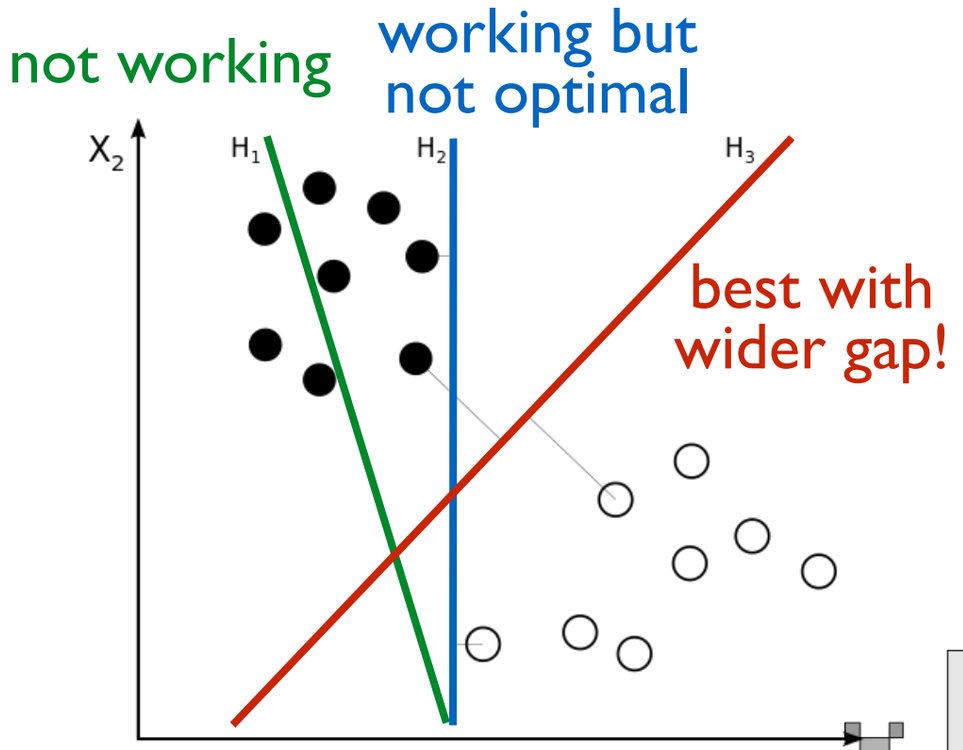
INTRODUCTION TO COMPUTATIONAL PHYSICS

Kai-Feng Chen
National Taiwan University

What's the best way to find a **LINE** to separate the black's and white's?

THE SUPPORT VECTOR MACHINE

- **Support vector machines (SVM)** are supervised learning models commonly used for classification and regression analysis.
- A data point can be viewed as a p-dimensional vector, and one wants to separate the points with a (p–1)-dimensional hyperplane. There are multiple hyperplanes that might classify the data; one reasonable choice is the hyperplane that represents the largest separation, or margin, between the given two classes.
- That is, in the SVM, the categories / classes are divided by **a clear gap which is as wide as possible**.
- So usually it works good for the cases that are difficult to separate!



START WITH LINEAR SVM

- Consider a training data set of ***n*** points (***vectors***):

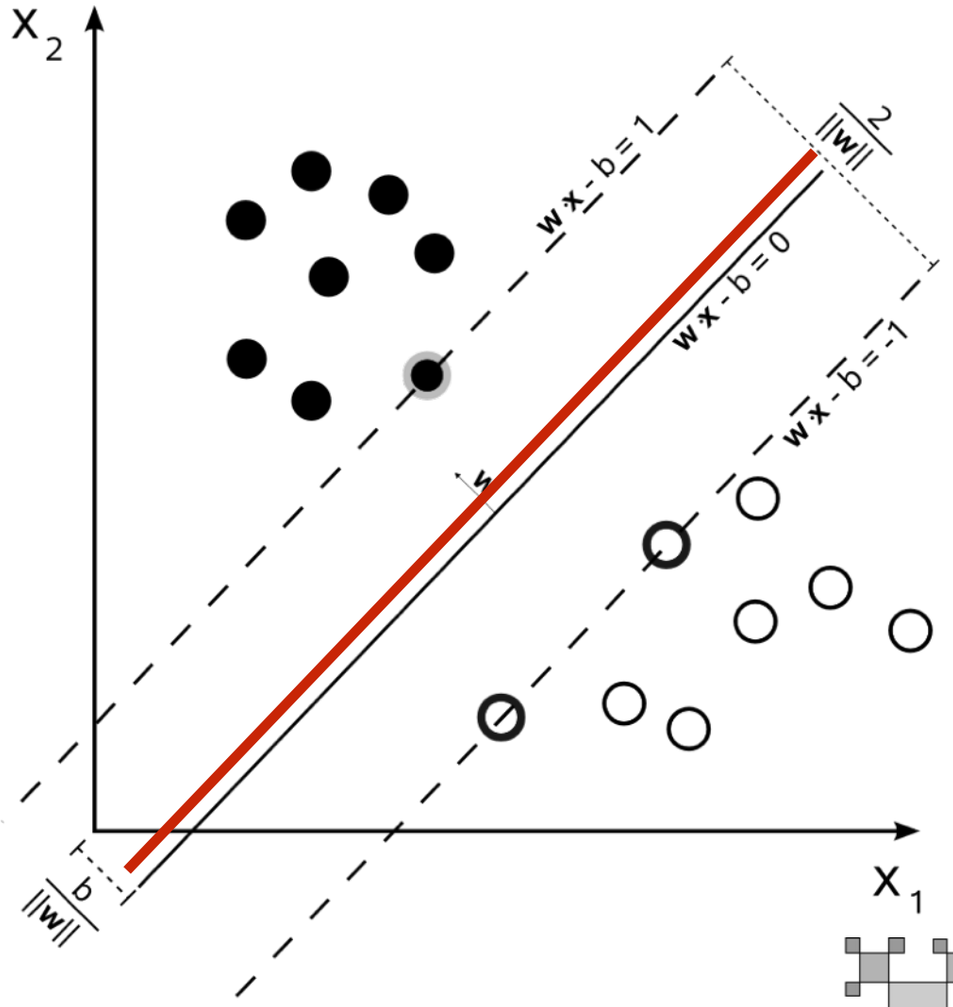
$$(\overrightarrow{x}_1, y_1), ..., (\overrightarrow{x}_n, y_n) \quad \text{where } y_i = \pm 1$$

We want to find the "maximum-margin hyperplane" to separate the groups of ***y*=+1** and **–1**.

- A hyperplane can be expressed as

$$\overrightarrow{w} \cdot \overrightarrow{x} - b = 0$$

where w is the normal vector to the hyperplane, and the parameter $b/|w|$ determines the offset of the hyperplane from the origin.



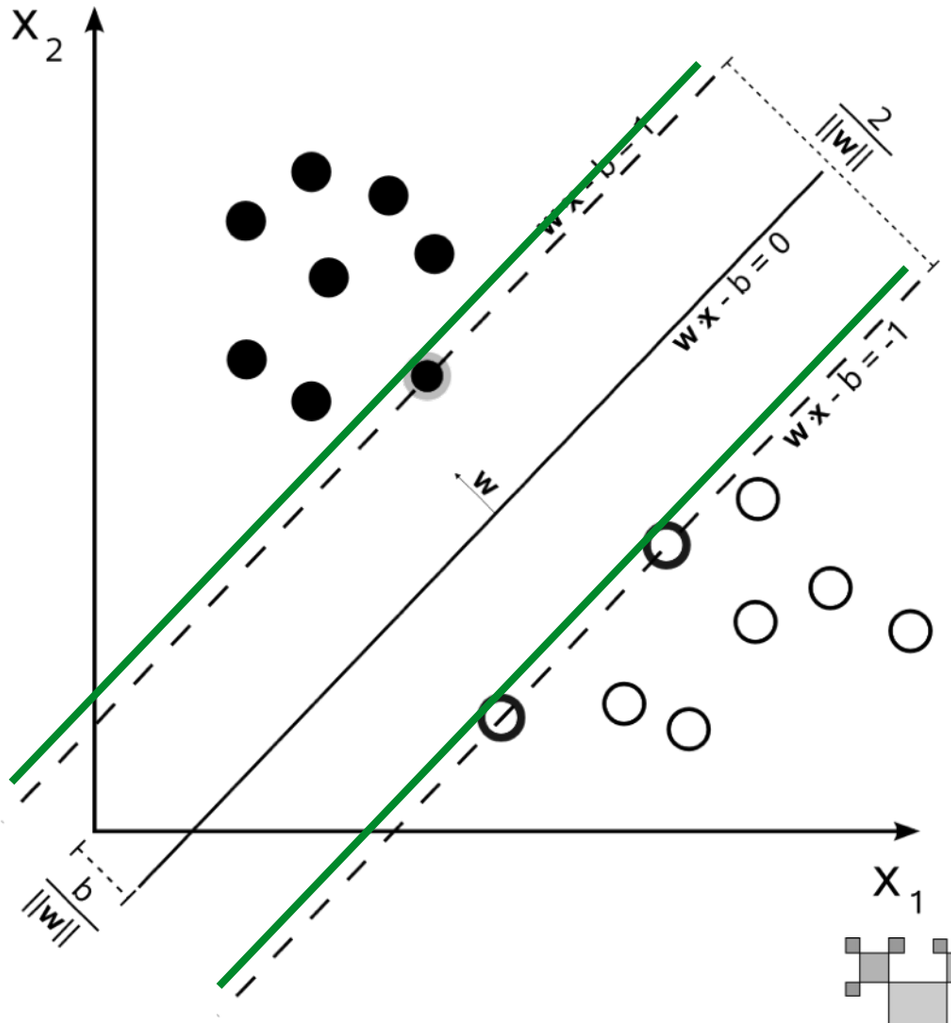
LINEAR SVM WITH HARD MARGIN

- If the training data is linearly separable, we can select two parallel hyperplanes with maximal distance/region between them (*maximal "margin"*).
- These hyperplanes can be described by the following equations:

$$\overrightarrow{w} \cdot \overrightarrow{x} - b = \pm 1$$

- We have to prevent data points from falling into the margin, thus the following constraints apply:

$$\overrightarrow{w} \cdot \overrightarrow{x}_i - b \geq +1, \quad \text{if } y_i = +1$$
$$\overrightarrow{w} \cdot \overrightarrow{x}_i - b \leq -1, \quad \text{if } y_i = -1$$



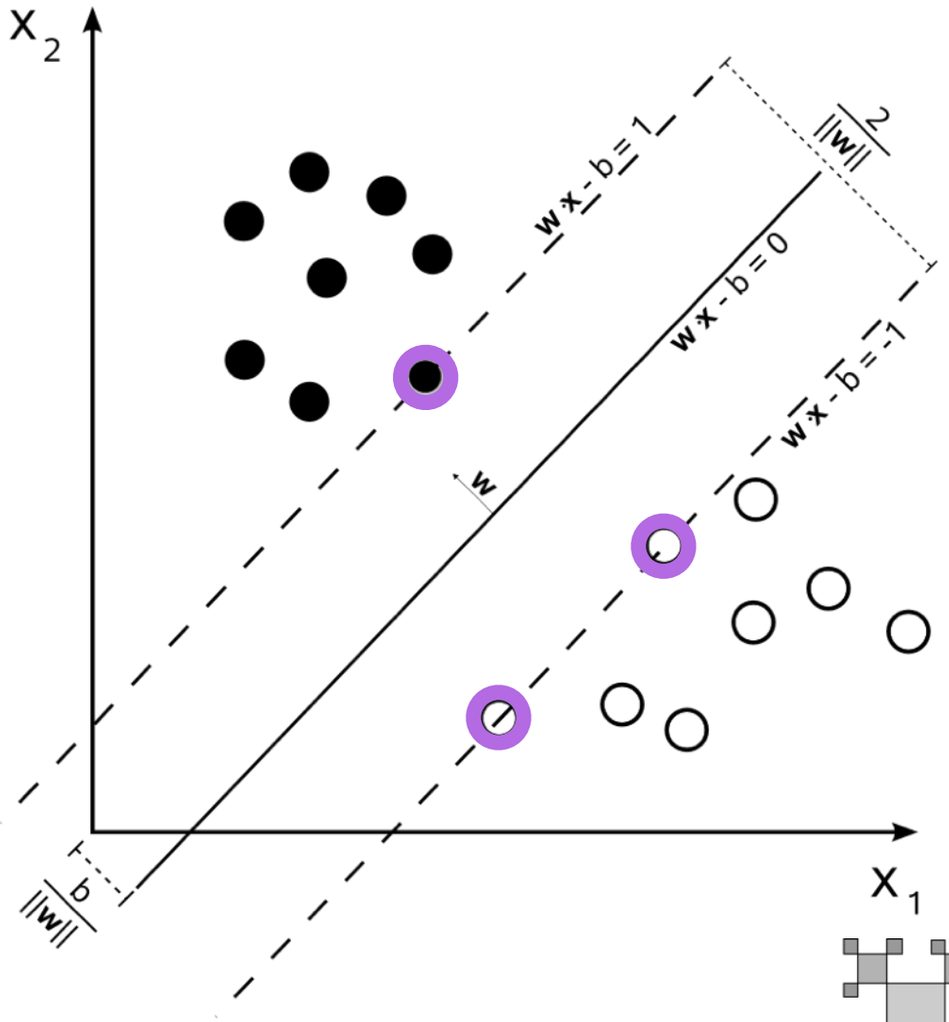
LINEAR SVM WITH HARD MARGIN (II)

- The constraints imply each data point must lie on the correct side of the margin. One can put this together to formulate an optimization problem:

Minimize $\frac{1}{2}|w|^2$ subject to

$$y_i(\overrightarrow{w} \cdot \overrightarrow{x}_i - b) \geq 1$$

for all $1 \leq i \leq n$

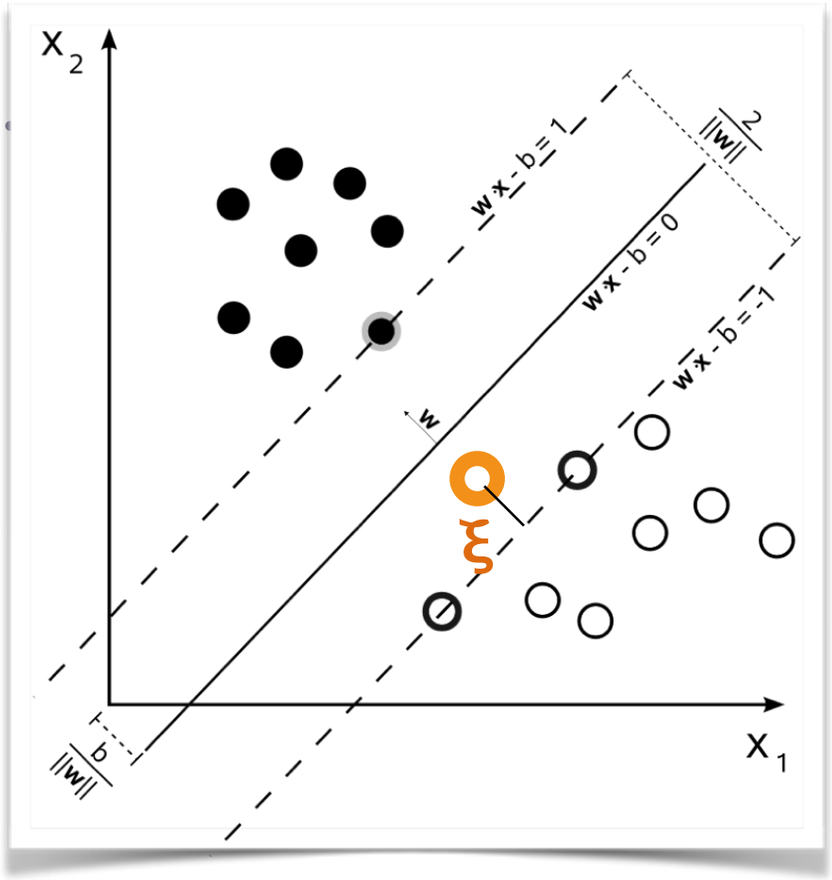


- A consequence of this geometric description is that the max-margin hyperplane is completely determined by those data points which lie nearest to it ⇒ **support vectors**.

SOFT MARGIN

- SVM can be extend to the cases where the data are not fully linearly separable. In order to deal with such a situation, one can introduce "slack variables" (ξ):

$$\overrightarrow{w} \cdot \overrightarrow{x}_i - b \geq +1 - \xi_i, \quad \text{if } y_i = +1$$
$$\overrightarrow{w} \cdot \overrightarrow{x}_i - b \leq -1 + \xi_i, \quad \text{if } y_i = -1$$



- Surely we want the error term to be as small as possible, hence one can add an additional cost to the target function to be minimized:

The regularization parameter C is a balance between the error term and the margin space.

$$\text{Minimize } \frac{1}{2}|w|^2 + C\sum_i \xi_i \text{ subject to}$$
$$y_i(\overrightarrow{w} \cdot \overrightarrow{x}_i - b) + \xi_i \geq 1$$
$$\text{for all } 1 \leq i \leq n \text{ and } \xi \geq 0$$

USING SVM WITH SCIKIT-LEARN

- We will not spend time to explain how to really solve or optimize the SVM (*but you are encouraged to try it out by yourself*). Instead we will use scikit-learn package directly to demonstrate how to use it.
- Let's deploy our handwriting **ones versus zeros** example again:

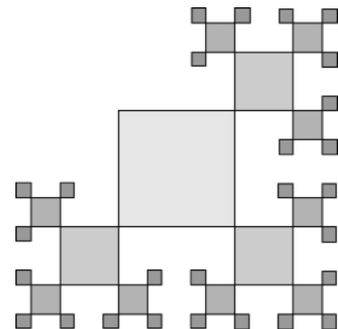
partial ex_ml3_1.py

```
import numpy as np
from sklearn import svm ← just import it!

. . . . . ← data preparation part is the same

clf = svm.SVC(kernel='linear', C=1.0)
clf.fit(x_train, y_train)

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)
```

← initial a SVM w/ linear kernel (take C = 1 for now)

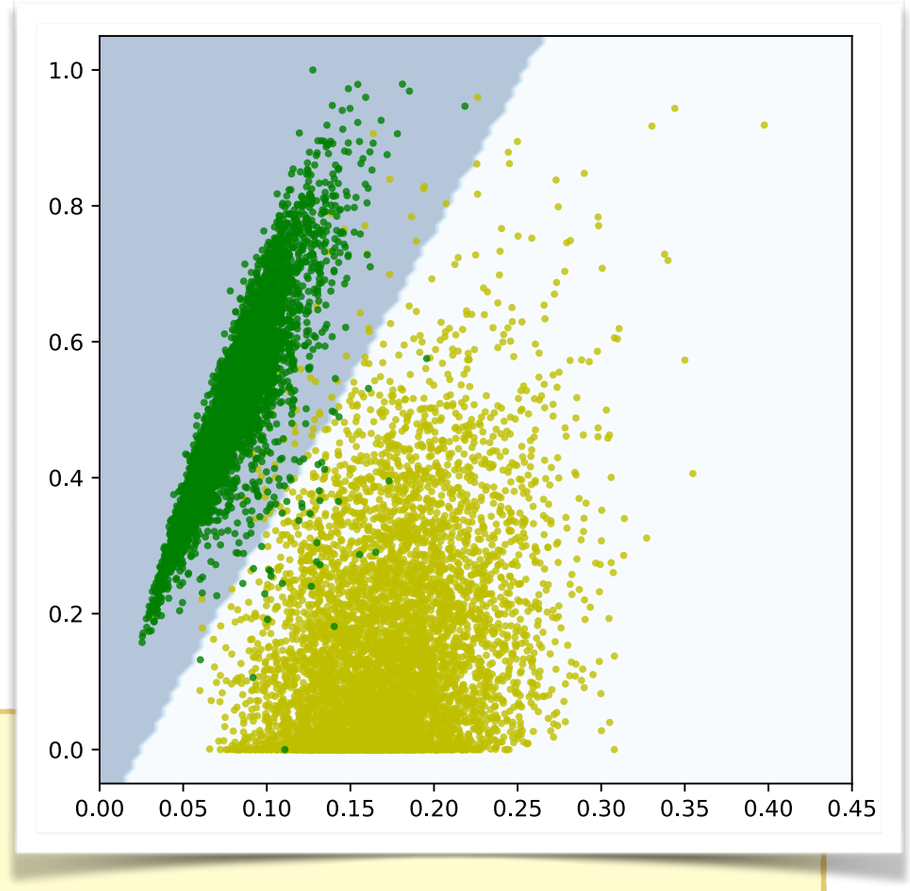
```
Performance (training):
0.9925779707856297
Performance (testing):
0.9947990543735225
```

Slightly better results?



USING SVM WITH SCIKIT-LEARN (II)

- Let's demonstrate the separation power of SVM directly with a **"border line"** between the data points:



partial ex_ml3_1a.py

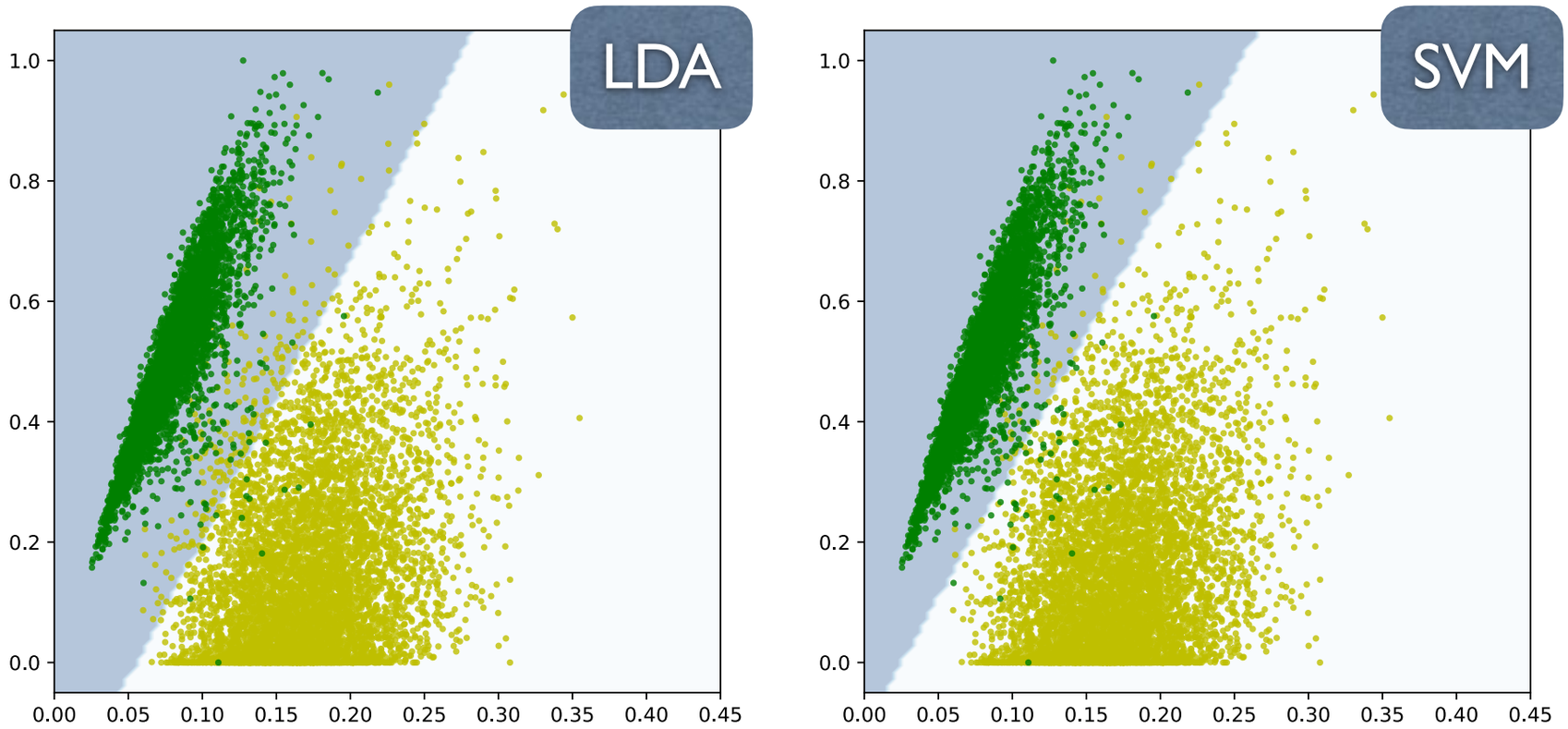
```
clf = svm.SVC(kernel='linear', C=1.0)
clf.fit(x_train, y_train)

fig = plt.figure(figsize=(6,6), dpi=80)
```

↓ use contour to show the borders!

```
xv, yv = np.meshgrid(np.linspace(0.,0.45,100),\
                     np.linspace(-0.05,1.05,100))
zv = clf.predict(np.c_[xv.ravel(), yv.ravel()])
plt.contourf(xv, yv, zv.reshape(xv.shape), . . . . . .)

plt.scatter(x_train[:,0][y_train==0], x_train[:,1][y_train==0], \
            c = 'y', s=5, alpha=0.8)
plt.scatter(x_train[:,0][y_train==1], x_train[:,1][y_train==1], \
            c = 'g', s=5, alpha=0.8)
plt.show()
```

LINEAR SVM VERSUS LDA

- If we also draw a border line based on our previous LDA study, it would look like this (and sufficiently different from the situation for SVM?):



Remember: **LDA** tends to make the average distribution away from each other, while **SVM** concerns more about the difficult data points near boundaries (as the supporting vectors!).

BEFORE MOVING AHEAD...

- Before moving toward the next topic, let's try to **inject all of the pixels directly into linear SVM** and see how good we can separate all of the handwriting digits at once.
- Remark: a full, seriously tuned SVM can reach a superior performance with an error rate <1.5% on MNIST data. But it may take days to run/tune the code. Here we are going to give you a simple setting, which shows you how to get a "starting point".

partial ex_ml3_2.py

```
mnist = np.load('mnist.npz')
x_train = mnist['x_train'][:10000]/255.
y_train = mnist['y_train'][:10000]
x_test = mnist['x_test']/255.
y_test = mnist['y_test']

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

← take only 10K images to speed up the training

← input all 10 digits as 10 classes

↑ flatten the inputs as 1D array

FULL DIGITS SEPARATION WITH LINEAR SVM

- With Scikit-learn package everything is rather straightforward!

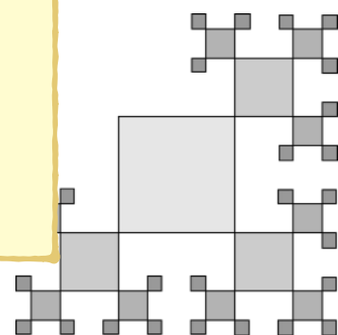
partial ex_ml3_2.py

```
clf = svm.SVC(kernel='linear', verbose=True)
clf.fit(x_train, y_train)
```
← this training will take a while!

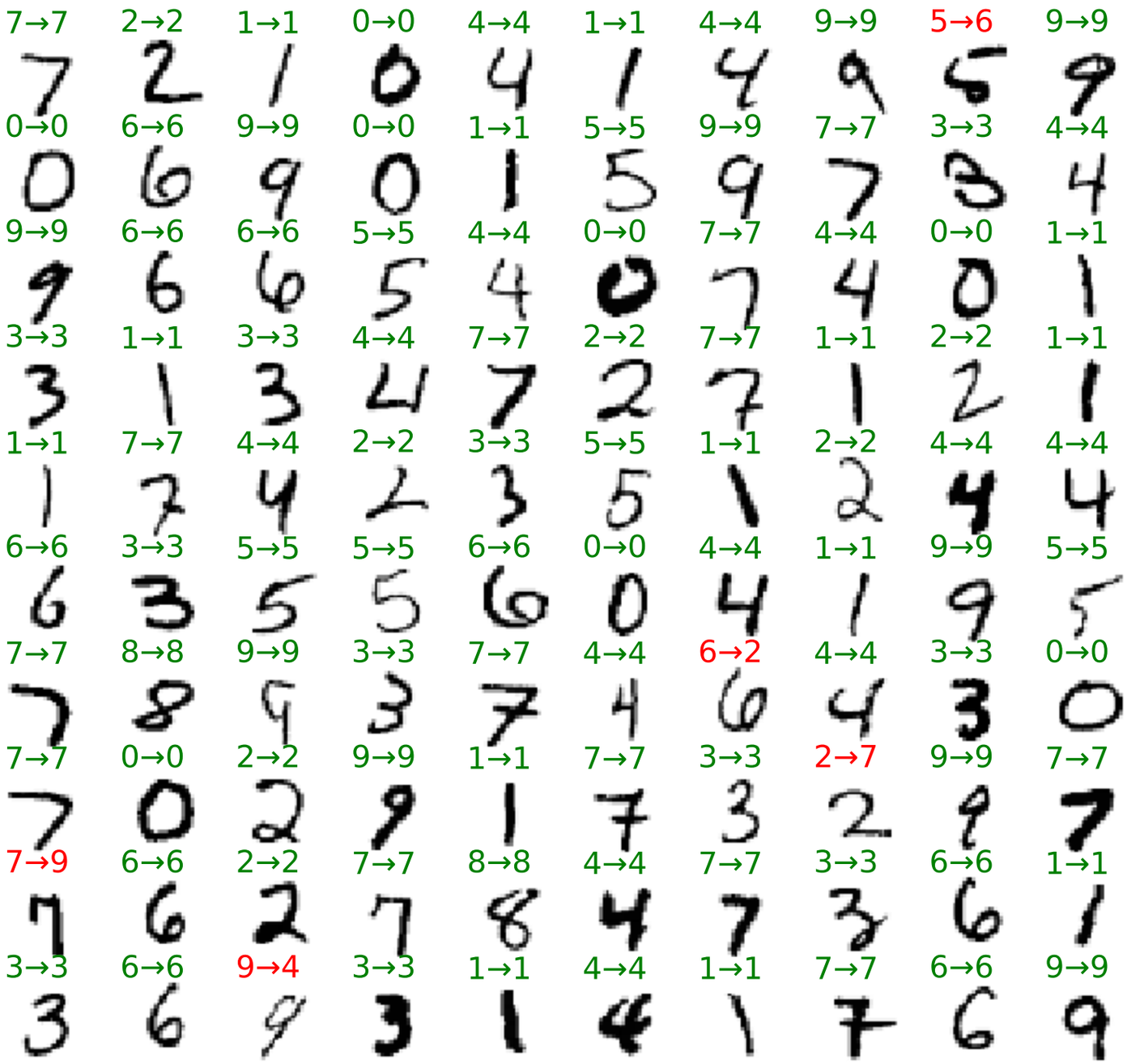
```
s_train = clf.score(x_train, y_train)
s_test = clf.score(x_test, y_test)
print('Performance (training):', s_train)
print('Performance (testing):', s_test)

p_test = clf.predict(x_test)

fig = plt.figure(figsize=(10,10), dpi=80)
for i in range(100):
    plt.subplot(10,10,i+1)
    plt.axis('off')
    plt.imshow(mnist['x_test'][i], cmap='Greys')
    c='Green'
    if y_test[i]!=p_test[i]: c='Red'
    plt.text(0.,0.,'$%d\\to%d$' % \
        (y_test[i],p_test[i]),color=c,fontsize=15)
plt.show()
```
← mark as red if there is mis-tag.

↑ show the first 100 digits images

```
optimization finished,
#iter = 2864
obj = -10.419231, rho =
1.347649
nSV = 133, nBSV = 0
Total nSV = 2630
Performance (training):
0.9969
Performance (testing):
0.917
```
↖ an error rate of ~8.3%, still room for improvement!



FULL DIGITS SEPARATION WITH LINEAR SVM (II)



- Only several misidentifications found in the first 100 digits!
- You may find the training accuracy of **99.7%** and testing accuracy of **91.7%**; such situation is a typical **overfitting/ overtraining**.
- We will discuss more about such symptom very soon.

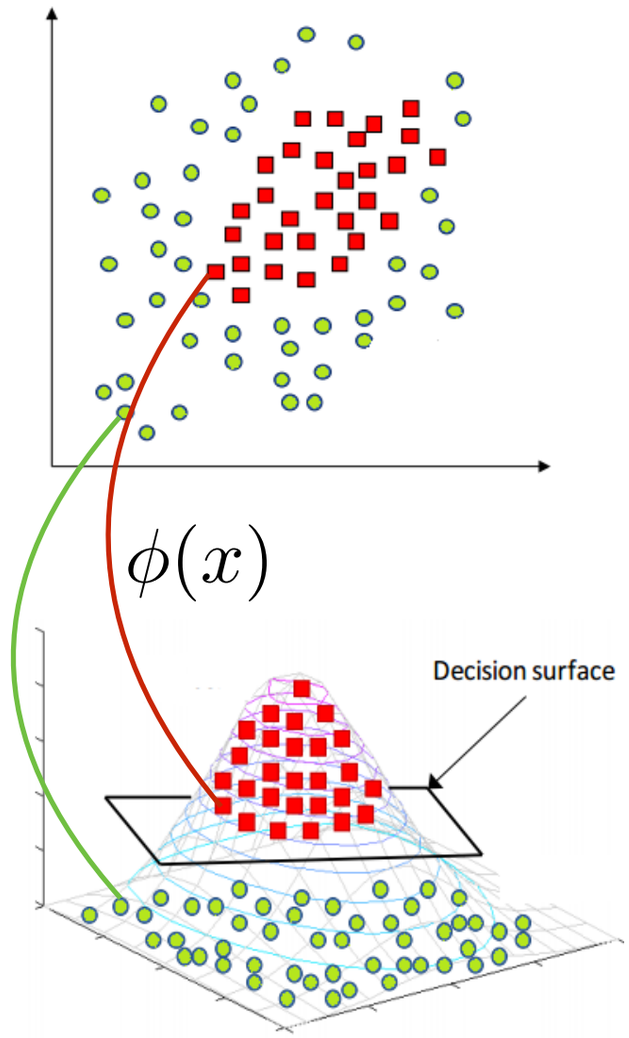
HOW ABOUT NONLINEAR KERNEL?

- Can we quickly improve our tool with a non-linear method, for example, non-linear SVM? (*Sounds more powerful at least!*)
- The idea is to transform the data with a **kernel trick**, and allows the algorithm to fit the margin hyperplane in a **transformed feature space**. The classifier finds a hyperplane in the transformed space, the plane can be be nonlinear in the original space. Some common kernels:
  - *Polynomial*

$$k(\vec{x}_i, \vec{x}_j) = (\gamma \vec{x}_i \cdot \vec{x}_j + \eta)^d$$

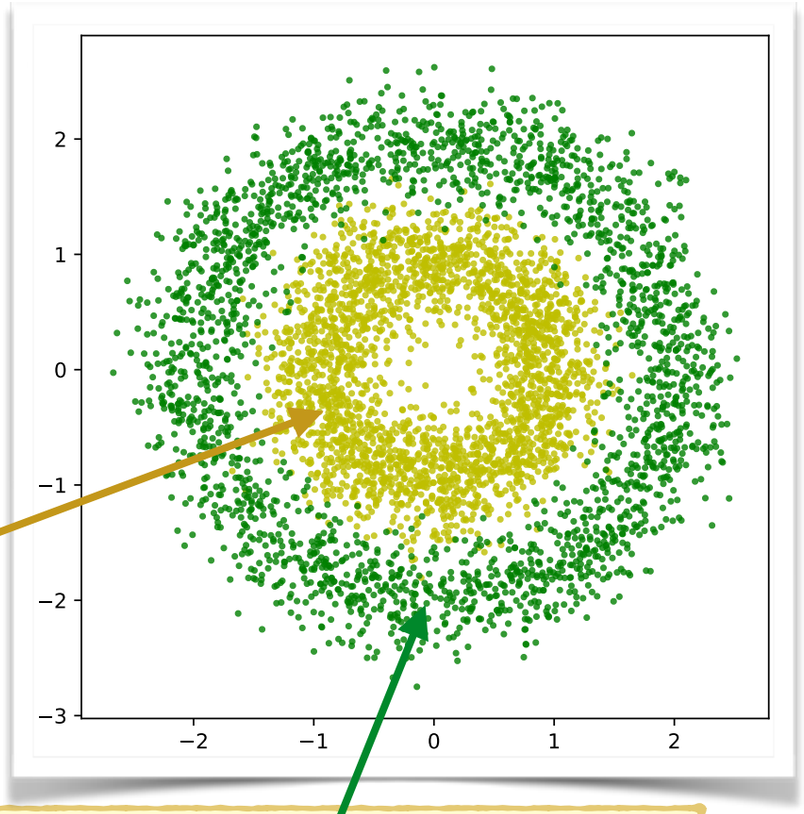
  - *Gaussian / Radial basis function (RBF)*

$$k(\vec{x}_i, \vec{x}_j) = \exp(-\gamma |\vec{x}_i - \vec{x}_j|^2)$$



AN ABSOLUTELY NONLINEAR CASE

- One can easy generate some data which is obviously **NOT** linear separable at all, for example, **two doughnuts**?



partial ex_ml3_3.py

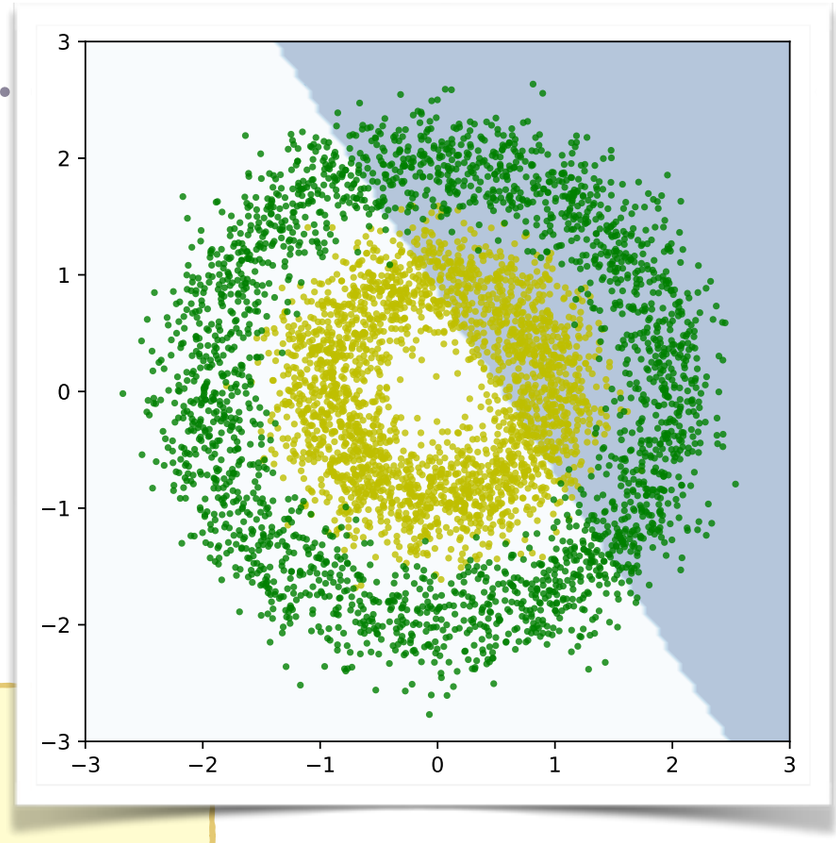
```
y_train = np.random.randint(0,2,5000)
rho = np.abs(np.random.randn(5000)/4.+1.+y_train)
phi = np.random.rand(5000)*np.pi*2.
x_train = np.c_[rho*np.cos(phi),rho*np.sin(phi)]

fig = plt.figure(figsize=(6,6), dpi=80)
plt.scatter(x_train[:,0][y_train==0], x_train[:,1][y_train==0], \
            c = 'y', s=5, alpha=0.8)
plt.scatter(x_train[:,0][y_train==1], x_train[:,1][y_train==1], \
            c = 'g', s=5, alpha=0.8)
plt.show()
```

NEARLY RANDOM SEPARATION?

- If you in any case inject this "two doughnuts" data into linear SVM, it will just give you a nearly random separation:

partial **ex_ml3_3a.py**

```
clf = svm.SVC(kernel='linear', C=1.)
clf.fit(x_train, y_train)

s_train = clf.score(x_train, y_train)
print('Performance (training):', s_train)

fig = plt.figure(figsize=(6,6), dpi=80)

xv, yv = np.meshgrid(np.linspace(-3.,3.,100), \
                     np.linspace(-3.,3.,100))
zv = clf.predict(np.c_[xv.ravel(), yv.ravel()])
plt.contourf(xv, yv, zv.reshape(xv.shape), \
    alpha=.3, cmap='Blues')
```




```
Performance (training):
0.562
```

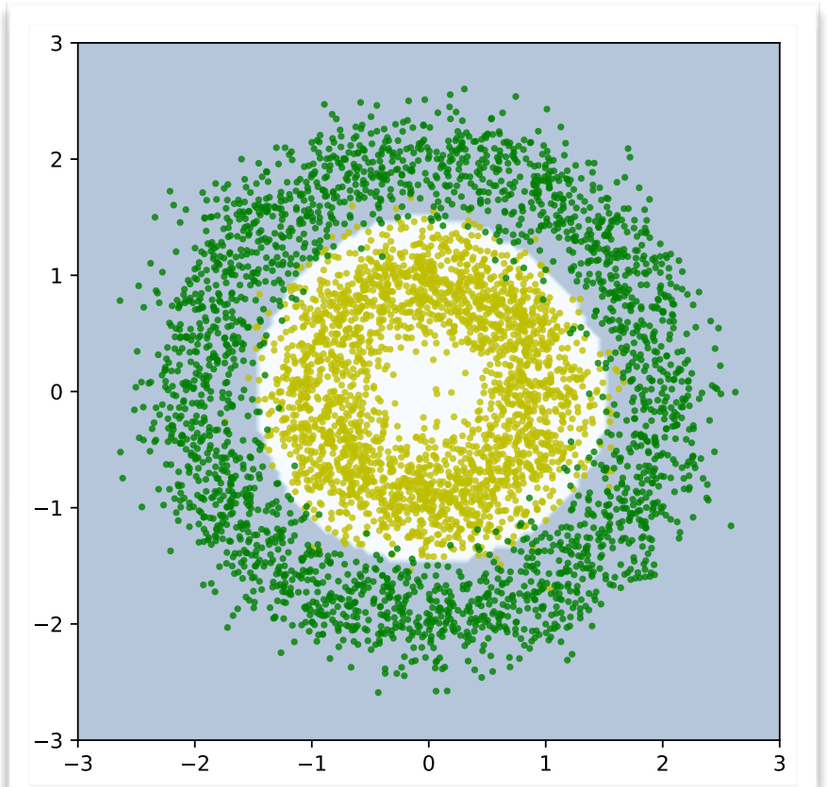
LET'S JUST SWITCH IT WITHIN THE CODE..?

- Let's try the RBF/Gaussian kernel and see how it works?
- Now you can see it can do a very nice job by introducing a nonlinear boundary!

partial **ex_ml3_3b.py**

```
. . . . . .
clf = svm.SVC(kernel='rbf', C=1.)
clf.fit(x_train, y_train)

s_train = clf.score(x_train, y_train)
print('Performance (training):', s_train)
```

```
Performance (training): 0.9754
```



Let's "roll back" to the initial/simplified problem of

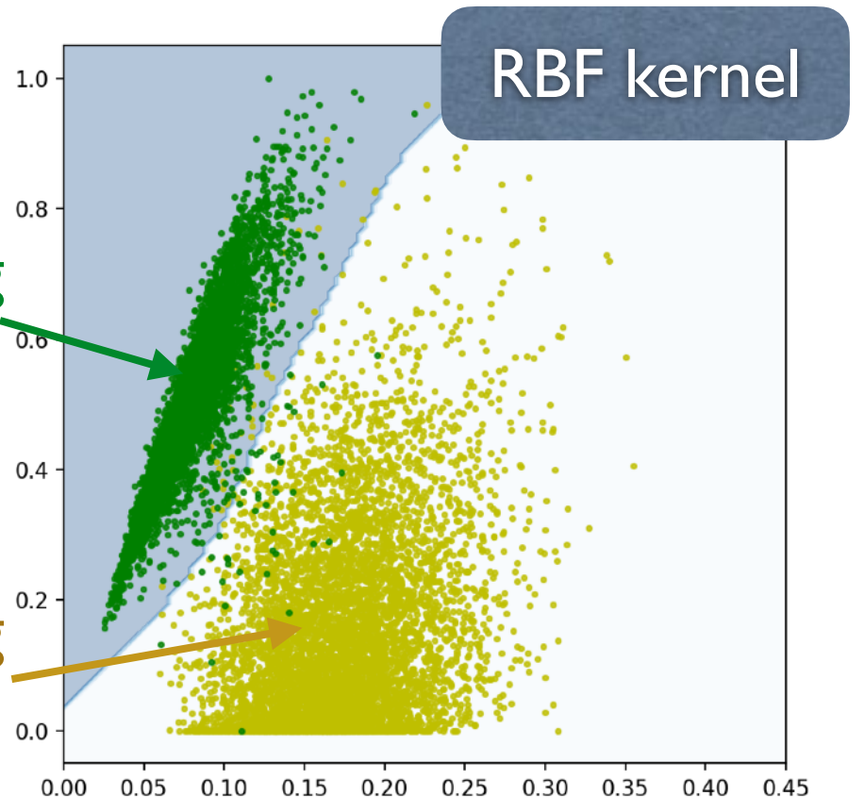
Digit 0 & 1 classification with 2 features only

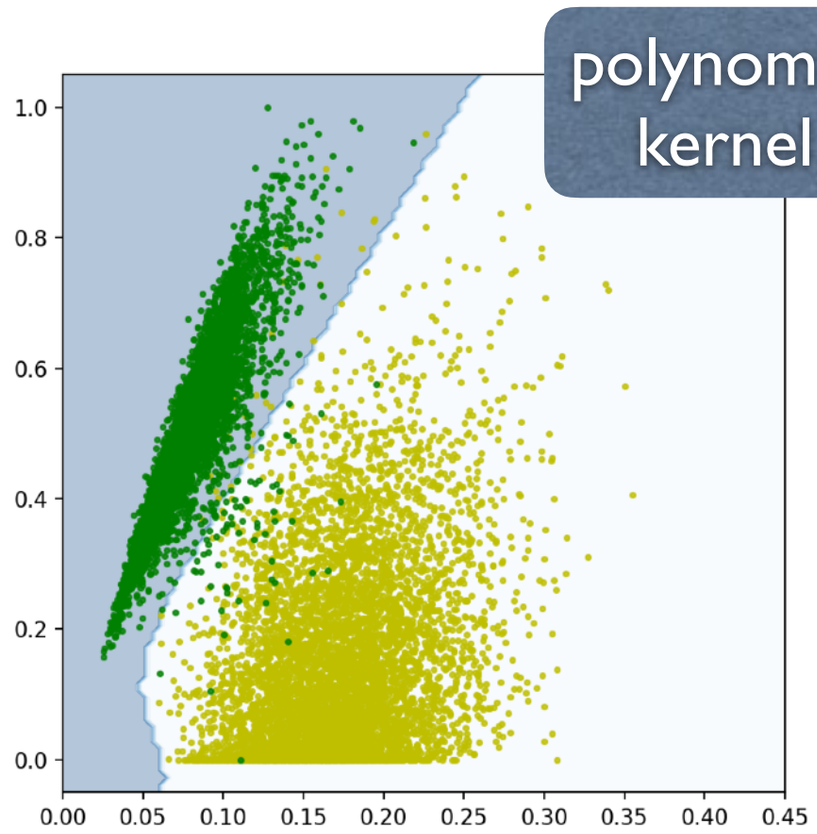
EFFECT OF A NONLINEAR KERNEL

- Switching to a non-linear kernel is easy!

Handwriting true ones

Handwriting true zeros





partial ex_ml3_4.py

```
. . . . . .
clf = svm.SVC(kernel='rbf', C=1.,)
clf.fit(x_train, y_train)
```

partial ex_ml3_4.py

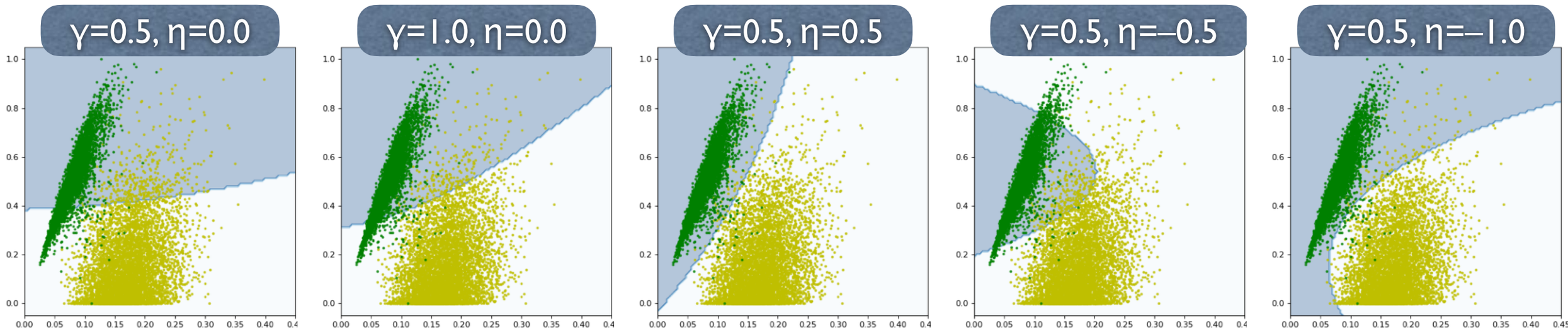
```
. . . . . .
clf = svm.SVC(kernel='poly', C=1.,)
clf.fit(x_train, y_train)
```

Very different behavior, but more than that actually!

HOW ABOUT USING A NONLINEAR KERNEL? (II)

- The non-linear kernels usually have some tunable parameters. For example:

Polynomial kernel: $k(\vec{x}_i, \vec{x}_j) = (\gamma \vec{x}_i \cdot \vec{x}_j + \eta)^d$



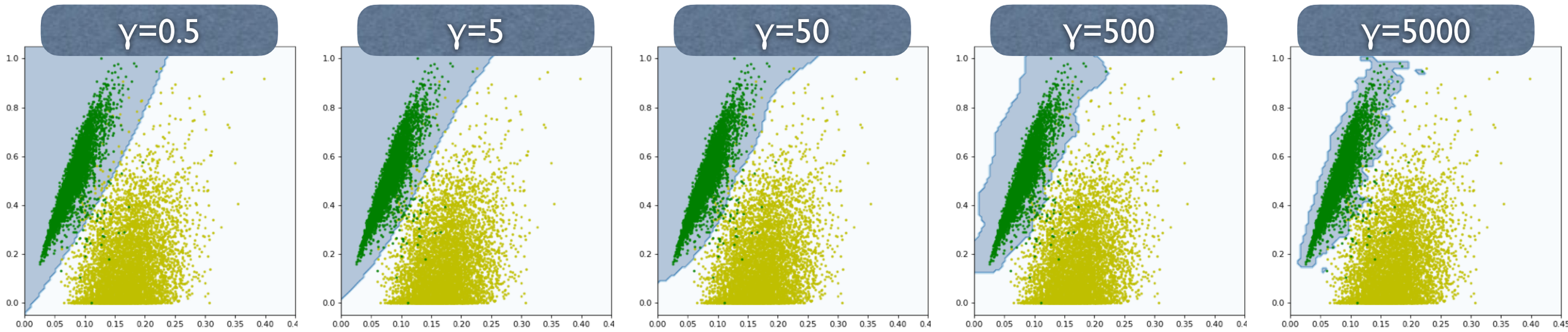
Those parameters γ, η (and the regularization C) used in SVM models are usually called the **Hyperparameters**, and they should be tuned to get the best performance.

(in the scikit-learn package γ = gamma, η = coef0)

HOW ABOUT USING A NONLINEAR KERNEL? (III)

- This also happens to the RBF (Gaussian) kernel. The γ parameter indicates the width of Gaussian function:

RBF kernel: $k(\vec{x}_i, \vec{x}_j) = \exp(-\gamma|\vec{x}_i - \vec{x}_j|^2)$



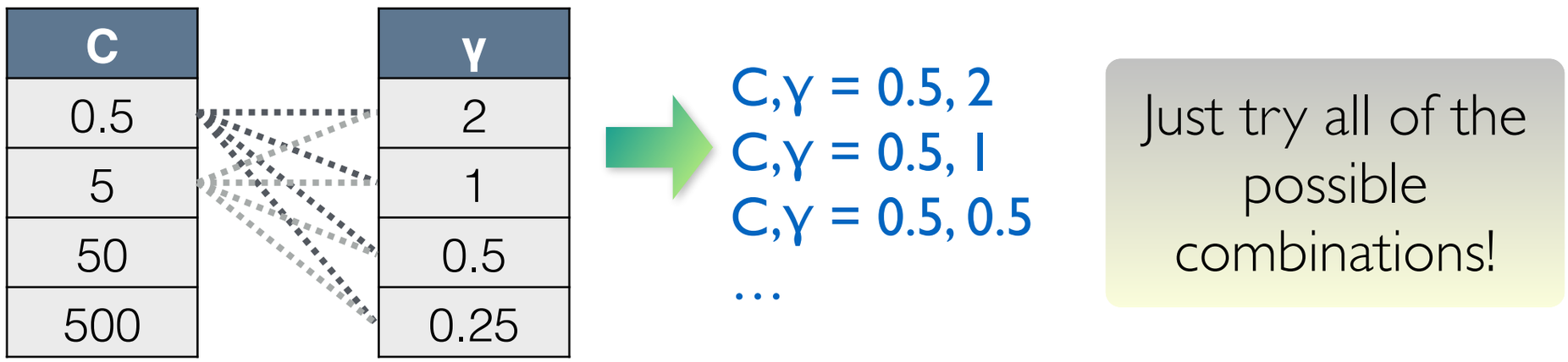
Small γ = wide Gaussian Large γ = narrower Gaussian

Obviously it is important to choose a good parameter! Recommend to check **sklearn.svm.SVC manual** and understand what are the options/default values.

https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

HYPERPARAMETER OPTIMIZATION

- In most of the ML algorithms, hyperparameter optimization is a step need to be carried out to get the optimal performance.
- The tuning can be carried out by simply trying several reasonable (based on experience) setups, or make an exhaustive searching in the allowed parameter space.
- As an example, let's tune the two hyperparameters in the **rbf kernel**, the regularization constant C and the kernel hyperparameter γ, with a classical **grid search**:



HYPERPARAMETER OPTIMIZATION (II)

- Within scikit-learn, there is a tool can automatically help to try all of the proposed combinations. Surely you can also perform such an optimization by yourself!

partial ex_ml3_4a.py

```
from sklearn import svm
from sklearn.model_selection import GridSearchCV

. . . . . .

clf = svm.SVC(kernel='rbf')

param = {'C':[0.5,5.,50.,500.], 'gamma':[8.0,4.0,2.0,1.0,0.5,0.25]}
grid = GridSearchCV(clf, param, verbose=3)
grid.fit(x_train, y_train)

print('Best SVM:')
print(grid.best_estimator_)

s_train = grid.score(x_train, y_train)
s_test = grid.score(x_test, y_test)
print('Performance (training):', s_train)
print('Performance (testing):', s_test)
```

↖ The grid search tool has a very similar interface.

HYPERPARAMETER OPTIMIZATION (III)

- Terminal output, as an automatic grid search:

```
Fitting 5 folds for each of 24 candidates, totalling 120 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent
workers.
[CV] C=0.5, gamma=8.0 ................................................
[CV] ..................... C=0.5, gamma=8.0, score=0.994, total=   0.1s
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.1s remaining:
0.0s
[CV] C=0.5, gamma=8.0 ................................................
[CV] ..................... C=0.5, gamma=8.0, score=0.994, total=   0.1s
. . . . . .
[CV] .................. C=500.0, gamma=0.25, score=0.990, total=   0.1s
[CV] C=500.0, gamma=0.25 .............................................
[CV] .................. C=500.0, gamma=0.25, score=0.996, total=   0.1s
[CV] C=500.0, gamma=0.25 .............................................
[CV] .................. C=500.0, gamma=0.25, score=0.994, total=   0.1s
[Parallel(n_jobs=1)]: Done 120 out of 120 | elapsed:   16.6s finished
Best SVM:
SVC(C=5.0, gamma=8.0)
Performance (training): 0.9939202526648243
Performance (testing): 0.9952718676122931
```

FULL DIGITS RECOGNITION W/ NONLINEAR SVM

- Let's revisit the handwriting digits recognition (full version) again, by simply modify the ending example from the previous lecture to a nonlinear kernel.

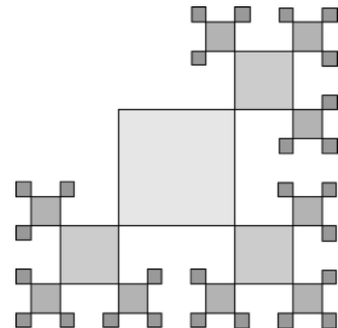
With slightly tuned SVM + RBF kernel, the performance can be better than the previous version!

```
optimization finished, #iter = 1523
obj = -158.940057, rho = -0.326238
nSV = 928, nBSV = 0
Total nSV = 6202
Performance (training): 1.0
Performance (testing): 0.9664
```

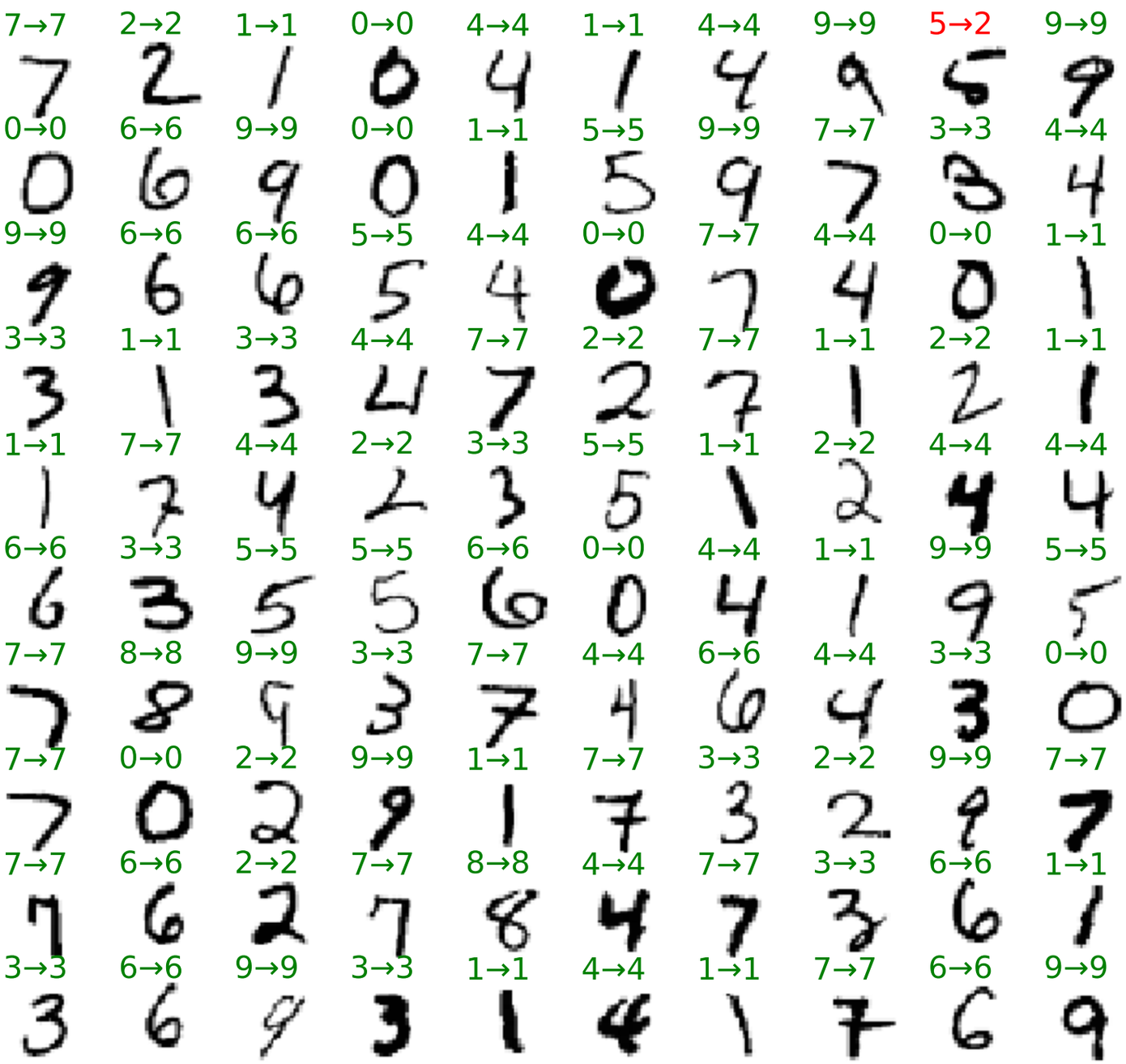
it was 0.917 before!

partial ex_ml3_5.py

```
. . . . . .

clf = svm.SVC(C=5., gamma=0.05, verbose=True)
clf.fit(x_train, y_train)

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)
```

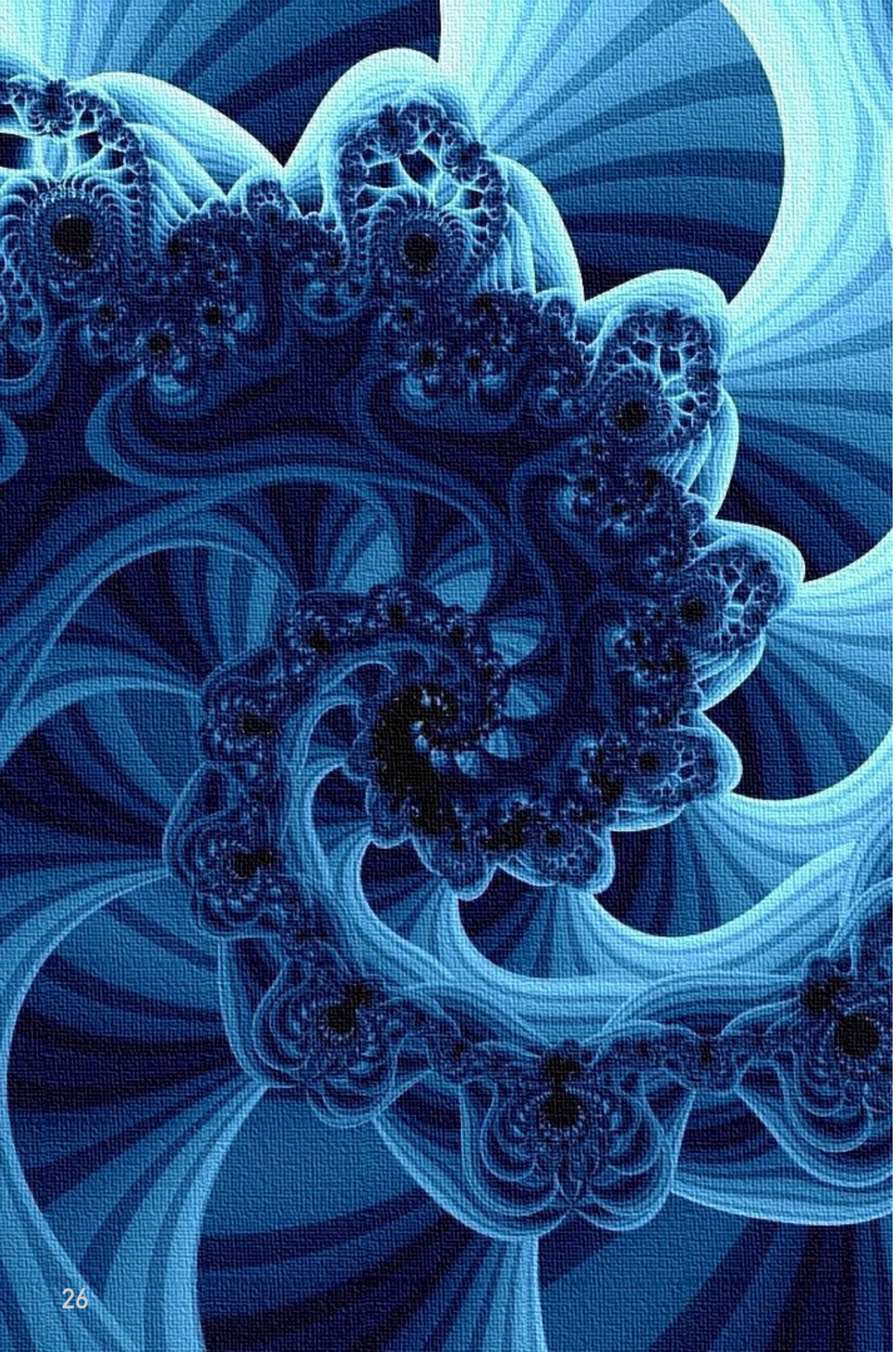


FULL DIGITS RECOGNITION W/ NONLINEAR SVM (II)



Only 1 misidentification found in the first 100 digits!

In fact if you **inject all of the training samples** (60K instead of 10K images). The performance would be superior (**~98.4%** of accuracy), but it will take really a while to run!



MODULE SUMMARY

- In this module we introduced the classical algorithm, Support Vector Machine, which is suitable for classification problem, as well as the importance of hyperparameter tuning.
- Next module we will start to enter the core discussions based on the most popular algorithm, the **artificial neural network**.