INTRODUCTION TO COMPUTATIONAL PHYSICS

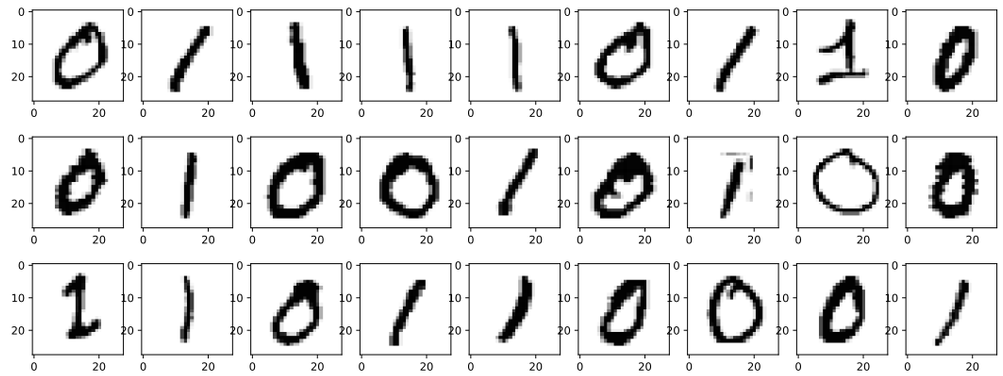
Kai-Feng Chen
National Taiwan University

Let's continue our ML introduction with how to measure the performance of a classifier!

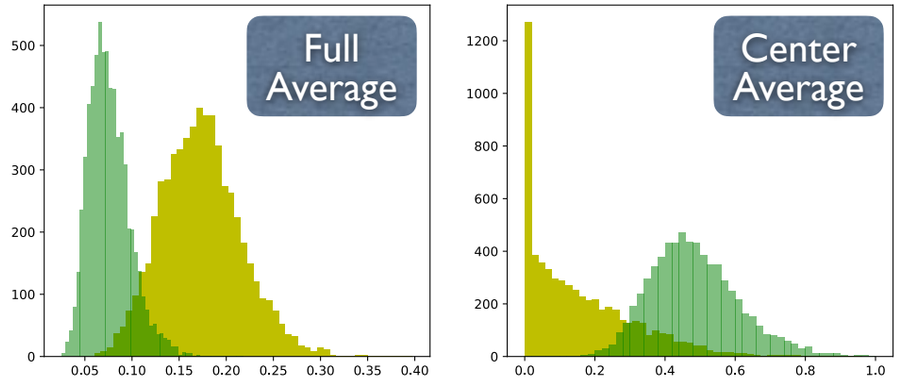
RECAP FROM THE PREVIOUS MODULE

- Here are what we have done so far:

Extract hand-writing 0 and 1 images from MINST data set

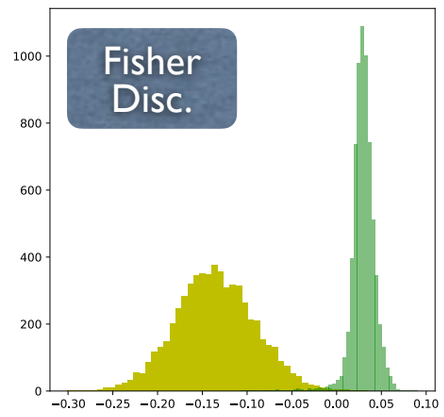


Construct two features: average of full pixels & average of centered pixels



Input the two features to the simplest Linear (Fisher's) discriminant

× –0.9766

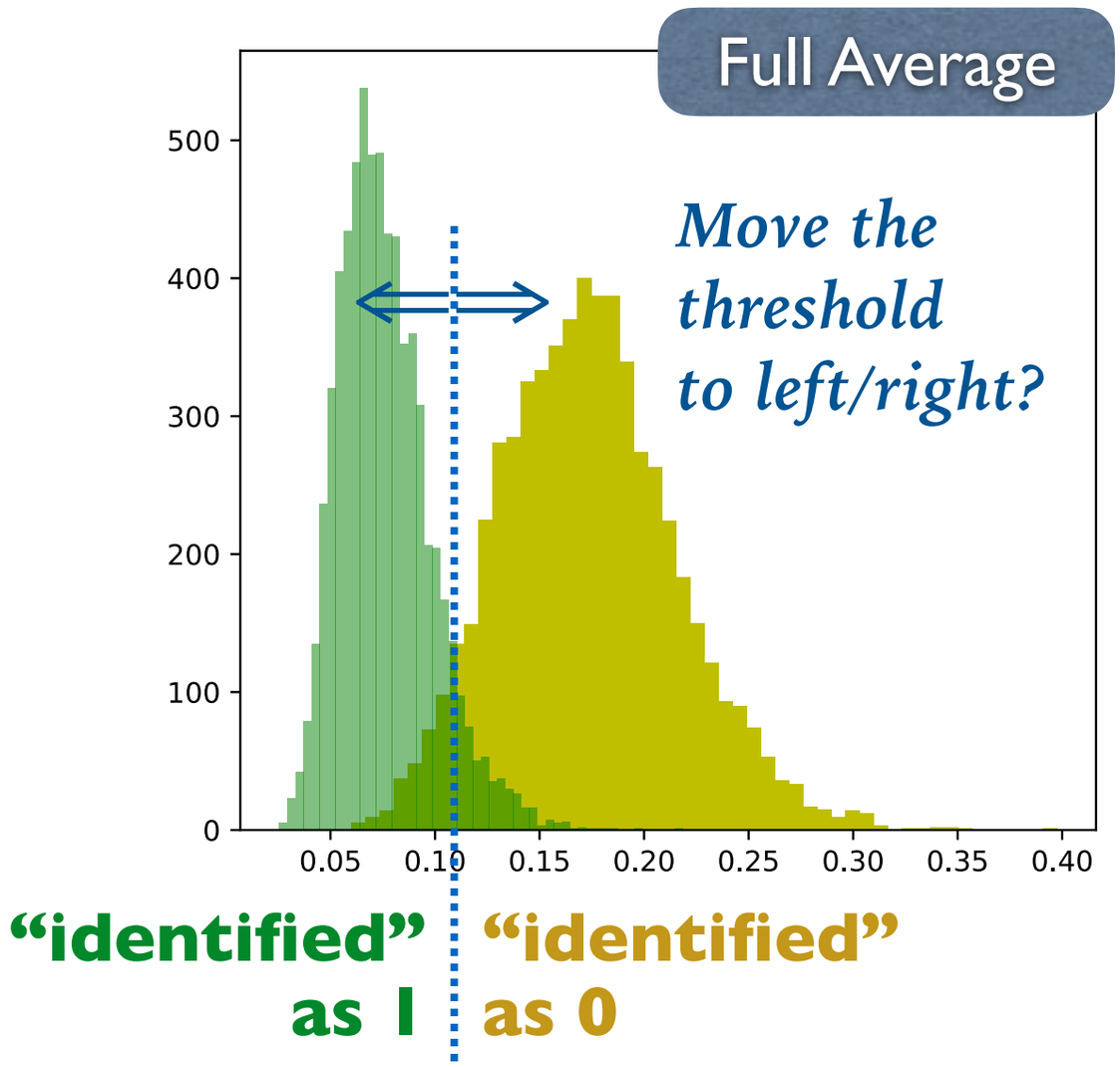


× 0.2150

Better performance found, but how to quantify it?

SEPARATION BETWEEN 0 AND 1

- Remember we have extracted the "features" from the hand-writing digits, e.g. the averages of full pixels.
- How could we benchmark such a feature? We can **introduce a threshold**, and use it to measure the performance.



- If a threshold of **<0.11** is set:
  93.0% of the "ones" are selected;
  94.5% of the "zeros" are rejected.
  (*or **5.5%** of the zeros are misidentified*)
- If a threshold of **<0.16** is set:
  99.8% of the "ones" are selected;
  61.2% of the "zeros" are rejected.
  (*or **38.8%** of the zeros are misidentified*)

The actual performance depends on your selected threshold.

BENCHMARK THE PERFORMANCE

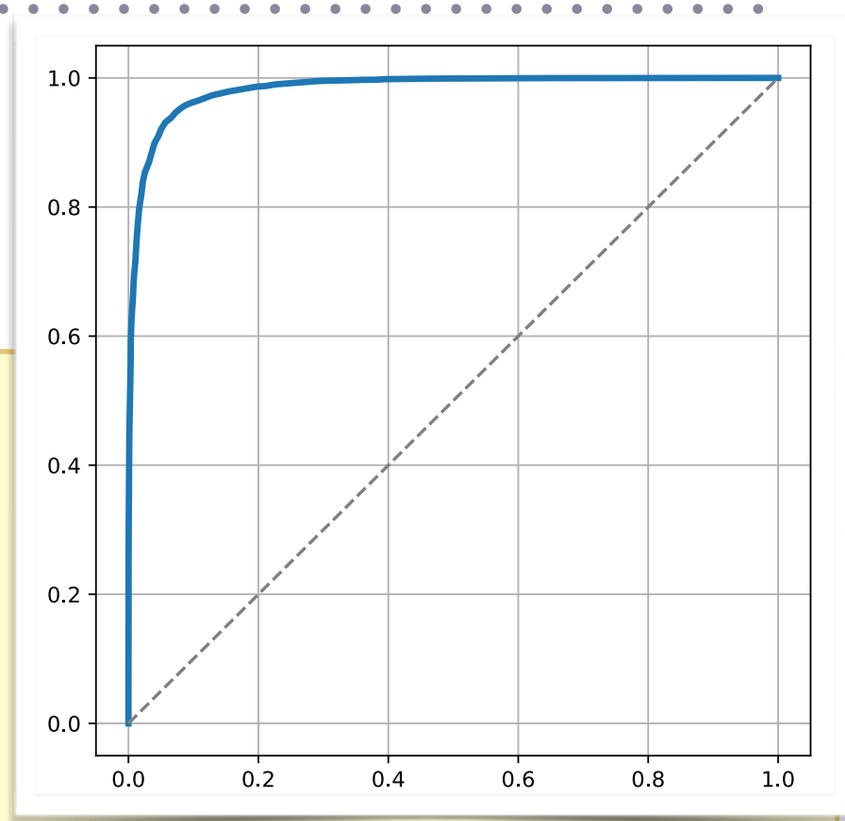
- Let's reformulate the problem as selecting **ones (*as signal*)**, and rejecting **zeros (*as background*)**.
- There is generally no perfect case with 100% efficiency and 0% background contamination. In most of the cases we are dealing with a relatively high signal efficiency but with some background remaining in the end.
- The question is that how could we provide a proper way to benchmark the performance of your variable (and the subsequent ML tools).
- For binary classification, a good way to represent this feature is the **ROC curve** (*receiver operating characteristic curve*). Or you can rank your algorithm simply based on the chance of getting a wrong result!

BENCHMARK THE PERFORMANCE (II)

- Let's produce such a **ROC curve** based on the distribution of full averaged pixel densities:

```
mnist = np.load('mnist.npz')
x_train = mnist['x_train']
y_train = mnist['y_train']

sample0 = x_train[y_train==0]/255.
sample1 = x_train[y_train==1]/255.

all_mean0 = sample0.mean(axis=(1,2))
all_mean1 = sample1.mean(axis=(1,2))

thresholds = np.linspace(0.0,0.4,200)

roc_y = np.array([(all_mean1<th).sum()/len(all_mean1) for th in thresholds])
roc_x = np.array([(all_mean0<th).sum()/len(all_mean0) for th in thresholds])

fig = plt.figure(figsize=(6,6), dpi=80)
plt.plot(roc_x, roc_y, lw=3)
plt.plot([0,1],[0,1], ls='--', c='gray')
plt.grid()
plt.show()
```

Signal efficiency



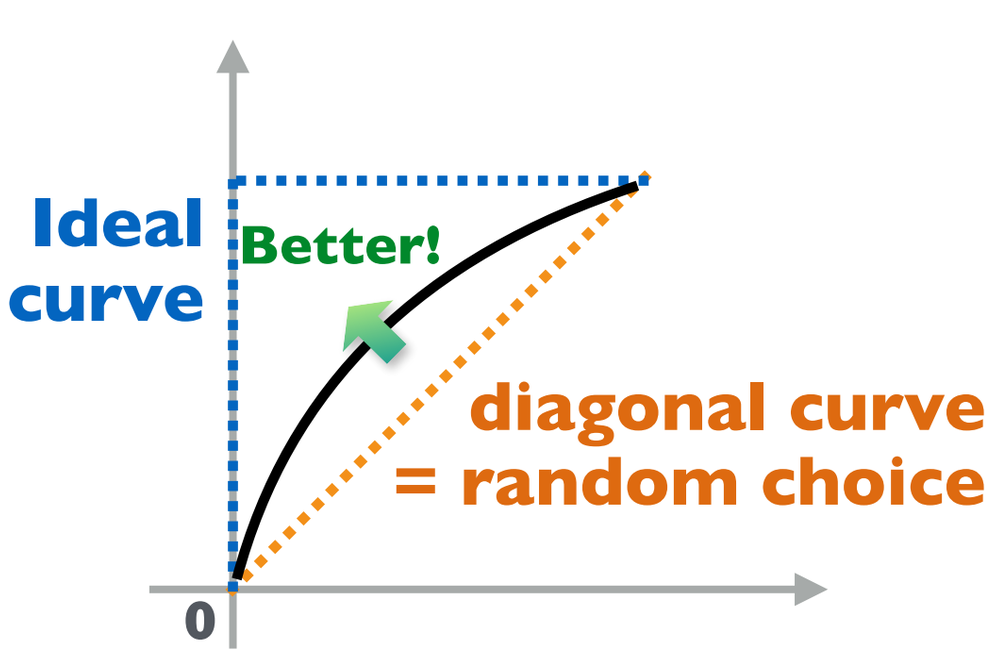
Background contamination

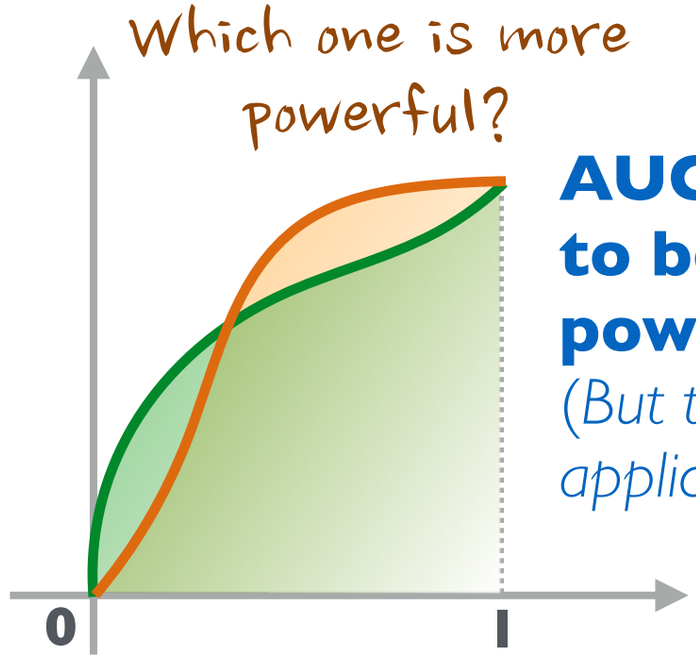
scanning over the thresholds

partial ex_ml2_1.py

ROC AND AUC

- The **ROC curve** illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied.
- When the curve is "banding" away from the diagonal line, it indicates a superior performance; while the ideal curve is yield a point in the upper left corner.
- The performance can be also represented by the **AUC** (area under the curve), which can vary from 0.5 (*as an uninformative classifier*), up to 1.0 (*ideal classifier*).





AUC may be the way to benchmark the power of the classifier!
(But this could be application-dependent!)

ROC AND AUC (II)

- Let's compare the performance of the two feature variables, as well as the combined Fisher's discriminant.

partial ex_ml2_2.py

```
var0 = np.vstack([sample0.mean(axis=(1,2)), \
                  sample0[:,10:18,11:17].mean(axis=(1,2))])
var1 = np.vstack([sample1.mean(axis=(1,2)), \
                  sample1[:,10:18,11:17].mean(axis=(1,2))])
. . . . . .
out0 = (var0.T*weight).sum(axis=1)
out1 = (var1.T*weight).sum(axis=1)

roc1_y = np.array([(var1[0]<th).sum()/len(var1[0]) \
                   for th in np.linspace(0.0,0.4,200)])
roc1_x = np.array([(var0[0]<th).sum()/len(var0[0]) \
                   for th in np.linspace(0.0,0.4,200)])
roc2_y = np.array([(var1[1]>th).sum()/len(var1[1]) \
                   for th in np.linspace(-0.01,1.,200)])
roc2_x = np.array([(var0[1]>th).sum()/len(var0[1]) \
                   for th in np.linspace(-0.01,1.,200)])
roc3_y = np.array([(out1>th).sum()/len(out1) \
                   for th in np.linspace(-0.3,0.1,200)])
roc3_x = np.array([(out0>th).sum()/len(out0) \
                   for th in np.linspace(-0.3,0.1,200)])
```

← for full average

← for center average

← Fisher disc.

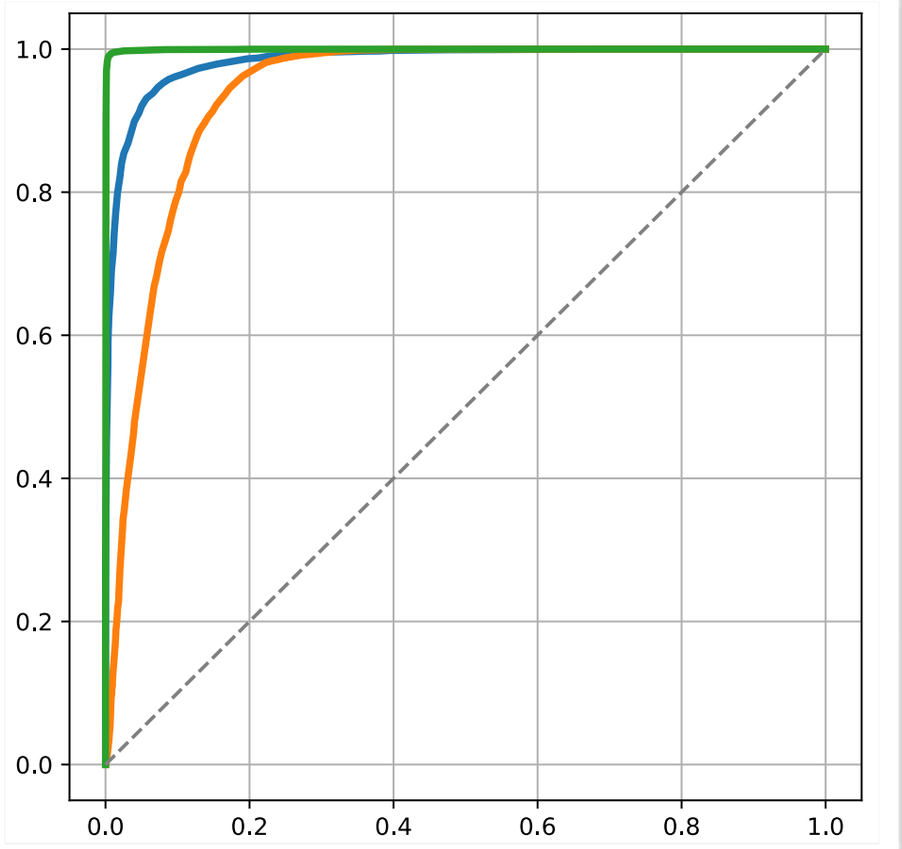
ROC AND AUC (III)

- Now we can compare 3 ROC and calculate 3 AUC:

partial ex_ml2_2.py

```
auc1, auc2, auc3 = 0., 0., 0.
for i in range(200-1):
    h = abs(roc1_x[i+1]-roc1_x[i])
    auc1 += h*(roc1_y[i+1]+roc1_y[i])*0.5
    h = abs(roc2_x[i+1]-roc2_x[i])
    auc2 += h*(roc2_y[i+1]+roc2_y[i])*0.5
    h = abs(roc3_x[i+1]-roc3_x[i])
    auc3 += h*(roc3_y[i+1]+roc3_y[i])*0.5

print('AUC(average of all pixels): ',auc1)
print('AUC(average of centered pixels): ',auc2)
print('AUC(Fisher discriminant): ',auc3)
```

↖ simple trapezoid's rule for area.

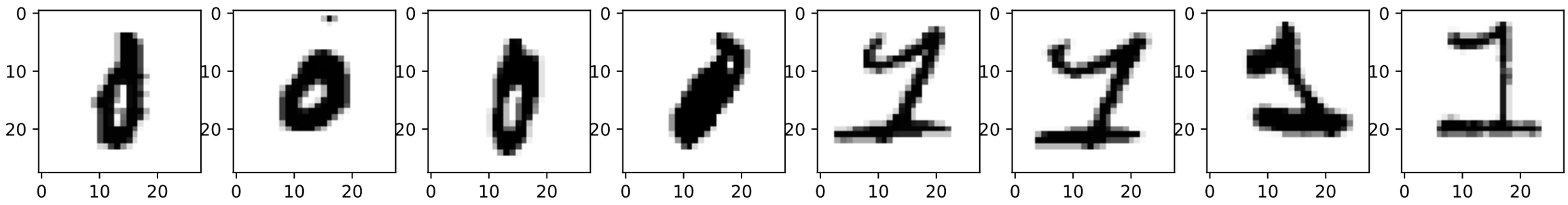



```
AUC(average of all pixels):  0.9835292813693863
AUC(average of centered pixels):  0.9385434043226454
AUC(Fisher discriminant):  0.9991817016088953
```

Performance ranking:
Fisher disc > **average full pixels** > **average center pixels**

COMMENT: PERFORMANCE

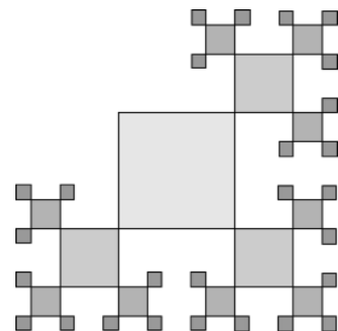
- You may find the result looks quite good and everything seems to be too easy! But this is simply due to the fact that separation of handwriting 0 and 1 is very easy by itself.
- In such a simplified problem we have reached a failure rate of **~0.7% (by setting the threshold at –0.011)**. But remember the best algorithm can reach **0.21%**, and with all 10 digits in the consideration together!
- Here are a couple of failed cases (“zeros” and “ones”):



- Obviously one has to improve the algorithm further…

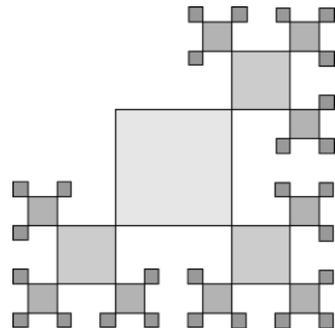
COMMENT: FEATURES

- You may claim, this is due to the fact that we have only include two features/variables! One should invent much more stuff and included them in the classification!
- Yes indeed it would work much better if we can, improve the features, and include more variables in the study. **Including full 768 pixels directly can be also an option** (*we will do that latter in our neural network example*), or **with some ML technique to find the features directly** (*will be discussed in our convolutional network example*).
- However, human designed features have a strong benefit: **we know what we are doing exactly**, although it may not reach its maximum power. In such a situation one can control the systematics (*if it is a worry in your study*) much better.



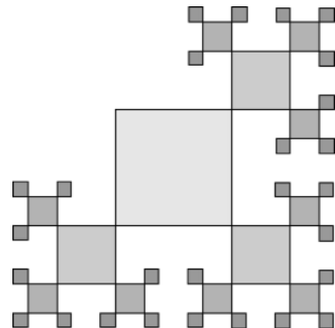
COMMENT: WHAT MACHINE REALLY LEARNED?

- The program we have prepared took only seconds to calculate and give us two weights in the end. **What machine actually learned in this example?**
- Remember the spirit of ML is that we do not tune the algorithm directly; let the algorithm to tune itself from data. So **indeed our discriminant has "learned" two parameters from the input data.**
- If we go for a much more complicated algorithm in the following lecture, there will be much more parameters to tune and you may sense the "learning" part more. **A deep neural network can easily take days or even weeks to train**.



COMMENT: TRAINING AND TESTING

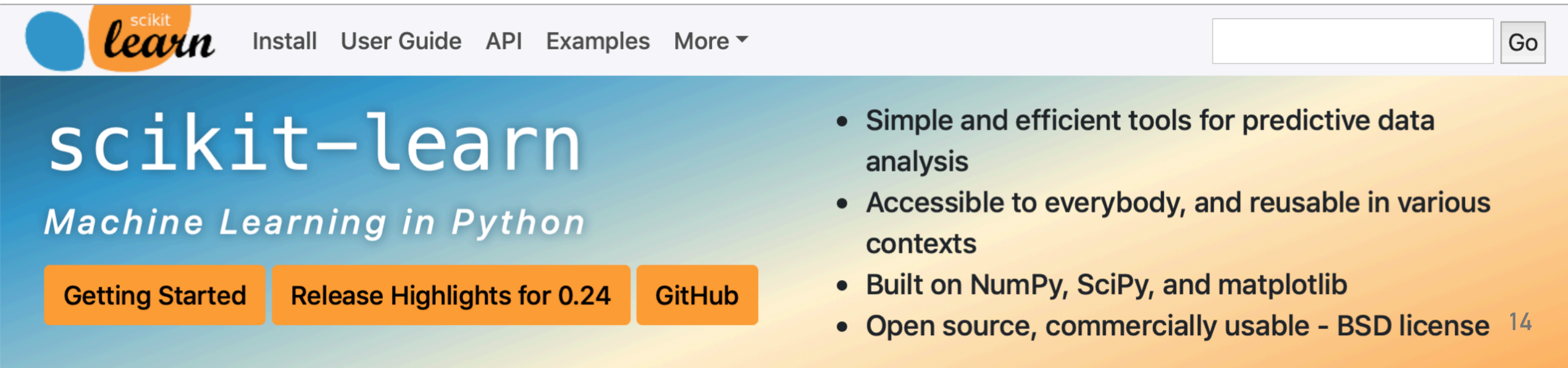
- At the beginning we have said that the typical ML cycle involves training, testing, and maybe another step of validation. And these tasks should be **carried out with statistically independent samples**.
- Indeed this should be carried out properly — as we have estimate the two weights from the training samples, the performance of the discriminant should be determined from the independent testing samples rather than the same training data to **avoid bias**.
- We will ***strictly*** execute these steps from now on. In particular when we move to a more complex algorithm, which will generate a more significant bias by comparing the performance in training and in testing data.



USING THE SCIKIT-LEARN TOOL

- Surely it is more efficient to use some existing tool other than the home made code!
- Here comes the **Scikit-learn**, which is a machine learning library with Python.
- It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, and is designed to interoperate with **NumPy** and **SciPy**.

http://scikit-learn.org/



USING THE SCIKIT-LEARN TOOL (II)

- If you are using **Anaconda** package, it should be already installed (*depending on the version*)! You may check it by:

```
$ conda list scikit-learn
```

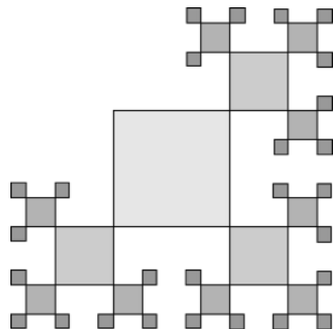
- If you need to install the tool by yourself, it can be carried out by **conda** packager:

```
$ conda install -c conda-forge scikit-learn
```

- or by the **pip** packager:

```
$ pip install -U scikit-learn
```

Or go to https://scikit-learn.org/stable/install.html
for other options / more detailed explanations.



LDA WITH SCIKIT-LEARN

- Let's repeat the simple 2D **Linear Discriminant Analysis** study with the scikit-learn tool:

partial ex_ml2_3.py

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
```
↑ import LDA from scikit-learn

```
mnist = np.load('mnist.npz')
x_train = mnist['x_train'][mnist['y_train']<=1]/255.
y_train = mnist['y_train'][mnist['y_train']<=1]
x_test = mnist['x_test'][mnist['y_test']<=1]/255.
y_test = mnist['y_test'][mnist['y_test']<=1]
```
← Prepare both the training and testing data

```
x_train = np.array([[img.mean(),img[10:18,11:17].mean()] for img in x_train])
x_test = np.array([[img.mean(),img[10:18,11:17].mean()] for img in x_test])

clf = LinearDiscriminantAnalysis()
f_train = clf.fit_transform(x_train, y_train)
```
← "training"

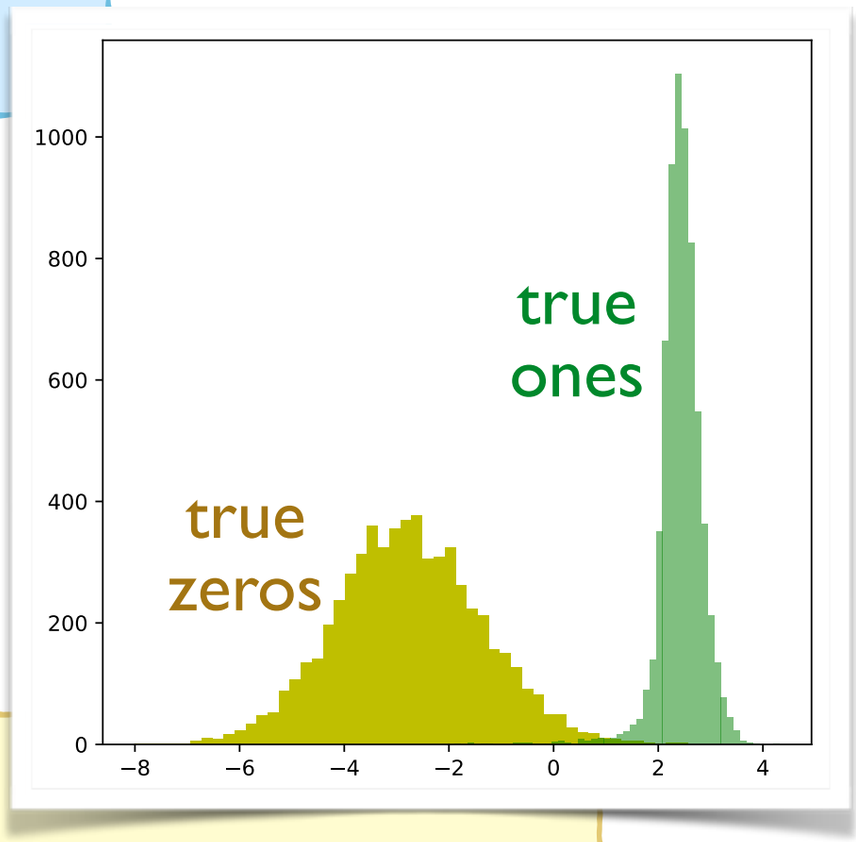
```
s_train = clf.score(x_train, y_train)
s_test = clf.score(x_test, y_test)
print('Performance (training):', s_train)
print('Performance (testing):', s_test)
```
← Evaluate the performance for training and testing data

LDA WITH SCIKIT-LEARN (II)

- The output scores shows a good consistency between training and testing data.

```
Performance (training): 0.9829451243584683
Performance (testing): 0.9867612293144208
```

- And the transformed distribution is pretty much the same as the previous Fisher's discriminant:



partial ex_ml2_3.py

```
fig = plt.figure(figsize=(6,6), dpi=80)
plt.hist(f_train[y_train==0], bins=50, color='y')
plt.hist(f_train[y_train==1], bins=50, color='g', alpha=0.5)
plt.show()
```

DIMENSION REDUCTION WITH LDA

- Another very common way of using LDA is to **reduce the input dimensions**. LDA transforms the input dimensions with linear combination of input features.
- In the following example we take the full 784 pixels from 3 different digits as input and transform them into two dimensions.

partial ex_ml2_4.py

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

mnist = np.load('mnist.npz')
x_train = mnist['x_train'][mnist['y_train']>=7]/255.
y_train = mnist['y_train'][mnist['y_train']>=7]

x_train = np.array([img.reshape((784,)) for img in x_train[:3000]])
y_train = y_train[:3000]
```

← take out 7/8/9 three digits

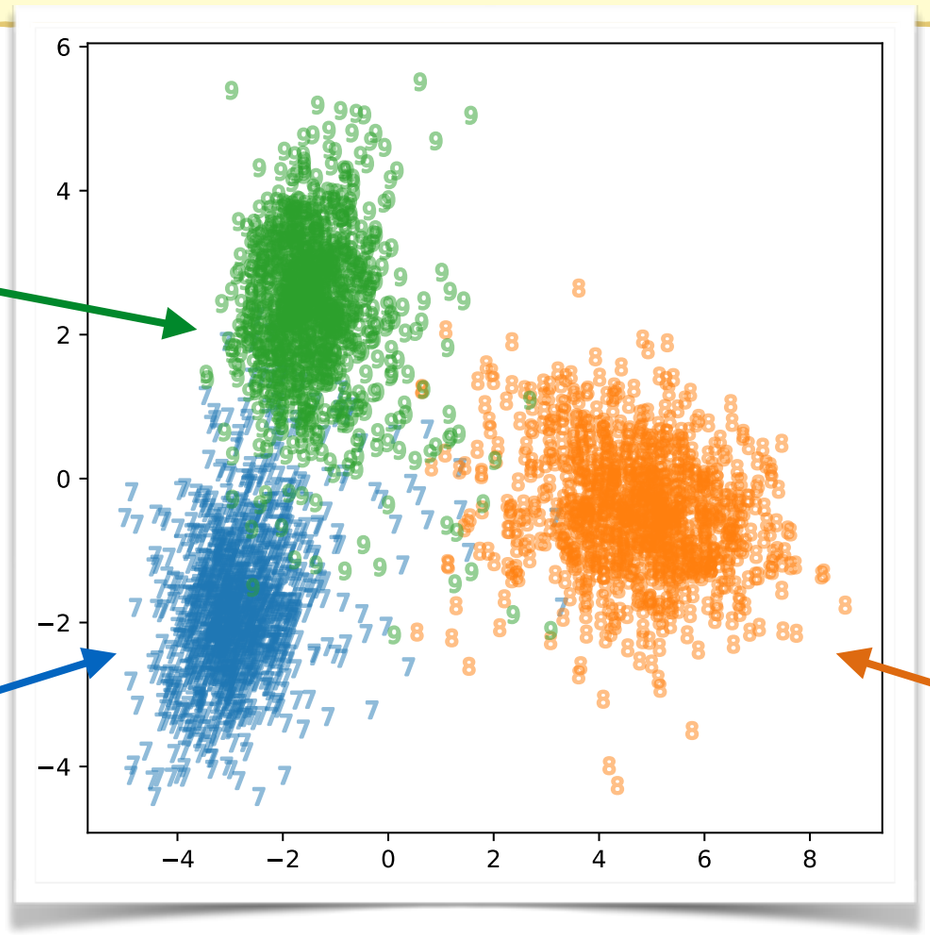
↑ flatten the inputs as 1D array

↑ take the first 3000 samples

DIMENSION REDUCTION WITH LDA (II)

partial ex_ml2_4.py

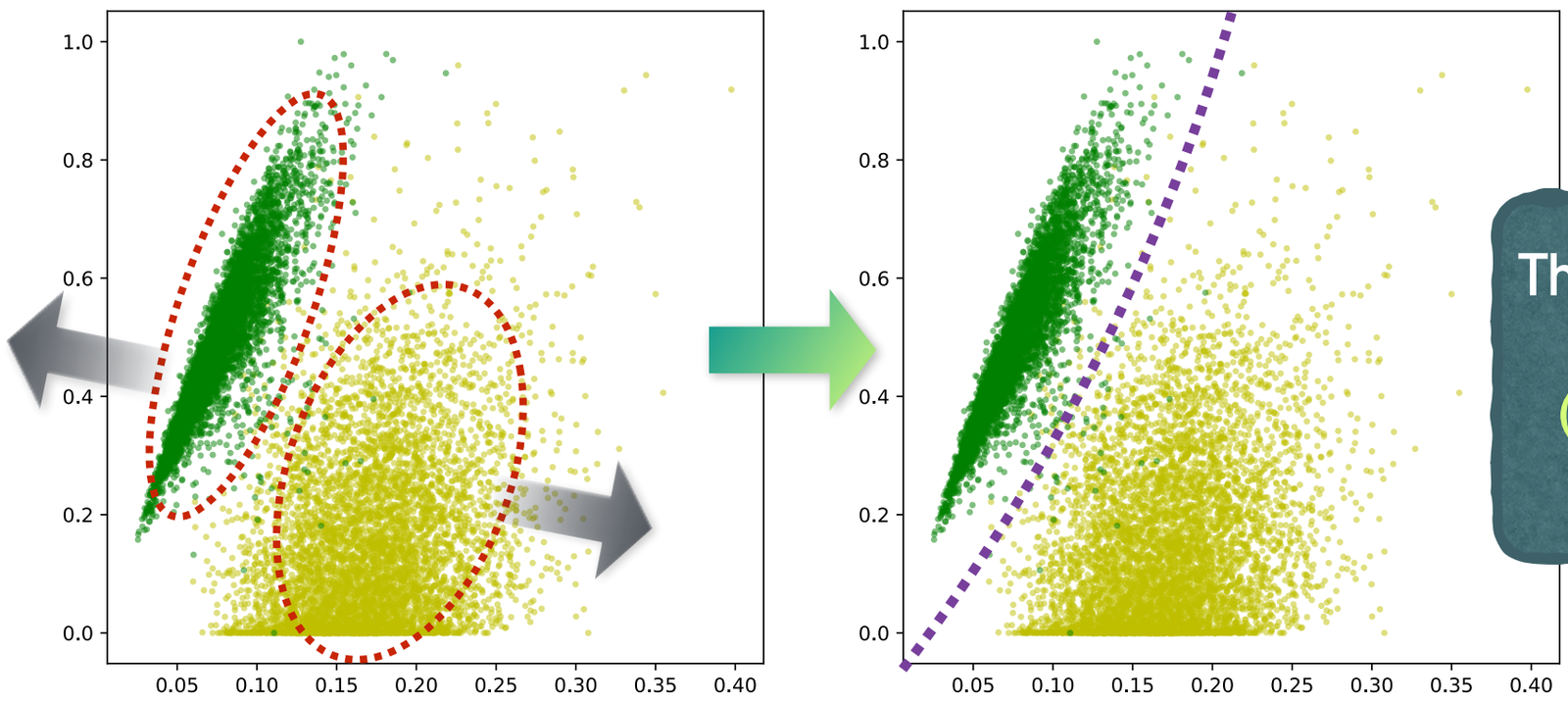
```
clf = LinearDiscriminantAnalysis(n_components=2)
f_train = clf.fit_transform(x_train, y_train)

fig = plt.figure(figsize=(6,6), dpi=80)
for i in range(7,10):
    plt.scatter(f_train[:,0][y_train==i], f_train[:,1][y_train==i],
                s=50, marker='$'+str(i)+'$', alpha=0.5)
plt.show()
```



- Now each of the digits can be described by two variables, and you can see they are quite “distinguishable”!

COMING BACK TO THE ORIGINAL PROBLEM...

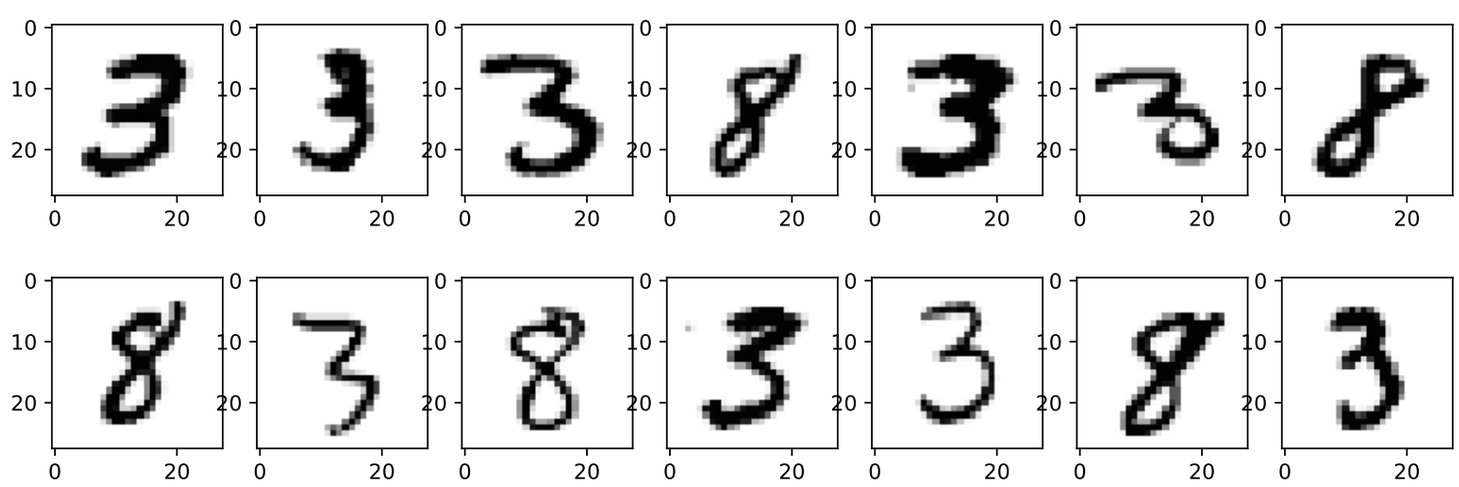
- The LDA is separating the distributions by maximizing the distance between the classes with their mean and covariance in the consideration, as a group-wise effort.
- But since we are discussing about "classification" here, why we cannot just find a **border line** between the groups? One can even consider a non-linear border, right?



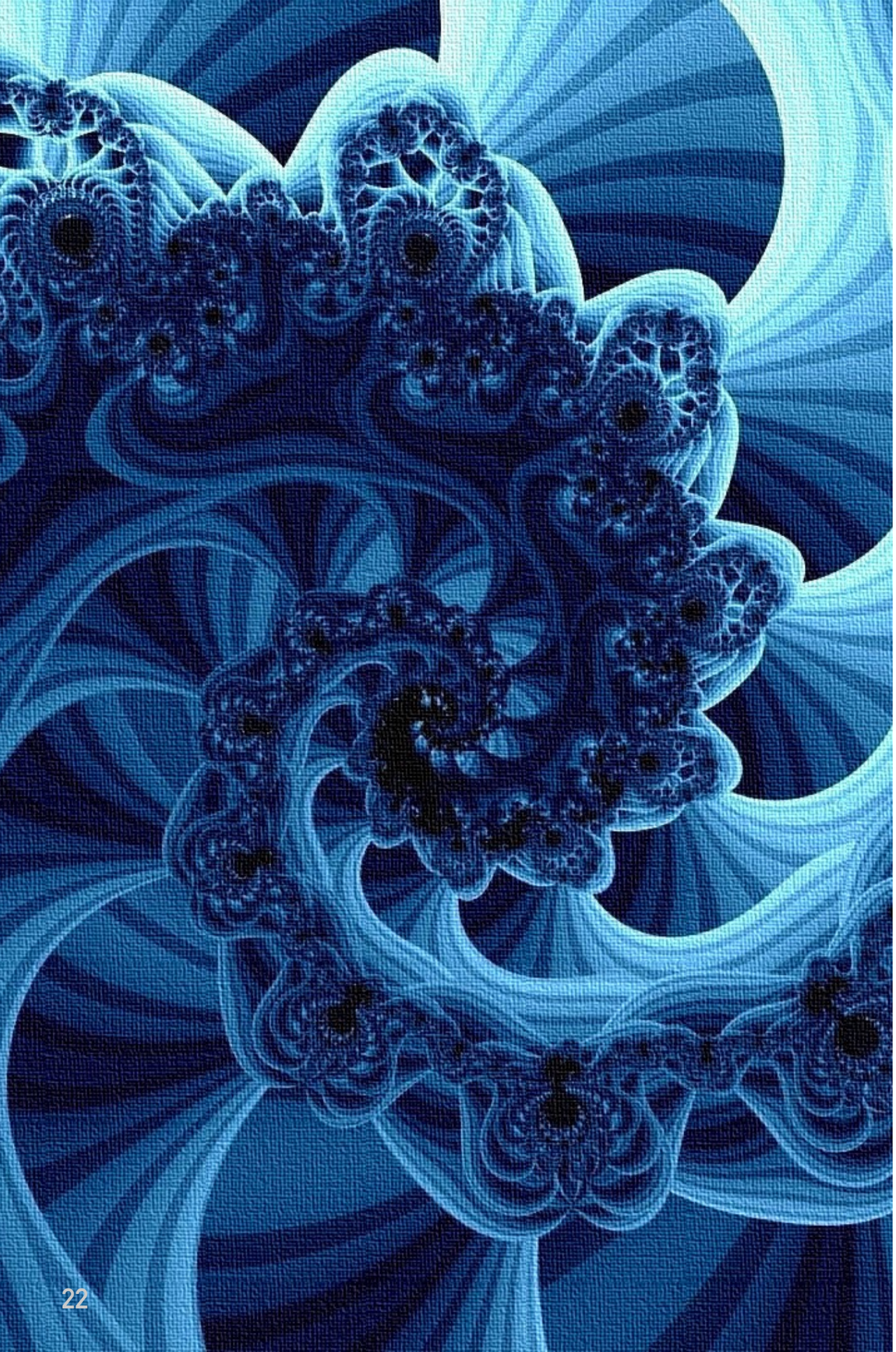
This is exactly the idea of the Support Vector Machine (SVM), to be introduced in the next module!

JUST TRY IT OUT!

- As we just stated, separating 0 and 1 is probably the easiest case. Some other cases it may not be so straightforward. For example, comparing 3 and 8:



- By comparing the average pixel density for these two digits, does it provide some separation power?
- If not, can you think of some simple feature to separate them?



MODULE SUMMARY

- In this module we continued to introduce the basis of machine learning, in particular how the performance of a binary classifier being measured in general. Afterwards we introduce the Scikit-learn package, and the LDA tool within.
- Next module we will introduce another poplar ML algorithm, the Support Vector Machine.