Lecture 3-1: Brief on machine learning

Kai-Feng Chen
National Taiwan University
ALL THOSE FANCY IDEAS…

- During past few years there were many very fancy names and ideas floating around:
  - **Machine Learning (ML)**
  - **Neural Network (NN)**
  - **Deep learning (DL)**
  - **Artificial Intelligence (AI)**

- But what is what actually?
Since an early flush of optimism in 1950s, smaller subsets of artificial intelligence – first machine learning, then deep learning, a subset of machine learning – have created ever larger disruptions.

And the neural network: a computing system or algorithm inspired by the biological neural networks, and is widely used in ML or DL applications.
And those terms became more and more apparent after AlphaGo beat human players…
LET’S START WITH MACHINE LEARNING…

- Quote from Wikipedia: “Machine learning is a field of computer science that gives computer systems the ability to learn (i.e. progressively improve performance on a specific task) with data, without being explicitly programmed.”

- So the key idea is to let your program to “learn from data”: Based on a set of data, your program can learn/train from it, and deploy your program to predict the properties of unknown data.
  - If each sample is more than a single input number: a multi-dimensional entry, it is said to have multiple attributes.
  - Used to perform studies across multiple dimensions while taking into account the effects of all variables on the responses of interest.

Multivariate analysis (MVA)
ML TRAINING & TESTING

- Typical ML operation steps with **independent data samples:**
  Training $\Rightarrow$ Testing ($\Rightarrow$ Validation) $\Rightarrow$ Deployment

**Learning Phase**
- **Training sample:** algorithm internal parameters (weights) tuning
- **Validation sample:** algorithm selection / hyperparameter tuning

**Deployment Phase**
- **Unknown data:** apply to real applications!

*Not always necessary but it is nice to have!*
PROBLEM SETTINGS IN CATEGORIES

■ **Supervised learning** — the data comes with additional features that we want to predict, as a “teacher.” The common problems can be:
  - **Classification**: want to separate the data into the targeting classes based on the input attributes.
  - **Regression**: want to enforce the output to match one or more continuous variables.

■ **Unsupervised learning** — no expected output features given to the learning algorithm, leaving it to work on its own to develop the structures in the input attributes.
  - **Clustering**: want to divide the data into groups. The groups are not known beforehand (*unlike the case for classification*).
INITIAL EXAMPLE:
BINARY CLASSIFICATION

- Binary classification is to classify the elements of a given set into two groups. Can be implemented as a supervised learning in the context of ML problems.
- For example, separating cats and dogs:
Well, one should not jump too far as an initial step. We should start with something much, much simpler and can be handled easily.

Let’s practice this classical problem by separating hand-writing zeros and ones!
The previous 0 and 1 images are collected from the famous MNIST (Modified National Institute of Standards and Technology) database.

It is a database of handwritten digits that is commonly used for training various image processing systems, including ML. The data contains 60,000 training images and 10,000 testing images. Each image has been normalized to 28×28 pixels.

The best performing convolutional neural network can recognize those testing images up to an error rate as low as 0.21%.

We will use these images thought out these 3 lectures about ML!
LOADING THE DATA

You can obtain the mnist.npz file from CEIBA or the lecture web:

```python
import numpy as np

mnist = np.load('mnist.npz')  # Just use the NumPy tool to read the data in!
x_train = mnist['x_train']  # Get the training data
y_train = mnist['y_train']

print('x shape:', x_train.shape)
print('y shape:', y_train.shape)

print('1st sample in x:', x_train[0])
print('1st sample in y:', y_train[0])
```

```
x shape: (60000, 28, 28)  # The training data has 60K of 28×28 images
y shape: (60000,)
1st sample in x:
[[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 . . . .
[0 0 0 0 0 0 0 0 0 0 0 0 3 18 18 126 136 175 26 166 255 247 127 0 0 0 0]]
1st sample in y: 5  # The first one is ‘5’
```
Let’s plot a given digit as an image!
And indeed it’s a 28×28 image!

```python
import numpy as np
import matplotlib.pyplot as plt

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

fig = plt.figure(figsize=(6, 6), dpi=80)
plt.imshow(x_train[0], cmap='Greys')
plt.show()
```

Yes, it’s a 5!
Now let’s focus on 0 and 1 only to simplify the situation:

```python
import numpy as np
import matplotlib.pyplot as plt

mnist = np.load('mnist.npz')
x_train = mnist['x_train']
y_train = mnist['y_train']
zero_and_one = x_train[y_train<=1]

fig = plt.figure(figsize=(6,9), dpi=80)
for i in range(6):
    plt.subplot(3,2,i+1)
    plt.imshow(zero_and_one[i], cmap='Greys')
plt.show()
```

l301-example-01b.py
- Remember each hand-writing digit consists of an image of $28 \times 28 = 784$ pixels, and each pixel is a number between 0 and 255.

- The target is to write a program to process these 784 inputs and identify the true digit. Taking the whole 784 numbers is not a straightforward task in general!

- The most straightforward way to do this is to find some features, which can be used in the subsequent classification task.
  - Specifically selected variable describes the signature of the input data.
  - The input dimensions can be dramatically reduced.
  - This step is usually called the feature extraction.
PREPARING THE INPUTS? (II)

- As an example, how about if we calculate the average pixel density and compare them? At least this value might be quite different for the written 0 and 1, since one uses more ink to write 0’s!
- To be simplified, let’s also convert the data to a float point number and normalize them to be within $[0,1]$:
Maybe we can do it easier, by just calculating the average of the centered 6×8 pixels, since there is obviously a “hole” for the 0’s?
Example code to compare the distributions:

```python
mnist = np.load('mnist.npz')
x_train = mnist['x_train']
y_train = mnist['y_train']
sample0 = x_train[y_train==0]/255.  # Extract those “0” and “1” images
sample1 = x_train[y_train==1]/255.
all_mean0 = sample0.mean(axis=(1,2))  # Average along x and y axes, keep the image index (axis 0)
all_mean1 = sample1.mean(axis=(1,2))
center_mean0 = sample0[:,10:18,11:17].mean(axis=(1,2))  # Only average the centered 6x8 pixels
center_mean1 = sample1[:,10:18,11:17].mean(axis=(1,2))
fig = plt.figure(figsize=(12,5), dpi=80)
plt.subplot(1,2,1)
plt.hist(all_mean0, bins=50, color='y')
plt.hist(all_mean1, bins=50, color='g', alpha=0.5)
plt.subplot(1,2,2)
plt.hist(center_mean0, bins=50, color='y')
plt.hist(center_mean1, bins=50, color='g', alpha=0.5)
plt.show()
```
Now we can extract these two features out of each image, and they actually distributed differently for 0 and 1:

Can we already separate the digits by looking at these distributions?
In principle we can already start to separate the images by looking at the resulting distribution, e.g.:

- 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 an 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!
Let’s produce such a ROC curve based on the distribution of full averaged pixel densities:

```python
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([0,0],[1,1], ls='--')
plt.grid()
plt.show()
```

Background contamination

Signal efficiency
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 could be a way to show the power of the classifier! (But this is application-dependent!)
Let’s compare the performance of the two feature variables in hand!

roc1_y = np.array([[all_mean1<th].sum() / len(all_mean1) for th in np.linspace(0.0, 0.4, 200)])
roc1_x = np.array([[all_mean0<th].sum() / len(all_mean0) for th in np.linspace(0.0, 0.4, 200)])

roc2_y = np.array([[center_mean1>th].sum() / len(center_mean1) for th in np.linspace(-0.01, 1., 200)])
roc2_x = np.array([[center_mean0>th].sum() / len(center_mean0) for th in np.linspace(-0.01, 1., 200)])

auc1, auc2 = 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
print('AUC(avg of all pixels): ', auc1)
print('AUC(avg of centered pixels): ', auc2)
In fact we have not touched any machine learning algorithm yet! But there is nothing to surprise since preparing data is an important task for ML studies.

Now let’s practice the easiest/simplest algorithm: **Linear discriminant analysis (LDA)**, or even simpler, the **Fisher’s discriminant**, by combining the multiple features into one variable:

\[
F = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + \cdots
\]

Calculate the weights \( w_i \) to maximize the separation \( S \).
FISHER'S DISCRIMINANT

- **Fisher's linear discriminant** is a method used in statistics, pattern recognition and machine learning to find a linear combination of features that characterizes or separates two or more classes of objects or events.

- Consider a set of observables: \( \overrightarrow{x} = (x_1, x_2, x_3, \cdots) \)

- For 2 different event classes, the **mean** and **covariance** of the observables are: \( \overrightarrow{\mu}_0, \overrightarrow{\mu}_1, \Sigma_0, \Sigma_1 \)

  \[
  \overrightarrow{\mu} = \langle \overrightarrow{x} \rangle \quad \Sigma = \langle (\overrightarrow{x} - \overrightarrow{\mu}) \cdot (\overrightarrow{x} - \overrightarrow{\mu})^T \rangle
  \]

- The separation \( S \) is given by

  \[
  S = \frac{(\overrightarrow{w} \cdot \overrightarrow{\mu}_1 - \overrightarrow{w} \cdot \overrightarrow{\mu}_0)^2}{\overrightarrow{w}^T \Sigma_1 \overrightarrow{w} + \overrightarrow{w}^T \Sigma_0 \overrightarrow{w}}
  \]

- The optimal weights can be determined by maximizing the \( S \):

  \[
  \overrightarrow{w} \propto (\Sigma_0 + \Sigma_1)^{-1}(\overrightarrow{\mu}_1 - \overrightarrow{\mu}_0)
  \]
So all we need to do is to calculate the **mean** and the **covariance** of the input features, and NumPy has the functionality to do it quickly!

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

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))])

mu0 = var0.mean(axis=1)  # mean values, in shape of (2,)
mu1 = var1.mean(axis=1)

cov0 = np.cov(var0)  # covariance matrix, in shape of (2,2)
cov1 = np.cov(var1)

weight = np.dot(linalg.inv(cov1+cov0),mu1-mu0)  # weight calculation
norm = np.sqrt((weight**2).sum())
weight /= norm

print('Resulting weights = ',weight)
```

*FISHER’S DISCRIMINANT (II)*
FISHER’S DISCRIMINANT (III)

- This is what we get:

Resulting weights = [−0.97661612  0.2149906 ]

Then the transformed discriminant will be

\[ F = -0.9766 \times \text{(full average)} + 0.2150 \times \text{(centered average)} \]

- And...just plot it!

```python
out0 = (var0.T*weight).sum(axis=1)
out1 = (var1.T*weight).sum(axis=1)
fig = plt.figure(figsize=(6,6), dpi=80)
plt.hist(out0, bins=50, color='y')
plt.hist(out1, bins=50, color='g', alpha=0.5)
plt.show()
```

Resulting weights = [−0.97661612  0.2149906 ]

Separation is obviously better!
WHAT DOES IT MEAN?

- It actually makes a projection along a given axis in 2D, and it will maximize the separation power:

- It also gives a nearly perfect ROC!
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!

Here are a couple of failed cases:

Obviously one has to improve the algorithm further…
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.
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” its 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.
At the beginning of this lecture, 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.
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?
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/
Let’s repeat the simple 2D LDA study with scikit-learn tool:

```python
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']<=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]

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)
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)
```

**LDA WITH SCIKIT-LEARN**

Let’s repeat the simple 2D LDA study with scikit-learn tool:

- Prepare both training and testing data
- Evaluate the performance for training and testing data
- Import LDA from scikit-learn

```python
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']<=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]

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)
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)
```

**L301-example-04.py (partial)**
The output scores show a good consistency between training and testing data.

Performance (training): 0.982945124358
Performance (testing): 0.986761229314

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

```python
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()
```
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.

```python
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.  # take out 7/8/9 three digits
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]
```

**DIMENSION REDUCTION WITH LDA**

take out 7/8/9 three digits  
flatten the inputs as 1D array

↑↑ take the first 3000 samples
Now each of the digits can be described by two variables, and you can see they are quite “distinguishable”!
COME 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?
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!
Consider a training data set of \( n \) points (vectors):

\[
\left( \overrightarrow{x}_1, y_1 \right), \ldots, \left( \overrightarrow{x}_n, y_n \right)
\]
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 \( \overrightarrow{w} \) is the normal vector to the hyperplane, and the parameter \( b/|\overrightarrow{w}| \) determines the offset of the hyperplane from the origin.
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:

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

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

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

\[
\begin{align*}
\text{Minimize } & \frac{1}{2} |w|^2 \\
\text{subject to } & y_i (\vec{w} \cdot \vec{x}_i - b) \geq 1 \\
& \text{for all } 1 \leq i \leq n
\end{align*}
\]

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.
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” (\(\xi\)):

\[
\begin{align*}
\vec{w} \cdot \vec{x}_i - b & \geq +1 - \xi_i, \quad \text{if } y_i = +1 \\
\vec{w} \cdot \vec{x}_i - b & \leq -1 + \xi_i, \quad \text{if } y_i = -1
\end{align*}
\]

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:

Minimize \(\frac{1}{2}|w|^2 + C \sum_i \xi_i\) subject to

\[
y_i(\vec{w} \cdot \vec{x}_i - b) + \xi_i \geq 1
\]

for all \(1 \leq i \leq n\) and \(\xi \geq 0\)

The regularization parameter \(C\) is a balance between the error term and the margin space.
In this lecture we will not spend time to explain how to really solve or optimize the SVM. Instead we will use scikit-learn package directly to demonstrate how to use it.

Let’s deploy our handwriting ones versus zeros example again:

```python
import numpy as np
from sklearn import svm

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)
```

---

Slightly better results?

Performance (training): 0.992577970786
Performance (testing): 0.994799054374

initial a SVM w/ linear kernel take C = 1 for now
Let's demonstrate the separation power of SVM directly with a "border" between the data points:

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

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

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),
            alpha=.3, cmap='Blues')

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()
```
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 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”.

```python
mnist = np.load('mnist.npz')
x_train = mnist['x_train'][::10000]/255.  # take only 10K images to speed up the training
y_train = mnist['y_train'][::10000]
x_test = mnist['x_test']/255.
y_test = mnist['y_test']  # input all 10 digits as 10 classes

x_train = np.array([[img.reshape((784,)) for img in x_train]])  # flatten the inputs as 1D array
x_test = np.array([[img.reshape((784,)) for img in x_test]])
```
A FULL DIGITS SEPARATION WITH SVM

```python
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'  # mark as red if there is mis-tag.
    plt.text(0.0, 0.0, '%d\to%d' % (y_test[i], p_test[i]), color=c, fontsize=15)
plt.show()
```

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!

↑ show the first 100 digits
A FULL DIGITS SEPARATION WITH 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 in the next lectures.
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 non-linear 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)**
  \[ k(\vec{x}_i, \vec{x}_j) = \exp(-\gamma |\vec{x}_i - \vec{x}_j|^2) \]
A TOTALLY NONLINEAR CASE

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

```python
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:

```python
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)

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!

```python
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 test this with our previous problem: Separating handwriting digits!
NONLINEAR SVM + DIGITS SEPARATION?

- Well, it improves only a little bit with the RBF kernel (~0.4%), but totally failed with polynomial kernel, why?
- This is because we still need to tune the parameters for those non-linear kernels, otherwise it will not show its power.
- We will continue to discuss this in our lecture next week.

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

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

- Optimization finished, #iter = 411
  - obj = -315.265385, rho = -0.578260
  - nSV = 481, nBSV = 430
  - Total nSV = 5197
  - Performance (training): **0.9295**
  - Performance (testing): **0.9213**

- Optimization finished, #iter = 958
  - obj = -1757.941589, rho = 0.880989
  - nSV = 1889, nBSV = 1887
  - Total nSV = 9970
  - Performance (training): **0.1648**
  - Performance (testing): **0.1595**
We will continue our discussions for nonlinear methods next week!
Practice data:
There is a data of 2 features from 3 classes, stored in the l301practice.npz file (can be downloaded from CEIBA or the lecture web). The following piece of code can be used to load it:

```python
import numpy as np

data = np.load('l301practice.npz')
x_train = data['x_train']
y_train = data['y_train']
x_test = data['x_test']
y_test = data['y_test']
```

The `x_train`, `y_train` contains 12000 samples, and `x_test`, `y_test` contains 6000 samples.
Practice 01a:
In the `x_train` data there are 2 different input variables (as 2 features). Please use the scatter plot to draw them, and separating for the 3 input classes based on the value stored in `y_train`. You may get a similar plot like this, if you take `l301-example-04b.py` as a template:
Practice 02b:
Take the practice data and inject them into a SVM. See how good can you separate the 3 classes with the linear kernel? Please estimate the accuracy for both training and testing data.

Performance (training): 0.xxxxx
Performance (testing): 0.yyyyy

You can visualize the separation as well!
HANDS-ON SESSION

Practice 02:
Take the l301_example_04.py as a template code, apply the following modifications:
- Instead of separation handwriting 0 and 1, let’s separate 8 and 9.
- Instead of putting in only two features, inject all 784 pixels into LDA. See how good we can separate them?

Performance (training): 0.xxxxx
Performance (testing): 0.yyyyy