SDSC6015 Stochastic Optimization and Online Learning

Xiang ZHOU

School of Data Science

City University of Hong Kong

Chapter 1: Introduction to Optimization Problem in Machine Learning

> 2 September 2020

Statistical learning protocol

- Observe $Z_1, \cdots, Z_N$. We assume that it is an i.i.d. sequence from an unknown probability distribution
- Make decision (or choose action/approximation) $a(Z_1, \cdots, Z_N)$
- Suffer an (average) loss $\mathbb{E}\ell(a(Z_1, \cdots, Z_n), Z)$ where ℓ is a given loss function.

Objective: Minimize the risk with respect to the decision a

$$\min_a \mathbb{E}\ell(a(Z_1, \dots, Z_n), Z)$$

In most case of machine learning, the risk is the *empirical average* from individuals and a is in parametric form associated with parameters denoed by θ:

$$\min_\theta L(\theta) = \min_\theta \sum_{j=1}^N \ell_j(\theta)$$

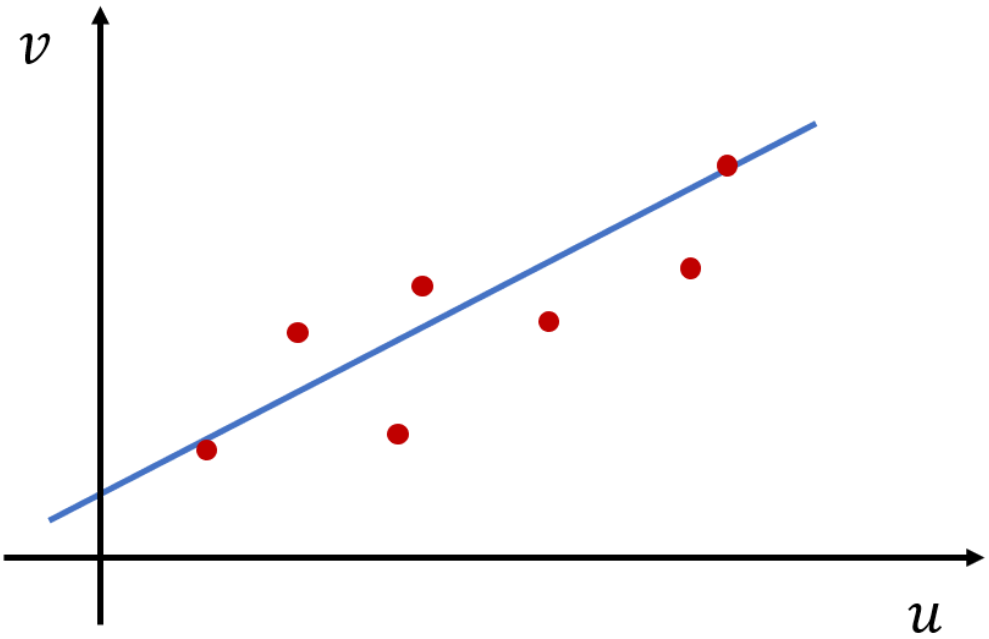
where $\ell_j(\theta)$ is the loss associated with the sample Z_j.

Optimzation in Machine Learning Models

1. Linear Regression
2. Ridge Regression, Lasso Regression
3. Pinciple Component Analysis (PCA)
4. Logistic Regression
5. Support Vector Machine
6. Neural Network
7. Generative Adversarial Network (GAN)
8. Reinforcement Learning

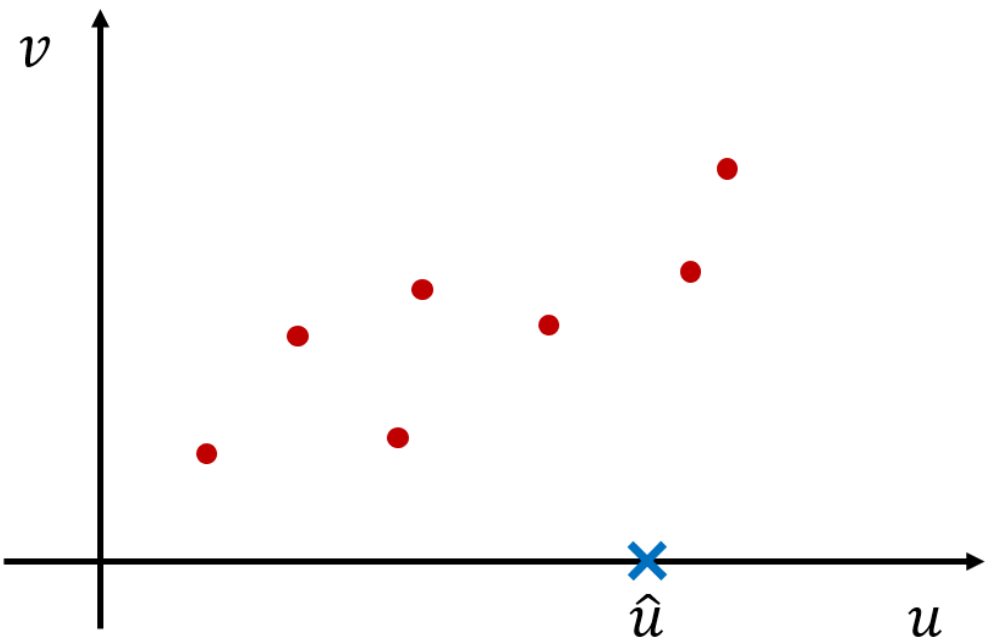
Linear Regression

$$\min_{\theta} L(\theta) = \sum_{i=1}^{N} \left(\theta^{\top} u^{(i)} - v^{(i)}\right)^2$$



We have a few samples of $\{(\mathbf{u}^{(i)}, v^{(i)})\}_{i=1}^{N}$ pairs, $\mathbf{u}^{(i)} \in \mathbb{R}^n$, $v^{(i)} \in \mathbb{R}$. $\mathbf{u}^{(i)}$ is called the input variable and $v^{(i)}$ is called the output variable. (Here $\mathbf{u}^{(i)}$, $v^{(i)}$ can be anything in real life, for example, the height and the weight.)

$$\dim(\mathbf{u}) = n$$

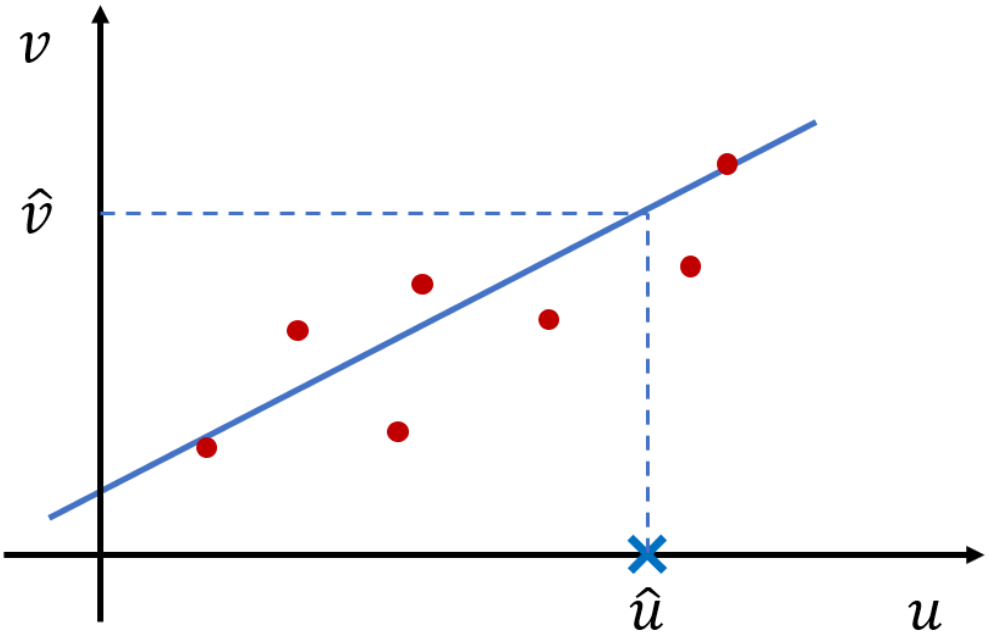


How to calculate the corresponding $\hat{v}$ for a new test input $\hat{\mathbf{u}}$?

One simple idea is to approximate $\mathbf{v}$ by a linear function of $\mathbf{u}$.

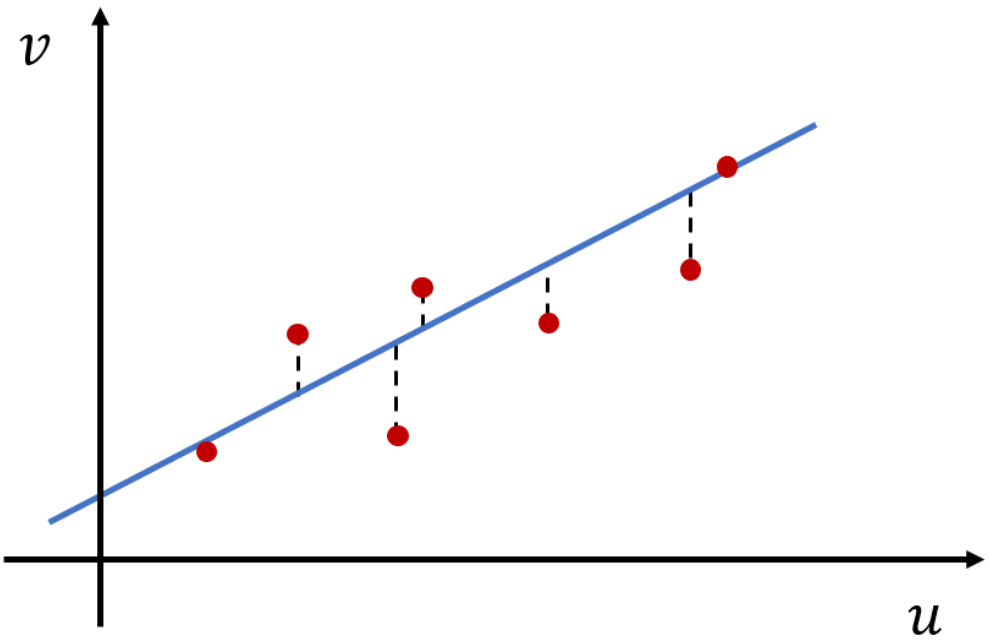
$$h(u) = h_\theta(\mathbf{u}) = \theta_0 u_0 + \theta_1 u_1 + \cdots + \theta_n u_n$$

where $\mathbf{u} = (u_0, u_1, u_2, \cdots, u_n)$ and $\boldsymbol{\theta} = (\theta_0, \theta_1, \theta_2, \cdots, \theta_n)$ is the parameter. Here we add one dimension $u_0 = 1$ for $\mathbf{u}$ to represent the constant term.



$\boldsymbol{\theta}$ should be chosen so that $h_\theta(\mathbf{u}^{(i)})$ and $v^{(i)}$ are "close" for all samples. The objective function of our problem can be formulated as

$$\min_{\theta} L(\theta) = \sum_{i=1}^{N} \left(\boldsymbol{\theta}^\top \mathbf{u}^{(i)} - v^{(i)}\right)^2$$



Define

$$U = \begin{bmatrix} u_0^{(1)} & u_2^{(1)} & \cdots & u_n^{(1)} \\ ®u_0^{(2)} & u_2^{(2)} & \cdots & u_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ u_0^{(N)} & u_2^{(N)} & \cdots & u_n^{(N)} \end{bmatrix}, V = \begin{bmatrix} v^{(1)} \\ v^{(2)} \\ \vdots \\ v^{(N)} \end{bmatrix}$$

The objective function can be rewritten into

$$\begin{aligned} L(\theta) &= \min_{\theta} \sum_{i=1}^{N} \left(\theta^\top \mathbf{u}^{(i)} - v^{(i)}\right)^2 \\ &= (U\theta - V)^\top (U\theta - V) \\ &= \theta^\top U^\top U\theta - 2\theta^\top U^\top V - V^\top V \end{aligned}$$

Solving Linear Regression is a (unconstrained) quadratic optimization.
Take the derivitave and set it to zero,

$$U^\top U\theta - U^\top V = 0$$

If $U^\top U$ is full rank, θ^* is given by:

$$\theta^* = (U^\top U)^{-1} U^\top V$$

Demonstration of Python Code

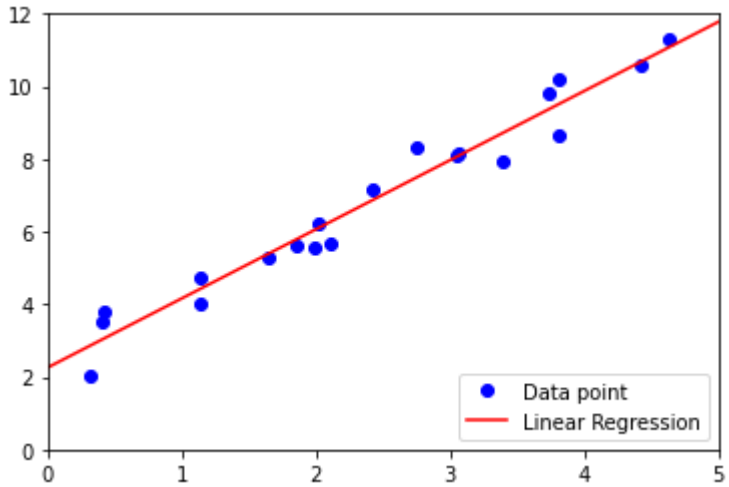
- `sklearn.linear_model`
- `LinearRegression().fit(input,output)`

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
dim=1; N=20
X = 5*np.random.random((N,dim))
y=2*(X)+1+2*np.random.random((N,1))

fig1=plt.figure()
ax1=fig1.add_subplot(1,1,1)
ax1.set_xlim(xmin =0, xmax =5)
ax1.set_ylim(ymin =0, ymax =12)
p0, =ax1.plot(X[:,0],y,"ob")

reg = LinearRegression().fit(X, y)
xplot=np.linspace(0,5,10)
yplot=reg.predict(xplot.reshape(-1,1))
p1, =ax1.plot(xplot,yplot,'r-')
plt.legend([p0,p1],["Data point","Linear Regression"],loc='lower right')
```

Out[1]:

```
<matplotlib.legend.Legend at 0x7fd132437fa0>
```



Ridge Regression and Lasso Regression

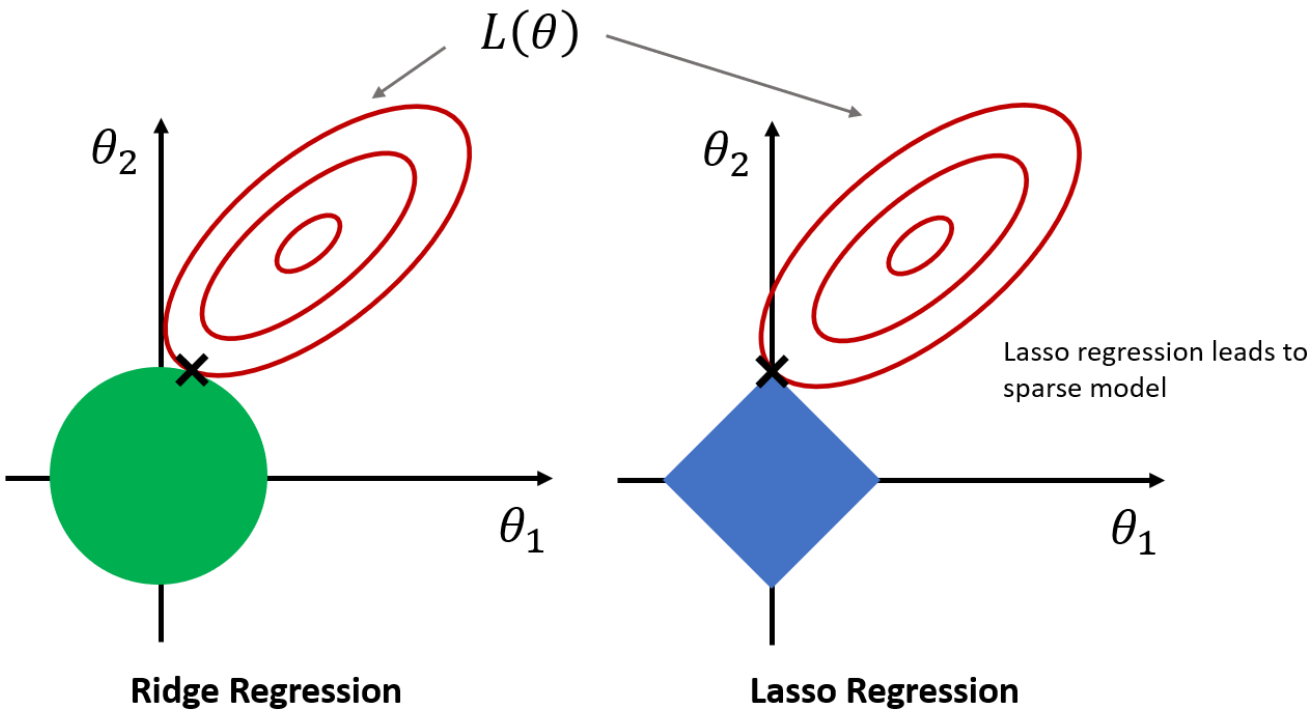
$$\min_{\theta} V(\theta) = L(\theta) + \lambda R(\theta)$$

where $L(\theta)$ **is the squared loss**

$$L(\theta) = \sum_{i=1}^{N} \left(\theta^{\top}\mathbf{u}^{(i)} - \mathbf{v}^{(i)}\right)^2$$

R **is the regularization (penalty to achieve some important machine learning goal)**

- For Ridge Regression: $R(\theta) = \sum_{j=0}^{n} \theta_j^2$
- For Lasso (Least absolute shrinkage and selection operator) Regression: $R(\theta) = \sum_{j=0}^{n} |\theta_j|$



- Adding a regularization term to an error function can **control over-fitting by discouraging the coefficients from reaching large values**.
- Lasso regression can lead to **sparse model** by driving some coefficients to zero.

To see this, first notice that minimizing the above objective function is equivalent to minimizing

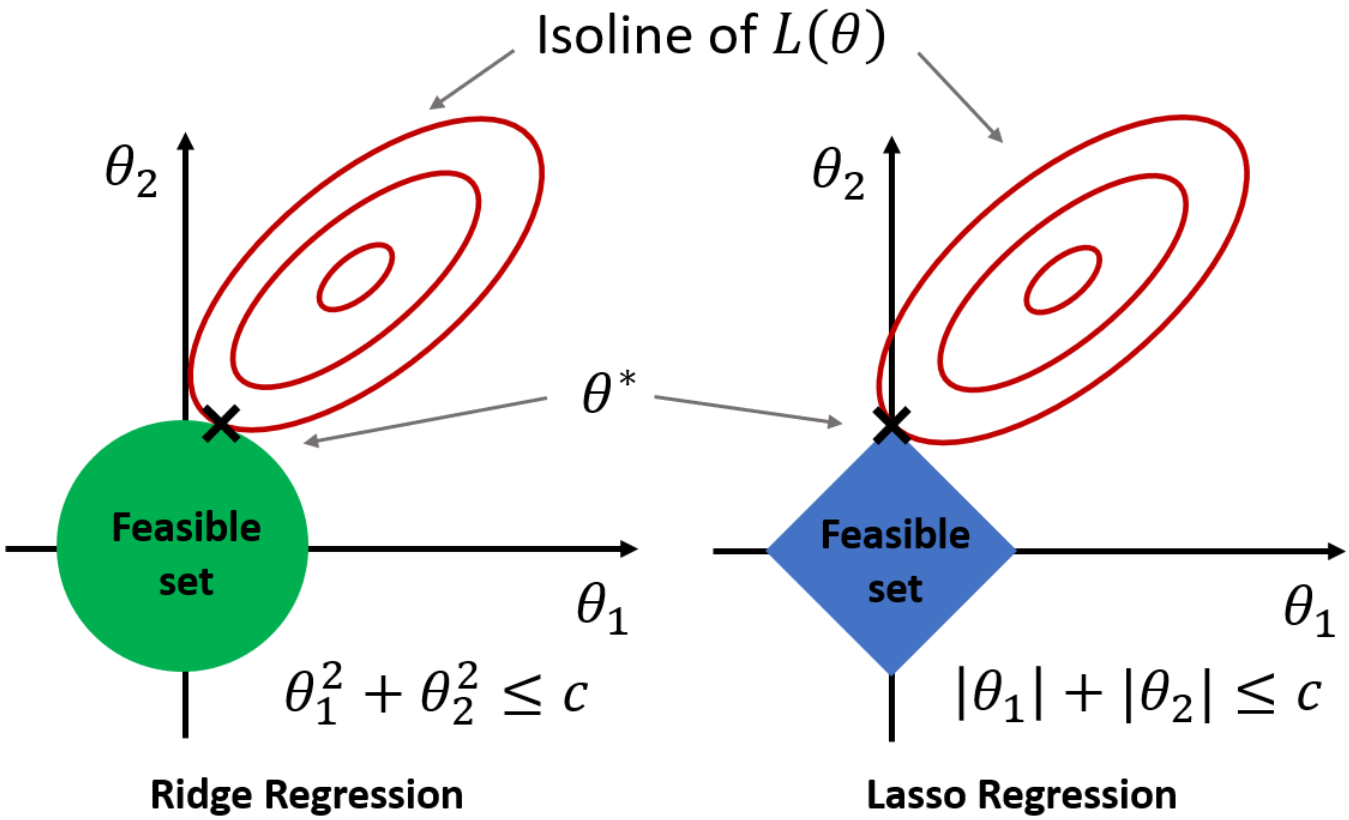
$$L(\theta) = \sum_{i=1}^{N} \left(\theta^{\top}\mathbf{u}^{(i)} - \mathbf{v}^{(i)}\right)^2$$

subject to the constraint

$$R(\theta) \leq c$$

where c is a constant and λ can be seen as the Lagrange multiplier.

The coefficients are constrained by the feasible set so they will not grow too large.
With a carefully chosen parameter, the solution of Lasso regression will rest on the axis, and the corresponding cosfficient reaches zero.



Illustrative Python Code in 1D

However, Ridge/Lasso shows their power only for high dim problem

In [2]:

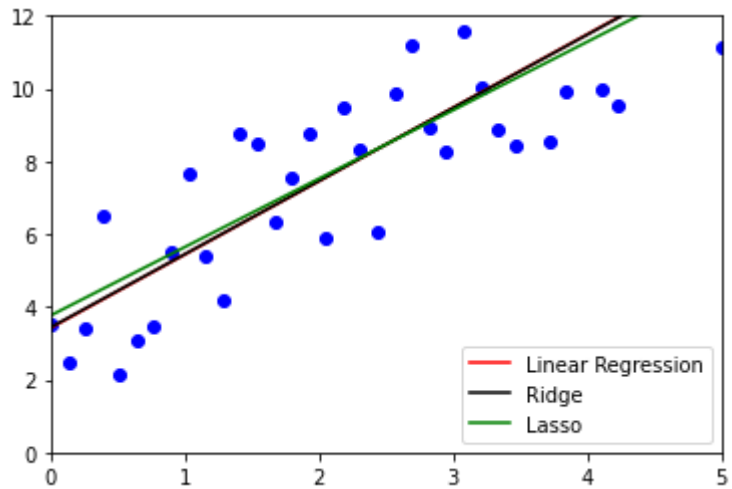
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Lasso
from sklearn.linear_model import Ridge

X = np.linspace(0.0, 5.0, num=40)
y=2*(X)+1+5*np.random.random((40))

##############################################################
X=X.reshape((-1,1))
reg0 = LinearRegression().fit(X, y)
reg1 = Ridge(alpha=0.3).fit(X, y)
reg2 = Lasso(alpha=0.3).fit(X, y)
```

In [3]:

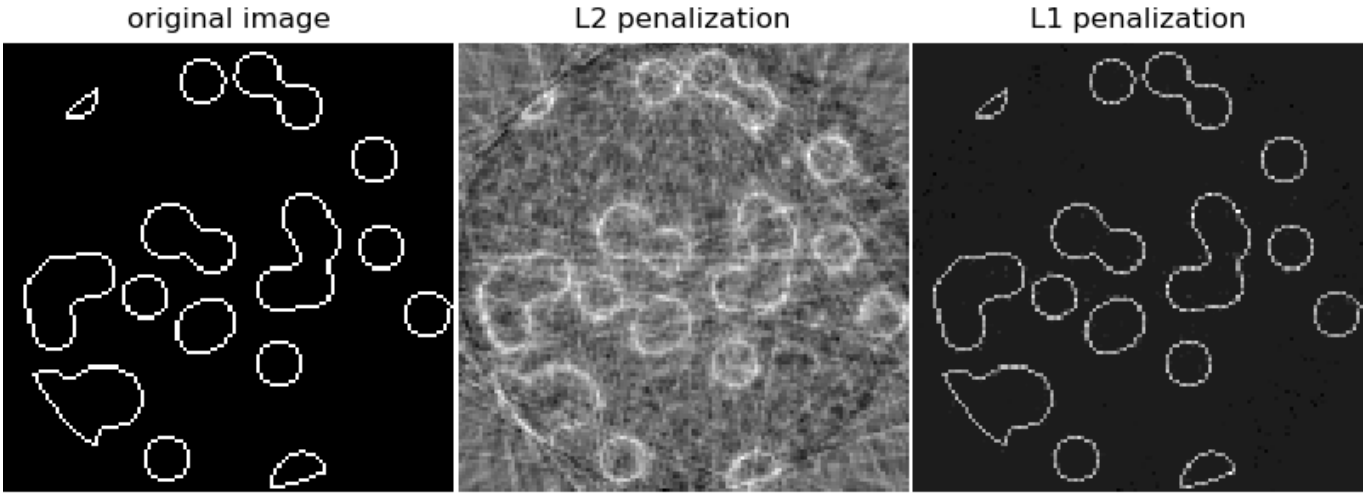
```
fig1=plt.figure()
ax1=fig1.add_subplot(1,1,1)
ax1.set_xlim(xmin =0, xmax =5)
ax1.set_ylim(ymin =0, ymax =12)
p0, =ax1.plot(X,y,"ob")

xplot=np.linspace(0,5,10)
yplot0=reg0.predict(xplot.reshape(-1,1))
p1, =ax1.plot(xplot,yplot0,"-r")
yplot2=reg1.predict(xplot.reshape(-1,1))
p2, =ax1.plot(xplot,yplot2,"-k")
yplot1=reg2.predict(xplot.reshape(-1,1))
p3, =ax1.plot(xplot,yplot1,"-g")

plt.legend([p1,p2,p3],["Linear Regression","Ridge","Lasso"],loc='lower right')
```

Out[3]:

```
<matplotlib.legend.Legend at 0x7fd1324860d0>
```



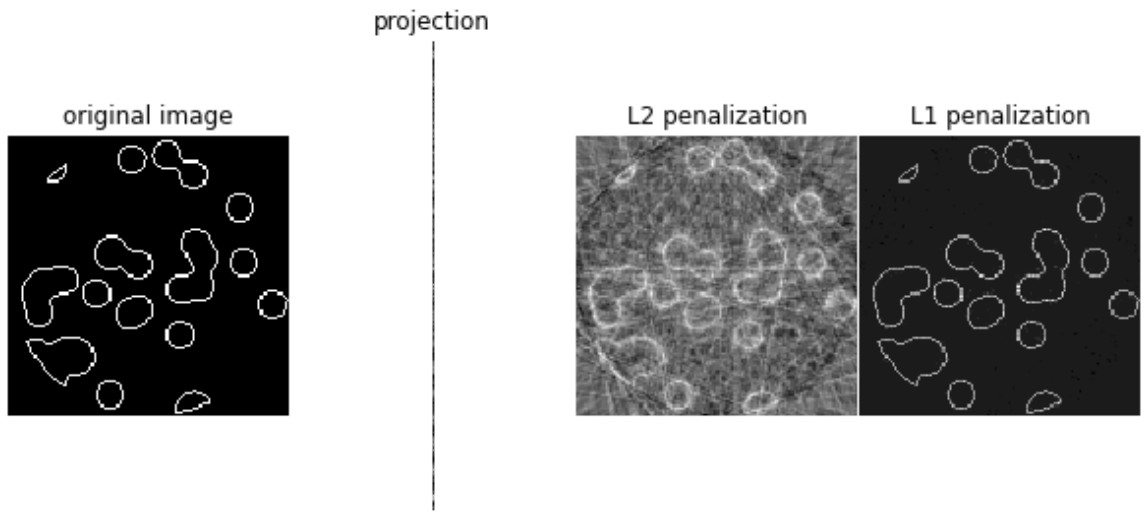
Application in compressive sensing: tomography reconstruction with L1 prior (Lasso)



In [4]:

```
# https://scikit-learn.org/stable/auto_examples/applications/plot_tomography_l1_
reconstruction.html
import numpy as np
from scipy import sparse
from scipy import ndimage
from sklearn.linear_model import Lasso
from sklearn.linear_model import Ridge
import matplotlib.pyplot as plt


def _weights(x, dx=1, orig=0):
    x = np.ravel(x)
    floor_x = np.floor((x - orig) / dx).astype(np.int64)
    alpha = (x - orig - floor_x * dx) / dx
    return np.hstack((floor_x, floor_x + 1)), np.hstack((1 - alpha, alpha))


def _generate_center_coordinates(l_x):
    X, Y = np.mgrid[:l_x, :l_x].astype(np.float64)
    center = l_x / 2.
    X += 0.5 - center
    Y += 0.5 - center
    return X, Y

#Compute the tomography design matrix.
def build_projection_operator(l_x, n_dir):
    X, Y = _generate_center_coordinates(l_x)
    angles = np.linspace(0, np.pi, n_dir, endpoint=False)
    data_inds, weights, camera_inds = [], [], []
    data_unravel_indices = np.arange(l_x ** 2)
    data_unravel_indices = np.hstack((data_unravel_indices,
                                      data_unravel_indices))
    for i, angle in enumerate(angles):
        Xrot = np.cos(angle) * X - np.sin(angle) * Y
        inds, w = _weights(Xrot, dx=1, orig=X.min())
        mask = np.logical_and(inds >= 0, inds < l_x)
        weights += list(w[mask])
        camera_inds += list(inds[mask] + i * l_x)
        data_inds += list(data_unravel_indices[mask])
    proj_operator = sparse.coo_matrix((weights, (camera_inds, data_inds)))
    return proj_operator


def generate_synthetic_data():
    rs = np.random.RandomState(0)
    n_pts = 36
    x, y = np.ogrid[0:l, 0:l]
    mask_outer = (x - l / 2.) ** 2 + (y - l / 2.) ** 2 < (l / 2.) ** 2
    mask = np.zeros((l, l))
    points = l * rs.rand(2, n_pts)
    mask[(points[0]).astype(np.int), (points[1]).astype(np.int)] = 1
    mask = ndimage.gaussian_filter(mask, sigma=l / n_pts)
    res = np.logical_and(mask > mask.mean(), mask_outer)
    return np.logical_xor(res, ndimage.binary_erosion(res))


# Generate synthetic images, and projections
l = 128
proj_operator = build_projection_operator(l, l // 7)
```

```
data = generate_synthetic_data()
proj = proj_operator * data.ravel()[:, np.newaxis]
proj += 0.15 * np.random.randn(*proj.shape)

# Reconstruction with L2 (Ridge) penalization
rgr_ridge = Ridge(alpha=0.2)
rgr_ridge.fit(proj_operator, proj.ravel())
rec_l2 = rgr_ridge.coef_.reshape(l, l)

# Reconstruction with L1 (Lasso) penalization
# the best value of alpha was determined using cross validation
# with LassoCV
rgr_lasso = Lasso(alpha=0.001)
rgr_lasso.fit(proj_operator, proj.ravel())
rec_l1 = rgr_lasso.coef_.reshape(l, l)

plt.figure(figsize=(8, 3.3))
plt.subplot(141)
plt.imshow(data, cmap=plt.cm.gray, interpolation='nearest')
plt.axis('off')
plt.title('original image')
plt.subplot(142)
plt.imshow(proj, cmap=plt.cm.gray, interpolation='nearest')
plt.title('projection')
plt.axis('off')
plt.subplot(143)
plt.imshow(rec_l2, cmap=plt.cm.gray, interpolation='nearest')
plt.title('L2 penalization')
plt.axis('off')
plt.subplot(144)
plt.imshow(rec_l1, cmap=plt.cm.gray, interpolation='nearest')
plt.title('L1 penalization')
plt.axis('off')

plt.subplots_adjust(hspace=0.01, wspace=0.01, top=1, bottom=0, left=0,
                    right=1)

plt.show()
```



Principle Component Analysis (PCA)

To find the maximal eigenvalue u_1 (with length 1) of the sample covariance matrix S

$$Su_i = \lambda_i u_i$$

where $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n \geq 0$

The positive definite matrix S

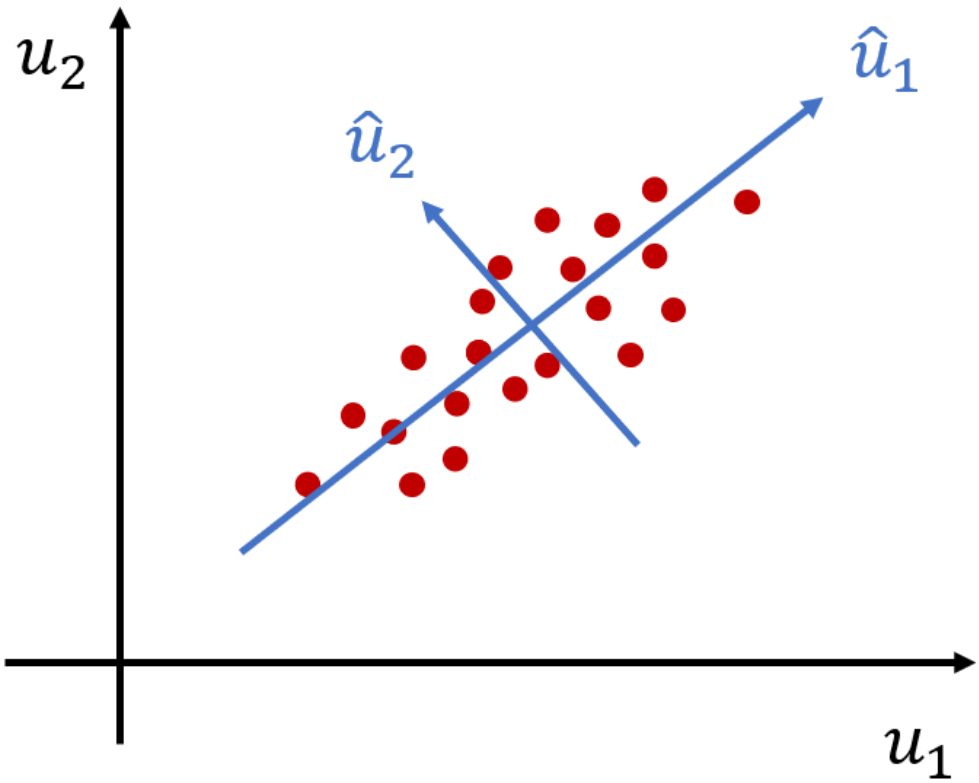
$$S = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{u}^{(i)} - \bar{\mathbf{u}})(\mathbf{u}^{(i)} - \bar{\mathbf{u}})^\top, \qquad \bar{\mathbf{u}} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{u}^{(i)}$$

- The largest eigenvector maximizes the Rayley quotient

$$u_1 = \arg\max_{\|u\|_2=1} u^\top S u$$

- Introduce the Lagrangian multiplier λ_1, then u_1 solves the unconstrained quadratic optmization problem

$$\max_{\hat{u}_1} \; V(\hat{u}_1) = \hat{u}_1^\top S \hat{u}_1 + \lambda_1(1 - \hat{u}_1^\top \hat{u}_1)$$



- Consider a dataset $\{u^{(i)}\}_{i=1}^N$. We want to project the data onto to a space with lower dimensionality and the projected data should explain the variance of the original data to a maximal extent.
- The explained variance tells you how much information (variance) can be attributed to each of the principal components. So a **maximization** is used for the *principle component*.
- To begin with, we calculate the projection onto a one-dimensional space. Define the direction with a vector $\hat{u}_1$. Since only the direction of $\hat{u}_1$ matters to us, we let $\hat{u}_1$ be a unit vector so that $\hat{u}_1^\top \hat{u}_1 = 1$. The projected value of each $\mathbf{u}^{(i)}$ is $\hat{u}_1^\top \mathbf{u}^{(i)}$.

The **variance of the projected data** is given by

$$\frac{1}{N}\sum_{i=1}^{N}\{\hat{u}_1^\top \mathbf{u}^{(i)} - \hat{u}_1^\top \bar{\mathbf{u}}\}^2 = \hat{u}_1^\top S \hat{u}_1$$

where

$$S = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{u}^{(i)} - \bar{\mathbf{u}})(\mathbf{u}^{(i)} - \bar{\mathbf{u}})^\top$$

$$\bar{\mathbf{u}} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{u}^{(i)}$$

subject to

$$\hat{u}_1^\top \hat{u}_1 = 1$$

Using the Lagrange mutiplier method, the problem can be reform as an unconstrained problem of

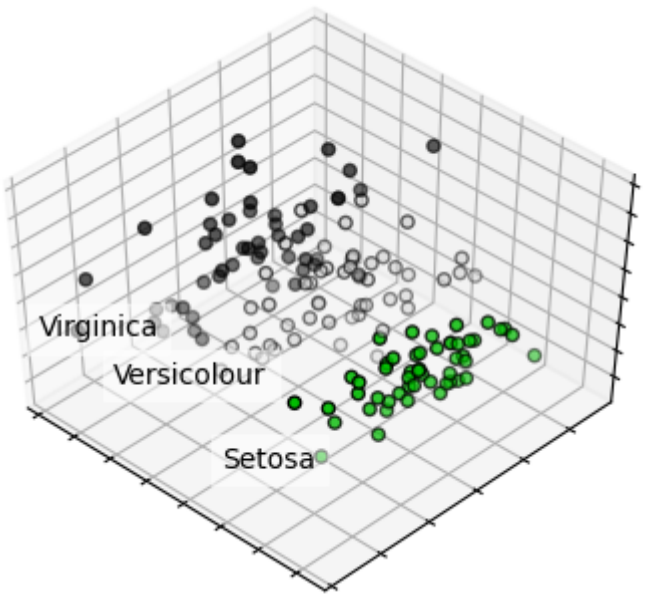
$$\max_{\hat{u}_1} \ \hat{u}_1^\top S \hat{u}_1 + \lambda_1(1 - \hat{u}_1^\top \hat{u}_1)$$

Taking the derivative with respect to $\hat{u}_1$ and set it to zero, we have

$$S\hat{u}_1 = \lambda_1 \hat{u}_1$$
$$\hat{u}_1^\top S \hat{u}_1 = \lambda_1$$

So the variance will be a maximum when we set $\hat{u}_1$ equal to the eigenvector having the largest eigenvalue λ_1. And $\hat{u}_1$ is the first principle component.

Application: Reduce four-dim iris data to three-dim



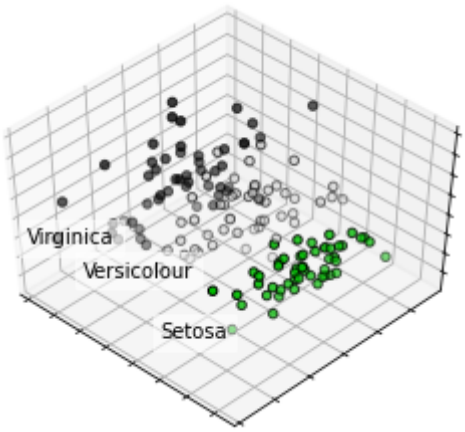
In [5]:

```
#https://scikit-learn.org/stable/auto_examples/decomposition/plot_pca_iris.html#
sphx-glr-auto-examples-decomposition-plot-pca-iris-py
```

In [6]:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

from sklearn import decomposition
from sklearn import datasets

np.random.seed(5)

centers = [[1, 1], [-1, -1], [1, -1]]
iris = datasets.load_iris()
X = iris.data
print('Dimensions: %s x %s' % (X.shape[0], X.shape[1]))
y = iris.target

pca = decomposition.PCA(n_components=3)
pca.fit(X)
X = pca.transform(X)

fig = plt.figure(1, figsize=(4, 3))
ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)

for name, label in [('Setosa', 0), ('Versicolour', 1), ('Virginica', 2)]:
    ax.text3D(X[y == label, 0].mean(),
              X[y == label, 1].mean() + 1.5,
              X[y == label, 2].mean(), name,
              horizontalalignment='center',
              bbox=dict(alpha=.5, edgecolor='w', facecolor='w'))
# Reorder the labels to have colors matching the cluster results
y = np.choose(y, [1, 2, 0]).astype(np.float)
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=y, cmap=plt.cm.nipy_spectral,
           edgecolor='k')

ax.w_xaxis.set_ticklabels([])
ax.w_yaxis.set_ticklabels([])
ax.w_zaxis.set_ticklabels([])

plt.show()
```

```
Dimensions: 150 x 4
```

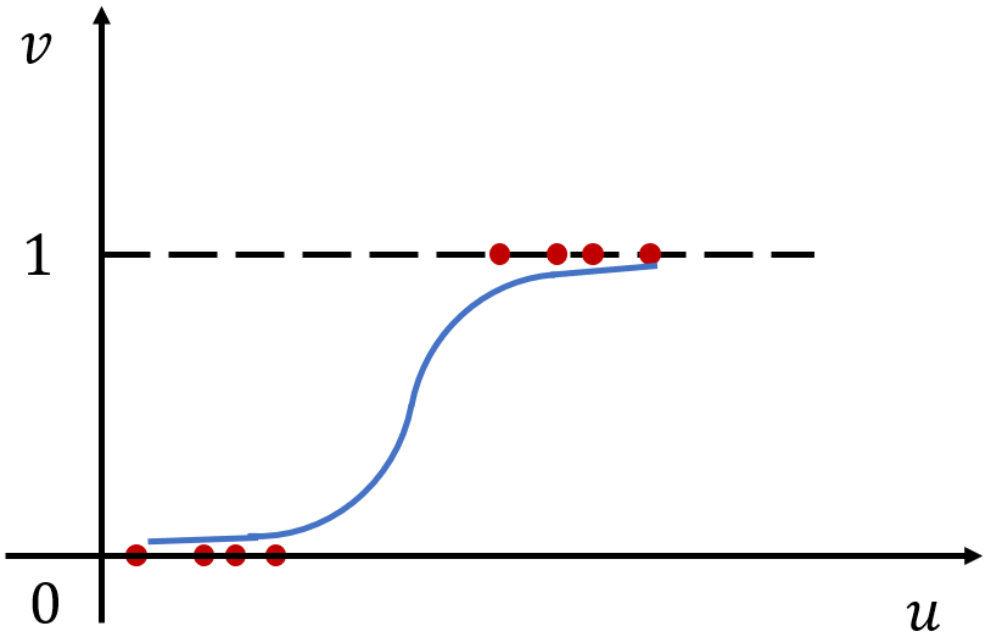


Logistic Regression for Binary Classification

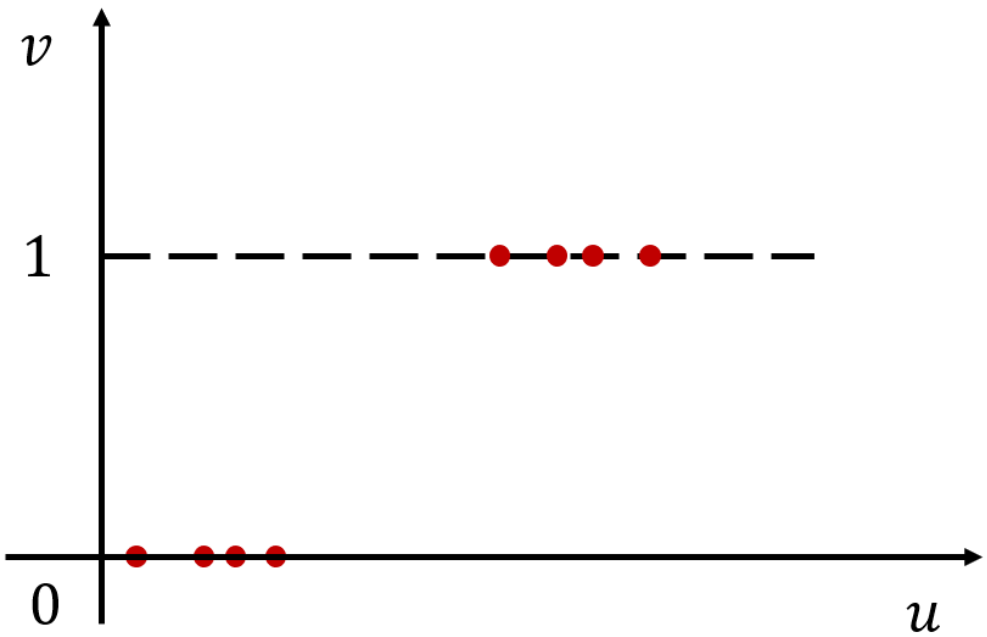
Consider binary classification problem with N training examples $\{\mathbf{u}^{(i)}, v^{(i)}\}_{i=0}^{N}$, $v \in \{0, 1\}$. The loss function (**Logistic Loss**) of logistic regression is

$$\max_{\theta} V(\theta) = \sum_{i=1}^{N} \left\{ -\log[1 + \exp(-\theta^{\top}\mathbf{u}^{(i)})] - [1 - v^{(i)}]\theta^{\top}\mathbf{u}^{(i)} \right\}$$

We will see why this form.



Similar to regression model, we have a few samples of $\{(\mathbf{u}^{(i)}, v^{(i)})\}_{i=1}^{N}$ pairs. However, v is now binary. i.e. $v^{(i)} \in \{0, 1\}$.



Consider mapping the linear function to a value within $[0, 1]$ with sigmoid function

$$g(z) = \frac{1}{1 + \exp(-z)}$$

Define

$$h_\theta(\mathbf{u}) = g(\boldsymbol{\theta}^\top \mathbf{u}) = \frac{1}{1 + \exp(-\boldsymbol{\theta}^\top \mathbf{u})}$$

Assume $v^{(i)}$ are independent Bernoulli random variables with probability of $h_\theta(\mathbf{u}^{(i)})$. Their probability mass functions are given by

$$p(v^{(i)}|\mathbf{u}^{(i)}; \boldsymbol{\theta}) = [h_\theta(u^{(i)})]^{v^{(i)}}[1 - h_\theta(u^{(i)})]^{1-v^{(i)}}$$

The likelihood function is

$$\prod_{i=1}^{N} \left\{ [h_\theta(u^{(i)})]^{v^{(i)}}[1 - h_\theta(u^{(i)})]^{1-v^{(i)}} \right\}$$

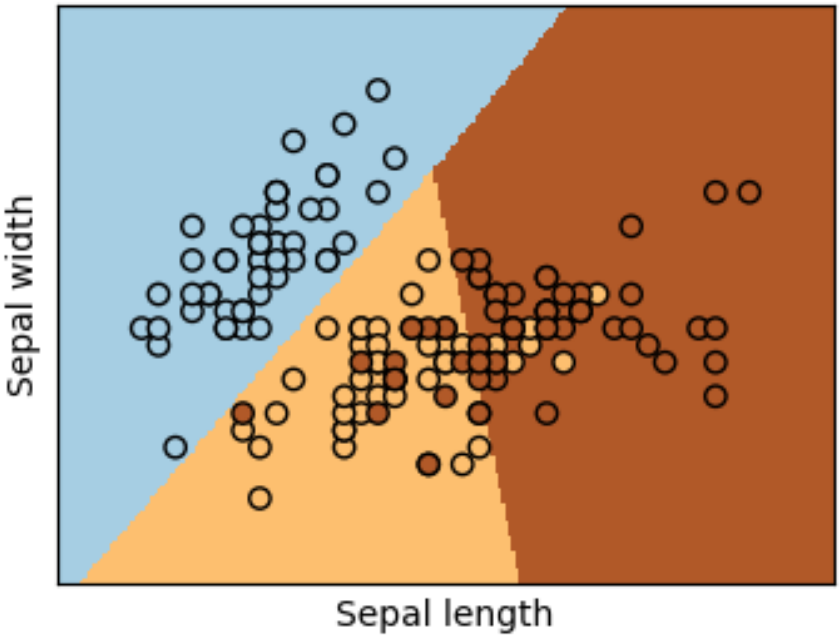
The loglikelihood is

$$\begin{aligned}
&\sum_{i=1}^{N} \log\left\{ [h_\theta(u^{(i)})]^{v^{(i)}}[1 - h_\theta(u^{(i)})]^{1-v^{(i)}} \right\} \\
&= \sum_{i=1}^{N} \left\{ v^{(i)} \log(h_\theta(u^{(i)})) + [1 - v^{(i)}] \log[1 - h_\theta(u^{(i)})] \right\} \\
&= \sum_{i=1}^{N} \Big\{ - v^{(i)} \log(1 + \exp(-\boldsymbol{\theta}^\top u^{(i)})) + \\
&\qquad\qquad [1 - v^{(i)}] \left[\log \exp(-\boldsymbol{\theta}^\top \mathbf{u}) - \log(1 + \exp(-\boldsymbol{\theta}^\top \mathbf{u})) \right] \Big\} \\
&= \sum_{i=1}^{N} \left\{ - \log[1 + \exp(-\boldsymbol{\theta}^\top \mathbf{u}^{(i)})] - [1 - v^{(i)}] \boldsymbol{\theta}^\top \mathbf{u}^{(i)} \right\}
\end{aligned}$$

The problem can be formulate as

$$\max_{\boldsymbol{\theta}} V(\boldsymbol{\theta}) = \sum_{i=1}^{N} \left\{ - \log[1 + \exp(-\boldsymbol{\theta}^\top \mathbf{u}^{(i)})] - [1 - v^{(i)}] \boldsymbol{\theta}^\top \mathbf{u}^{(i)} \right\}$$

Application to Logistic Regression of 3-class Classification on Iris dataset

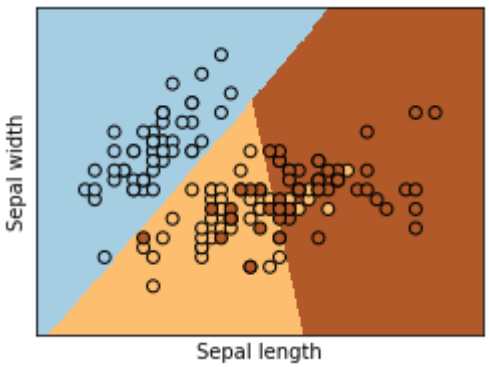


In [7]:

```
#https://scikit-learn.org/stable/auto_examples/linear_model/plot_iris_logistic.html?highlight=logistic%20regression
```

In [8]:

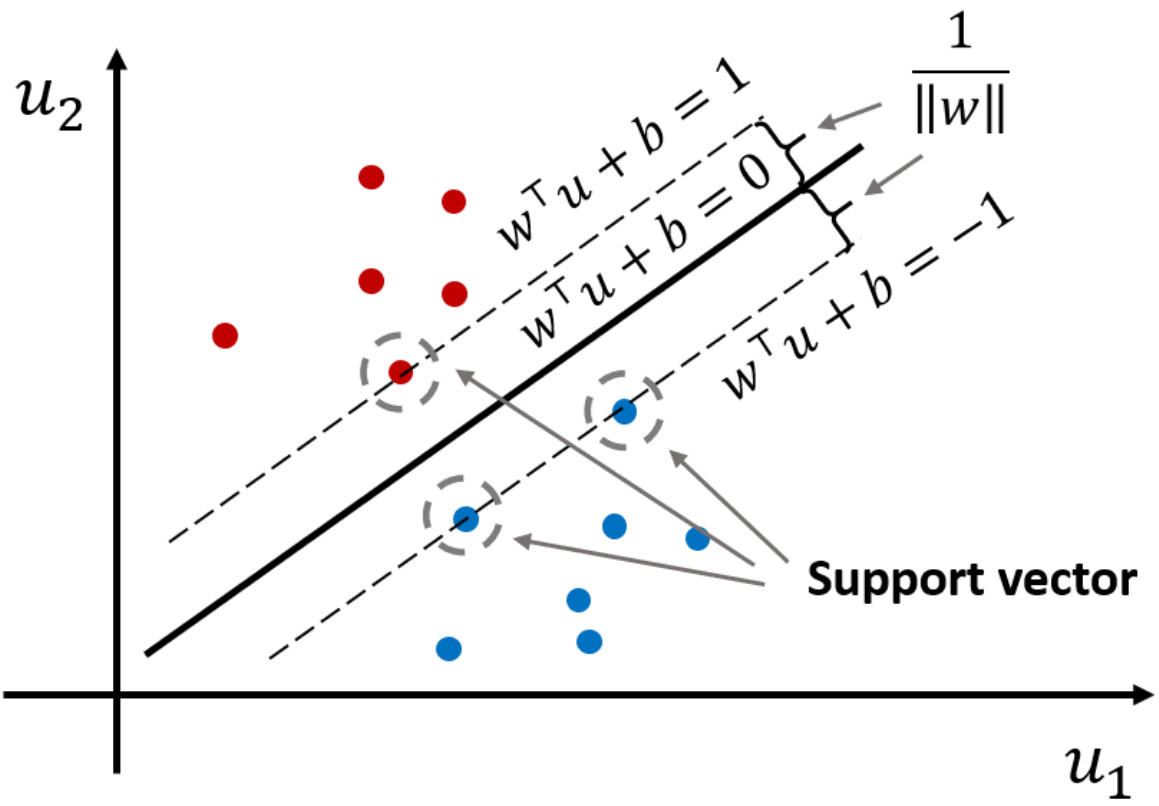
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn import datasets

# import some data to play with
iris = datasets.load_iris()
X = iris.data[:, :2]  # we only take the first two features.
Y = iris.target

logreg = LogisticRegression(C=1e5)

# Create an instance of Logistic Regression Classifier and fit the data.
logreg.fit(X, Y)

# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, x_max]x[y_min, y_max].
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
h = .02  # step size in the mesh
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
Z = logreg.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.figure(1, figsize=(4, 3))
plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)

# Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=Y, edgecolors='k', cmap=plt.cm.Paired)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')

plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.xticks(())
plt.yticks(())

plt.show()
```



Support Vector Machine for classification

$$\begin{aligned} &\max_{\mathbf{w},b} \frac{1}{\|\mathbf{w}\|_2} \\ &s.t. \quad v^{(i)}[\mathbf{w}^\top \mathbf{u}^{(i)} + b] \geq 1, \\ &\qquad i = 1, \ldots, N \end{aligned}$$

where $(\mathbf{u}^{(i)}, v^{(i)})$ are datapoints. $v \in \{-1, +1\}$



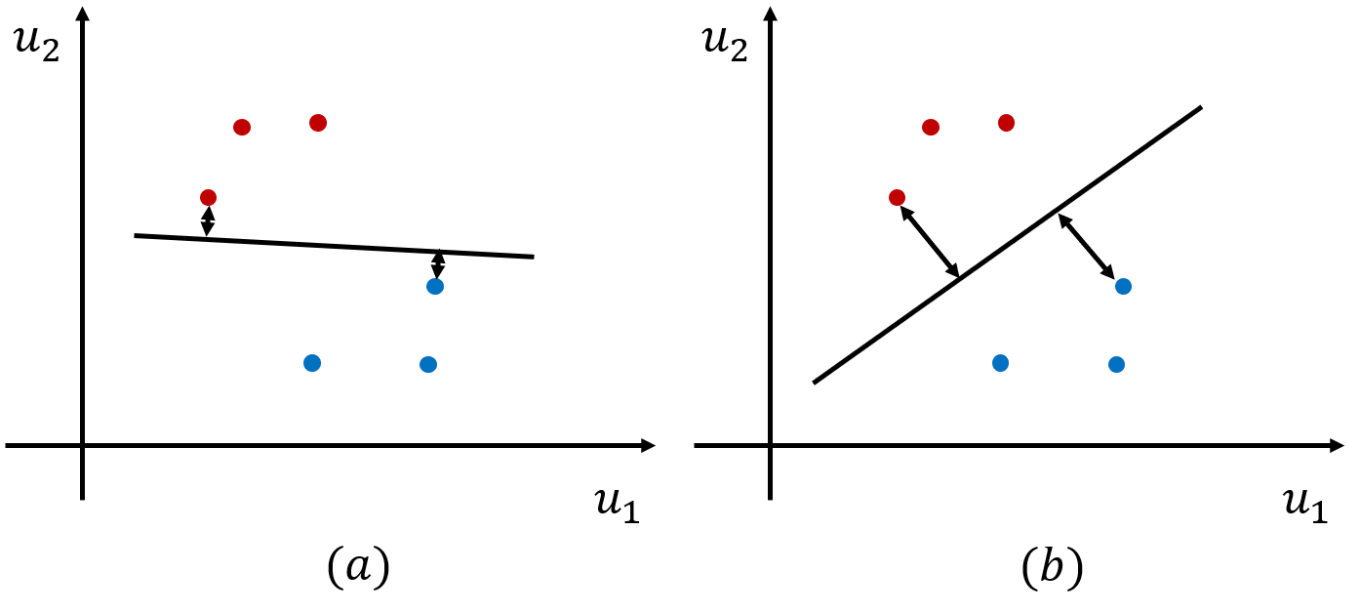
Consider binary classification problem with N training examples $\{\mathbf{u}^{(i)}, v^{(i)}\}_{i=0}^{N}$, where $u^{(i)} \in \mathbb{R}^n$, $v^{(i)} \in \{-1, 1\}$.
Assume that the samples can be seperated by a hyperplane. i.e., there exists a linear function of $\mathbf{u}$

$$h_{\mathbf{w},b}(u) = b + w_1 u_1 + w_2 u_2 + \ldots + w_n u_n$$

such that for all $i = 1, \ldots, N$

$$h_{\mathbf{w},b}(\mathbf{u}^{(i)}) \begin{cases} > 0, \text{if } v^{(i)} = 1 \\ < 0, \text{if } v^{(i)} = -1 \end{cases}$$

We shall consider what is a "good" separating hyperplane.



The distance between the seperating hyperplane and the nearest data point can serve as an evaluation. This distance is called the margin,

$$d(\mathbf{w}, b) = \min_{i=1,\ldots,N} d^i,$$

where d^i is the ***distance*** from the data i to the hyerplane

$$d^i = \begin{cases} \frac{\mathbf{w}^\top u^{(i)}+b}{\|w\|}, \text{if } \upsilon^{(i)} = 1 \\ -\frac{\mathbf{w}^\top u^{(i)}+b}{\|w\|}, \text{if } \upsilon^{(i)} = -1 \end{cases}$$
$$= \frac{\upsilon^{(i)}[\mathbf{w}^\top \mathbf{u}^{(i)} + b]}{\|w\|}$$

A larger margin implies that the seperating hyperplane can distinguish the two different classes more significantly. We aim at maximizing

$$\max_{\mathbf{w},b} d(\mathbf{w}, b) = \frac{\min_{i=1,\ldots,N} \upsilon^{(i)}[\mathbf{w}^\top \mathbf{u}^{(i)} + b]}{\|w\|}$$

Let $\frac{\min_{i=1,\ldots,N} \upsilon^{(i)}[\mathbf{w}^\top \mathbf{u}^{(i)}+b]}{\|w\|} = r$, obviously, $\upsilon^{(i)}[\mathbf{w}^\top \mathbf{u}^{(i)} + b] \geq r,\ i = 1, \ldots, N$.
The problem can be rewritten as

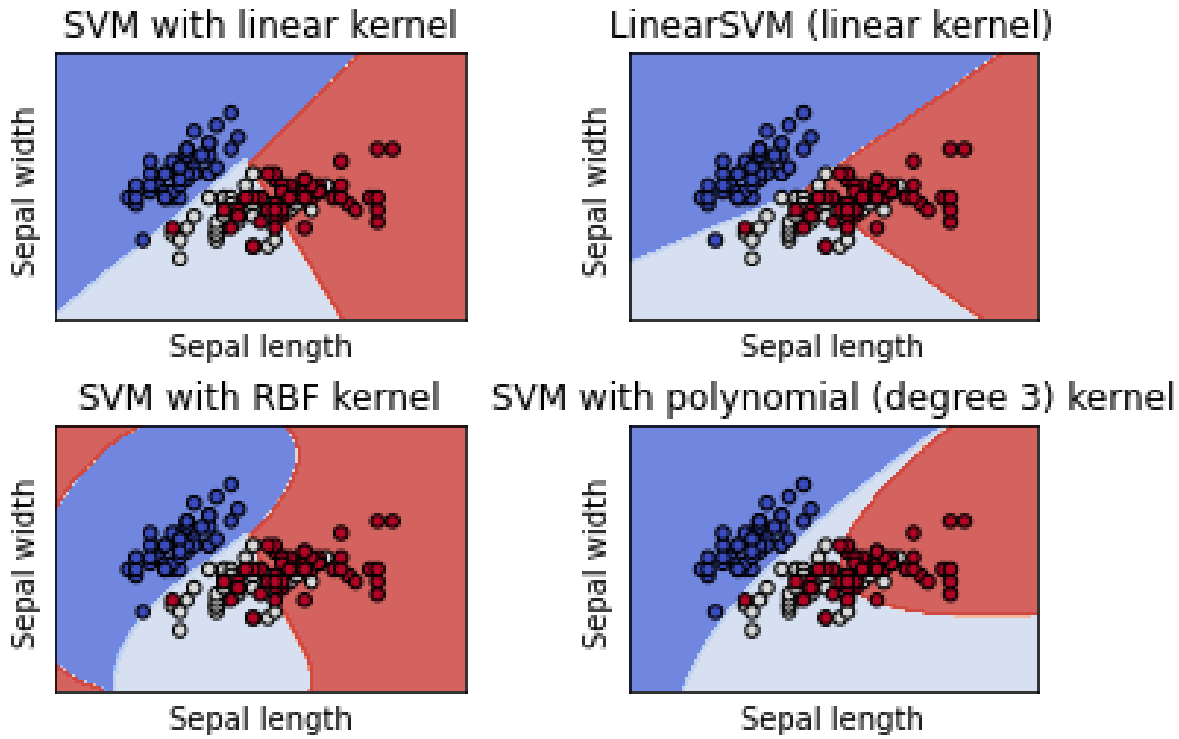
$$\max_{\mathbf{w},b,r} \frac{r}{\|w\|}$$
$$s.t.\ \ \upsilon^{(i)}[\mathbf{w}^\top \mathbf{u}^{(i)} + b] \geq r, i = 1, \ldots, N$$

Assume $r = 1$, this will not change the optimal value of the problem. The problem becomes the **constrained** optimzation (*quadratic programming*)

$$\min_{\mathbf{w},b} \|w\|^2$$

$$s.t. \quad v^{(i)}[\mathbf{w}^\top \mathbf{u}^{(i)} + b] \geq 1, i = 1, \ldots, N$$

Application of SVM to iris dataset



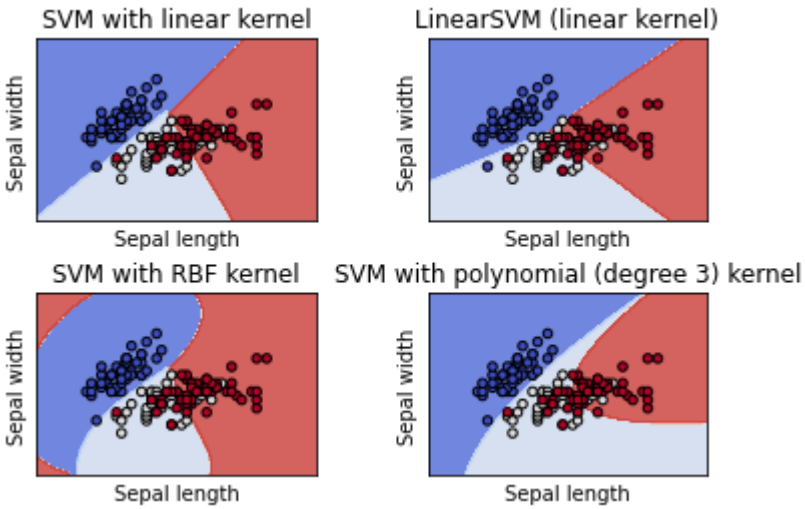
In [9]:

```
#https://scikit-learn.org/stable/auto_examples/svm/plot_iris_svc.html#sphx-glr-auto-examples-svm-plot-iris-svc-py
```

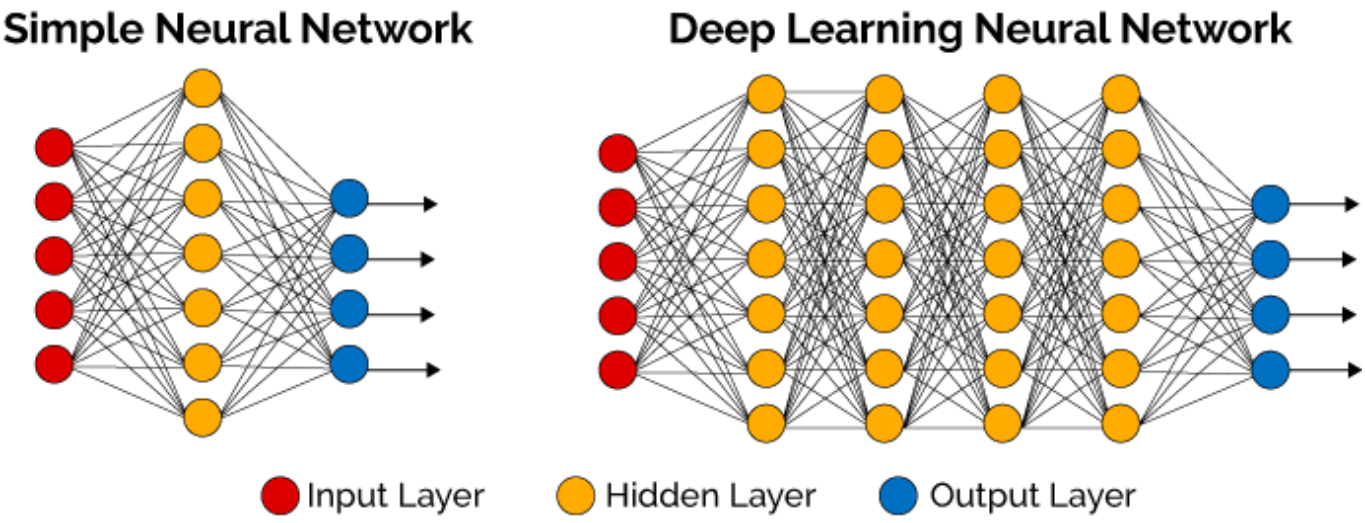
In [10]:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets


def make_meshgrid(x, y, h=.02):
    """Create a mesh of points to plot in

    Parameters
    ----------
    x: data to base x-axis meshgrid on
    y: data to base y-axis meshgrid on
    h: stepsize for meshgrid, optional

    Returns
    -------
    xx, yy : ndarray
    """
    x_min, x_max = x.min() - 1, x.max() + 1
    y_min, y_max = y.min() - 1, y.max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    return xx, yy


def plot_contours(ax, clf, xx, yy, **params):
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    out = ax.contourf(xx, yy, Z, **params)
    return out


# import some data to play with
iris = datasets.load_iris()
# Take the first two features. We could avoid this by using a two-dim dataset
X = iris.data[:, :2]
y = iris.target

# we create an instance of SVM and fit out data. We do not scale our
# data since we want to plot the support vectors
C = 1.0  # SVM regularization parameter
models = (svm.SVC(kernel='linear', C=C),
          svm.LinearSVC(C=C, max_iter=10000),
          svm.SVC(kernel='rbf', gamma=0.7, C=C),
          svm.SVC(kernel='poly', degree=3, gamma='auto', C=C))
models = (clf.fit(X, y) for clf in models)

# title for the plots
titles = ('SVM with linear kernel',
          'LinearSVM (linear kernel)',
          'SVM with RBF kernel',
          'SVM with polynomial (degree 3) kernel')

# Set-up 2x2 grid for plotting.
fig, sub = plt.subplots(2, 2)
plt.subplots_adjust(wspace=0.4, hspace=0.4)

X0, X1 = X[:, 0], X[:, 1]
xx, yy = make_meshgrid(X0, X1)
```

```
for clf, title, ax in zip(models, titles, sub.flatten()):
    plot_contours(ax, clf, xx, yy,
                  cmap=plt.cm.coolwarm, alpha=0.8)
    ax.scatter(X0, X1, c=y, cmap=plt.cm.coolwarm, s=20, edgecolors='k')
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xlabel('Sepal length')
    ax.set_ylabel('Sepal width')
    ax.set_xticks(())
    ax.set_yticks(())
    ax.set_title(title)

plt.show()
```



Neural Network



One-hidden-layer neural network

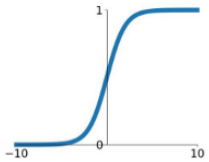
Use the family of nonlinear function to approximate

$$f(\mathbf{x}; \mathbf{w}, b) = \sigma(\mathbf{w}^\top \mathbf{x} + b)$$

- $\sigma(\cdot)$: activation function

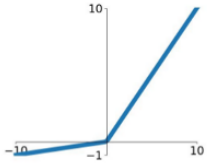
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



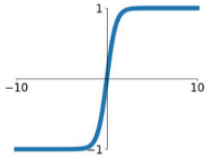
Leaky ReLU

$\max(0.1x, x)$



tanh

$\tanh(x)$

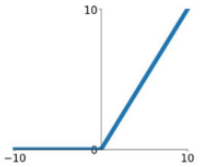


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

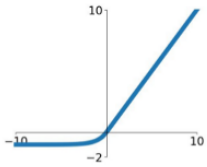
ReLU

$\max(0, x)$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



- The class of functions in approximation:

$$y \approx \sum_{i=1}^{m} c_i f(\mathbf{x}; \mathbf{w}_i, b_i)$$

- The (emperical) mean squared loss

$$L(\theta) = \sum_{j=1}^{N} \left| y_j - \sum_{i=1}^{m} c_i f(\mathbf{x}_j; \mathbf{w}_i, b_i) \right|^2$$

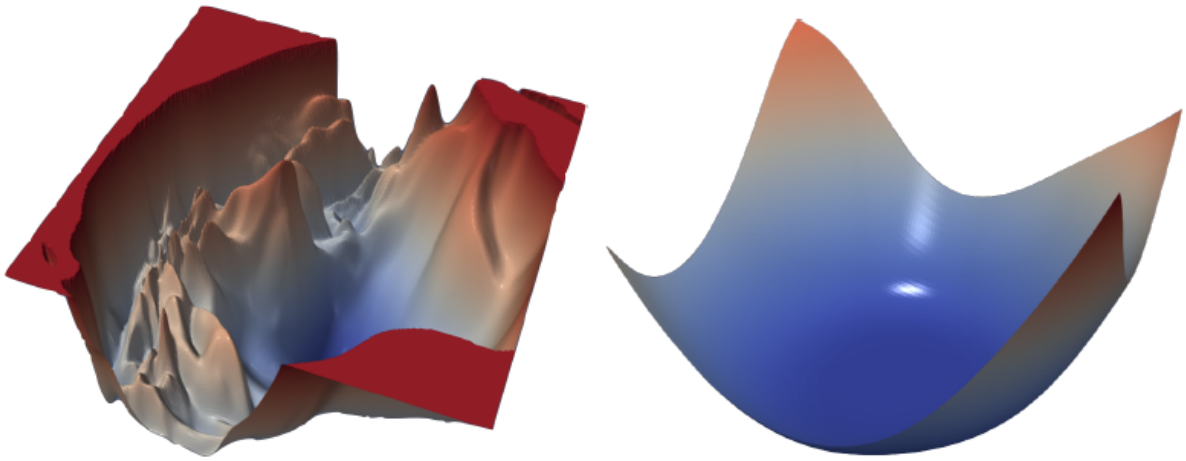
  where all parameters are summarized as a variable $\theta = (c_i, \mathbf{w}_i, b_i)$, $i = 1, \ldots, m$. $\mathbf{w}_i \in \mathbb{R}^n$.
- There are $m * (2 + n)$ unknowns.

Exercise [Python programming]

- σ is an activation function (sigmoid or RELU)
- Fit the Runge function function $y = \frac{1}{1+25x^2}$ at N equidistant points with $x_i \in [-1, 1]$.
- Define the loss function: $L(\theta)$
- Compute the gradient of the loss function
- The input is a vector with the size $m * (n + 2) = 3m$
- Refer to the format of the MATLAB subroutine 'fminunc'.

DNN: Deep Neural Network

- Multple layers
- Foundation of deep learning
- Non-convex optimization
- Challenging to train in practice
- Alchemy?



(a) ResNet-110, no skip connections (b) DenseNet, 121 layers

Loss surfaces of ResNet-110-noshort and DenseNet for CIFAR-10.
Source:https://arxiv.org/pdf/1712.09913.pdf (https://arxiv.org/pdf/1712.09913.pdf).

Training Deep neural networks is subtle and tricky in practice. Lots of experience and tricks from engineering are essential for the suceess of training.

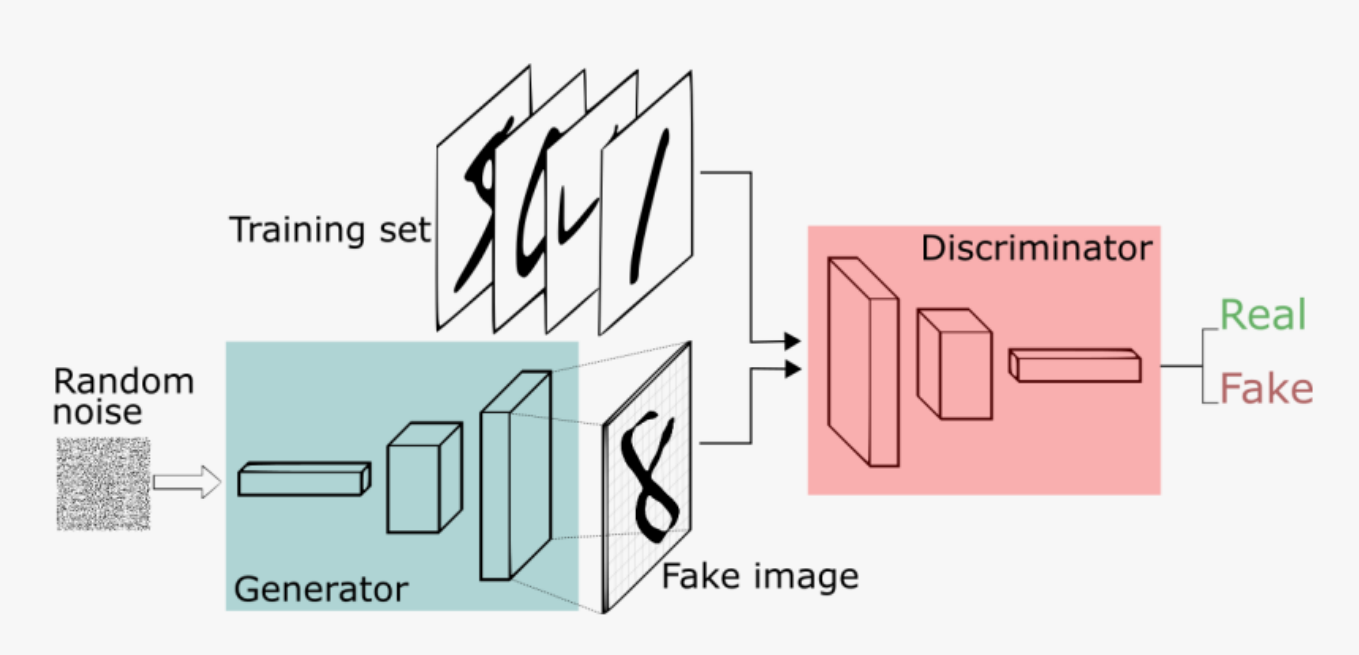
In [11]:

```
import torch
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt


transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

################################################################################3
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
################################################################################
###
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
################################################################################
###
for epoch in range(2):  # loop over the dataset multiple times
```

```
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
########################################################################
##
dataiter = iter(testloader)
images, labels = dataiter.next()

# print images
imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))
net = Net()
net.load_state_dict(torch.load(PATH))
outputs = net(images)
_, predicted = torch.max(outputs, 1)

print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
                              for j in range(4)))
```

```
---------------------------------------------------------------------
-------
ModuleNotFoundError                       Traceback (most recent cal
l last)
<ipython-input-11-f68010b764ab> in <module>
----> 1 import torch
      2 import torchvision
      3 import torchvision.transforms as transforms
      4 import matplotlib.pyplot as plt
      5

ModuleNotFoundError: No module named 'torch'
```

Generative Adversarial Network (GAN)



Generative Adversarial Network (GAN)

$$\min_G \max_D V(G,D) = \int_x p_{data}(x)\log(D(x))dx + \int_z p_z(z)\log(1-D(g(z)))dz$$

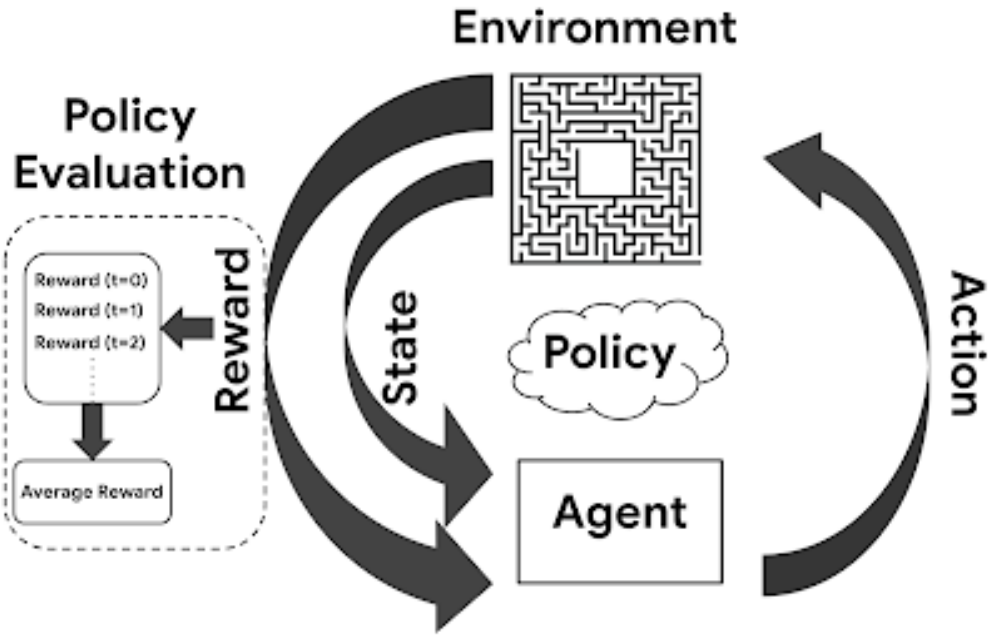
$$V(G,D) \approx \sum_j \log(D(x_j)) + \sum_j \log(1-D(g(z_j)))$$

- $x \in X$ is the real data with p_{data} as its probability density function (unknown, only data x_j available)
- $z \in Z$ is the latent variable with p_z as its probability density function (known and simple to sample z_j),
- $D : X \to [0,1]$ is the discriminator function and $G : Z \to X$ is the generator function.
- This is a min-max problem.

Online GAN training websites

- https://poloclub.github.io/ganlab/ (https://poloclub.github.io/ganlab/)
- https://reiinakano.com/gan-playground/ (https://reiinakano.com/gan-playground/)

Reinforcement Learning



Optimal Control (Infinite horizaon stochastic control; Markov Decision Process):

$$\min_{\pi} V_{\pi}(x_0) = \mathbb{E} \sum_{t=0}^{\infty} \alpha^t r(x_t, \pi(x_t))$$

Bellman equation of Optimality

$$V^*(i) = \min_{u \in U(i)} \sum_{j} p_u(r, j|i)[r + \alpha V^*(j)]$$

In [ ]:

```
import gym
env = gym.make("CartPole-v1")
observation = env.reset()
for _ in range(1000):
  env.render()
  action = env.action_space.sample() # your agent here (this takes random action
s)
  observation, reward, done, info = env.step(action)

  if done:
    observation = env.reset()
env.close()
```

Summary

- All machine learning algorithms boil down to various optimizaton problems.
- They share a common challenge that the objective function is the sum of almost ``identical'' individual functions

$$L(\theta) = \sum_i L_i(\theta)$$

- The non-smooth regularization (like ℓ_1) $R(\theta)$ imposes additional difficulty
- Most machine learning packages nicely wrap the state-of-the-art optimization algorithms into **black-box**:

```
model.fit()
```

- This course examines the basic theories and algorithms behind these methods, which have been developed recently.

Stochastic Optimization and Online Learning

- The statistical learning theory and the convential optimzation method fail at addressing one the key features of the new massive data and the dynamic stream of data.
- Stochastic optimization use only a single or a smal fraction of the total dataset at each iteration step.
- Online learning is more general and even has no probabilistic assumption of the data $Z_1, \ldots, Z_N, \ldots$.