Part 2: Fundamental Algorithms

And how to use them

Joaquin Vanschoren, Eindhoven University of Technology

Linear models

Linear models make a prediction using a linear function of the input features.

- Can be very powerful for or datasets with many features.
- If you have more features than training data points, any target y can be perfectly modeled (on the training set) as a linear function.
- Even non-linear data (or non-linearly seperable data) can be modelled with linear models with a bit of preprocessing.
    - Basis for 'Generalized Linear Models' (e.g. kernelized SVMs, see lecture 2)

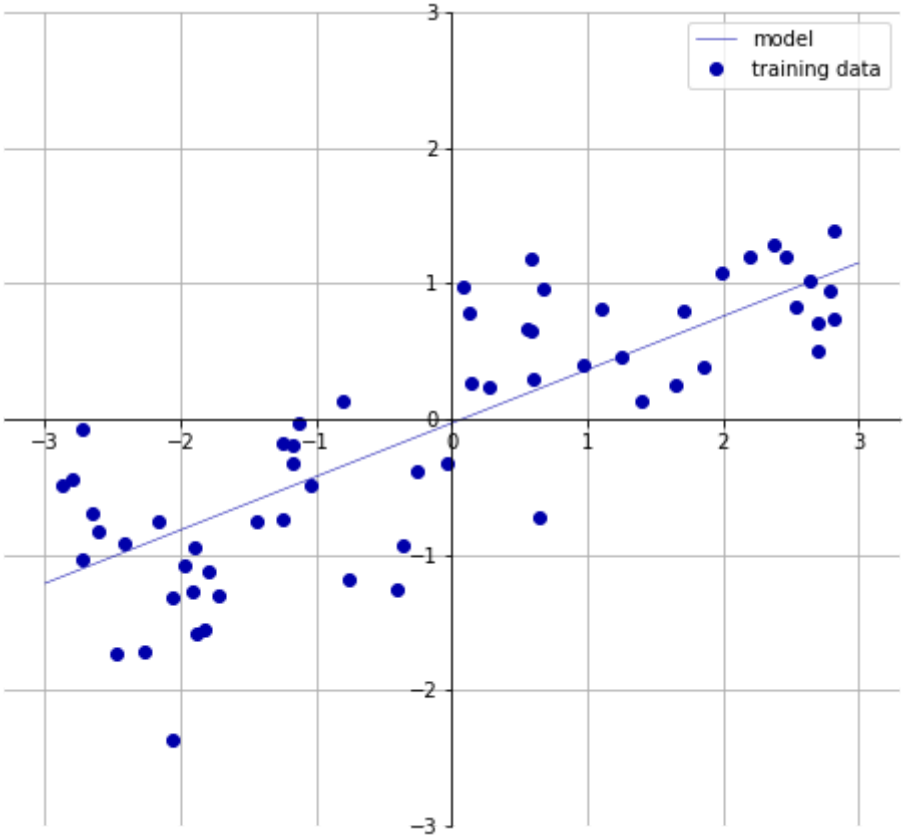
Linear models for regression

Prediction formula for input features x. w_i and b are the *model parameters* that need to be learned.

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

There are many different algorithms, differing in how w and b are learned from the training data.

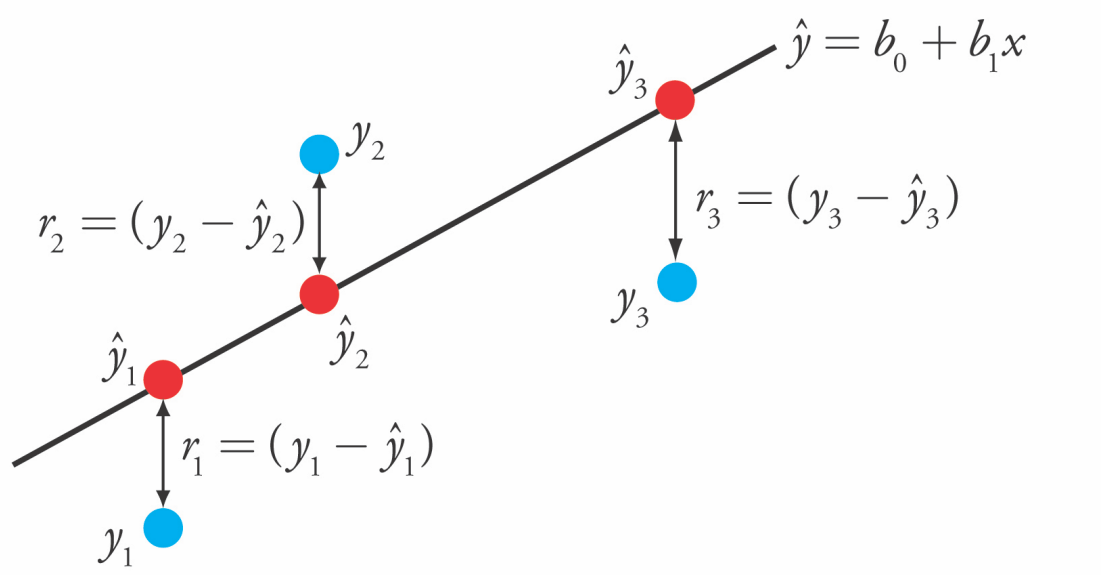
```
w[0]: 0.393906  b: -0.031804
```



Linear Regression aka Ordinary Least Squares

- Finds the parameters w and b that minimize the *mean squared error* between predictions (red) and the true regression targets (blue), y, on the training set.
    - MSE: Sum of the squared differences (residuals) between the predictions $\hat{y}_i$ and the true values y_i.

$$\mathcal{L}_{MSE} = \sum_{n=0}^{N}(y_n - \hat{y}_n)^2 = \sum_{n=0}^{N}(y_n - (\mathbf{wx_n} + b))^2$$



Solving ordinary least squares

- Convex optimization problem with unique closed-form solution (if you have more data points than model parameters w)
- It has no hyperparameters, thus model complexity cannot be controlled.
- It **very easily overfits**. What does that look like?
    - model parameters w become very large (steep incline/decline)
    - a small change in the input x results in a very different output y

Linear regression can be found in `sklearn.linear_model`. We'll evaluate it on the Boston Housing dataset.

```
lr = LinearRegression().fit(X_train, y_train)
```

```
Weights (coefficients): [ -412.711   -52.243  -131.899   -12.004   -15.511
28.716    54.704
   -49.535    26.582    37.062   -11.828   -18.058   -19.525    12.203
  2980.781  1500.843   114.187   -16.97     40.961   -24.264    57.616
  1278.121 -2239.869   222.825    -2.182    42.996   -13.398   -19.389
    -2.575   -81.013     9.66      4.914    -0.812    -7.647    33.784
   -11.446    68.508   -17.375    42.813     1.14 ]
Bias (intercept): 30.934563673645666

Training set score (R^2): 0.95
Test set score (R^2): 0.61
```

Ridge regression

- Same formula as linear regression
- Adds a penalty term to the least squares sum:

$$\mathcal{L}_{Ridge} = \sum_{n=0}^{N} (y_n - (\mathbf{w}\mathbf{x_n} + b))^2 + \alpha \sum_{i=0}^{p} w_i^2$$

- Requires that the coefficients (w) are close to zero.
    - Each feature should have as little effect on the outcome as possible
- Regularization: explicitly restrict a model to avoid overfitting.
- Type of L2 regularization: prefers many small weights
    - L1 regularization prefers sparsity: many weights to be 0, others large

`Ridge` can also be found in `sklearn.linear_model`.

```
ridge = Ridge().fit(X_train, y_train)
```

```
Training set score: 0.89
Test set score: 0.75
```

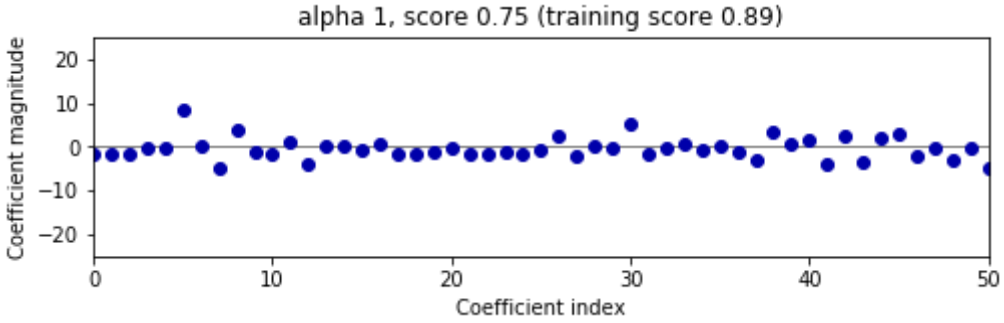
Test set score is higher and training set score lower: less overfitting!

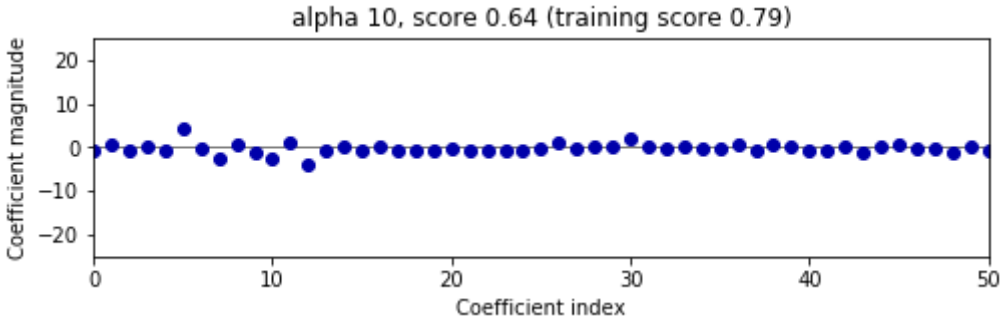
The strength of the regularization can be controlled with the `alpha` parameter. Default is 1.0.

- Increasing alpha forces coefficients to move more toward zero (more regularization)
- Decreasing alpha allows the coefficients to be less restricted (less regularization)

We can plot the weight values for differents levels of regularization. Move the slider to increase/decrease regularization. Increasing regularization decreases the values of the coefficients, but never to 0.

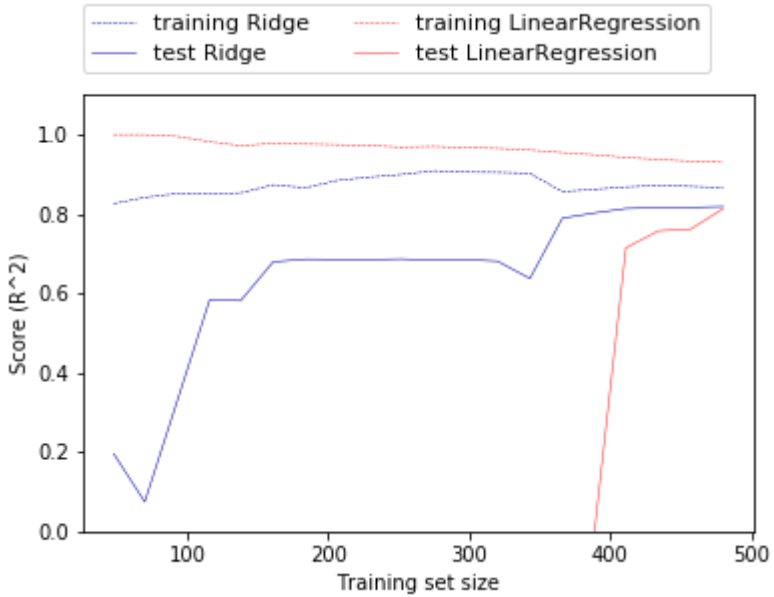

alpha 0.1, score 0.77 (training score 0.93)
Coefficient magnitude
Coefficient index
20
10
0
−10
−20
0
10
20
30
40
50


alpha 1, score 0.75 (training score 0.89)
Coefficient magnitude
Coefficient index
20
10
0
−10
−20
0
10
20
30
40
50


alpha 10, score 0.64 (training score 0.79)
Coefficient magnitude
Coefficient index
20
10
0
−10
−20
0
10
20
30
40
50

Other ways to reduce overfitting:

- Add more training data: with enough training data, regularization becomes less important
    - Ridge and linear regression will have the same performance
- Use less features, remove unimportant ones or find a lower-dimensional embedding (e.g. PCA)
    - Less degrees of freedom
- Scaling the data may also help



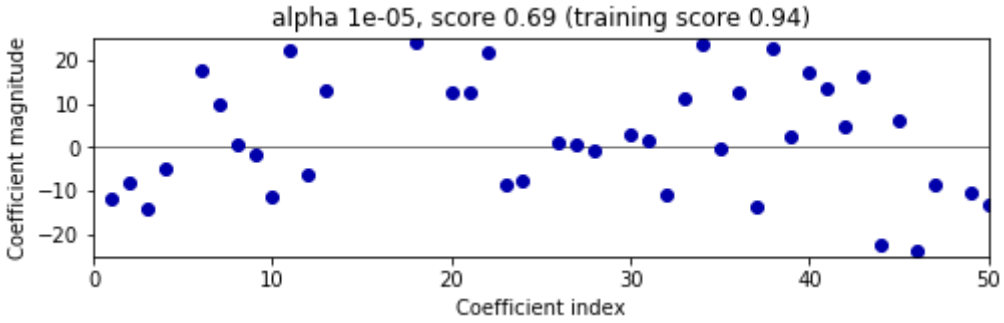
Lasso (Least Absolute Shrinkage and Selection Operator)

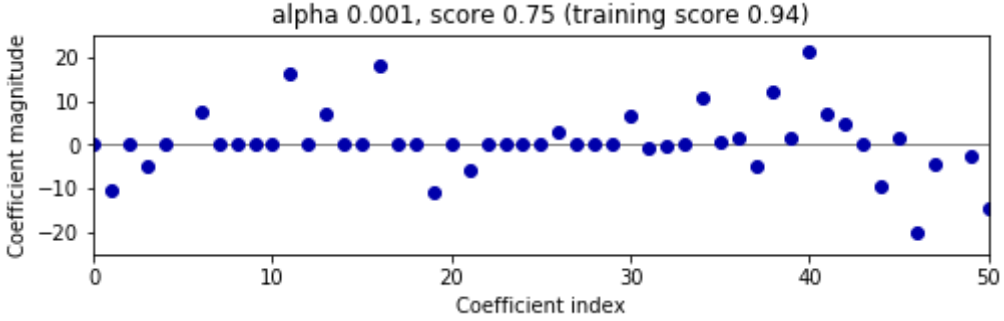
- Another form of regularization
- Adds a penalty term to the least squares sum:

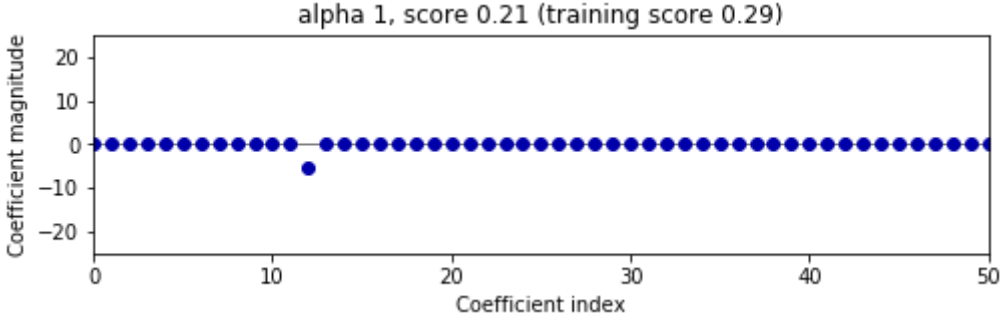
$$\mathcal{L}_{Lasso} = \sum_{n=0}^{N}(y_n - (\mathbf{w}\mathbf{x_n} + b))^2 + \alpha \sum_{i=0}^{p} |w_i|$$

- Prefers coefficients to be exactly zero (L1 regularization).
- Some features are entirely ignored by the model: automatic feature selection.
- Same parameter `alpha` to control the strength of regularization.
- Convex, but no longer strictly convex (and NOT differentiable). Weights can be optimized using (for instance) *coordinate descent*
- New parameter `max_iter`: the maximum number of coordinate descent iterations
    - Should be higher for small values of `alpha`

We can again analyse what happens to the weights. Increasing regularization under L1 leads to many coefficients becoming exactly 0.


alpha 1e-05, score 0.69 (training score 0.94)
Coefficient magnitude
Coefficient index


alpha 0.001, score 0.75 (training score 0.94)
Coefficient magnitude
Coefficient index


alpha 1, score 0.21 (training score 0.29)
Coefficient magnitude
Coefficient index

Linear models for Classification

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

There are many algorithms for learning linear classification models, differing in:

- Loss function: evaluate how well the linear model fits the training data
- Regularization techniques

Most common techniques:

- Logistic regression:
    - `sklearn.linear_model.LogisticRegression`
- Linear Support Vector Machine:
    - `sklearn.svm.LinearSVC`

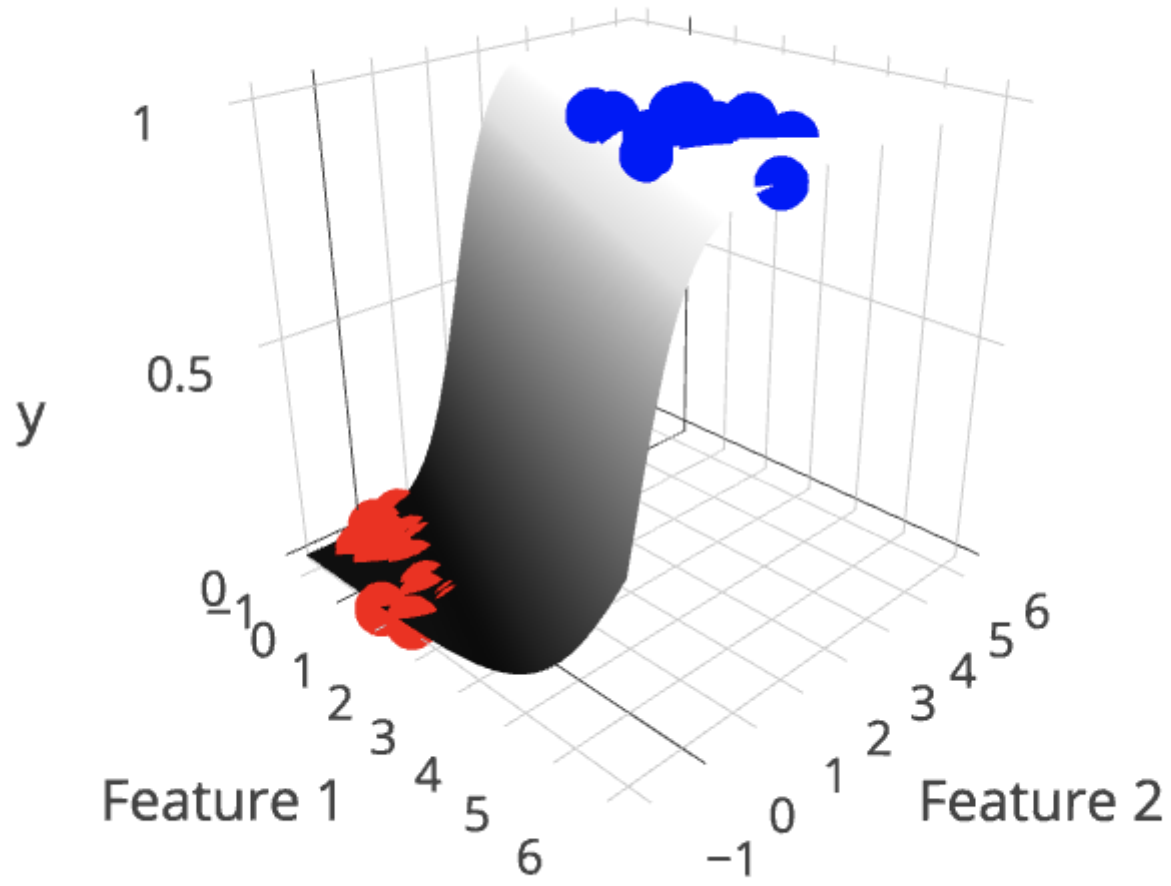
Logistic regression

- Transforms the classification problem into a regression problem
- Maps positive examples to value 1, others to value 0
- Fits a *logistic* (or *sigmoid*) function to predict whether a given sample belongs to class 1.
    - y value can be seen as a probability that the example is positive

$$z = f(x) = w_0 * x_0 + w_1 * x_1 + \ldots + w_p * x_p$$

$$\hat{y} = Pr[1|x_1, \ldots, x_k] = g(z) = \frac{1}{1 + e^{-z}}$$

On 2-dimensional data:



- The logistic function is chosen because it maps values (-Inf,Inf) to a probability [0,1]
- We add a new dimension for the dependent variable y and fit the logistic function g(z) so that it separates the samples as good as possible. The positive (blue) points are mapped to 1 and the negative (red) points to 0.
- After fitting, the logistic function provides the probability that a new point is positive. If we need a binary prediction, we can threshold at 0.5.
- There are different ways to find the optimal parameters w that fit the training data best

Loss function: cross-entropy

- Since logistic regression returns a probability, we want to use that in the loss function rather than choosing an an arbitrary threshold (e.g. positive in y > 0.5).
- We can measure the difference between the actual probabilities p_i and the predicted probabilities q_i is the cross-entropy $H(p, q)$:

$$H(p, q) = -\sum_i p_i log(q_i)$$

- Note: This is also called *maximum likelihood* estimation because instead of minimizing cross-entropy $H(p, q)$, you can maximize *log-likelihood* $-H(p, q)$
- In binary classification, $i = 0, 1$ and $p_1 = y$, $p_0 = 1 - y$, $q_1 = \hat{y}$, $q_0 = 1 - \hat{y}$
- This yields *binary cross-entropy*:

$$H(p, q) = -ylog(\hat{y}) - (1 - y)log(1 - \hat{y})$$

Cross-entropy loss

- Loss function: the average of all cross-entropies in the sample (of N data points):

$$\mathcal{L}_{log}(\mathbf{w}) = \sum_{n=1}^{N} H(p_n, q_n) = \sum_{n=1}^{N} \Big[- y_n log(\hat{y_n}) - (1 - y_n) log(1 - \hat{y_n}) \Big]$$

  with

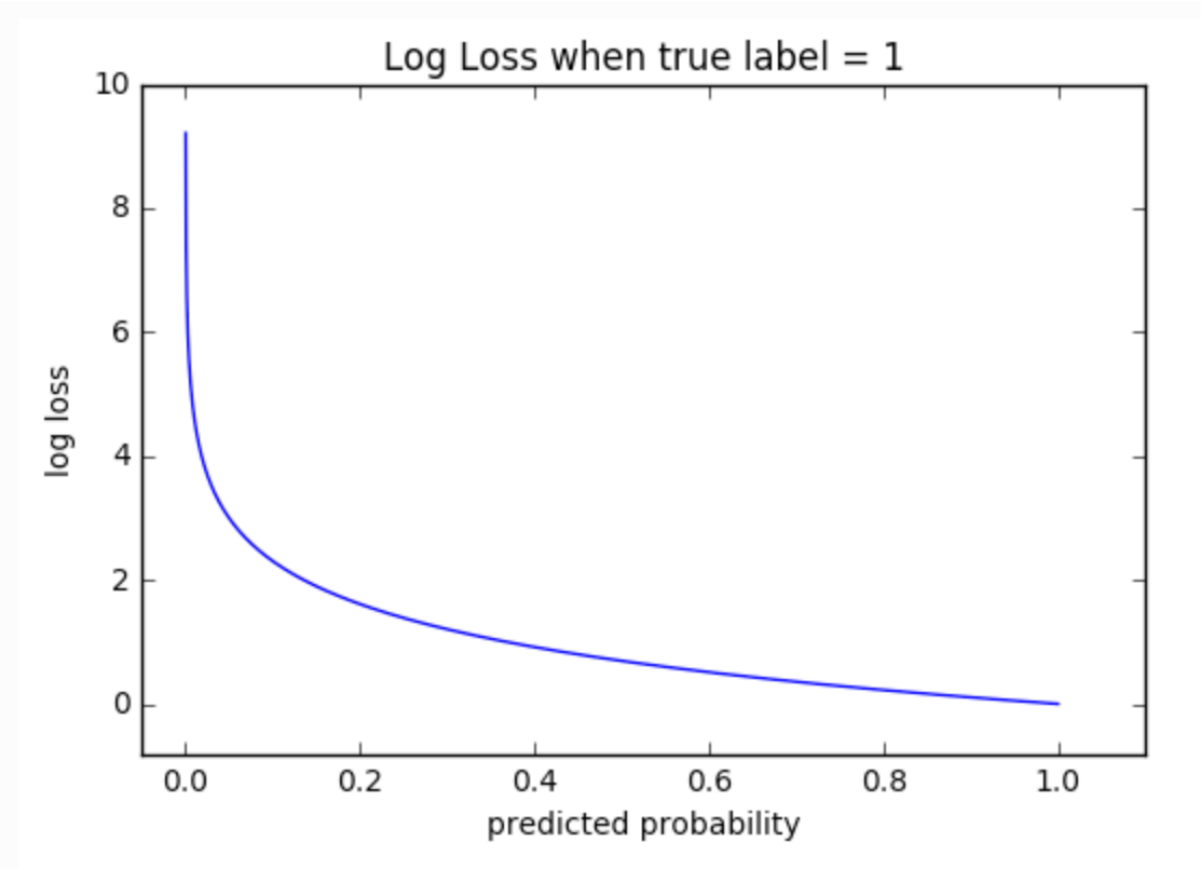
$$\hat{y_n} = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}}$$

- This is called *logistic loss*, *log loss* or *cross-entropy loss*
- We can (and should always) add a regularization term, either L1 or L2, e.g. for L2:

$$\mathcal{L}_{log}{}'(\mathbf{w}) = \mathcal{L}_{log}(\mathbf{w}) + \alpha \sum_{i} w_i^2$$

  - Note: sklearn uses C instead of α, and it is the inverse (smaller values, more regularization)

Cross-entropy loss



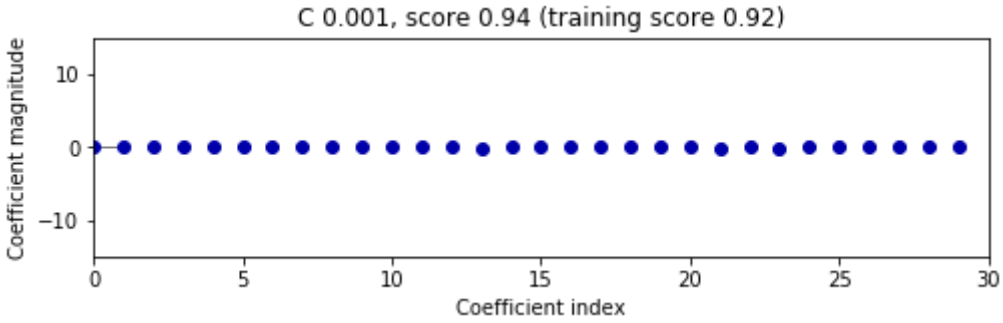
Optimization methods (solvers)

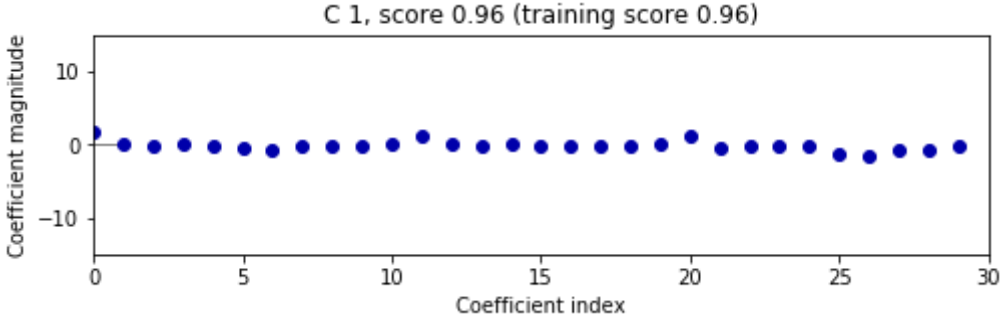
- There are different ways to optimize cross-entropy loss.
- Gradient descent
    - The logistic function is differentiable, so we can use (stochastic) gradient descent
    - Stochastic Average Gradient descent (SAG): only updates gradient in one direction at each step
- Coordinate descent (default, called `liblinear` in sklearn)
    - Faster, may converge more slowly, may more easily get stuck in local minima
- Newton-Rhapson (or Newton Conjugate Gradient):
    - Finds optima by computing second derivatives (more expensive)
    - Works well if solution space is (near) convex
    - Also known as *iterative re-weighted least squares*
- Quasi-Newton methods
    - Approximate, faster to compute
    - E.g. Limited-memory Broyden–Fletcher–Goldfarb–Shanno (`lbfgs`)

Model selection: Logistic regression

```
logreg = LogisticRegression(C=1).fit(X_train, y_train)
```

Adjust the slider to see the effect of C and L1/L2 regularization

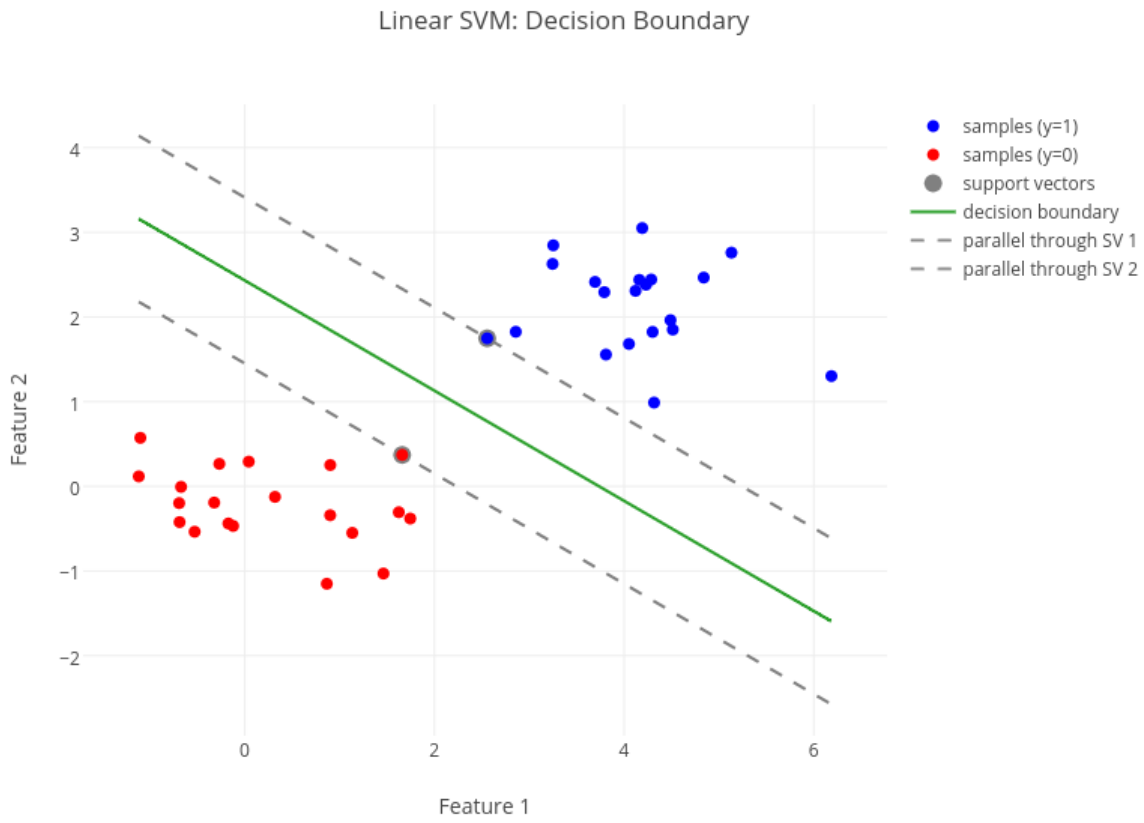

C 0.001, score 0.94 (training score 0.92)
Coefficient magnitude
10
0
−10
0
5
10
15
20
25
30
Coefficient index


C 1, score 0.96 (training score 0.96)
Coefficient magnitude
10
0
−10
0
5
10
15
20
25
30
Coefficient index


C 100, score 0.97 (training score 0.97)
Coefficient magnitude
10
0
−10
0
5
10
15
20
25
30
Coefficient index

Linear Support Vector Machine

Don't minimize the (misclassification) loss, but maximize the *margin* between the classes.
That likely generalizes better!



Optimization and prediction

- Find a small number of data points to define the decision boundary (support vectors)
    - Each support vector has a weight (some are more important than others)
    - Called *dual coefficients* (dual: for every point vs for every feature)
    - Hence, this is a non-parametric model
- Prediction is identical to (weighted) kNN:
    - Points closest to a red support vector are classified red, others blue
- The objective function penalizes every point predicted to be on the `wrong` side of its hyperplane
    - This is called *hinge loss*
- This results in a convex optimization problem solved using the *Langrange Multipliers* method
    - Can also be solved using gradient descent

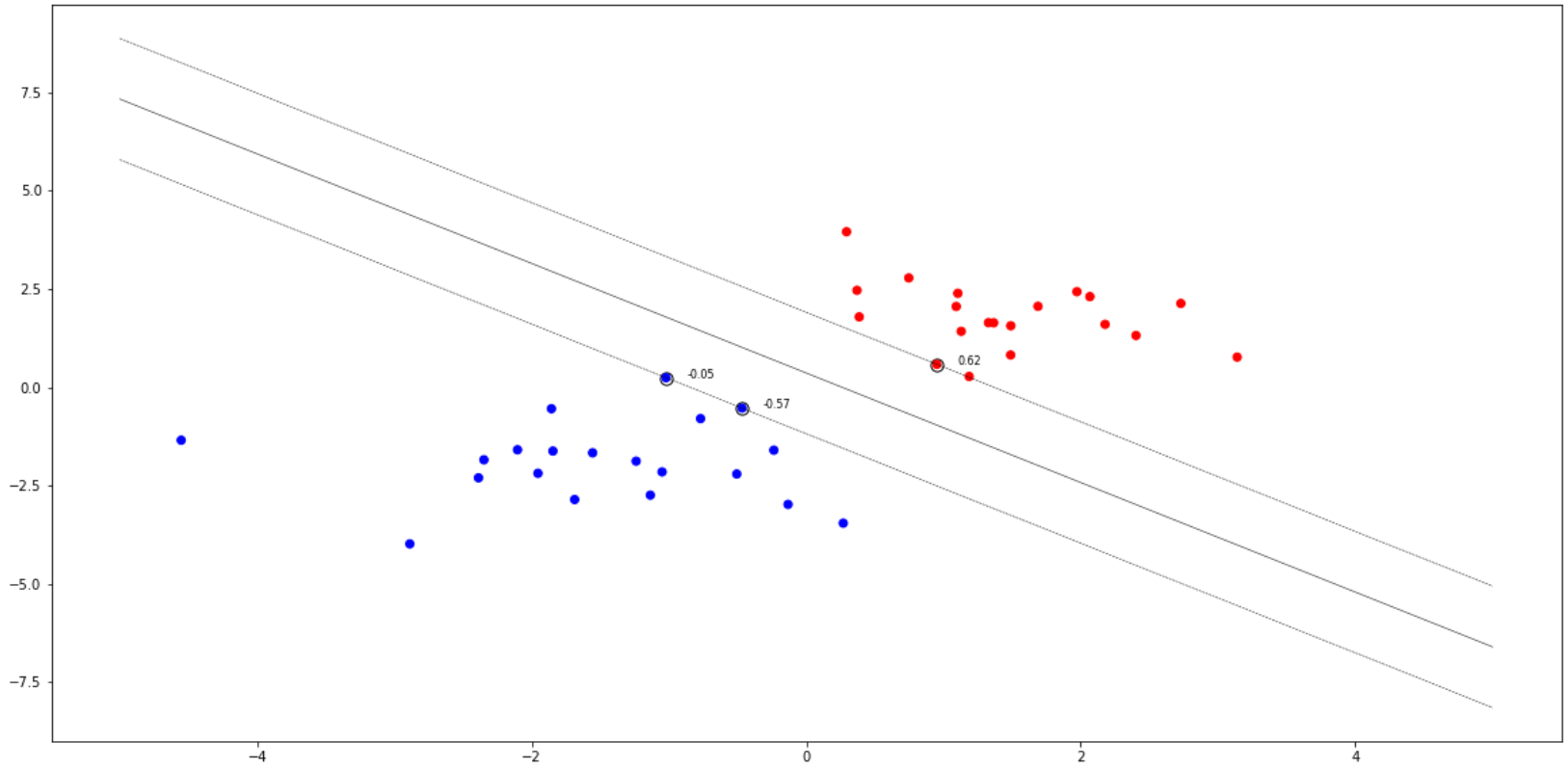
Example: SVMs in scikit-learn

- We can use the `svm.SVC` classifier
    - or `svm.SVR` for regression
    - it only support the dual loss function
- 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

- We can allow for violations of the margin constraint by introducing a *slack variable* $\xi^{(i)}$ for every data point. The new objective (to be minimized) becomes:

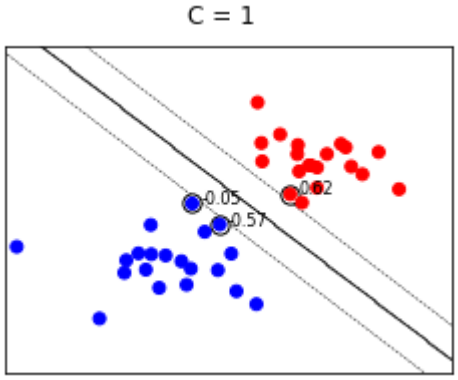
$$\frac{||w||^2}{2} + C(\sum_i \xi^{(i)})$$

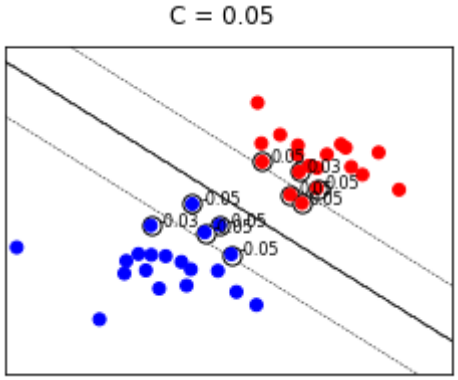
- C is a penalty for misclassification
    - Large C: large error penalties
    - Small C: less strict about violations (more regularization)
- This is known as the *soft margin* SVM (or *large margin* SVM)

C and regularization

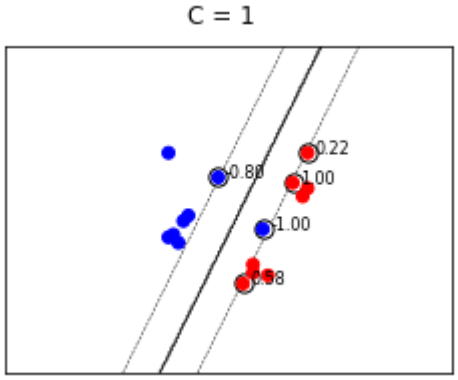
- Hence, we can use C to control the size of the margin and overfitting:
    - Small C: Violations allowed, simple model, more underfitting
    - Large C: Model is more strict, more overfitting
- The penalty term $C(\sum_i \xi^{(i)})$ acts as an L1 regularizer on the dual coefficients
    - Also known 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'`)

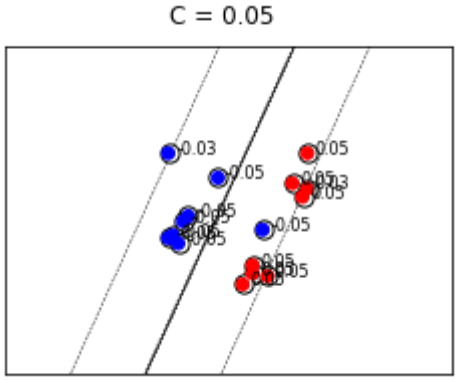
Effect on linearly separable data



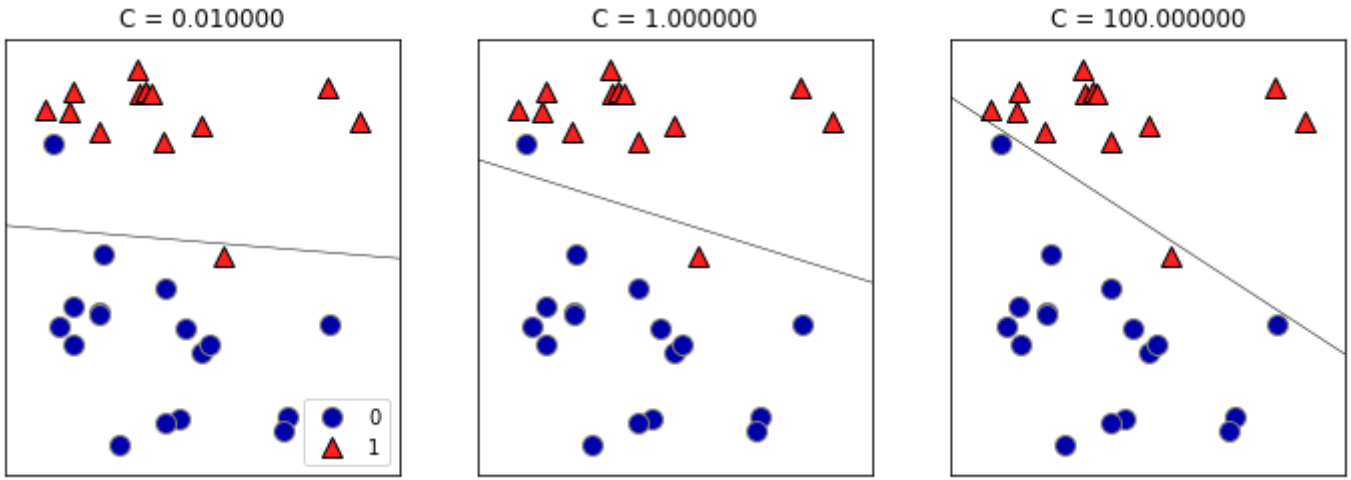


Effect on non-linearly separable data





SVM: High *C* values (less regularization): fewer misclassifications but smaller margins, overfitting.



Kernelization (Generalized linear models)

Feature Maps

- Remember linear models?

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

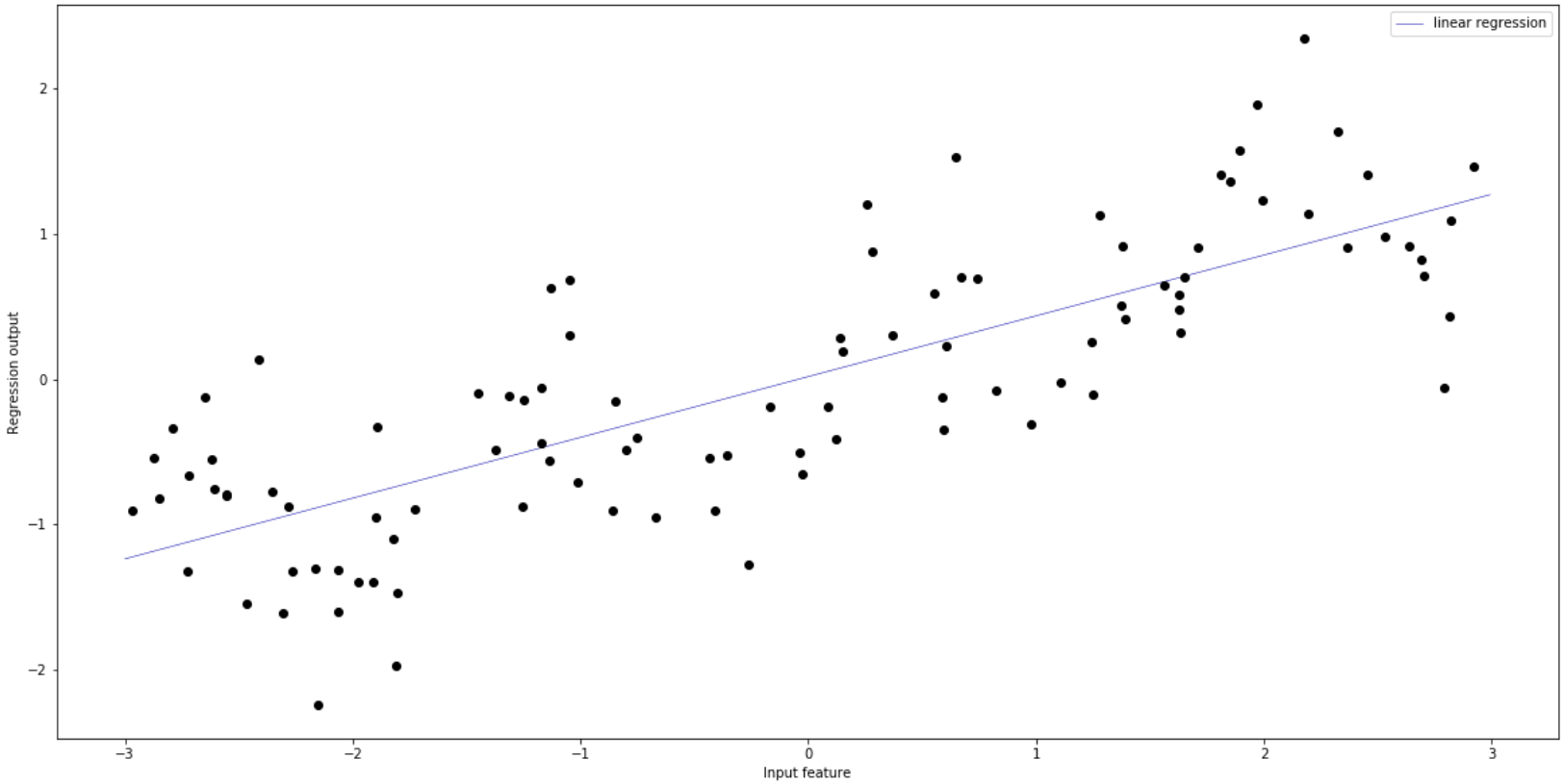
- When we cannot fit the data well with linear models, we can learn more complex models by simply adding more dimensions
- Feature map (or *basis expansion*) ϕ: $X \rightarrow \mathbb{R}^d$

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

- You still may need MANY dimensions to fit the data
  - Memory and computational cost
  - More likely overfitting

Example: Ridge regression

```
Coefficients: [0.418]
```



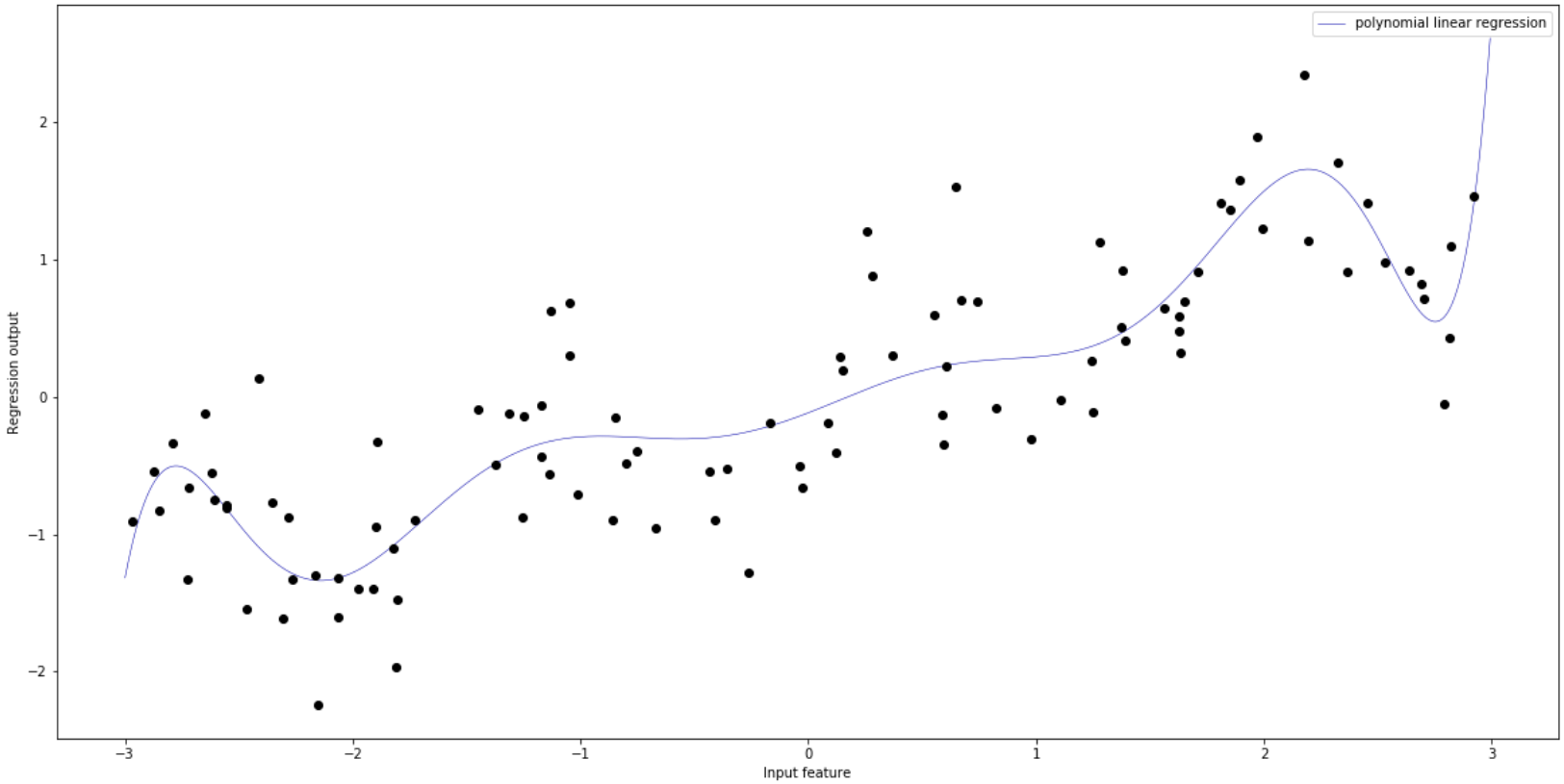
- Add all polynomials x^d up to degree D. How large should D be?
- We can also compute all polynomials and all interactions between features (e.g. $x \cdot x^2$). This leads to D^2 features.

Out[3]:

	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]
```



How expensive is this?

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

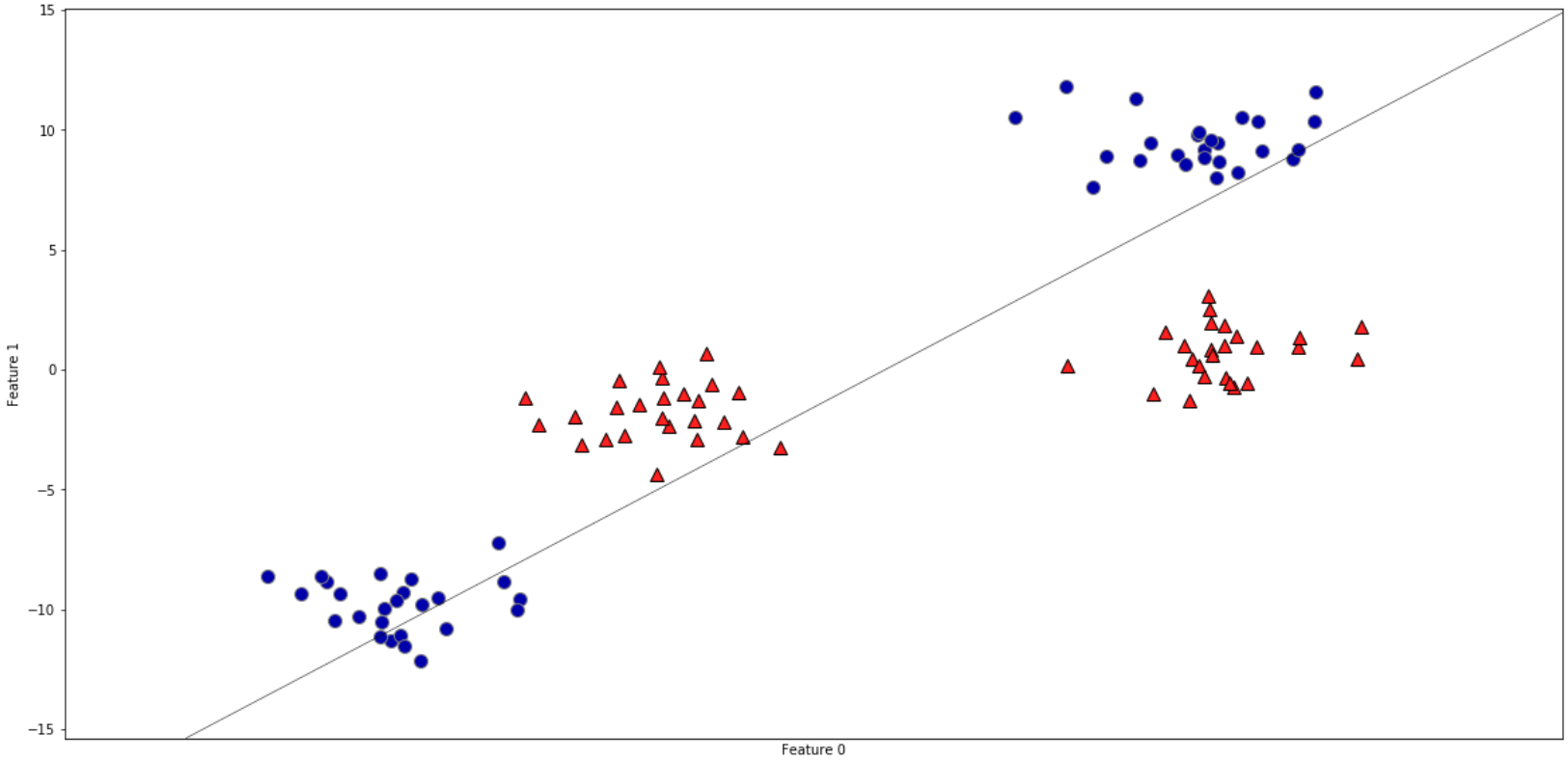
$$w^* = (X^T X + \lambda I)^{-1} X^T Y$$

- 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

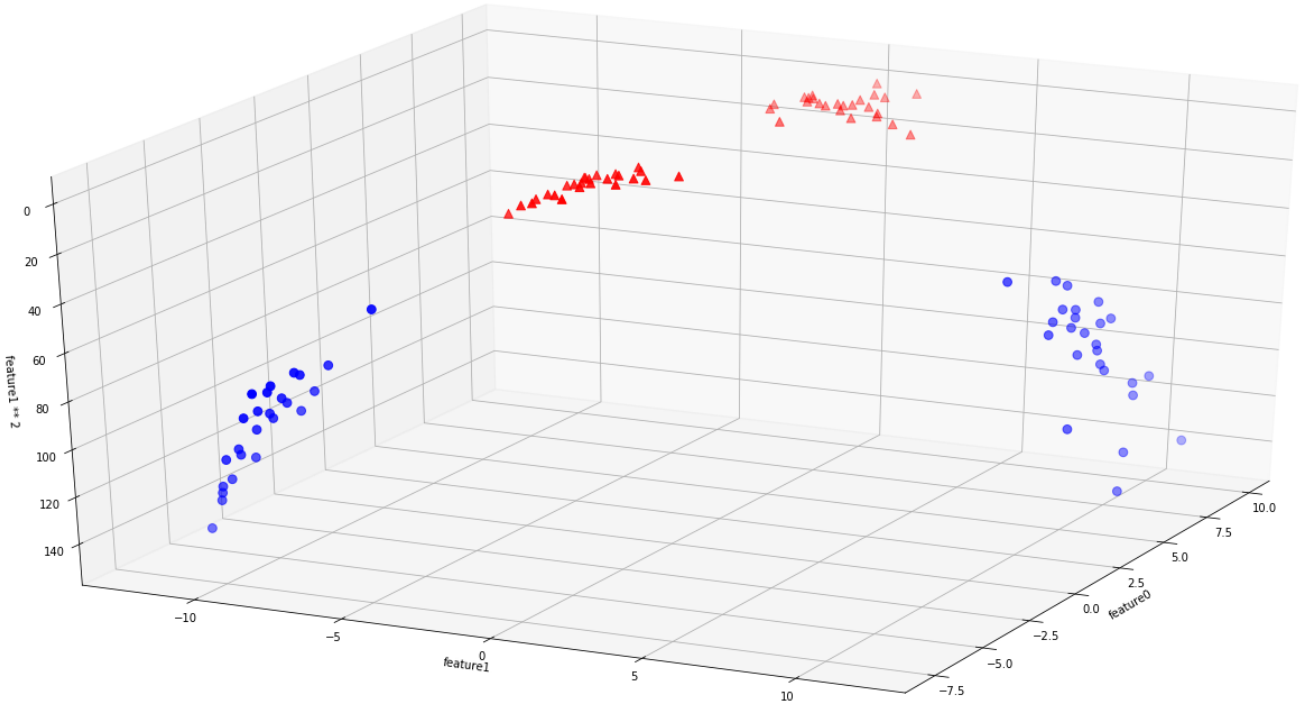
$$w^* = (\Phi(X)^T \Phi(X) + \lambda I)^{-1} \Phi(X)^T Y$$

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

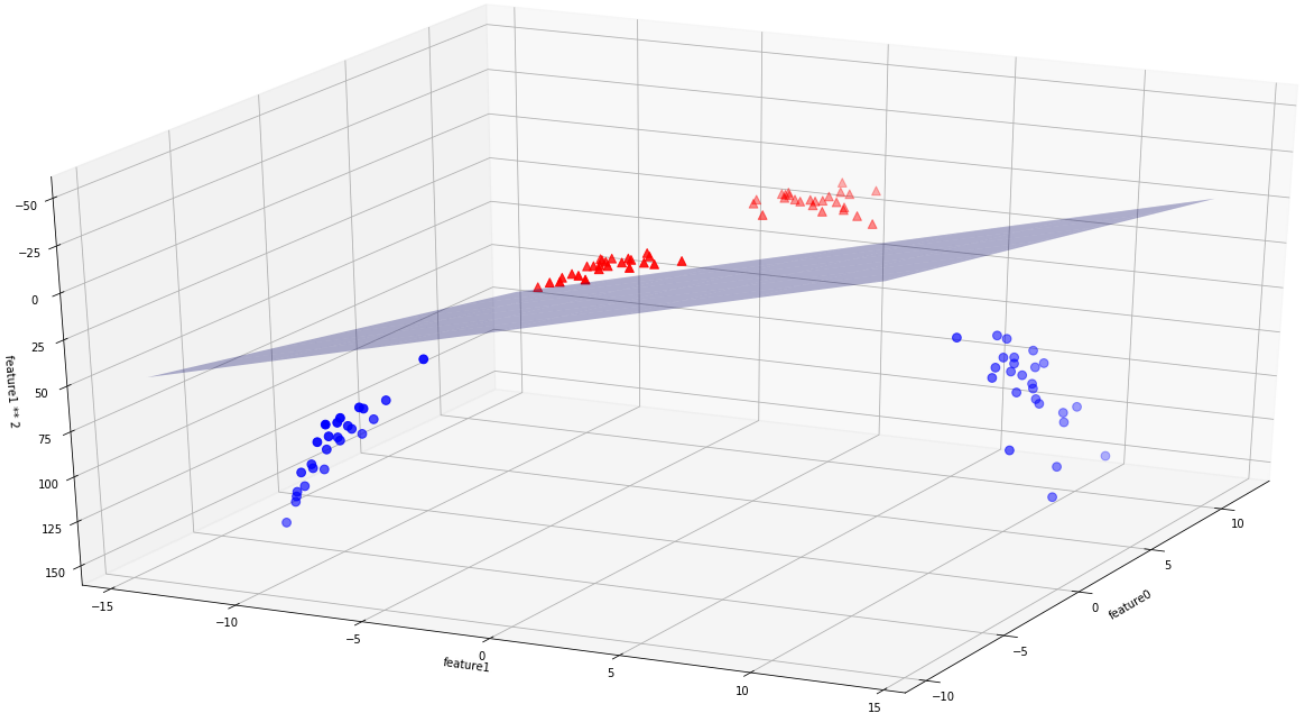
You can do the same for classification.



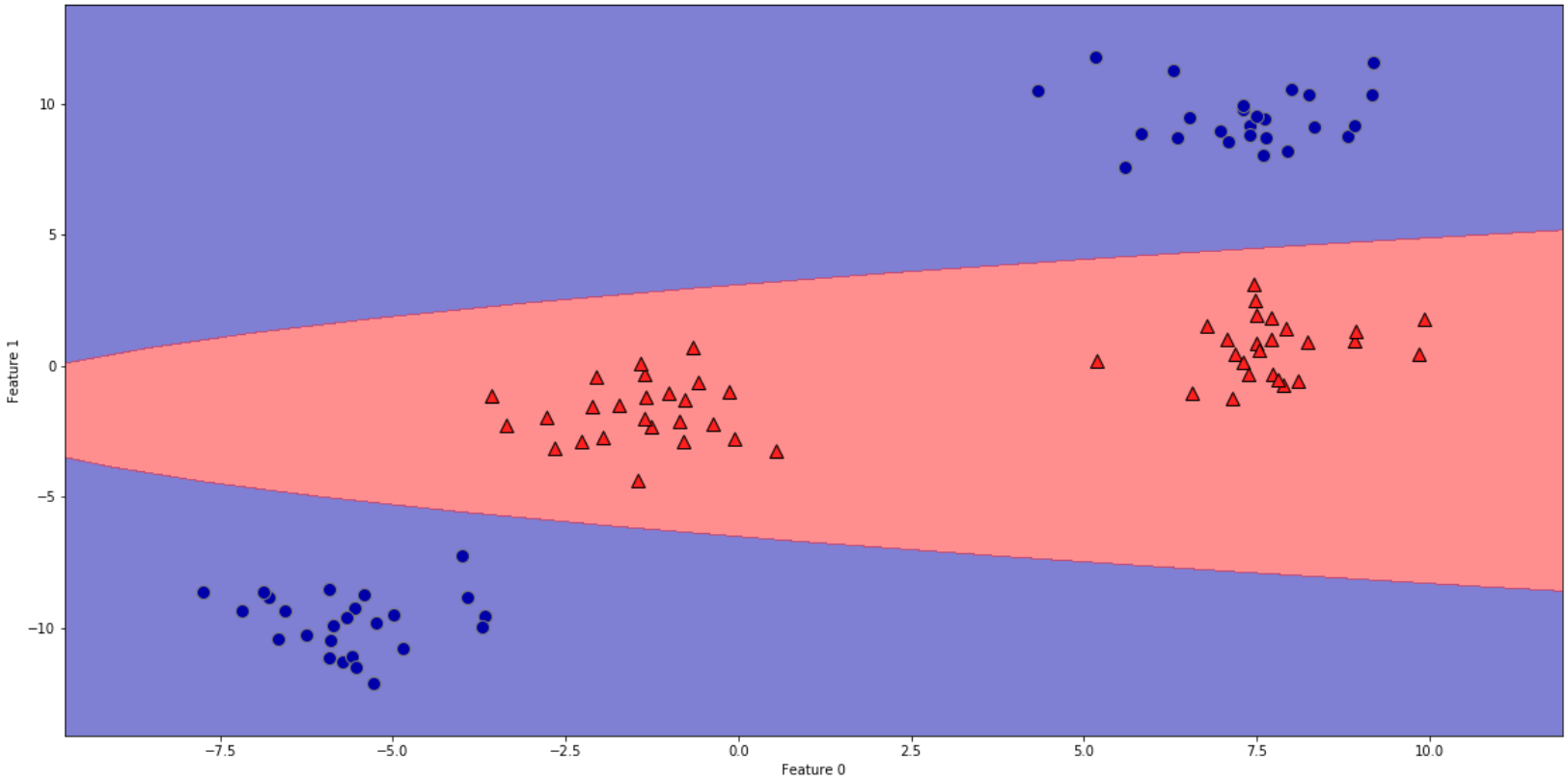
We can add a new feature by taking the squares of feature1 values



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



Kernelization

- So, $\Phi(x)$ can be used to generate many more features based on the original feature x
- Useful, but expensive to evaluate.
- A *kernel function* corresponding to a feature transformation Φ is

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

- It measures a kind of *similarity* between x_i and x_j, $\langle . , . \rangle$ is the dot product
- Turns out, we can often evaluate $k(x_i, x_j)$ directly, *without* evaluating $\Phi(x_i), \Phi(x_j)$

Kernel trick

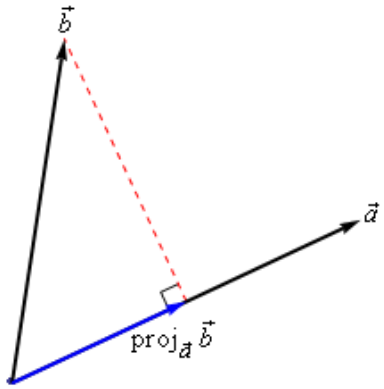
- Evaluating the kernel directly can be *much* cheaper.
- Example: a simple *quadratic* feature map for $x = (x_1, \ldots, x_d)$ has dimension $\mathcal{O}(d^2)$:
$$\Phi(x) = (x_1, \ldots, x_d, x_1^2, \ldots, x_d^2, \sqrt{2}x_1x_2, \ldots, \sqrt{2}x_{d-1}x_d)$$
- The corresponding quadratic kernel is:
$$k(x_i, x_j) = \langle \Phi(x_i), \Phi(x_j) \rangle = \langle x_i, x_j \rangle + \langle x_i, x_j \rangle^2$$
- 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

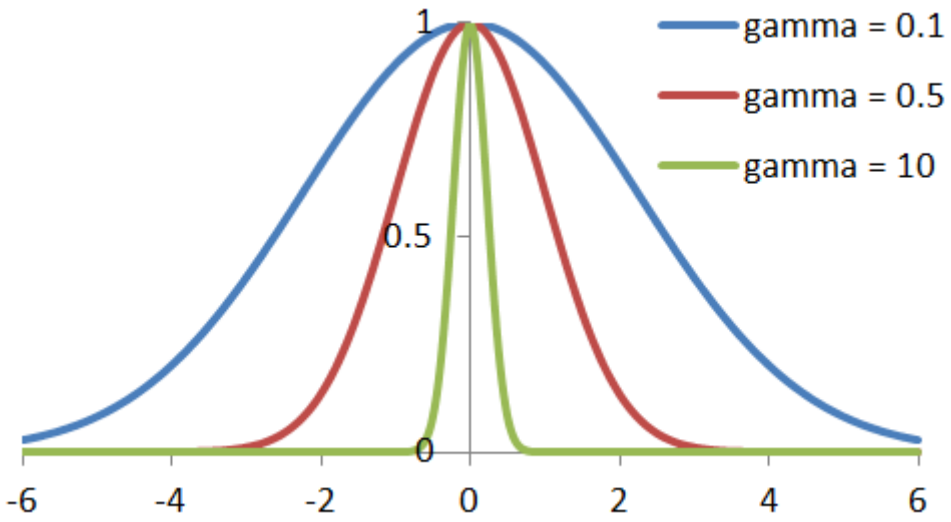
- Input space is same as output space: $X = \mathcal{H} = \mathbb{R}^d$
- Feature map $\Phi(x) = x$
- Kernel: $k(x_i, x_j) = x_i \cdot x_j = x_i^T x_j$
- Geometrically, we can view these as *projections* of x_j on a hyperplane defined by x_i
    - Nearby points will have nearby projections



Kernels: examples

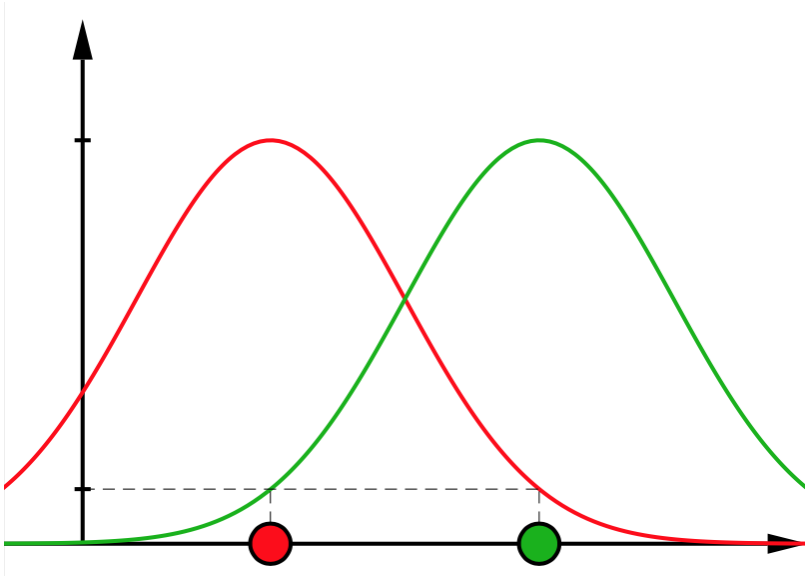
- The inner product is a kernel. The standard inner product is the **linear kernel**:
  $k(x_1, x_2) = x_1^T x_2$
- Kernels can be constructed from other kernels k_1 and k_2:
  - For $\lambda \geq 0$, $\lambda . k_1$ is a kernel
  - $k_1 + k_2$ is a kernel
  - $k_1 . k_2$ is a kernel (thus also k_1^n)
- 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 function' (or **Gaussian**) kernel is defined as: $k(x_1, x_2) = exp(-\gamma||x_1 - x_2||^2)$, for $\gamma \geq 0$



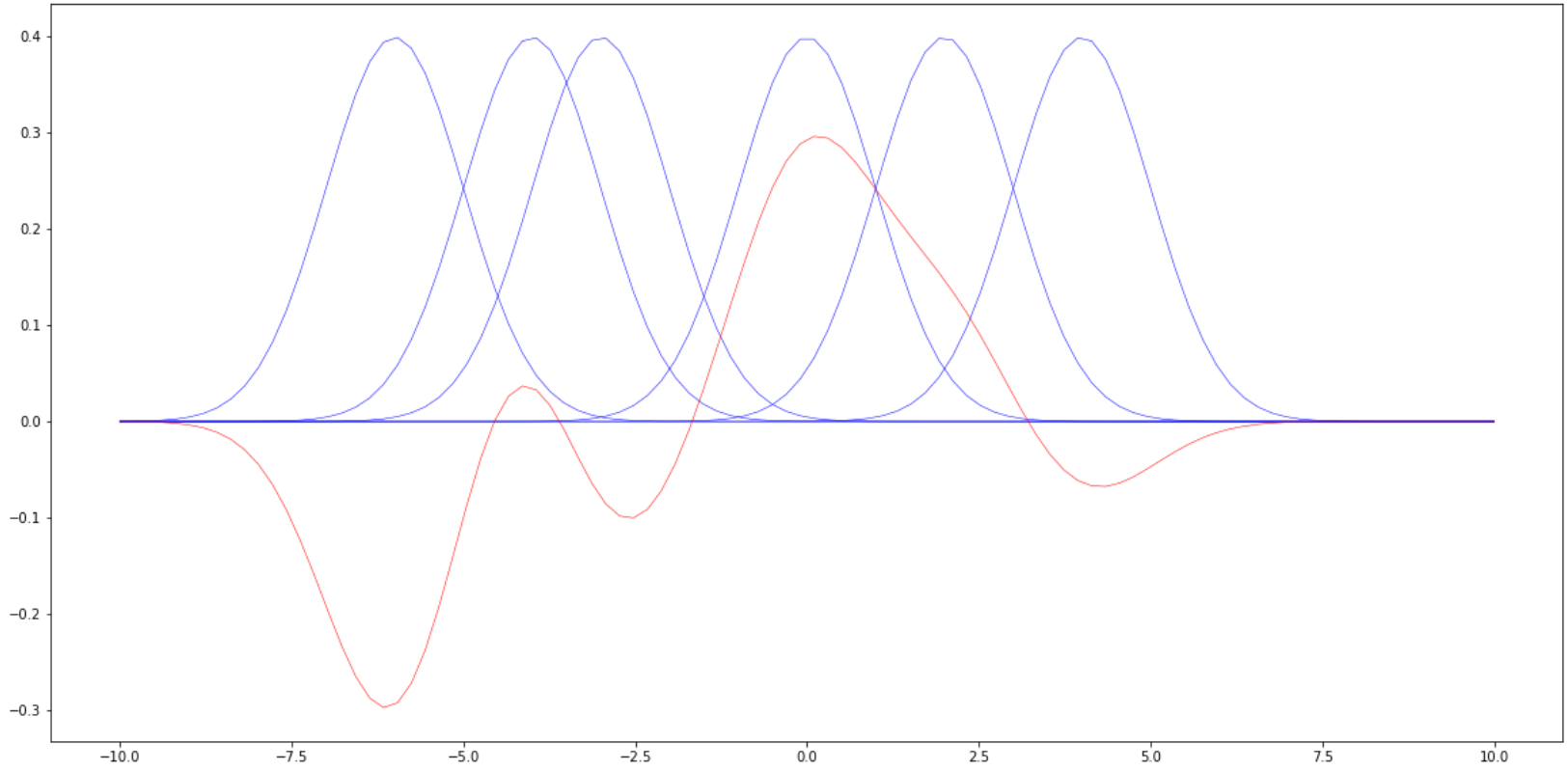
Gaussian kernel: intuition

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



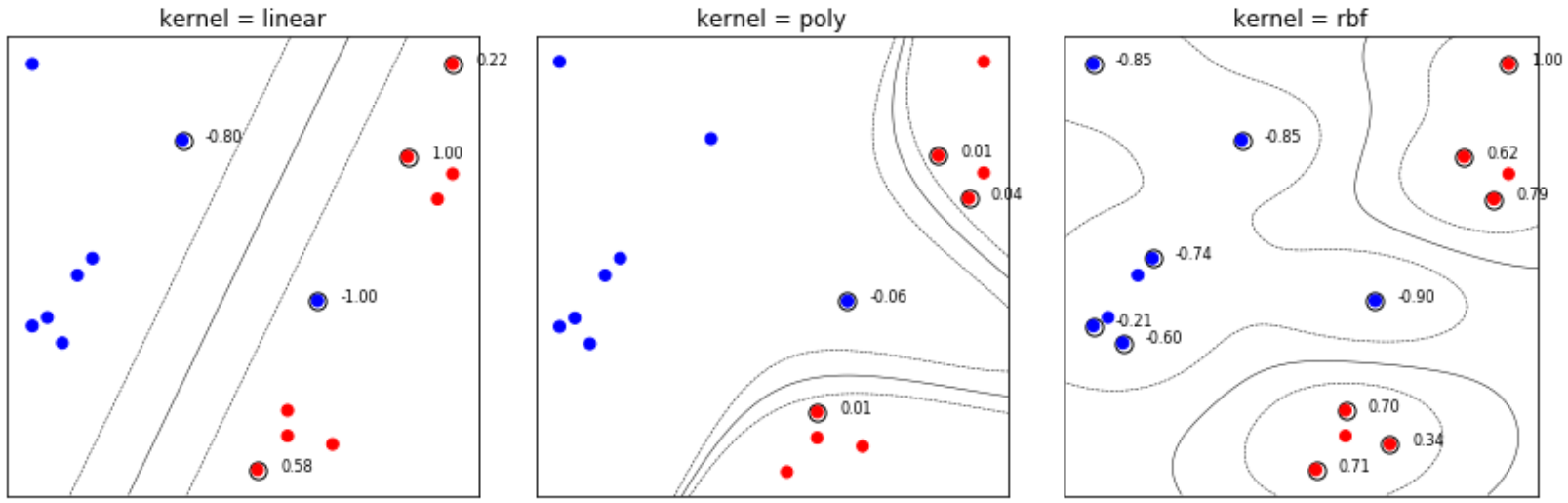
Example (for regression):

- 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)
- Linear kernels will produce a linear function, Gaussian kernels can produce very complex functions.


0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

Example (for classification):

- In the RBF SVM, every support vector generates a 2D Gaussian, the final prediction is the sum of those
- At prediction time, you evaluate each Gaussian (a kind of distance between the new point and the support vector) and sum up the values



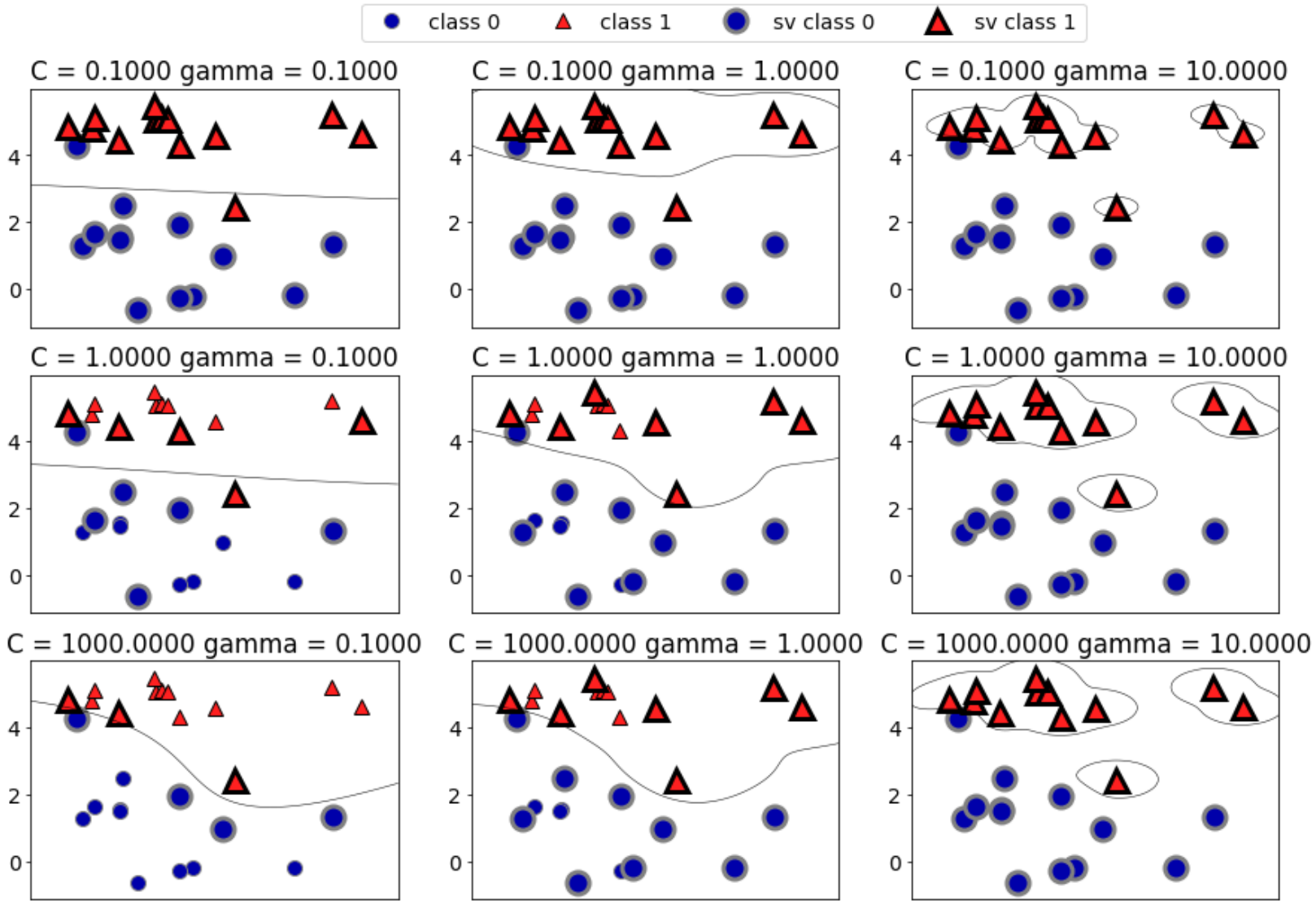
Local vs Global kernels

- 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

Tuning SVM parameters

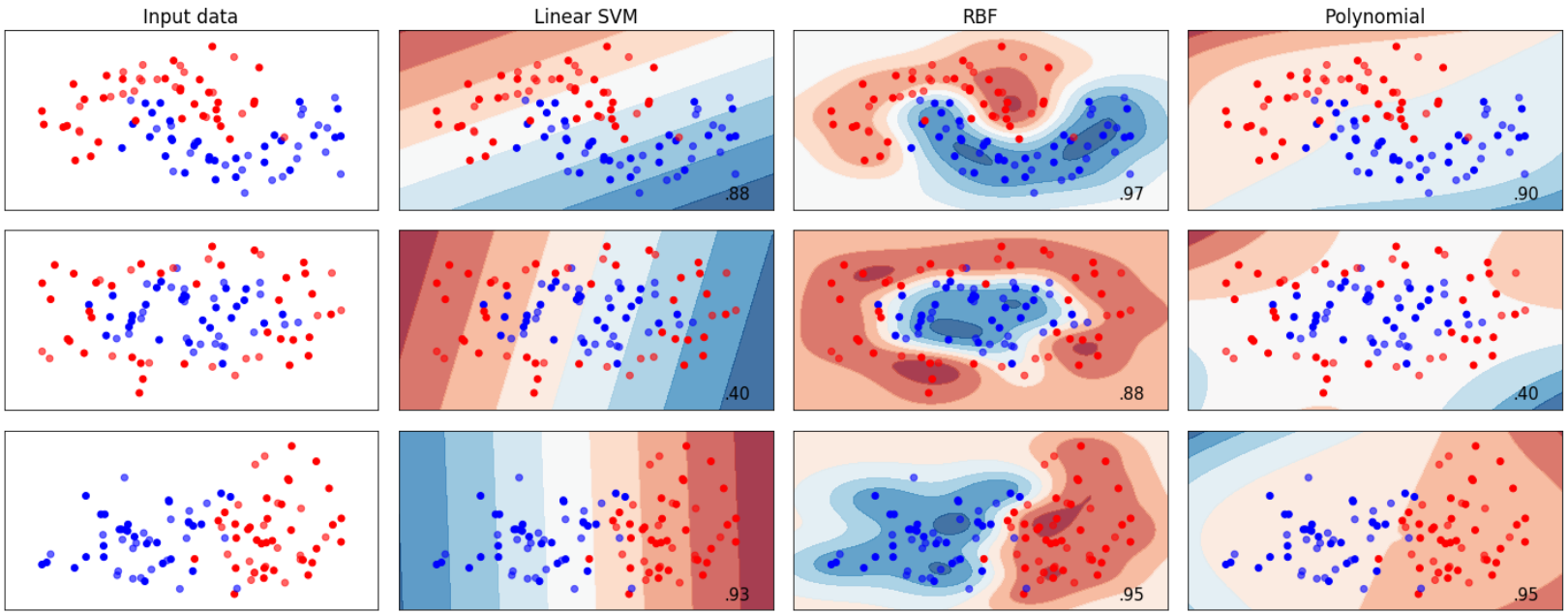
Several important parameters:

- 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
- We'll get back to this in Lecture 4 (pipelines).

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

- 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

Ensemble learning

Ensembles are methods that combine multiple machine learning models (weak learners) to create more powerful models. Most popular are:

- **Bagging**: Reduce variance: Build many trees on random samples and do a vote over the predictions
    - **RandomForests**: Build randomized trees on random bootstraps of the data
- **Boosting**: Reduce bias: Build trees iteratively, each correcting the mistakes of the previous trees
    - **Adaboost**: Ensemble of weighted trees, increasing importance of misclassified points
    - **Gradient boosting machines**: Gradually update importance of hard points until ensemble is correct
    - **XGBoost**: Faster implemenation of gradient boosting machines
- **Stacking**: Build group of base models, and train a meta-model to learn how to combine the base model predicions

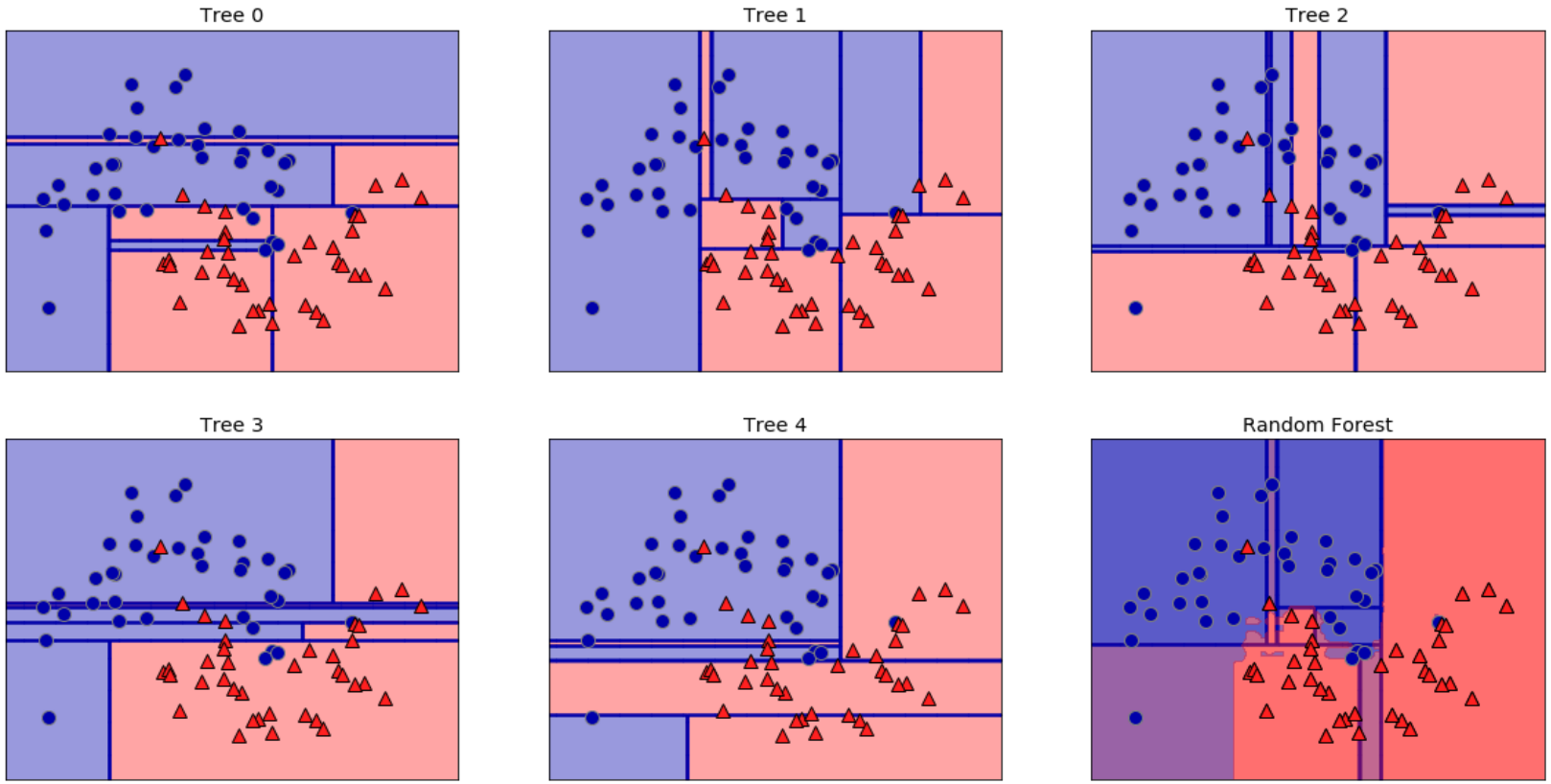
Bagging (Bootstrap Aggregating)

Reduce overfitting by averaging out individual predictions (variance reduction)

- Take a *bootstrap sample* of your data
    - Randomly sample with replacement
    - Build a tree on each bootstrap
- Repeat `n_estimators` times
    - Higher values: more trees, more smoothing
    - Make prediction by aggreting the individual tree predictions
- Can be done with any model (but usually trees)
    - Since Bagging only reduces variance (not bias), it makes sense to use models that are high variance, low bias
- RandomForest: Randomize trees by considering only a random subset of features of size `max_features` *in each node*
    - Higher variance, lower bias than normal trees
    - Small `max_features` yields more different trees, more smoothing
    - Default: $sqrt(n\_features)$ for classification, $log2(n\_features)$ for regression

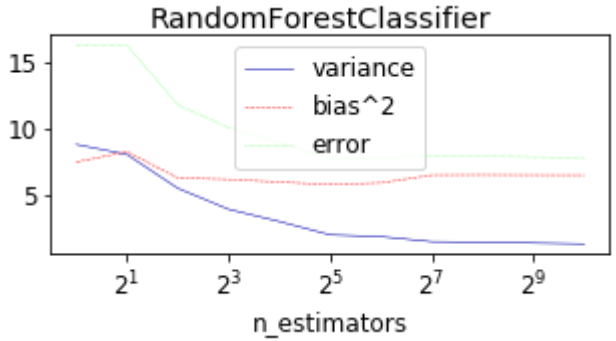
Making predictions:

- Classification: soft voting (softmax)
  - Every member returns probability for each class
  - After averaging, the class with highest probability wins
- Regression:
  - Return the *mean* of all predictions
- Each base model gets the same weight in the final prediction


Tree 0
Tree 1
Tree 2
Tree 3
Tree 4
Random Forest

Effect on bias and variance

- 'High bias' models are simple and stable: they make the same mistakes no matter which training sample they get
- 'High variance' models are complex and unstable: they make very different predictions depending on the training sample.
- In RandomForests, increasing the number of estimators decreases variance
- Bias is mostly unaffected, but will increase if the forest becomes too large (oversmoothing)



Scikit-learn algorithms:

- `RandomForestClassifier` (or Regressor)
- `ExtraTreesClassifier`: Grows deeper trees, faster

Most important parameters:

- `n_estimators` (higher is better, but diminishing returns)
    - Will start to underfit (bias error component increases slightly)
- `max_features` (default is typically ok)
    - Set smaller to reduce space/time requirements
- parameters of trees, e.g. `max_depth` (less effect)

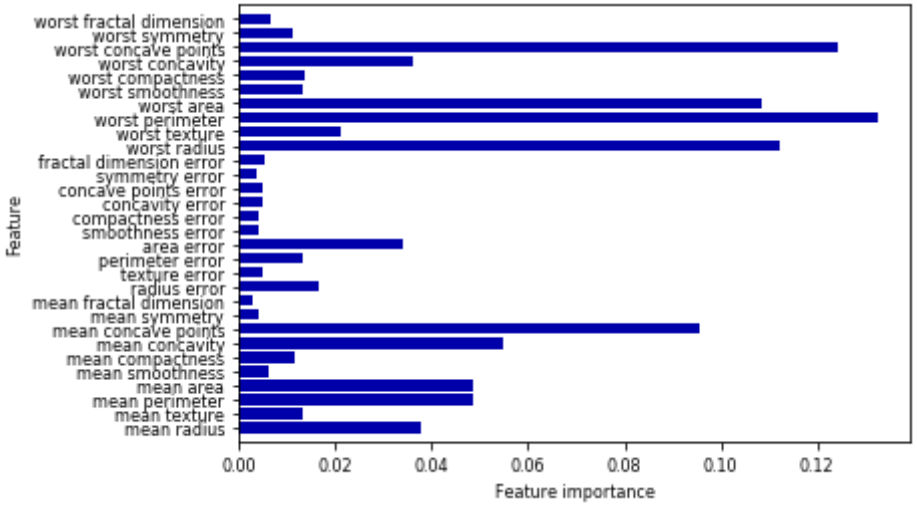
`n_jobs` sets the number of parallel cores to run
`random_state` should be fixed for reproducibility

RandomForest allow another way to evaluate performance: out-of-bag (OOB) error

- While growing forest, estimate test error from training samples
- For each tree grown, 33-36% of samples are not selected in bootstrap
    - Called the 'out of bootstrap' (OOB) samples
    - Predictions are made as if they were novel test samples
    - Through book-keeping, majority vote is computed for all OOB samples from all trees
- OOB estimated test error is rather accurate in practice
    - As good as CV estimates, but can be computed on the fly (without repeated model fitting)
    - Tends to be slightly pessimistic

Feature importance

RandomForests provide often reliable feature importances, based on many alternative hypotheses (trees)



Strengths, weaknesses and parameters

RandomForest are among most widely used algorithms:

- Don't require a lot of tuning
- Typically very accurate models
- Handles heterogeneous features well
- Implictly selects most relevant features

Downsides:

- less interpretable, slower to train (but parallellizable)
- don't work well on high dimensional sparse data (e.g. text)

Adaptive Boosting (AdaBoost)

- Builds an ensemble of *weighted* weak learners
    - Typically shallow trees or stumps
- Each base model tries to correct the mistakes of the previous ones
    - Sequential, not parallel
    - We give misclassified samples more weight
- Force next model to get these points right by either:
    - Passing on the weight to the loss (e.g. weighted Gini index)
    - Sample data with probability = sample weights
        - Misclassified samples are sampled multiple times so they get a higher weight
- Do weighted vote over all models

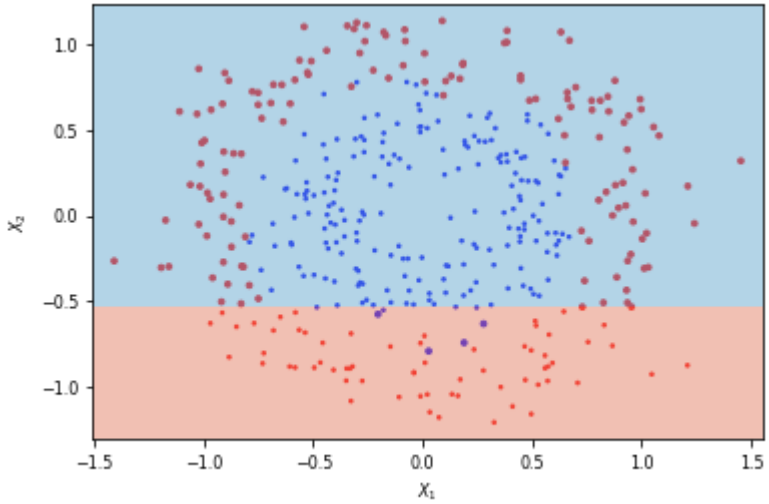
AdaBoost algorithm

- Reset sample weights to $\frac{1}{N}$
- Build a model, using it's own algorithm (e.g. decision stumps with gini index)
- Give it a weight related to its error E

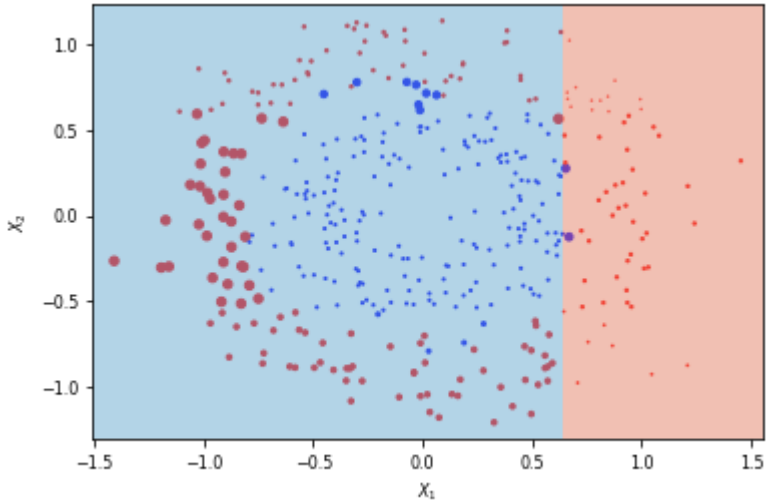
$$w_i = \lambda \log(\frac{1-E}{E})$$

  - Good trees get more weight than bad trees
  - Error is mapped from [0,Inf] to [-1,1], use small minimum error to avoid infinities
  - Learning rate λ (shrinkage) decreases impact of individual classifiers
    - Small updates are often better but requires more iterations
- Update the sample weights
  - Increase weight of incorrectly predicted samples: $s_{n,i+1} = s_{n,i} e^{w_i}$
  - Decrease weight of correctly predicted samples: $s_{n,i+1} = s_{n,i} e^{-w_i}$
  - Normalize weights to add up to 1
- Sample new points according to $s_{n,i+1}$
- Repeat for I rounds

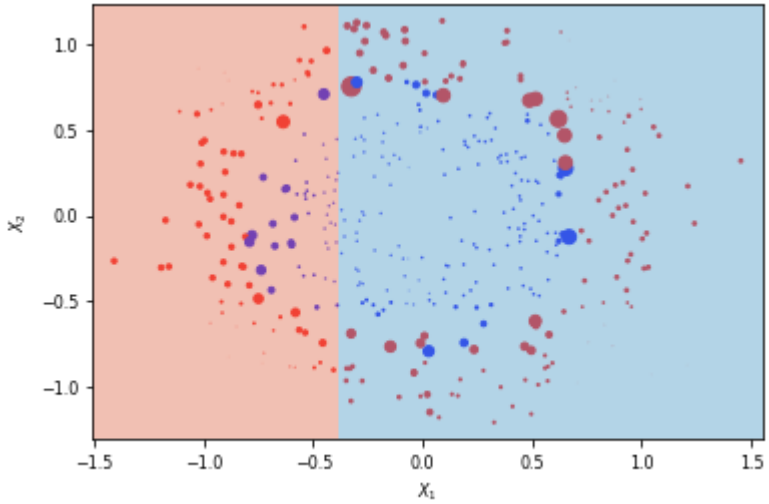
Base model 1, error: 0.35, weight: 0.56



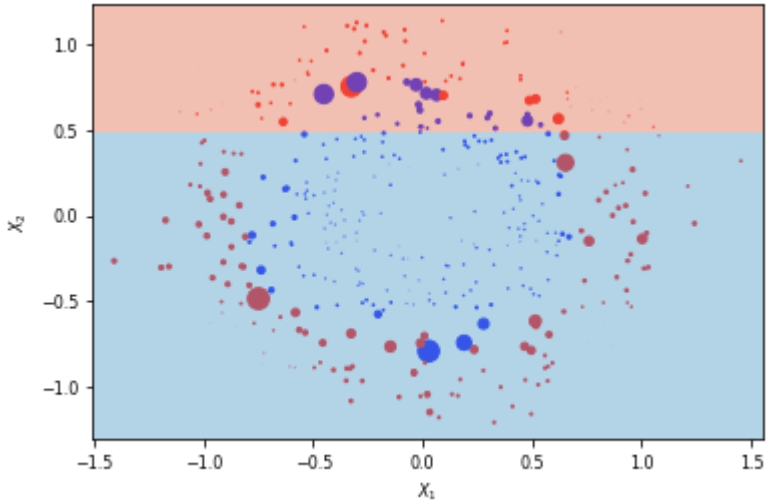
Base model 5, error: 0.21, weight: 1.19

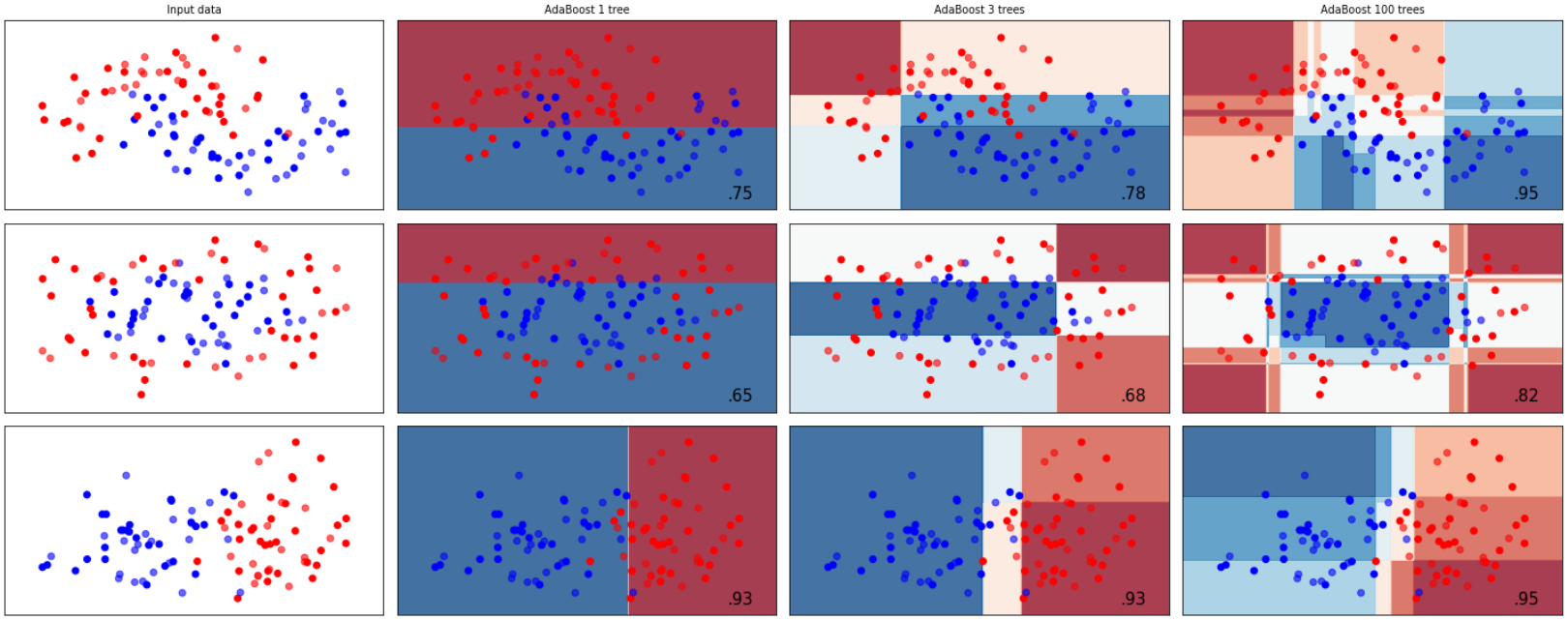


```
Base model 38, error: 0.35, weight: 0.56
```



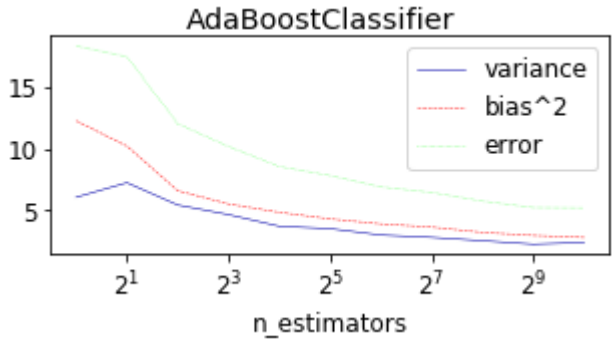

```
Base model 55, error: 0.31, weight: 0.70
```




Input data
AdaBoost 1 tree
AdaBoost 3 trees
AdaBoost 100 trees
.75
.78
.95
.65
.68
.82
.93
.93
.95

AdaBoost reduces bias (and a little variance)

- Boosting too much will eventually increase variance



AdaBoost Recap

- Representation: weighted ensemble of base models
  - Base models can be built by any algorithm
  - Classification: weighted vote over all base models
  - Regression:

$$y = \sum_{i=1}^{N} w_i tree_i(X)$$

- Loss function: weighted loss function of base models
- Optimization: Greedy search

Gradient Boosted Regression Trees (Gradient Boosting Machines)

Several differences to AdaBoost:

- Start with initial guess (e.g. 1 leaf, average value of all samples)
- Base-models are shallow trees (depth 2-4, not stumps)
- Models are weighted (scaled) by same amount (learning rate)
- Subsequent models aim to predict the error of the previous model
    - *Additive model*: final prediction is the sum of all base-model predictions
- Iterate until I trees are built (or error converges)

GradientBoosting Intuition (Regression)

- Do initial prediction M_0 (e.g. average target value)
- Compute the *pseudo-residual* (error) *for every sample* n: $r_n = y_n - y_n^{(M_i)}$
    - Where $y_n^{(M_i)}$ is the prediction for y_n by model M_i

- Build new model M_1 to predict the pseudo-residual of M_0
- New prediction at step I:

$$y_n = y_n^{(M_{i-1})} + \lambda * y_n^{(M_i)} = y_n^{(M_0)} + \sum_{i=1}^{I} \lambda * y_n^{(M_i)}$$

    - λ is the learning rate (or *shrinkage*)
    - Taking small steps in right direction reduces variance (but requires more iterations)
- Compute new pseudo-residuals, and repeat
    - Each step, the pseudo-residuals get smaller
- Stop after given number of iterations, or when the residuals don't decrease anymore (early stopping)

```
gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train)
```

```
Accuracy on training set: 1.000
Accuracy on test set: 0.965
```

```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)
```

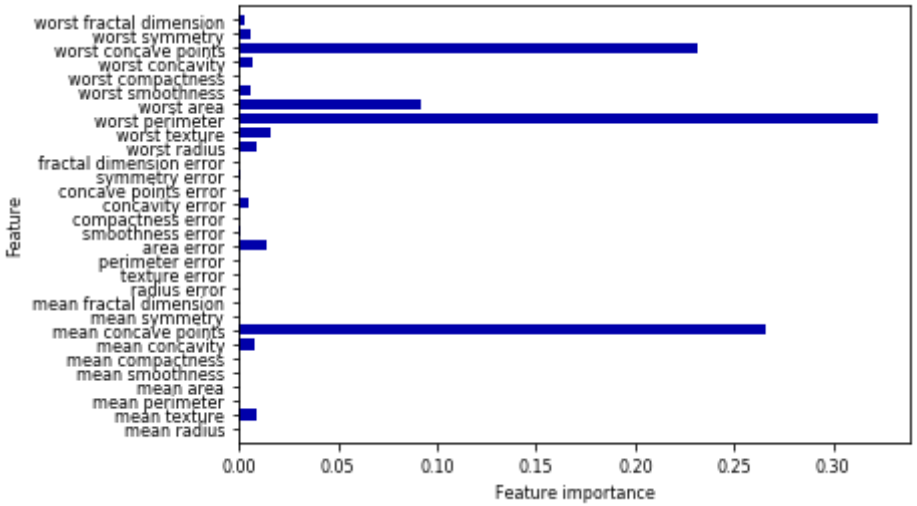
```
Accuracy on training set: 0.991
Accuracy on test set: 0.972
```

```
gbrt = GradientBoostingClassifier(random_state=0, learning_rate=0.01)
gbrt.fit(X_train, y_train)
```

```
Accuracy on training set: 0.988
Accuracy on test set: 0.965
```

Gradient boosting machines use much simpler trees

- Hence, tends to completely ignore some of the features



Strengths, weaknesses and parameters

- Among the most powerful and widely used models
- Work well on heterogeneous features and different scales
- Require careful tuning, take longer to train.
- Does not work well on high-dimensional sparse data

Main hyperparameters:

- `n_estimators`: Higher is better, but will start to overfit
- `learning_rate`: Lower rates mean more trees are needed to get more complex models
    - Set `n_estimators` as high as possible, then tune `learning_rate`
- `max_depth`: typically kept low (<5), reduce when overfitting
- `n_iter_no_change`: early stopping: algorithm stops if improvement is less than a certain tolerance `tol` for more than `n_iter_no_change` iterations.

XGBoost

XGBoost is another python library for gradient boosting
Install separately, `conda install -c conda-forge xgboost`

- The main difference lies the use of approximation techniques to make it faster.
    - About 5x faster *per core*. Thus more boosting iterations in same amount of time
- Sketching: Given 10000 possible splits, it will only consider 300 "good enough" splits by default
    - Controlled by the `sketch_eps` parameter (default 0.03)
- Loss function approximation with Taylor Expansion: more efficient way to evaluate splits
- Allows plotting of the learning curve
- Allows to stop and continue later (warm-start)

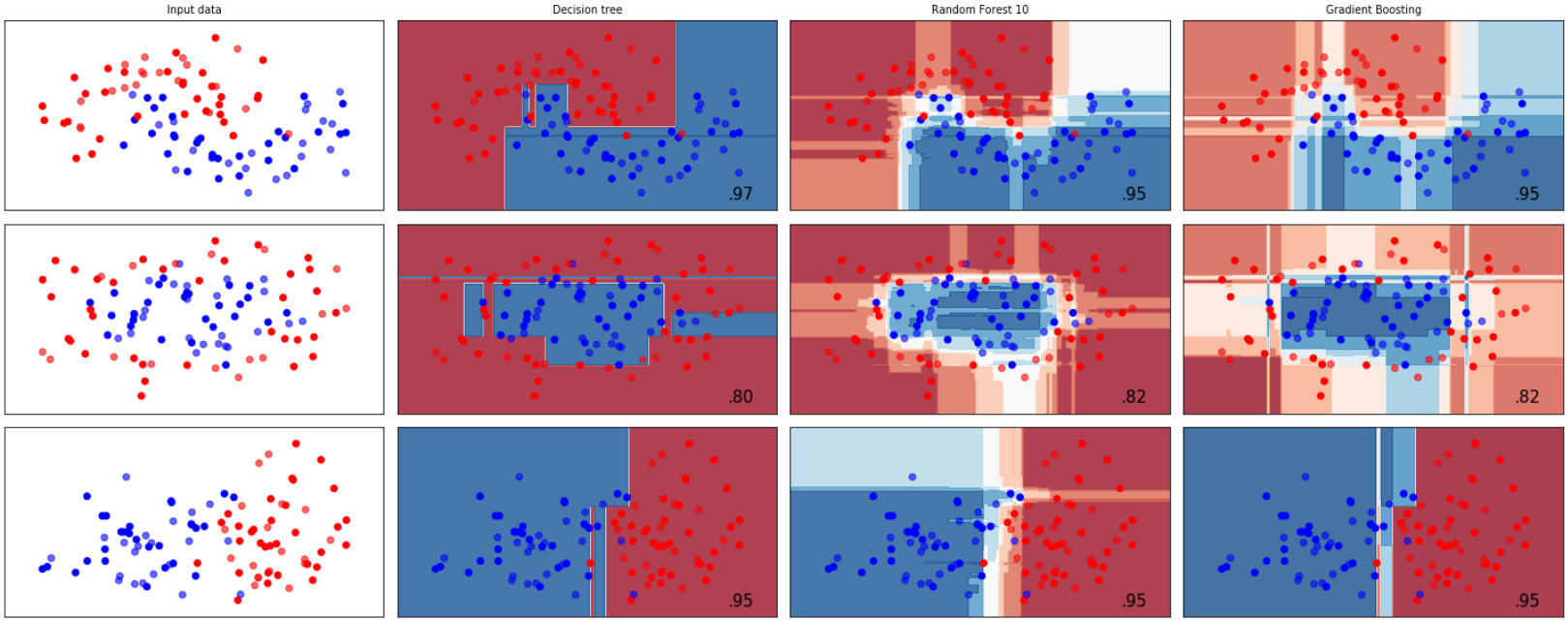
Further reading: XGBoost Documentation (https://xgboost.readthedocs.io/en/latest/parameter.html#parameters-for-tree-booster) Paper (http://arxiv.org/abs/1603.02754)

LightGBM

Another fast boosting technique

- Uses *gradient-based sampling*:
  - use all instances with large gradients (e.g. 10% largest)
  - randomly sample instances with small gradients, ignore the rest
  - intuition: samples with small gradients are already well-trained.
  - requires adapted information gain criterion
- Does smarter encoding of categorical features

Comparison


Input data
Decision tree
Random Forest 10
Gradient Boosting
.97
.95
.95
.80
.82
.82
.95
.95
.95

Algorithm overview

Name	Representation	Loss function	Optimization	Regularization
Classification trees	Decision tree	Information Gain (KL div.) / Gini index	Hunt's algorithm	Tree depth,...
Regression trees	Decision tree	Min. quadratic distance	Hunt's algorithm	Tree depth,...
Bagging	Ensemble of any model	/	/	Number of models,...
RandomForest	Ensemble of random trees	/	/	Number of trees,...
AdaBoost	Ensemble of models (trees)	Weighted loss of base models	Greedy search	Number of trees,...
GradientBoosting	Ensemble of models (trees)	Ensemble loss	Gradient descent	Number of trees,...

Summary

- Bagging / RandomForest is a variance-reduction technique
    - Build many high-variance (overfitting) models
        - Typically deep (randomized) decision trees
        - The more different the models, the better
    - Aggregation (soft voting or averaging) reduces variance
    - Parallellizes easily
- Boosting is a bias-reduction technique
    - Build many high-bias (underfitting) models
        - Typically shallow decision trees
        - Sample weights are updated to create different trees
    - Aggregation (soft voting or averaging) reduces bias
    - Doesn't parallelize easily. Slower to train, much faster to predict.
    - Smaller models, typically more accurate than RandomForests.
- You can build ensembles with other models as well
    - Especially if they show high variance or bias
- It is also possible to build *heterogeneous* ensembles
    - Models from different algorithms
    - Often a meta-classifier is trained on the predictions: Stacking