Lecture 18 Part 1 - Linear Discriminant Analysis: Logistic Regression

Final Discussions about Fisher's LDA

Limitations of LDA:

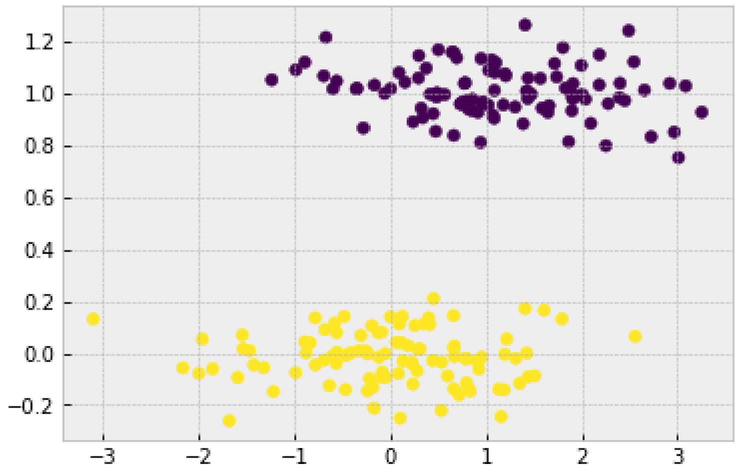
1. LDA produces at most $C-1$ feature projections, where C is the number of classes.
2. If the classification error estimates establish that more features are needed, some other method must be employed to provide those additional features.
3. LDA is a parametric method (it assumes unimodal Gaussian likelihoods).
4. If the distributions are significantly non-Gaussian, the LDA projections may not preserve complex structure in the data needed for classification.
5. LDA will also fail if discriminatory information is not in the mean but in the variance of the data.

A popular variant of LDA are the **Multi-Layer Perceptrons** (or MLPs).

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('bmh')
```

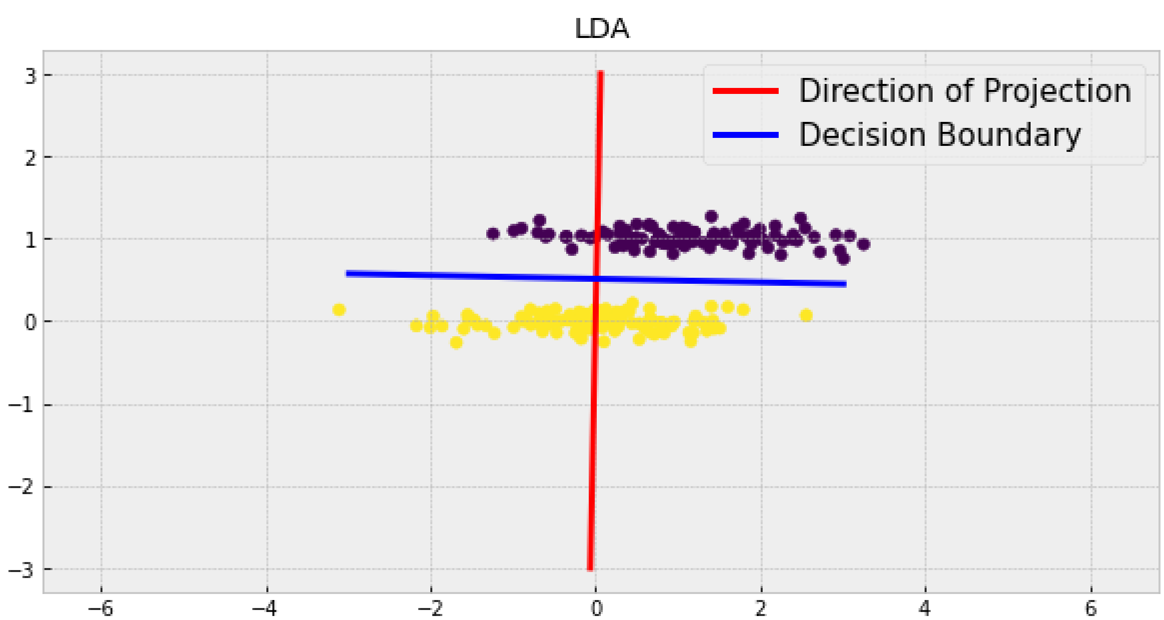
In [2]:
```
def fisherDiscriminant(data,t):
    data1 = data[t==0,:]
    data2 = data[t==1,:]
    mean1 = np.atleast_2d(np.mean(data1,0))
    mean2 = np.atleast_2d(np.mean(data2,0))
    Sw1 = np.dstack([(data1[i,:]-mean1).T@(data1[i,:]-mean1) for i in range(data1.shap
    Sw2 = np.dstack([(data2[i,:]-mean2).T@(data2[i,:]-mean2) for i in range(data2.shap
    Sw = np.sum(Sw1,2) + np.sum(Sw2,2)
    w = np.linalg.inv(Sw)@(mean2 - mean1).T
    w = w/np.linalg.norm(w)
    data_t = data@w
    return w, data_t

def discriminant(data, labels, v):
    v_perp = np.array([v[1], -v[0]])
    b = ((np.mean(data[labels==0,:],axis=0)+np.mean(data[labels==1,:],axis=0))/2)@v
    lambda_vec = np.linspace(-3,3,len(data))
    v_line = lambda_vec * v
    decision_boundary = b * v + lambda_vec * v_perp
    return v_line, decision_boundary

# Generate Synthetic Data
N1 = 100 #number of points for class1
```

```
N2 = 100 #number of points for class0
covM = [1,0.01]*np.eye(2) # covariance matrix
data = np.random.multivariate_normal([0,0], covM, N1) #generate points  for class 1
X = np.vstack((data, np.random.multivariate_normal([1,1], covM, N2))) #generate points
labels = np.hstack((np.ones(N1),np.zeros(N2)))

plt.scatter(X[:,0],X[:,1],c=labels); plt.show();
```



In [3]:

```
v, Y = fisherDiscriminant(X,labels)

plt.figure(figsize=(10,5))
plt.scatter(X[:,0],X[:,1],c=labels)

v_line, decision_boundary = discriminant(X, labels, v);

plt.plot(v_line[0], v_line[1], 'red', linewidth=3, label='Direction of Projection')
plt.plot(decision_boundary[0,:], decision_boundary[1,:],'blue',linewidth=3, label='Dec
plt.title('LDA'); plt.axis('equal'); plt.legend(loc='best',fontsize=15);
```



Limitations of LDA:

1. LDA produces at most $K-1$ feature projections, where K is the number of classes.
2. If the classification error estimates establish that more features are needed, some other method must be employed to provide those additional features.
3. LDA is a parametric method (it assumes unimodal Gaussian likelihoods).
4. If the distributions are significantly non-Gaussian, the LDA projections may not preserve complex structure in the data needed for classification.
5. LDA will also fail if discriminatory information is not in the mean but in the variance of the data.

A popular variant of LDA are the **Multi-Layer Perceptrons** (or MLPs).

Least Squares for Classification

Not advisable

In [4]:
```
from IPython.display import Image
Image('figures/Figure4.4.png', width=700)
```

Out[4]:

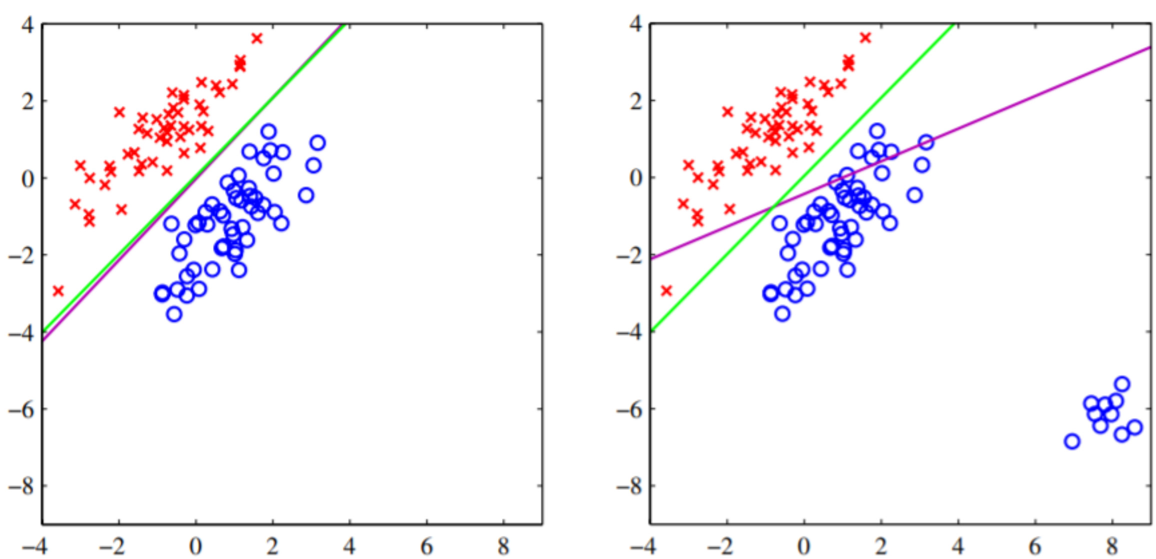


Figure 4.4 The left plot shows data from two classes, denoted by red crosses and blue circles, together with the decision boundary found by least squares (magenta curve) and also by the logistic regression model (green curve), which is discussed later in Section 4.3.2. The right-hand plot shows the corresponding results obtained when extra data points are added at the bottom left of the diagram, showing that least squares is highly sensitive to outliers, unlike logistic regression.

More discussions on board notes.

Logistic Regression

Let's import a classification dataset: the iris dataset.

In [8]:
```
from sklearn.datasets import load_iris
```

In [9]:
```
iris = load_iris(as_frame=False)
```

In [10]:
```
iris.keys()
```

Out[10]:
```
dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names', 'file
name', 'data_module'])
```

Datasets loaded by `sklearn` generally have a similar disctionary structure, including the following:

- A `DESCR` key describing the dataset
- A `data` key containing an array with one row per instance and one column per feature
- A `target` key containing an array with the labels

In [11]:
```
print(iris.DESCR)
```

```
.. _iris_dataset:

Iris plants dataset
--------------------

**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica

    :Summary Statistics:

    ============== ==== ==== ======= ===== ====================
                    Min  Max   Mean    SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length:   4.3  7.9   5.84   0.83    0.7826
    sepal width:    2.0  4.4   3.05   0.43   -0.4194
    petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)
    petal width:    0.1  2.5   1.20   0.76    0.9565  (high!)
    ============== ==== ==== ======= ===== ====================

    :Missing Attribute Values: None
    :Class Distribution: 33.3% for each of 3 classes.
    :Creator: R.A. Fisher
    :Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
    :Date: July, 1988

The famous Iris database, first used by Sir R.A. Fisher. The dataset is taken
from Fisher's paper. Note that it's the same as in R, but not as in the UCI
Machine Learning Repository, which has two wrong data points.

This is perhaps the best known database to be found in the
pattern recognition literature.  Fisher's paper is a classic in the field and
is referenced frequently to this day.  (See Duda & Hart, for example.)  The
data set contains 3 classes of 50 instances each, where each class refers to a
type of iris plant.  One class is linearly separable from the other 2; the
latter are NOT linearly separable from each other.

.. topic:: References

   - Fisher, R.A. "The use of multiple measurements in taxonomic problems"
     Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to
     Mathematical Statistics" (John Wiley, NY, 1950).
   - Duda, R.O., & Hart, P.E. (1973) Pattern Classification and Scene Analysis.
     (Q327.D83) John Wiley & Sons.  ISBN 0-471-22361-1.  See page 218.
   - Dasarathy, B.V. (1980) "Nosing Around the Neighborhood: A New System
     Structure and Classification Rule for Recognition in Partially Exposed
     Environments".  IEEE Transactions on Pattern Analysis and Machine
     Intelligence, Vol. PAMI-2, No. 1, 67-71.
   - Gates, G.W. (1972) "The Reduced Nearest Neighbor Rule".  IEEE Transactions
     on Information Theory, May 1972, 431-433.
```

```
   - See also: 1988 MLC Proceedings, 54-64.  Cheeseman et al"s AUTOCLASS II
     conceptual clustering system finds 3 classes in the data.
   - Many, many more ...
```

In [ ]:
```python
# iris.data
```

In [12]:
```python
iris.data.shape, iris.target.shape
```

Out[12]:
```
((150, 4), (150,))
```

In [13]:
```python
iris.feature_names
```

Out[13]:
```
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']
```

There are 150 samples, and each sample has 4 features (sepal length, sepal width, petal length and petal width).

In [14]:
```python
np.unique(iris.target, return_counts=True)
```

Out[14]:
```
(array([0, 1, 2]), array([50, 50, 50], dtype=int64))
```

In [15]:
```python
iris.target_names
```

Out[15]:
```
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```

Each sample is classified as one of 3 classes: 0 is setosa, 1 is versicolor, and 2 is virginica.

For visualization purposes, let's consider only features **petal length** and **petal width**.

In [16]:
```python
X = iris.data[:,2:]

X.shape
```

Out[16]:
```
(150, 2)
```

Training a Binary Classifier

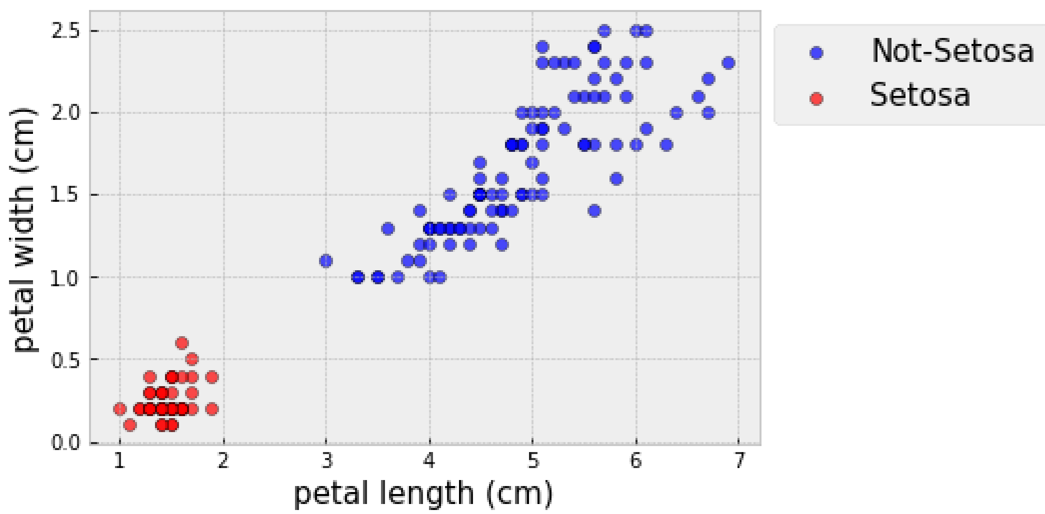
Let's simplify the problem for now and only try to identify one iris species - for example, the **class setosa**. This "setosa-detector" will be an example of a binary classifier, capable of distinguishing between just two classes, setosa and not-setosa. Let's create the target vectors for this classification task:

In [17]:
```python
t = 1*(iris.target == 0)
```

In [18]:
```python
t
```

Out[18]:
```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 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, 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, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

In [19]:
```python
plt.scatter(X[t==0,0], X[t==0,1], c='b', edgecolors='k', alpha=0.7, label='Not-Setosa'
plt.scatter(X[t==1,0], X[t==1,1], c='r', edgecolors='k', alpha=0.7, label='Setosa')
plt.xlabel(iris.feature_names[2], size=15)
plt.ylabel(iris.feature_names[3], size=15)
plt.legend(bbox_to_anchor=(1,1),fontsize=15);
```



Logistic Regression, also called *Logit* Regression or Logistic Discrimination

Note that, despite of its name, logistic regression is a model for classification, not regression.

The **logistic discriminant** is a linear model for binary classification that can be extended to multi-class classification using the one-vs-all approach.

Logistic regression is a **probabilistic discriminative classifier**.

In logistic logistic discrimination, we do not model the class-conditional densities, $p(x|C_i)$, but rather their ratio. Let us assume we are working with a two-class problem and assume that the log likelihood ratio (or **odds ratio**) is linear:

$$\log \frac{P(x|C_1)}{P(x|C_2)} = \mathbf{w}^T x + b$$

- This is only true, if the classes are Gaussian-distributed!

- But logistic discrimination has a wider scope of applicability; for example, x may be composed of discrete attributes or may be a mixture of continuous and discrete attributes. Using Bayes' rule, we have:

$$\begin{aligned}\text{logit } P(C_1|x) &= \log \frac{P(C_1|x)}{1-P(C_1|x)} \\ &= \log \frac{P(x|C_1)}{P(x|C_2)} + \log \frac{P(C_1)}{P(C_2)} \\ &= \mathbf{w}^T x + w_0\end{aligned}$$

where $w_0 = b + \log \frac{P(C_1)}{P(C_2)}$.

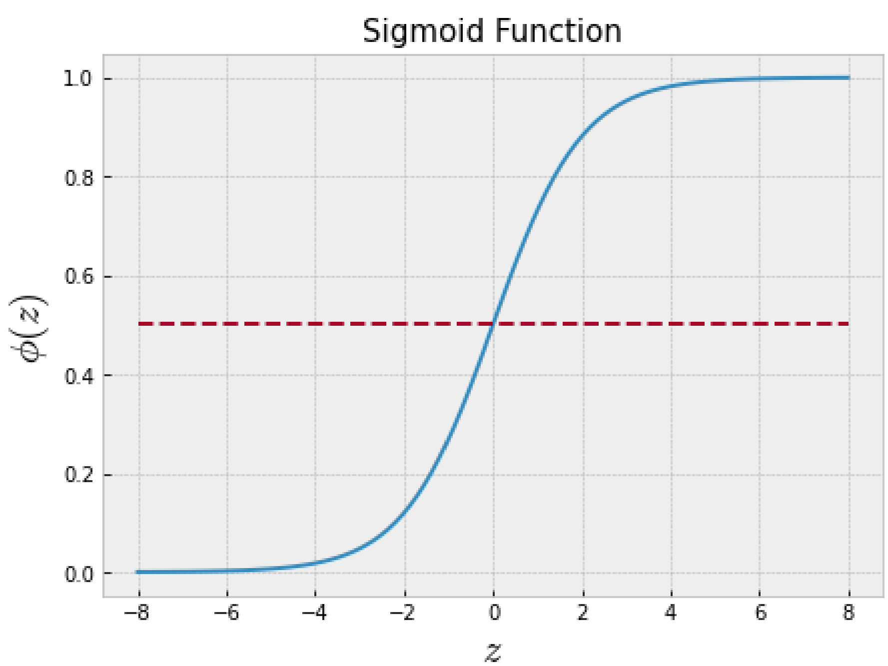
Rearranging terms, we get the probabilistic classification:

$$y = P(C_1|x) = \frac{1}{1+\exp(-(\mathbf{w}^T x + w_0))}$$

This is the sigmoid function:

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

In [5]:
```
z = np.linspace(-8,8,100)

plt.figure(figsize=(7,5))
plt.plot(z, 1/(1+np.exp(-z)))
plt.plot(z, [0.5]*len(z),'--')
plt.xlabel('$z$',size=20); plt.ylabel('$\phi(z)$',size=20);
plt.title('Sigmoid Function',size=15);
```



- We can see that $\phi(z) \to 1$ as $z \to \infty$, since $\exp(-z)$ becomes very small for large values of z.

- Similarly, $\phi(z) \to 0$ as $z \to -\infty$ as the result of an increasingly large denominator.

Thus, we conclude that this sigmoid function takes real number values as input and transforms them to values in the range $[0, 1]$ with an intercept at $\phi(z) = 0.5$.

This is the same as using a sigmoid function as the **activation function** in the perceptron diagram.

Thus the output of the sigmoid function is then interpreted as the probability of particular sample belonging to C_1, given its features x parameterized by the weights w.

- For example, if we compute $\phi(z) = 0.8$ for a particular sample, it means that the chance that this sample is in C_1 is 80%.

The predicted probability can then simply be converted into a binary outcome via a quantizer (unit step function):

$$\begin{aligned} \hat{t} = y &= \begin{cases} 1, & \phi(z) \geq 0.5 \\ 0, & \text{otherwise} \end{cases} \\ &= \begin{cases} 1, & \phi(\mathbf{w}^T x + w_0) \geq 0.5 \\ 0, & \text{otherwise} \end{cases} \end{aligned}$$

or,

$$\begin{aligned} \hat{t} = y &= \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases} \\ &= \begin{cases} 1, & \mathbf{w}^T x + w_0 \geq 0 \\ 0, & \mathbf{w}^T x + w_0 < 0 \end{cases} \end{aligned}$$

The Objective Function

Let $\{(x_i, t_i)\}_{i=1}^N$ be the set of input samples and its class labels, where $t_i \in \{0, 1\}$. Assuming the data samples are i.i.d., we can build the observed data likelihood:

$$\mathcal{L}^0 = \prod_{i=1}^{N} P(y_i | x_i; \mathbf{w}) = \phi(z_i)^{t_i} (1 - \phi(z_i))^{1-t_i}$$

We can apply the "trick" (log-likelihood) to build the data likelihood":

$$\mathcal{L} = \sum_{i=1}^{N} t_i \log \phi(z_i) + (1 - t_i) \log(1 - \phi(z_i))$$

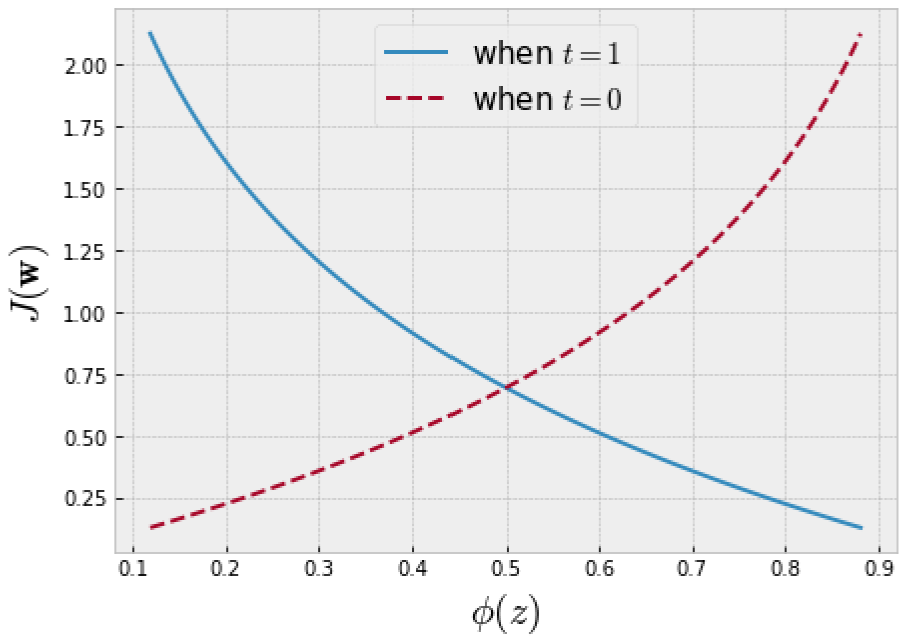
where $\phi(z) = \frac{1}{1+\exp(-z)}$ and $z_i = \mathbf{w}^T x_i + w_0$.

We want to maximize this likelihood to the data, or we can also write it as a minimization optimization:

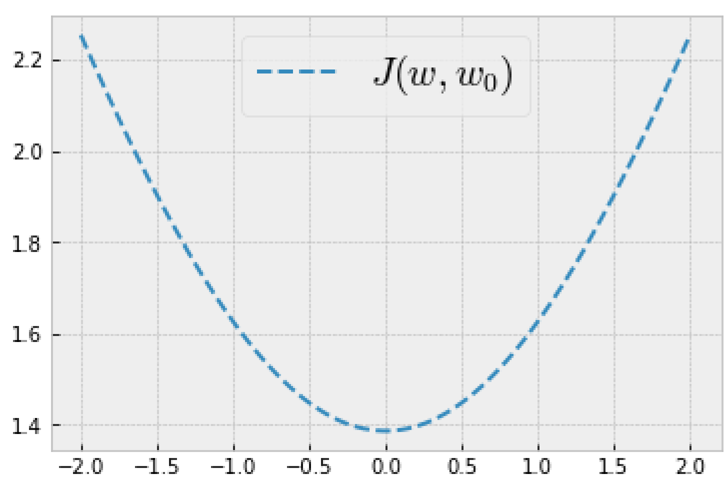
$$J(\mathbf{w}, w_0) = \sum_{i=1}^{N} -t_i \log \phi(z_i) - (1-t_i)\log(1-\phi(z_i))$$

This objective function is also known as **cross-entropy**.

In [6]:
```
z = np.linspace(-2,2,100)

phi = lambda z: 1/(1+np.exp(-z))

plt.figure(figsize=(7,5))
plt.plot(phi(z), -np.log(phi(z)),label='when $t=1$')
plt.plot(phi(z), -np.log(1-phi(z)),'--',label='when $t=0$')
plt.legend(fontsize=15); plt.xlabel('$\phi(z)$',size=20)
plt.ylabel('$J(\mathbf{w})$',size=20);
```



In [7]:
```
plt.plot(z, -np.log(phi(z))-np.log(1-phi(z)),'--',label='$J(w,w_0)$')
plt.legend(fontsize=20);
```



We can see that the cost approaches 0 if we correctly predict that a sample belongs to class 1. Similarly, we can see on the y axis that the cost also approaches 0 if we correctly predict class 0. However, if the prediction is wrong, the cost goes towards infinity: we penalize wrong predictions with an increasingly larger cost.

As we do not have the global *picture* of what the objective function, $J(\mathbf{w})$, we apply a search method to navigate through the objective function to find the *local optima* starting from an initial value, namely, **gradient descent**.

$$\mathbf{w}^{(t+1)} \longleftarrow \mathbf{w}^{(t)} - \eta \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}^{(t)}}$$

$$\mathbf{w_0}^{(t+1)} \longleftarrow \mathbf{w_0}^{(t)} - \eta \frac{\partial J(\mathbf{w})}{\partial \mathbf{w_0}^{(t)}}$$

where η is the learning rate (or step size).

$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}^{(t)}} = \sum_{i=1}^{N} -t_i \frac{1}{\phi(z_i)} \frac{\partial \phi(z_i)}{\partial z_i} \frac{\partial z_i}{\partial \mathbf{w}} - (1-t_i) \frac{1}{1-\phi(z_i)} \left(- \frac{\partial \phi(z_i)}{\partial z_i} \frac{\partial z_i}{\partial \mathbf{w}} \right)$$

where $\frac{\partial \phi(z_i)}{\partial z_i} = \phi'(z_i)$ and $\frac{\partial z_i}{\partial \mathbf{w}} = x_i$. Substituting:

$$\begin{aligned} \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}^{(t)}} &= \sum_{i=1}^{N} -t_i \frac{\phi'(z_i)}{\phi(z_i)} x_i + (1-t_i) \frac{\phi'(z_i)}{1-\phi(z_i)} x_i \\ &= \sum_{i=1}^{N} \left(\frac{t_i}{\phi(z_i)} - \frac{1-t_i}{1-\phi(z_i)} \right) \phi'(z_i) x_i \end{aligned}$$

where $\phi'(z_i) = \phi(z_i)(1-\phi(z_i))$, then applying some substitutions we have:

$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}^{(t)}} = \sum_{i=1}^{N} (t_i - y_i) x_i$$

and, similarly,

$$\frac{\partial J(\mathbf{w})}{\partial w_0^{(t)}} = \sum_{i=1}^{N} (t_i - y_i)$$

Finally,

$$\mathbf{w}^{(t+1)} \longleftarrow \mathbf{w}^{(t)} - \eta \sum_{i=1}^{N} (t_i - y_i) x_i$$

$$\mathbf{w_0}^{(t+1)} \longleftarrow \mathbf{w_0}^{(t)} - \eta \sum_{i=1}^{N} (t_i - y_i)$$

Initialization: it is best to initialize $\mathbf{w}$ with random values close to 0; generally they are drawn uniformly from the interval $[-0.01, 0.01]$.

- The reason for this is that if the initial $\mathbf{w}$ are large in magnitude, the weighted sum may also be large and may saturate the sigmoid.
- If the initial weights are close to 0, the sum will stay in the middle region where the derivative is nonzero and an update can take place.
- If the weighted sum is large in magnitude (smaller than −5 or larger than +5), the derivative of the sigmoid will be almost 0 and weights will not be updated.
  - When we stack up a lot of these perceptrons in layers and add a few layers, this "close to 0" gradient will lead to a phenomenon in neural networks known as the **vanishing gradient**.

In [20]:
```
from sklearn.linear_model import LogisticRegression

LogisticRegression?
```

In [22]:
```
log_reg = LogisticRegression(penalty='none')

log_reg
```

Out[22]:
```
LogisticRegression(penalty='none')
```

In [23]:
```
# Training

log_reg.fit(X, t)
```

Out[23]:
```
LogisticRegression(penalty='none')
```

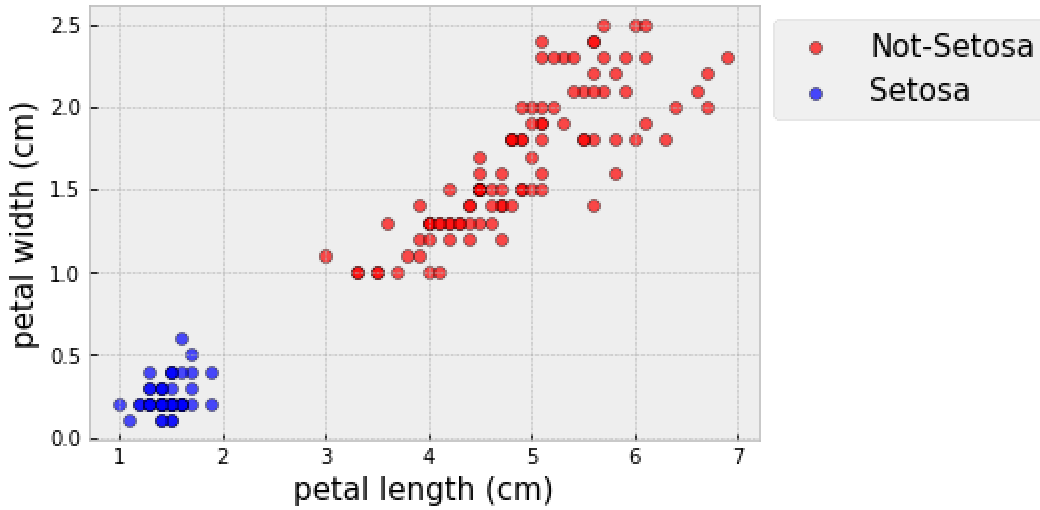
In [24]:
```
# Attribute intercept = w0

log_reg.intercept_
```

Out[24]:
```
array([41.42457221])
```

In [25]:
```
log_reg.coef_
```

Out[25]:
```
array([[-11.46429691, -16.37221147]])
```

In [26]:
```
plt.scatter(X[t==0,0], X[t==0,1], c='r', edgecolors='k', alpha=0.7, label='Not-Setosa'
plt.scatter(X[t==1,0], X[t==1,1], c='b', edgecolors='k', alpha=0.7, label='Setosa')
plt.xlabel(iris.feature_names[2], size=15)
plt.ylabel(iris.feature_names[3], size=15)
plt.legend(bbox_to_anchor=(1,1),fontsize=15);
```



In [27]:
```
# generate synthetic data in the 2-D feature space

xx, yy = np.meshgrid(np.arange(0, 8, 0.01),
                     np.arange(-1, 3, 0.01))

mesh = np.vstack((xx.ravel(), yy.ravel())).T

xx.shape, yy.shape, mesh.shape
```

Out[27]:
```
((400, 800), (400, 800), (320000, 2))
```

In [28]:
```
sigmoid = lambda x: 1/(1+np.exp(-x))
```

In [29]:
```
# Making predictions

#y = sigmoid(w^T@x + b) = sigmoid(x@w^T + b)

sigmoid(mesh@log_reg.coef_.T + log_reg.intercept_)
```

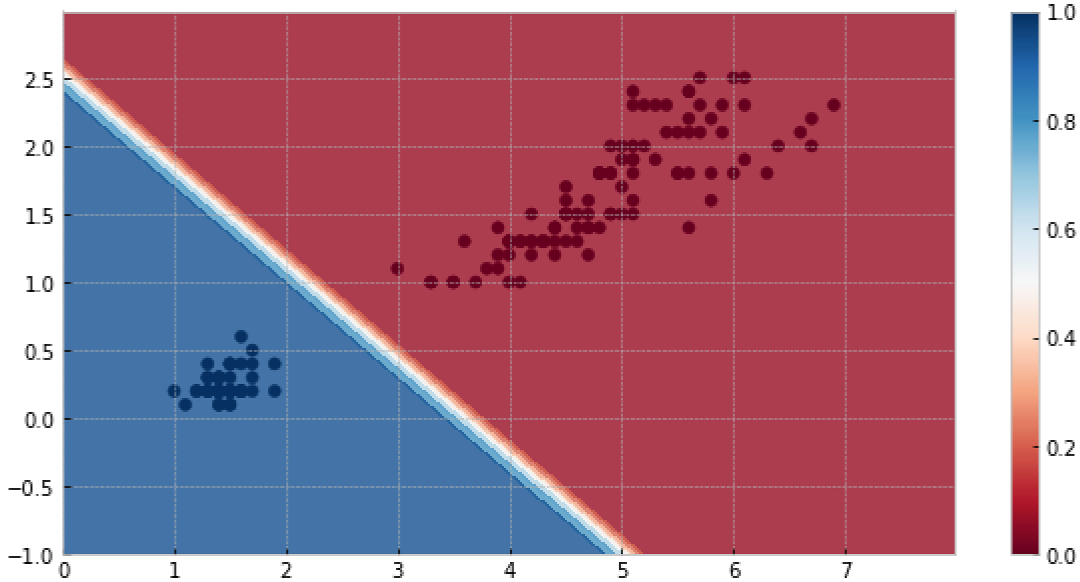
Out[29]:
```
array([[1.00000000e+00],
       [1.00000000e+00],
       [1.00000000e+00],
       ...,
       [1.11892967e-43],
       [9.97732180e-44],
       [8.89662266e-44]])
```

In [30]:
```
probs = log_reg.predict_proba(mesh)

probs
```

Out[30]:
```
array([[0.00000000e+00, 1.00000000e+00],
       [0.00000000e+00, 1.00000000e+00],
       [0.00000000e+00, 1.00000000e+00],
       ...,
       [1.00000000e+00, 1.11892967e-43],
       [1.00000000e+00, 9.97732180e-44],
       [1.00000000e+00, 8.89662266e-44]])
```

In [31]:
```
plt.figure(figsize=(10,5))
plt.contourf(xx, yy, probs[:,1].reshape(xx.shape), cmap=plt.cm.RdBu, alpha=.8)
plt.scatter(X[:,0], X[:,1], c=t, cmap=plt.cm.RdBu)
```

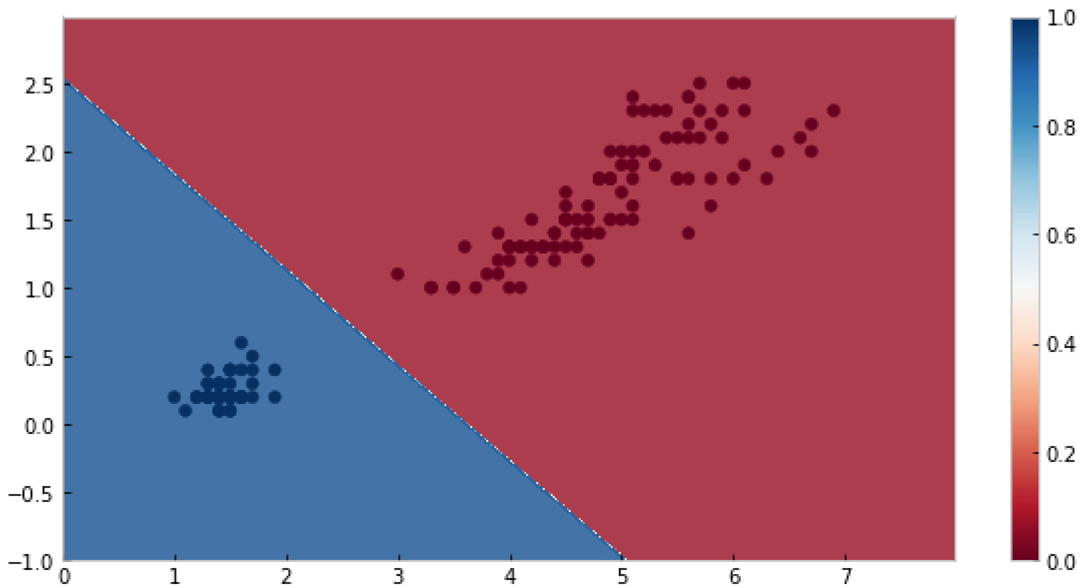
```
plt.rcParams['axes.grid'] = False #runtime configuration (rc) settings
plt.colorbar();
```



In [32]:
```
labels = log_reg.predict(mesh)

labels
```

Out[32]:
```
array([1, 1, 1, ..., 0, 0, 0])
```

In [33]:
```
plt.figure(figsize=(10,5))
plt.contourf(xx, yy, labels.reshape(xx.shape), cmap=plt.cm.RdBu, alpha=.8)
plt.scatter(X[:,0], X[:,1], c=t, cmap=plt.cm.RdBu)
plt.rcParams['axes.grid'] = False #runtime configuration (rc) settings
plt.colorbar();
```



Multiple Classes

Considering the extension of linear discriminants to $K > 2$ classes. We might be tempted be to build a K-class discriminant by combining a number of two-class discriminant functions. However, this leads to some serious difficulties.

- Consider the use of $K-1$ classifiers each of which solves a two-class problem of separating points in a particular class C_k from points not in that class. This is known as a **one-versus-all** classifier.

- An alternative is to introduce $K(K-1)/2$ binary discriminant functions, one for every possible pair of classes. This is known as a **one-versus-one** classifier. Each point is then classified according to a majority vote amongst the discriminant functions. However, this too runs into the problem of ambiguous regions.

In [2]:
```
Image('figures/Figure4.2.png',width=700)
# Source: Bishop textbook
```

Out[2]:

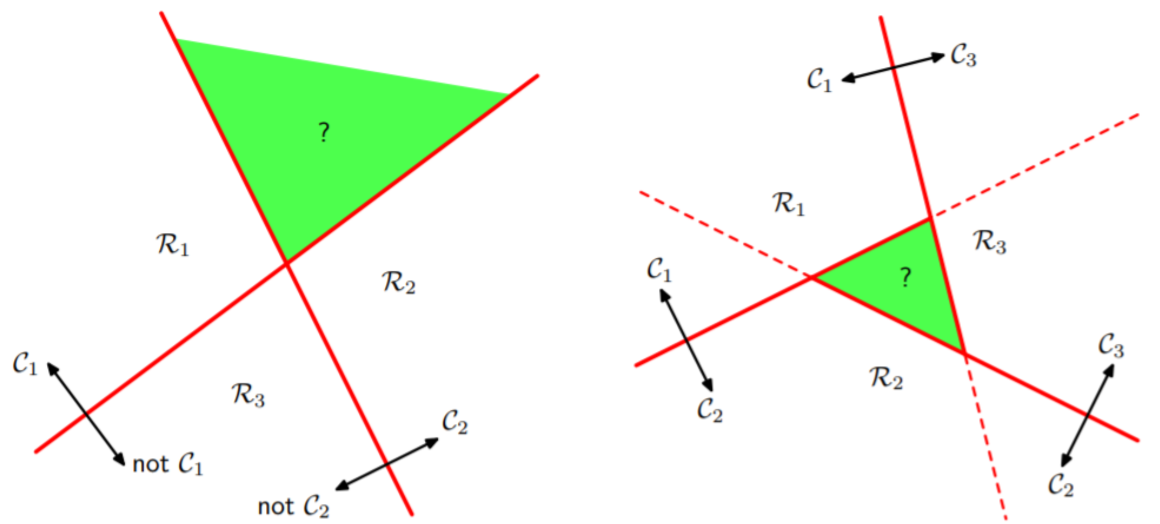


Figure 4.2 Attempting to construct a K class discriminant from a set of two class discriminants leads to ambiguous regions, shown in green. On the left is an example involving the use of two discriminants designed to distinguish points in class $\mathcal{C}_k$ from points not in class $\mathcal{C}_k$. On the right is an example involving three discriminant functions each of which is used to separate a pair of classes $\mathcal{C}_k$ and $\mathcal{C}_j$.

Softmax Regression

We can avoid these difficulties by considering a **single K-class discriminant** comprising K linear functions of the form

$$y_k(\mathbf{x}) = \phi(\mathbf{w}_k^T\mathbf{x} + b_k)$$

and then assigning a point $\mathbf{x}$ to class C_k if $y_k(\mathbf{x}) > y_j(\mathbf{x})$ for all $j \neq k$. The decision boundary between class C_k and class C_j is therefore given by $y_k(\mathbf{x}) = y_j(\mathbf{x})$ and hence corresponds to a $(D-1)$-dimensional hyperplane defined by

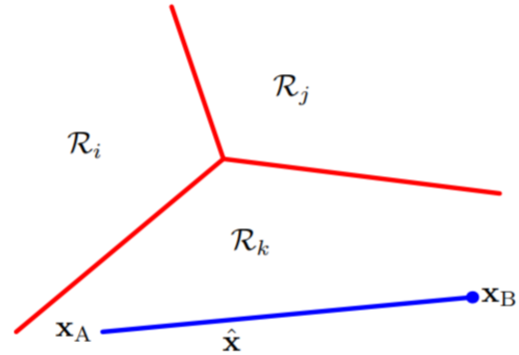
$$(\mathbf{w}_k - \mathbf{w}_j)^T\mathbf{x} + (b_k - bj) = 0$$

This has the same form as the decision boundary for the two-class case.

In [3]:
```python
Image('figures/Figure4.3.png',width=800)
# Source: Bishop textbook
```

Out[3]:

Figure 4.3 Illustration of the decision regions for a multiclass linear discriminant, with the decision boundaries shown in red. If two points $\mathbf{x}_A$ and $\mathbf{x}_B$ both lie inside the same decision region $\mathcal{R}_k$, then any point $\widehat{\mathbf{x}}$ that lies on the line connecting these two points must also lie in $\mathcal{R}_k$, and hence the decision region must be singly connected and convex.



The decision regions of such a discriminant are always **singly connected** and **convex**.

Once you have computed the score of every class for the instance $\mathbf{x}$, i.e. $y_k(\mathbf{x})$, you can estimate the probability p_k that the instance belongs to class k by running the scores through the **softmax function**.

$$p_k = \frac{\exp(y_k(\mathbf{x}))}{\sum_{j=1}^{K} \exp(y_j(\mathbf{x}))}$$

The function computes the exponential of every score, then normalizes them (dividing by the sum of all the exponentials). The scores are generally called logits or log-odds (although they are actually unnormalized log-odds).

The derivations will be derived in the same way as earlier (left as exercise).

In [34]:
```python
X = iris.data[:,2:]

t = iris.target

X.shape, t
```

Out[34]:
```
((150, 2),
 array([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, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]))
```

In [35]:
```python
log_reg = LogisticRegression(multi_class='multinomial', penalty='none')
```

In [36]:
```python
log_reg.fit(X, t)
```

Out[36]:
```
LogisticRegression(multi_class='multinomial', penalty='none')
```

In [37]:
```python
probs = log_reg.predict_proba(mesh)

probs
```

Out[37]:
```
array([[1.00000000e+00, 4.01323679e-27, 2.53553820e-51],
       [1.00000000e+00, 4.41989278e-27, 2.95787298e-51],
       [1.00000000e+00, 4.86775468e-27, 3.45055443e-51],
       ...,
       [3.20848214e-59, 1.50381706e-14, 1.00000000e+00],
       [2.75036456e-59, 1.41971965e-14, 1.00000000e+00],
       [2.35765851e-59, 1.34032519e-14, 1.00000000e+00]])
```

In [38]:
```
pred = log_reg.predict(mesh)

pred
```

Out[38]:
```
array([0, 0, 0, ..., 2, 2, 2])
```

In [39]:
```
plt.figure(figsize=(10,5))
plt.contourf(xx, yy, pred.reshape(xx.shape), cmap=plt.cm.RdBu, alpha=.8)
plt.scatter(X[:,0], X[:,1], c=t, cmap=plt.cm.RdBu)
plt.rcParams['axes.grid'] = False #runtime configuration (rc) settings
plt.colorbar();
```