CS5489 - Machine Learning

Lecture 3a - Linear Classifiers

Dr. Antoni B. Chan

Dept. of Computer Science, City University of Hong Kong

Outline

1. Discriminative linear classifiers
2. Logistic regression
3. Support vector machines (SVM)

Classification with Generative Model

- Steps to build a classifier
  1. Collect training data (features $\mathbf{x}$ and class labels y)
  2. Learn class-conditional distribution (CCD), $p(\mathbf{x}|y)$.
  3. Use Bayes' rule to calculate class probability, $p(y|\mathbf{x})$.
- **Note:** the data is used to learn the CCD -- the classifier is secondary.
  - Density estimation is an "ill-posed" problem -- which density to use? how much data is needed?

- Advice from Vladimir Vapnik (inventor of SVM):

> When solving a problem, try to avoid solving a more general problem as an intermediate step.

- **Discriminative solution**
  - Solve for the classifier $p(y|\mathbf{x})$ directly!

- Terminology
  - **"Discriminative"** - learn to directly discriminate the classes apart using the features.
  - **"Generative"** - learn model of how the features are generated from different classes.

Revisit the Naive Bayes Gaussian Classifier

- CCDs: assume the same variance for all Gaussians:
  - $p(\mathbf{x}|y=1) = \prod_{i=1}^{D} \mathcal{N}(x_i|\mu_i, \sigma^2)$
  - $p(\mathbf{x}|y=2) = \prod_{i=1}^{D} \mathcal{N}(x_i|\nu_i, \sigma^2)$
- prior:
  - $p(y=1) = \pi_1$, $p(y=2) = \pi_2$.

- look at the log-ratio of CCDs,

$$
\begin{aligned}
\log \frac{p(\mathbf{x}|y=1)}{p(\mathbf{x}|y=2)} &= \log \frac{\prod_{i=1}^{D} \mathcal{N}(x_i|\mu_i, \sigma^2)}{\prod_{i=1}^{D} \mathcal{N}(x_i|\nu_i, \sigma^2)} \\
&= \sum_{i=1}^{D} \log \mathcal{N}(x_i|\mu_i, \sigma^2) - \log \mathcal{N}(x_i|\nu_i, \sigma^2) \\
&= \sum_{i=1}^{D} -\frac{1}{2\sigma^2}(x_i - \mu_i)^2 + \frac{1}{2\sigma^2}(x_i - \nu_i)^2 \\
&= \frac{1}{2\sigma^2} \sum_{i=1}^{D} (2x_i\mu_i - \mu_i^2 - 2x_i\nu_i + \nu_i^2) \\
&= \frac{1}{2\sigma^2} \sum_{i=1}^{D} 2(\mu_i - \nu_i)x_i - \mu_i^2 + \nu_i^2
\end{aligned}
$$

- Thus

$$
\log \frac{p(\mathbf{x}|y=1)}{p(\mathbf{x}|y=2)} = \frac{1}{\sigma^2} \sum_{i=1}^{D} (\mu_i - \nu_i)x_i + \frac{1}{2\sigma^2} \sum_{i=1}^{D} (\nu_i^2 - \mu_i^2)
$$

- **Bayes decision rule:** Compute the posterior probability of each class $p(y=j|\mathbf{x})$
  - select class 1 when:

$$
\begin{aligned}
\log p(y=1|\mathbf{x}) &> \log p(y=2|\mathbf{x}) \\
\log p(\mathbf{x}|y=1) + \log p(y=1) &> \log p(\mathbf{x}|y=2) + \log p(y=2) \\
\log \frac{p(\mathbf{x}|y=1)}{p(\mathbf{x}|y=2)} + \log \frac{p(y=1)}{p(y=2)} &> 0
\end{aligned}
$$

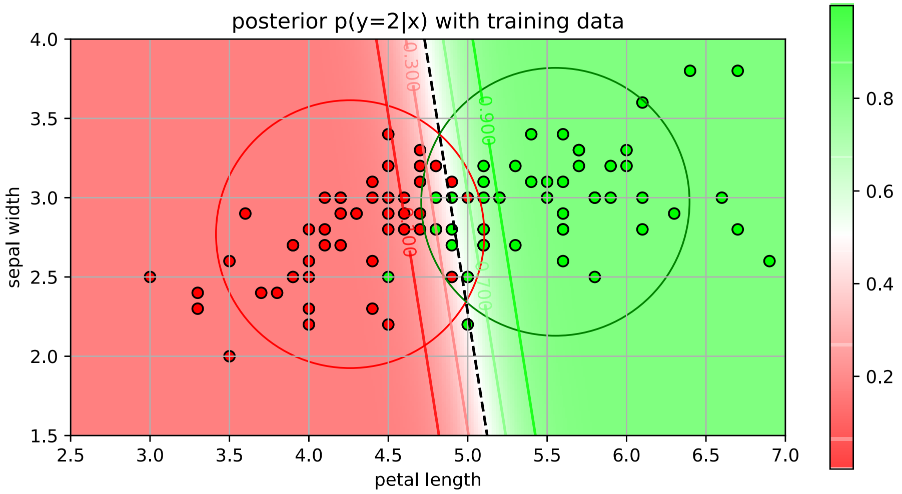
- substituting for the CCDs and priors, the BDR is:
  - select class $y=1$ when:

$$
\sum_{i=1}^{D} \frac{1}{\sigma^2}(\mu_i - \nu_i)x_i + \frac{1}{2\sigma^2} \sum_{i=1}^{D} (\nu_i^2 - \mu_i^2) + \log \frac{\pi_1}{\pi_2} > 0
$$

- Example

In [8]:
```
pfig
```

Out[8]:



- BDR in this case is a **linear** function
  - select class $y = 1$ when:

$$\sum_{i=1}^{D} \underbrace{\frac{1}{\sigma^2}(\mu_i - \nu_i)}_{w_i} x_i + \underbrace{\frac{1}{2\sigma^2}\sum_{i=1}^{D}(\nu_i^2 - \mu_i^2) + \log\frac{\pi_1}{\pi_2}}_{b} > 0$$

    - w_i is a per-feature weight
    - b is a bias term

- the BDR in this case is a **linear** classifier:
  - select class $y = 1$ when
    - $\sum_{i=1}^{D} w_i x_i + b > 0$
    - equivalently, $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b > 0$
- Here we obtain the weights $\mathbf{w}$ by learning the CCDs
  - assuming Naive Bayes Gaussians with shared variance.
  - this is a generative model since we learn how the data is generated for each class (CCDs).

- How to learn the linear classifier in a discriminative way?
  - directly learn the posterior $p(y|\mathbf{x})$.
  - we will look at a generic linear classifier.

Linear Classifier

- **Setup**
  - Observation (feature vectors) $\mathbf{x} \in \mathbb{R}^d$
  - Class $y \in \{-1, +1\}$
- **Goal**: given a feature vector $\mathbf{x}$, predict its class y.
  - Calculate a *linear function* of the feature vector $\mathbf{x}$.
    - $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = \sum_{j=1}^{d} w_j x_j + b$
      - $\mathbf{w} \in \mathbb{R}^d$ are the weights of the linear function.
      - multiply each feature value with a weight, and then add together.
  - Predict from the value:
    - if $f(\mathbf{x}) > 0$ then predict Class $y = 1$
    - if $f(\mathbf{x}) < 0$ then predict Class $y = -1$
    - Equivalently, $y = \mathrm{sign}(f(\mathbf{x}))$

Geometric Interpretation

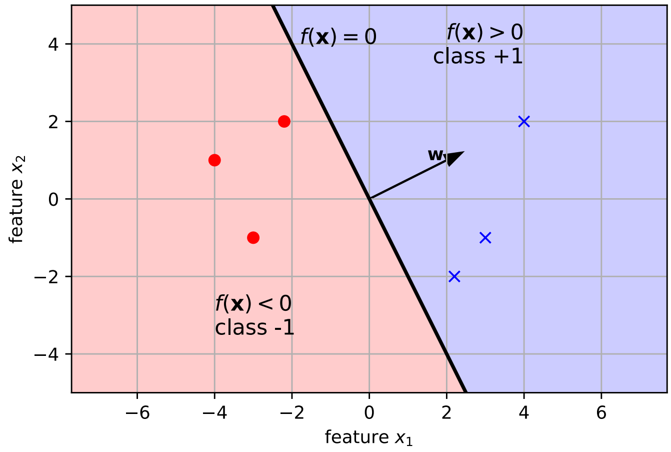
- The linear classifier separates the features space into 2 *half-spaces*
  - corresponding to feature values belonging to Class +1 and Class -1
  - the class boundary is normal to $\mathbf{w}$.
    - also called the *separating hyperplane*.

- Example:

$$\mathbf{w} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, b = 0$$

In [11]:
```
linclass
```

Out[11]:



Separating Hyperplane

- In a d-dimensional feature space, the parameters are $\mathbf{w} \in \mathbb{R}^d$.
- The equation $\mathbf{w}^T\mathbf{x} + b = 0$ defines a $(d-1)$-dim. linear surface:
    - for $d = 2$, $\mathbf{w}$ defines a 1-D line.
    - for $d = 3$, $\mathbf{w}$ defines a 2-D plane.
    - ...
    - in general, we call it a hyperplane.

Learning the classifier

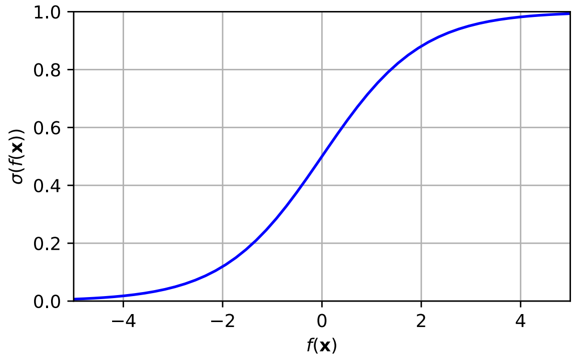
- How to set the classifier parameters $(\mathbf{w}, b)$?
    - Learn them from training data!
- Classifiers differ in the objectives used to learn the parameters $(\mathbf{w}, b)$.
    - We will look at two examples:
        - *logistic regression*
        - *support vector machine (SVM)*

Logistic regression

- Use a probabilistic approach
    - Map the linear function $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$ to probability values between 0 and 1 using a *sigmoid* function.
    - $\sigma(z) = \frac{1}{1+e^{-z}}$

In [13]:
```
sigmoidplot
```

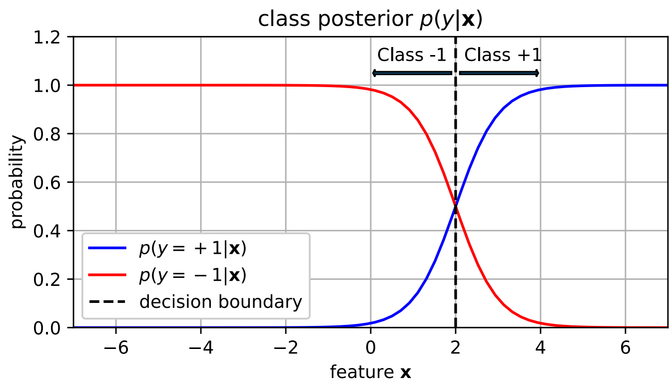
Out[13]:



- Given a feature vector x, the probability of a class is:
  - $p(y=+1|\mathbf{x}) = \sigma(f(\mathbf{x}))$
  - $p(y=-1|\mathbf{x}) = 1-\sigma(f(\mathbf{x}))$
- Note: here we are directly modeling the class posterior probability!
  - not the class-conditional $p(\mathbf{x}|y)$

In [15]:
```
lrexample
```

Out[15]:



Learning the parameters

- Given training data $\{\mathbf{x}_i, y_i\}_{i=1}^N$, learn the function parameters $(\mathbf{w}, b)$ using maximum likelihood estimation.
- maximize the likelihood of the data $\{\mathbf{x}_i, y_i\}$ according to the posterior:

$$(\mathbf{w}^*, b^*) = \operatorname*{argmax}_{\mathbf{w},b} \sum_{i=1}^N \log p(y_i|\mathbf{x}_i)$$

- posterior is a Bernoulli distribution (given $\mathbf{x}$):

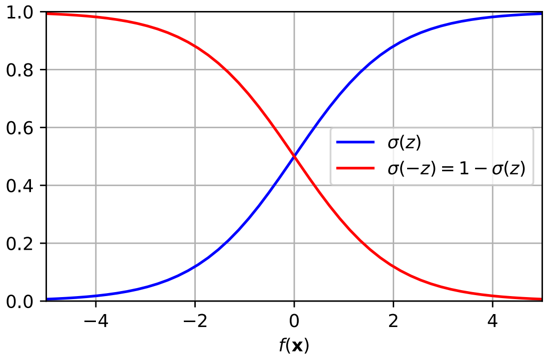
$$p(y|\mathbf{x}) = \begin{cases} \sigma(f(\mathbf{x})), & y=1 \\ 1-\sigma(f(\mathbf{x})), & y=-1 \end{cases}$$

- Note the following property:

$$1-\sigma(z) = \sigma(-z)$$

In [17]:
```
sigmoidplot
```

Out[17]:



- Thus,

$$p(y|\mathbf{x}) = \begin{cases} \sigma(f(\mathbf{x})), & y = 1 \\ \sigma(-f(\mathbf{x})), & y = -1 \end{cases}$$

- Simplifying the 2 cases into one equation,

$$p(y|\mathbf{x}) = \sigma(yf(\mathbf{x}))$$

- Taking the log,

$$\begin{aligned} \log p(y|\mathbf{x}) &= \log \sigma(yf(\mathbf{x})) \\ &= \log \frac{1}{1+e^{-yf(\mathbf{x})}} \\ &= -\log(1+e^{-yf(\mathbf{x})}) \end{aligned}$$

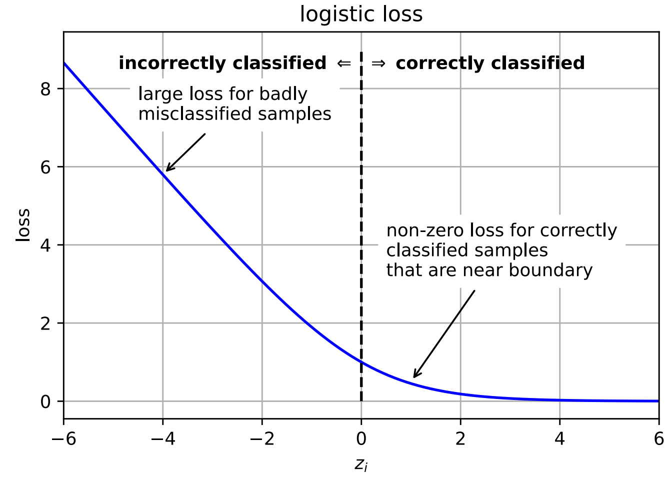
- Substituting into the MLE formulation:

$$\begin{aligned} (\mathbf{w}^*, b^*) &= \underset{\mathbf{w},b}{\operatorname{argmax}} \sum_{i=1}^{N} \log p(y_i|\mathbf{x}_i) \\ &= \underset{\mathbf{w},b}{\operatorname{argmin}} \sum_{i=1}^{N} \log(1+e^{-y_i f(\mathbf{x}_i)}) \end{aligned}$$

- the term on the right is a *data-fit term*
  - wants to make the parameters $(\mathbf{w}, b)$ to well fit the data.
  - Define $z_i = y_i f(\mathbf{x}_i)$
    - Interesting observation:
      - $z_i > 0$ when sample $\mathbf{x}_i$ is classified correctly
      - $z_i < 0$ when sample $\mathbf{x}_i$ is classified incorrectly
      - $z_i = 0$ when sample is on classifier boundary
  - logistic loss function: $L(z_i) = \log(1 + \exp(-z_i))$

In [19]:
```
lossfig
```

Out[19]:

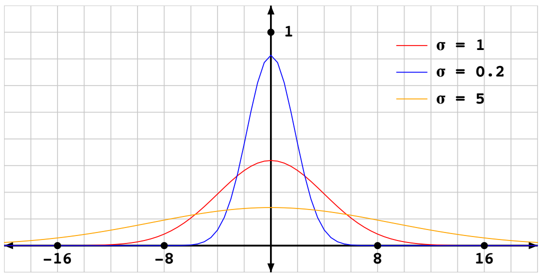


Regularization

- to prevent *overfitting*, add a prior distribution on $\mathbf{w}$.
  - prefer solutions that are likely under the prior.

$$(\mathbf{w}^*, b^*) = \operatorname*{argmax}_{\mathbf{w},b} \log p(\mathbf{w}) + \sum_{i=1}^{N} \log p(y_i|\mathbf{x}_i)$$

- assume Gaussian distribution on $\mathbf{w}$ with variance $C/2$
  - $p(\mathbf{w}) = \mathcal{N}(\mathbf{w}|0, \frac{C}{2}\mathbf{I})$
    - small values of C keep $\mathbf{w}$ close to 0.
    - large values of C allow larger values of $\mathbf{w}$.
  - $\log p(\mathbf{w}) = -\frac{1}{C}\mathbf{w}^T\mathbf{w} + \text{constant}$



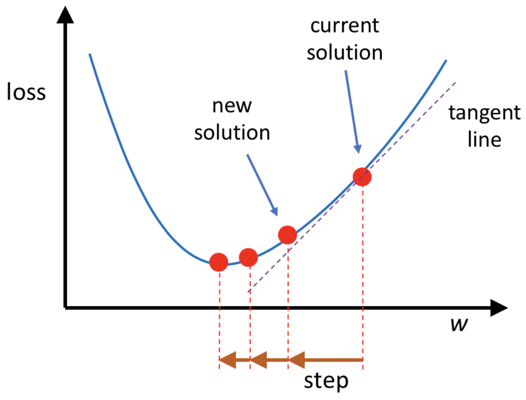
- Substituting,

$$(\mathbf{w}^*, b^*) = \operatorname*{argmin}_{\mathbf{w},b} \frac{1}{C}\mathbf{w}^T\mathbf{w} + \sum_{i=1}^{N} \log(1 + \exp(-y_i f(\mathbf{x}_i)))$$

- the first term is the *regularization term*
  - Note: $\mathbf{w}^T\mathbf{w} = \sum_{j=1}^{d} w_j^2$
  - penalty term that keeps entries in $\mathbf{w}$ from getting too large.
  - C is the regularization *hyperparameter*
    - larger C values apply less penalty on large $\mathbf{w}$ → allow large values in $\mathbf{w}$.
    - smaller C values apply more penalty on large $\mathbf{w}$ → discourage large values in $\mathbf{w}$.
- the second term is the *data fit term* - same as before.

Optimization

- **no closed-form solution**
  - use an iterative optimization algorithm to find the optimal solution
  - e.g., *gradient descent* - step downhill in each iteration.
    - $\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{dE}{d\mathbf{w}}$
    - where E is the objective function
    - η is the *learning rate* (how far to step in each iteration).

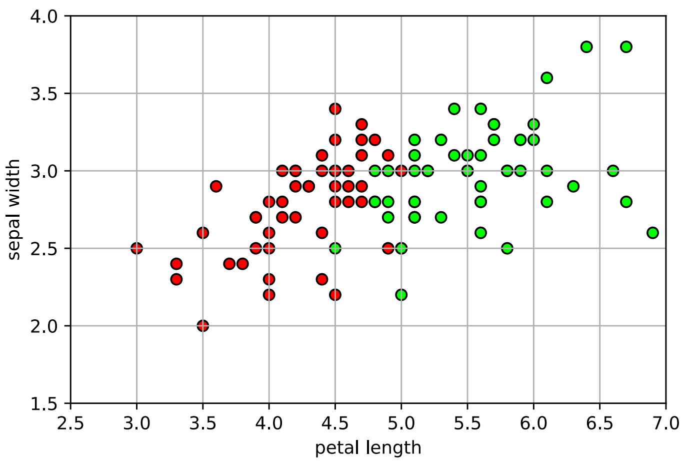


Example: Iris Data

In [20]:
```
# load iris data each row is (petal length, sepal width, class)
irisdata = loadtxt('iris2.csv', delimiter=',', skiprows=1)

X = irisdata[:,0:2]  # the first two columns are features (petal length, sepal width)
Y = irisdata[:,2]    # the third column is the class label (versicolor=1, virginica=2)
                     #  --> automaticaly mapped to (-1, +1) when training classifier

print(X.shape)
```

```
(100, 2)
```

In [22]:
```
# show the data
plt.figure()
plt.scatter(X[:,0], X[:,1], c=Y, cmap=mycmap, edgecolors='k')
irisaxis(axbox)
```



In [23]:
```
# randomly split data into 50% train and 50% test set
trainX, testX, trainY, testY = \
  model_selection.train_test_split(X, Y,
  train_size=0.5, test_size=0.5, random_state=4487)

print(trainX.shape)
print(testX.shape)
```

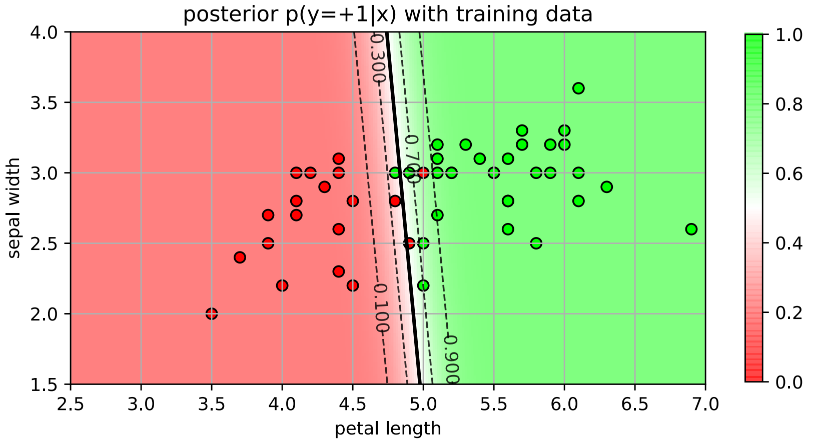
```
(50, 2)
(50, 2)
```

In [24]:
```
# learn logistic regression classifier
# (C is a regularization hyperparameter)
logreg = linear_model.LogisticRegression(C=100)
logreg.fit(trainX, trainY)

print("w =", logreg.coef_)
print("b =", logreg.intercept_)
```

```
w = [[9.51275841 0.89596567]]
b = [-48.68254369]
```

- Equation:
  - $f(\mathbf{x}) = (4.87 * \text{petal\_length}) - (0.62 * \text{sepal\_width}) - 21.68$
- Interpretation:
  - large petal length makes $f(\mathbf{x})$ positive, so large petal length is associated with class +1.

In [26]:
```
# show the posterior and training data
plt.figure(figsize=(8,6))
plot_posterior(logreg, axbox, mycmap)
plt.scatter(trainX[:,0], trainX[:,1], c=trainY, cmap=mycmap, edgecolors='k')
plt.title('posterior p(y=+1|x) with training data');
```

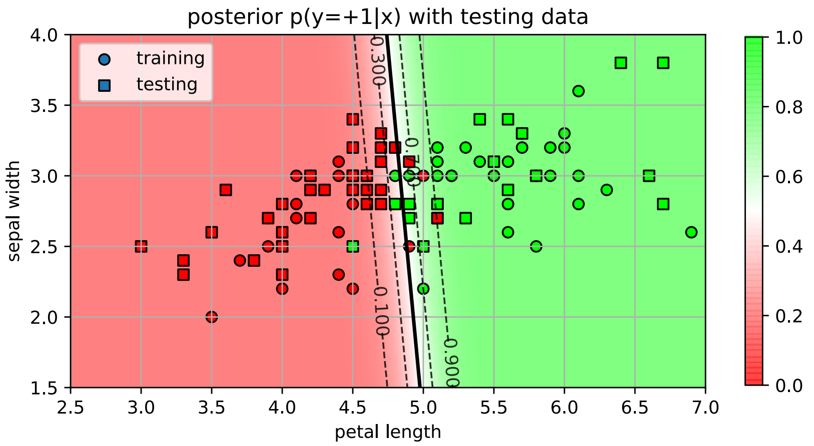


In [27]:
```
# predict from the model
predY = logreg.predict(testX)

# calculate accuracy
acc = metrics.accuracy_score(testY, predY)
print("test accuracy =", acc)
```

```
test accuracy = 0.92
```

In [29]:
```
postfig
```

Out[29]:

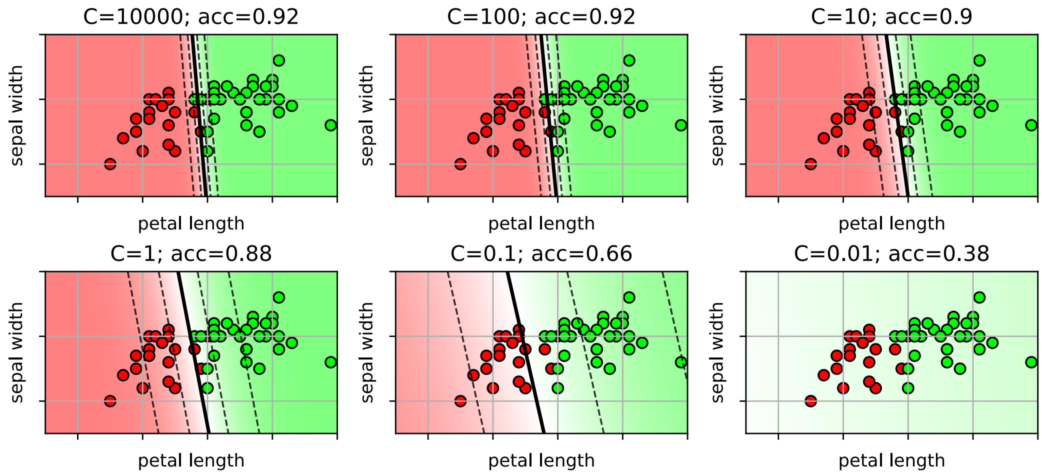


Selecting the regularization hyperparameter

- the regularization hyperparameter C has a large effect on the decision boundary and the accuracy.
  - larger C makes the classifier more confident (posterior probabilities saturate to 0 and 1)
    - more likely to overfit
  - smaller C makes the classifer less confident (wider range of posterior probabilities).
    - less likely to overfit
- How to set the value of C?

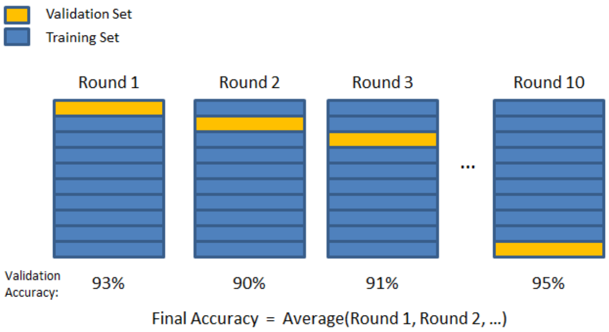
In [31]:
```
lrC
```

Out[31]:



Cross-validation

- Use *cross-validation* on the training set to select the best value of C.
  - Run many experiments on the training set to see which parameters work on different versions of the data.
    - Split the data into batches of training and validation data.
    - Try a range of C values on each split.
    - Pick the value that works best over all splits.



- **Procedure**
  1. select a range of C values to try
  2. Repeat K times
     A. Split the training set into training data and validation data
     B. Learn a classifier for each value of C
     C. Record the accuracy on the validation data for each C
  3. Select the value of C that has the highest average accuracy over all K folds.
  4. Retrain the classifier using all data and the selected C.

- scikit-learn already has built-in `cross_validation` module (more later).

- for logistic regression, use *LogisticRegressionCV* class

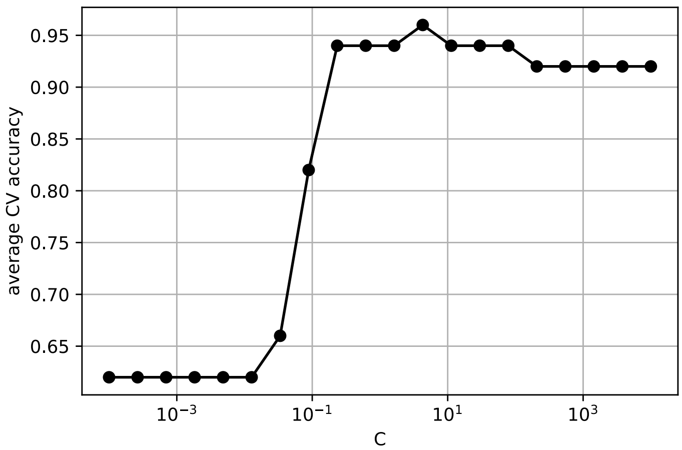
In [32]:
```
# learn logistic regression classifier using CV
#  Cs is an array of possible C values
#  cv is the number of folds
#  n_jobs=-1 means run in parallel with all cores
logreg = linear_model.LogisticRegressionCV(Cs=logspace(-4,4,20), cv=5, n_jobs=-1)
logreg.fit(trainX, trainY)

print("w=", logreg.coef_)
print("b=", logreg.intercept_)

# predict from the model
predY = logreg.predict(testX)

# calculate accuracy
acc = metrics.accuracy_score(testY, predY)
print("test accuracy=", acc)
```

```
w= [[4.61911023 0.72397804]]
b= [-24.24716682]
test accuracy= 0.9
```

Which C was selected?

In [33]:
```
print("C =", logreg.C_)
# calculate the average score for each C
avgscores = mean(logreg.scores_[2],0)  # 2 is the class label
plt.semilogx(logreg.Cs_, avgscores, 'ko-')
plt.xlabel('C'); plt.ylabel('average CV accuracy'); plt.grid(True);
```

```
C = [4.2813324]
```



Multi-class classification

- So far, we have only learned a classifier for 2 classes (+1, -1)
  - called a **binary classifier**
- For more than 2 classes, split the problem up into several binary classifier problems.
  - **1-vs-rest**
    - *Training:* for each class, train a classifier for that class versus the other classes.
      - For example, if there are 3 classes, then train 3 binary classifiers: 1 vs {2,3}; 2 vs {1,3}; 3 vs {1,2}
    - *Prediction:* calculate probability for each binary classifier. Select the class with highest probability.

Example on 3-class Iris data

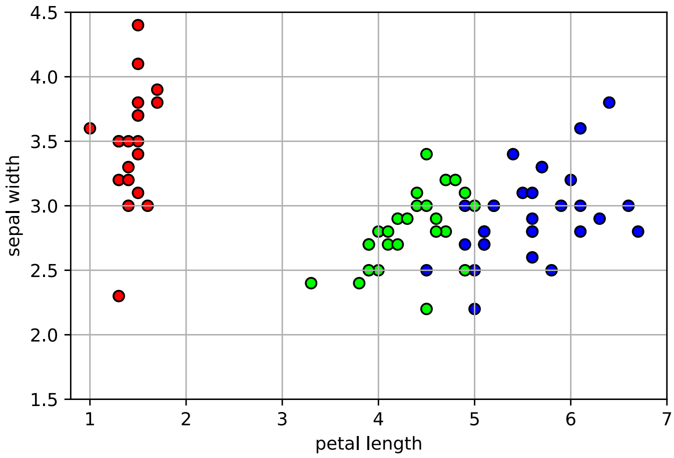
In [34]:
```
# load iris data each row is (petal length, sepal width, class)
irisdata = loadtxt('iris3.csv', delimiter=',', skiprows=1)

X = irisdata[:,0:2]  # the first two columns are features (petal length, sepal width)
Y = irisdata[:,2]    # the third column is the class label (setosa=0, versicolor=1, virginica=2)

print(X.shape)
```

```
(150, 2)
```

In [35]:
```
# randomly split data into 50% train and 50% test set
trainX, testX, trainY, testY = \
  model_selection.train_test_split(X, Y,
  train_size=0.5, test_size=0.5, random_state=4487)

print(trainX.shape)
print(testX.shape)
```

```
(75, 2)
(75, 2)
```

In [36]:
```
# look at training data

axbox3 = [0.8, 7, 1.5, 4.5]
# make a colormap for viewing 3 classes
mycmap3 = matplotlib.colors.LinearSegmentedColormap.from_list('mycmap', ["#FF0000", "#00FF00", "#0000FF

plt.scatter(trainX[:,0], trainX[:,1], c=trainY, cmap=mycmap3, edgecolors='k')
plt.axis(axbox3); plt.grid(True);
plt.xlabel('petal length'); plt.ylabel('sepal width');
```



In [37]:
```
# learn logistic regression classifier (one-vs-all)
mlogreg = linear_model.LogisticRegression(C=10, multi_class='ovr')
mlogreg.fit(trainX, trainY)

# now contains 3 hyperplanes and 3 bias terms (one for each class)
print("w=", mlogreg.coef_)
print("b=", mlogreg.intercept_)

# predict from the model
predY = mlogreg.predict(testX)

# calculate accuracy
acc = metrics.accuracy_score(testY, predY)
print("test accuracy=", acc)
```

```
w= [[-3.54884501  1.11513222]
 [-0.03185278 -2.23119433]
 [ 5.41926127 -1.69411622]]
b= [  6.26247277   6.1229662  -21.79941354]
test accuracy= 0.9733333333333334
```

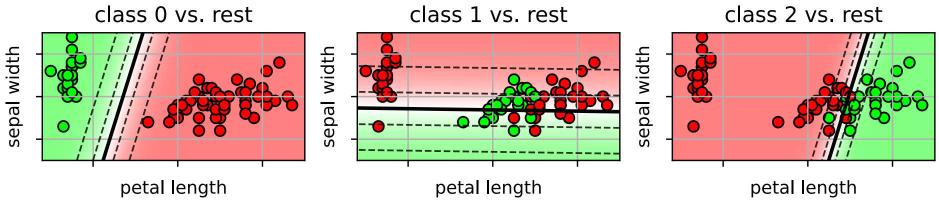
- the individual 1-vs-rest binary classifiers

In [39]:
```
print("w=", mlogreg.coef_)
```

```
print("b=", mlogreg.intercept_)

mlrfig
```

```
w= [[-3.54884501  1.11513222]
 [-0.03185278 -2.23119433]
 [ 5.41926127 -1.69411622]]
b= [  6.26247277   6.1229662  -21.79941354]
```

Out[39]:

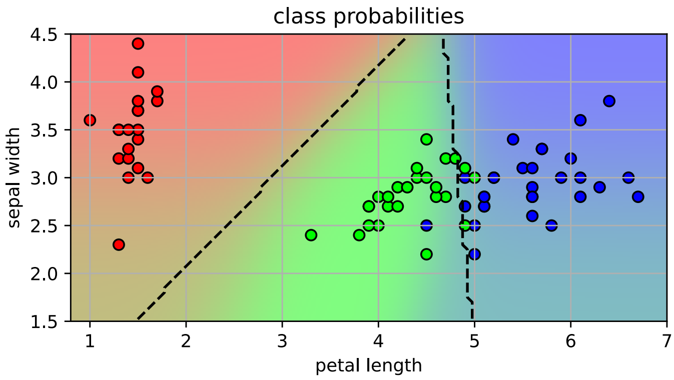


- the final classifier, combining all 1 vs rest classifiers

In [41]:

```
lr3class
```

Out[41]:



Multiclass logistic regression

- Another way to get a multi-class classifier is to define a multi-class objective.
  - One weight vector $\mathbf{w}_c$ for each class c.
  - linear function for each class, $f_c(\mathbf{x}) = \mathbf{w}_c^T \mathbf{x}$.
- Define probabilities with **softmax** function
  - analogous to sigmoid function for binary logistic regression.

$$p(y = c|\mathbf{x}) = \frac{e^{f_c(\mathbf{x})}}{e^{f_1(\mathbf{x})} + \cdots + e^{f_K(\mathbf{x})}}$$

  - The class with largest response of $f_c(\mathbf{x})$ will have the highest probability.

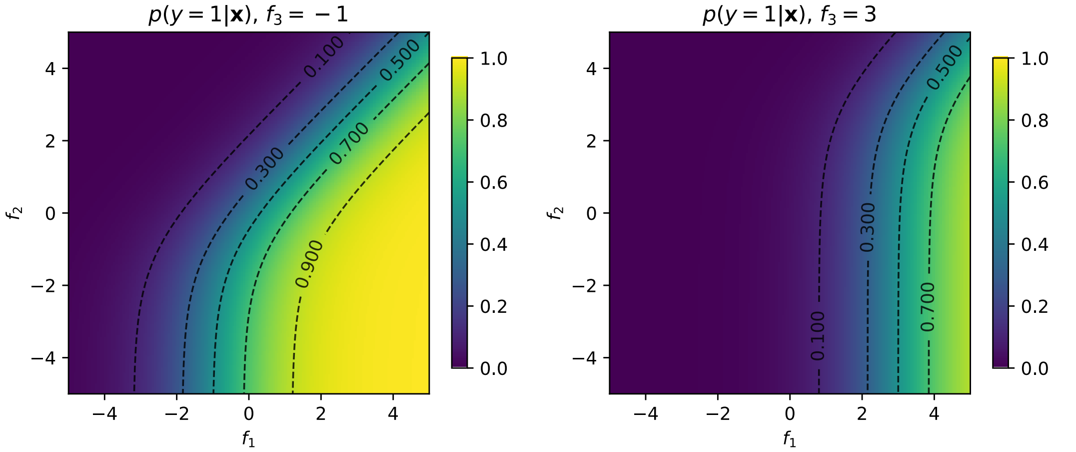
- Example with $K = 3$.

$$p(y = 1|\mathbf{x}) = \frac{e^{f_1(\mathbf{x})}}{e^{f_1(\mathbf{x})} + e^{f_2(\mathbf{x})} + e^{f_3(\mathbf{x})}}$$

In [43]:

```
sfmfig
```

Out[43]:



Parameter estimation

- Estimate the $\{\mathbf{w}_j\}$ parameters using MLE.
- Let $(\mathbf{x}, \mathbf{y})$ be a data sample pair:
  - $\mathbf{x}$ feature vector.
  - $\mathbf{y} = [y_1, \cdots, y_K]$ is a one-hot vector, where $y_c = 1$ when class c, and 0 otherwise.
- Data likelihood of $(\mathbf{x}, \mathbf{y})$.

$$\begin{aligned}\text{likelihood:} \quad & p(\mathbf{y}|\mathbf{x}) = \prod_{j=1}^{K} p(y=j|\mathbf{x})^{y_j} \\ \text{log-likelihood:} \quad & \log p(\mathbf{y}|\mathbf{x}) = \sum_{j=1}^{K} y_j \log p(y=j|\mathbf{x}) \\ \text{negative log-likelihood:} \quad & -\log p(\mathbf{y}|\mathbf{x}) = -\sum_{j=1}^{K} y_j \log p(y=j|\mathbf{x})\end{aligned}$$

  - equivalent to the *cross-entropy loss*

- Given dataset $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$
  - maximize the data log-likelihood:

$$\max_{\{\mathbf{w}_j\}} \sum_{i=1}^{N} \log p(\mathbf{y}_i|\mathbf{x}_i) = \max_{\{\mathbf{w}_j\}} \sum_{i=1}^{N} \sum_{j=1}^{K} y_{ij} \log p(y=j|\mathbf{x}_i)$$

  - i.e., minimize the cross-entropy loss

In [44]:

```
# learn logistic regression classifier
mlogreg = linear_model.LogisticRegression(C=10,
            multi_class='multinomial')
            # use multi-class and corresponding solver
mlogreg.fit(trainX, trainY)

# now contains 3 hyperplanes and 3 bias terms (one for each class)
print("w=", mlogreg.coef_)
print("b=", mlogreg.intercept_)

# predict from the model
predY = mlogreg.predict(testX)

# calculate accuracy
acc = metrics.accuracy_score(testY, predY)
print("test accuracy=", acc)
```

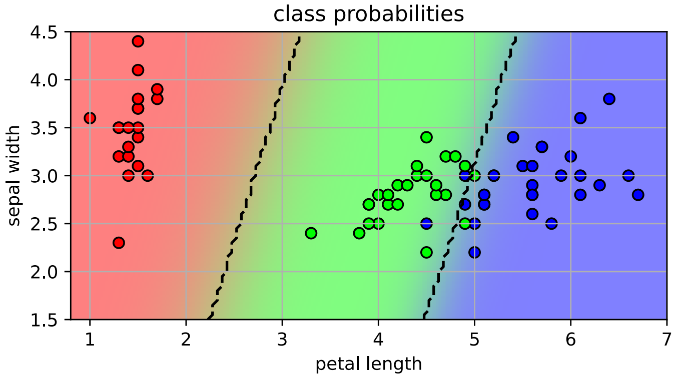
```
w= [[-4.13092437  1.30718735]
```

```
 [-0.71717021  0.23609022]
 [ 4.84809458 -1.54327757]]
b= [ 11.46078594   5.40723484 -16.86802078]
test accuracy= 0.9733333333333334
```

In [46]:

```
lr3classm
```

Out[46]:



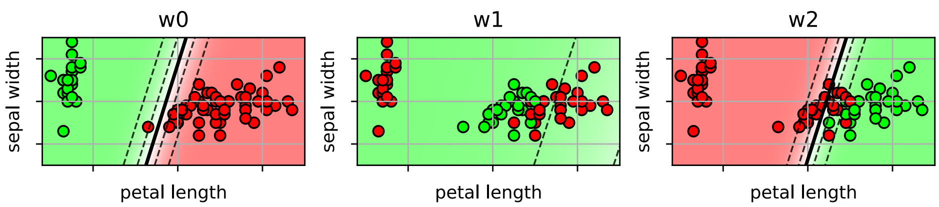
- individual weight vectors work together to partition the space

In [48]:

```
print("w=", mlogreg.coef_)
print("b=", mlogreg.intercept_)

lr31vr
```

```
w= [[-4.13092437  1.30718735]
 [-0.71717021  0.23609022]
 [ 4.84809458 -1.54327757]]
b= [ 11.46078594   5.40723484 -16.86802078]
```

Out[48]:



Logistic Regression Summary

- **Classifier:**
  - linear function $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$
  - Given a feature vector $\mathbf{x}$, the probability of a class is:
    - $p(y = +1|\mathbf{x}) = \sigma(f(\mathbf{x}))$
    - $p(y = -1|\mathbf{x}) = 1 - \sigma(f(\mathbf{x}))$
    - *sigmoid* function: $\sigma(z) = \frac{1}{1+e^{-z}}$
  - logistic loss function: $L(z) = \log(1 + \exp(-z))$
- **Training:**
  - Maximize the likelihood of the training data.
  - Use regularization to prevent overfitting.
    - Use cross-validation to pick the regularization hyperparameter C.

- **Classification:**
  - Given a new sample $\mathbf{x}^*$:
    - pick class with highest probability $p(y|\mathbf{x}^*)$:

$$y^* = \begin{cases} +1, p(y=+1|\mathbf{x}^*) > p(y=-1|\mathbf{x}^*) \\ -1, \text{otherwise} \end{cases}$$

- alternatively, just use $f(\mathbf{x}^*)$

$$y^* = \begin{cases} +1, f(\mathbf{x}^*) > 0 \\ -1, \text{otherwise} \end{cases} = \text{sign}(f(\mathbf{x}_*))$$

- **Extend to multi-class:**
  - K linear functions, one for each class.
  - compute probability using softmax function
  - MLE equivalent to cross-entropy loss

In [ ]:

```

```