Non Linear Classification using Kernel Methods

Aayush Mudgal
12008

Sheallika Singh
12665

Abstract

Linear Models are not rich enough to capture many of the real-world patterns and it is often desired to capture non-linear patterns in the data. We briefly look towards Support Vector Machines and Nearest Neighbors as Classification methods, that can be kernelized. We also look at different formulations of SVM's, namely C-SVM and nu-SVM. We depict the wide applicability and ease of Kernel SVMs through real-world problems like in face detection, handwritten character detection, spam/non-spam classification.

1 Introduction

Kernel based methods have shown significant success in classification, regression as well as unsupervised learning problems. Kernel-based algorithms have shown to be successful in a wide area of applications including in the context of object detection, gene expression, handwritten character recognition, spam mail detection, textual classification([5][4][9][7][6]. In Many of the classifications problems, the classes may not be linearly separable. Clearly linear algorithms tend to give poor results in these tasks.However `Kernel Methods` (wherever applicable), makes linear model work in non-linear settings. Such a mapping of data to a higher dimensions using kernel functions, helps to exhibit linear patterns in this higher dimension. It is also shown that different kernel methods for classification can be reduced to optimization of a convex cost function. We primarily look at two cases of Suport Vector Machines [3], namely C-SVM and Nu-SVM, and their Kernel counterparts.We also look at other non-linear classifier K-nearest neighbors and its kernelised version.

2 Main Body

2.1 Non-Linear Algorithms in Kernel Feature Spaces

In most of the practical cases the Learning Set($\mathcal{L}$) is not linearly separable in the input space. Kernel functions essentially help to map the input space $\phi : \mathcal{X} \rightarrow \mathcal{F}$ by a non-linear mapping ϕ to another feature space $\mathcal{F}$. $\mathcal{F}$ is usually of much higher dimension than the input space. Such higher dimensional representation of the input vectors, may result in vectors being linearly separated in the mapped feature space. The idea is thus to learn develop the classifier in this feature space.

`Kernel Trick`: Any Learning algorithm that works entirely with inner products can be kernelized. A valid Kernel mapping [1], allows to work in the Feature space without explicitly requiring to calculate the vector representation in the Feature space ($\mathcal{F}$). The easy computation of the kernel function (as it is in terms of inner products), helps to keep the computational complexity of the algorithm in check and most often kernelization comes at a very little overhead. Kernel trick thus allows us to get the effect of working in $\mathcal{F}$ through a non-linear mapping where it might be possible to develop linear algorithms. Since a kernel function is quite similar to just an inner product, it can be visualized as a similarity metric. Such a visualization is quite common specially in the case of

[1]k: $\mathcal{X}x\mathcal{X} \rightarrow \mathbb{R}$ is a valid kernel iff $\forall x, y \in \mathcal{X}, k(x, y) = \langle\phi(x), \phi(y)\rangle$, where $\phi : \mathcal{X} \rightarrow \mathcal{F}$, where F is a valid inner product space

structured data (textual data, graphical data). Still the major tasks that remains is while choosing the right type of the algorithm. Radial Basis Function (rbf) [2], Polynomial[3], Linear are some of the most common types of kernels.

2.2 SVM Formulation (Separable Case)

Given an instance of Learning Set $\mathcal{L}$ given by $\{(x_i, y_i)|i = 1..n, x_i \in \mathcal{X}, y_i \in \{-1, 1\}\}$, the support vector machines (SVM)[3] tries to maximize the margin.Separating hyperplane is in the midway between the margin planes. Maximising the margin we get convex optimization problem for which the dual problem is as follows:

$$\begin{aligned} \underset{\bar{\alpha}}{min}\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j\langle x_i, x_j\rangle - \sum_{i=1}^{n}\alpha_i \\ s.t.\alpha_i \geq 1, \forall i \in \{1..n\} \\ \sum_{i=1}^{n}\alpha_i y_i = 0 \end{aligned}$$

From Strong duality, the primal and dual problems have the same solution. Thus solving the dual problem (a convex optimization problem) gives us the optimal solution $\bar{\alpha}^*$. The separating hyperplane can be recalculated as follows: $w^* = \sum_{i=1}^{n}\bar{\alpha_i}^* y_i * x_i$. Complementary slackness condition is used to find the optimal w_0^*, which is given by $w_0^* = y_j - \sum_{i=1}^{n}\alpha_i^* y_i^* \langle x_i, x_j\rangle$, for some j satisfying $\alpha_i^* > 0$

2.3 C-SVM Formulation (Non-Separable Case)

As observed in Section (**??**), the Normal SVM algorithm makes a very strong assumption that the learning set is linearly separable. However such an assumption normally does not hold in practical purposes. Therefore C-SVM a modification of the normal SVM is more popularly used. As it relaxes the condition that $y_i g(x_i) \geq 1, \forall i \in \{1..n\}$. With the introduction of slack variables the constraint is modified as, $y_i g(x_i) \geq 1 - \xi_i, \forall i \in \{1..n\} and \xi_i \geq 0$. Such a modification allows to give some penalties for mis-classification. The regular correctly classified vectors that are on the margin and beyond are represented by $y_i g(x_i) \geq 1$ and $\xi_i = 0$. Points that are correctly classified but between the margin are represented by $0 < y_i g(x_i) < 1$, and $0 < \xi_i < 1$, while the misclassified points are represented by $y_i g(x_i) < 0$ and $\xi_i > 1$ The C-SVM requires the solution of the following convex optimization problem given by Equation (2.3). C is a regularization parameter (which is tuned through cross-validation). Regularization parameter acts as a balance between widening the margin and allowing misclassified and margin points. A small C favors a larger margin. C=0, is equivalent to the normal svm problem.

$$\begin{aligned} \underset{w,w_0,\xi}{min}\frac{||w||^2}{2} + C\sum_{i=1}^{n}\xi_i \\ s.t.y_i g(x_i) \geq 1 - \xi_i, \forall i \in \{1..n\}(x_i, y_i) \in \mathcal{L} \\ \xi_i \geq \forall i \in \{1..n\}(x_i, y_i) \in \mathcal{L} \end{aligned}$$

[2] $K(x, x^{'}) = exp\big(-\gamma||x - x^{'}||^2\big)$ is a general case rbf kernel

[3] $K(x, x^{'}) = (x^T x^{'} + c)^d$ is a general case polynomial kernel of degree d

The dual problem of Equation (2.3) is given as follows:

$$\underset{\bar{\alpha}}{min}\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j\langle x_i, x_j\rangle - \sum_{i=1}^{n}\alpha_i$$
$$s.t. 0 \le \alpha_i \le C, \forall i \in \{1..n\}$$
$$\sum_{i=1}^{n}\alpha_i y_i = 0$$

From strong duality the solution to the dual problem gives the solution to the primal problem. Hence the separating hyper-plane can be calculated from the solution of the dual problem as follows.

$$w^* = \sum_{i=1}^{n}\bar{\alpha_i}^* y_i * x_i$$

From the application of KKT conditions it follows that

$$w_0^* = y_j - \sum_{i=1}^{n}\alpha_i^* y_i^* \langle x_i, x_j\rangle$$
$$for some j, s.t. 0 \le \alpha_j \le 1$$

2.4 Nu-SVM Formulation

ν-SVC([2],[8]) is a variant of the soft margin problem of finding the optimal hyperplane [8]. Here parameter C is replaced by a parameter $\nu \in [0, 1]$ which is the lower and upper bound on the number of examples that are support vectors and that lie on the wrong side of the hyperplane, respectively. In case of C-SVM, C could have taken any real positive value, as opposed to the additional bound here. We get a quadratic optimization problem as a dual problem which can be kernelised in the similar manner as C-SVC (shown later in section 2.4.1

Proposition:(Connection between C-SVC and ν-SVC[1][8]) If ν-SVC leads to $\rho \ge 0$, then C-SVC, with C set a priori to 1/mρ, leads to the same decision function. Despite the bound on the number of support vectors, ν SVM is difficult to optimize and thus not very suitable for big datasets.

2.4.1 Kernelization of SVM and C-SVM and ν-SVM

Both SVM and C-SVM can be solved by solving the corresponding dual problems. The dual problem is almost similar to that of the Normal SVM problem, except that α_i is bounded above by the regularization parameter. Input feature vectors, i.e. $x \in \mathcal{X}$ occur in the dual optimization problem in the form of inner products. Hence the dual optimization problem can be kernalized in case of both SVM and C-SVM. The inner product $\langle x_i, x_j\rangle$ would be replaced by $\langle\phi(x_i), \phi(x_j)\rangle$, if the input space is mapped to a RKHS feature space $\mathcal{F}$, where the mapping is satisfied by some kernel function K. The Kernel Trick allows us to simplify the inner product $\langle\phi(x_i), \phi(x_j)\rangle$ by $K(x_i, x_k$. However the algorithm can be kernelized only if the input vectors appear in the form of inner products also during the prediction step. In both the formulations any test vector x is given a label by:

$$sgn(w^* x + w_0)$$
$$\implies sgn(\sum_{i=1}^{n}\alpha_i^* y_i\langle\phi(x_i), \phi(x)\rangle$$
$$\implies sgn(\sum_{i=1}^{n}\alpha_i^* y_i\langle K(x_i, x)$$

Hence both the optimization problem and the decision function can be written in terms of only inner product of input space vectors, both SVM and C-SVM can be kernelized

3 Kernel Nearest neighbor[10]

In the k-nearest neighbor a query point is classified by finding the k closest neighbors (w.r.t. distance between the and query point) and then assigning the majority label among these k points to the

query point. The norm distance, which is used in k-nearest neighbor algorithm, can be denoted as: $d(x, y) = ||x - y||$
The square of norm distance can be written as: $d^2(x, y) = \langle x, x\rangle - 2 \times \langle x, y\rangle + \langle y, y\rangle$
This square of norm distance can be written in form of inner product and thus we can use 'Kernel Trick' to kernelise k-nearest neighbor algorithm.

4 Simulation/Results

4.0.1 Spam-Non Spam Dataset

This is a 2-class dataset consisting of about 2800 e-mail messages with text,and subject. Each of which is classified into spam (480) or non-spam messages. The 2-D visualization plots for different Kernel functions variations over the basic C-SVM classifiers along with the respective accuracies are depicted in Figure(1).

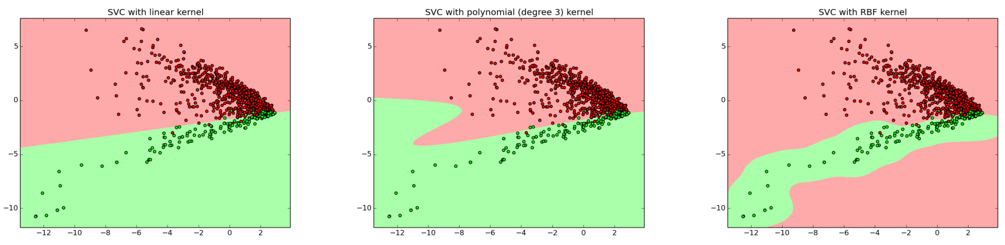

Figure 1: Different Kernel Functions for C-SVM

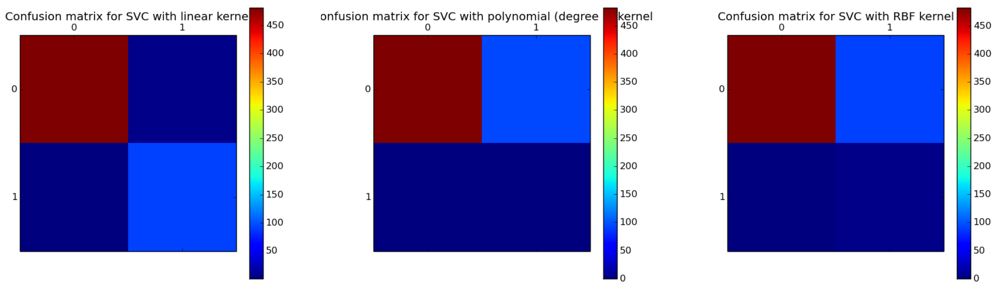

Figure 2: Confusion Matrix Plot for different Kernel Function for C-SVM

4.0.2 Handwritten Character Dataset

We chose the famous MNIST Data-set [4] of 60,000 handwritten digits, wherein each image is of size 28x28 pixels, and can thus be considered as points in a 784 dimensional space We tested different kernel functions for each of C-SVM, nu-SVM and Kernel KNN as depicted in Figure (3), Figure(4), Figure(5). We plot the visualizations for the digits (0,8,9) as they are among the most mis-classified digits.

We could not compute the gram matrix in case of Kernel KNN classifier, because of the huge size of the matrix, and the limited computational power. And also, since Kernel KNN classifier was performing significantly worse for the 3-digit restricted data-set.

[4]Data-set is available at `http://yann.lecun.com/exdb/mnist/`

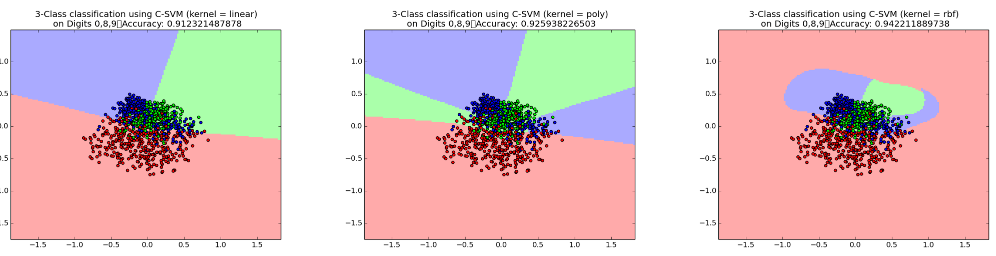

Figure 3: Plot for digits (0,8,9) along with different Kernel Function for C-SVM

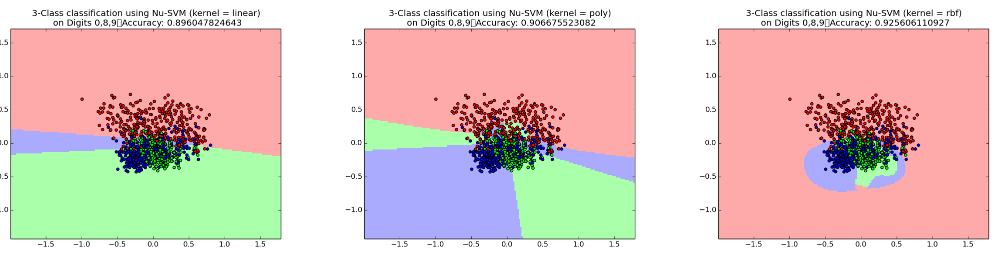

Figure 4: Plot for digits (0,8,9) along with different Kernel Function for Nu-SVM

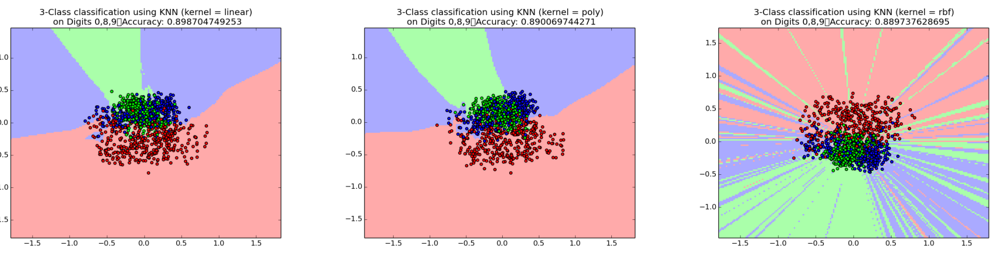

Figure 5: Plot for digits (0,8,9) along with Kernel Function for Kernelized KNN

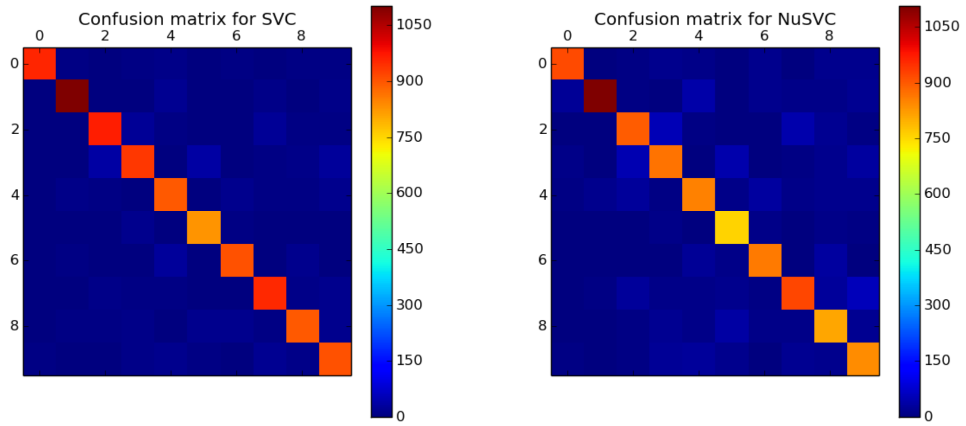

Figure 6: Confusion Matrix Plot for MNIST dataset

4.0.3 Face Detection Data-set

We use the ATT FaceDatabase [5] and the FaceRec library [6] for face detection task. We used FisherFaces as a Feature Extractor, followed by our choice algorithms. We obtained the best validation accuracy of 97.75% for the KNN with RBF kernel and number of nearest neighbor equals to 4. FisherFaces acts as a dimensional reduction techniques, making KNN scalable for larger data

5 Conclusions

In the case of the two class classification problem of spam and non-spam messages, we observer that linear classifiers clearly outperform the other Kernel classifiers. Thus implying that there was linearity in the input space. In such a situation, usage of kernel functions does not seem to add benefit.

However in the case of MNIST data-set where the input space was observed to be non-linear, we observe that the Kernelized Classifiers, almost always reported a higher accuracy. KNN Classifier with a euclidean metric was the worst classifier for this task. This primarily can be related to the way that different digits can be written in different styles, which have different distances among them.

We also noted that Kernel Nearest Neighbor Classifier suffered from longer training times, and also increased requirement of computations during the testing time. Kernel functions compensated for the increase in the training time, through significant increases in accuracy in some cases. We observe that C-SVM emerges as the choice classifier to be used in practice.

The success of a kernel classifier thus deeply dependent on the chosen kernel function. Any failure of Kernel method is probably because of the incapability in selecting the desired kernel function that could model such complex data. Construction of a proper kernel remains an important obstacle for the successful application of these algorithms in some cases.

References

[1] D. Burges and C. Crisp. A geometric interpretation of v- svm classifiers. *Advances in neural information processing systems*, 12(12):244, 2000.

[2] P.-H. Chen, C.-J. Lin, and B. Schölkopf. A tutorial on ν-support vector machines. *Applied Stochastic Models in Business and Industry*, 21(2):111–136, 2005.

[3] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[4] D. Decoste and B. Schölkopf. Training invariant support vector machines. *Machine learning*, 46(1-3):161–190, 2002.

[5] Y. LeCun, L. Jackel, L. Bottou, C. Cortes, J. S. Denker, H. Drucker, I. Guyon, U. Muller, E. Sackinger, P. Simard, et al. Learning algorithms for classification: A comparison on handwritten digit recognition. *Neural networks: the statistical mechanics perspective*, 261:276, 1995.

[6] K.-R. Müller, S. Mika, G. Rätsch, K. Tsuda, and B. Schölkopf. An introduction to kernel-based learning algorithms. *Neural Networks, IEEE Transactions on*, 12(2):181–201, 2001.

[7] D. Roobaert and M. M. Van Hulle. View-based 3d object recognition with support vector machines. In *Neural Networks for Signal Processing IX, 1999. Proceedings of the 1999 IEEE Signal Processing Society Workshop.*, pages 77–84. IEEE, 1999.

[8] B. Schölkopf, A. J. Smola, R. C. Williamson, and P. L. Bartlett. New support vector algorithms. *Neural computation*, 12(5):1207–1245, 2000.

[9] S. P. Schölkopf, V. Vapnik, and A. Smola. Improving the accuracy and speed of support vector machines. *Advances in neural information processing systems*, 9:375–381, 1997.

[10] K. Yu, L. Ji, and X. Zhang. Kernel nearest neighbor algorithm. *Neural Processing Letters*, 15(2):147–156, 2002.

[5]Dataset is available at `http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html`

[6]FaceRec library is available for Python at `https://github.com/bytefish/facerec`

```
# Part of code used for development
# sklearn does not provide an interface for Kernel KNN Classifier, the
    same was thus implemented by seeking help from
    https://github.com/jsantarc/Kernel-Nearest-Neighbor-Algorithm-in-Python-
```

```
# visualization code for MNIST data-set
import os, struct
from array import array as pyarray
from numpy import arange, array, int8, uint8, zeros, array, append
from struct import unpack
from skimage.feature import hog
from sklearn.cross_validation import StratifiedShuffleSplit
import cv2
import matplotlib.pyplot as plt
from sklearn.grid_search import GridSearchCV
from sklearn.decomposition import PCA
from time import time
from operator import itemgetter
from scipy.stats import randint
import cv2
import numpy as np
from sklearn import svm, ensemble , tree
from sklearn.metrics import confusion_matrix
import pylab as pl
from cPickle import dump, load
from skimage.feature import hog
import skimage
from matplotlib.colors import ListedColormap
from sklearn.utils import shuffle
import numpy as np
from sklearn import neighbors, datasets
from sklearn.metrics.pairwise import pairwise_kernels
import numpy as np
FILTER="hog"
def load_mnist(dataset="training", digits=np.arange(10), path="."):
   """
   Loads MNIST files into 3D numpy arrays

   Adapted from: http://abel.ee.ucla.edu/cvxopt/_downloads/mnist.py
   and http://g.sweyla.com/blog/2012/mnist-numpy/
   """

   if dataset == "training":
      fname_img = os.path.join(path, 'train-images.idx3-ubyte')
      fname_lbl = os.path.join(path, 'train-labels.idx1-ubyte')
   elif dataset == "testing":
      fname_img = os.path.join(path, 't10k-images.idx3-ubyte')
      fname_lbl = os.path.join(path, 't10k-labels.idx1-ubyte')
   else:
      raise ValueError("dataset must be 'testing' or 'training'")

   flbl = open(fname_lbl, 'rb')
   magic_nr, size = struct.unpack(">II", flbl.read(8))
   lbl = pyarray("b", flbl.read())
   flbl.close()

   fimg = open(fname_img, 'rb')
   magic_nr, size, rows, cols = struct.unpack(">IIII", fimg.read(16))
   img = pyarray("B", fimg.read())
   fimg.close()

   ind = [ k for k in range(size) if lbl[k] in digits ]
   N = len(ind)
   images = zeros((N, rows, cols), dtype=uint8)
   labels = zeros((N, 1), dtype=int8)
   for i in range(len(ind)):
      images[i] = array(img[ ind[i]*rows*cols : (ind[i]+1)*rows*cols
          ]).reshape((rows, cols))
      labels[i] = lbl[ind[i]]

   return images, labels
def extract_hog(X_train_name,Y_name, save=True, filetype="Train"):
   hogs = []
   count =0
   total = len(Y_name)
   for imgfile in X_train_name:
      cv2.imwrite("temp.png", imgfile)
      img = cv2.imread("temp.png")
      #print img
      img = cv2.resize(img,(28,28))
      gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
      fd = hog(gray_img,normalise =True, orientations=9,
          pixels_per_cell=(14, 14), cells_per_block=(1,1),
          visualise=False)
      # comp = skimage.feature.hog(img)
      hogs.append( fd )
      # lens.add(comp.shape)
      #print type(comp)
      count+=1
      print "done", count, total
   X_train = np.array(hogs , 'float64' )

   if save==True:
      dump( X_train, open( FILTER+filetype+"1", "wb" ) )
      dump(Y_name, open("Y_"+filetype+"1", "wb"))
      print "Files stored: "+FILTER+filetype +"and Y_"+filetype
   return X_train, np.array(Y_name)

def refineSets(images, labels, size):
   result = zeros(10)
   count =0
   X=[]
   Y=[]

   for i in range(0,len(labels)):
      label = labels[i]
      feature = images[i]
      if result[label]<=size:
         X.append(feature)
         Y.append(label)
         count+=1
         result[label]+=1
      if count > size*10:
         break

   return array(X), np.array(Y)


def saveFeatures(X_name, Y_name, save=True,filetype="Train"):

   if FILTER=="hog":
      return extract_hog(X_name,Y_name,save, filetype)

# help sought from
    https://github.com/jsantarc/Kernel-Nearest-Neighbor-Algorithm-in-Python-
def KernelKNNClassifierFit(X,Y,Kernel,Parameters):
   Y= np.array(Y)
```

```
    #Number of training samples
    N=len(X);
    #Array sorting value of kernels k(x,x)
    Gram_Matrix=np.zeros(N);
    #Calculate Kernel vector between same vectors Kxx[n]=k(X[n,:],X[n,:])
    #dummy for kernel name
    for n in range(0,N):

      Gram_Matrix[n]=pairwise_kernels(X[n],
          metric=Kernel,filter_params=Parameters)
    return Gram_Matrix

def predict(X_test,X_train,Kernel,Parameters, Gram_Matrix, Y_train,
    n_neighbors=1):

    Nz=len(X_test)
    #Empty list of predictions
    yhat= np.zeros(Nz);
    #number of samples for classification
    #Number of training samples
    Nx=len(X_train);
    #Dummy variable Vector of ones used to get rid of one loop for k(z,z)
    Ones=np.ones(Nx);

    #squared Euclidean distance in kernel space for each training sample
    Distance=np.zeros(Nx)
    # Index of sorted values
    Index= np.zeros(Nx)

    # calculate pairwise kernels beteen Training samples and prediction
        samples
    Kxz=pairwise_kernels(X_train,X_test,
        metric=Kernel,filter_params=Parameters)

    NaborsNumberAdress=range(0,n_neighbors)

    #Calculate Kernel vector between same vectors Kxx[n]=k(Z[n,:],Z[n,:])

    for n in range(0,Nz):
       # calculate squared Euclidean distance in kernel space for each
           training sample
       #for one prediction
       #for m in range(0,Nx)
       #Distance[m]=|phi(x[m])-phi(z[n])|^2=k(x,x)+k(z,z)-2k(z,x)

       Distance =Gram_Matrix+pairwise_kernels(X_test[n],
           metric=Kernel,filter_params=Parameters)*Ones-2*Kxz[:,n]

       #Distance indexes sorted from smallest to largest
       Index=np.argsort(Distance.flatten());
       Index=Index.astype(int)

       #get the labels of the nearest feature vectors
       yl=list(Y_train[Index[NaborsNumberAdress]])
       #perform majority vote
       yhat[n]=max(set(yl), key=yl.count)
    return(yhat)


if __name__ == "__main__":

    # train_images, train_labels = refineSets(train_images, train_labels,
        1111) # working with smaller subset
    # test_images, test_labels = refineSets(test_images, test_labels,
        111) # working with a smaller subset
```

```
retrain = raw_input("Feature Extraction??")

if retrain=="y":
   train_images,train_labels = load_mnist('training', digits=[8,9])
   test_images,test_labels = load_mnist('testing',digits=[8,9])
   X_train, Y_train = saveFeatures(train_images,train_labels)
   X_test, Y_test= saveFeatures(test_images, test_labels,save=True,
       filetype="Test")
   Y_train=Y_train.flatten()
   Y_test = Y_test.flatten()
else:
   with open(FILTER+"Train1", 'rb') as fp:
      X_train = load(fp)
   with open("Y_Train1", 'rb') as fp:
      Y_train = load(fp)
   with open(FILTER+"Test1", 'rb') as fp:
      X_test = load(fp)
   with open("Y_Test1", 'rb') as fp:
      Y_test = load(fp)
   Y_train=Y_train.flatten()
   Y_test = Y_test.flatten()

# hog features extracted
# i have the hog features, I need to apply SVM the kernel svm and
    other stuff... and nearest neighbor classifier too,
# i need to do some visualizations.. Need to add confusion matrix
# Visualization help sought from
    https://github.com/saradhix/mnist_svm/blob/master/plot_mnist_svm.py
num_samples_to_plot = 1000
X_train, Y_train = shuffle(X_train, Y_train)
X_train, Y_train = X_train[:num_samples_to_plot],
    Y_train[:num_samples_to_plot]
pca = PCA(n_components=2)
X = pca.fit_transform(X_train)

print X.shape
y=Y_train


h = .02 # step size in the mesh
n_neighbors=4
# Create color maps
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

for k in ['poly']:
   # we create an instance of Neighbours Classifier and fit the data.
   if k=='linear':
      clf = KernelKNNClassifierFit(X,y,'linear',0)
      clf1 = KernelKNNClassifierFit(X_train,Y_train,'linear',0)
      Z1 = predict(X_test,
          X_train,'linear',0,clf1,Y_train,n_neighbors)
      accuracy = np.mean(Z1==Y_test)
      print accuracy
      raw_input()
   elif k=='poly':
      clf= KernelKNNClassifierFit(X,y,'poly',3)
      clf1 = KernelKNNClassifierFit(X_train,Y_train,'poly',3)
      Z1 = predict(X_test, X_train,'poly',3,clf1,Y_train,n_neighbors)
      accuracy = np.mean(Z1==Y_test)
      print accuracy
elif k=='rbf':
      clf= KernelKNNClassifierFit(X,y,'rbf',1)
      clf1 = KernelKNNClassifierFit(X_train,Y_train,'rbf',1)
      Z1 = predict(X_test, X_train,'rbf',1,clf1,Y_train,n_neighbors)
```

```
            accuracy = np.mean(Z1==Y_test)
            print accuracy


        # I have gramMatrix in clf


        # Plot the decision boundary. For that, we will assign a color to
            each
        # point in the mesh [x_min, m_max]x[y_min, y_max].
        x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
        y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
        xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                             np.arange(y_min, y_max, h))
        if k=='linear':
            Z=predict(np.c_[xx.ravel(), yy.ravel()], X,'linear',0,
                clf,y,n_neighbors)
        if k=='poly':
            Z=predict(np.c_[xx.ravel(), yy.ravel()],
                X,'poly',3,clf,y,n_neighbors)
        if k=='rbf':
            Z=predict(np.c_[xx.ravel(), yy.ravel()],
                X,'cosine',1,clf,y,n_neighbors)
        print Z
        # Put the result into a color plot
        Z = Z.reshape(xx.shape)
        plt.figure()
        plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

        # Plot also the training points
        plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold)
        plt.xlim(xx.min(), xx.max())
        plt.ylim(yy.min(), yy.max())

        plt.title("3-Class classification using KNN (kernel = "+k+")\n
            "+"on Digits 0,8,9\tAccuracy: " +str(accuracy))
        plt.savefig("KNN089-Final"+k+".png")
```

```
# code for SVC, nuSVC MNIST
import os, struct
from array import array as pyarray
from numpy import arange, array, int8, uint8, zeros, array, append
from struct import unpack
from skimage.feature import hog
from sklearn.tree import DecisionTreeClassifier
from sklearn.cross_validation import StratifiedShuffleSplit
import cv2
import matplotlib.pyplot as plt
from sklearn.grid_search import GridSearchCV
from sklearn.metrics import confusion_matrix

from time import time
from operator import itemgetter
from scipy.stats import randint
import cv2
import numpy as np
from sklearn import svm, ensemble , tree
from sklearn.metrics import confusion_matrix
import pylab as pl
from cPickle import dump, load
from skimage.feature import hog
import skimage
from sklearn.neighbors import KNeighborsClassifier
from matplotlib.colors import ListedColormap
FILTER="hog"
def load_mnist(dataset="training", digits=np.arange(10), path="."):
   """
   Loads MNIST files into 3D numpy arrays

   Adapted from: http://abel.ee.ucla.edu/cvxopt/_downloads/mnist.py
   and http://g.sweyla.com/blog/2012/mnist-numpy/
   """

   if dataset == "training":
      fname_img = os.path.join(path, 'train-images.idx3-ubyte')
      fname_lbl = os.path.join(path, 'train-labels.idx1-ubyte')
   elif dataset == "testing":
      fname_img = os.path.join(path, 't10k-images.idx3-ubyte')
      fname_lbl = os.path.join(path, 't10k-labels.idx1-ubyte')
   else:
      raise ValueError("dataset must be 'testing' or 'training'")

   flbl = open(fname_lbl, 'rb')
   magic_nr, size = struct.unpack(">II", flbl.read(8))
   lbl = pyarray("b", flbl.read())
   flbl.close()

   fimg = open(fname_img, 'rb')
   magic_nr, size, rows, cols = struct.unpack(">IIII", fimg.read(16))
   img = pyarray("B", fimg.read())
   fimg.close()

   ind = [ k for k in range(size) if lbl[k] in digits ]
   N = len(ind)

   images = zeros((N, rows, cols), dtype=uint8)
   labels = zeros((N, 1), dtype=int8)
   for i in range(len(ind)):
      images[i] = array(img[ ind[i]*rows*cols : (ind[i]+1)*rows*cols
          ]).reshape((rows, cols))
      labels[i] = lbl[ind[i]]

   return images, labels
def extract_hog(X_train_name,Y_name, save=True, filetype="Train"):
   hogs = []
   count =0
   total = len(Y_name)
   for imgfile in X_train_name:
      cv2.imwrite("temp.png", imgfile)
      img = cv2.imread("temp.png")
      #print img
      img = cv2.resize(img,(28,28))
      gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
      fd = hog(gray_img,normalise =True, orientations=9,
          pixels_per_cell=(14, 14), cells_per_block=(1,1),
          visualise=False)
      # comp = skimage.feature.hog(img)
      hogs.append( fd )
      # lens.add(comp.shape)
      #print type(comp)
      count+=1
      print "done", count, total
   X_train = np.array(hogs , 'float64' )

   if save==True:
      dump( X_train, open( FILTER+filetype, "wb" ) )
      dump(Y_name, open("Y_"+filetype, "wb"))
      print "Files stored: "+FILTER+filetype +"and Y_"+filetype
```

```
    return X_train, np.array(Y_name)

def refineSets(images, labels, size):
    result = zeros(10)
    count =0
    X=[]
    Y=[]

    for i in range(0,len(labels)):
        label = labels[i]
        feature = images[i]
        if result[label]<=size:
            X.append(feature)
            Y.append(label)
            count+=1
            result[label]+=1
        if count > size*10:
            break

    return array(X), np.array(Y)
def showconfusionmatrix(cm,typeModel):
    pl.matshow(cm)
    pl.title('Confusion matrix for '+typeModel)
    pl.colorbar()
    pl.show()

def report(grid_scores, n_top=3):
    """Report top n_top parameters settings, default n_top=3.

    Args
    ----
    grid_scores -- output from grid or random search
    n_top -- how many to report, of top models

    Returns
    -------
    top_params -- [dict] top parameter settings found in
                  search
    """
    top_scores = sorted(grid_scores,
                        key=itemgetter(1),
                        reverse=True)[:n_top]
    for i, score in enumerate(top_scores):
        print("Model with rank: {0}".format(i + 1))
        print(("Mean validation score: "
               "{0:.3f} (std: {1:.3f})").format(
               score.mean_validation_score,
               np.std(score.cv_validation_scores)))
        print("Parameters: {0}".format(score.parameters))
        print("")

    return top_scores[0].parameters
def run_gridsearch(X, y, clf, param_grid, cv=5):
    """Run a grid search for best Decision Tree parameters.

    Args
    ----
    X -- features
    y -- targets (classes)
    cf -- scikit-learn Decision Tree
    param_grid -- [dict] parameter settings to test
    cv -- fold of cross-validation, default 5

    Returns
    -------
```

```
   top_params -- [dict] from report()
   """
   print "GridSearchCV starting"
   grid_search = GridSearchCV(clf,
                     param_grid=param_grid,
                     cv=cv)
   start = time()
   print "Fit starting"
   grid_search.fit(X, y)

   print(("\nGridSearchCV took {:.2f} "
        "seconds for {:d} candidate "
        "parameter settings.").format(time() - start,
            len(grid_search.grid_scores_)))

   top_params = report(grid_search.grid_scores_, 3)
   return top_params

def saveFeatures(X_name, Y_name, save=True,filetype="Train"):

   if FILTER=="hog":
      return extract_hog(X_name,Y_name,save, filetype)


if __name__ == "__main__":

   # train_images, train_labels = refineSets(train_images, train_labels,
       1111) # working with smaller subset
   # test_images, test_labels = refineSets(test_images, test_labels,
       111) # working with a smaller subset
   retrain = raw_input("Feature Extraction??")

   if retrain=="y":
      train_images,train_labels = load_mnist('training')
      test_images,test_labels = load_mnist('testing')
      X_train, Y_train = saveFeatures(train_images,train_labels)
      X_test, Y_test= saveFeatures(test_images, test_labels,save=True,
          filetype="Test")
      Y_train=Y_train.flatten()
      Y_test = Y_test.flatten()
   else:
      with open(FILTER+"Train", 'rb') as fp:
         X_train = load(fp)
      with open("Y_Train", 'rb') as fp:
         Y_train = load(fp)
      with open(FILTER+"Test", 'rb') as fp:
         X_test = load(fp)
      with open("Y_Test", 'rb') as fp:
         Y_test = load(fp)
      Y_train=Y_train.flatten()
      Y_test = Y_test.flatten()

   # hog features extracted

   # i have the hog features, I need to apply SVM the kernel svm and
       other stuff... and nearest neighbor classifier too,
   # i need to do some visualizations.. Need to add confusion matrix
   # C_range = np.logspace(-2,2,4)
   gamma_range=[2,4,6]
   param_grid = {
         "kernel" :['poly','rbf', 'linear'],
         "gamma": gamma_range,
         'nu' : [0.5]

      }
   clf = svm.NuSVC(kernel='rbf',gamma=6)
   clf.fit(X_train,Y_train)
   predicted = clf.predict(X_test)
   cm = confusion_matrix(predicted, Y_test)
   showconfusionmatrix(cm, "NuSVC")
   print "NuSVC accuracy" ,np.mean(Y_test==predicted)



   clf = svm.SVC(kernel='rbf',gamma=6)
   clf.fit(X_train,Y_train)
   predicted = clf.predict(X_test)
   cm = confusion_matrix(predicted, Y_test)
   showconfusionmatrix(cm, 'SVC')
   print "SVC accuracy" ,np.mean(Y_test==predicted)



   # dump(ts_gs, open( "model1", "wb" ))

# GridSearchCV took 836.51 seconds for 24 candidate parameter settings.
# Model with rank: 1
# Mean validation score: 0.963 (std: 0.002)
# Parameters: {'kernel': 'rbf', 'C': 4.6415888336127775, 'gamma': 2}

# Model with rank: 2
# Mean validation score: 0.963 (std: 0.002)
# Parameters: {'kernel': 'rbf', 'C': 4.6415888336127775, 'gamma': 4}

# Model with rank: 3
# Mean validation score: 0.963 (std: 0.002)
# Parameters: {'kernel': 'rbf', 'C': 4.6415888336127775, 'gamma': 6}

   # print clf.score(hog_test_images,test_labels)

   # i have the hog features, I need to apply SVM the kernel svm and
      other stuff... and nearest neighbor classifier too,
   # i need to do some visualizations.. Need to add confusion matrix

# Nu-SVM parameters
```

```
# code for Kernel KNN MNIST
import os, struct
from array import array as pyarray
from numpy import arange, array, int8, uint8, zeros, array, append
from struct import unpack
from skimage.feature import hog
from sklearn.tree import DecisionTreeClassifier
from sklearn.cross_validation import StratifiedShuffleSplit
import cv2
import matplotlib.pyplot as plt
from sklearn.grid_search import GridSearchCV
from sklearn.decomposition import PCA
from time import time
from operator import itemgetter
from scipy.stats import randint
import cv2
import numpy as np
from sklearn import svm, ensemble , tree
from sklearn.metrics import confusion_matrix
import pylab as pl
from cPickle import dump, load
from skimage.feature import hog
```

```
import skimage
from matplotlib.colors import ListedColormap
from sklearn.utils import shuffle
import numpy as np
from sklearn import neighbors, datasets
from sklearn.metrics.pairwise import pairwise_kernels
import numpy as np
FILTER="hog"
def load_mnist(dataset="training", digits=np.arange(10), path="."):
   """
   Loads MNIST files into 3D numpy arrays

   Adapted from: http://abel.ee.ucla.edu/cvxopt/_downloads/mnist.py
   and http://g.sweyla.com/blog/2012/mnist-numpy/
   """

   if dataset == "training":
      fname_img = os.path.join(path, 'train-images.idx3-ubyte')
      fname_lbl = os.path.join(path, 'train-labels.idx1-ubyte')
   elif dataset == "testing":
      fname_img = os.path.join(path, 't10k-images.idx3-ubyte')
      fname_lbl = os.path.join(path, 't10k-labels.idx1-ubyte')
   else:
      raise ValueError("dataset must be 'testing' or 'training'")

   flbl = open(fname_lbl, 'rb')
   magic_nr, size = struct.unpack(">II", flbl.read(8))
   lbl = pyarray("b", flbl.read())
   flbl.close()

   fimg = open(fname_img, 'rb')
   magic_nr, size, rows, cols = struct.unpack(">IIII", fimg.read(16))
   img = pyarray("B", fimg.read())
   fimg.close()

   ind = [ k for k in range(size) if lbl[k] in digits ]
   N = len(ind)

   images = zeros((N, rows, cols), dtype=uint8)
   labels = zeros((N, 1), dtype=int8)
   for i in range(len(ind)):
      images[i] = array(img[ ind[i]*rows*cols : (ind[i]+1)*rows*cols
          ]).reshape((rows, cols))
      labels[i] = lbl[ind[i]]

   return images, labels
def extract_hog(X_train_name,Y_name, save=True, filetype="Train"):
   hogs = []
   count =0
   total = len(Y_name)
   for imgfile in X_train_name:
      cv2.imwrite("temp.png", imgfile)
      img = cv2.imread("temp.png")
      #print img
      img = cv2.resize(img,(28,28))
      gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
      fd = hog(gray_img,normalise =True, orientations=9,
          pixels_per_cell=(14, 14), cells_per_block=(1,1),
          visualise=False)
      # comp = skimage.feature.hog(img)
      hogs.append( fd )
      # lens.add(comp.shape)
      #print type(comp)
      count+=1
      print "done", count, total
   X_train = np.array(hogs , 'float64' )

   if save==True:
      dump( X_train, open( FILTER+filetype+"1", "wb" ) )
      dump(Y_name, open("Y_"+filetype+"1", "wb"))
      print "Files stored: "+FILTER+filetype +"and Y_"+filetype
   return X_train, np.array(Y_name)

def refineSets(images, labels, size):
   result = zeros(10)
   count =0
   X=[]
   Y=[]

   for i in range(0,len(labels)):
      label = labels[i]
      feature = images[i]
      if result[label]<=size:
         X.append(feature)
         Y.append(label)
         count+=1
         result[label]+=1
      if count > size*10:
         break

   return array(X), np.array(Y)


def saveFeatures(X_name, Y_name, save=True,filetype="Train"):

   if FILTER=="hog":
      return extract_hog(X_name,Y_name,save, filetype)

# help sought from
    https://github.com/jsantarc/Kernel-Nearest-Neighbor-Algorithm-in-Python-
def KernelKNNClassifierFit(X,Y,Kernel,Parameters):
   Y= np.array(Y)
   #Number of training samples
   N=len(X);
   #Array sorting value of kernels k(x,x)
   Gram_Matrix=np.zeros(N);
   #Calculate Kernel vector between same vectors Kxx[n]=k(X[n,:],X[n,:])
   #dummy for kernel name
   for n in range(0,N):

     Gram_Matrix[n]=pairwise_kernels(X[n],
         metric=Kernel,filter_params=Parameters)
   return Gram_Matrix

def predict(X_test,X_train,Kernel,Parameters, Gram_Matrix, Y_train,
    n_neighbors=1):

   Nz=len(X_test)
   #Empty list of predictions
   yhat= np.zeros(Nz);
   #number of samples for classification
   #Number of training samples
   Nx=len(X_train);
   #Dummy variable Vector of ones used to get rid of one loop for k(z,z)
   Ones=np.ones(Nx);

   #squared Euclidean distance in kernel space for each training sample
   Distance=np.zeros(Nx)
   # Index of sorted values
   Index= np.zeros(Nx)
```

```
    # calculate pairwise kernels beteen Training samples and prediction
        samples
    Kxz=pairwise_kernels(X_train,X_test,
        metric=Kernel,filter_params=Parameters)

    NaborsNumberAdress=range(0,n_neighbors)

    #Calculate Kernel vector between same vectors Kxx[n]=k(Z[n,:],Z[n,:])

    for n in range(0,Nz):
       # calculate squared Euclidean distance in kernel space for each
           training sample
       #for one prediction
       #for m in range(0,Nx)
       #Distance[m]=|phi(x[m])-phi(z[n])|^2=k(x,x)+k(z,z)-2k(z,x)

       Distance =Gram_Matrix+pairwise_kernels(X_test[n],
           metric=Kernel,filter_params=Parameters)*Ones-2*Kxz[:,n]

       #Distance indexes sorted from smallest to largest
       Index=np.argsort(Distance.flatten());
       Index=Index.astype(int)

       #get the labels of the nearest feature vectors
       yl=list(Y_train[Index[NaborsNumberAdress]])
       #perform majority vote
       yhat[n]=max(set(yl), key=yl.count)
    return(yhat)


if __name__ == "__main__":

    # train_images, train_labels = refineSets(train_images, train_labels,
        1111) # working with smaller subset
    # test_images, test_labels = refineSets(test_images, test_labels,
        111) # working with a smaller subset
    retrain = raw_input("Feature Extraction??")

    if retrain=="y":
       train_images,train_labels = load_mnist('training' )
       test_images,test_labels = load_mnist('testing')
       X_train, Y_train = saveFeatures(train_images,train_labels)
       X_test, Y_test= saveFeatures(test_images, test_labels,save=True,
           filetype="Test")
       Y_train=Y_train.flatten()
       Y_test = Y_test.flatten()
    else:
       with open(FILTER+"Train", 'rb') as fp:
          X_train = load(fp)
       with open("Y_Train", 'rb') as fp:
          Y_train = load(fp)
       with open(FILTER+"Test", 'rb') as fp:
          X_test = load(fp)
       with open("Y_Test", 'rb') as fp:
          Y_test = load(fp)
       Y_train=Y_train.flatten()
       Y_test = Y_test.flatten()

    # hog features extracted
    # i have the hog features, I need to apply SVM the kernel svm and
        other stuff... and nearest neighbor classifier too,
    # i need to do some visualizations.. Need to add confusion matrix
    # Visualization help sought from
        https://github.com/saradhix/mnist_svm/blob/master/plot_mnist_svm.py
```

```
   n_neighbors=4

   for k in ['poly', 'linear','rbf']:
      # we create an instance of Neighbours Classifier and fit the data.
      if k=='linear':
         clf1 = KernelKNNClassifierFit(X_train,Y_train,'linear',0)
         Z1 = predict(X_test,
             X_train,'linear',0,clf1,Y_train,n_neighbors)
         accuracy = np.mean(Z1==Y_test)
         print "linear",accuracy
      elif k=='cosine':
         clf1 = KernelKNNClassifierFit(X_train,Y_train,'cosine',1)
         Z1 = predict(X_test,
             X_train,'cosine',1,clf1,Y_train,n_neighbors)
         accuracy = np.mean(Z1==Y_test)
         print "Cosine",accuracy
      elif k=='rbf':
         clf1 = KernelKNNClassifierFit(X_train,Y_train,'rbf',2)
         Z1 = predict(X_test, X_train,'rbf',2,clf1,Y_train,n_neighbors)
         accuracy = np.mean(Z1==Y_test)
         print "RBF",accuracy
      # I have gramMatrix in clf

# Cosine 0.94500723589
# linear 0.945489628558
# RBF 0.945489628558
```

```
# code for Kernel KNN classifier spam-non spam
from os import listdir
from os.path import isfile, join
import sys
import numpy
import cPickle as pickle
import collections, re
import scipy.sparse
from sklearn import svm
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.lda import LDA
from nltk.stem import PorterStemmer, WordNetLemmatizer
from sklearn.linear_model import Perceptron
from sklearn.feature_extraction.text import TfidfVectorizer
import cv2
import numpy as np
from sklearn import svm, ensemble , tree
from sklearn.metrics import confusion_matrix
import pylab as pl
from cPickle import dump, load
from skimage.feature import hog
import skimage
from matplotlib.colors import ListedColormap
from sklearn.utils import shuffle
import numpy as np
from sklearn import neighbors, datasets
from sklearn.metrics.pairwise import pairwise_kernels
import numpy as np
from sklearn.decomposition import PCA

def createCorpus(data,i, binaryX="False", stopWords=None,
    lemmatize="False", tfidf= "False", useidf="True"): # will vectorize
    BOG using frequency as the parameter and will return the required
    arrays
    X_train =[]
    X_test=[]
    Y_train=[]
    Y_test=[]

    for key in data:
       if key in i:

          for filename in data[key]:
             text = data[key][filename][0]
             if lemmatize == "True":
                port = WordNetLemmatizer()
                text = " ".join([port.lemmatize(k,"v") for k in
                    text.split()])
             X_test.append(text)
             Y_test.append(data[key][filename][1])
       else:
          for filename in data[key]:
             text = data[key][filename][0]
             if lemmatize == "True":
                port = WordNetLemmatizer()
                text = " ".join([port.lemmatize(k,"v") for k in
                    text.split()])
             X_train.append(text)
             Y_train.append(data[key][filename][1])
    if tfidf == "False":
       vectorizer = CountVectorizer(min_df=1, binary= binaryX,
           stop_words=stopWords)
       X_train_ans = vectorizer.fit_transform(X_train)
       X_test_ans = vectorizer.transform(X_test)
       return X_train_ans, Y_train, X_test_ans,Y_test
    elif tfidf == "True":
       vectorizer = TfidfVectorizer(min_df=1, use_idf=useidf)
       X_train_ans = vectorizer.fit_transform(X_train)
       X_test_ans = vectorizer.transform(X_test)

       return X_train_ans, Y_train, X_test_ans,Y_test

# help sought from
    https://github.com/jsantarc/Kernel-Nearest-Neighbor-Algorithm-in-Python-
def KernelKNNClassifierFit(X,Y,Kernel,Parameters):
    Y= numpy.array(Y)
    #Number of training samples
    N=len(X);
    #Array sorting value of kernels k(x,x)
    Gram_Matrix=numpy.zeros(N);
    #Calculate Kernel vector between same vectors Kxx[n]=k(X[n,:],X[n,:])
    #dummy for kernel name
    for n in range(0,N):

      Gram_Matrix[n]=pairwise_kernels(X[n],
          metric=Kernel,filter_params=Parameters)
    return Gram_Matrix

def predict(X_test,X_train,Kernel,Parameters, Gram_Matrix, Y_train,
    n_neighbors=1):
    Y_train=np.array(Y_train)
    Nz=len(X_test)
    #Empty list of predictions
    yhat= numpy.zeros(Nz);
    #number of samples for classification
    #Number of training samples
    Nx=len(X_train);
    #Dummy variable Vector of ones used to get rid of one loop for k(z,z)
    Ones=numpy.ones(Nx);
    #squared Euclidean distance in kernel space for each training sample
    Distance=numpy.zeros(Nx)
    # Index of sorted values
    Index= numpy.zeros(Nx)

    # calculate pairwise kernels beteen Training samples and prediction
        samples
    Kxz=pairwise_kernels(X_train,X_test,
        metric=Kernel,filter_params=Parameters)

    NaborsNumberAdress=range(0,n_neighbors)

    #Calculate Kernel vector between same vectors Kxx[n]=k(Z[n,:],Z[n,:])

    for n in range(0,Nz):
       # calculate squared Euclidean distance in kernel space for each
           training sample
       #for one prediction
       #for m in range(0,Nx)
       #Distance[m]=|phi(x[m])-phi(z[n])|^2=k(x,x)+k(z,z)-2k(z,x)

       Distance =Gram_Matrix+pairwise_kernels(X_test[n],
           metric=Kernel,filter_params=Parameters)*Ones-2*Kxz[:,n]

       #Distance indexes sorted from smallest to largest
       Index=numpy.argsort(Distance.flatten());
       Index=Index.astype(int)

       #get the labels of the nearest feature vectors
       yl=list(Y_train[Index[NaborsNumberAdress]])
       #perform majority vote
       yhat[n]=max(set(yl), key=yl.count)
    return(yhat)

def crossValidation(data,Parameters):
    n_neighbors=4
    accuracy=0        # with frequency
    for i in [0]:
       testSet = [2*i+1,2*i+2]
       X_train, Y_train,X_test,Y_test = createCorpus(data,testSet,
           binaryX="False", stopWords="english", lemmatize="False") #
           with frequency
       X_train= X_train.todense()
       X_test=X_test.todense()
       # print "Fitting"
       num_samples_to_plot = 1000
       X_train, Y_train = shuffle(X_train, Y_train)
       X_train, Y_train = X_train[:num_samples_to_plot],
           Y_train[:num_samples_to_plot]
       pca = PCA(n_components=2)
       X = pca.fit_transform(X_train)

       print X.shape
       y=Y_train


       h = .02 # step size in the mesh
       n_neighbors=4
       # Create color maps
       cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
       cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

       for k in ['cosine', 'linear', 'rbf']:
          # we create an instance of Neighbours Classifier and fit the
              data.
          if k=='linear':
             clf = KernelKNNClassifierFit(X,y,'linear',0)
             clf1 = KernelKNNClassifierFit(X_train,Y_train,'linear',0)
             Z1 = predict(X_test,
                 X_train,'linear',0,clf1,Y_train,n_neighbors)
             accuracy = np.mean(Z1==Y_test)
             print accuracy
          elif k=='cosine':
             clf= KernelKNNClassifierFit(X,y,'cosine',1)
             clf1 = KernelKNNClassifierFit(X_train,Y_train,'cosine',1)
             Z1 = predict(X_test,
                 X_train,'cosine',1,clf1,Y_train,n_neighbors)
             accuracy = np.mean(Z1==Y_test)
             print accuracy
          elif k=='cosine':
             clf= KernelKNNClassifierFit(X,y,'rbf',1)
             clf1 = KernelKNNClassifierFit(X_train,Y_train,'rbf',1)
             Z1 = predict(X_test,
                 X_train,'rbf',1,clf1,Y_train,n_neighbors)
             accuracy = np.mean(Z1==Y_test)
             print accuracy


          # I have gramMatrix in clf


          # Plot the decision boundary. For that, we will assign a color
              to each
          # point in the mesh [x_min, m_max]x[y_min, y_max].
          x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
          y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
          xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                          np.arange(y_min, y_max, h))
          if k=='linear':
             Z=predict(np.c_[xx.ravel(), yy.ravel()], X,'linear',0,
                 clf,y,n_neighbors)
          elif k=='cosine':
             Z=predict(np.c_[xx.ravel(), yy.ravel()],
                 X,'cosine',1,clf,y,n_neighbors)
          elif k=='rbf':
             Z=predict(np.c_[xx.ravel(), yy.ravel()],
                 X,'rbf',1,clf,y,n_neighbors)
          print Z
          # Put the result into a color plot
          Z = Z.reshape(xx.shape)
          plt.figure()
          plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

          # Plot also the training points
          plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold)
          plt.xlim(xx.min(), xx.max())
          plt.ylim(yy.min(), yy.max())

          plt.title("2-Class classification KernelKNN (kernel = "+k
              +")\nAccuracy="+str(accuracy))
          plt.savefig("KNN-SPAM"+k+".png")



 if __name__ == "__main__":
    loadedData=pickle.load( open( "loadedData", "rb" ) )
    # questions = ["3a","3b","3c", "2ab"]
    # questions = ["1a"]
    #for q in questions:
    #  print "---------------------------"
    for C in [1]:
       crossValidation(loadedData,C)
    #  print "--------------------------"

 #RBF: Accuracy 0.886620992173 Gamma 0.1
 # Cosine Accuracy 0.888695925627 Gamma 1
```

```
# code for faceDetection, modified for use from
    https://github.com/bytefish/facerec

import sys, os
sys.path.append("../..")
# import facerec modules
from facerec import *
from facerec.feature import Fisherfaces, SpatialHistogram, Identity
from facerec.distance import EuclideanDistance, ChiSquareDistance
from facerec.classifier import NearestNeighbor
from facerec.classifier import SVM

from facerec.model import PredictableModel
from facerec.validation import KFoldCrossValidation
from facerec.visual import subplot
from facerec.util import minmax_normalize
from facerec.serialization import save_model, load_model
# import numpy, matplotlib and logging
import numpy as np
# try to import the PIL Image module
try:
   from PIL import Image
except ImportError:
   import Image
import matplotlib.cm as cm
import logging
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from facerec.lbp import LPQ, ExtendedLBP
import cv2
from os.path import exists, isdir, basename, join, splitext
from os import makedirs,system
from glob import glob
from random import sample, seed
from scipy import ones, mod, array
from sklearn import svm, ensemble , tree
from sklearn.metrics import confusion_matrix
import pylab as pl
from cPickle import dump, load
from skimage.feature import hog
import skimage
import pickle
from sklearn.cross_validation import KFold
import subprocess

numTrain=10
numClasses=40
trainDir = sys.argv[1]

def get_class(datasetpath, numClasses):
   classes_paths = [files
                 for files in glob(datasetpath.strip() + "/*")
                 if isdir(files)]
   classes_paths.sort()
   classes = [basename(class_path) for class_path in classes_paths]
   classes = classes[:numClasses]
   return classes

def imgfiles(path, extensions):
   all_files = []
   all_files.extend([join(path, basename(fname))
                for fname in glob(path + "/*")
                if splitext(fname)[-1].lower() in extensions])
   return all_files

def readImage(X_fileName):
   feats = []
   for imgfile in X_fileName:
      im = Image.open(imgfile)
      im = im.convert("L")
      feats.append(np.asarray(im, dtype=np.uint8))
   return feats
def all_images(numTotal, dir_path, classes):
   all_images = []
   all_images_class_labels = []
   for i, imageclass in enumerate(classes):
      path = join(dir_path, imageclass)
      extensions = [".pgm"]
      imgs = imgfiles(path, extensions)
      imgs = sample(imgs, numTotal)
      all_images = all_images + imgs
      class_labels = list(i * ones(numTotal))
      all_images_class_labels = all_images_class_labels + class_labels
   all_images_class_labels = array(all_images_class_labels, 'int')
   all_images = readImage(all_images)
   return all_images, all_images_class_labels

if __name__ == "__main__":

   classes = get_class(trainDir, numClasses)
   X , y = all_images(numTrain , trainDir, classes) # images are read
      suitably

   feature = Fisherfaces()
   # Define a 1-NN classifier with Euclidean Distance:
   # classifier = NearestNeighbor(dist_metric=EuclideanDistance(), k=1)
   classifier=NearestNeighbor(dist_metric=EuclideanDistance(), k=1)
   # Define the model as the combination
   my_model = PredictableModel(feature=feature, classifier=classifier)
   # Compute the Fisherfaces on the given data (in X) and labels (in y):
   my_model.compute(X, y)
   # We then save the model, which uses Pythons pickle module:
   dump( my_model, open( "model", "wb" ) )
   with open("model", 'rb') as fp:
      model= load(fp)

   # Then turn the first (at most) 16 eigenvectors into grayscale
   # images (note: eigenvectors are stored by column!)
   E = []
   for i in xrange(min(model.feature.eigenvectors.shape[1], 16)):
      e = model.feature.eigenvectors[:,i].reshape(X[0].shape)
      E.append(minmax_normalize(e,0,255, dtype=np.uint8))
   # Plot them and store the plot to "python_fisherfaces_fisherfaces.pdf"
   subplot(title="Fisherfaces", images=E, rows=4, cols=4,
      sptitle="Fisherface", colormap=cm.jet, filename="fisherfaces.png")
   # Perform a 10-fold cross validation
   cv = KFoldCrossValidation(model, k=10)
   cv.validate(X, y)
```

```
# And print the result:
cv.print_results()
```