MATH637, Spring 2019

Homework 2: MNIST

Due Friday, March 29th, 5:00pm

The MNIST database (Modified National Institute of Standards and Technology database) is one of the most popular datasets in machine learning and data science. Everyone in the field works with MNIST, so we should do that too.

The MNIST database contains 60,000 training images and 10,000 testing images of 28×28 grayscale images of handwritten digits. Each of the sample in the dataset can be regarded as a vector in $\mathbb{R}^{28\times 28}$, and our task is to match the samples with their corresponding labels in $\{0, 1, \ldots, 9\}$ using a machine learning algorithm.

Figure 1: Sample images from MNIST test dataset.

At first, this seems to be a very high-dimensional and difficult problem that cannot be solved easily, at least by the primitive tools we've learnt in the class so far. However, the reality is that classifying digits with MNIST is a problem so clean even simple algorithms can easily achieve 90% prediction accuracy. One intuition is as follows: although the dataset lives in a high-dimensional space, its intrinsic geometric dimension is quite low. If we try to project the dataset toward a two-dimensional plane (constructed by using Principal Component Analysis, a tool that we will learn later in the semester), we will have something like Figure 2. We can see that most of the clusters are separated from each other, and one can imagine that on a bit higher dimension we can solve the classification problem quite easily. Another intuition: it is difficult to be confused between a zero and a one or a seven, regardless of the resolution of the images, so at least some parts of the problem should be straightforward.

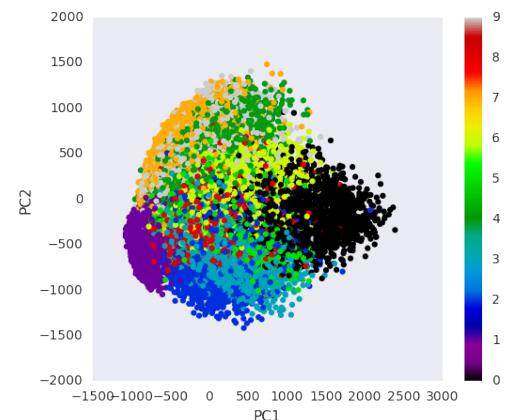

Figure 2: Projection of MNIST on a two-dimensional plane.

The idea of this homework is to play around with the dataset and to make some attempts at solving the classification problem using your preferred algorithm.

1. Read the dataset

   There are many ways to download/import MNIST. One approach is to use *keras.datsets.mnist* in the library *tensorflow*. The detailed codes are as follows:

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist
(X_train, y_train),(X_test, y_test) = mnist.load_data()
```

   The dataset is huge and training with X_{train} (60,000 samples) might be slow. I suggest we take a random subset of size 5000 of the training data

```
import random
I=random.sample(range(60000), 5000)
X_train=X_train[I]
y_train=y_train[I]
```

and start with them first before running on the whole training set.

2. Pre-processing

   It is worth noting that the algorithms implemented in *sklearn* only receive two dimensional arrays as input, i.e., the input should be of dimension (n, d), where n is the number of training observations, and d is the number of features. The loaded arrays X on the previous step are of dimension $(6000, 28, 28)$ (you can see that by looking at *X_train.shape*) because each observation is a 28×28 grayscale image. We need to change that.

   Step 1: Create arrays x_{train}, x_{test} *of dimension* $(6000, 784)$ *from* X_{train}, X_{test} *using the function* ***flatten***.

   The pixels of our image encode the light intensity with values in $\{0, 1, \ldots, 255\}$. To make standard algorithms more stable, a common practice is to normalize the vectors so that the variables only receive values in $[0, 1]$, or [-1, 1]. I recommend doing that here.

   Step 2: Divide x_{train}, x_{test} *by 255 to scale the coordinates to* $[0, 1]$.

3. Classification

   Now we have clean training set and test set. Use an algorithm of your choice to classify the images. If you have no preference, try SVM with rbf kernel:

```
clf=SVC(kernel='rbf', gamma=0.01)
clf.fit(x_train, y_train)
s=clf.predict(x_test)
```

   and use

```
clf.score(x_test, y_test)
```

   to see the prediction accuracy.

   If you want to visualize the results of the prediction, here are some optional sample codes

```
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    j=random.randint(0,10000)
    plt.imshow(X_test[j], cmap=plt.cm.get_cmap("binary"))
    plt.xlabel(s[j])
plt.show()
```

4. Tunable parameters

   Whichever method you used in the previous part comes with (at least) a tunable parameter (in the above example, it is γ).

   Task: Create an array of some sensible values for that parameter and use the prediction score on the test set as a measure of accuracy to pick the optimal value. Provide a plot that visualizes the prediction accuracy across the set of parameter values you chose.