Support Vector Machines and kernelization

Feature Maps

- Remember linear models?

$$\hat{y} = \mathbf{wx} = \sum_{i=0}^{p} w_i \cdot x_i = w_0 \cdot x_0 + w_1 \cdot x_1 + \ldots + w_p \cdot x_p$$

- When we cannot fit the data well with linear models, we can learn more complex models by simply adding more dimensions

- Feature map (or *basis expansion*) ϕ: $X \rightarrow \mathbb{R}^d$

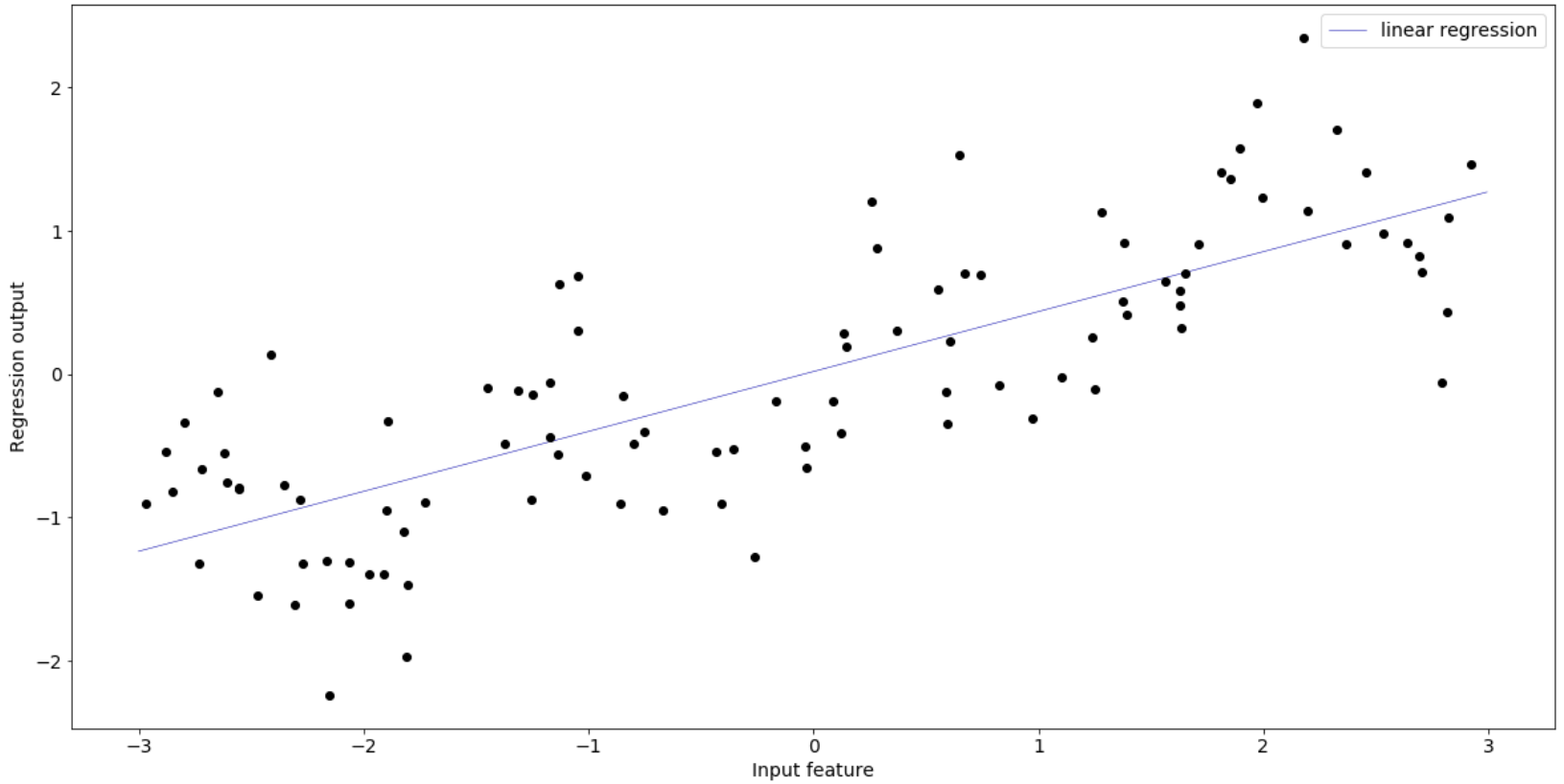
$$y = \mathbf{w}^T\mathbf{x} \rightarrow y = \mathbf{w}^T\phi(\mathbf{x})$$

- You still may need MANY dimensions to fit the data

    - Memory and computational cost
    - More likely overfitting

Example: Ridge regression

```
Coefficients: [0.418]
```



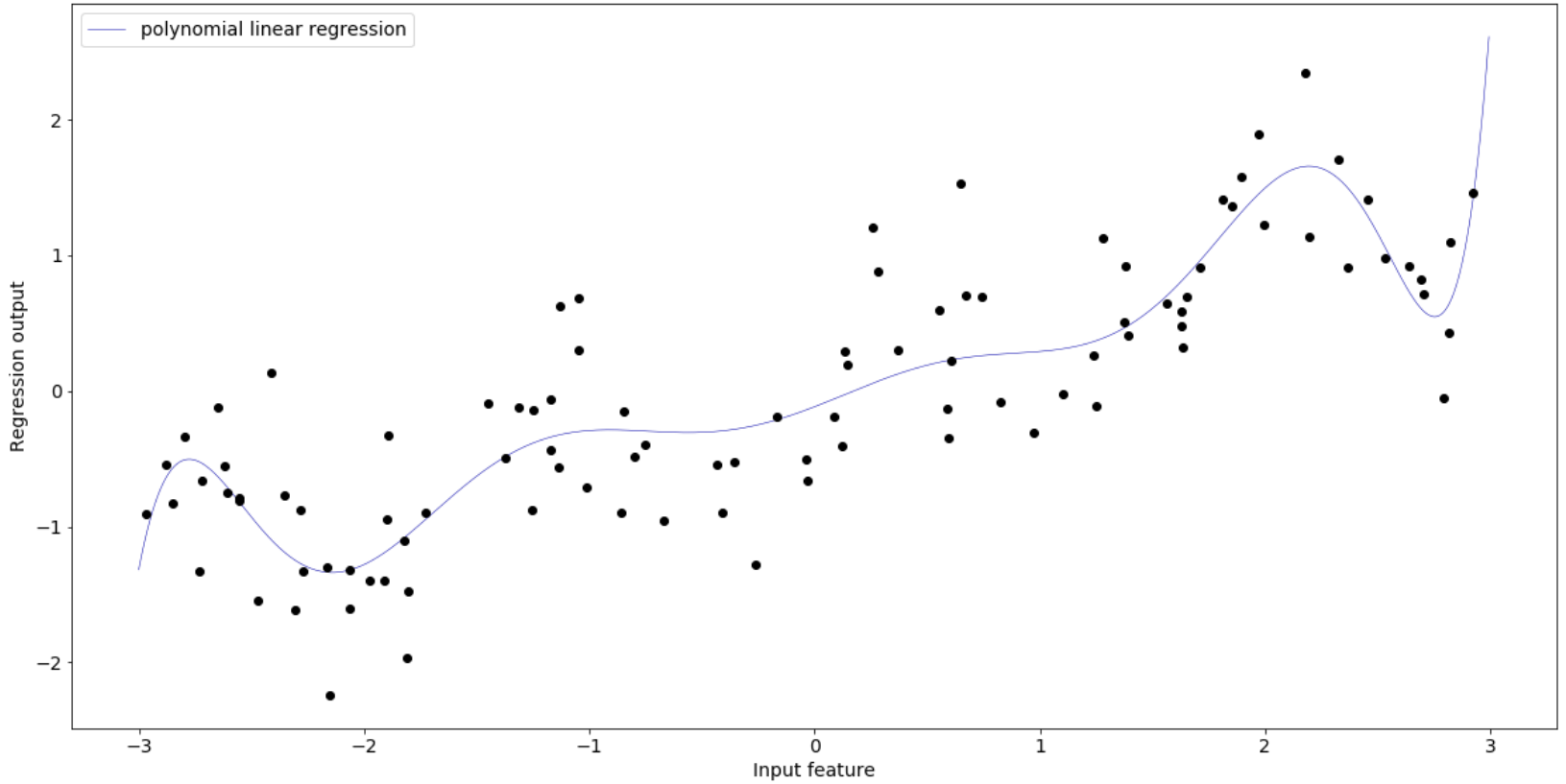
- Add all polynomials x^d up to degree D. How large should D be?
- We can also compute all polynomials and all interactions between features (e.g. $x \cdot x^2$). This leads to D^2 features.

Out[31]:

	x0	x0^2	x0^3	x0^4	x0^5	x0^6	x0^7	x0^8	x0^9	x0^10
0	-0.75	0.57	-0.43	0.32	-0.24	0.18	-0.14	0.1	-0.078	0.058
1	2.7	7.3	20	53	1.4e+02	3.9e+02	1.1e+03	2.9e+03	7.7e+03	2.1e+04
2	1.4	1.9	2.7	3.8	5.2	7.3	10	14	20	27
3	0.59	0.35	0.21	0.12	0.073	0.043	0.025	0.015	0.0089	0.0053
4	-2.1	4.3	-8.8	18	-37	77	-1.6e+02	3.3e+02	-6.8e+02	1.4e+03

Fit Ridge again:

```
Coefficients: [ 0.643  0.297 -0.69  -0.264  0.41   0.096 -0.076 -0.014  0.
004  0.001]
```



- Ridge has a closed-form solution which we can compute with linear algebra:

$$w^* = (X^T X + \lambda I)^{-1} X^T Y$$

- Since X has n rows (examples), and d columns (features), $X^T X$ has dimensionality dxd
- Hence Ridge is quadratic in the number of features, $\mathcal{O}(d^2 n)$
- After the feature map Φ, we get

$$w^* = (\Phi(X)^T \Phi(X) + \lambda I)^{-1} \Phi(X)^T Y$$

- Since Φ increases d a *lot*, $\Phi(X)^T \Phi(X)$ becomes *huge*
- To be continued...

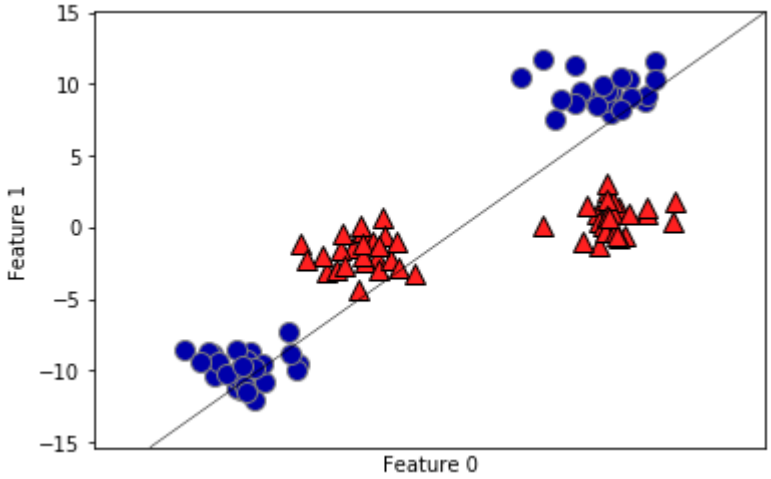
Linear models for Classification (recap)

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:

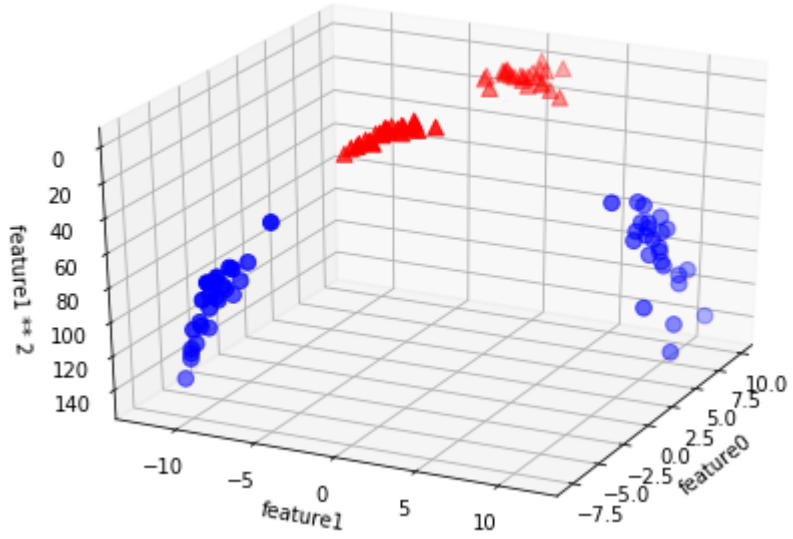
$$\hat{y} = w_0 * x_0 + w_1 * x_1 + \ldots + w_p * x_p + b > 0$$

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

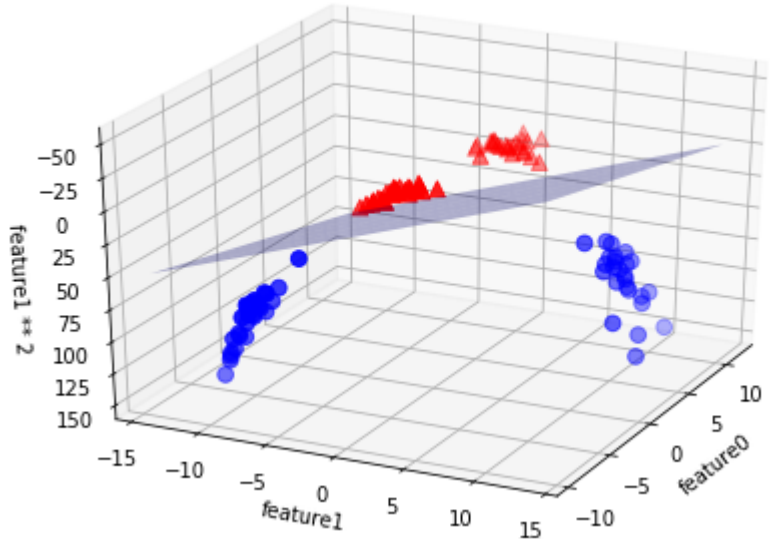
Again, we can add dimensions when our linear model doesn't fit the data well



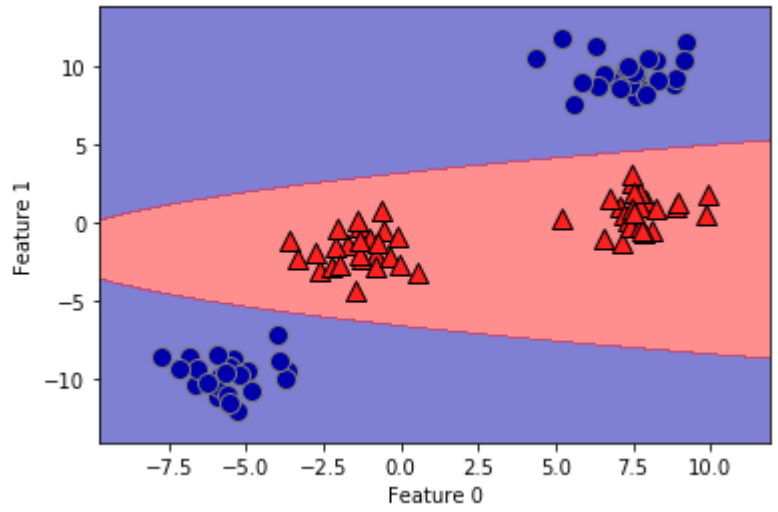
We can add a new feature by taking the squares of feature1 values



Now we can fit a linear model

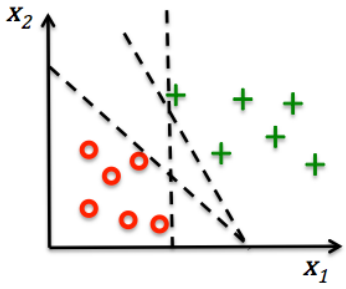


As a function of the original features, the linear SVM model is not actually linear anymore, but more of an ellipse

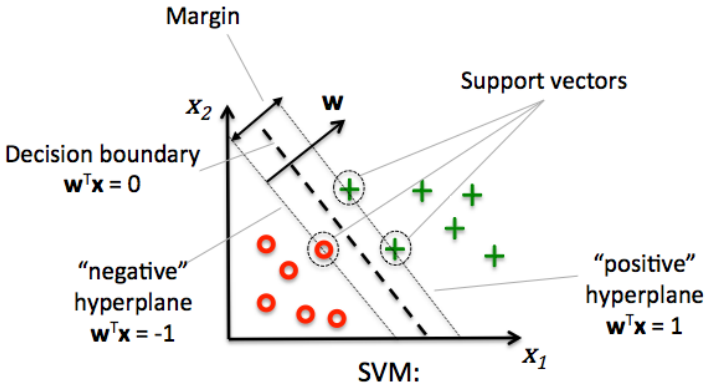


Support vector machines (SVMs)

- 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:

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

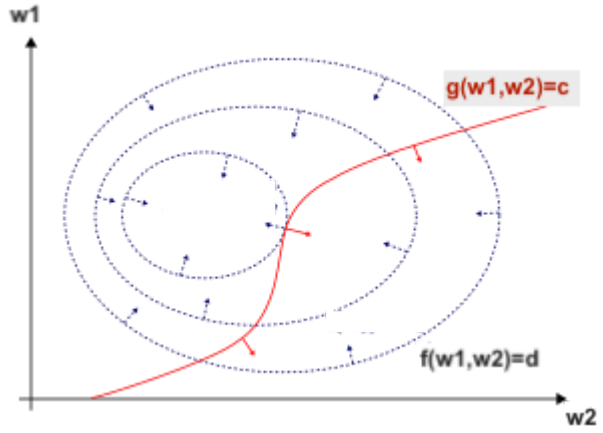
Maximizing $\frac{2}{||w||}$ can be done by minimizing $\frac{||w||^2}{2}$

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

- It gets more complicated when the data is not linearly separable

Geometric interpretation

- Assume 2 coefficients w_1 and w_2
- Quadratic objective function $f = \frac{||w||^2}{2}$
- Constraint $y^{(i)}(b + \mathbf{w^T x^{(i)}}) > 1 \quad \forall i$
  - $g(w_1, w_2) = 1$ 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 to minimize:

$$\mathcal{L}_{Primal} = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{n} a_i\, y_i(\mathbf{x_i} * \mathbf{w} + b) + \sum_{i=1}^{n} a_i$$

so that

$$a_i \geq 0$$

$$\mathbf{w} = \sum_{i=1}^{n} a_i\, y_i\, \mathbf{x_i}$$

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

with n the number of training examples and a the *dual variables*, which act like weights for each training example. We find the optimal set of a's first, then the w's can be easily computed.

Hence, instead of learning the weights, we learn which examples are important and the model follows from that.

It has a Dual formulation is to minimize (See 'Elements of Statistical Learning'):

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

so that

$$a_i \geq 0$$

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

- Observe the term $\mathbf{x_i} . \mathbf{x_j}$. When the inputs only appear in inner products, we call the method *kernelized*

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

- a_i will be *0* if the point lies outside the margin, on the right side of the decision boundary
- 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 is an *inner product of the inputs*

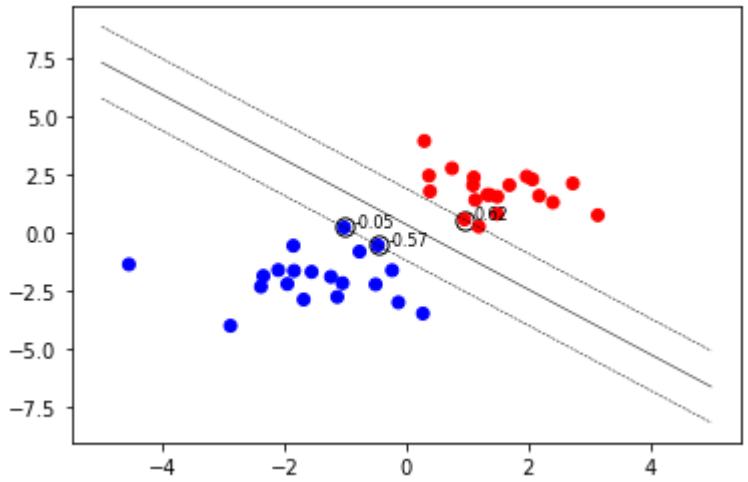
SVMs in scikit-learn

- 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

```
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 a *slack variable* $\xi^{(i)}$ for every data point

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

The new objective (to be minimized) becomes:

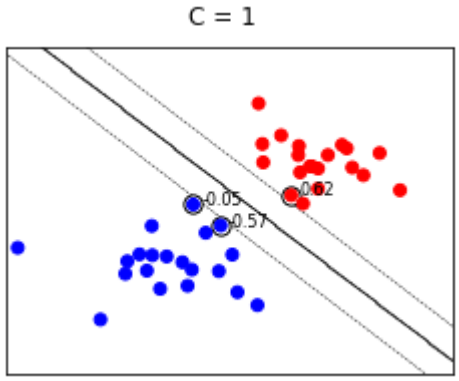
$$\frac{||w||^2}{2} + C(\sum_i \xi^{(i)})$$

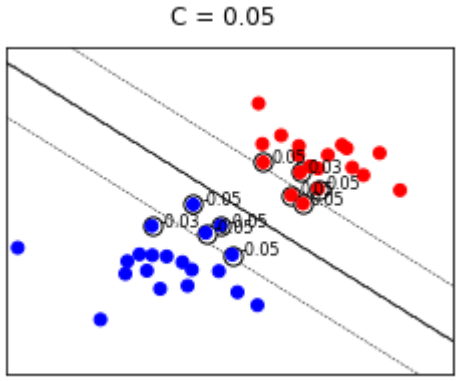
- C is a penalty for misclassification
    - Large C: large error penalties
    - Small C: less strict about violations (more regularization)
- This is known as the *soft margin* SVM (or *large margin* SVM)
    - Some support vectors are exactly on the margin hyperplane, with margin = 1
    - Others 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

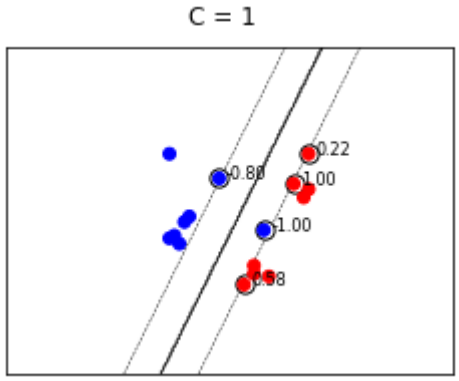
- Hence, we can use C to control the size of the margin and tune the bias-variance trade-off
    - Small C: Increases bias, reduces variance, more underfitting
    - Large C: Reduces bias, increases variance, more overfitting
- The penalty term $C(\sum_i \xi^{(i)})$ acts as an L1 regularizer on the dual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

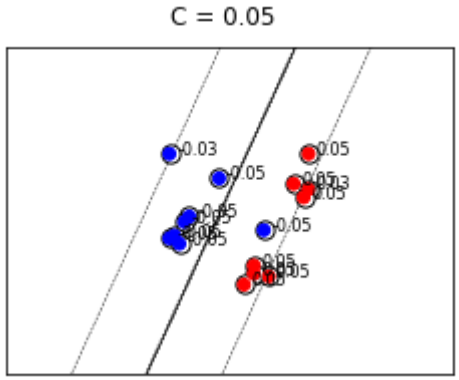
Effect on linearly separable data





Effect on non-linearly separable data





Hinge loss vs zero-one loss

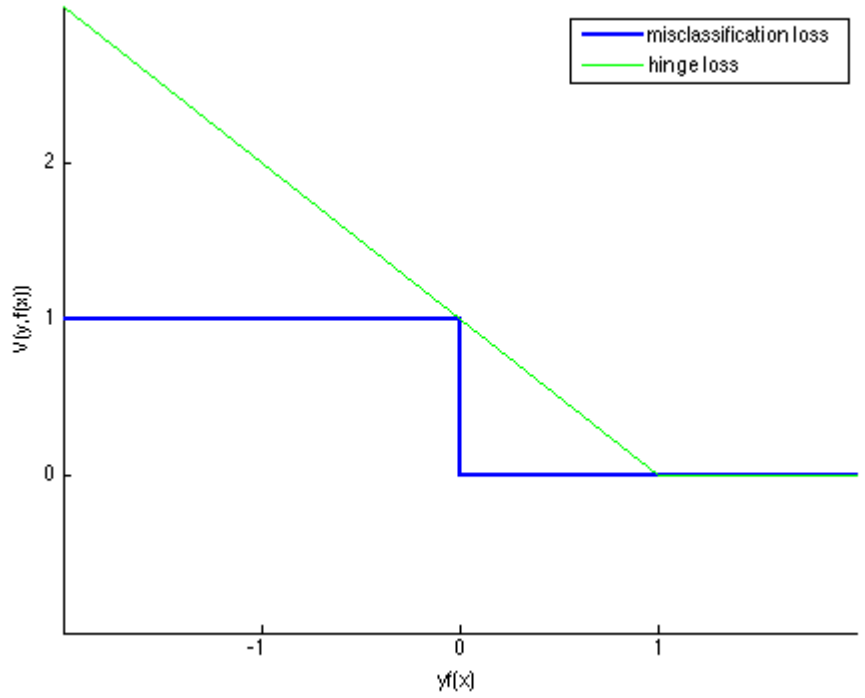
We are trying to:

- 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

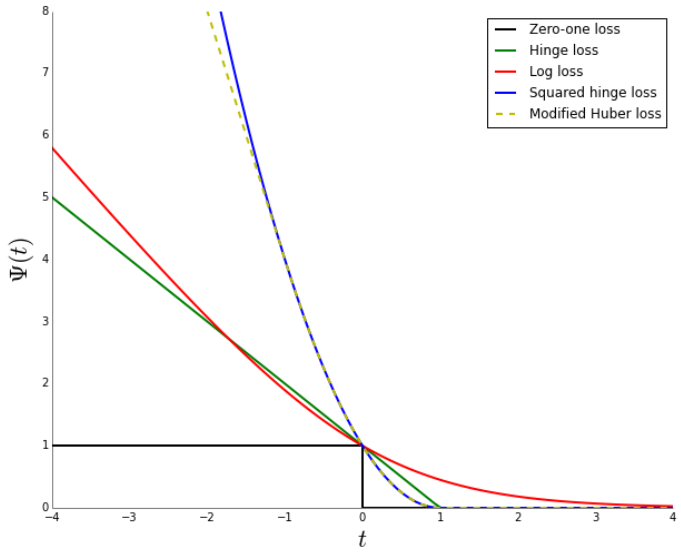
The best convex relation is hinge loss: $L(\gamma) = \max\{0, 1 - \gamma\}$
It measures the margin violation ξ_i. It will be in [0,1] for points inside the margin, and even larger if on the wrong side of the boundary



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)



Kernelization

- A method is *kernelized* if inputs only appear inside inner products $x_i \cdot x_j = \langle x_i, x_j \rangle$
- We said that SVMs can be *kernelized* through it's dual formulation:

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

- A *kernel function* corresponding to a transformation Φ is

$$k(x_i, x_j) = \langle \Phi(x_i), \Phi(x_j) \rangle$$

- $\Phi(x)$ can be used to generate many more features based on the original feature x
- Turns out, we can often evaluate $k(x_i, x_j)$ directly, *without* evaluating $\Phi(x_i), \Phi(x_j)$

Kernel trick

- Evaluating the kernel directly can be *much* cheaper.
- Example: a simple *quadratic* feature map for $x = (x1, \ldots, xd)$ has dimension $\mathcal{O}(d^2)$:

$$\Phi(x) = (x_1, \ldots, x_d, x_1^2, \ldots, x_d^2, \sqrt{2}x_1x_2, \ldots, \sqrt{2}x_{d-1}x_d)$$

- The corresponding quadratic kernel is:

$$k(x_i, x_j) = \langle \Phi(x_i), \Phi(x_j) \rangle = \langle x_i, x_j \rangle + \langle x_i, x_j \rangle^2$$

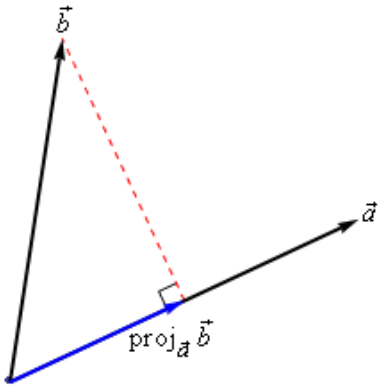
- We can skip the computation of $\Phi(x_i)$ and $\Phi(x_j)$ and compute $k(x_i, x_j)$ in $\mathcal{O}(d)$ instead of $\mathcal{O}(d^2)$!

Kernel functions

- It is useful to think of a kernel as a similarity score between 2 vectors (points)
    - Not mathematically equivalent
- There are many ways to design such a similarity score (also for text, graphs,...)
- Computationally *much* cheaper
- We can access very large (even infinite) feature spaces $\mathcal{H}$
- Thinking in terms of similarity is much more intuitive than thinking in high-dimensional feature spaces

Linear kernel

- Input space is same as output space: $X = \mathcal{H} = \mathbb{R}^d$
- Feature map $\Phi(x) = x$
- Kernel: $k(x_i, x_j) = x_i \cdot x_j = x_i^T x_j$
- Geometrically, we can view these as *projections* of x_j on a hyperplane defined by x_i
    - Nearby points will have nearby projections



Kernel matrix

For points of $x_1, \ldots, x_n \in X$ and an inner product $\langle \cdot, \cdot \rangle$, the kernel matrix (or Gram matrix) is defined as:

$$K = \begin{bmatrix} \langle x_1, x_1 \rangle & \ldots & \langle x_1, x_n \rangle \\ \vdots & \ddots & \vdots \\ \langle x_n, x_1 \rangle & \ldots & \langle x_n, x_n \rangle \end{bmatrix}$$

- For the Euclidean inner product $k(x_i, x_j) = x_i^T x_j$, we have $K = XX^T$
- Size is *nxn*, irrespective of number of dimensions d
- Once kernel matrix is computed, cost depends on number of data points only

Kernels

More generally, a (Mercer) kernel on a space X is a function

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

With the properties:

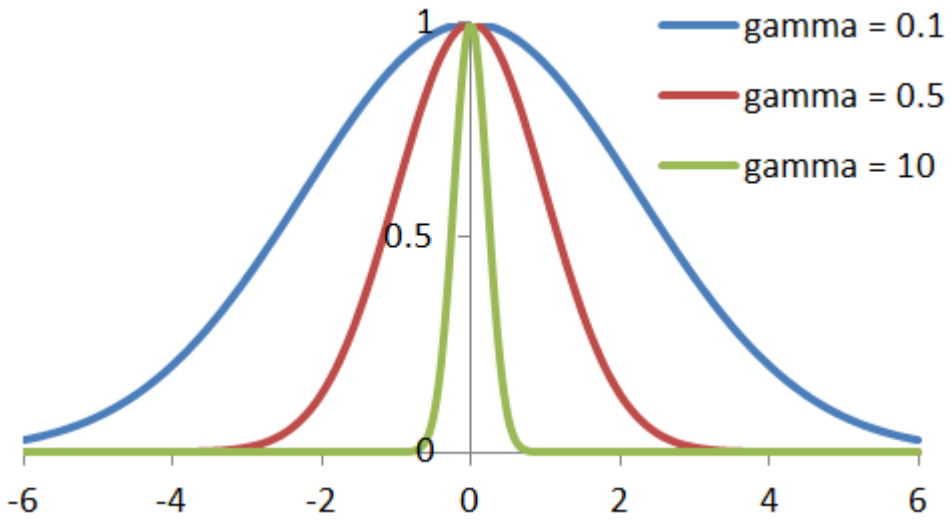
- Symmetry: $k(x_1, x_2) = k(x_2, x_1) \;\; \forall x_1, x_2 \in X$
- Positive definite: for each finite subset of data points $x_1, \ldots, x_n$, the kernel Gram matrix is positive semi-definite
    - Intuitively, $k(x_1, x_2) \geq 0$

This is also what we expect from similarity functions

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 $\gamma \geq 0$

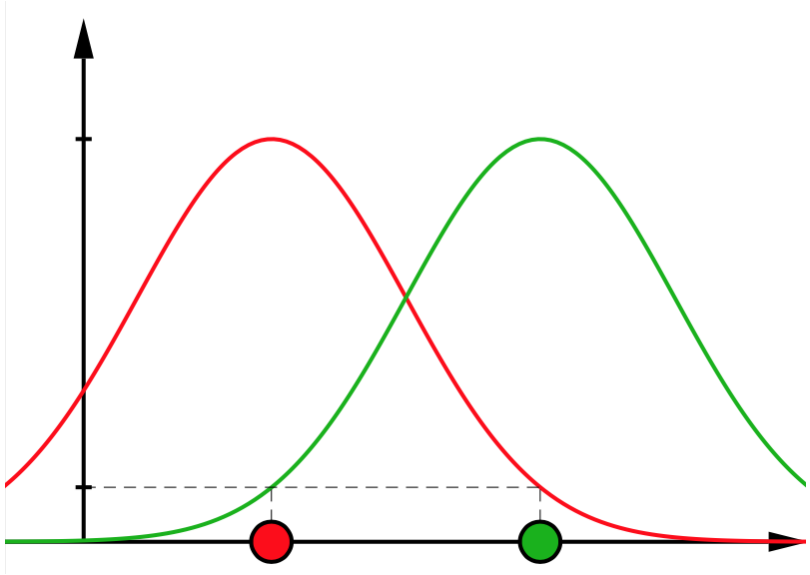


The Kernel Trick (summary)

- Explictly adding nonlinear features can make linear models much more powerful, but also much more expensive
- Given a kernelized ML algorithm, we can swap out the inner product for a new kernel function.
- Kernel functions allow us to directly compute distances (scalar products) in an implicit high dimensional space (you don't actually construct it)
- 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

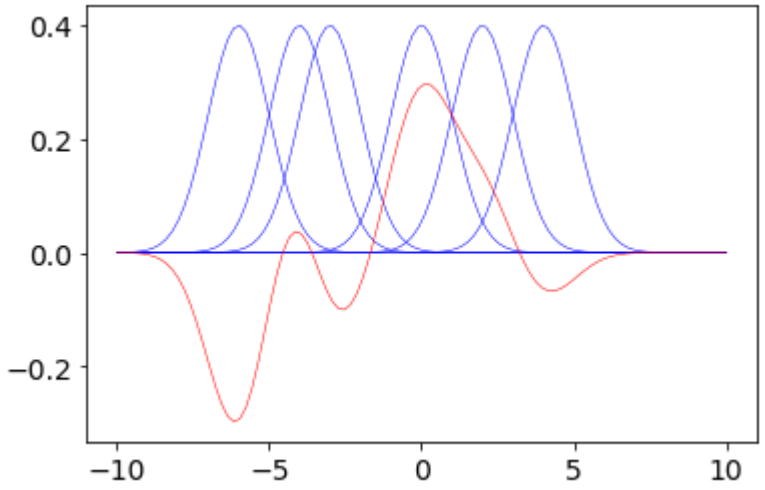
Gaussian kernel: intuition

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

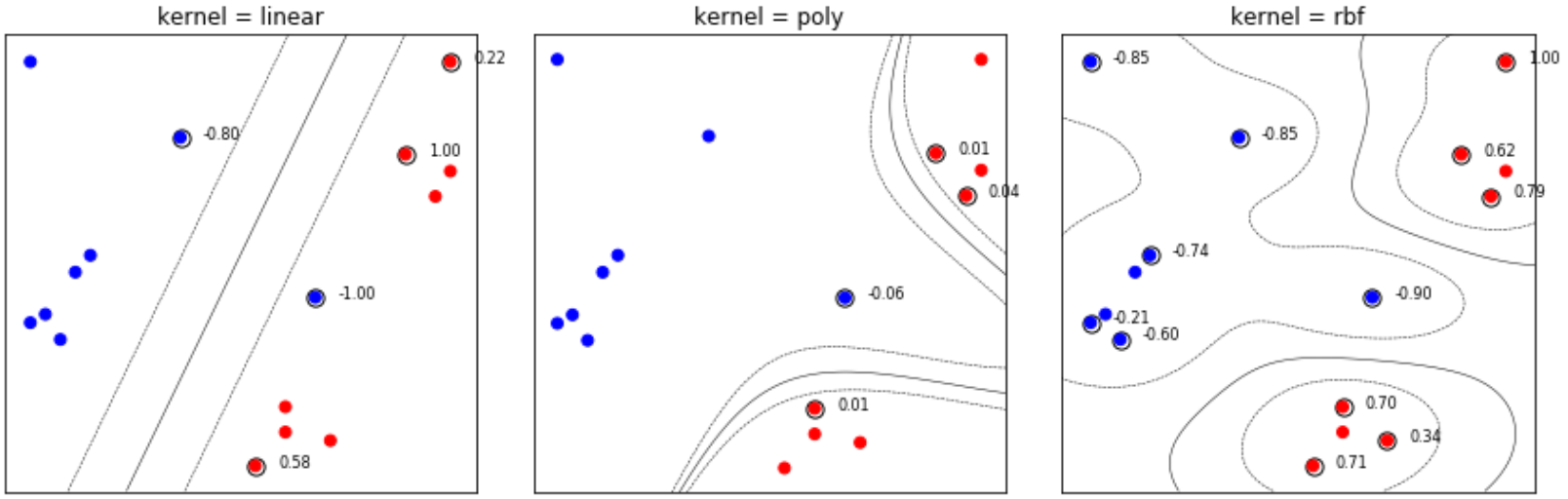


Example (for regression):

- We have 6 input points: [-6,-4,-3,0,2,4]
    - We fit a kernel over each (blue)
- We learn a coefficient for each: e.g. [-.8,.5,-0.5,.7,0.3,-0.2]
- Resulting preditions (red curve)



With SVMs: Comparing the decision boundaries:



Local vs Global kernels

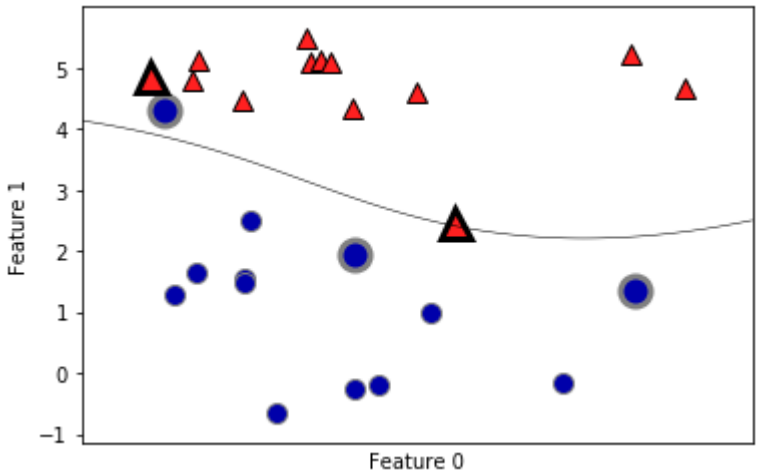
- With a linear or polynomial kernel, one support vector can affect the whole model space
    - These are called *global kernels*
- The RBF kernel only affects the region around the support vector (depending on how wide it is)
    - This a called a *local* kernel
    - Can capture local abnormalities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

Kernel parameters

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

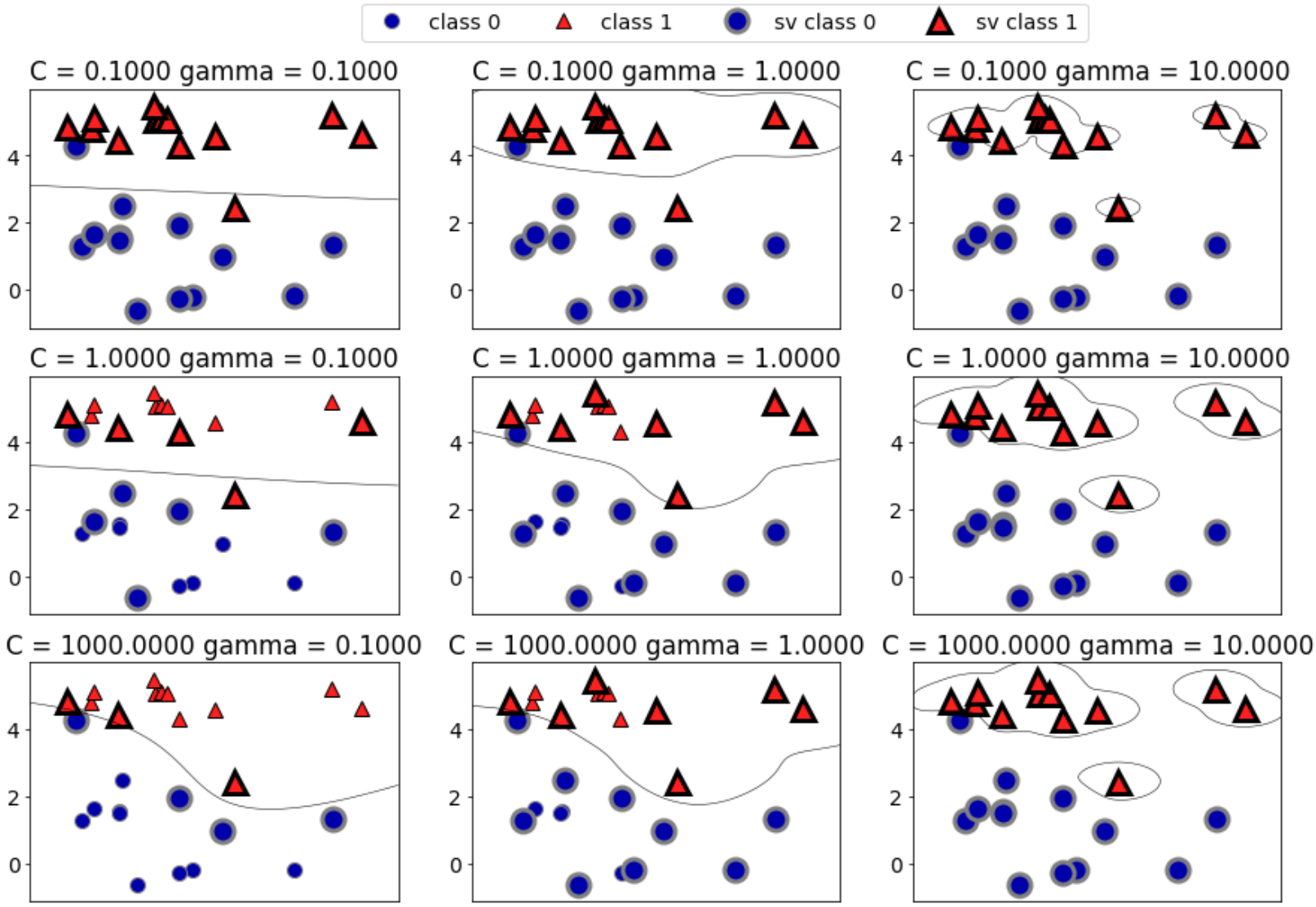
Given the support vectors, their weigths, and the kernel, we can plot the decision boundary



Tuning SVM parameters

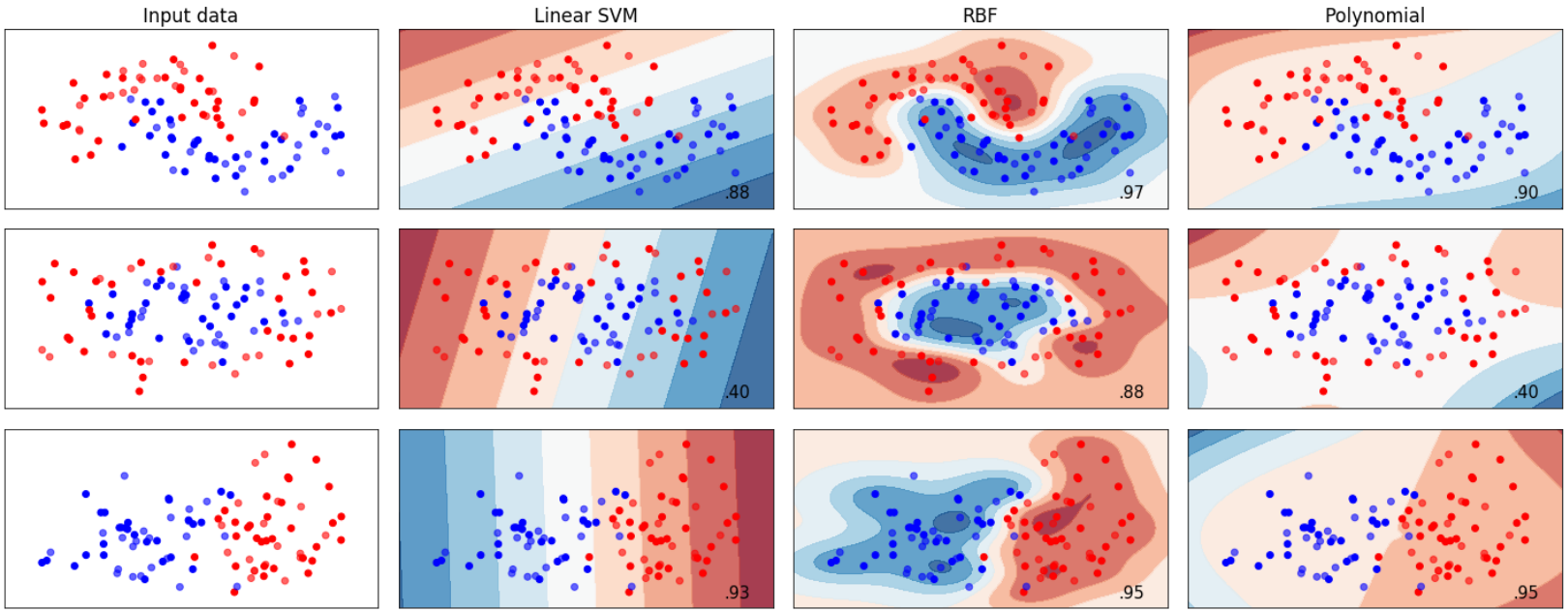
Several important parameters:

- 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
0
2
4

- 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*

Generalized linear models

We can generalize the SVM objective as follows:

$$\mathcal{J}(w) = \mathcal{R}(||w||) + \mathcal{L}(\langle w, \Phi(x_1)\rangle, \ldots, \langle w, \Phi(x_n)\rangle)$$

Where $\mathcal{R}$ is a (non-decreasing) regularization score and $\mathcal{L}$ is an *arbitrary* loss function

- The *Representer Theorem* says that if J(w) has a minimizer, it has a minimizer of the form

$$w^* = \sum_{i=1}^{n} \alpha_i \Phi(x_i)$$

- This is what we already discovered for SVMs (but with a lot more work)

Generalized linear models

- In the same way, we can define:
    - Kernelized SVMs
    - Kernelized Ridge regression
    - 1-layer neural networks
        - The 'kernel' here is the activation function
- We can also define kernels for text, graphs, and many other types of data