Handwritten Digit Recognition Using scikit-learn

**Handwritten Digit Recognition Using scikit-learn**

In this project, I have used scikit-learn library to do machine learning classification on the MNIST database of handwritten digits. I used the following methods:

* K-Nearest Neighbors
* Random Forest
* Linear SVC

The MNIST dataset is a well-known dataset consisting of 28x28 grayscale images. For each image, we know the corresponding digits (from 0 to 9). It is available here: <http://yann.lecun.com/exdb/mnist/index.html>

**Loading the Dataset**

To load the dataset, I used the python-mnist package.

from mnist import MNIST

from PIL import Image, ImageDraw

# Load dataset

mndata = MNIST('./data/')

images, labels = mndata.load\_training()

# Pick the fifth image from the dataset (it's a 9)

i = 4

image, label = images[i], labels[i]

# Print the image

output = Image.new("L", (28, 28))

output.putdata(image)

output.save("output.png")

# Print label

print(label)

RUN

**K-Nearest Neighbors**

The k-nearest neighbors algorithm is one of the simplest algorithms for classification. Let's represent the training data as a set of points in the feature space (e.g. for objects with 2 features, they are points of the 2D plan). Classification is based on the K closest points of the training set to the object we wish to classify. The object is classified by a majority vote of its k-nearest neighbors.

The training phase of the K-Nearest Neighbors algorithm is fast, however the classification phase may be slow due to computation of K distances.

from mnist import MNIST

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy\_score

print("Loading dataset...")

mndata = MNIST("./data/")

images, labels = mndata.load\_training()

clf = KNeighborsClassifier()

# Train on the first 10000 images:

train\_x = images[:10000]

train\_y = labels[:10000]

print("Train model")

clf.fit(train\_x, train\_y)

# Test on the next 100 images:

test\_x = images[10000:10100]

expected = labels[10000:10100].tolist()

print("Compute predictions")

predicted = clf.predict(test\_x)

print("Accuracy: ", accuracy\_score(expected, predicted))

RUN

**Random Forest**

Random forests are an ensemble learning method that can be used for classification. It works by using a multitude of decision trees and it selects the class that is the most often predicted by the trees.

A decision tree contains at each vertex a "question" and each descending edge is an "answer" to that question. The leaves of the tree are the possible outcomes. A decision tree can be built automatically from a training set.

Each tree of the forest is created using a random sample of the original training set, and by considering only a subset of the features (typically the square root of the number of features). The number of trees is controlled by cross-validation.

from mnist import MNIST

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy\_score

print("Loading dataset...")

mndata = MNIST("./data/")

images, labels = mndata.load\_training()

clf = RandomForestClassifier(n\_estimators=100)

# Train on the first 10000 images:

train\_x = images[:10000]

train\_y = labels[:10000]

print("Train model")

clf.fit(train\_x, train\_y)

# Test on the next 1000 images:

test\_x = images[10000:11000]

expected = labels[10000:11000].tolist()

print("Compute predictions")

predicted = clf.predict(test\_x)

print("Accuracy: ", accuracy\_score(expected, predicted))

RUN

**Linear Support Vector Classification.**

In a Support Vector Machine (SVM) model, the dataset is represented as points in space. The space is separated in clusters by several hyperplanes. Each hyperplan tries to maximize the margin between two classes (i.e. the distance to the closest points is maximized).

Scikit-learn provided multiple Support Vector Machine classifier implementations. SVC supports multiple kernel functions (used to split with non-linearly) but the training time complexity is quadradic with the number of samples. Multiclass classification is done with a one-vs-one scheme. On the other hand, LinearSVC only supports linear kernels but the training time is linear with the number of samples. The multiclass classification is done with a one-vs-others scheme.

I used LinearSVC here because it is fast enough, however with smaller training set, the standard SVC could be used.

from mnist import MNIST

from sklearn.svm import LinearSVC

from sklearn.metrics import accuracy\_score

print("Loading dataset...")

mndata = MNIST("./data/")

images, labels = mndata.load\_training()

clf = LinearSVC()

# Train on the first 10000 images:

train\_x = images[:10000]

train\_y = labels[:10000]

print("Train model")

clf.fit(train\_x, train\_y)

# Test on the next 1000 images:

test\_x = images[10000:11000]

expected = labels[10000:11000].tolist()

print("Compute predictions")

predicted = clf.predict(test\_x)

print("Accuracy: ", accuracy\_score(expected, predicted))

RUN

**Conclusion**

In this project, I used 3 different approaches for Supervised Machine Learning classification. It's not possible to say which one is the best to classify this MNIST dataset because that depends on a multitude of criteria and they can be fine-tuned to improve their performance (which I didn't here). The K-nearest neighbors algorithm is fast to train the data but is slow to compute the results. On the other hand, the Random Forest is faster to classify the data. The results obtained with LinearSVC were less good, but this could probably be improved by using better parameters.