

Support Vector Machines and kernelization

```
[27]: # Global imports and settings
      from preamble import *
      %matplotlib inline
      plt.rcParams['savefig.dpi'] = 120 # Use 300 for PDF, 100 for slides
      #InteractiveShell.ast_node_interactivity = "all"
      HTML(''<style>html, body{overflow-y: visible !important} .CodeMirror{min-
<IPython.core.display.HTML object>
```

Linear SVMs

Revisited

Linear models for Classification (recap)

Aims to find a (hyper)plane that separates the examples of each class.

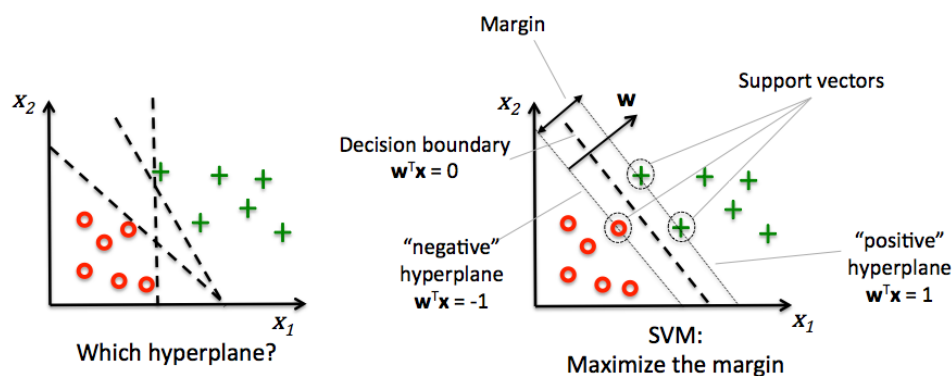
For binary classification (2 classes), we aim to fit the following function:

$$\hat{y} = w_0 * x_0 + w_1 * x_1 + \dots + w_p * x_p + b > 0$$

When $\hat{y} < 0$, predict class -1, otherwise predict class +1

Support vector machines

- In several other linear models, we minimized (misclassification) error
- In SVMs, the optimization objective is to maximize the *margin*
- The **margin** is the distance between the separating hyperplane and the *support vectors*
- The **support vectors** are the training samples closest to the hyperplane
- Intuition: large margins generalize better, small margins may be prone to overfitting



SVC Image

Maximum margin For now, we assume that the data is linearly separable.

The *positive hyperplanes* is defined as:

$$b + \mathbf{w}^T \mathbf{x}_+ = 1$$

with \mathbf{x}_+ the positive support vectors.

Likewise, the *negative hyperplanes* is defined as:

$$b + \mathbf{w}^T \mathbf{x}_- = -1$$

Subtracting them yields:

$$\mathbf{w}^T(\mathbf{x}_+ - \mathbf{x}_-) = 2$$

We can normalize by the length of vector w , defined as

$$\|w\| = \sqrt{\sum_{j=1}^m w_j^2}$$

Yielding

$$\frac{\mathbf{w}^T(\mathbf{x}_+ - \mathbf{x}_-)}{\|w\|} = \frac{2}{\|w\|}$$

The left side can be interpreted as the distance between to positive and negative hyperplane, which is the *margin* that we want to maximize.

Hence, we want to maximize $\frac{2}{\|w\|}$ under the constraint that all samples are classified correctly:

$$\begin{aligned} b + \mathbf{w}^T \mathbf{x}^{(i)} &\geq 1 \quad \text{if } y^{(i)} = 1 \\ b + \mathbf{w}^T \mathbf{x}^{(i)} &\leq -1 \quad \text{if } y^{(i)} = -1 \end{aligned}$$

i.e. all negative examples should fall on one side of the negative hyperplane and vice versa.

Or:

$$y^{(i)}(b + \mathbf{w}^T \mathbf{x}^{(i)}) \geq 1 \quad \forall i$$

Maximizing $\frac{2}{\|w\|}$ can be done by minimizing $\frac{\|w\|^2}{2}$

This is a quadratic objective with linear constraints, and can hence be solved using quadratic programming, and more specifically with the *Lagrangian multiplier method*.

Primal and Dual formulations The Primal formulation of the Lagrangian objective function is:

$$\min L_P = \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^l a_i y_i (\mathbf{x}_i \cdot \mathbf{w} + b) + \sum_{i=1}^l a_i$$

so that

$$\begin{aligned} a_i &\geq 0 \\ \mathbf{w} &= \sum_{i=1}^l a_i y_i \mathbf{x}_i \\ \sum_{i=1}^l a_i y_i &= 0 \end{aligned}$$

with l the number of training examples and a the *dual variable*, which acts like a weight for each training example.

It has a Dual formulation as follows:

$$\min L_D(a_i) = \sum_{i=1}^l a_i - \frac{1}{2} \sum_{i,j=1}^l a_i a_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j)$$

so that

$$\begin{aligned} a_i &\geq 0 \\ \sum_{i=1}^l a_i y_i &= 0 \end{aligned}$$

See 'Elements of Statistical Learning' for more detail.
Why are we doing this?

- Because now we can solve the problem by just computing the inner products of $\mathbf{x}_i, \mathbf{x}_j$, which will be important when we want to solve non-linearly separable cases.

Ok, what now?

- Knowing the dual coefficients a_i we can find the weights w for the maximal margin separating hyperplane:

$$\mathbf{w} = \sum_{i=1}^l a_i y_i \mathbf{x}_i$$

- Hence, we can classify a new sample \mathbf{u} by looking at the sign of $\mathbf{w} * \mathbf{u} + b$
- Most of the a_i will turn out to be 0
- The training samples for which a_i is not 0 are the *support vectors*
- Hence, the SVM model is completely defined by the support vectors and their coefficients

SVMs and kNN

Remember, we will classify a new sample u by looking at the sign of:

$$f(x) = \mathbf{w} * \mathbf{u} + b = \sum_{i=1}^l a_i y_i \mathbf{x}_i * \mathbf{u} + b$$

Weighted k-nearest neighbor is a generalization of the k-nearest neighbor classifier would classify by looking at the sign of:

$$f(x) = \sum_{i=1}^k a_i y_i \text{dist}(x_i, u)$$

Hence: SVM's predict exactly the same way as k-NN, only: - They only consider the truly important points (the support vectors) - Thus *much* faster - The number of neighbors is the number of support vectors - The distance function can be different

SVMs in scikit-learn

- We can use the `svm.SVC` classifier
 - or `svm.SVR` for regression
- To build a linear SVM use `kernel=linear`
- It returns the following:
 - `support_vectors_`: the support vectors
 - `dual_coef_`: the dual coefficients a , i.e. the weights of the support vectors
 - `coef_`: only for linear SVMs, the feature weights w

```
[28]: from sklearn import svm
```

```
# Linearly separable data
```

```
X = np.r_[np.random.randn(20, 2) - [2, 2], np.random.randn(20, 2) + [2, 2]]
Y = [0] * 20 + [1] * 20
```

```
# Fit the model
```

```
clf = svm.SVC(kernel='linear')
clf.fit(X, Y)
```

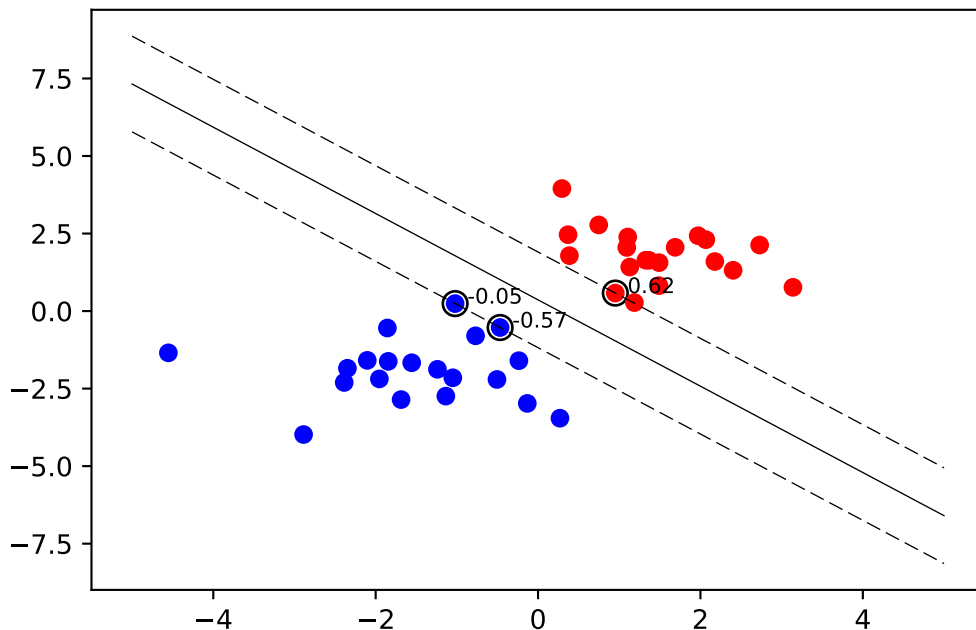
```
# Get the support vectors and weights
```

```
print("Support vectors:")
print(clf.support_vectors_[:])
print("Coefficients:")
print(clf.dual_coef_[:])
```

```
Support vectors:
[[-0.244 -1.027]
 [ 1.299  0.232]]
Coefficients:
[[-0.504  0.504]]
```

SVM result. The circled samples are support vectors, together with their coefficients.

```
[29]: mglearn.plots.plot_svm_linear()
```



Dealing with nonlinearly separable data

- If the data is not linearly separable, (hard) margin maximization becomes meaningless
 - The constraints would contradict
- We can allow for violations of the margin constraint by introducing *slack variables* $\xi^{(i)}$

$$b + \mathbf{w}^T \mathbf{x}^{(i)} \geq 1 - \xi^{(i)} \quad \text{if } y^{(i)} = 1$$

$$b + \mathbf{w}^T \mathbf{x}^{(i)} \leq -1 + \xi^{(i)} \quad \text{if } y^{(i)} = -1$$

- The new objective (to be minimized) becomes:

$$\frac{\|\mathbf{w}\|^2}{2} + C(\sum_i \xi^{(i)})$$

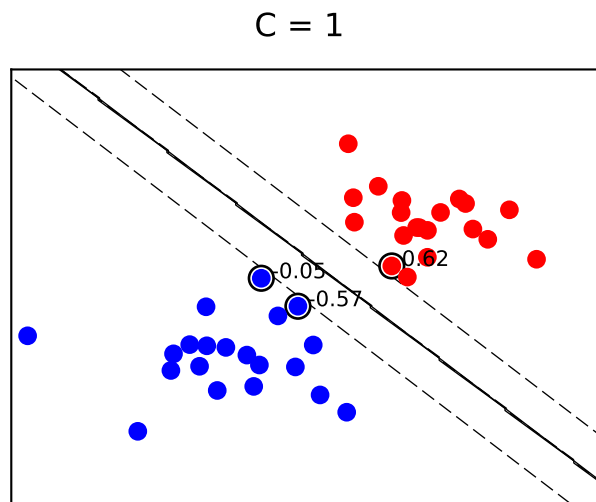
- C is a penalty for misclassification
 - Large C : large error penalties
 - Small C : less strict about violations (more regularization)
- This is known as the *soft margin SVM* (or *large margin SVM*)
 - Some support vectors are exactly on the margin hyperplane, with margin = 1
 - Others are margin violators, with margin < 1 and a positive slack variable: $\xi^{(i)} > 0$
 - * If $\xi^{(i)} \geq 1$, they are misclassified

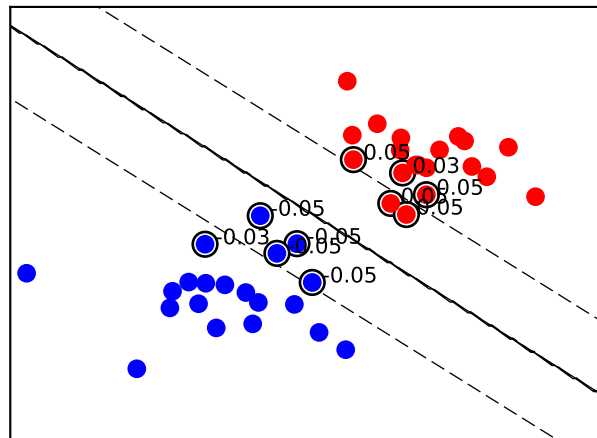
C and regularization

- Hence, we can use C to control the size of the margin and tune the bias-variance trade-off
 - Small C : Increases bias, reduces variance, more underfitting
 - Large C : Reduces bias, increases variance, more overfitting
- The penalty term $C(\sum_i \xi^{(i)})$ acts as an L1 regularizer on the dual coefficients
 - Also known as hinge loss
 - This induces sparsity: large C values will set many dual coefficients to 0, hence fewer support vectors
 - Small C values will typically lead to more support vectors (more points fall within the margin)
 - Again, it depends on the data how flexible or strict you need to be
- The *least squares SVM* is a variant that does L2 regularization
 - Will have many more support vectors (with low weights)
 - In `scikit-learn`, this is only available for the `LinearSVC` classifier (`loss='squared_hinge'`)

Effect on linearly separable data

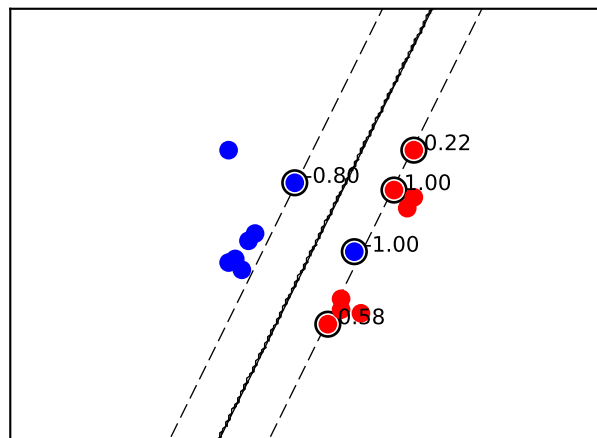
```
[30]: mglearn.plots.plot_svm_margins()
```

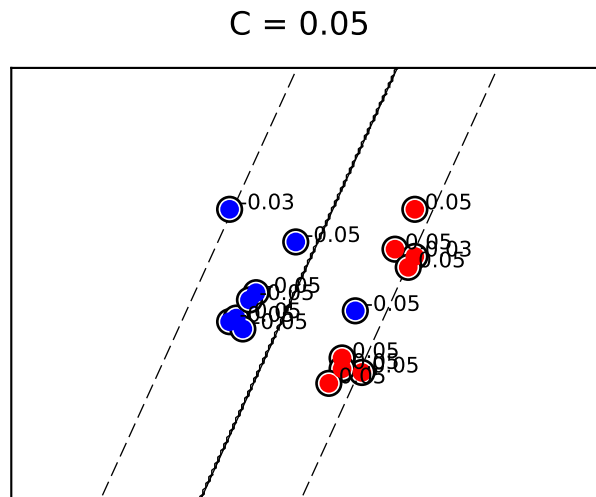


$$C = 0.05$$


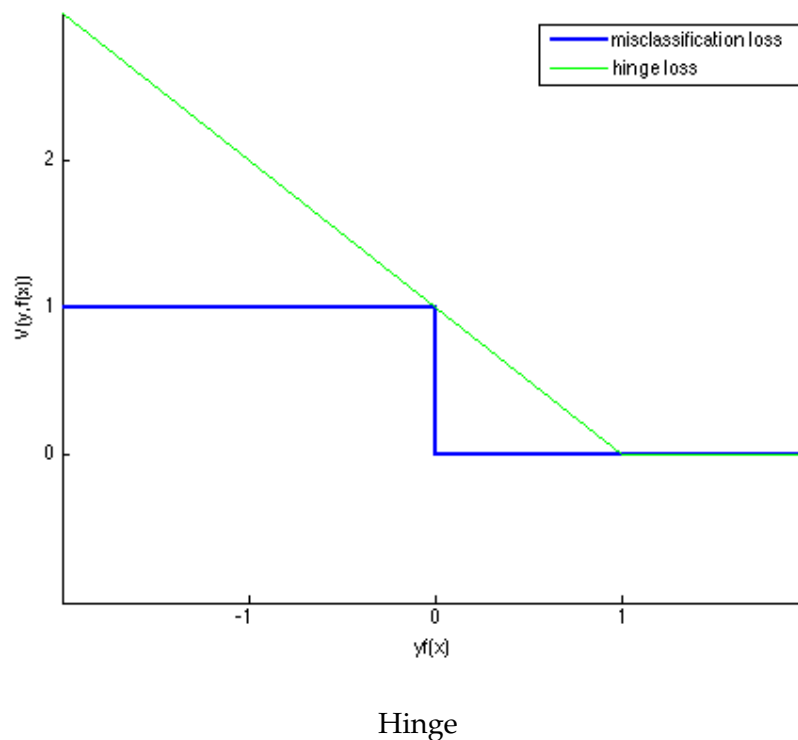
Effect on non-linearly separable data

```
[31]: mglearn.plots.plot_svm_margins_nonlin()
```

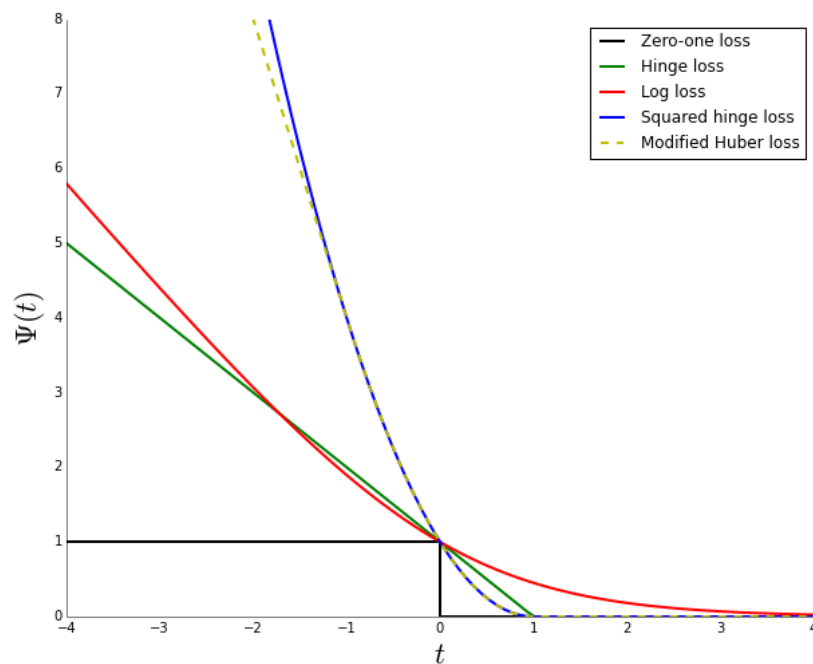
$$C = 1$$




Hinge loss We are trying to: - Maximize the margin - Minimize the sum of margin violations
 We could also try to maximize the margin and minimize the number of misclassifications -
 Turns out that the corresponding objective function is not convex, NP-hard
 The best convex relation is hinge loss: $L(\gamma) = \max\{0, 1 - \gamma\}$
 It measures the margin violation ξ_i .



Other loss functions It is possible to use generalize SVMs by training them with other loss functions and gradient descent as the optimizer
 See the `SGDClassifier - SGDClassifier(loss='hinge')` will act like an SVM



Hinge

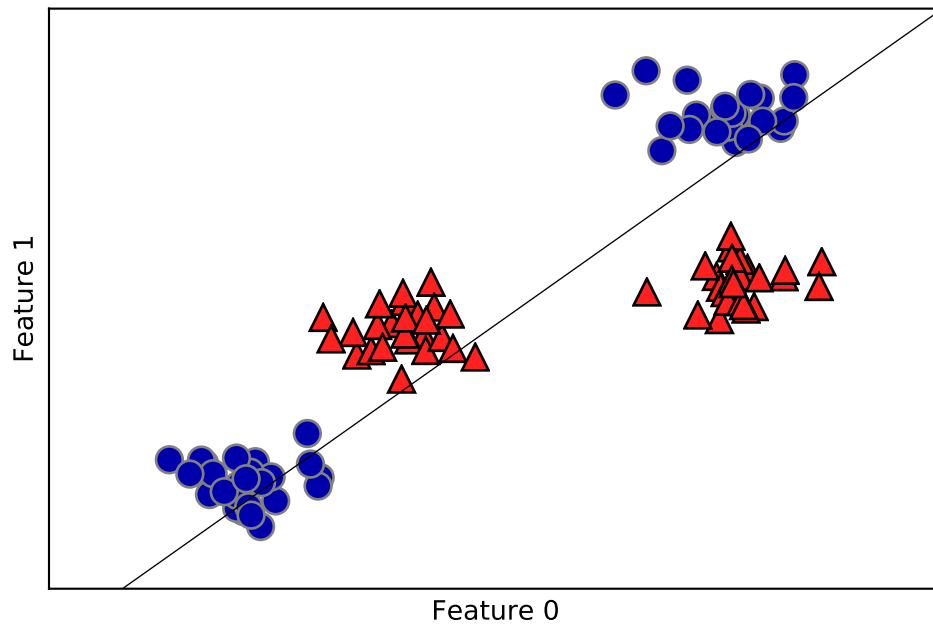
Kernelized Support Vector Machines

- Linear models work well in high dimensional spaces.
- You can *create* additional dimensions yourself.
- Let's start with an example.

Our linear model doesn't fit the data well

```
[32]: from sklearn.datasets.samples_generator import make_blobs
      from sklearn.svm import LinearSVC
      X, y = make_blobs(centers=4, random_state=8)
      y = y % 2
      linear_svm = LinearSVC().fit(X, y)

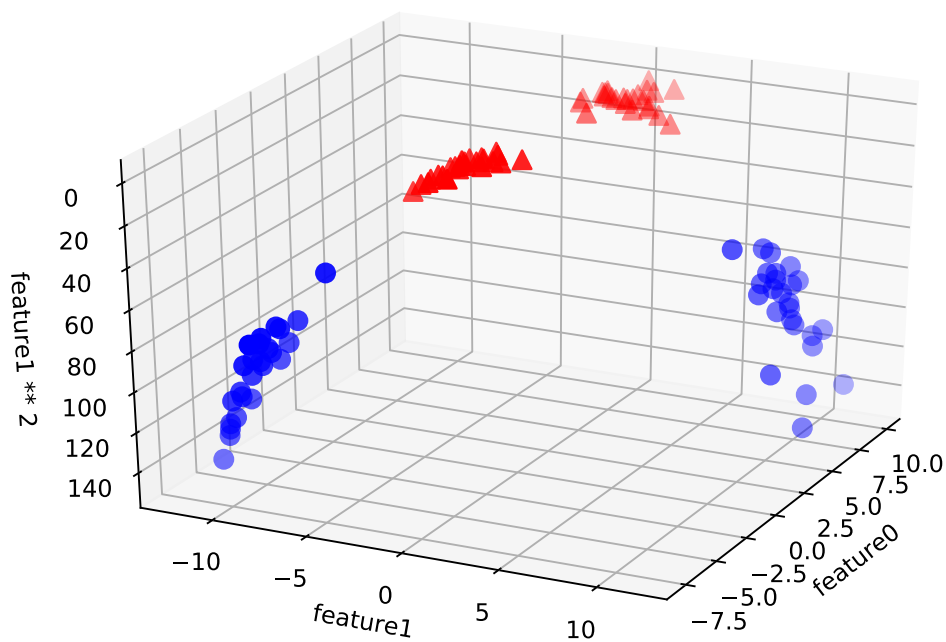
      mglearn.plots.plot_2d_separator(linear_svm, X)
      mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
      plt.xlabel("Feature 0")
      plt.ylabel("Feature 1");
```

We can add a new feature by taking the squares of feature1 values

```
[33]: # add the squared first feature
X_new = np.hstack([X, X[:, 1:] ** 2])

from mpl_toolkits.mplot3d import Axes3D, axes3d
figure = plt.figure()
# visualize in 3D
ax = Axes3D(figure, elev=-152, azimuth=-26)
# plot first all the points with y==0, then all with y == 1
mask = y == 0
ax.scatter(X_new[mask, 0], X_new[mask, 1], X_new[mask, 2], c='b',
           cmap=mglearn.cm2, s=60)
ax.scatter(X_new[~mask, 0], X_new[~mask, 1], X_new[~mask, 2], c='r', marker='^',
           cmap=mglearn.cm2, s=60)
ax.set_xlabel("feature0")
ax.set_ylabel("feature1")
ax.set_zlabel("feature1 ** 2");
```



Now we can fit a linear model

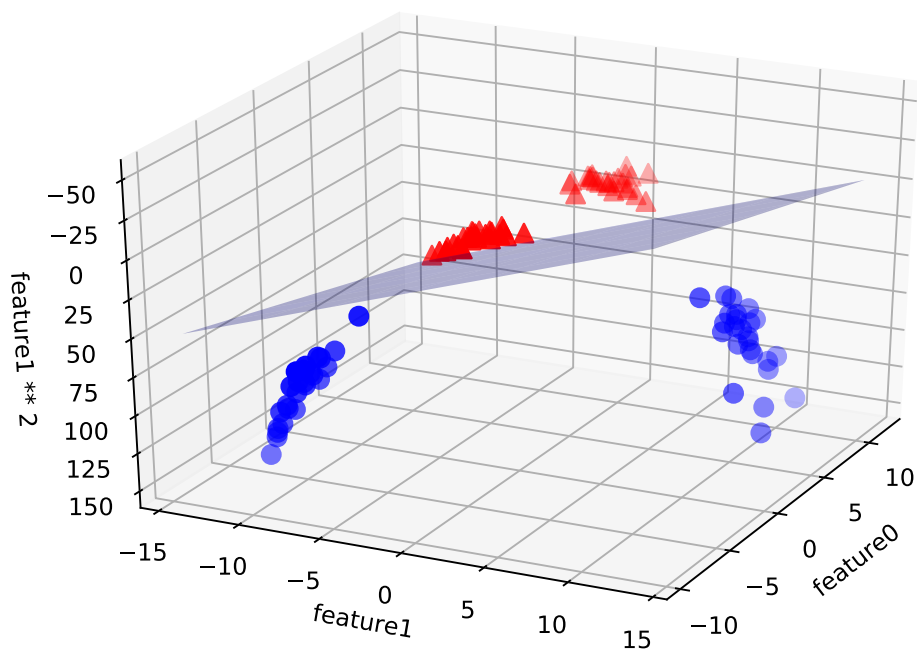
```
[34]: linear_svm_3d = LinearSVC().fit(X_new, y)
      coef, intercept = linear_svm_3d.coef_.ravel(), linear_svm_3d.intercept_

      # show linear decision boundary
      figure = plt.figure()
      ax = Axes3D(figure, elev=-152, azimuth=-26)
      xx = np.linspace(X_new[:, 0].min() - 2, X_new[:, 0].max() + 2, 50)
      yy = np.linspace(X_new[:, 1].min() - 2, X_new[:, 1].max() + 2, 50)

      XX, YY = np.meshgrid(xx, yy)
      ZZ = (coef[0] * XX + coef[1] * YY + intercept) / -coef[2]
      ax.plot_surface(XX, YY, ZZ, rstride=8, cstride=8, alpha=0.3)
      ax.scatter(X_new[mask, 0], X_new[mask, 1], X_new[mask, 2], c='b',
                  cmap=mglearn.cm2, s=60)
      ax.scatter(X_new[~mask, 0], X_new[~mask, 1], X_new[~mask, 2], c='r', marker='^',
                  cmap=mglearn.cm2, s=60)

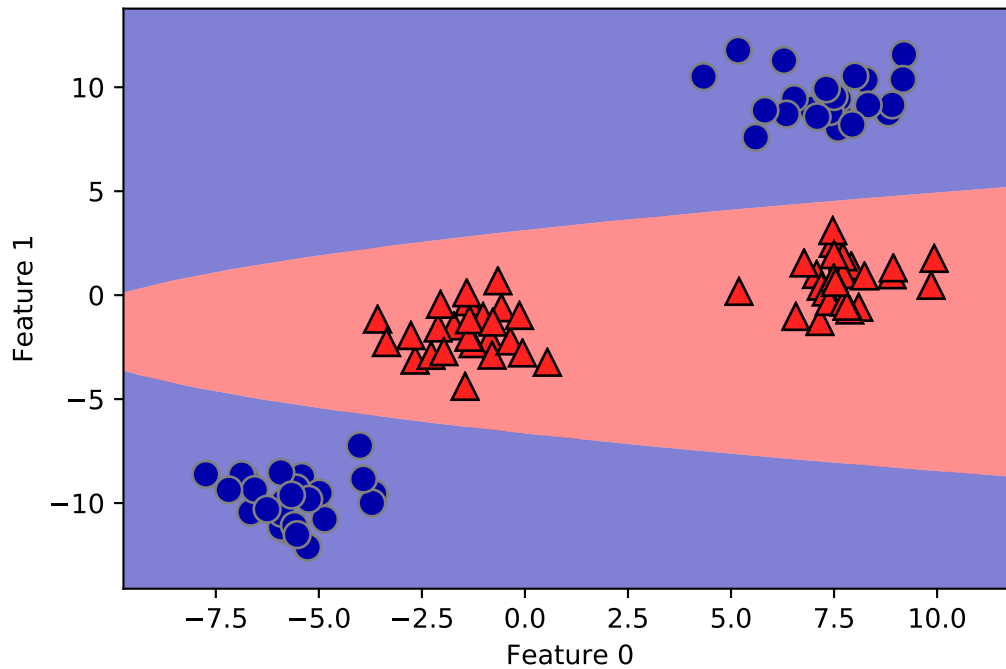
      ax.set_xlabel("feature0")
      ax.set_ylabel("feature1")
      ax.set_zlabel("feature1 ** 2")

      <matplotlib.text.Text at 0x11dd7f8d0>
```



As a function of the original features, the linear SVM model is not actually linear anymore, but more of an ellipse

```
[35]: ZZ = YY ** 2
      dec = linear_svm_3d.decision_function(np.c_[XX.ravel(), YY.ravel(), ZZ.ravel()])
      plt.contourf(XX, YY, dec.reshape(XX.shape), levels=[dec.min(), 0, dec.max()],
                    cmap=mglearn.cm2, alpha=0.5)
      mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
      plt.xlabel("Feature 0")
      plt.ylabel("Feature 1");
```



Kernels

A (Mercer) Kernel on a space X is a (similarity) function

$$k : X \times X \rightarrow \mathbb{R}$$

Of two arguments with the properties:

- Symmetry: $k(x_1, x_2) = k(x_2, x_1) \quad \forall x_1, x_2 \in X$
- Positive definite: for each finite subset of data points x_1, \dots, x_n , the kernel Gram matrix is positive semi-definite

Kernel matrix = $K \in \mathbb{R}^{n \times n}$ with $K_{ij} = k(x_i, x_j)$

What is this good for?

Mercer's Theorem states that

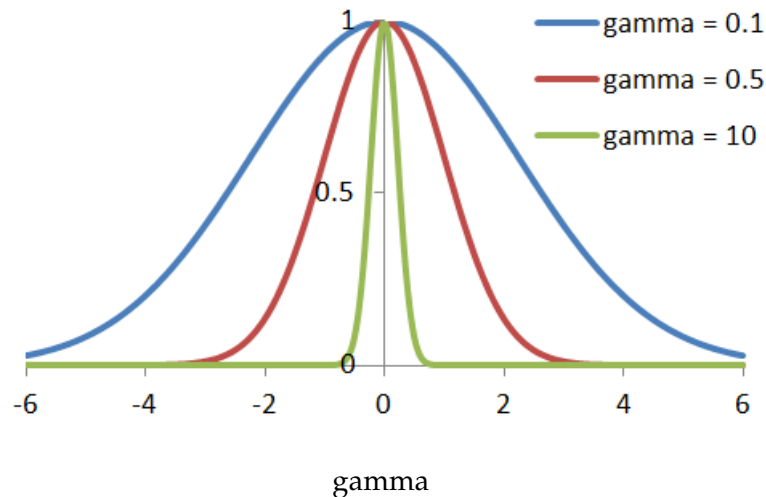
- there exists a Hilbert space \mathcal{H} of continuous functions $X \rightarrow \mathbb{R}$ - basically, a possibly infinite-dimensional vector space with inner product where all operations are meaningful - and a continuous "feature map" $\phi : X \rightarrow \mathcal{H}$ - so that the kernel computes the inner product of the features $k(x_1, x_2) = \langle \phi(x_1), \phi(x_2) \rangle$

Hence, a kernel can be thought of as a 'shortcut' computation for the 2-step procedure feature map + inner product

Kernels: examples

- The inner product is a kernel. The standard inner product is the **linear kernel**:
 $k(x_1, x_2) = x_1^T x_2$
- Kernels can be constructed from other kernels k_1 and k_2 :
 - For $\lambda \geq 0$, $\lambda.k_1$ is a kernel
 - $k_1 + k_2$ is a kernel

- $k_1.k_2$ is a kernel (thus also k_1^n)
- This allows to construct the **polynomial kernel**:
 $k(x_1, x_2) = (x_1^T x_2 + b)^d$, for $b \geq 0$ and $d \in \mathbb{N}$
- The ‘radial’ **Gaussian kernel** is defined as:
 $k(x_1, x_2) = \exp(-\gamma \|x_1 - x_2\|^2)$, for $\gamma \geq 0$

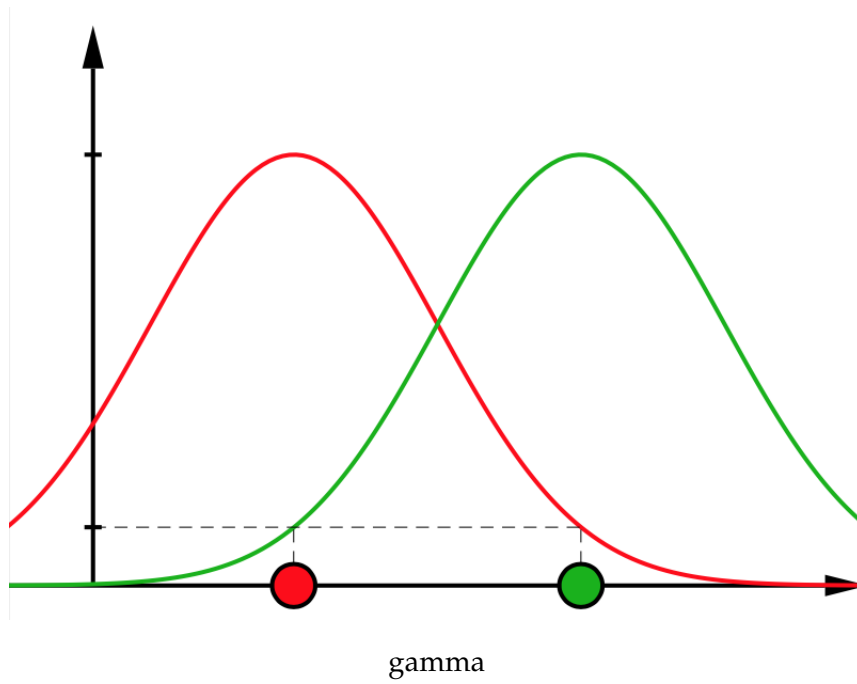


The Kernel Trick

- Adding nonlinear features can make linear models much more powerful
- Often we don't know which features to add, and adding many features might make computation very expensive
- Mathematical trick (*kernel trick*) allows us to directly compute distances (scalar products) in the high dimensional space
 - We can search for the nearest support vector in the high dimensional space
- A *kernel function* is a distance (similarity) function with special properties for which this trick is possible
 - Polynomial kernel: computes all polynomials up to a certain degree of the original features
 - Gaussian kernel, or radial basis function (RBF): considers all possible polynomials of all degrees
 - * Infinite high dimensional space (Hilbert space), where the importance of the features decreases for higher degrees

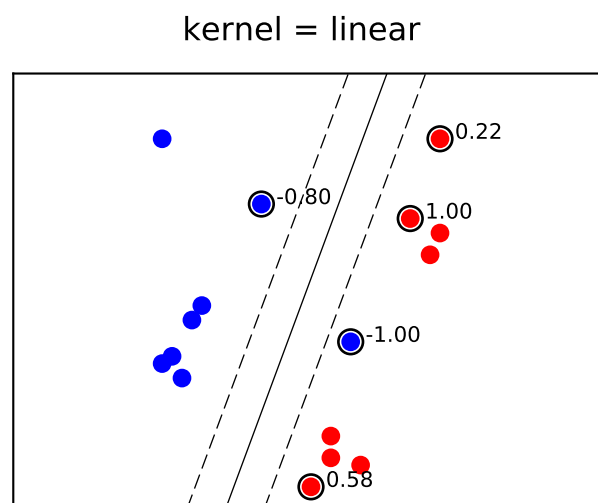
The kernel trick: intuition

- There exists many feature map (and hence Hilbert space) for the same kernel, but they are all equivalent
- The Reproducing Kernel Hilbert Space (RKHS) has feature map
 $\phi : X \rightarrow C(X); x \rightarrow k(x, \cdot)$ Where C is the space of continuous functions $X \rightarrow \mathbb{R}$
- Thus, an input $x \in X$ is mapped to the basis function $\phi(x) = k(x, \cdot)$
 - For every point, the mappings are continuous functions $k(x, \cdot)$

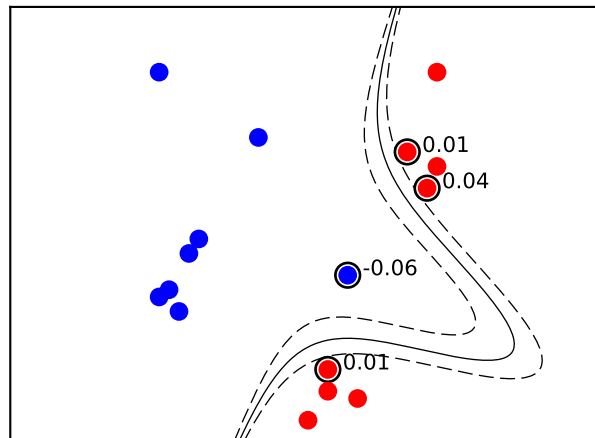


- Kernel computes $\langle k(x_1, \cdot), k(x_2, \cdot) \rangle = k(x_1, x_2)$

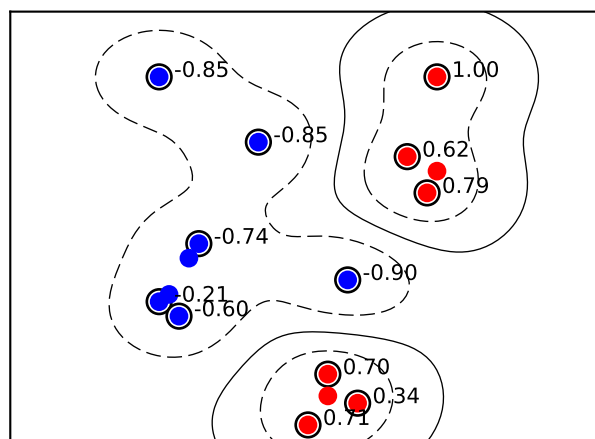
```
[36]: mglearn.plots.plot_svm_kernels()
```



kernel = poly



kernel = rbf



Understanding SVMs

To make a prediction for a new point, the distance to each of the support vectors is measured.

- The weight of each support vector is stored in the `dual_coef_` attribute of `SVC`
- The distance between data points is measured by the kernel
 - Gaussian kernel: $krbf(x_1, x_2) = \exp(\gamma \|x_1 - x_2\|^2)$
 - * γ controls the width of the kernel and can be tuned

Given the support vectors, their weights, and the kernel, we can plot the decision boundary

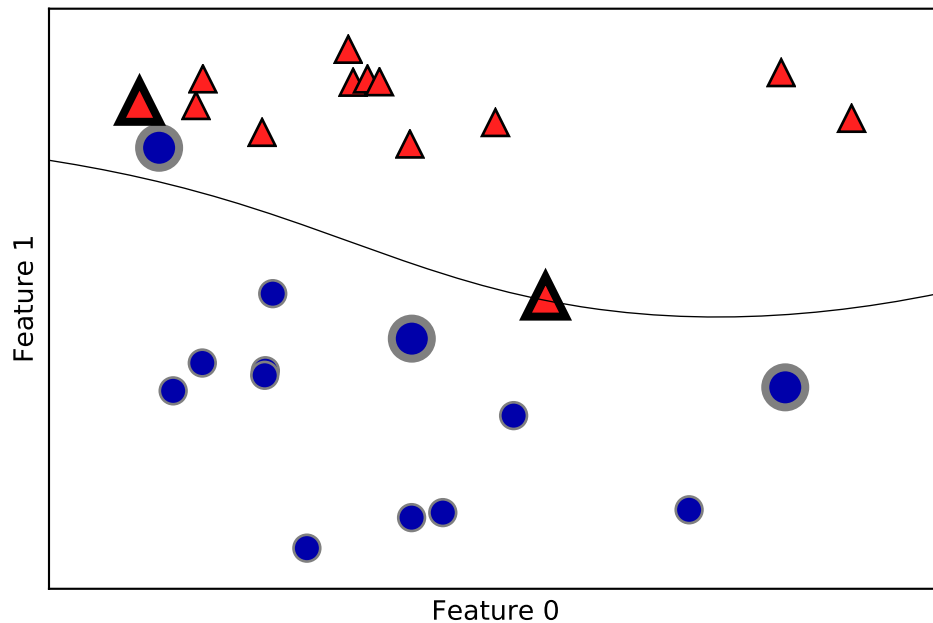
```
[37]: from sklearn.svm import SVC
```

```
X, y = mglearn.tools.make_handcrafted_dataset()
```

```

svm = SVC(kernel='rbf', C=10, gamma=0.1).fit(X, y)
mglearn.plots.plot_2d_separator(svm, X, eps=.5)
# plot data
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
# plot support vectors
sv = svm.support_vectors_
# class labels of support vectors are given by the sign of the dual coefficient
sv_labels = svm.dual_coef_.ravel() > 0
mglearn.discrete_scatter(sv[:, 0], sv[:, 1], sv_labels, s=15, markeredgewidth=3)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1");

```



Tuning SVM parameters

Several important parameters:

- gamma (kernel width): high values means that points are further apart
 - Leads to many support vectors, narrow Gaussians, overfitting
 - Low values lead to underfitting
- C (our linear regularizer): limits the weights of the support vectors
 - Higher values: more regularization, less overfitting
- For polynomial kernels, the *degree* (exponent) defines the complexity of the models

```

[38]: plt.rcParams.update({'font.size': 14})
fig, axes = plt.subplots(3, 3, figsize=(15, 10))

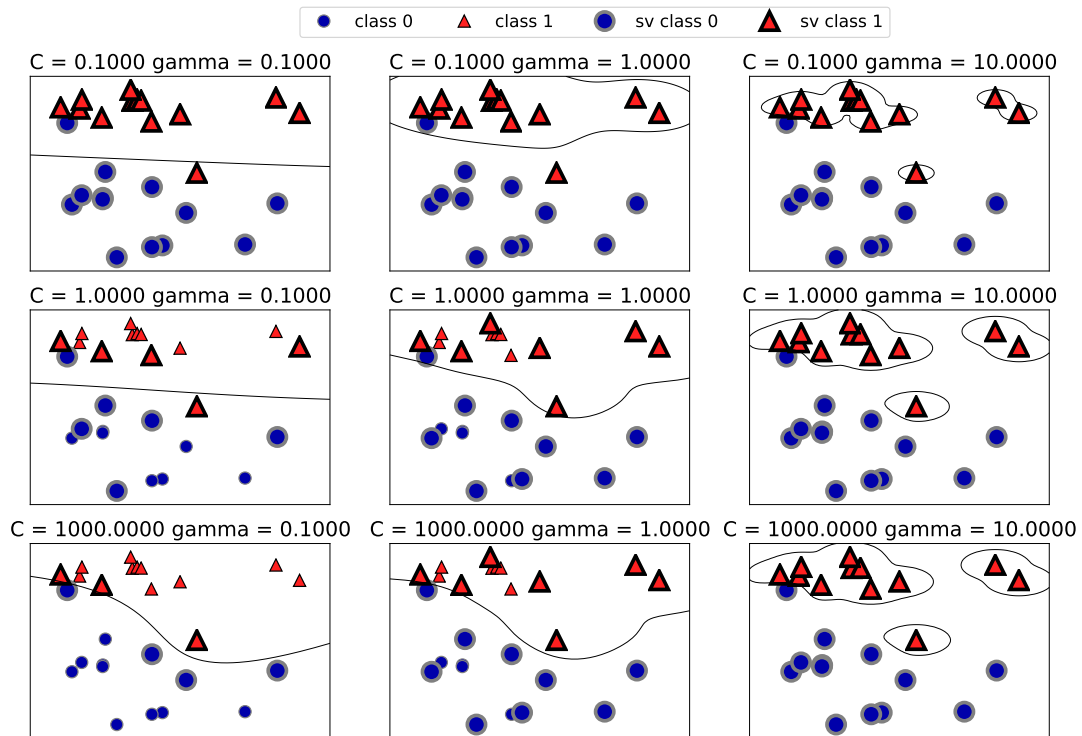
for ax, C in zip(axes, [-1, 0, 3]):
    for a, gamma in zip(ax, range(-1, 2)):

```



```
mglearn.plots.plot_svm(log_C=C, log_gamma=gamma, ax=a)

axes[0, 0].legend(["class 0", "class 1", "sv class 0", "sv class 1"],
                  ncol=4, loc=(.9, 1.2));
```

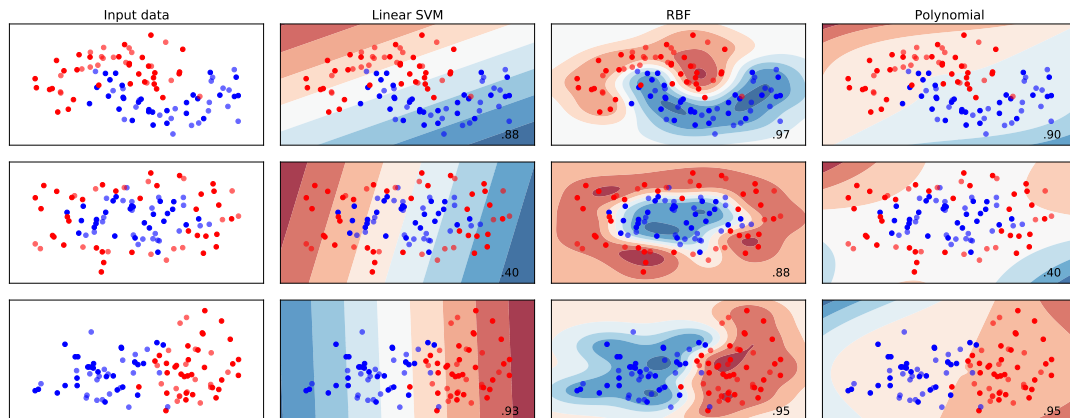


- Low gamma (left): wide Gaussians, very smooth decision boundaries
- High gamma (right): narrow Gaussians, boundaries focus on single points (high complexity)
- Low C (top): each support vector has very limited influence: many support vectors, almost linear decision boundary
- High C (bottom): Stronger influence, decision boundary bends to every support vector

```
[39]: import plot_classifiers
names = ["Linear SVM", "RBF", "Polynomial"]

classifiers = [
    SVC(kernel="linear", C=0.025),
    SVC(gamma=2, C=1),
    SVC(kernel="poly", degree=3, C=0.1)
]

plot_classifiers.plot_classifiers(names, classifiers, figsize=(20,8))
```



Preprocessing Data for SVMs

- SVMs are very sensitive to hyperparameter settings
- They expect all features to be approximately on the same scale
 - If not, they overfit easily

```
[40]: from sklearn.model_selection import train_test_split
      from sklearn.datasets import load_breast_cancer
      cancer = load_breast_cancer()

      X_train, X_test, y_train, y_test = train_test_split(
          cancer.data, cancer.target, random_state=0)

      svc = SVC()
      svc.fit(X_train, y_train)

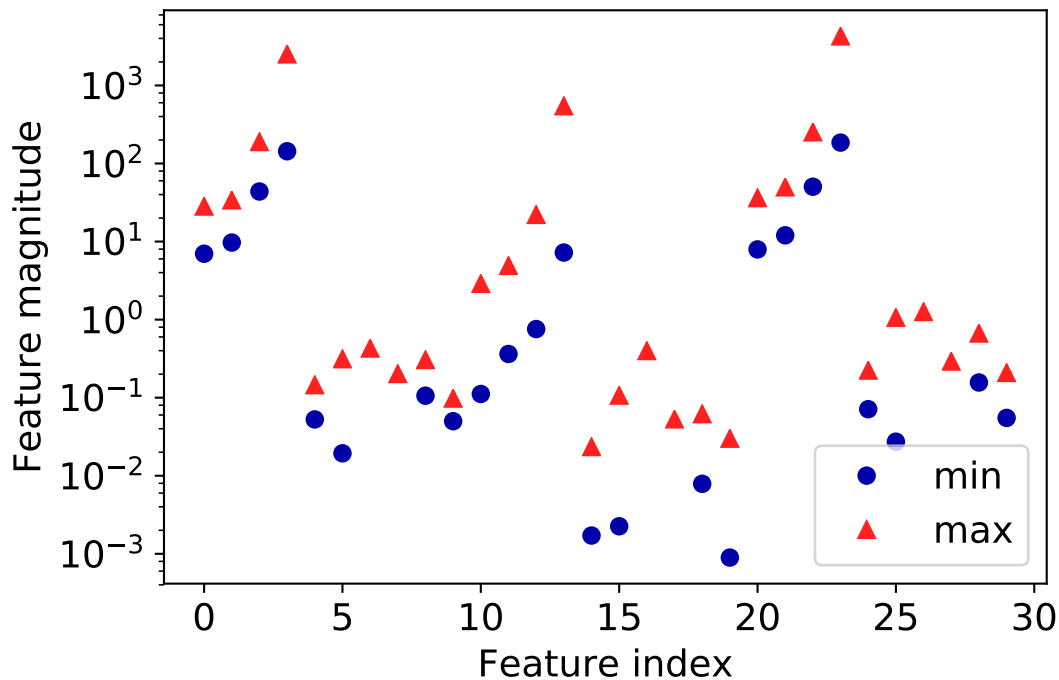
      print("Accuracy on training set: {:.2f}".format(svc.score(X_train, y_train)))
      print("Accuracy on test set: {:.2f}".format(svc.score(X_test, y_test)))
```

Accuracy on training set: 1.00

Accuracy on test set: 0.63

We can plot the scales of the features by plotting their min and max value

```
[41]: plt.plot(X_train.min(axis=0), 'o', label="min")
      plt.plot(X_train.max(axis=0), '^', label="max")
      plt.legend(loc=4)
      plt.xlabel("Feature index")
      plt.ylabel("Feature magnitude")
      plt.yscale("log")
```



We can scale all features between 0 and 1 Note: the sklearn.preprocessing package supports many preprocessing techniques, including the 'MinMaxScaler'

```
[42]: # Compute the minimum value per feature on the training set
min_on_training = X_train.min(axis=0)
# Compute the range of each feature (max - min) on the training set
range_on_training = (X_train - min_on_training).max(axis=0)

# subtract the min, divide by range
# afterwards min=0 and max=1 for each feature
X_train_scaled = (X_train - min_on_training) / range_on_training
print("Minimum for each feature\n{}".format(X_train_scaled.min(axis=0)))
print("Maximum for each feature\n{}".format(X_train_scaled.max(axis=0)))
```

[illegible]

- We must now apply the SAME transformation on the test set
 - Don't rescale the test set separately
 - Don't apply rescaling before making train test splits
- sklearn offers `pipelines` which make this easier
 - Wrapper around series of operators

```
[43]: # use THE SAME transformation on the test set,
      # using min and range of the training set.
      X_test_scaled = (X_test - min_on_training) / range_on_training
```

Much better results, but they can still be tuned further

```
[44]: svc = SVC()
      svc.fit(X_train_scaled, y_train)

      print("Accuracy on training set: {:.3f}".format(
          svc.score(X_train_scaled, y_train)))
      print("Accuracy on test set: {:.3f}".format(svc.score(X_test_scaled, y_test)))
```

Accuracy on training set: 0.948

Accuracy on test set: 0.951

```
[45]: svc = SVC(C=1000)
      svc.fit(X_train_scaled, y_train)

      print("Accuracy on training set: {:.3f}".format(
          svc.score(X_train_scaled, y_train)))
      print("Accuracy on test set: {:.3f}".format(svc.score(X_test_scaled, y_test)))
```

Accuracy on training set: 0.988

Accuracy on test set: 0.972

Strengths, weaknesses and parameters

- SVMs allow complex decision boundaries, even with few features.
- Work well on both low- and high-dimensional data
- Don't scale very well to large datasets (>100000)
- Require careful preprocessing of the data and tuning of the parameters.
- SVM models are hard to inspect

Important parameters: * regularization parameter C * choice of the kernel and kernel-specific parameters * Typically strong correlation with C