CS5489 - Machine Learning

Lecture 4a - Non-Linear Classifiers

Dr. Antoni B. Chan

Dept. of Computer Science, City University of Hong Kong

Outline

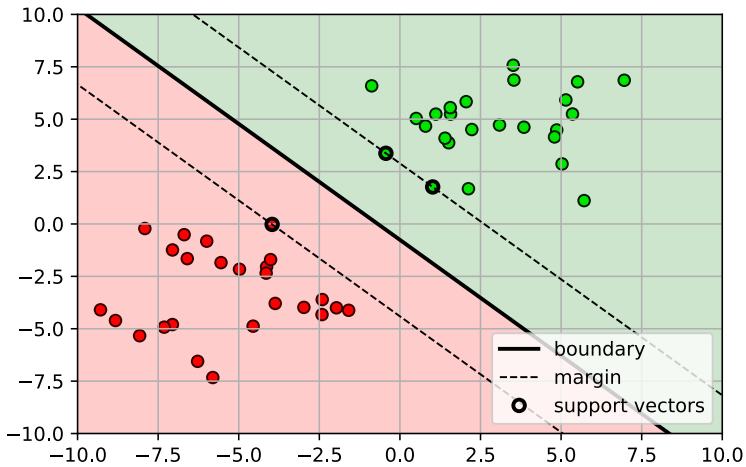
1. Nonlinear classifiers
2. Kernel trick and kernel SVM
3. Ensemble Methods - Boosting, Random Forests
4. Classification Summary

Linear Classifiers

- So far we have only looked at *linear classifiers*
  - separate classes using a hyperplane (line, plane).
  - e.g., support vector machine, logistic regression

In [4]:
```
maxmfig
```

Out[4]:

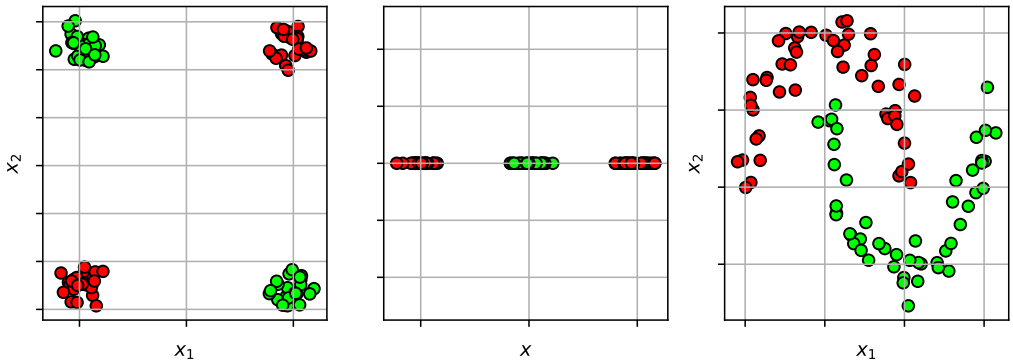


Non-linear decision boundary

- What if the data is separable, but not linearly separable?

In [6]:
```
nlfig
```

Out[6]:

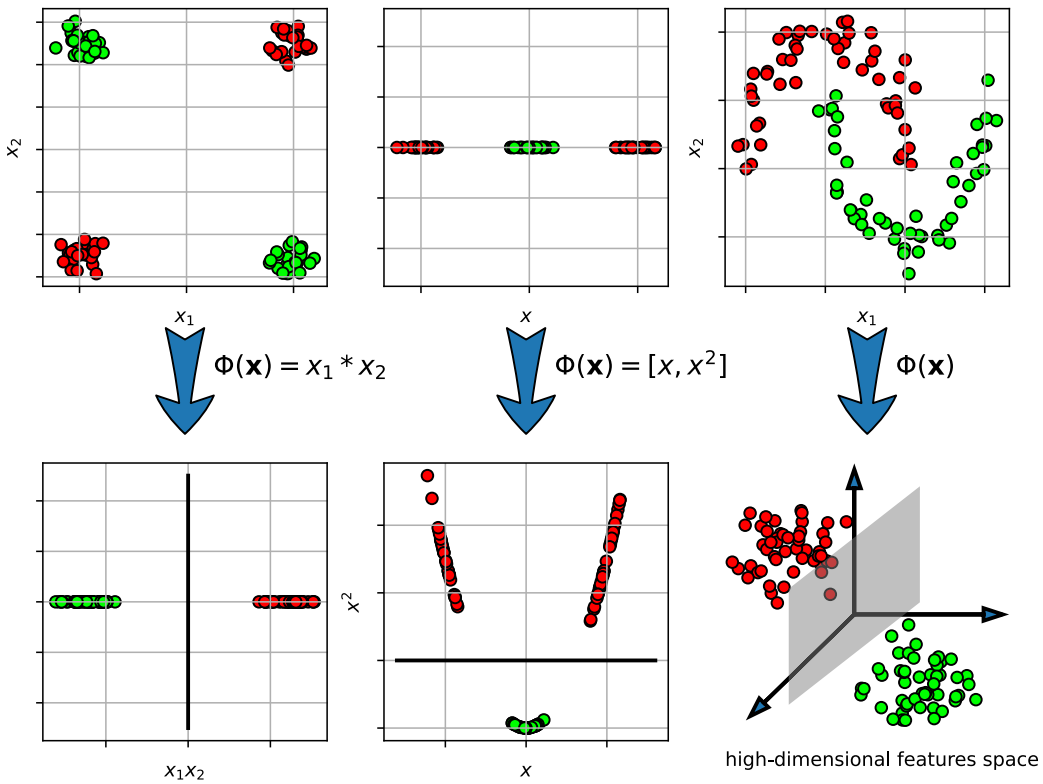


Idea - transform the input space

- map from input space $\mathbf{x} \in \mathbb{R}^d$ to a new high-dimensional space $\mathbf{z} \in \mathbb{R}^D$.
  - $\mathbf{z} = \Phi(\mathbf{x})$, where $\Phi(\mathbf{x})$ is the transformation function.
- learn the linear classifier in the new space
  - if dimension of new space is large enough ($D > d$), then the data should be linearly separable

In [8]:
```
nl2fig
```

Out[8]:



Example

- Let's try it...
  - 2-dimensional vector inputs
    - $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$
  - transformation consists of quadratic terms

- $\Phi(\mathbf{x}) = \begin{bmatrix} x_1^2 \\ x_1 x_2 \\ x_2^2 \end{bmatrix}$

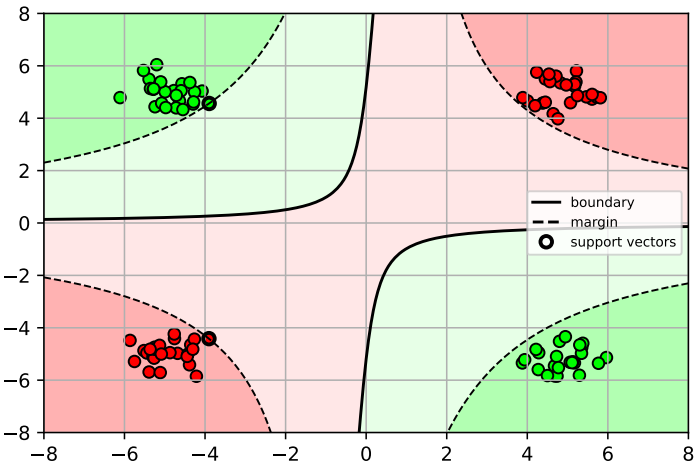
In [10]:
```
# define the transformation function for a vector x
# (a lambda function is an anonymous function)
phi = lambda x: array([x[0]**2, x[1]**2, x[0]*x[1]])

# apply function phi to each row of data matrix X1
PX = apply_along_axis(phi, 1, X1)

# fit SVM with transformed data
clf = svm.SVC(kernel='linear', C=inf)
clf.fit(PX, Y1)
```

Out[10]:
```
SVC(C=inf, kernel='linear')
```

In [11]:
```
# make plot
axbox = [-8, 8, -8, 8]
plt.figure()
plot_posterior_svm(clf, axbox, X1, phi=phi)
plt.scatter(X1[:,0], X1[:,1], c=Y1, cmap=mycmap, edgecolors='k');
```



SVM with transformed input

- Given a training set $\{\mathbf{x}_i, y_i\}_{i=1}^N$, the original SVM training is:

$$\underset{\mathbf{w},b}{\operatorname{argmin}} \frac{1}{2}\mathbf{w}^T\mathbf{w} \quad \text{s.t. } y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1, \quad 1 \leq i \leq N$$

- Apply high-dimensional transform to input $\mathbf{x} \rightarrow \Phi(\mathbf{x})$:

$$\underset{\mathbf{w},b}{\operatorname{argmin}} \frac{1}{2}\mathbf{w}^T\mathbf{w} \quad \text{s.t. } y_i(\mathbf{w}^T\Phi(\mathbf{x}_i) + b) \geq 1, \quad 1 \leq i \leq N$$

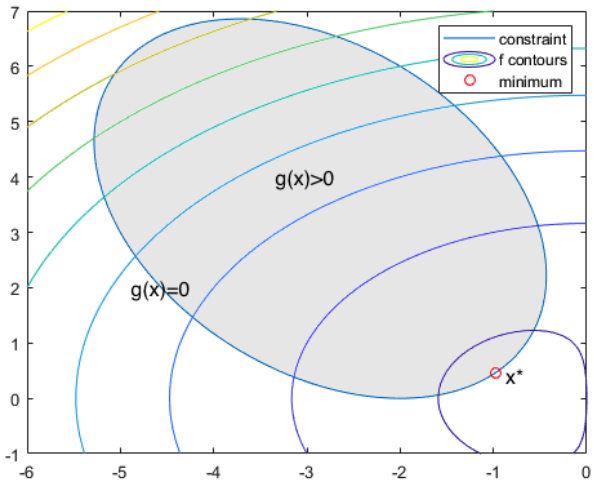
- **Note:** the hyperplane $\mathbf{w} \in \mathbb{R}^D$ is now in the high-dimensional space!
  - if D is very large,
    - calculating feature vector $\Phi(\mathbf{x}_i)$ could be time consuming.
    - optimization could be very inefficient in high-dimensional space.
  - To solve this problem requires some optimization theory...

Review of Constrained Optimization

- Consider an optimization problem with inequality constraints:

$$\min_{\mathbf{x}} f(\mathbf{x}) \text{ s.t. } g(\mathbf{x}) \geq 0$$

  - $f(\mathbf{x})$ is the objective function (for SVM, it's the inverse margin).
  - $g(\mathbf{x})$ is the constraint function (for SVM, it's the margin constraint).



- Use Langrange multipliers to solve this problem:
  - introduce Lagrange multiplier: $\lambda \geq 0$
  - form the Lagrangian: $L(\mathbf{x}, \lambda) = f(\mathbf{x}) - \lambda g(\mathbf{x})$
  - find the stationary point $(\mathbf{x}^*, \lambda^*)$ of the Lagrangian
    - find solution of $\frac{dL}{d\mathbf{x}} = 0$ and $\frac{dL}{d\lambda} = 0$.
  - at the solution, the Langrange multiplier indicates the mode of the inequality constraint
    - when $\lambda^* > 0$, then $g(\mathbf{x}^*) = 0$ (called "active equality").
    - when $\lambda^* = 0$, then $g(\mathbf{x}^*) > 0$ (called "inactive").

Duality

- We can rewrite the original (primal) problem into its dual form:
  - dual function: $q(\lambda) = \min_{\mathbf{x}} L(\mathbf{x}, \lambda)$
  - dual problem: $\max_{\lambda \geq 0} q(\lambda)$
- Solve for λ, rather than original variable $\mathbf{x}$.
  - can recover the value of $\mathbf{x}$ from λ.

- If the optimization problem is convex...
  - solving the dual is equivalent to solving the **primal**
  - $\min_{\mathbf{x}, g(\mathbf{x}) \geq 0} f(\mathbf{x}) = \max_{\lambda \geq 0} q(\lambda)$.
- **Note:** The SVM problem is convex, so we can obtain an equivalent dual problem.

Lagrange multipliers & SVM

- introduce a Langrange multiplier α_i for each constrain

$$L(\mathbf{w}, \alpha) = \frac{1}{2}\mathbf{w}^T\mathbf{w} - \sum_i \alpha_i[y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1]$$

- Lagrange multiplier tells us which points are on the margin:
  - If $\alpha_i = 0$, then $y_i(\mathbf{w}^T\mathbf{x}_i + b) > 1$ ($\mathbf{x}_i$ is beyond margin).
  - If $\alpha_i > 0$, then $y_i(\mathbf{w}^T\mathbf{x}_i + b) = 1$ ($\mathbf{x}_i$ is on the margin).
    - i.e., the point is a *support vector*.

SVM Dual Problem

- The SVM problem can be rewritten as a *dual* problem:

$$\underset{\alpha}{\operatorname{argmax}} \sum_i \alpha_i - \frac{1}{2}\sum_{i=1}^N \sum_{j=1}^N \alpha_i\alpha_j y_i y_j \Phi(\mathbf{x}_i)^T\Phi(\mathbf{x}_j)$$

$$\text{s.t.} \sum_{i=1}^N \alpha_i y_i = 0, \quad \alpha_i \geq 0$$

- The new variable α_i corresponds to each training sample $(\mathbf{x}_i, y_i)$.
  - instead of solving for $\mathbf{w}$, we now solve for α

- Recover the hyperplane $\mathbf{w}$ using α:
  - weighted combination of (transformed) data points.
  - $\mathbf{w} = \sum_{i=1}^N \alpha_i y_i \Phi(\mathbf{x}_i)$
- Classify a new point $\mathbf{x}_*$,

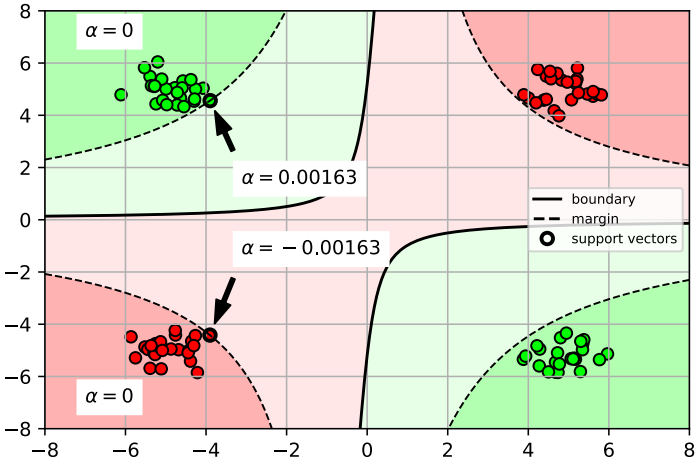
$$y_* = \text{sign}(\mathbf{w}^T\Phi(\mathbf{x}_*) + b) \tag{1}$$

$$= \text{sign}(\sum_{i=1}^N \alpha_i y_i \Phi(\mathbf{x}_i)^T\Phi(\mathbf{x}_*) + b) \tag{2}$$

- Interpretation of α_i
  - $\alpha_i = 0$ when the sample $\mathbf{x}_i$ is not on the margin.
  - $\alpha_i > 0$ when the sample $\mathbf{x}_i$ is on the margin (or violates).
    - i.e., the sample $\mathbf{x}_i$ is a support vector.

In [13]:
```
alfig
```

Out[13]:



Kernel function

- the SVM dual problem is completely written in terms of *inner product* between the high-dimensional feature vectors: $\Phi(\mathbf{x}_i)^T\Phi(\mathbf{x}_j)$
- So rather than explicitly calculate the high-dimensional vector $\Phi(\mathbf{x}_i)$,
  - we only need to calculate the inner product between two high-dim feature vectors.
- We call this a **kernel function**
  - $k(\mathbf{x}_i, \mathbf{x}_j) = \Phi(\mathbf{x}_i)^T\Phi(\mathbf{x}_j)$
  - calculating the kernel will be less expensive than explicitly calculating the high-dimensional feature vector and the inner product.

Example: Polynomial kernel

- input vector $\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} \in \mathbb{R}^d$
- kernel between two vectors is a p-th order polynomial:
  - $k(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^T\mathbf{x}')^p = (\sum_{i=1}^d x_i x_i')^p$
- For example, $p = 2$,

$$k(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^T\mathbf{x}')^2 = (\sum_{i=1}^d x_i x_i')^2 \tag{3}$$

$$= \sum_{i=1}^d \sum_{j=1}^d (x_i x_i' x_j x_j') = \Phi(\mathbf{x})^T\Phi(\mathbf{x}') \tag{4}$$

  - transformed feature space is the quadratic terms of the input vector:

$$\Phi(\mathbf{x}) = \begin{bmatrix} x_1x_1 \\ x_1x_2 \\ \vdots \\ x_2x_1 \\ x_2x_2 \\ \vdots \\ x_dx_1 \\ \vdots \\ x_dx_d \end{bmatrix}$$

- Comparison of number of multiplications
  - for kernel: $O(d)$
  - explicit transformation Φ: $O(d^2)$

Kernel trick

- Replacing the inner product with a kernel function in the optimization problem is called the **kernel trick**.
  - *turns a linear classification algorithm into a non-linear classification algorithm*.
  - the shape of the decision boundary is determined by the kernel.

Kernel SVM

- Replace inner product in linear SVM with kernel function:

$$\underset{\alpha}{\operatorname{argmax}} \sum_i \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j)$$

$$\text{s. t.} \sum_{i=1}^{N} \alpha_i y_i = 0, \quad \alpha_i \geq 0$$

- Prediction
  - $y_* = \operatorname{sign}(\sum_{i=1}^{N} \alpha_i y_i k(\mathbf{x}_i, \mathbf{x}_*) + b)$

Example: Kernel SVM with polynomial kernel

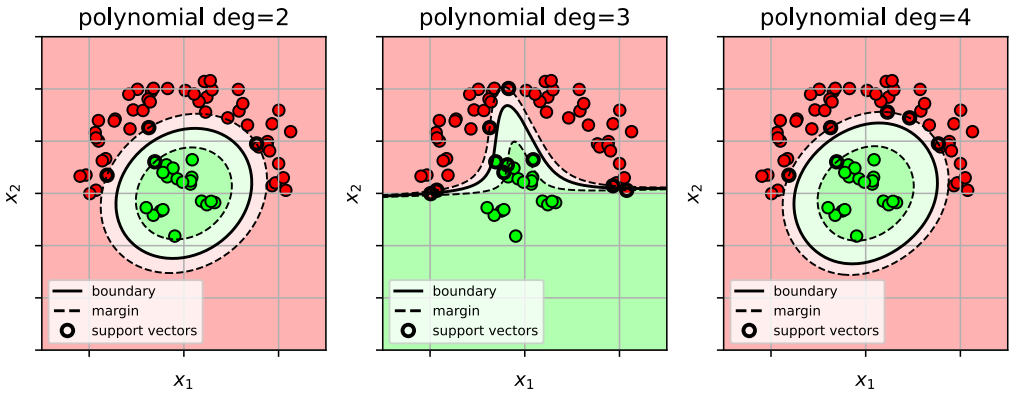
- decision surface is a "cut" of a polynomial surface
- higher polynomial-order yields more complex decision boundaries.

In [15]:
```
# fit SVM (poly kernel with different degrees)
degs = [2,3,4]

clf = {}
for d in degs:
    clf[d] = svm.SVC(kernel='poly', degree=d, C=100)
    clf[d].fit(X4, Y4)
```

In [17]:
```
polysvmfig
```

Out[17]:

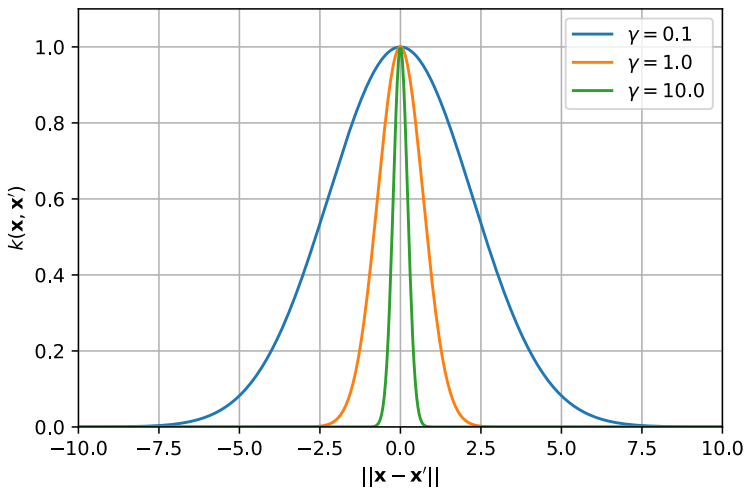


RBF kernel

- RBF kernel (radial basis function)
  - $k(\mathbf{x}, \mathbf{x}') = e^{-\gamma\|\mathbf{x}-\mathbf{x}'\|^2}$
  - similar to a Gaussian
- gamma $\gamma > 0$ is the inverse bandwidth parameter of the kernel
  - controls the smoothness of the function
  - small γ → wider Gaussian → smooth functions
  - large γ → thin Gaussian → wiggly function

In [19]:
```
rbffig
```

Out[19]:



Kernel SVM with RBF kernel

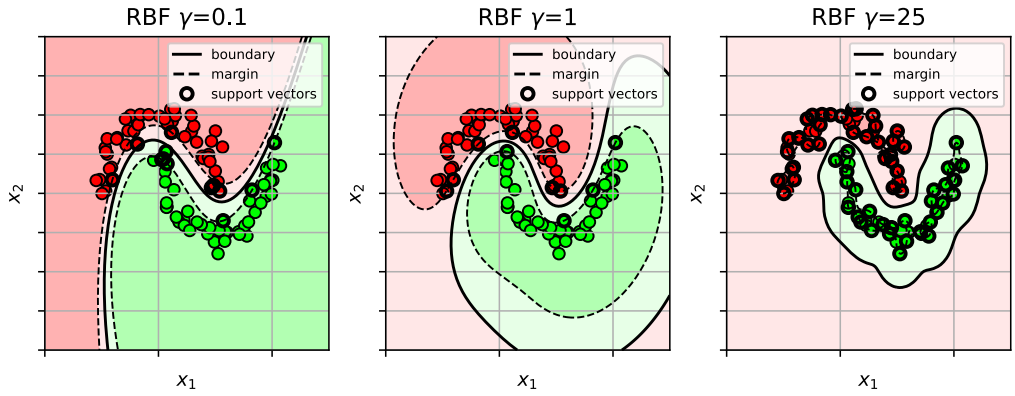
- try different γ
  - each γ yields different levels of smoothness of the decision boundary

In [20]:
```
# fit SVM (RBF)
gammas = [0.1, 1, 25]

clf = {}
for i in gammas:
    clf[i] = svm.SVC(kernel='rbf', gamma=i, C=1000)
    clf[i].fit(X3, Y3)
```

In [22]:
```
rbfsvmfig
```

Out[22]:

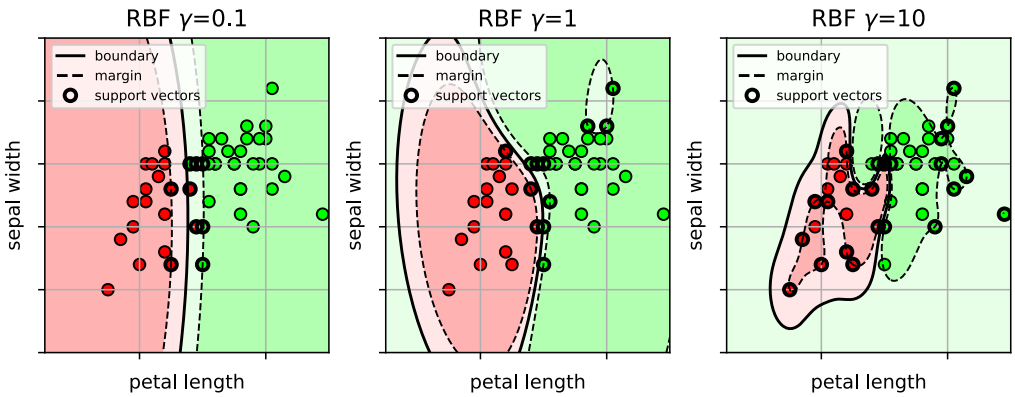


Example on Iris data

- Large γ yields a complicated wiggly decision boundary.

In [26]:
```
irbffig
```

Out[26]:



How to select the best kernel parameters?

- use cross-validation over possible kernel parameter (γ) and SVM C parameter
  - if a lot of parameters, can be computationally expensive!

In [27]:
```
# setup the list of parameters to try
paramgrid = {'C': logspace(-2,3,20),
             'gamma': logspace(-4,3,20) }

print(paramgrid)

# setup the cross-validation object
# pass the SVM object w/ rbf kernel, parameter grid, and number of CV folds
svmcv = model_selection.GridSearchCV(svm.SVC(kernel='rbf'), paramgrid, cv=5,
                                     n_jobs=-1, verbose=True)

# run cross-validation (train for each split)
svmcv.fit(trainX, trainY);

print("best params:", svmcv.best_params_)
```

```
{'C': array([1.00000000e-02, 1.83298071e-02, 3.35981829e-02, 6.15848211e-02,
       1.12883789e-01, 2.06913808e-01, 3.79269019e-01, 6.95192796e-01,
       1.27427499e+00, 2.33572147e+00, 4.28133240e+00, 7.84759970e+00,
```

```
       1.43844989e+01, 2.63665090e+01, 4.83293024e+01, 8.85866790e+01,
       1.62377674e+02, 2.97635144e+02, 5.45559478e+02, 1.00000000e+03]), 'gamm
a': array([1.00000000e-04, 2.33572147e-04, 5.45559478e-04, 1.27427499e-03,
       2.97635144e-03, 6.95192796e-03, 1.62377674e-02, 3.79269019e-02,
       8.85866790e-02, 2.06913808e-01, 4.83293024e-01, 1.12883789e+00,
       2.63665090e+00, 6.15848211e+00, 1.43844989e+01, 3.35981829e+01,
       7.84759970e+01, 1.83298071e+02, 4.28133240e+02, 1.00000000e+03])}
Fitting 5 folds for each of 400 candidates, totalling 2000 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  26 tasks      | elapsed:    1.4s
```

```
best params: {'C': 0.37926901907322497, 'gamma': 0.4832930238571752}
```

```
[Parallel(n_jobs=-1)]: Done 1420 tasks      | elapsed:    2.2s
[Parallel(n_jobs=-1)]: Done 1977 out of 2000 | elapsed:    2.4s remaining:
0.0s
[Parallel(n_jobs=-1)]: Done 2000 out of 2000 | elapsed:    2.4s finished
```

In [28]:

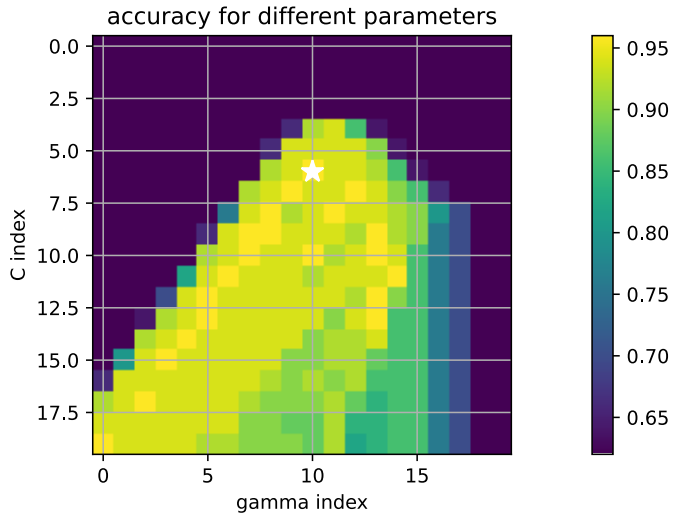
```
# show the test error for the first 25 parameter sets
N = 25
for m,p in zip(svmcv.cv_results_['mean_test_score'][0:N], svmcv.cv_results_['params'][0:N]):
    print("mean={:.4f} {}".format(m,p))
```

```
mean=0.6200 {'C': 0.01, 'gamma': 0.0001}
mean=0.6200 {'C': 0.01, 'gamma': 0.00023357214690901214}
mean=0.6200 {'C': 0.01, 'gamma': 0.000545559478116852}
mean=0.6200 {'C': 0.01, 'gamma': 0.0012742749857031334}
mean=0.6200 {'C': 0.01, 'gamma': 0.002976351441631319}
mean=0.6200 {'C': 0.01, 'gamma': 0.006951927961775605}
mean=0.6200 {'C': 0.01, 'gamma': 0.01623776739188721}
mean=0.6200 {'C': 0.01, 'gamma': 0.0379269019073225}
mean=0.6200 {'C': 0.01, 'gamma': 0.08858667904100823}
mean=0.6200 {'C': 0.01, 'gamma': 0.2069138081114788}
mean=0.6200 {'C': 0.01, 'gamma': 0.4832930238571752}
mean=0.6200 {'C': 0.01, 'gamma': 1.1288378916846884}
mean=0.6200 {'C': 0.01, 'gamma': 2.6366508987303554}
mean=0.6200 {'C': 0.01, 'gamma': 6.1584821106602545}
mean=0.6200 {'C': 0.01, 'gamma': 14.38449888287663}
mean=0.6200 {'C': 0.01, 'gamma': 33.59818286283781}
mean=0.6200 {'C': 0.01, 'gamma': 78.47599703514607}
mean=0.6200 {'C': 0.01, 'gamma': 183.29807108324337}
mean=0.6200 {'C': 0.01, 'gamma': 428.1332398719387}
mean=0.6200 {'C': 0.01, 'gamma': 1000.0}
mean=0.6200 {'C': 0.018329807108324356, 'gamma': 0.0001}
mean=0.6200 {'C': 0.018329807108324356, 'gamma': 0.00023357214690901214}
mean=0.6200 {'C': 0.018329807108324356, 'gamma': 0.000545559478116852}
mean=0.6200 {'C': 0.018329807108324356, 'gamma': 0.0012742749857031334}
mean=0.6200 {'C': 0.018329807108324356, 'gamma': 0.002976351441631319}
```

In [31]:

```
paramfig
```

Out[31]:



In [32]:
```
# show classifier with training data
plt.figure()
plot_posterior_svm(svmcv.best_estimator_, axbox, trainX)
plt.scatter(trainX[:,0], trainX[:,1], c=trainY, cmap=mycmap, edgecolors='k')
plt.xlabel('petal length'); plt.ylabel('sepal width')
plt.title('class boundary with training data');
plt.show()
```

In [33]:
```
# predict from the model
predY = svmcv.predict(testX)

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

```
test accuracy = 0.88
```

Custom kernel function

- we can use any kernel function, as long as it is *positive definite*:
  - 1) it can be written as an inner-product of a feature transformation: $k(\mathbf{x}_1, \mathbf{x}_2) = \langle \Phi(\mathbf{x}_1), \Phi(\mathbf{x}_2) \rangle$.
  - 2) for all possible datasets $\mathbf{X} = \{\mathbf{x}_1, \cdots \mathbf{x}_N\}$ of all possible sizes N, the kernel matrix $K = [k(\mathbf{x}_i, \mathbf{x}_j)]_{i,j}$ is a positive definite matrix.
    - $\mathbf{K}$ is a *positive definite matrix* iff $\mathbf{z}^T \mathbf{K} \mathbf{z} > 0, \forall \mathbf{z}$

- in sklearn, pass a callable function as the kernel parameter.

Example: Laplacian kernel

- $k(\mathbf{x}_1, \mathbf{x}_2) = \exp(-\alpha||\mathbf{x}_1 - \mathbf{x}_2||)$

In [34]:
```
from scipy import spatial

# create a custom kernel function
# Laplacian kernel: exp( -alpha*||x1-x2|| )
def mykernel(X1, X2, alpha=1.0):
    # X1,X2 are (N1 x d) and (N2 x d) matrices of N1 and N2 d-dim vectors
    # alpha is the hyperparameter

    # compute pairwise euclidean distance
    D = spatial.distance.cdist(X1, X2, metric='euclidean')

    # return the kernel matrix
    return exp(-alpha*D)
```

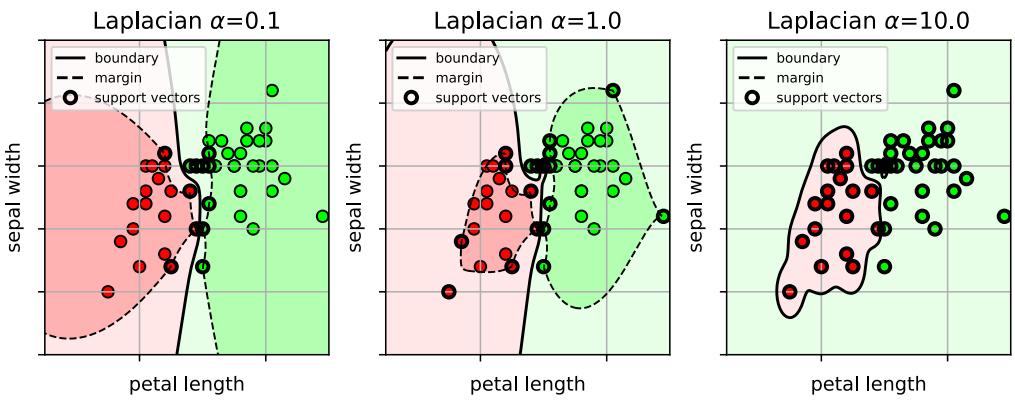
In [35]:
```
alphas = [0.1, 1., 10.]

clf = {}
for i in alphas:
    # make a temporary kernel function with the selected alpha value
    tmpkern = lambda X1,X2,alpha=i: mykernel(X1,X2,alpha=alpha)

    # create the SVM with custom kernel function
    clf[i] = svm.SVC(kernel=tmpkern, C=100)
    clf[i].fit(trainX, trainY)
```

In [37]:
```
ilapfig
```

Out[37]:



Kernel SVM Summary

- **Kernel Classifier:**
  - Kernel function defines the shape of the non-linear decision boundary.
    - implicitly transforms input feature into high-dimensional space.
    - uses linear classifier in high-dim space.
    - the decision boundary is non-linear in the original input space.
- **Training:**
  - Maximize the margin of the training data.
    - i.e., maximize the separation between the points and the decision boundary.
  - Use cross-validation to pick the hyperparameter C and the kernel hyperparameters.

- **Advantages:**
  - non-linear decision boundary for more complex classification problems
  - some intuition from the type of kernel function used.
  - kernel function can also be used to do non-vector data.
- **Disadvantages:**
  - sensitive to the kernel function used.
  - sensitive to the C and kernel hyperparameters.
  - computationally expensive to do cross-validation.
  - need to calculate the kernel matrix
    - N^2 terms where N is the size of the training set
    - for large N, uses a large amount of memory and computation.

Kernels on other types of data

- **Histograms:** $\mathbf{x} = [x_1, \cdots, x_d]$, x_i is a bin value.
  - Bhattacharyya:

$$k(\mathbf{x}, \mathbf{x}') = \sum_{i=1}^{d} \sqrt{x_i x_i'}$$

  - histogram intersection:

$$k(\mathbf{x}, \mathbf{x}') = \sum_{i} \min(x_i, x_i')$$

  - χ^2-RBF:

$$k(\mathbf{x}, \mathbf{x}') = e^{-\gamma \chi^2(\mathbf{x}, \mathbf{x}')}$$

    - γ is a inverse bandwidth parameter
    - χ^2 distance: $\chi^2(\mathbf{x}, \mathbf{x}') = \sum_{i=1}^{d} \frac{(x_i - x_i')^2}{\frac{1}{2}(x_i + x_i')}$

- **Strings**: $\mathbf{x}$ = "...." (strings can be different sizes)

$$k(\mathbf{x}, \mathbf{x}') = \sum_{s} w_s \phi_s(\mathbf{x}) \phi_s(\mathbf{x}')$$

  - $\phi_s(\mathbf{x})$ is the number of times substring s appears in $\mathbf{x}$.
  - $w_s > 0$ is a weight.
- **Sets**: $\mathbf{X} = \{\mathbf{x}_1, \cdots, \mathbf{x}_n\}$ (sets can be different sizes)
  - intersection kernel:

$$k(\mathbf{X}, \mathbf{X}') = 2^{|\mathbf{X} \cap \mathbf{X}'|}$$

    - $|\mathbf{X} \cap \mathbf{X}'|$ = number of common elements.