Leaf recognition

Name: Weiyi Lan, Yiqun Xu, Yang Zong

Section (6)

Abstract:

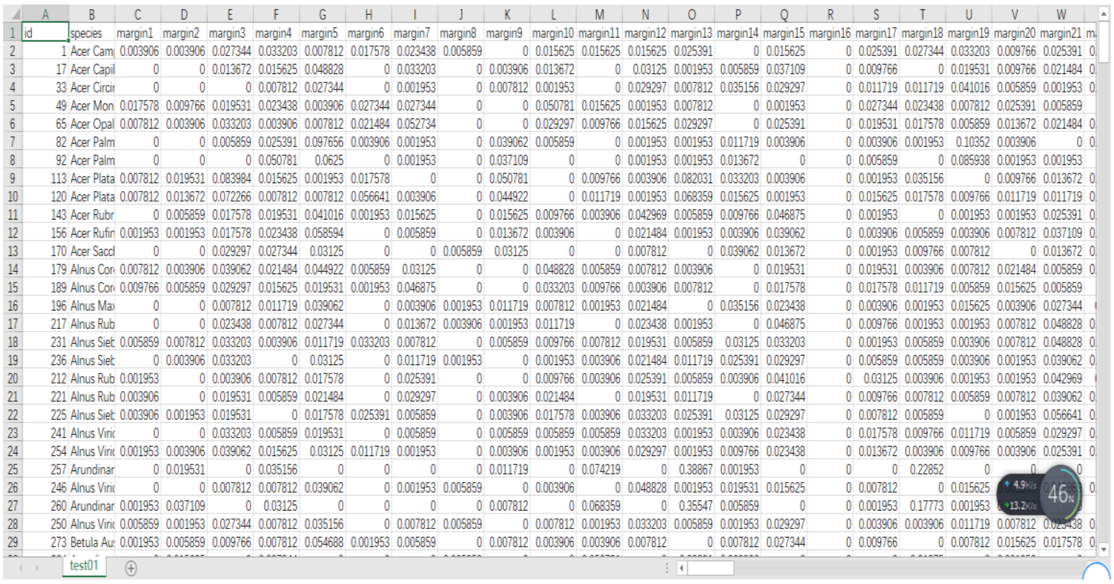
There are estimated to be nearly half a million species of plant all over the world. Here comes a problem that how can we classify these leaves, in other words, how can we get to know the name of leaves quickly and easily? Maybe we can use the knowledge in machine learning to do that. Machine learning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

Background:

Modern description methods are used for plant classification through leaf recognition. These methods usually include color transformation, feature detection and description, dimension reduction, and classification. However, these methods use an original image as the input image from which to extract the features to be recognized. In this condition, computational complexity will increase. To reduce computational time, in the proposed method the Region of Interest (ROI) is extracted before extracting features from the image. Quality of image also plays an important role in increasing leaf classification rate. A good quality image gives better classification rate than noisy images. Using CNN, we can deal with the black-and-white graphs and get the vectors from different graphs and match the data in CSV file. Also we can get the similarity and difference among 100 hundreds kinds of leaves so that we can know the leaves better.

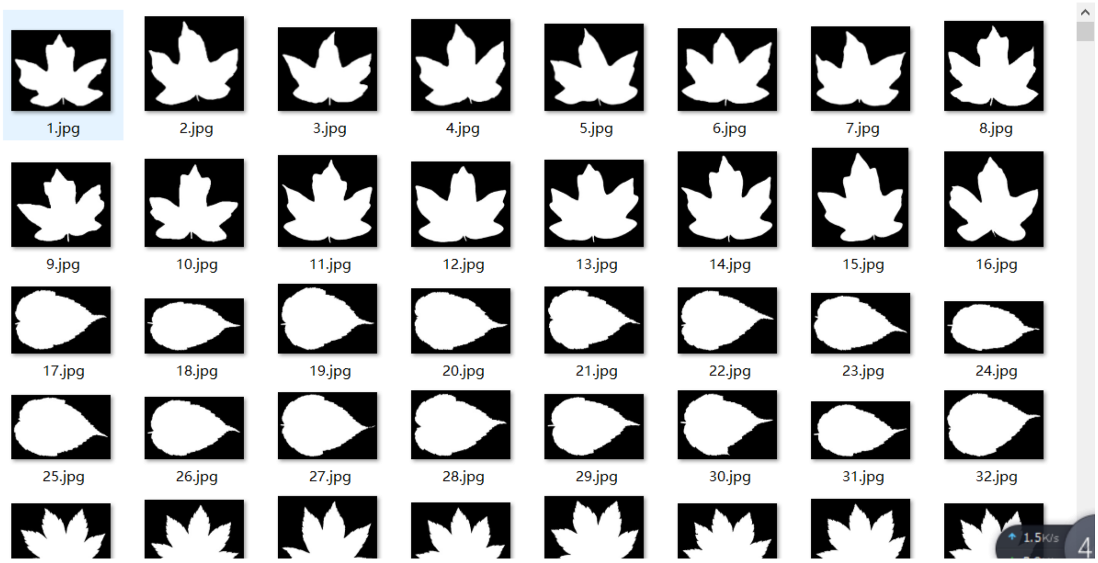
Dataset:

The dataset we use comes from UCI machine learning repository. And 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.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
1	id	species	margin1	margin2	margin3	margin4	margin5	margin6	margin7	margin8	margin9	margin10	margin11	margin12	margin13	margin14	margin15	margin16	margin17	margin18	margin19	margin20	margin21
2	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
3	17	Acer Capil	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
4	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
5	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
6	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
7	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
8	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
9	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
10	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
11	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
12	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
13	170	Acer Saccl	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
14	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
15	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
16	196	Alnus Max	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
17	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
18	231	Alnus Sieb	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
19	236	Alnus Sieb	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
20	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
21	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
22	225	Alnus Sieb	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
23	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
24	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
25	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
26	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]
27	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]
28	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
29	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

test01



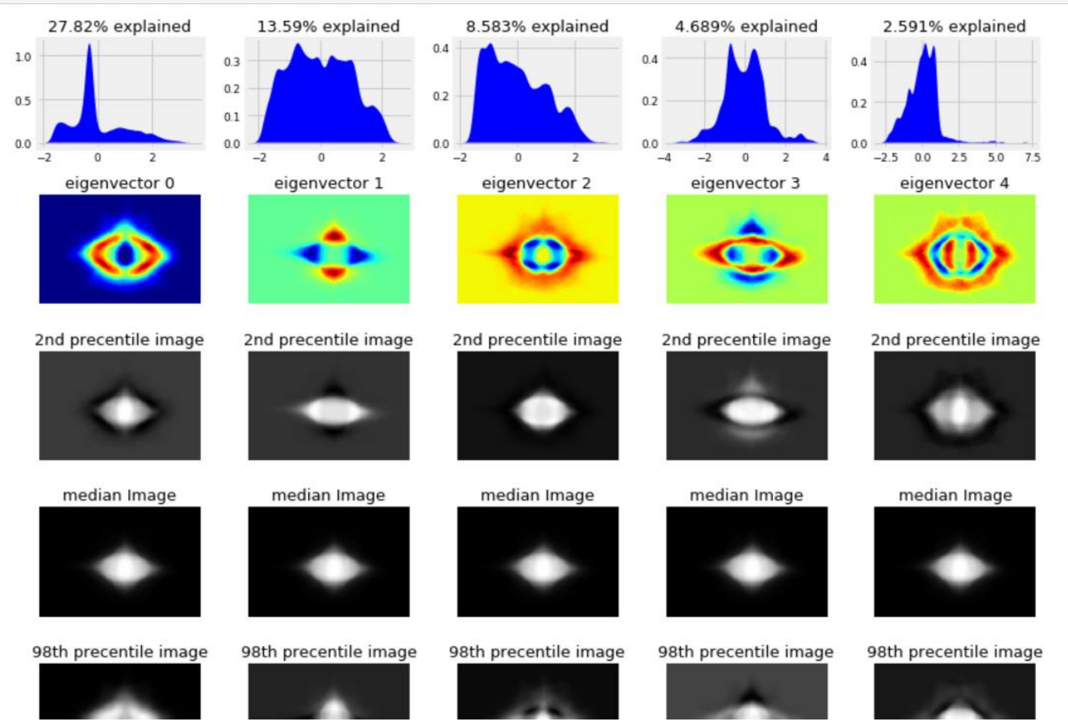
Code with document:

Part 1: 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')
```

mode a GaussianModel to reduce the dimensionality, and the

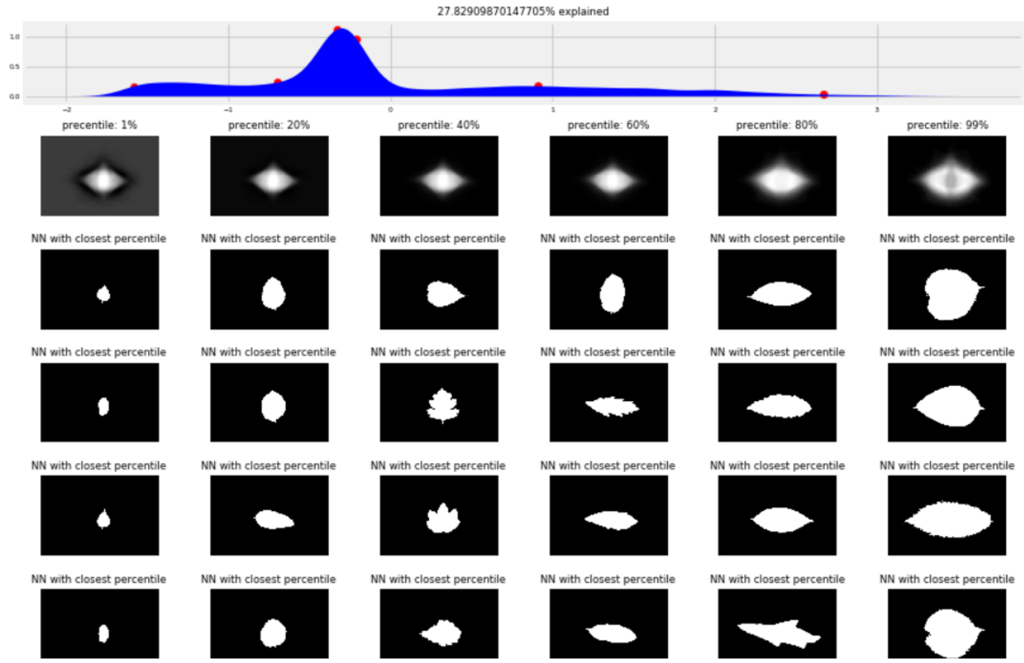
eigenvalue left are most different ones.



Show model variations around mean image

[9]:

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

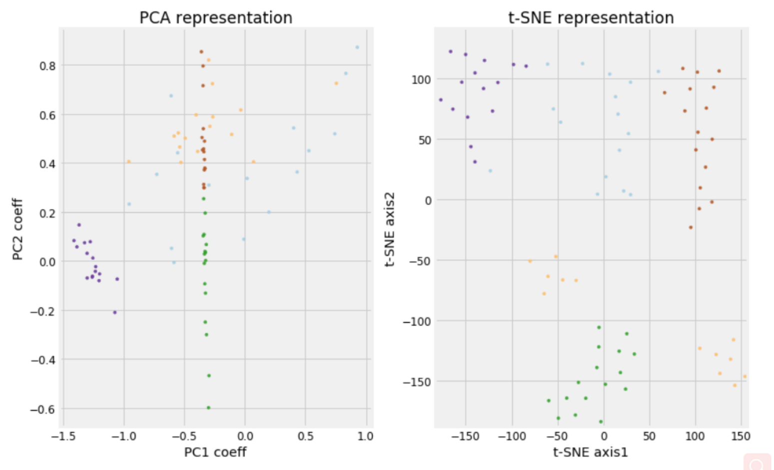


Analyze Eigenvector 1 as an example: from left to right, the scale of the leaves becomes larger and larger.

3]:

```
#%% plot scatter of 2 PCs and t-SNE of all PCs (with labels as colors)
matplotlib.rcParams['font.size'] = 12
matplotlib.rcParams['figure.figsize'] = (12,8)

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

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

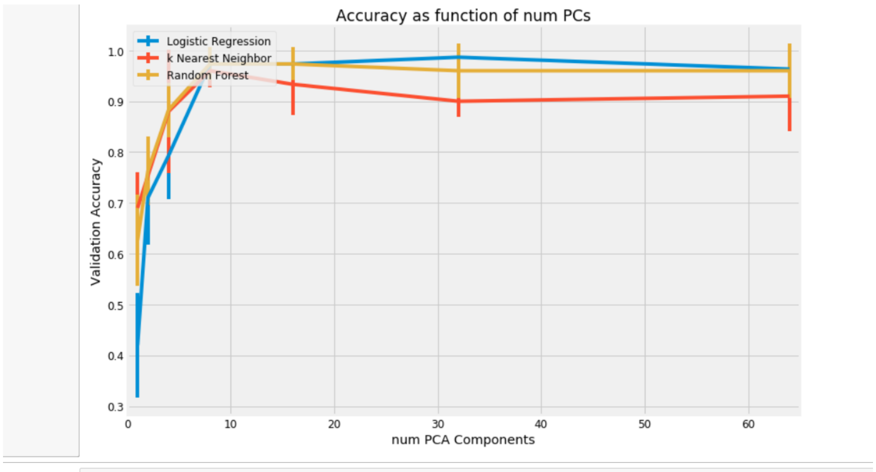
Show Scatter plot of Leaf images as points in high-dimensional space. PCA use only two parameters. And t-SNE means that showing the high-dimensional pictures into 2d picture.



Train the PCA using KNN:

```
for numPCs in numPCsToUse:
    stratifiedCV = model_selection.StratifiedKFold(n_splits=5, random_state=1)
    logRegAccuracy = []; kNN_Accuracy = []; RF_Accuracy = []
    for trainInds, validInds in stratifiedCV.split(X_PCA_train, y_train):
        X_train_cv = X_PCA_train[trainInds,:numPCs]
        X_valid_cv = X_PCA_train[validInds,:numPCs]

        y_train_cv = y_train[trainInds]
        y_valid_cv = y_train[validInds]

        logReg.fit(X_train_cv, y_train_cv)
        kNN.fit(X_train_cv, y_train_cv)
        RF.fit(X_train_cv, y_train_cv)

        logRegAccuracy.append(accuracy_score(y_valid_cv, logReg.predict(X_valid_cv)))
        kNN_Accuracy.append(accuracy_score(y_valid_cv, kNN.predict(X_valid_cv)))
        RF_Accuracy.append(accuracy_score(y_valid_cv, RF.predict(X_valid_cv)))

    logRegMeanAccuracy.append(np.array(logRegAccuracy).mean())
    logRegAccuracyStd.append(np.array(logRegAccuracy).std())

    kNN_MeanAccuracy.append(np.array(kNN_Accuracy).mean())
    kNN_AccuracyStd.append(np.array(kNN_Accuracy).std())

    RF_MeanAccuracy.append(np.array(RF_Accuracy).mean())
    RF_AccuracyStd.append(np.array(RF_Accuracy).std())
```

show the Result:



Show Model Accuracy as function of num PCA components

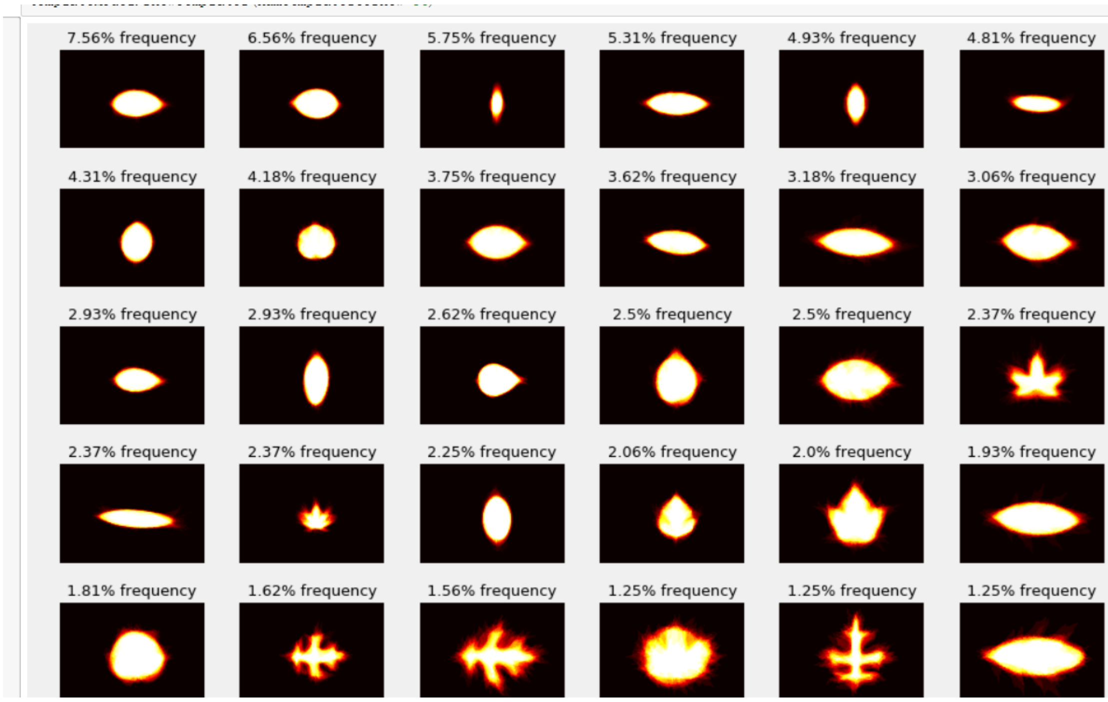
Summary:

We can find that we can use PCA and T-SNE to reduce the dimension. We can get eigenvalues from the process and use clustering the data. As a conclusion, we can know that there are some rules in the dataset and we can use it to do the machine learning.

Part2 use k-mean to process data

```
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')
```

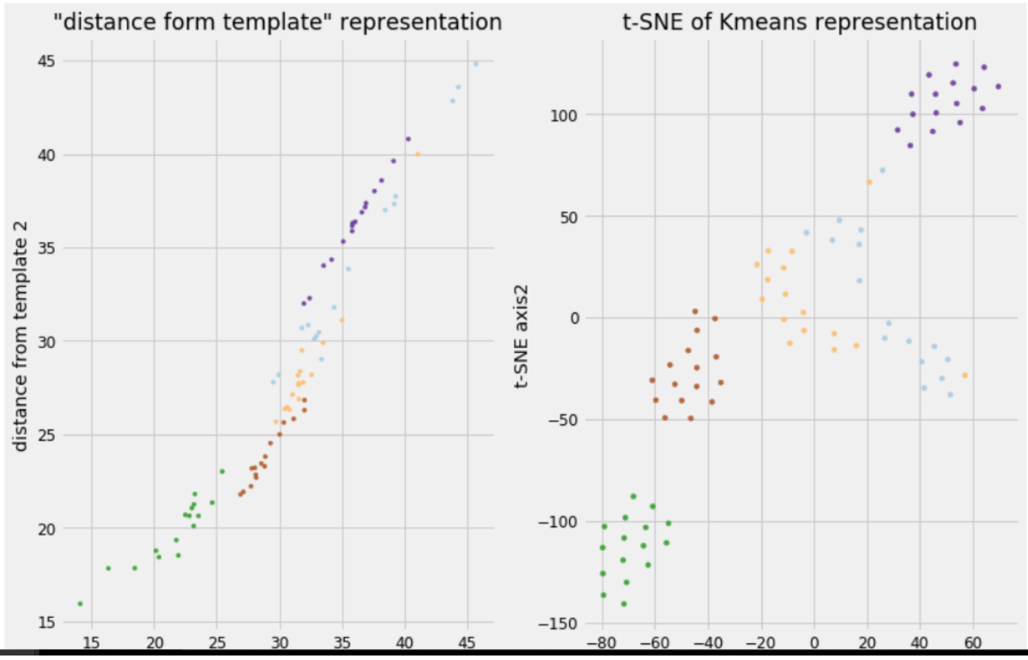
Mode a k-mean model



Apply k-means and cluster all the data

Try k=36;

```
[28]: 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')
```



Visualize "distance from cluster centers" feature space

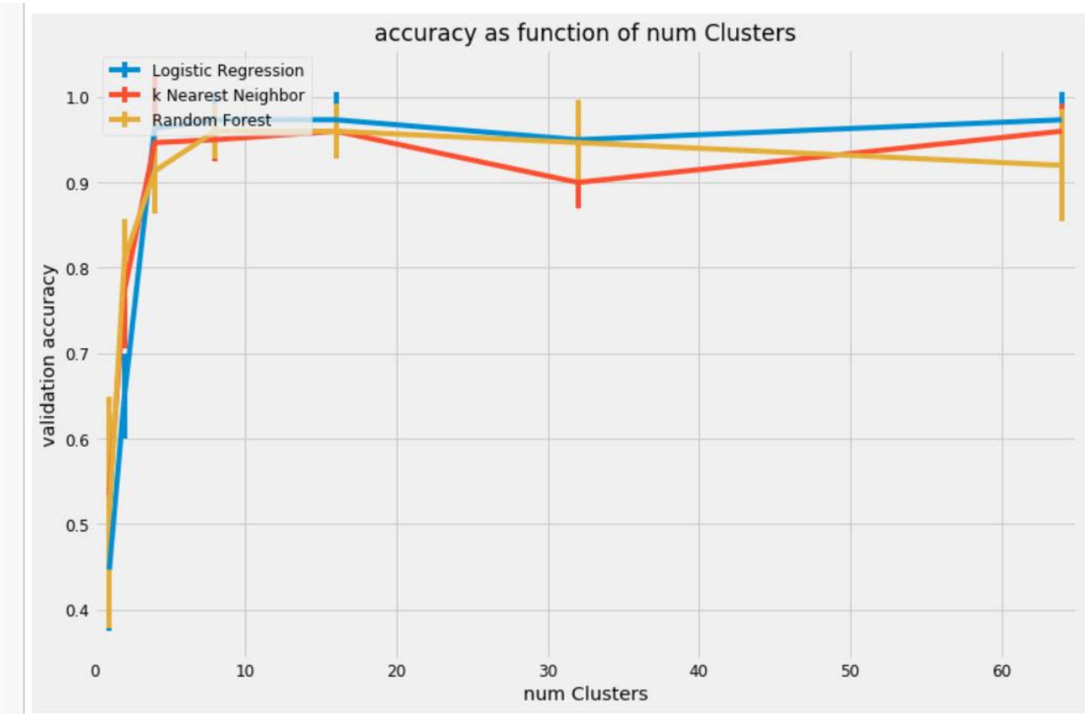
Set 2 points and use a triangle method to decide the third points and doing

the cluster work. T-SNE is also used for drawing a 2d picture with high-dimension parameters.

```
for k in numClustersToUse:
    stratifiedCV = model_selection.StratifiedKFold(n_splits=5, random_state=1)
    logRegAccuracy = []; kNN_Accuracy = []; RF_Accuracy = []

    templateModel = KmeansModel(X_train, numClusters=k)
    X_kmeans_train = templateModel.RepresentUsingModel(X_train, representationMethod='distFromAllClusters')

    for trainInds, validInds in stratifiedCV.split(X_kmeans_train, y_train):
        X_train_cv = X_kmeans_train[trainInds,:]
        X_valid_cv = X_kmeans_train[validInds,:]

        y_train_cv = y_train[trainInds]
        y_valid_cv = y_train[validInds]

        logReg.fit(X_train_cv, y_train_cv)
        kNN.fit(X_train_cv, y_train_cv)
        RF.fit(X_train_cv, y_train_cv)

        logRegAccuracy.append(accuracy_score(y_valid_cv, logReg.predict(X_valid_cv)))
        kNN_Accuracy.append(accuracy_score(y_valid_cv, kNN.predict(X_valid_cv)))
        RF_Accuracy.append(accuracy_score(y_valid_cv, RF.predict(X_valid_cv)))

    logRegMeanAccuracy.append(np.array(logRegAccuracy).mean())
    logRegAccuracyStd.append(np.array(logRegAccuracy).std())

    kNN_MeanAccuracy.append(np.array(kNN_Accuracy).mean())
    kNN_AccuracyStd.append(np.array(kNN_Accuracy).std())

    RF_MeanAccuracy.append(np.array(RF_Accuracy).mean())
    RF_AccuracyStd.append(np.array(RF_Accuracy).std())
```

Train the mode and get the accuracy:

Result:



Summery:

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.

Summery about the first 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.

Part4：simple train using CNN

Train and test type: csv.file

Train set:1240

Single convolutional layer

Activation method: relu;

Train method: softmax;

Loss: categorical_crossentropty;

Opimizer: sgd;

```
]: # 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
```

```
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
```

The original one:

```
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
```

Change softmax to sigmoid:

```
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
```

Change categorical_crossentropy to binary_crossentropy:

```
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
```

Change the epoch from 15 to 8:

```
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
```

Change the optimizer from rgd to adam:

Summary:

when we use the epoch 15, softmax , sgd categorical_crossentropy we can get the best accuracy. Binary_crossentropy may not suitable for our dataset because we can't make out data binary.

Part5:using CNN to train csv file and image at the same time

Train set: image and 1240 csv

Test set: 357 csv

```
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: 1.000
Epoch 4/10
12384/12384 [==============================] - 1509s 122ms/step - loss: 0.0292 - acc: 0.9913 - val_loss: 3.6244e-04 - val_acc: 1.000
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: 1.000
Epoch 7/10
12384/12384 [==============================] - 1512s 122ms/step - loss: 0.0217 - acc: 0.9939 - val_loss: 1.9166e-04 - val_acc: 1.000
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: 1.000
Validation loss: 0.00009
Validation accuracy: 100.00
```

Then we also do the same work as part 4, just change some methods, parameters activation and the size of the image. But the process is too long and we forget to get the screen shot, so we only post the highest one.

Summary:

we found that this one is the best one. When we use the image and the csv file at the same time, our accuracy is almost 1.

Results:

First, we use PCA and K-mean to prove that the eigenvalues have some rules and we can do some train using models.

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.

Discussion:

1. Whether it is okay for us to reduce some parameters and then we can still get such a high accuracy.
2. If we only use the image part, can we get the similar accuracy.
3. Is there any other model except we can use to train the dataset, and how is the accuracy?
4. Is there any other method we can use to prove the feasibility of the machine learning?

Reference:

Dataset:https://archive.ics.uci.edu/ml/datasets/One-hundred+plant+species+leaves+data+set

Course: https://onlinecourses.science.psu.edu/stat505/node/49

Researchpaper:

https://www.researchgate.net/publication/266632357_Plant_Leaf_Classification_using_Probabilistic_Integration_of_Shape_Texture_and_Margin_Features

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

feature extraction from images:https://www.python.org/dev/peps/pep-0008/