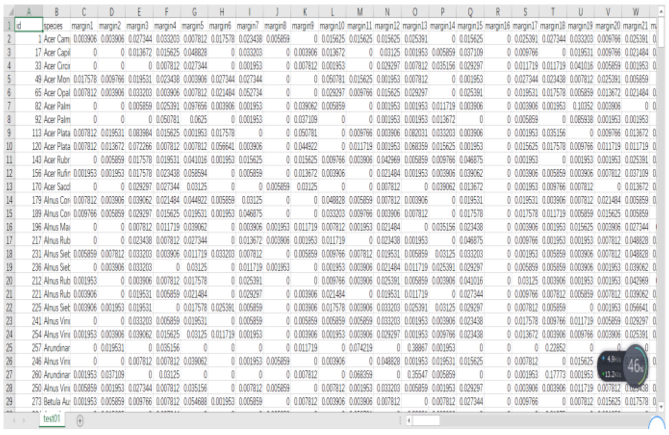
The Algorithm

In the clustering problem, we are given a training set x(1),...,x(m)x(1),...,x(m), and want to group the data into a few cohesive "clusters." Here, we are given feature vectors for each data point x(i)∈Rnx(i)∈Rn as usual; but no labels y(i)y(i) (making this an unsupervised learning problem). Our goal is to predict k centroids **and** a label c(i)c(i) for each data point. The k-means clustering algorithm is as follows:

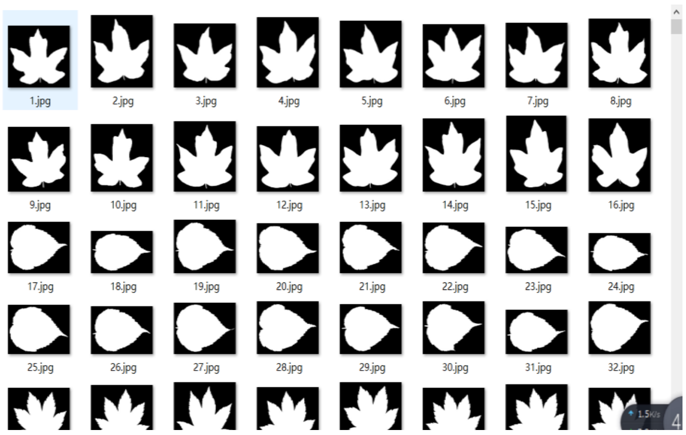
```
1. Initialize cluster centroids $\mu_1, \mu_2, \ldots, \mu_k \in \mathbb{R}^n$ randomly.
2. Repeat until convergence: {
       For every $i$, set
           $c^{(i)} := \arg\min_j \|x^{(i)} - \mu_j\|^2.$
       For each $j$, set
           $\mu_j := \frac{\sum_{i=1}^m 1\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^m 1\{c^{(i)} = j\}}.$
   }
```

3.Dataset:

Dataset:https://archive.ics.uci.edu/ml/datasets/One-hundred+plant+species+leaves+data+set
The dataset we use comes from UCI machine learning repository.
There are one-hundred plant species leaves as data. And Sixteen samples of leaf each of one-hundred plant species. For each sample, a shape descriptor, fine scale margin and texture histogram are given. And the number of each attribute is 64. There are screenshot below for both csv and image folder.



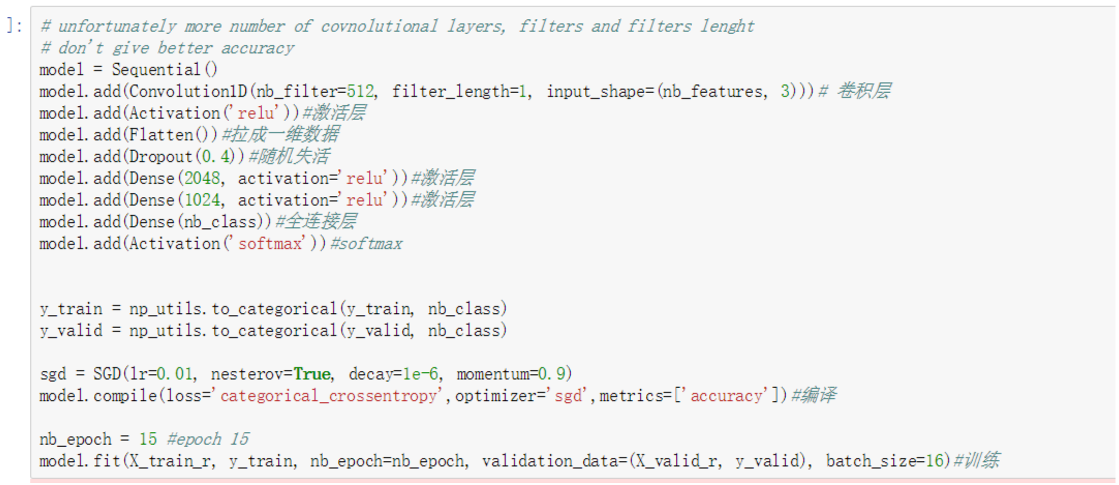
id	species	margin1	margin2	margin3	margin4	margin5	margin6	margin7	margin8	margin9	margin10	margin11	margin12	margin13	margin14	margin15	margin16	margin17	margin18	margin19	margin20	margin21
1	Acer Cam	0.003906	0.003906	0.027344	0.033203	0.007812	0.017578	0.023438	0.005859	0	0.015625	0.015625	0.015625	0.025391	0	0.015625	0	0.025391	0.027344	0.033203	0.009766	0.025391
17	Acer Capi	0	0	0.013672	0.015625	0.048828	0	0.033203	0	0.003906	0.013672	0	0.03125	0.001953	0.005859	0.037109	0	0.009766	0	0.019531	0.009766	0.021484
33	Acer Circi	0	0	0	0.007812	0.027344	0	0.001953	0	0.007812	0.001953	0	0.029297	0.007812	0.035156	0.029297	0	0.011719	0.011719	0.041016	0.005859	0.001953
49	Acer Mon	0.017578	0.009766	0.019531	0.023438	0.003906	0.027344	0.027344	0	0	0.050781	0.015625	0.001953	0.007812	0	0.001953	0	0.027344	0.023438	0.007812	0.025391	0.005859
65	Acer Opal	0.007812	0.003906	0.033203	0.003906	0.007812	0.021484	0.052734	0	0	0.029297	0.009766	0.015625	0.029297	0	0.025391	0	0.019531	0.017578	0.005859	0.013672	0.021484
82	Acer Palm	0	0	0.005859	0.025391	0.097656	0.003906	0.001953	0	0.039062	0.005859	0	0.001953	0.001953	0.011719	0.003906	0	0.003906	0.001953	0.10352	0.003906	0
92	Acer Palm	0	0	0	0.050781	0.0625	0	0.001953	0	0.037109	0	0	0.001953	0.001953	0.013672	0	0	0.005859	0	0.085938	0.001953	0.001953
113	Acer Plata	0.007812	0.019531	0.083984	0.015625	0.001953	0.017578	0	0	0.050781	0	0.009766	0.003906	0.082031	0.033203	0.003906	0	0.001953	0.035156	0	0.009766	0.013672
120	Acer Plata	0.007812	0.013672	0.072266	0.007812	0.007812	0.056641	0.003906	0	0.044922	0	0.011719	0.001953	0.068359	0.015625	0.001953	0	0.015625	0.017578	0.009766	0.011719	0.011719
143	Acer Rubr	0	0.005859	0.017578	0.019531	0.041016	0.001953	0.015625	0	0.015625	0.009766	0.003906	0.042969	0.005859	0.009766	0.046875	0	0.001953	0	0.001953	0.001953	0.025391
156	Acer Rufin	0.001953	0.001953	0.017578	0.023438	0.058594	0	0.005859	0	0.013672	0.003906	0	0.021484	0.001953	0.003906	0.039062	0	0.003906	0.005859	0.003906	0.007812	0.037109
170	Acer Sacd	0	0	0.029297	0.027344	0.03125	0	0	0.005859	0.03125	0	0	0.007812	0	0.039062	0.013672	0	0.001953	0.009766	0.007812	0	0.013672
179	Alnus Cor	0.007812	0.003906	0.039062	0.021484	0.044922	0.005859	0.03125	0	0	0.048828	0.005859	0.007812	0.003906	0	0.019531	0	0.019531	0.003906	0.007812	0.021484	0.005859
189	Alnus Cor	0.009766	0.005859	0.029297	0.015625	0.019531	0.001953	0.046875	0	0	0.033203	0.009766	0.003906	0.007812	0	0.017578	0	0.017578	0.011719	0.005859	0.015625	0.005859
196	Alnus Ma	0	0	0.007812	0.011719	0.039062	0	0.003906	0.001953	0.011719	0.007812	0.001953	0.021484	0	0.035156	0.023438	0	0.003906	0.001953	0.015625	0.003906	0.027344
217	Alnus Rub	0	0	0.023438	0.007812	0.027344	0	0.013672	0.003906	0.001953	0.011719	0	0.023438	0.001953	0	0.046875	0	0.009766	0.001953	0.001953	0.007812	0.048828
231	Alnus Siet	0.005859	0.007812	0.033203	0.003906	0.011719	0.033203	0.007812	0	0.005859	0.009766	0.007812	0.019531	0.005859	0.03125	0.033203	0	0.001953	0.005859	0.003906	0.007812	0.048828
236	Alnus Siet	0	0.003906	0.033203	0	0.03125	0	0.011719	0.001953	0	0.001953	0.003906	0.021484	0.011719	0.025391	0.029297	0	0.005859	0.005859	0.003906	0.001953	0.039062
212	Alnus Rub	0.001953	0	0.003906	0.007812	0.017578	0	0.025391	0	0	0.009766	0.003906	0.025391	0.005859	0.003906	0.041016	0	0.03125	0.003906	0.001953	0.001953	0.042969
221	Alnus Rub	0.003906	0	0.019531	0.005859	0.021484	0	0.029297	0	0.003906	0.021484	0	0.019531	0.011719	0	0.027344	0	0.009766	0.007812	0.005859	0.007812	0.039062
225	Alnus Siet	0.003906	0.001953	0.019531	0	0.017578	0.025391	0.005859	0	0.003906	0.017578	0.003906	0.033203	0.025391	0.03125	0.029297	0	0.007812	0.005859	0	0.001953	0.056641
241	Alnus Viri	0	0	0.033203	0.005859	0.019531	0	0.005859	0	0.005859	0.005859	0.005859	0.033203	0.001953	0.003906	0.023438	0	0.017578	0.009766	0.011719	0.005859	0.029297
254	Alnus Viri	0.001953	0.003906	0.039062	0.015625	0.03125	0.011719	0.001953	0	0.003906	0.001953	0.003906	0.029297	0.001953	0.009766	0.023438	0	0.013672	0.003906	0.009766	0.003906	0.025391
257	Arundinar	0	0.019531	0	0.035156	0	0	0	0	0.011719	0	0.074219	0	0.38867	0.001953	0	0	0	0.22852	0	0	0
246	Alnus Viri	0	0	0.007812	0.007812	0.039062	0	0.001953	0.005859	0	0.003906	0	0.048828	0.001953	0.019531	0.015625	0	0.007812	0	0.015625	[illegible]	[illegible]
260	Arundinar	0.001953	0.037109	0	0.03125	0	0	0	0	0.007812	0	0.068359	0	0.35547	0.005859	0	0	0.001953	0.17773	0.001953	[illegible]	[illegible]
250	Alnus Viri	0.005859	0.001953	0.027344	0.007812	0.035156	0	0.007812	0.005859	0	0.007812	0.001953	0.033203	0.005859	0.001953	0.029297	0	0.003906	0.003906	0.011719	0.007812	0.023438
273	Betula Au	0.001953	0.005859	0.009766	0.007812	0.054688	0.001953	0.005859	0	0.007812	0.003906	0.003906	0.007812	0	0.007812	0.027344	0	0.009766	0	0.007812	0.015625	0.017578



4.Methods:

1. CNN model to train the CSV files.

We have 3 kinds of attributes named shape margin and texture, and each attribute has 64 different parameters. So we think that we can use this concrete and rich train to do machine train at once, then we choose to use CNN as the training method.



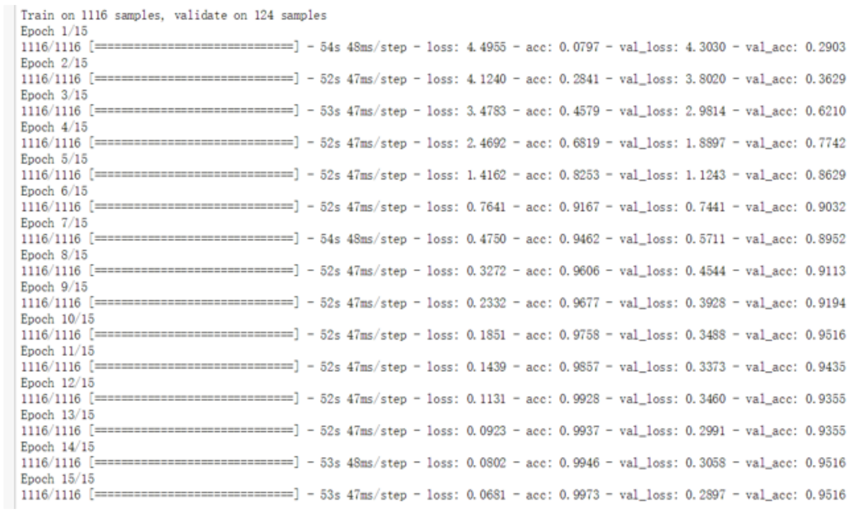
```
]: # unfortunately more number of covnolutional layers, filters and filters lenght
   # don't give better accuracy
   model = Sequential()
   model.add(Convolution1D(nb_filter=512, filter_length=1, input_shape=(nb_features, 3)))# 卷积层
   model.add(Activation('relu'))#激活层
   model.add(Flatten())#拉成一维数据
   model.add(Dropout(0.4))#随机失活
   model.add(Dense(2048, activation='relu'))#激活层
   model.add(Dense(1024, activation='relu'))#激活层
   model.add(Dense(nb_class))#全连接层
   model.add(Activation('softmax'))#softmax


   y_train = np_utils.to_categorical(y_train, nb_class)
   y_valid = np_utils.to_categorical(y_valid, nb_class)

   sgd = SGD(lr=0.01, nesterov=True, decay=1e-6, momentum=0.9)
   model.compile(loss='categorical_crossentropy',optimizer='sgd',metrics=['accuracy'])#编译

   nb_epoch = 15 #epoch 15
   model.fit(X_train_r, y_train, nb_epoch=nb_epoch, validation_data=(X_valid_r, y_valid), batch_size=16)#训练
```

As we can see from the picture, we use only one convolution layer, and the activation method is ‘relu’ – the most common one, the training method is ‘softmax’,because we know that softmax, the loss method is categorical_crossentropty; the Opimizer is ‘rgd’ and the epoch size is 15, the batch size is 16.

We need to explain the mehod named softmax. It is a generalization of the logistic function that "squashes" a *K*-dimensional vector of arbitrary real values to a *K*-dimensional vector of real values in the range (0, 1) that add up to 1.

$$\sigma : \mathbb{R}^K \to (0,1)^K$$

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

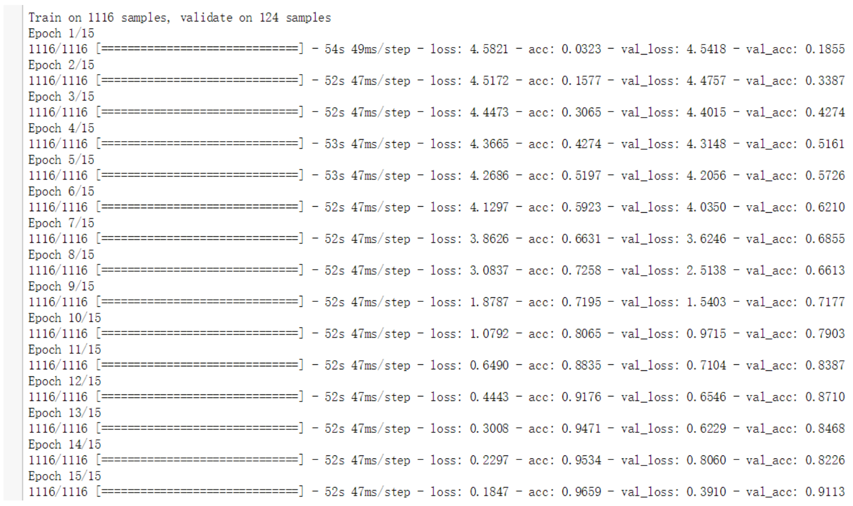


```
Train on 1116 samples, validate on 124 samples
Epoch 1/15
1116/1116 [============================] - 54s 48ms/step - loss: 4.4955 - acc: 0.0797 - val_loss: 4.3030 - val_acc: 0.2903
Epoch 2/15
1116/1116 [============================] - 52s 47ms/step - loss: 4.1240 - acc: 0.2841 - val_loss: 3.8020 - val_acc: 0.3629
Epoch 3/15
1116/1116 [============================] - 53s 47ms/step - loss: 3.4783 - acc: 0.4579 - val_loss: 2.9814 - val_acc: 0.6210
Epoch 4/15
1116/1116 [============================] - 52s 47ms/step - loss: 2.4692 - acc: 0.6819 - val_loss: 1.8897 - val_acc: 0.7742
Epoch 5/15
1116/1116 [============================] - 52s 47ms/step - loss: 1.4162 - acc: 0.8253 - val_loss: 1.1243 - val_acc: 0.8629
Epoch 6/15
1116/1116 [============================] - 52s 47ms/step - loss: 0.7641 - acc: 0.9167 - val_loss: 0.7441 - val_acc: 0.9032
Epoch 7/15
1116/1116 [============================] - 54s 48ms/step - loss: 0.4750 - acc: 0.9462 - val_loss: 0.5711 - val_acc: 0.8952
Epoch 8/15
1116/1116 [============================] - 52s 47ms/step - loss: 0.3272 - acc: 0.9606 - val_loss: 0.4544 - val_acc: 0.9113
Epoch 9/15
1116/1116 [============================] - 52s 47ms/step - loss: 0.2332 - acc: 0.9677 - val_loss: 0.3928 - val_acc: 0.9194
Epoch 10/15
1116/1116 [============================] - 52s 47ms/step - loss: 0.1851 - acc: 0.9758 - val_loss: 0.3488 - val_acc: 0.9516
Epoch 11/15
1116/1116 [============================] - 52s 47ms/step - loss: 0.1439 - acc: 0.9857 - val_loss: 0.3373 - val_acc: 0.9435
Epoch 12/15
1116/1116 [============================] - 52s 47ms/step - loss: 0.1131 - acc: 0.9928 - val_loss: 0.3460 - val_acc: 0.9355
Epoch 13/15
1116/1116 [============================] - 52s 47ms/step - loss: 0.0923 - acc: 0.9937 - val_loss: 0.2991 - val_acc: 0.9355
Epoch 14/15
1116/1116 [============================] - 53s 48ms/step - loss: 0.0802 - acc: 0.9946 - val_loss: 0.3058 - val_acc: 0.9516
Epoch 15/15
1116/1116 [============================] - 53s 47ms/step - loss: 0.0681 - acc: 0.9973 - val_loss: 0.2897 - val_acc: 0.9516
```

We can see the result , when the epoch comes to 8, the accuracy comes to the highest-95.16%

Then we change the training method from softmax to sigmoid.

We want to compare the two methods and control other parameters as the same.



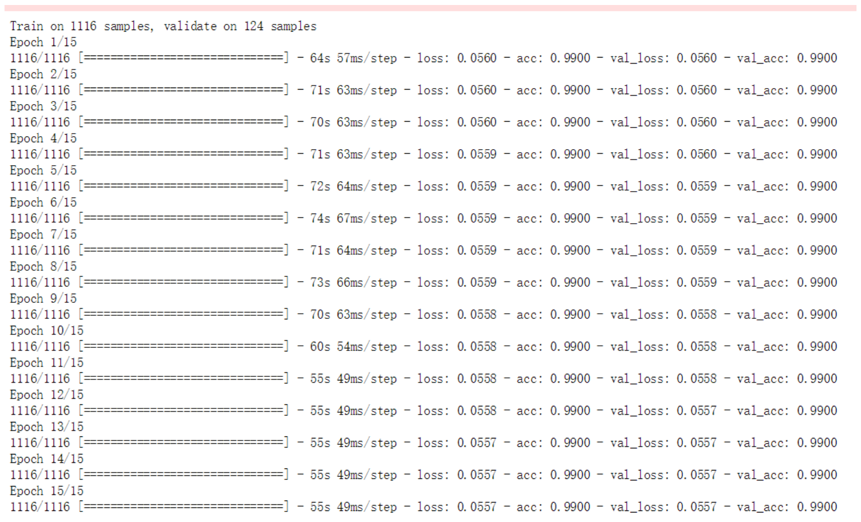
```
Train on 1116 samples, validate on 124 samples
Epoch 1/15
1116/1116 [============================] - 54s 49ms/step - loss: 4.5821 - acc: 0.0323 - val_loss: 4.5418 - val_acc: 0.1855
Epoch 2/15
1116/1116 [============================] - 52s 47ms/step - loss: 4.5172 - acc: 0.1577 - val_loss: 4.4757 - val_acc: 0.3387
Epoch 3/15
1116/1116 [============================] - 52s 47ms/step - loss: 4.4473 - acc: 0.3065 - val_loss: 4.4015 - val_acc: 0.4274
Epoch 4/15
1116/1116 [============================] - 53s 47ms/step - loss: 4.3665 - acc: 0.4274 - val_loss: 4.3148 - val_acc: 0.5161
Epoch 5/15
1116/1116 [============================] - 53s 47ms/step - loss: 4.2686 - acc: 0.5197 - val_loss: 4.2056 - val_acc: 0.5726
Epoch 6/15
1116/1116 [============================] - 52s 47ms/step - loss: 4.1297 - acc: 0.5923 - val_loss: 4.0350 - val_acc: 0.6210
Epoch 7/15
1116/1116 [============================] - 52s 47ms/step - loss: 3.8626 - acc: 0.6631 - val_loss: 3.6246 - val_acc: 0.6855
Epoch 8/15
1116/1116 [============================] - 52s 47ms/step - loss: 3.0837 - acc: 0.7258 - val_loss: 2.5138 - val_acc: 0.6613
Epoch 9/15
1116/1116 [============================] - 52s 47ms/step - loss: 1.8787 - acc: 0.7195 - val_loss: 1.5403 - val_acc: 0.7177
Epoch 10/15
1116/1116 [============================] - 52s 47ms/step - loss: 1.0792 - acc: 0.8065 - val_loss: 0.9715 - val_acc: 0.7903
Epoch 11/15
1116/1116 [============================] - 52s 47ms/step - loss: 0.6490 - acc: 0.8835 - val_loss: 0.7104 - val_acc: 0.8387
Epoch 12/15
1116/1116 [============================] - 52s 47ms/step - loss: 0.4443 - acc: 0.9176 - val_loss: 0.6546 - val_acc: 0.8710
Epoch 13/15
1116/1116 [============================] - 52s 47ms/step - loss: 0.3008 - acc: 0.9471 - val_loss: 0.6229 - val_acc: 0.8468
Epoch 14/15
1116/1116 [============================] - 52s 47ms/step - loss: 0.2297 - acc: 0.9534 - val_loss: 0.8060 - val_acc: 0.8226
Epoch 15/15
1116/1116 [============================] - 52s 47ms/step - loss: 0.1847 - acc: 0.9659 - val_loss: 0.3910 - val_acc: 0.9113
```

About the sigmoid method, we know that it is a bounded, differentiable, real function that is defined for all real input values and has a non-negative derivative at each point.

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}.$$

And our dataset have a lot of parameters so we think that softmax may be the most suitable one. The result also shows that it's true. The accuracy using sigmoid is about 91%, which is lower than 'softmax'.

Then we change the loss method to binary_crossentropty.



```
Train on 1116 samples, validate on 124 samples
Epoch 1/15
1116/1116 [==============================] - 64s 57ms/step - loss: 0.0560 - acc: 0.9900 - val_loss: 0.0560 - val_acc: 0.9900
Epoch 2/15
1116/1116 [==============================] - 71s 63ms/step - loss: 0.0560 - acc: 0.9900 - val_loss: 0.0560 - val_acc: 0.9900
Epoch 3/15
1116/1116 [==============================] - 70s 63ms/step - loss: 0.0560 - acc: 0.9900 - val_loss: 0.0560 - val_acc: 0.9900
Epoch 4/15
1116/1116 [==============================] - 71s 63ms/step - loss: 0.0559 - acc: 0.9900 - val_loss: 0.0560 - val_acc: 0.9900
Epoch 5/15
1116/1116 [==============================] - 72s 64ms/step - loss: 0.0559 - acc: 0.9900 - val_loss: 0.0559 - val_acc: 0.9900
Epoch 6/15
1116/1116 [==============================] - 74s 67ms/step - loss: 0.0559 - acc: 0.9900 - val_loss: 0.0559 - val_acc: 0.9900
Epoch 7/15
1116/1116 [==============================] - 71s 64ms/step - loss: 0.0559 - acc: 0.9900 - val_loss: 0.0559 - val_acc: 0.9900
Epoch 8/15
1116/1116 [==============================] - 73s 66ms/step - loss: 0.0559 - acc: 0.9900 - val_loss: 0.0559 - val_acc: 0.9900
Epoch 9/15
1116/1116 [==============================] - 70s 63ms/step - loss: 0.0558 - acc: 0.9900 - val_loss: 0.0558 - val_acc: 0.9900
Epoch 10/15
1116/1116 [==============================] - 60s 54ms/step - loss: 0.0558 - acc: 0.9900 - val_loss: 0.0558 - val_acc: 0.9900
Epoch 11/15
1116/1116 [==============================] - 55s 49ms/step - loss: 0.0558 - acc: 0.9900 - val_loss: 0.0558 - val_acc: 0.9900
Epoch 12/15
1116/1116 [==============================] - 55s 49ms/step - loss: 0.0558 - acc: 0.9900 - val_loss: 0.0557 - val_acc: 0.9900
Epoch 13/15
1116/1116 [==============================] - 55s 49ms/step - loss: 0.0557 - acc: 0.9900 - val_loss: 0.0557 - val_acc: 0.9900
Epoch 14/15
1116/1116 [==============================] - 55s 49ms/step - loss: 0.0557 - acc: 0.9900 - val_loss: 0.0557 - val_acc: 0.9900
Epoch 15/15
1116/1116 [==============================] - 55s 49ms/step - loss: 0.0557 - acc: 0.9900 - val_loss: 0.0557 - val_acc: 0.9900
```

The accuracy is 99%, all the same.

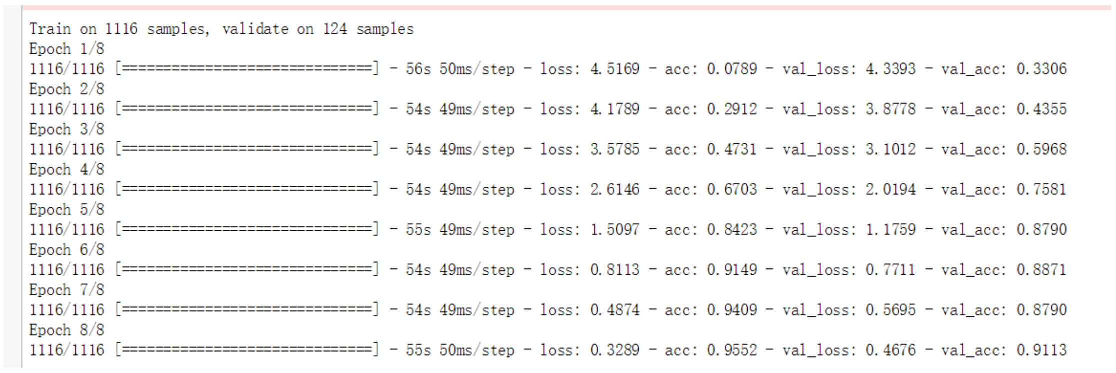
But we need to understand the difference between the two-loss method.

Binomial cross-entropy loss is a special case of multinomial cross-entropy loss for m=2m=2.

$$\mathcal{L}(\theta) = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(p_i) + (1-y_i)\log(1-p_i)] = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij}\log(p_{ij})$$

Where i indexes samples/observations and jj indexes classes, and yy is the sample label (binary for LSH, one-hot vector on the RHS) and $p_{ij} \in (0,1): \sum_j p_{ij} = 1 \forall i,j$ is the prediction for a sample.

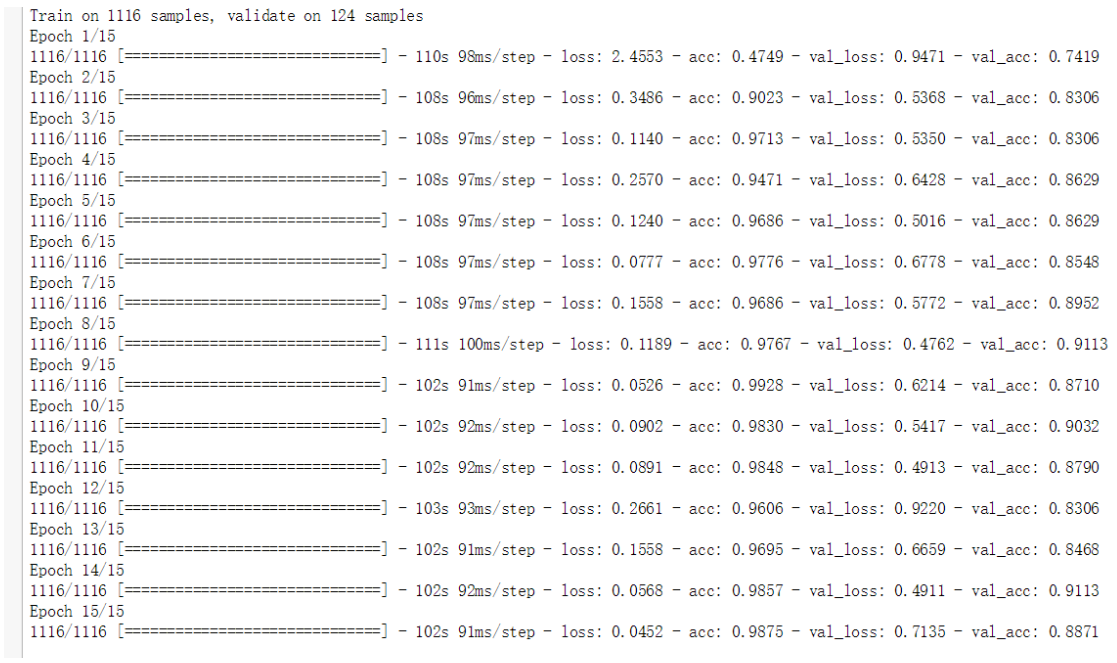
We can know that the binomial cross-entropy is not suitable for our model so we delete it.



```
Train on 1116 samples, validate on 124 samples
Epoch 1/8
1116/1116 [==============================] - 56s 50ms/step - loss: 4.5169 - acc: 0.0789 - val_loss: 4.3393 - val_acc: 0.3306
Epoch 2/8
1116/1116 [==============================] - 54s 49ms/step - loss: 4.1789 - acc: 0.2912 - val_loss: 3.8778 - val_acc: 0.4355
Epoch 3/8
1116/1116 [==============================] - 54s 49ms/step - loss: 3.5785 - acc: 0.4731 - val_loss: 3.1012 - val_acc: 0.5968
Epoch 4/8
1116/1116 [==============================] - 54s 49ms/step - loss: 2.6146 - acc: 0.6703 - val_loss: 2.0194 - val_acc: 0.7581
Epoch 5/8
1116/1116 [==============================] - 55s 49ms/step - loss: 1.5097 - acc: 0.8423 - val_loss: 1.1759 - val_acc: 0.8790
Epoch 6/8
1116/1116 [==============================] - 54s 49ms/step - loss: 0.8113 - acc: 0.9149 - val_loss: 0.7711 - val_acc: 0.8871
Epoch 7/8
1116/1116 [==============================] - 54s 49ms/step - loss: 0.4874 - acc: 0.9409 - val_loss: 0.5695 - val_acc: 0.8790
Epoch 8/8
1116/1116 [==============================] - 55s 50ms/step - loss: 0.3289 - acc: 0.9552 - val_loss: 0.4676 - val_acc: 0.9113
```

Then we can change the epoch number to 8 to see whether we can get the same accuracy using fewer epochs. The answer is no, because the accuracy is only 91%.

At last we change the optimizer to rgd to see the difference.



```
Train on 1116 samples, validate on 124 samples
Epoch 1/15
1116/1116 [==============================] - 110s 98ms/step - loss: 2.4553 - acc: 0.4749 - val_loss: 0.9471 - val_acc: 0.7419
Epoch 2/15
1116/1116 [==============================] - 108s 96ms/step - loss: 0.3486 - acc: 0.9023 - val_loss: 0.5368 - val_acc: 0.8306
Epoch 3/15
1116/1116 [==============================] - 108s 97ms/step - loss: 0.1140 - acc: 0.9713 - val_loss: 0.5350 - val_acc: 0.8306
Epoch 4/15
1116/1116 [==============================] - 108s 97ms/step - loss: 0.2570 - acc: 0.9471 - val_loss: 0.6428 - val_acc: 0.8629
Epoch 5/15
1116/1116 [==============================] - 108s 97ms/step - loss: 0.1240 - acc: 0.9686 - val_loss: 0.5016 - val_acc: 0.8629
Epoch 6/15
1116/1116 [==============================] - 108s 97ms/step - loss: 0.0777 - acc: 0.9776 - val_loss: 0.6778 - val_acc: 0.8548
Epoch 7/15
1116/1116 [==============================] - 108s 97ms/step - loss: 0.1558 - acc: 0.9686 - val_loss: 0.5772 - val_acc: 0.8952
Epoch 8/15
1116/1116 [==============================] - 111s 100ms/step - loss: 0.1189 - acc: 0.9767 - val_loss: 0.4762 - val_acc: 0.9113
Epoch 9/15
1116/1116 [==============================] - 102s 91ms/step - loss: 0.0526 - acc: 0.9928 - val_loss: 0.6214 - val_acc: 0.8710
Epoch 10/15
1116/1116 [==============================] - 102s 92ms/step - loss: 0.0902 - acc: 0.9830 - val_loss: 0.5417 - val_acc: 0.9032
Epoch 11/15
1116/1116 [==============================] - 102s 92ms/step - loss: 0.0891 - acc: 0.9848 - val_loss: 0.4913 - val_acc: 0.8790
Epoch 12/15
1116/1116 [==============================] - 103s 93ms/step - loss: 0.2661 - acc: 0.9606 - val_loss: 0.9220 - val_acc: 0.8306
Epoch 13/15
1116/1116 [==============================] - 102s 91ms/step - loss: 0.1558 - acc: 0.9695 - val_loss: 0.6659 - val_acc: 0.8468
Epoch 14/15
1116/1116 [==============================] - 102s 92ms/step - loss: 0.0568 - acc: 0.9857 - val_loss: 0.4911 - val_acc: 0.9113
Epoch 15/15
1116/1116 [==============================] - 102s 91ms/step - loss: 0.0452 - acc: 0.9875 - val_loss: 0.7135 - val_acc: 0.8871
```

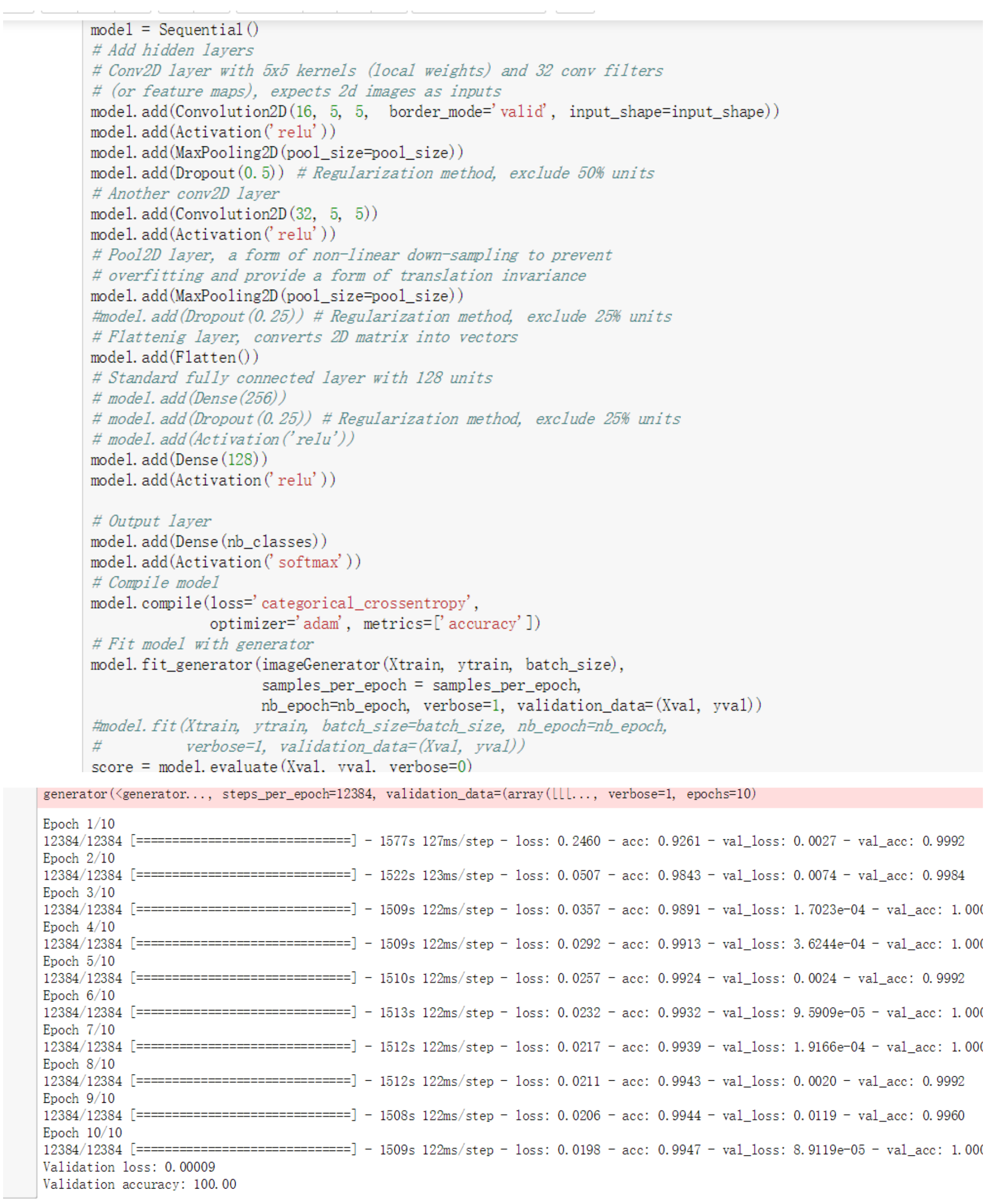
Adam stands for Adaptive Moment Estimation**.** Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients like AdaDelta , Adam *also keeps an exponentially decaying average of past gradients M(t), similar to momentum.*

The result shows that the sgd is the best one.

Summary:

when we use the epoch 15, softmax , sgd categorical_crossentropy we can get the best accuracy about 95 percent. Although when the loss mothod is Binary_crossentropy,we can get 99 percent accuracy,but it may not suitable for our dataset because we cannot make data binary.

2. **Use CNN to train CSV and image together:**
   We change the pixel for 64x64 ,96x96,40x40 and found tthat 64x64 is the best one. Then we expand the validation set so that we can have more val-data. And then set the parameter and method, use 4 convolution layers, use the ‘softmax’ ‘adam’, ‘categorical_crossentropy’, and we can get alomost 100 accuracy. So we end the CNN part.

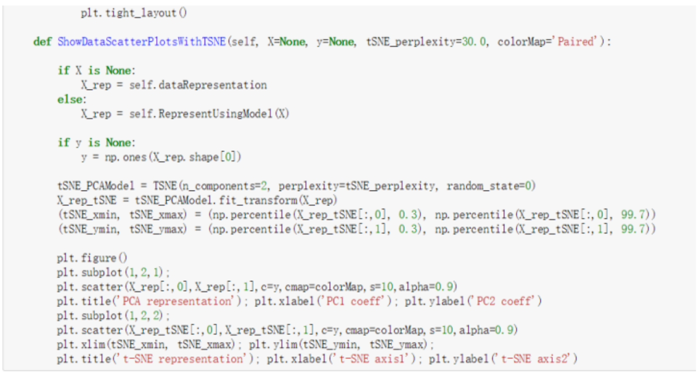


```
model = Sequential()
# Add hidden layers
# Conv2D layer with 5x5 kernels (local weights) and 32 conv filters
# (or feature maps), expects 2d images as inputs
model.add(Convolution2D(16, 5, 5,  border_mode='valid', input_shape=input_shape))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=pool_size))
model.add(Dropout(0.5)) # Regularization method, exclude 50% units
# Another conv2D layer
model.add(Convolution2D(32, 5, 5))
model.add(Activation('relu'))
# Pool2D layer, a form of non-linear down-sampling to prevent
# overfitting and provide a form of translation invariance
model.add(MaxPooling2D(pool_size=pool_size))
#model.add(Dropout(0.25)) # Regularization method, exclude 25% units
# Flattenig layer, converts 2D matrix into vectors
model.add(Flatten())
# Standard fully connected layer with 128 units
# model.add(Dense(256))
# model.add(Dropout(0.25)) # Regularization method, exclude 25% units
# model.add(Activation('relu'))
model.add(Dense(128))
model.add(Activation('relu'))

# Output layer
model.add(Dense(nb_classes))
model.add(Activation('softmax'))
# Compile model
model.compile(loss='categorical_crossentropy',
              optimizer='adam', metrics=['accuracy'])
# Fit model with generator
model.fit_generator(imageGenerator(Xtrain, ytrain, batch_size),
                    samples_per_epoch = samples_per_epoch,
                    nb_epoch=nb_epoch, verbose=1, validation_data=(Xval, yval))
#model.fit(Xtrain, ytrain, batch_size=batch_size, nb_epoch=nb_epoch,
#          verbose=1, validation_data=(Xval, yval))
score = model.evaluate(Xval, yval, verbose=0)
```

```
generator(<generator..., steps_per_epoch=12384, validation_data=(array([[[..., verbose=1, epochs=10)

Epoch 1/10
12384/12384 [==============================] - 1577s 127ms/step - loss: 0.2460 - acc: 0.9261 - val_loss: 0.0027 - val_acc: 0.9992
Epoch 2/10
12384/12384 [==============================] - 1522s 123ms/step - loss: 0.0507 - acc: 0.9843 - val_loss: 0.0074 - val_acc: 0.9984
Epoch 3/10
12384/12384 [==============================] - 1509s 122ms/step - loss: 0.0357 - acc: 0.9891 - val_loss: 1.7023e-04 - val_acc: 1.00
Epoch 4/10
12384/12384 [==============================] - 1509s 122ms/step - loss: 0.0292 - acc: 0.9913 - val_loss: 3.6244e-04 - val_acc: 1.00
Epoch 5/10
12384/12384 [==============================] - 1510s 122ms/step - loss: 0.0257 - acc: 0.9924 - val_loss: 0.0024 - val_acc: 0.9992
Epoch 6/10
12384/12384 [==============================] - 1513s 122ms/step - loss: 0.0232 - acc: 0.9932 - val_loss: 9.5909e-05 - val_acc: 1.00
Epoch 7/10
12384/12384 [==============================] - 1512s 122ms/step - loss: 0.0217 - acc: 0.9939 - val_loss: 1.9166e-04 - val_acc: 1.00
Epoch 8/10
12384/12384 [==============================] - 1512s 122ms/step - loss: 0.0211 - acc: 0.9943 - val_loss: 0.0020 - val_acc: 0.9992
Epoch 9/10
12384/12384 [==============================] - 1508s 122ms/step - loss: 0.0206 - acc: 0.9944 - val_loss: 0.0119 - val_acc: 0.9960
Epoch 10/10
12384/12384 [==============================] - 1509s 122ms/step - loss: 0.0198 - acc: 0.9947 - val_loss: 8.9119e-05 - val_acc: 1.00
Validation loss: 0.00009
Validation accuracy: 100.00
```

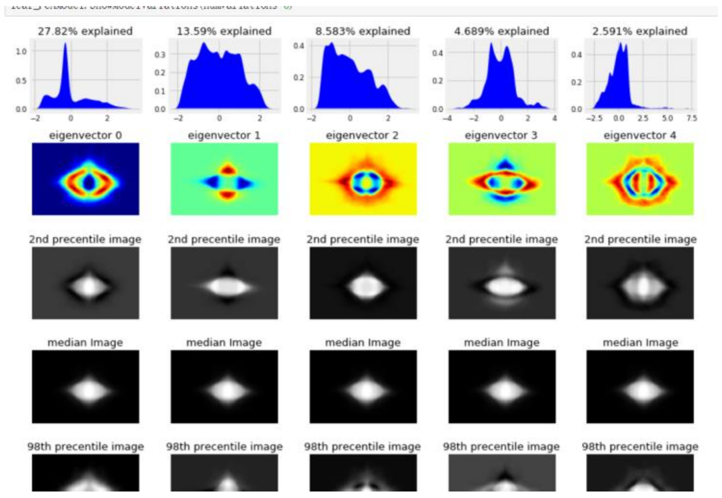
However, we found that when we train the csv and the image at the same time, the process is too slow, we need couple of hours to get the result, we want to discover whether it is okay to use some mehods such as PCA and k-means to reduce the dimension and also keep the accuracy of the data, and in that case we can do machine learing more easily than before. So we begin our following work-PCA and k-mean part.

3. use PCA to process data:



```
        plt.tight_layout()

    def ShowDataScatterPlotsWithTSNE(self, X=None, y=None, tSNE_perplexity=30.0, colorMap='Paired'):

        if X is None:
            X_rep = self.dataRepresentation
        else:
            X_rep = self.RepresentUsingModel(X)

        if y is None:
            y = np.ones(X_rep.shape[0])

        tSNE_PCAModel = TSNE(n_components=2, perplexity=tSNE_perplexity, random_state=0)
        X_rep_tSNE = tSNE_PCAModel.fit_transform(X_rep)
        (tSNE_xmin, tSNE_xmax) = (np.percentile(X_rep_tSNE[:,0], 0.3), np.percentile(X_rep_tSNE[:,0], 99.7))
        (tSNE_ymin, tSNE_ymax) = (np.percentile(X_rep_tSNE[:,1], 0.3), np.percentile(X_rep_tSNE[:,1], 99.7))

        plt.figure()
        plt.subplot(1,2,1);
        plt.scatter(X_rep[:,0],X_rep[:,1],c=y,cmap=colorMap,s=10,alpha=0.9)
        plt.title('PCA representation'); plt.xlabel('PC1 coeff'); plt.ylabel('PC2 coeff')
        plt.subplot(1,2,2);
        plt.scatter(X_rep_tSNE[:,0],X_rep_tSNE[:,1],c=y,cmap=colorMap,s=10,alpha=0.9)
        plt.xlim(tSNE_xmin, tSNE_xmax); plt.ylim(tSNE_ymin, tSNE_ymax);
        plt.title('t-SNE representation'); plt.xlabel('t-SNE axis1'); plt.ylabel('t-SNE axis2')
```

We build a Gaussian Model to reduce the dimensionality, and the eigenvalue left are most different ones.

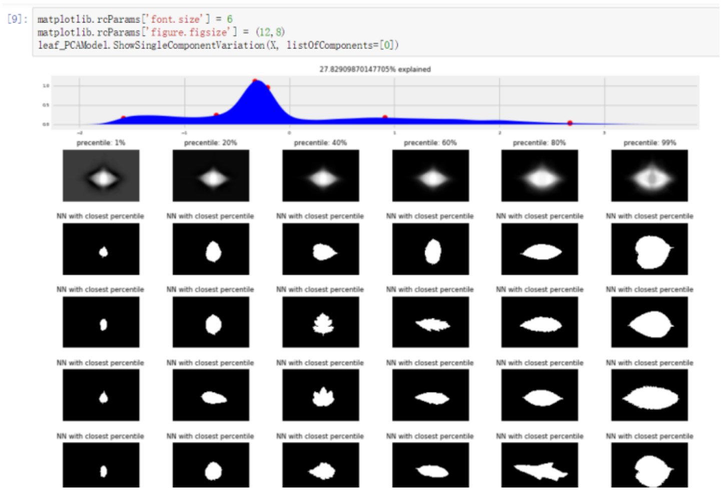


Then we can see model variations around mean image. The upper most row contains the data distributions of each eigenvector (i.e. the histogram along that "direction")

·The second row contains what we already saw in a previous plot, what we called the variance directions.

·The forth row contains the median image of leafs. notice that this row is identical for all eigenvectors

·The third row holds the 2nd percentile images of each eigenvector. it's easier to think of this as the median image minus the eigenvector image multiplied by some constant. i.e the image we see is the forth row image, minus the second row image, when the second row image is multiplied by a constant. The constant is chosen to show the varying degree of influence of this specific eigenvector on the "average" image, so we can visualize what type of variation this particular eigenvector tends to capture.



```
[9]: matplotlib.rcParams['font.size'] = 6
     matplotlib.rcParams['figure.figsize'] = (12,8)
     leaf_PCAModel.ShowSingleComponentVariation(X, listOfComponents=[0])
```

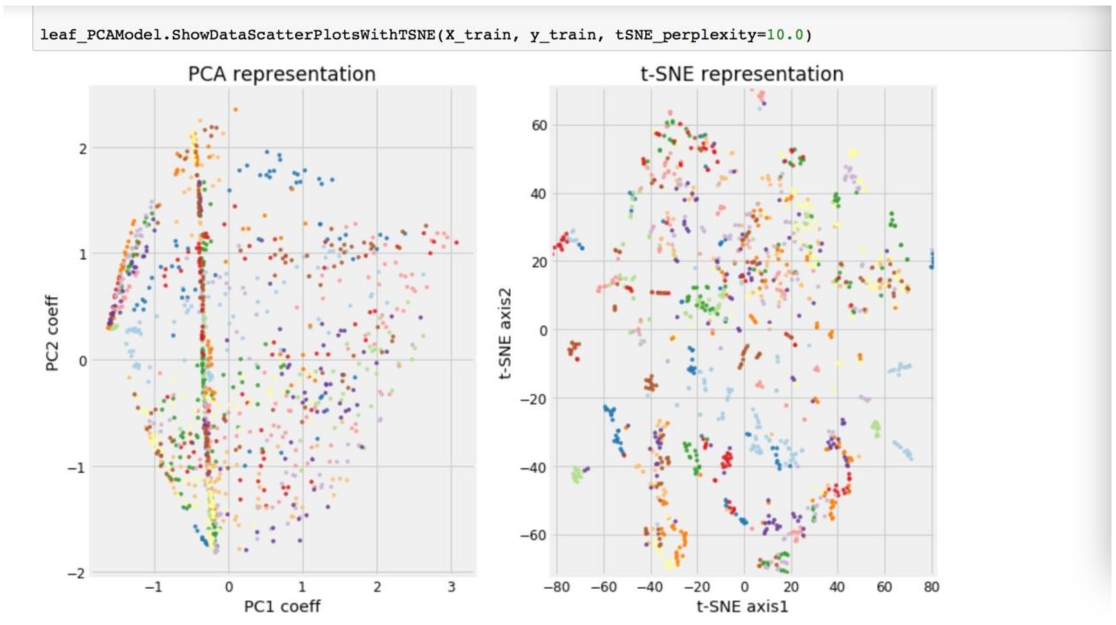
We take the engenvalue 1 as an example, we can find that from left to right, the leaves become larger and larger.

the first row shows the data distribution of the coefficients along this main variance direction. the red dots correspond to 1st, 20th, 40th, 60th, 80th and 99th percentiles of this distribution.

the second row is like the columns were in the previous plot. for example, we can see here in this particular case a gradual increase in leaf size from left to right.

the bottom 4 rows at each column hold real leaf images that have the first PCA coefficient be at the value of the corresponding percentile of that column. for example, the left most 4 bottom pictures are leafs with a PC1 coefficient to be approximately -1.6 and the right most 4 bottom pictures are leafs with a PC1 coefficient to be approximately 2.7.

In this particular case we can see that it's about leaf size since we see a gradual increase in leaf size from left to right.

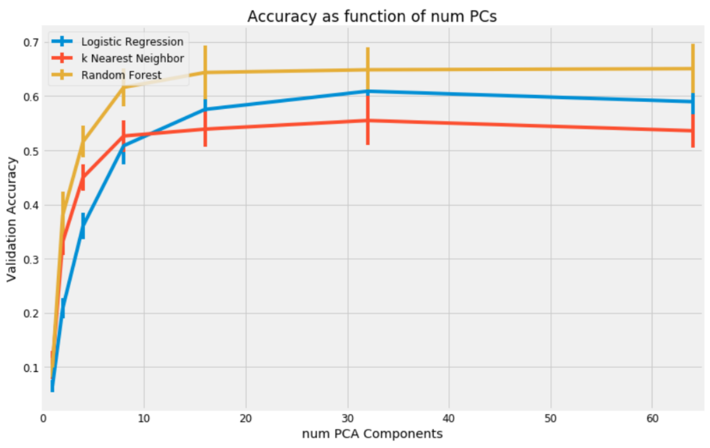


```
leaf_PCAModel.ShowDataScatterPlotsWithTSNE(X_train, y_train, tSNE_perplexity=10.0)
```

Then Plot the scatter plot of the first two principal component coeffients

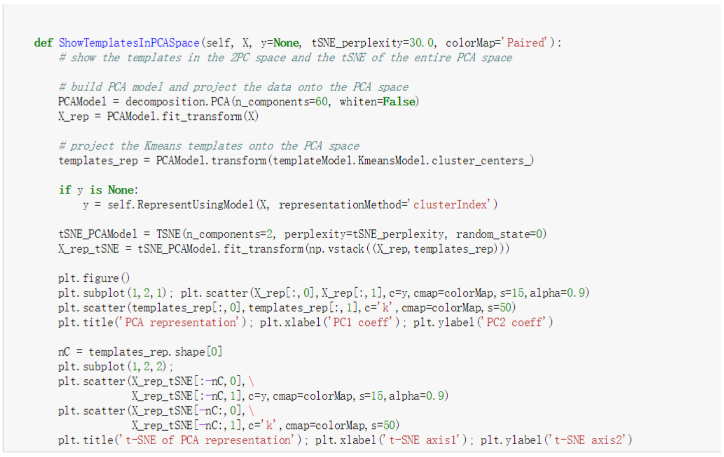
Plot a 2D approximation of the "high dimensional scatter plot" of the entire space using t-SNE

Distributed Stochastic Neighbor Embedding (t-SNE) is a technique for dimensionality reduction that is particularly well suited for the visualization of high-dimensional datasets. The technique can be implemented via Barnes-Hut approximations, allowing it to be applied on large real-world datasets.



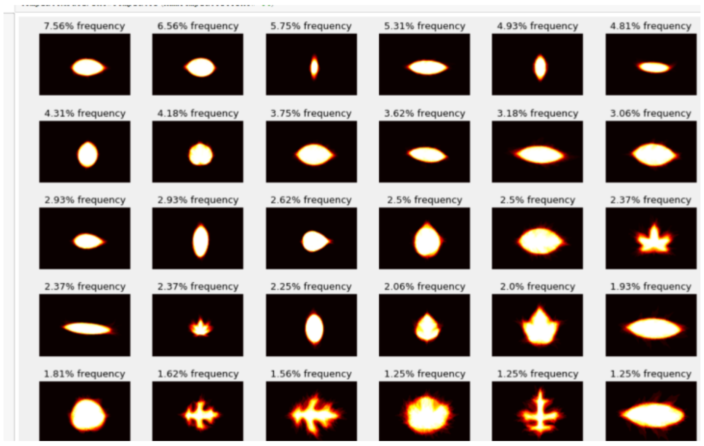
here is the result, we can use the accuracy is about PCA. About 60%.

4.K-mean:

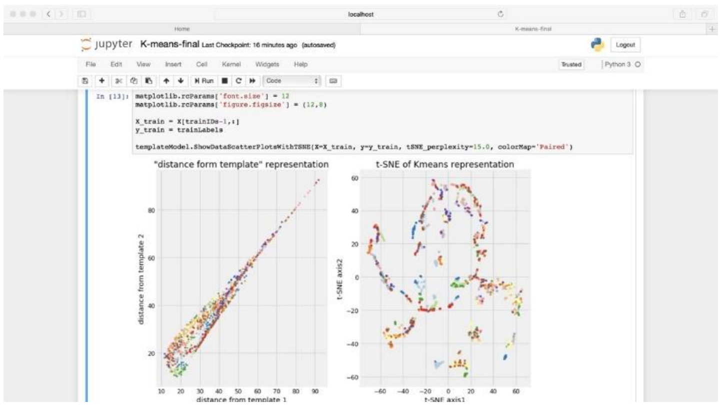


```
def ShowTemplatesInPCASpace(self, X, y=None, tSNE_perplexity=30.0, colorMap='Paired'):
    # show the templates in the 2PC space and the tSNE of the entire PCA space

    # build PCA model and project the data onto the PCA space
    PCAModel = decomposition.PCA(n_components=60, whiten=False)
    X_rep = PCAModel.fit_transform(X)

    # project the Kmeans templates onto the PCA space
    templates_rep = PCAModel.transform(templateModel.KmeansModel.cluster_centers_)

    if y is None:
        y = self.RepresentUsingModel(X, representationMethod='clusterIndex')

    tSNE_PCAModel = TSNE(n_components=2, perplexity=tSNE_perplexity, random_state=0)
    X_rep_tSNE = tSNE_PCAModel.fit_transform(np.vstack((X_rep, templates_rep)))

    plt.figure()
    plt.subplot(1,2,1); plt.scatter(X_rep[:,0],X_rep[:,1],c=y,cmap=colorMap,s=15,alpha=0.9)
    plt.scatter(templates_rep[:,0],templates_rep[:,1],c='k',cmap=colorMap,s=50)
    plt.title('PCA representation'); plt.xlabel('PC1 coeff'); plt.ylabel('PC2 coeff')

    nC = templates_rep.shape[0]
    plt.subplot(1,2,2);
    plt.scatter(X_rep_tSNE[:-nC,0], \
                X_rep_tSNE[:-nC,1],c=y,cmap=colorMap,s=15,alpha=0.9)
    plt.scatter(X_rep_tSNE[-nC:,0], \
                X_rep_tSNE[-nC:,1],c='k',cmap=colorMap,s=50)
    plt.title('t-SNE of PCA representation'); plt.xlabel('t-SNE axis1'); plt.ylabel('t-SNE axis2')
```

First we train the k-mean model



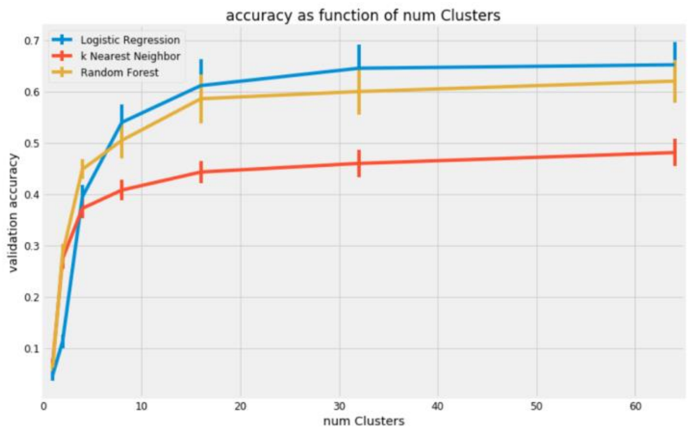
Then we train the number of k, when the k is 4,9,16 we can classify the actually shape of the leaves, when it comes to 36, we can see the picutre above, we can find that the shape is very clear.



```
In [13]: matplotlib.rcParams['font.size'] = 12
         matplotlib.rcParams['figure.figsize'] = (12,8)

         X_train = X[trainIDs-1,:]
         y_train = trainLabels

         templateModel.ShowDataScatterPlotsWithTSNE(X=X_train, y=y_train, tSNE_perplexity=15.0, colorMap='Paired')
```

localhost
Home
K-means-final
jupyter K-means-final Last Checkpoint: 16 minutes ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code
"distance form template" representation
distance from template 2
distance from template 1
t-SNE of Kmeans representation
t-SNE axis2
t-SNE axis1

Visualize "distance from cluster centers" feature space
Set 2 points and use a triangle method to decide the third points and doing the cluster work. T-SNE is also used for drawing a 2d picture with high-dimension parameters.



we build the k-means mode to show that when there is a center on the dataset. When the distance from the template is further, the difference between the eigenvalues are larger. For example, when there are two points respectively stay in the left and right side with the same distance, the eigenvalues are quietly different. And then we set 2 points which are next to each other, and then we can get the cluster and recognize the species.
The accuracy is about 60percent.
Summery about the last two parts:

1. PCA and K-Means image features are similarly useful in terms of classification.
2. The order between Logistic Regression and Random Forest has switched here compared to PCA case.

Even though these finding cannot be generalized because they heavily depend of this particular data distribution, we can speculate that there might be something complementary that Random Forest adds to the PCA feature representation, and that k-means features add to the classification abilities of the Logistic Regression classifier.

4. result:

Then we use CNN to train the csv file and the image folder. After comparing many kinds of parameters, we get the result that when we use the both files and use 4 Convolution layers. The activation='softmax' loss='categorical_crossentropy', optimizer='adam', we can get almost 100percent accuracy. Then because of the speed, we try to do some dimension reduction to make machine learning quicker. And then we use the k-means and PCA try it. And we found that there are still some features in the data, and the data can also be used to do machine learning. But the accuracy is onyly 60percent, so there are some disadvantages if we use the data.

6.Reference:

http://cs231n.github.io/convolutional-networks/
https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/
https://medium.com/@ageitgey/machine-learning-is-fun-part-3-deep-learning-and-convolutional-neural-networks-f40359318721
http://mlg.eng.cam.ac.uk/tutorials/06/es.pdf
https://en.wikipedia.org/wiki/Gaussian_process
https://www.tensorflow.org/
ftp://statgen.ncsu.edu/pub/thorne/molevoclass/AtchleyOct19.pdf
https://onlinecourses.science.psu.edu/stat505/node/51
http://stanford.edu/~cpiech/cs221/handouts/kmeans.html
https://stats.stackexchange.com/questions/260505/machine-learning-should-i-use-a-categorical-cross-entropy-or-binary-cross-entro
http://ruder.io/optimizing-gradient-descent/
http://lvdmaaten.github.io/tsne/
T-SNE: http://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html
Gaussianmodel:https://blog.dominodatalab.com/fitting-gaussian-process-models-python/
PCA example: https://towardsdatascience.com/pca-using-python-scikit-learn-e653f8989e60
PCA-KNN: https://www.kaggle.com/heibankeli/pca-knn
KNN:https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

k-means script: https://mubaris.com/2017/10/01/kmeans-clustering-in-python/

k-means exmaple: https://www.kaggle.com/naivecharles/k-means-neighborhood-clustering

CNN example:https://www.kaggle.com/tobikaggle/nn-through-keras-copied-mod-shuffle
CNN: https://github.com/keras-team/keras/blob/master/examples/imdb_cnn.py
CNN: https://www.kaggle.com/tonypoe/keras-cnn-example?scriptVersionId=589403
Tensorflow CNN：https://www.kaggle.com/jiexus/cnn-with-tensorflow/notebook
Batch generator: https://keras.io/preprocessing/image/
https://blog.csdn.net/sinat_26917383/article/details/74922230
GaussianModel:http://www.nehalemlabs.net/prototype/blog/2014/04/03/quick-introduction-to-gaussian-mixture-models-with-python/