Support Vector Machines and kernelization

Linear SVMs

Revisited

Linear models for Classification (recap)

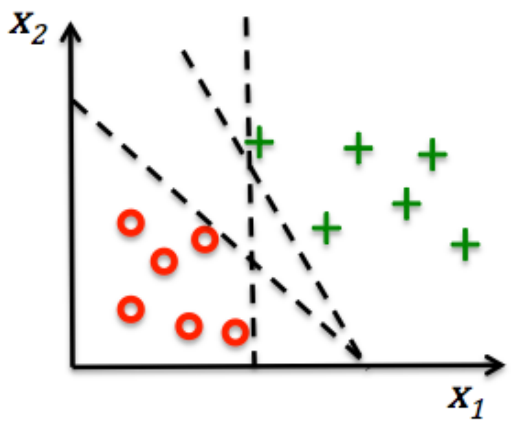
Aims to find a (hyper)plane that separates the examples of each class.
For binary classification (2 classes), we aim to fit the following function:

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

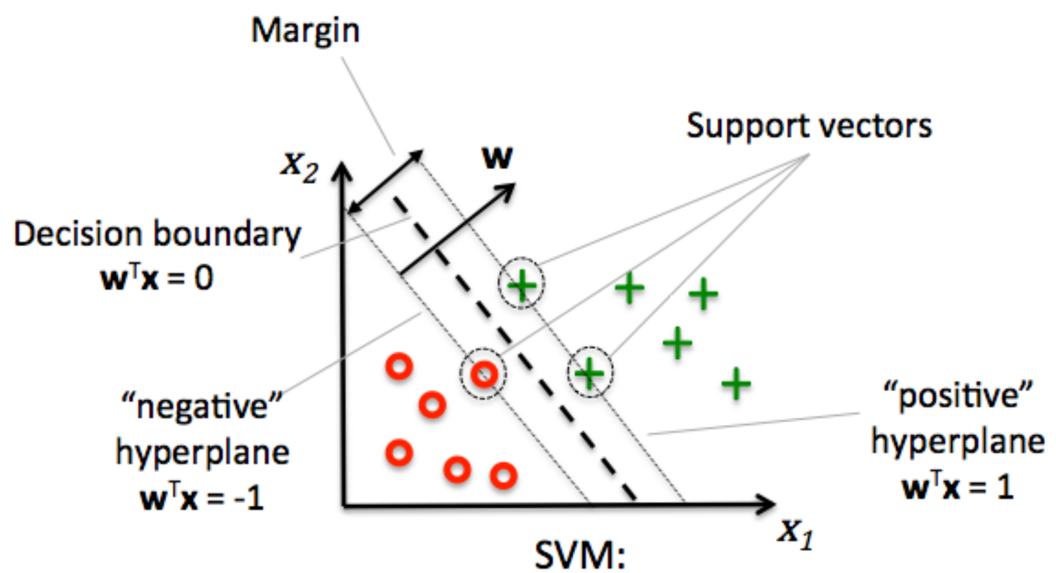
When $\hat{y} < 0$, predict class -1, otherwise predict class +1

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



Which hyperplane?



SVM:
Maximize the margin

Maximum margin

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

The *positive hyperplane* is defined as:

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

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

Likewise, the *negative hyperplane* is defined as:

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

Substracting them yields:

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

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

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

Yielding

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

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:

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

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

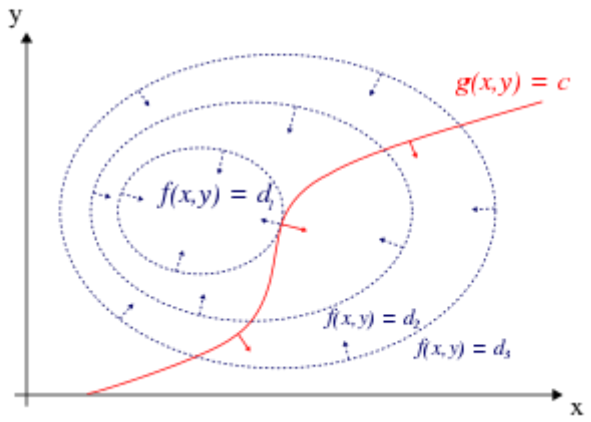
$$y^{(i)}(b + \mathbf{w^T x^{(i)}}) \geq 1 \quad \forall i$$

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*.

Geometric interpretation

- Assume 2 coefficients w_1 and w_2 (x and y in the image)
- Quadratic objective function $f = \frac{||w||^2}{2}$
- Constraint $y^{(i)}(b + \mathbf{w}^{\mathbf{T}}\mathbf{x}^{\mathbf{(i)}}) > 1 \quad \forall i$
  - $g(w_1, w_2) = 1$ in the image
- Find the point (w_1, w_2) that satifies g but maximizes f



Primal and Dual formulations

The Primal formulation of the Lagrangian objective function is:

$$minL_P = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{l} a_i y_i(\mathbf{x_i} * \mathbf{w} + b) + \sum_{i=1}^{l} a_i$$

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. We find the optimal set of a's first, then the w's can be easily computed.

It has a Dual formulation as follows:

$$minL_D(a_i) = \sum_{i=1}^{l} a_i - \frac{1}{2} \sum_{i,j=1}^{l} a_i a_j y_i y_j (\mathbf{x_i} . \mathbf{x_j})$$

so that

$$a_i \geq 0$$

$$\sum_{i=1}^{l} a_i y_i = 0$$

See 'Elements of Statistical Learning' for the complete derivation.

These are 2 very different optimization problems:

- In the primal, we optimize *p* variables (number of features)
- In the dual, we optimize *n* variables (number of instances)

Why are we considering this?

- In some problems, we have more features than data points
- 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.

In `sklearn`, the `LinearSVC` allows you to choose between the primal and the dual, while `SVC` always uses the dual

Making predictions

- 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
- Knowing the dual coefficients a_i (of which l are non-zero) 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 $\mathbf{w} * \mathbf{u} + b$

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.
It classifies points by looking at the sign of:
$f(x) = \sum_{i=1}^{k} a_i y_i dist(x_i, u)$

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 (a.k.a. the *kernel*) can be different

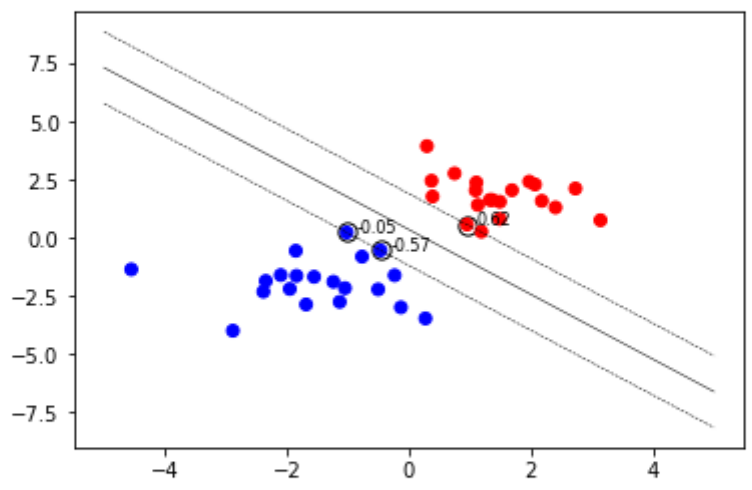
SVMs in scikit-learn

- 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 `weigths` of the support vectors
  - `coef_`: only for linear SVMs, the feature weights w

```python
clf = svm.SVC(kernel='linear')
clf.fit(X, Y)
print("Support vectors:", clf.support_vectors_[:])
print("Coefficients:", clf.dual_coef_[:])
```

```
Support vectors:
[[-1.021  0.241]
 [-0.467 -0.531]
 [ 0.951  0.58 ]]
Coefficients:
[[-0.048 -0.569  0.617]]
```

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



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 violatings of the margin constraint by introducing *slack variables* $\xi^{(i)}$

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

The new objective (to be minimized) becomes:

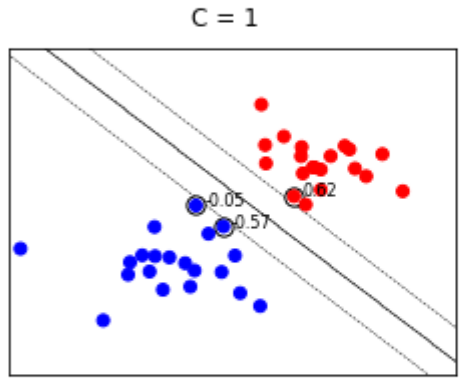
$$\frac{||w||^2}{2} + C(\sum_i \xi^{(i)})$$

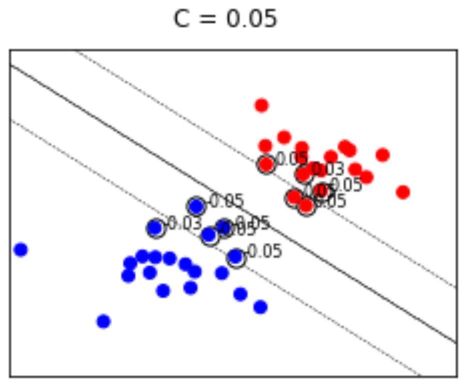
- C is a penalty for misclassification
  - Large C: large error penalties
  - Small C: less strict about violations (more regularization)
- This is known as the *soft margin* SVM (or *large margin* SVM)
  - Some support vectors are exactly on the margin hyperplane, with margin = 1
  - Others are margin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

C and regularization

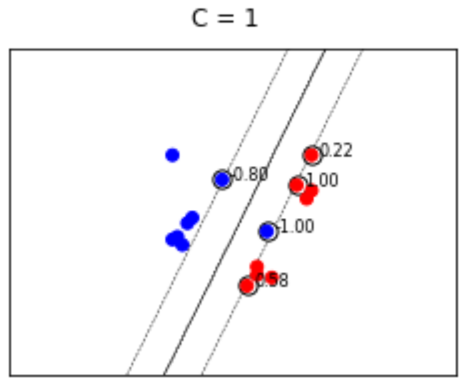
- Hence, we can use C to control the size of the margin and tune the bias-variance trade-off
  - Small C: Increases bias, reduces variance, more underfitting
  - Large C: Reduces bias, increases variance, more overfitting
- The penalty term $C(\sum_i \xi^{(i)})$ acts as an L1 regularizer on the dual 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'`)

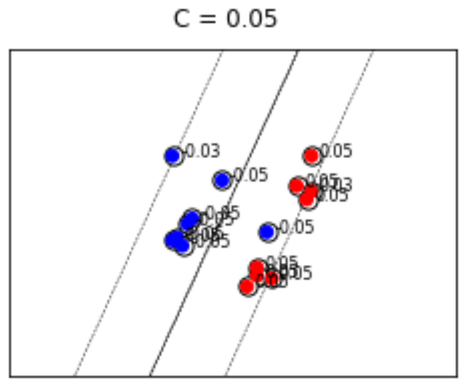
Effect on linearly separable data





Effect on non-linearly separable data





Hinge loss vs zero-one loss

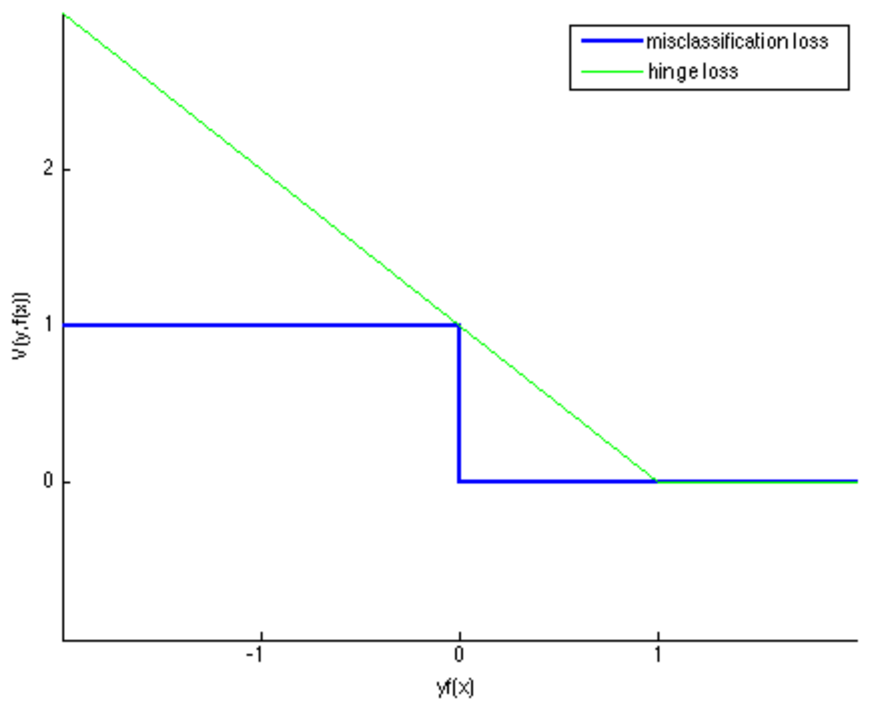
We are trying to:

- Maximize the margin
- Minimize the sum of margin violations

Why not maximize the margin and minimize the number of misclassifications (zero-one loss)

- Turns out that the corresponding objective function is not convex, NP-hard

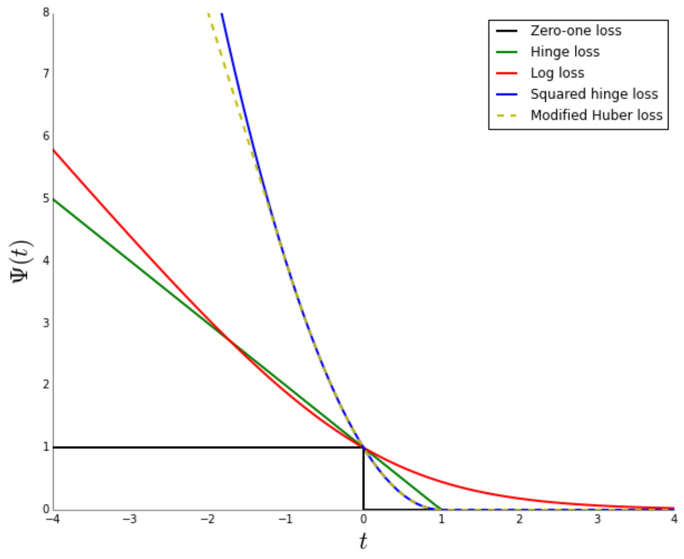
The best convex relation is hinge loss: $L(\gamma) = \max\{0, 1 - \gamma\}$
It measures the margin violation ξ_i.



Other loss functions

It is possible to use generalize SVMs by training them with other loss functions and gradient descent as the optimizer

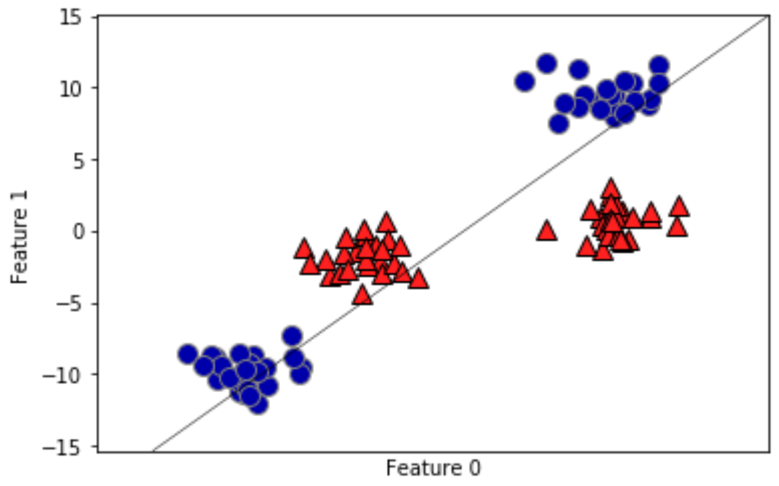
See the `SGDCLassifier` (`SGDCLassifier(loss='hinge')` will act like an SVM)



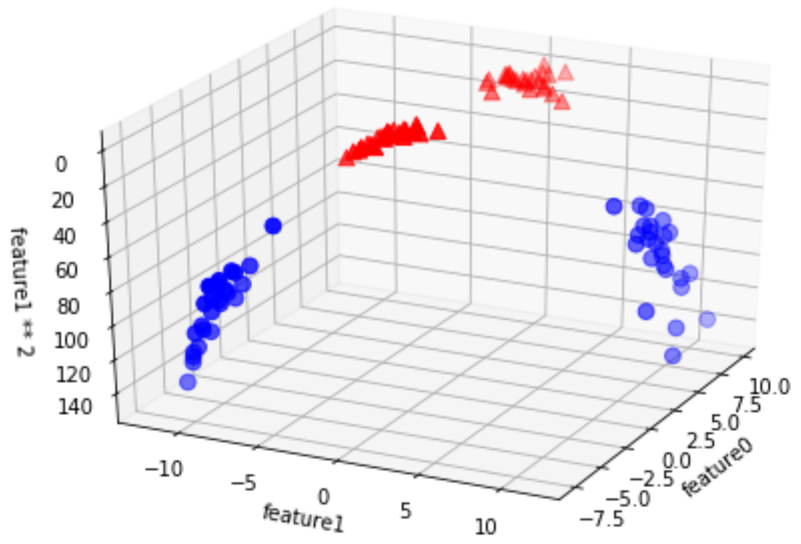
Kernelized Support Vector Machines

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

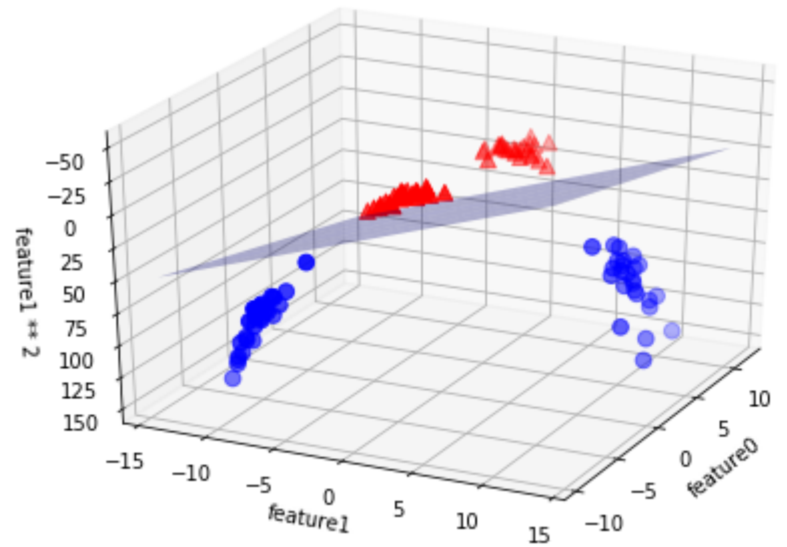
Our linear model doesn't fit the data well



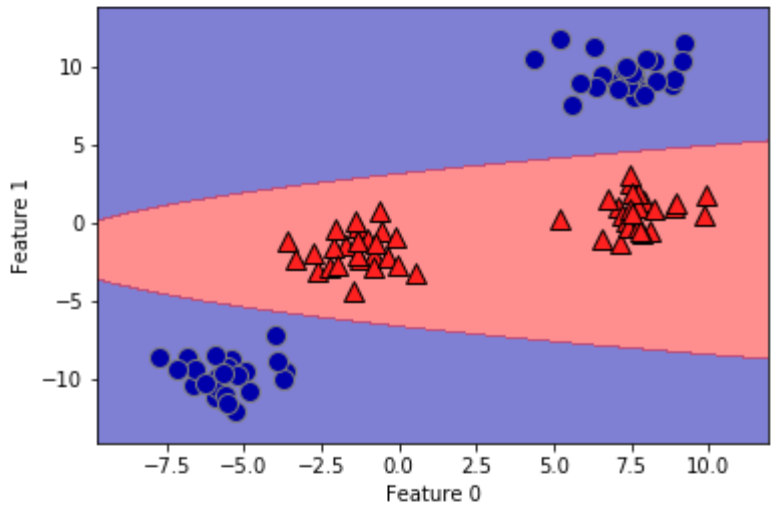
We can add a new feature by taking the squares of feature1 values



Now we can fit a linear model



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



Kernels

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

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

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 $x_1, \ldots, x_n$, the kernel Gram matrix is positive semi-definite

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

What is this good for?

Mercer's Theorem states that

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

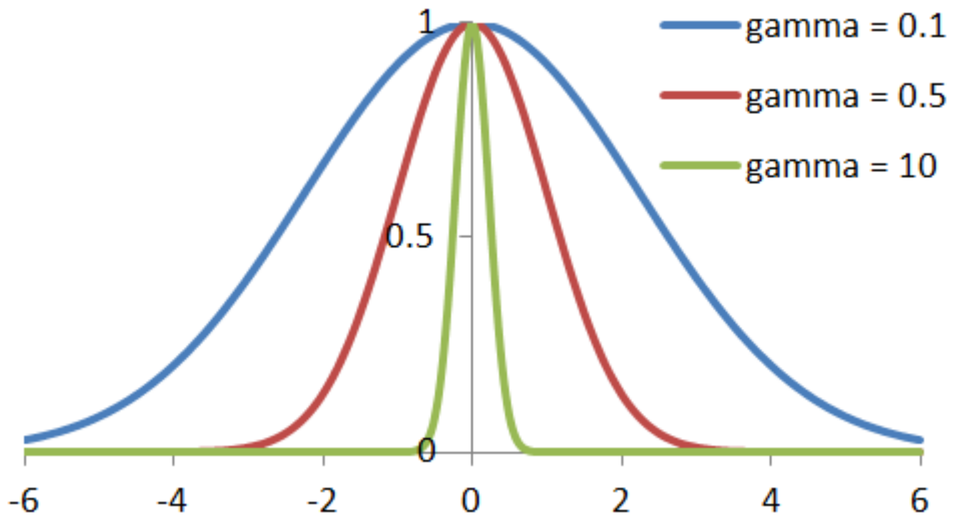
Hence, a kernel can be thought of as a 'shortcut' computation for the 2-step procedure feature map + inner product

- we don't need to construct a space of all polynomials of all features, we can define a kernel that returns the similarity between any two points by simply computing an 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 base fucntion' (or **Gaussian**) kernel is defined as: $k(x_1, x_2) = exp(-\gamma||x_1 - x_2||^2)$, for $\gamma \geq 0$



The Kernel Trick

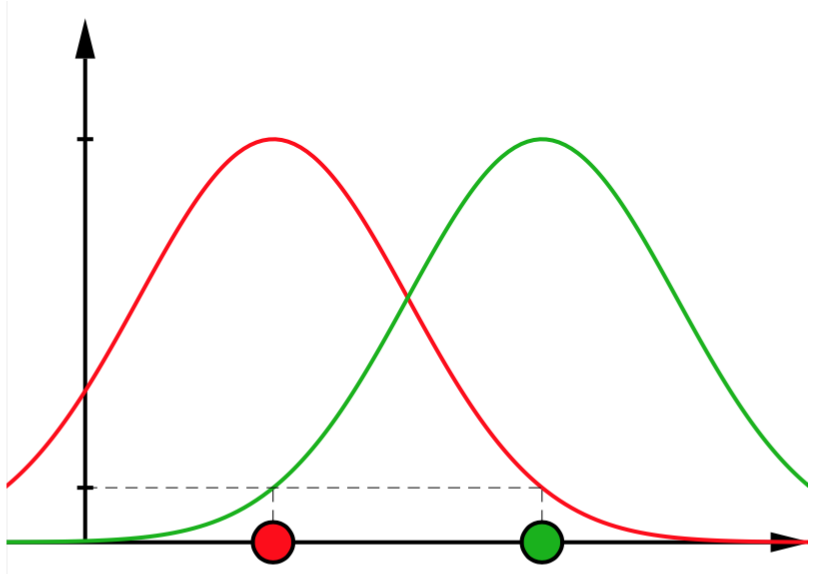
- Adding nonlinear features can make linear models much more powerful
- Often we don’t know which features to add, and adding many features might make computation very expensive
- Mathematical trick (*kernel trick*) allows us to directly compute distances (scalar products) in the high dimensional space
    - 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

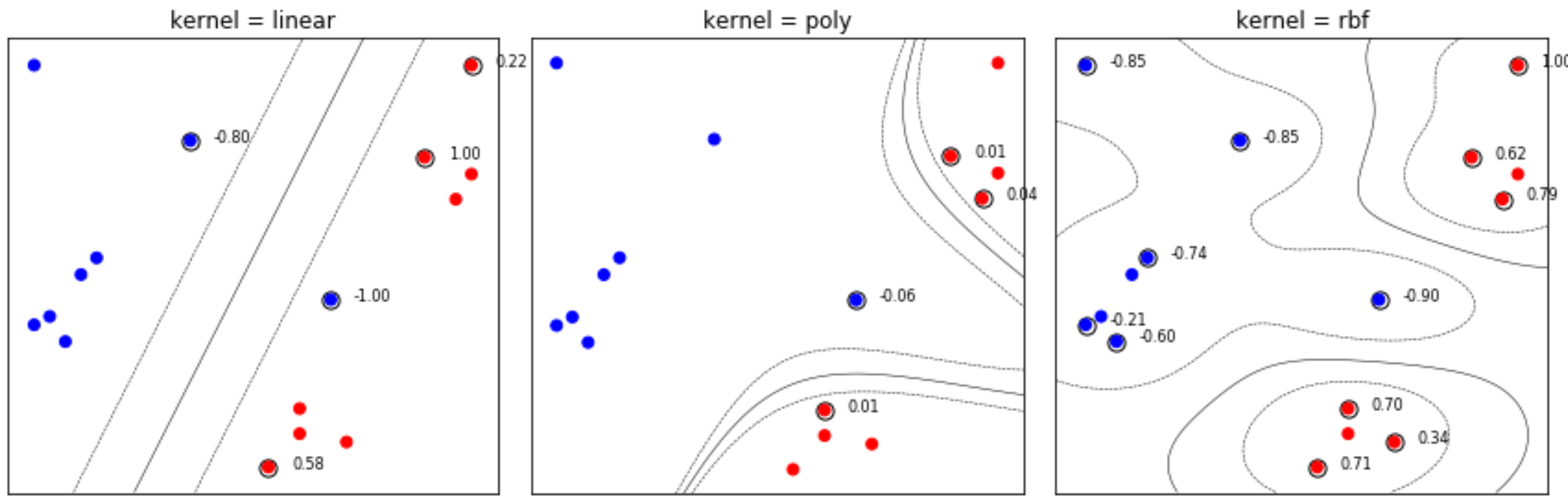
- There exist many feature maps (and hence Hilbert spaces) for the same kernel, but they are all equivalent
- The Reproducing Kernel Hilbert Space (RKHS) has feature map $\phi : X \to C(X); x \to k(x, \cdot)$ Where C is the space of continuous functions $X \to \mathbb{R}$
- Thus, an input $x \in X$ is mapped to the basis function $\phi(x) = k(x, \cdot)$
  - For every point, the mappings are continuous functions $k(x, \cdot)$
- Kernel computes ⟨k(x1,·),k(x2,·)⟩=k(x1,x2)

Example: Gaussian kernel, 2 points (green and red)

- Each point generates a function, the inner product is where they intersect
- The closer the points are, the more similar they are



Comparing the decision boundaries:



Local vs Global kernels

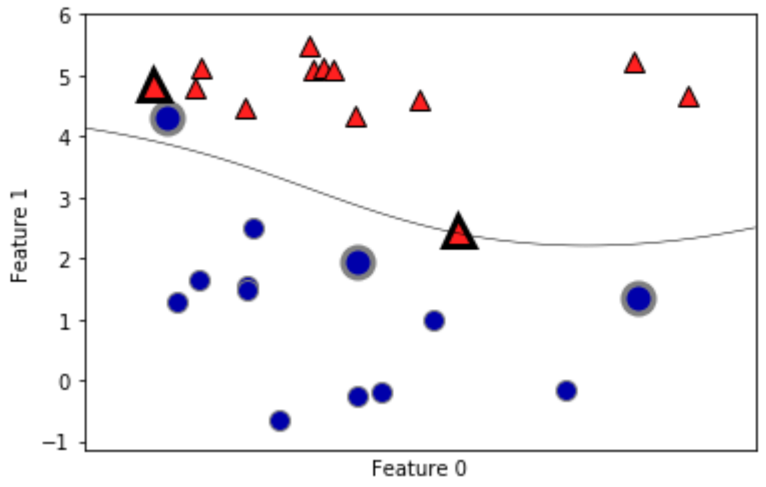
- With a linear or polynomial kernel, one support vector can affect the whole model space
  - These are called *global kernels*
- The RBF kernel only affects the region around the support vector (depending on how wide it is)
  - This a called a *local* kernel
  - Can capture local abnormalities that a global kernel can't
  - Also overfits easily if the kernels are very narrow

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

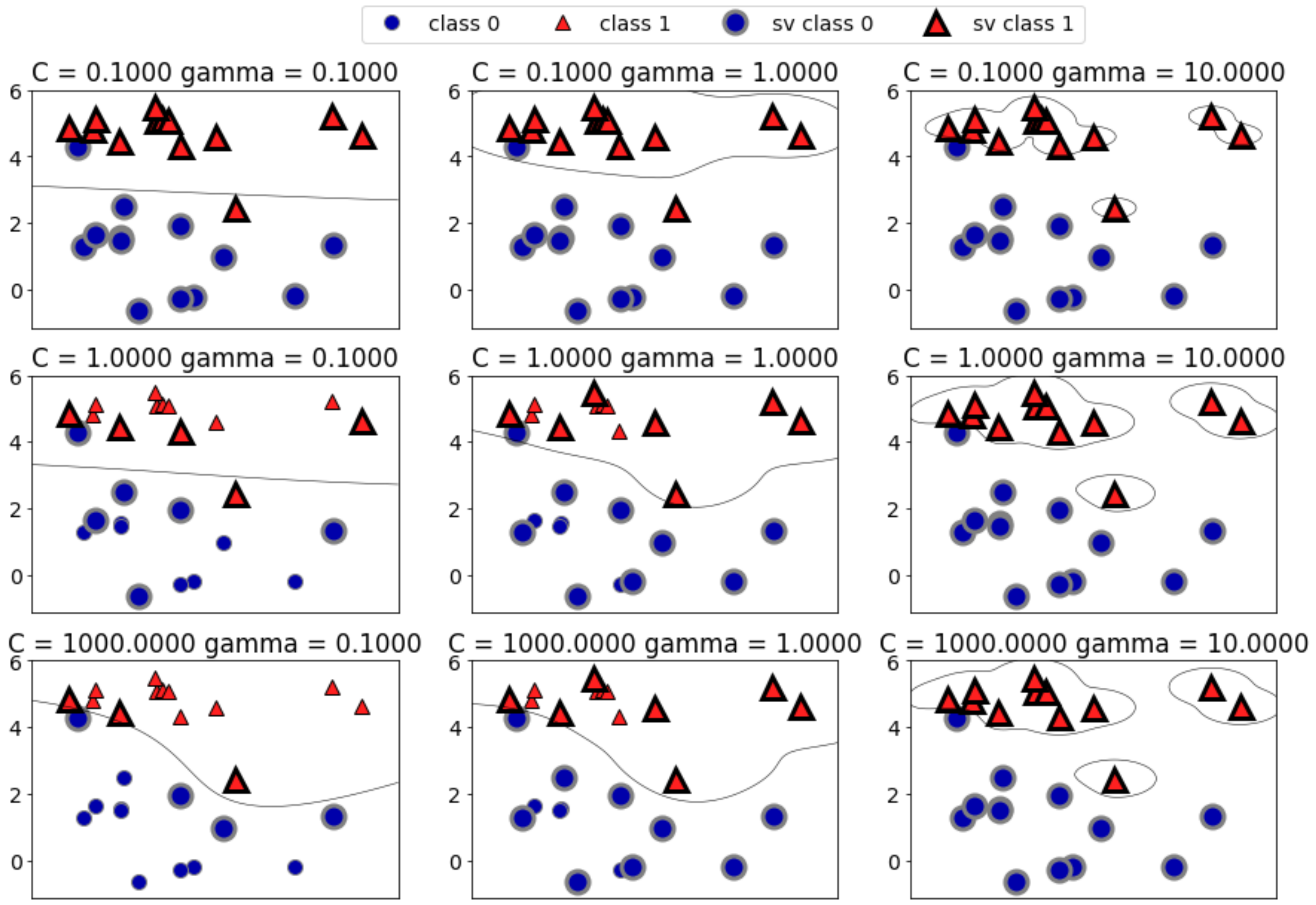
Given the support vectors, their weigths, and the kernel, we can plot the decision boundary



Tuning SVM parameters

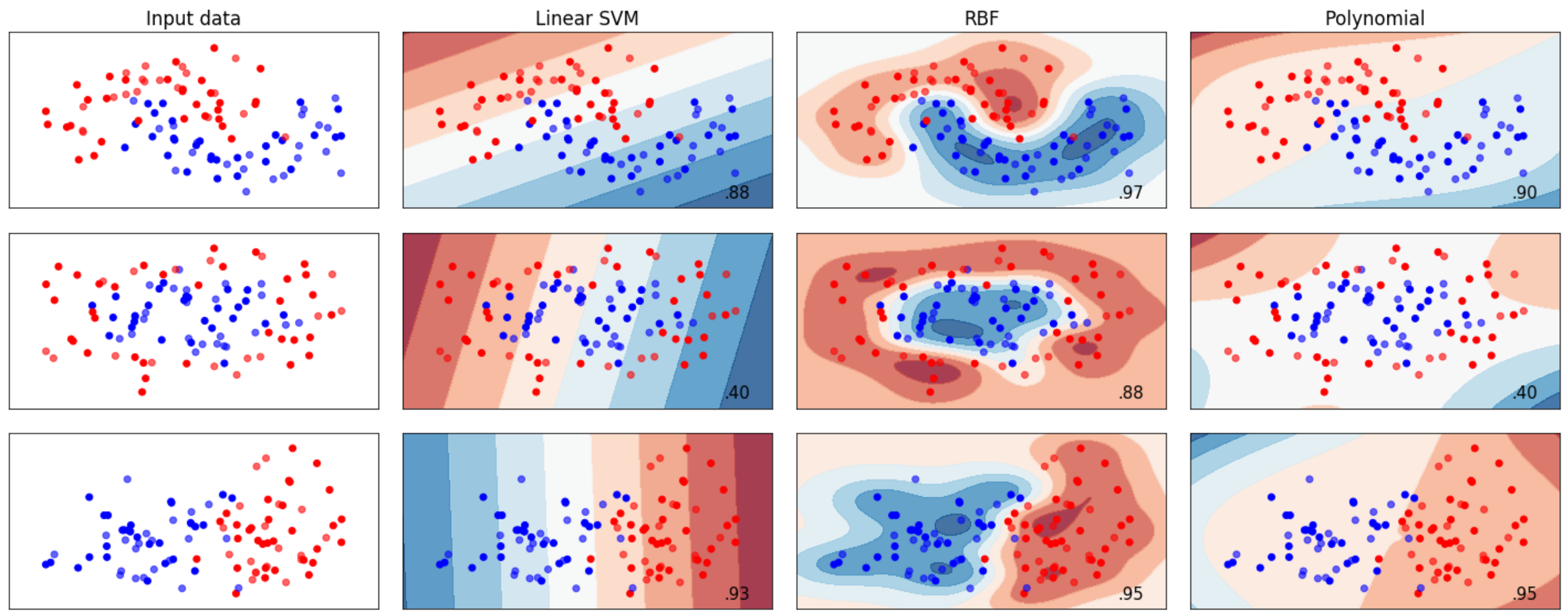
Several important parameters:

- gamma ((inverse) kernel width): high values means that points are further apart
    - High values mean narrow Gaussians, i.e. the influence of one point is very small
        - You need many support vectors
    - Leads to complex decision boundaries, overfitting
- C (our linear regularizer): 'cost' of misclassifying training examples
    - High C: force SVM to classify more examples correctly
        - Requires more support vectors, thus complex decision boundaries
- For polynomial kernels, the *degree* (exponent) defines the complexity of the models


class 0
class 1
sv class 0
sv class 1
C = 0.1000 gamma = 0.1000
C = 0.1000 gamma = 1.0000
C = 0.1000 gamma = 10.0000
C = 1.0000 gamma = 0.1000
C = 1.0000 gamma = 1.0000
C = 1.0000 gamma = 10.0000
C = 1000.0000 gamma = 0.1000
C = 1000.0000 gamma = 1.0000
C = 1000.0000 gamma = 10.0000
6
4
2
0

- 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 vectores, almost linear decision boundary
- High C (bottom): Stronger influence, decision boundary bends to every support vector

Kernel overview



Preprocessing Data for SVMs

- SVMs are very sensitive to hyperparameter settings
- They expect all features to be approximately on the same scale
- Data point similarity (e.g. RBF kernel) is computed the same way in all dimensions
- If some dimension is scaled differently, it will have a much larger/smaller impact

```
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)
```

```
Accuracy on training set: 1.00
Accuracy on test set: 0.63
```

- We can scale all features between 0 and 1
  - E.g. use `sklearn.preprocessing.MinMaxScaler`
- Remember, we must now apply the SAME transformation on the test set
  - 'Learn' the minima/maxima of training data
  - Apply them on the training and test splits separately
- sklearn offers `pipelines` which make this easier
  - Wrapper around series of operators

```
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
svc = SVC()
svc.fit(X_train_scaled, y_train)
```

```
Accuracy on training set: 0.948
Accuracy on test set: 0.951
```

Much better results, but they can still be tuned further

```
svc = SVC(C=1000)
svc.fit(X_train_scaled, y_train)
```

```
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 string correlation with C