Nonlinear Models and Metrics

APM Ch 7

APM Ch 13

IMLP 2.3.4

IMPLP 2.3.7

Table of Contents

Neural Networks

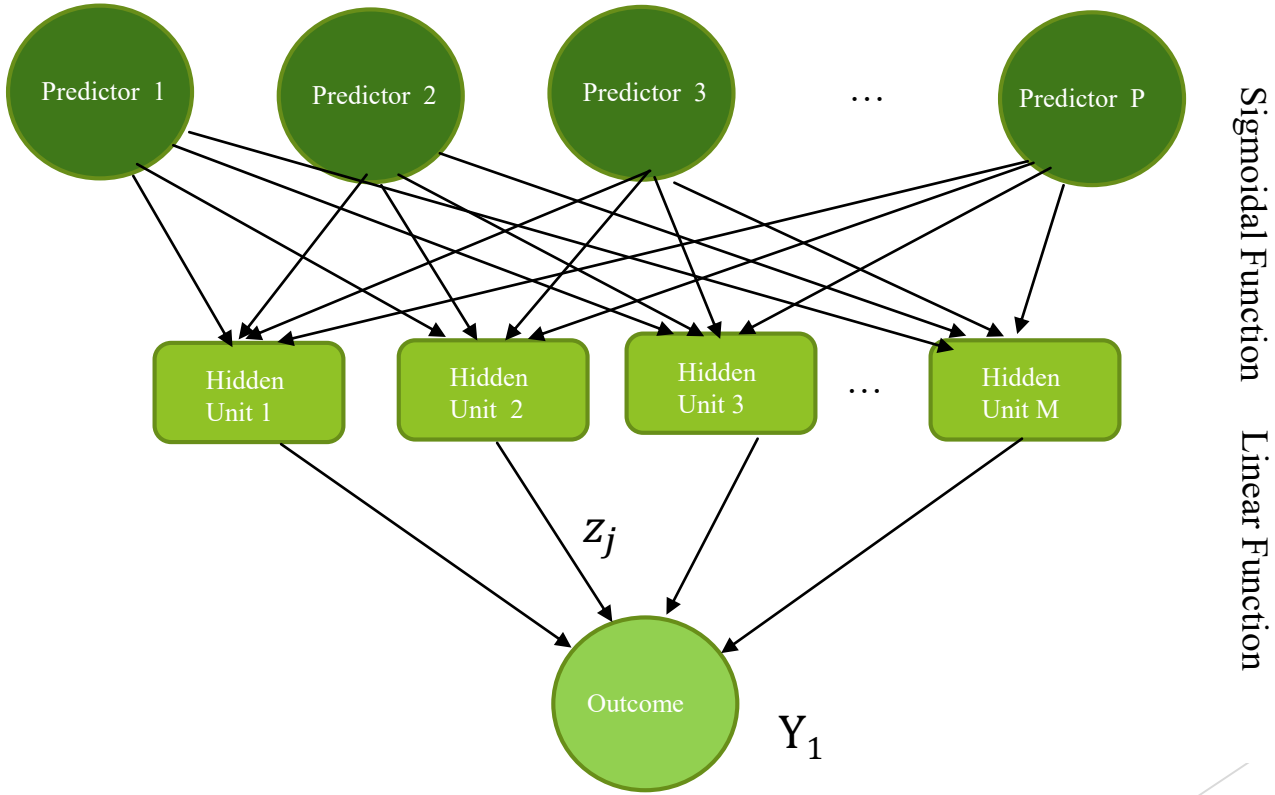
- Neural networks are powerful nonlinear regression techniques inspired by theories about how the brain works.
- outcome is modeled by an intermediary set of unobserved variables (called *hidden variables* or *hidden units* here).
- each hidden unit is a linear combination of some or all of the predictor variables.
- However, this linear combination is typically transformed by a nonlinear function $g(\cdot)$, such as the logistic (i.e., sigmoidal) function

Neural Networks for Regression

$$z_j(x) = \sigma\left(\alpha_{0j} + \sum_{i=1}^{P} x_i \alpha_{ij}\right),\ j \in \{1..M\}$$

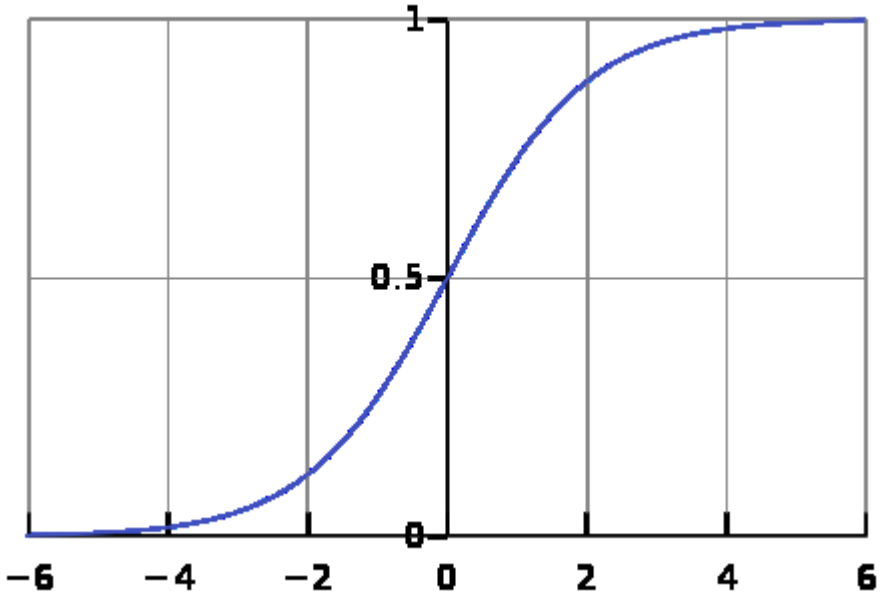
$$\sigma(u) = \frac{1}{1+e^u}$$

$$Y_1 = f(x) = \beta_0 + \sum_{j=1}^{M} \beta_j z_j$$



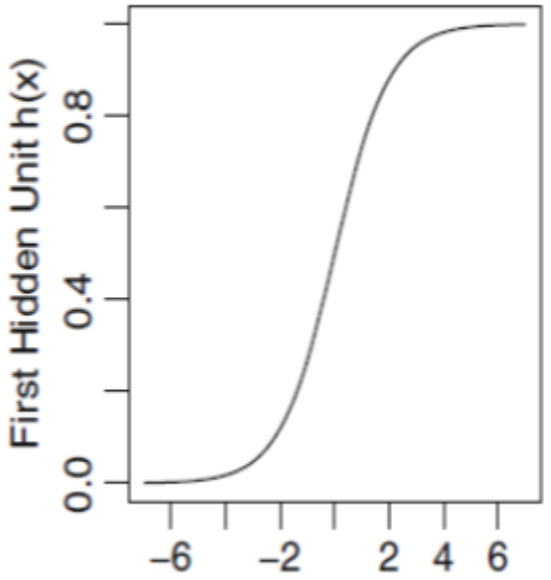
Neural Networks for Regression

- Given a sample, the list of predictors is fed into the input unit, one predictor per one input unit
- All inputs are then sent to each hidden unit. The output each hidden unit is sent to outcome unit which computes Y_1
- Each hidden unit derives “different” aspect of the input to contribute toward final decision
- The sigmodial functions always results in values between 0 to 1 (can be -1 to 1) and adds non-linearity

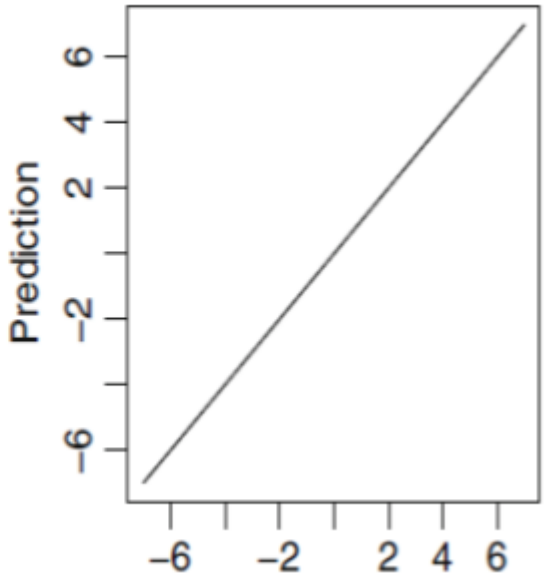


σ - Sigmoid Function

Neural Networks for Regression



$$z_1 = \alpha_{01} + \alpha_{11} x_1 + \alpha_{21} x_2$$



$$Y_1 = \beta_0 + \beta_1 z_1 + \beta_2 z_2$$

Neural Networks for Regression

- The coefficients α are similar to linear regression coefficients; coefficient α_{ij} is the effect of the i^{th} predictor on the j^{th} hidden unit.
- Once the number of hidden units is defined, each unit must be related to the outcome.
- For this type of network model and *P* predictors, and M hidden units, there are a total of *M*(*P* +1)+*M*+1 total parameters being estimated, which quickly becomes large as *P* increases.
- A neural network model with 228 predictors and three hidden units would estimate 3*(229)+4 = 691 parameters while a model with five hidden units would have 1,151 coefficients.

Neural Networks for Regression

- the parameters are usually optimized to minimize the sum of the squared residuals. This can be a challenging numerical optimization problem
- The parameters are usually initialized to random values and then specialized algorithms for solving the equations are used.
- The back-propagation algorithm (Rumelhart et al. 1986) is a highly efficient methodology that works with derivatives to find the optimal parameters.
- However, it is common that a solution to this equation is not a *global* solution, meaning that we cannot guarantee that the resulting set of parameters are uniformly better than any other set.

Neural Networks for Regression

- Also, neural networks have a tendency to over-fit the relationship between the predictors and the response due to the large number of regression coefficients.
- To combat the overfitting, *early stopping technique* is used and it and would stop the optimization procedure when some estimate of the error rate starts to increase.
- Another approach is weight decay indicated here:

$$\sum_{i=1}^{n}(yi - fi(\mathrm{x}))^{2} + \lambda\sum_{j=1}^{M}\sum_{i=0}^{P}\alpha_{ij}{}^{2} + \lambda\sum_{j=0}^{M}\beta_{j}{}^{2}$$

Neural Networks

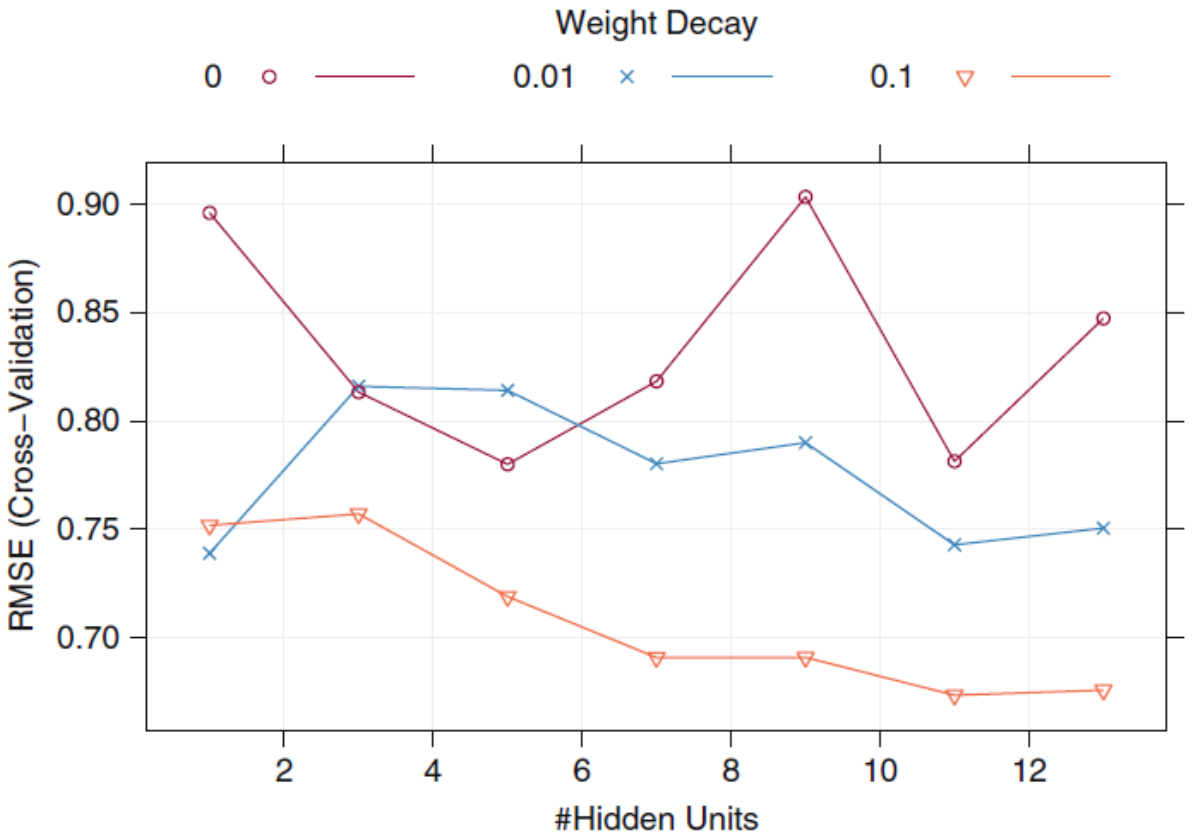
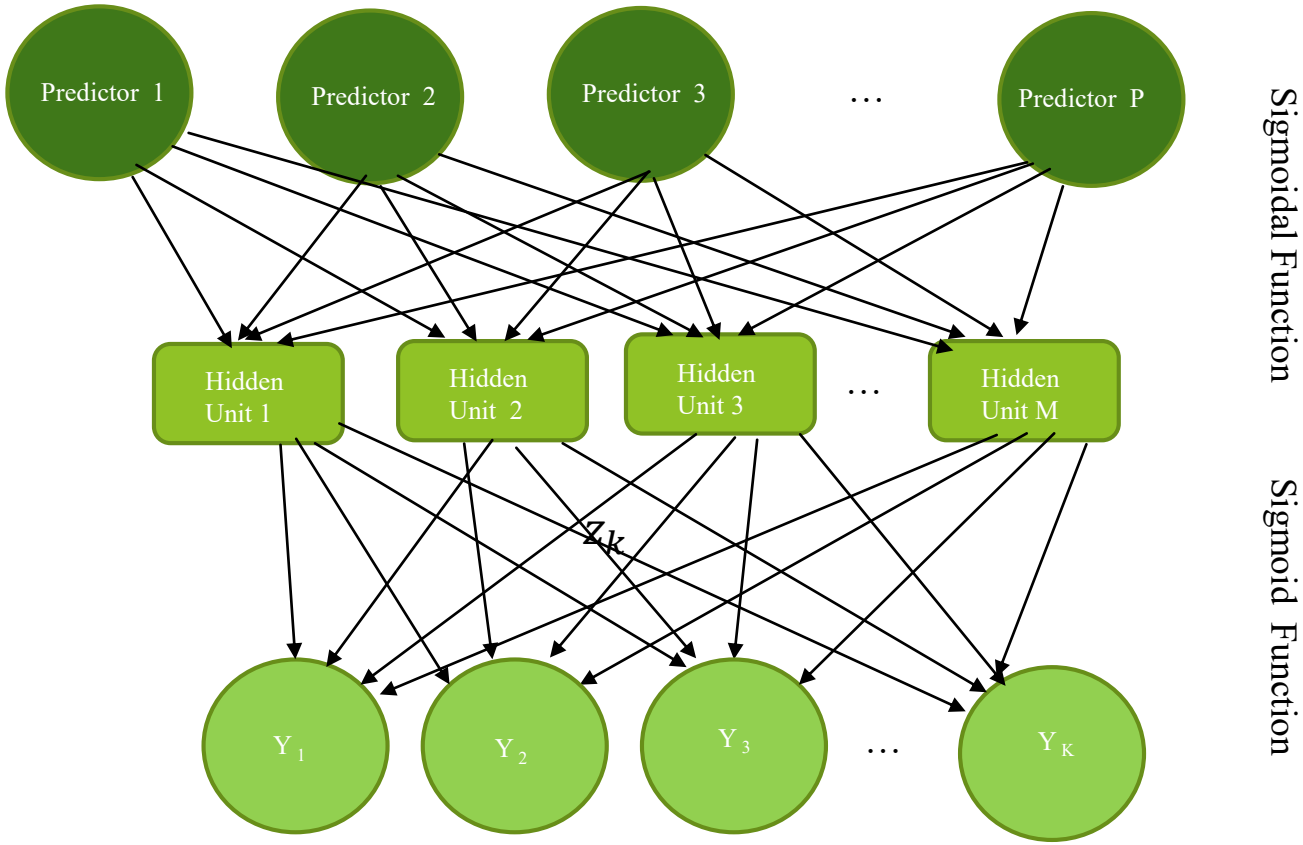


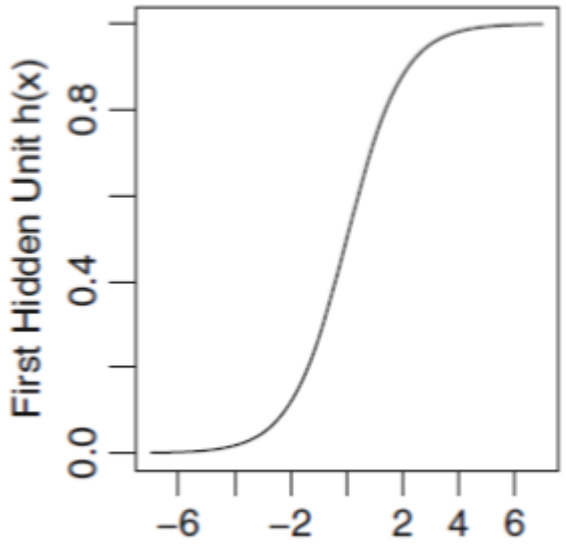
Fig. 7.2: RMSE profiles for the neural network model. The optimal model used $\lambda = 0.1$ and 11 hidden units

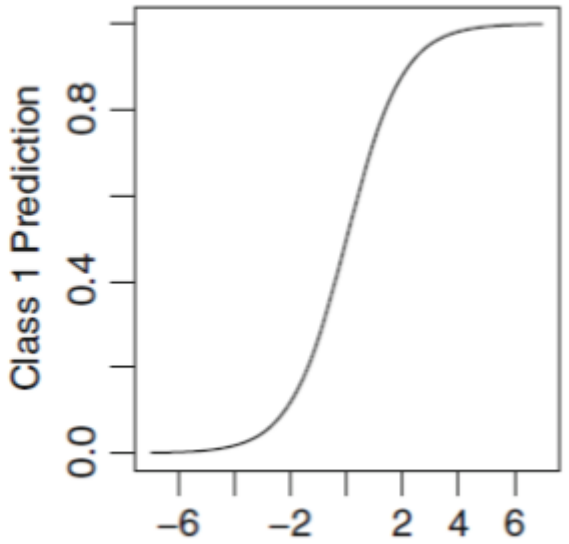
Neural Networks for Classification



- For classification, instead of Y_1, we will have output of Y_k, k = 1..K. for K-class classification. Each Y_k will be the probability that the sample belongs to class k.
- Regression therefore becomes a special case of classification K = 1

Neural Networks for Classification





$$z_1 = \alpha_{01} + \alpha_{11}x_2 + \alpha_{21}x_2$$

$$Y_1 = \beta_0 + \beta_1 z_1 + \beta_2 z_2$$

Forward Propagation

For K-class classification, there are K units at the bottom, with the l^{th} unit modeling the probability of class l. There are K target measurements Y_l, l= 1, . . . ,K, each being coded as probability for the l^{th} class. The vector form of the equations can be written as:

$$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X),\ m = 1, \ldots, M,$$
$$T_k = \beta_{0k} + \beta_k^T Z,\ k = 1, \ldots, K,$$
$$f_k(X) = g_k(T),\ k = 1, \ldots, K,$$

.....11.5

where

Z= $(Z_1, Z_2, Z_3, \ldots Z_M)$,
$\alpha_m = (\alpha_{0m}, \alpha_{1m}, \alpha_{2m}, \alpha_{3m}, \ldots \alpha_{nM})$
$\beta_k = (\beta_{0k}, \beta_{1k}, \beta_{2k}, \beta_{3k}, \ldots \beta_{Mk})$
X = $(x_1, x_2, x_3, x_4, \ldots x_n)$
T = $(T_1, T_2, T_3, \ldots T_K)$

NN Classification

The output function $g_k(T)$ allows a final transformation of the vector of outputs $T = (T_1, T_2, T_3, \ldots T_K)$

.

For K-class classification following softmax function is used which gives a probability between 0 to 1.

$$g_k(T) = \frac{e^{T_k}}{\sum_{\ell=1}^{K} e^{T_\ell}}.$$

For regression we typically choose the identity function $g_k(T) = T_k$.

Forward Propagation

For regression, there is only one T_k, namely T_1 or T , and in that case, g_1 or g is chosen to be identity function

$$Z_m = \sigma(\alpha_{om} + \alpha_m^T X),\ m = 1 \ldots M$$

$$T = \beta_0 + \beta^{T_1} Z$$

$$f(X) = T$$

where

$\alpha_m = (\alpha_{0m}, \alpha_{1m}, \alpha_{2m}, \alpha_{3m}, \ldots \alpha_{nM})$

$\beta = (\beta_{0k}, \beta_{1k}, \beta_{2k}, \beta_{3k}, \ldots \beta_{Mk})$

Back Propagation – Fitting the NN Model

The neural network model has unknown parameters, often called *weights*, and we seek values for them that make the model fit the training data well. We denote the complete set of weights by θ, which consists of

$$\begin{aligned} \{\alpha_{0m}, \alpha_m;\ m = 1, 2, \ldots, M\}\ \ & M(p+1) \text{ weights}, \\ \{\beta_{0k}, \beta_k;\ k = 1, 2, \ldots, K\}\ \ & K(M+1) \text{ weights}. \end{aligned} \tag{11.8}$$

where

$\alpha_m = (\alpha_{0m}, \alpha_{1m}, \alpha_{2m}, \alpha_{3m}, \ldots \alpha_{nM})$

$\beta_k = (\beta_{0k}, \beta_{1k}, \beta_{2k}, \beta_{3k}, \ldots \beta_{Mk})$

For regression, we use sum-of-squared errors as our measure of fit (error function)

$$R(\theta) = \sum_{k=1}^{K} \sum_{i=1}^{N} (y_{ik} - f_k(x_i))^2. \tag{11.9}$$

Back Propagation

For classification we use either squared error or cross-entropy (deviance):

$$R(\theta) = -\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik} \log f_k(x_i), \qquad (11.10)$$

and the corresponding classifier is $G(x) = \text{argmax}_k f_k(x)$. With the softmax activation function and the cross-entropy error function, the neural network model is exactly a linear logistic regression model in the hidden units, and all the parameters are estimated by maximum likelihood.

The generic approach to minimizing R(θ) is by gradient descent, called back-propagation in this setting. Because of the compositional form of the model, the gradient can be easily derived using the chain rule for differentiation. This can be computed by a forward and backward sweep over the network, keeping track only of quantities local to each unit.

Back Propagation

For classification we use either squared error or cross-entropy (deviance):

$$R(\theta) = -\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik} \log f_k(x_i), \qquad (11.10)$$

and the corresponding classifier is $G(x) = \text{argmax}_k f_k(x)$. With the softmax activation function and the cross-entropy error function, the neural network model is exactly a linear logistic regression model in the hidden units, and all the parameters are estimated by maximum likelihood.

The generic approach to minimizing R(θ) is by gradient descent, called back-propagation in this setting. Because of the compositional form of the model, the gradient can be easily derived using the chain rule for differentiation. This can be computed by a forward and backward sweep over the network, keeping track only of quantities local to each unit.

Derivatives

Here is back-propagation in detail for squared error loss. Let $z_{mi} = \sigma(\alpha_{0m} + \alpha_m^T x_i)$, from (11.5) and let $z_i = (z_{1i}, z_{2i}, \ldots, z_{Mi})$. Then we have

$$\begin{aligned} R(\theta) &\equiv \sum_{i=1}^{N} R_i \\ &= \sum_{i=1}^{N}\sum_{k=1}^{K}(y_{ik} - f_k(x_i))^2, \end{aligned} \tag{11.11}$$

with derivatives

$$\begin{aligned} \frac{\partial R_i}{\partial \beta_{km}} &= -2(y_{ik} - f_k(x_i))g_k'(\beta_k^T z_i)z_{mi}, \\ \frac{\partial R_i}{\partial \alpha_{m\ell}} &= -\sum_{k=1}^{K} 2(y_{ik} - f_k(x_i))g_k'(\beta_k^T z_i)\beta_{km}\sigma'(\alpha_m^T x_i)x_{i\ell}. \end{aligned} \tag{11.12}$$

Back Propagation

Given these derivatives, a gradient descent update at the $(r+1)$st iteration has the form

$$\beta_{km}^{(r+1)} = \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \beta_{km}^{(r)}},$$
$$\alpha_{m\ell}^{(r+1)} = \alpha_{m\ell}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \alpha_{m\ell}^{(r)}}, \tag{11.13}$$

Issues with NN

- Starting weights
- Overfitting is common
- Scaling Predictors
- How many hidden units/layers?

Computation

- Each node is called perceptron, so, nueral networks are also called Multilayer Perceptron (MLP).
- In scikit-learn it is called MLP

Computation

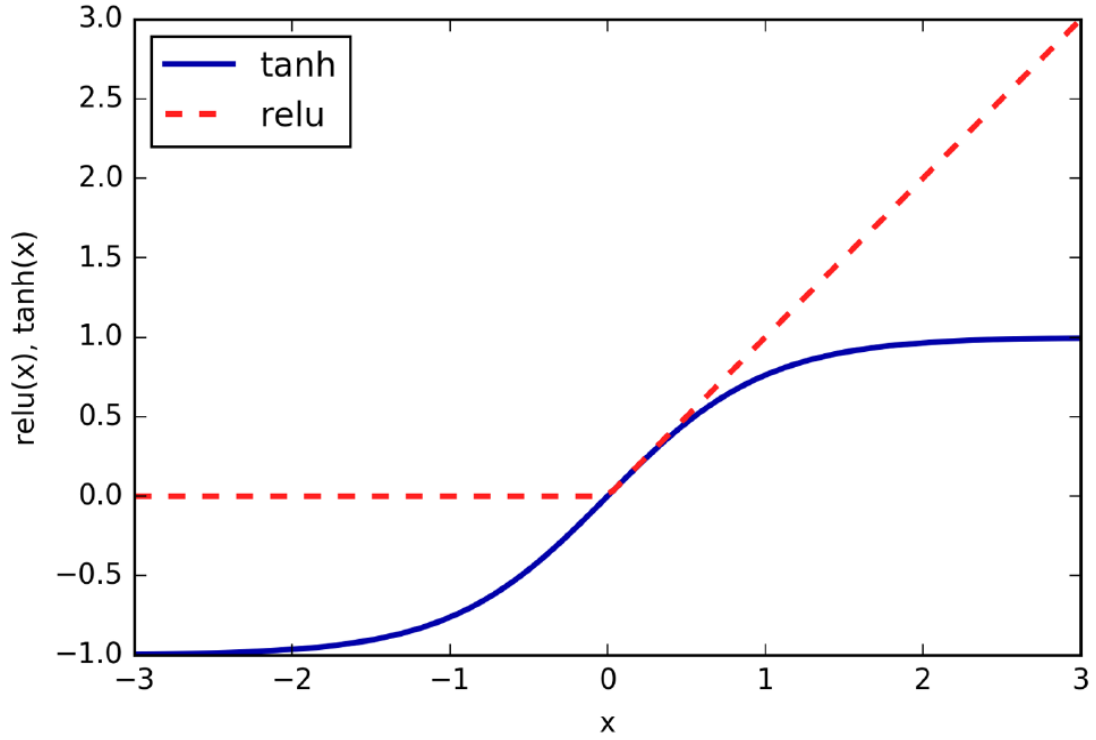


Figure 2-46. The hyperbolic tangent activation function and the rectified linear activation function

Computation

```
class sklearn.neural_network.MLPRegressor(hidden_layer_sizes=100,
activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto',
learning_rate='constant', le
arning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True,
random_state=None, tol=0.0001, verbose=False, warm_start=False,
momentum=0.9, nesterovs_momentum=True, early_stopping=False,
validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08,
n_iter_no_change=10, max_fun=15000)
```

```
class sklearn.neural_network.MLPClassifier(hidden_layer_sizes=100,
activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto',
learning_rate='constant', learning_rate_init=0.001, power_t=0.5,
max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False,
warm_start=False, momentum=0.9, nesterovs_momentum=True,
early_stopping=False, validation_fraction=0.1, beta_1=0.9,
beta_2=0.999, epsilon=1e-08,
n_iter_no_change=10, max_fun=15000)
```

Computation

```
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=100, noise=0.25, random_state=3)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
random_state=42)

mlp = MLPClassifier(solver='lbfgs',random_state=0).fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

Computation-Two Hidden Layers

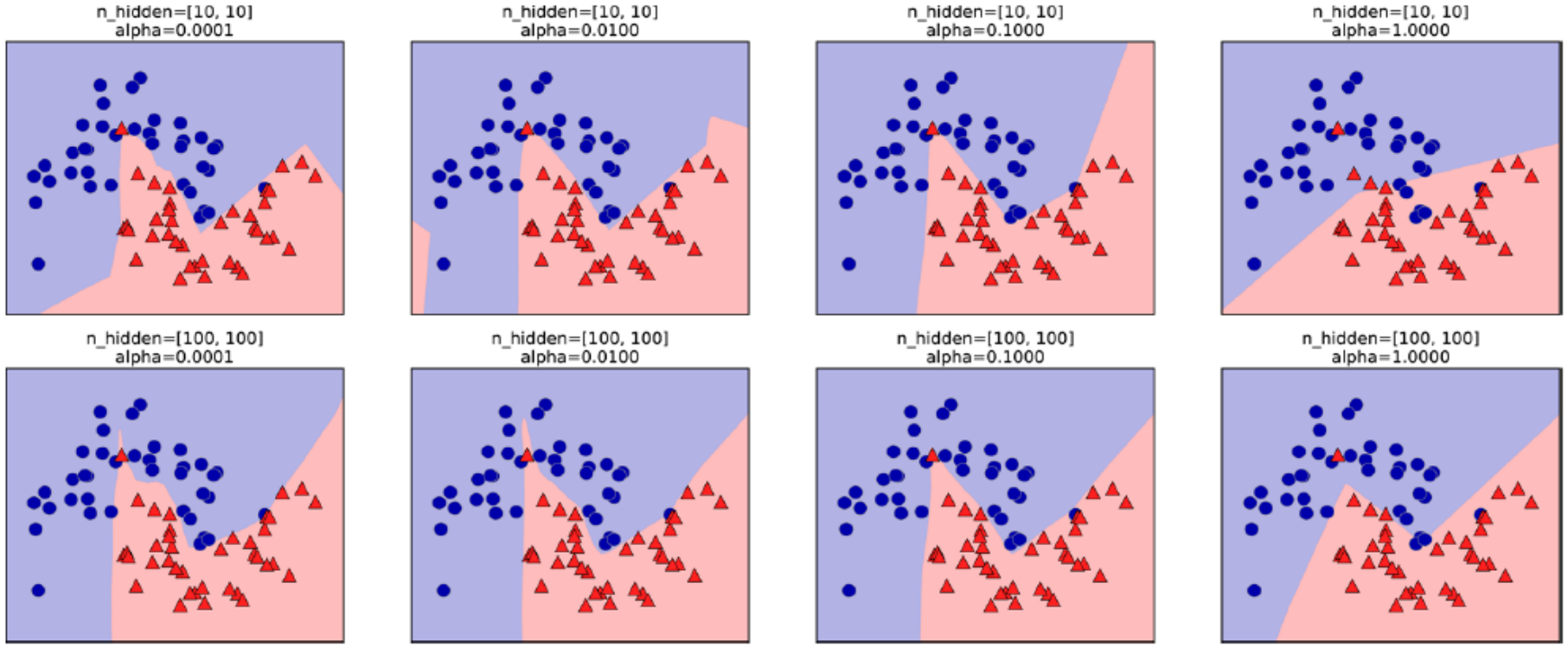


Figure 2-52. Decision functions for different numbers of hidden units and different settings of the alpha parameter

- Two layer NN with 10 nodes in each to 100 hidden nodes in each.

Class Work NN Classification – Feed Forward

- Assume there are 2 predictors P = 2 (x_1, x_2)
- Assume a NN with 2 hidden layers M=2 (z_1, z_2)
- Assume two outputs Y_1, Y_2 (K=2)
- Assume ALL weights of NN is initialized to 0.

$\alpha_1 = (\alpha_{01}, \alpha_{11}, \alpha_{21})$, $\alpha_2 = (\alpha_{02}, \alpha_{12}, \alpha_{22})$

$\beta_1 = (\beta_{01},\beta_{11},\beta_{22})$, $\beta_2 = (\beta_{02},\beta_{12},\beta_{22})$

- Compute the prediction [Y_1, Y_2] of the forward propagation with same (3, 2, [1,0])

Class Work NN Regression- Feed Forward

- Assume there are 2 predictors P = 2 (x_1, x_2)
- Assume a NN with 2 hidden layers M=2 (z_1, z_2)
- Assume ONE output Y_1 (K=1)
- Assume ALL weights of NN is initialized to 0

$\alpha_1 = (\alpha_{01}, \alpha_{11}, \alpha_{21}), \alpha_2 = (\alpha_{02}, \alpha_{12}, \alpha_{22})$

$\beta_1 = (\beta_{01},\beta_{11},\beta_{22}), \beta_2 = (\beta_{02},\beta_{12},\beta_{22})$

- Compute the prediction $[Y_1, Y_2]$ of the forward propagation with same (3, 2, [1,0])

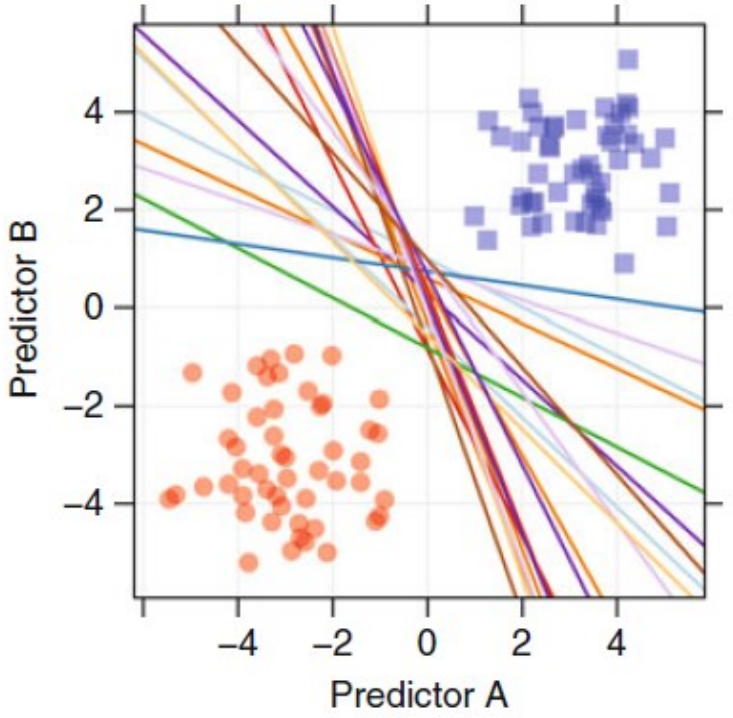
END SESSION

Support Vector Machines

- SVMs are a class of powerful, highly flexible modeling technique
- Initially it was created for classification only in 2010 and later on adopted to regression.
- It is the MOST powerful non linear model for tabular data.

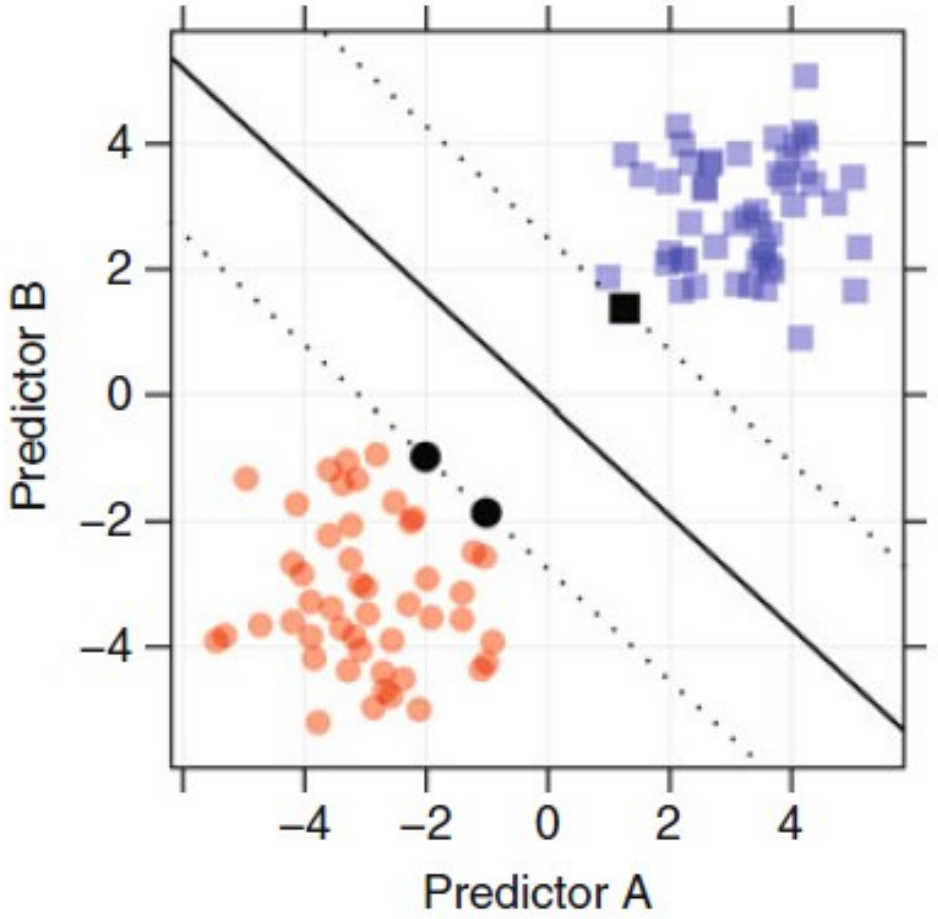
Example

Red – Class 1 and target value is 1 and blue is class 2, target value is -1.



All of the lines “correctly” separate the data. Which is the right answer?

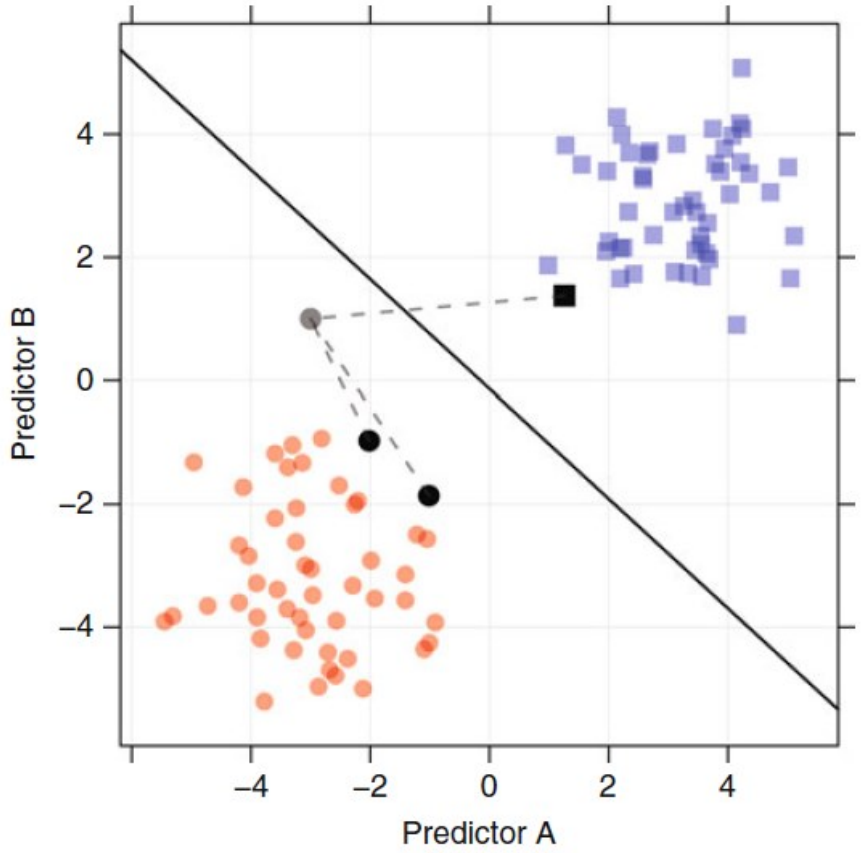
Example



- It will be useful create “support vectors” for each class, so anything to one side of the support vector will belong to a class.
- The points shown in black are called “margins”.
- We would like the support vector to be as far as possible from the linear boundary, so it is called the *maximum margin classifier. Note that support vectors are “equidistant” from the margin.*

SVM for classification

- When classifying a new sample, we would like to find its distance to the “margins”
- *How does one find the margin points in a data set*.



Finding margins for SVM

- The maximum margin classifier creates a decision value $D(\mathbf{u})$ that classifies samples such that if $D(\mathbf{u}) > 0$ we would predict a sample to be class #1, otherwise class #2.
- We are looking for the D(**u**) function.
- However the D(u) function is based on giving weight to predictors, and not to particular sample points.

$$D(\mathbf{u}) = \beta_0 + \boldsymbol{\beta}'\mathbf{u}$$
$$= \beta_0 + \sum_{j=1}^{P} \beta_j u_j.$$

Finding margins for SVM

- We introduce α parameter which gives 'weight' to the sample
- Of course this means we have too many parameters that are α, in fact as many as there are samples!
- Once the optimization is done, we find that α_i 's are all zero, except for those on the margin!

$$D(\mathbf{u}) = \beta_0 + \sum_{j=1}^{P} \beta_j u_j$$

$$= \beta_0 + \sum_{j=1}^{j=P} \sum_{i=1}^{i=n} y_i \alpha_i x_{ij} u_j$$

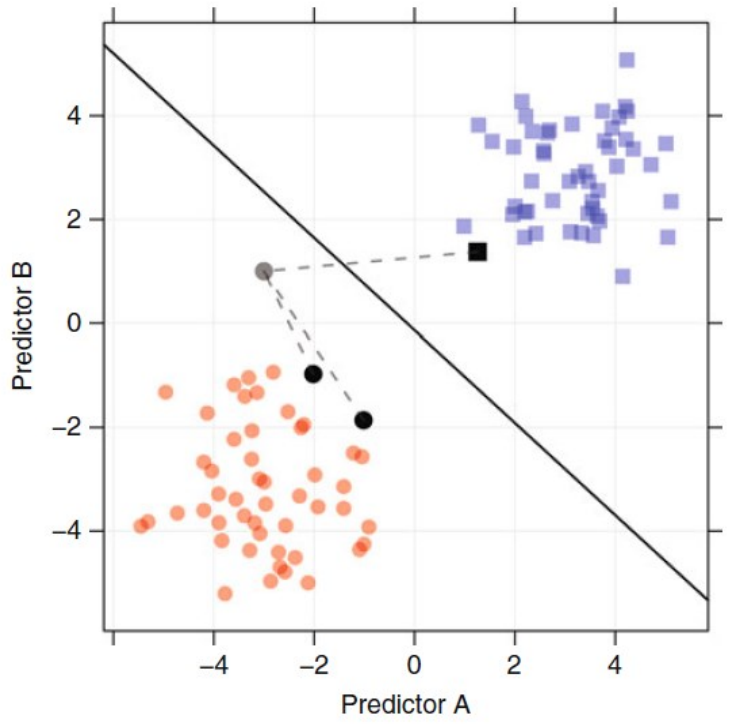
$$= \beta_0 + \sum_{i=1}^{n} y_i \alpha_i \mathbf{x}_i' \mathbf{u}$$

- When a new sample u arrives, first find the dot product of it with each sample, and multiply it with the "weight" for that sample and "outcome of that sample.
- Add the intercept beta_0.
- If the value is > 0 then it is Class 1 otherwise it is Class 2.

Example

$$D(u) = \beta_0 + \sum_{i=1}^{n} y_i \alpha_i \mathbf{x}_i' \mathbf{u}$$

	True class	Dot product	y_i	α_i	Product
SV 1	Class 2	−2.4	−1	1.00	2.40
SV 2	Class 1	5.1	1	0.34	1.72
SV 3	Class 1	1.2	1	0.66	0.79



B_0 is -4.372 and therefore D(u) is 0.583

What if the data is not linearly separable?

We introduce kernel function between x and u, for linear kernel, it is simply the dot product

$$D(\mathbf{u}) = \beta_0 + \sum_{i=1}^{n} y_i \alpha_i \mathbf{x}_i' \mathbf{u}$$

$$= \beta_0 + \sum_{i=1}^{n} y_i \alpha_i K(\mathbf{x}_i, \mathbf{u}),$$

$$\text{polynomial} = (scale\,(\mathbf{x}'\mathbf{u}) + 1)^{degree}$$

$$\text{radial basis function} = \exp(-\sigma\|\mathbf{x} - \mathbf{u}\|^2)$$

$$\text{hyperbolic tangent} = \tanh\,(scale\,(\mathbf{x}'\mathbf{u}) + 1)\,.$$

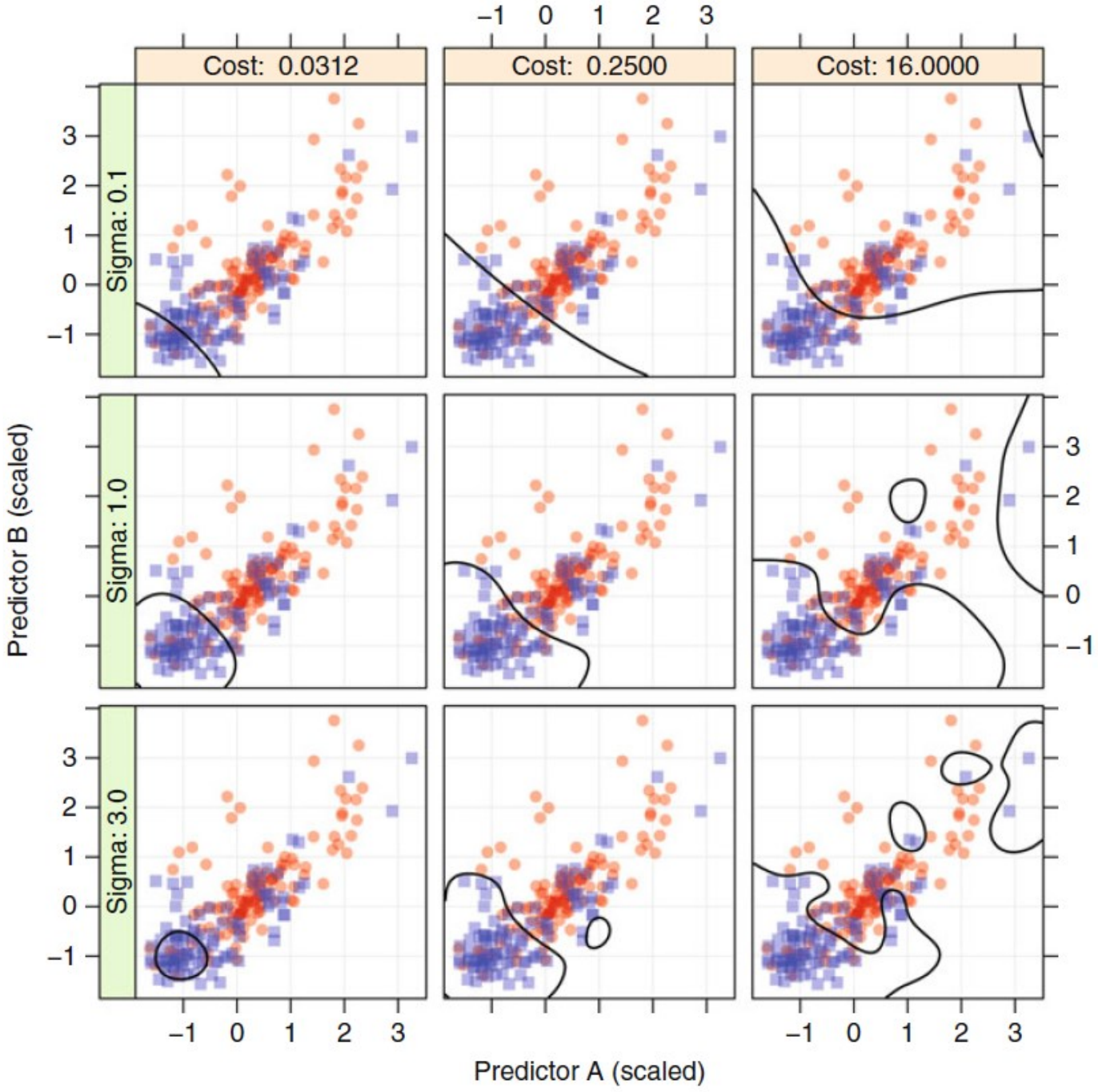
Where *scale*, *degree and* σ are parameters of the kernel.

Kernel Trick

- The *kernel trick* allows the SVM model produce extremely flexible decision boundaries. The choice of the kernel function parameters and the cost value control the complexity and should be tuned appropriately so that the model does not over-fit the training data.

Kernel Trick with Radial Basis Function and different σ

When the cost value is low, the models clearly under-fit the data. When the cost is relatively high (say a value of 16), the model can over-fit the data, especially if the kernel parameter has a large value.



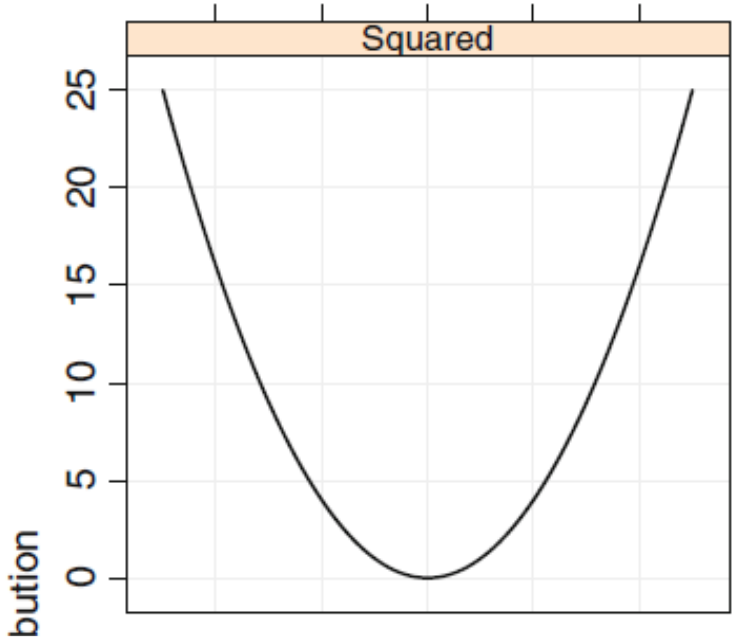
Support Vector Machines for Regression

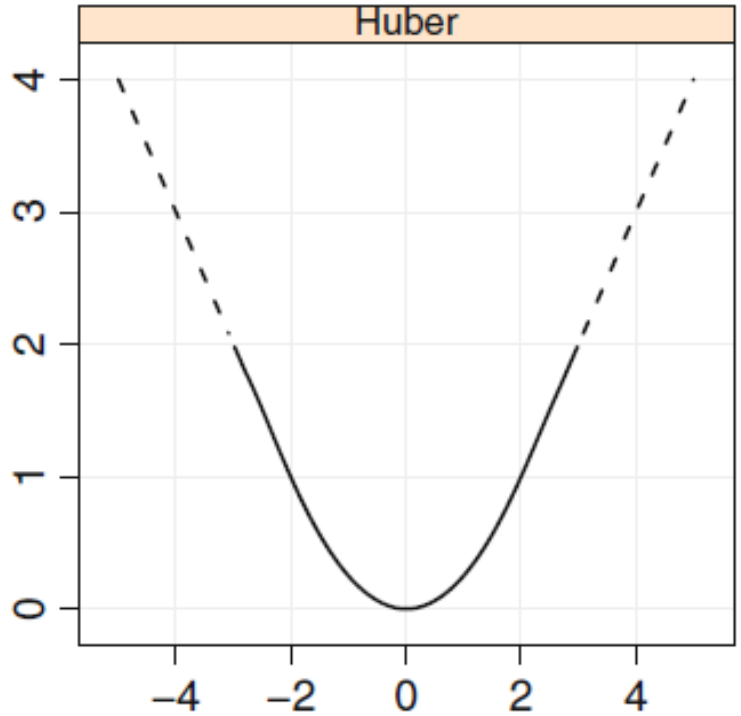
- There are several flavors of support vector regression and we focus on one particular technique called ϵ-*insensitive regression*.
- Recall that linear regression seeks to find parameter estimates that minimize SSE
- One drawback of minimizing SSE is that the parameter estimates can be influenced by just one observation that falls far from the overall trend in the data.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2,$$

When data may contain influential observations, an alternative minimization metric that is less sensitive, such as the Huber function, can be used to find the best parameter estimates. This function uses the squared residuals when they are "small" and uses the absolute residuals when the residuals are large.

Support Vector Machines

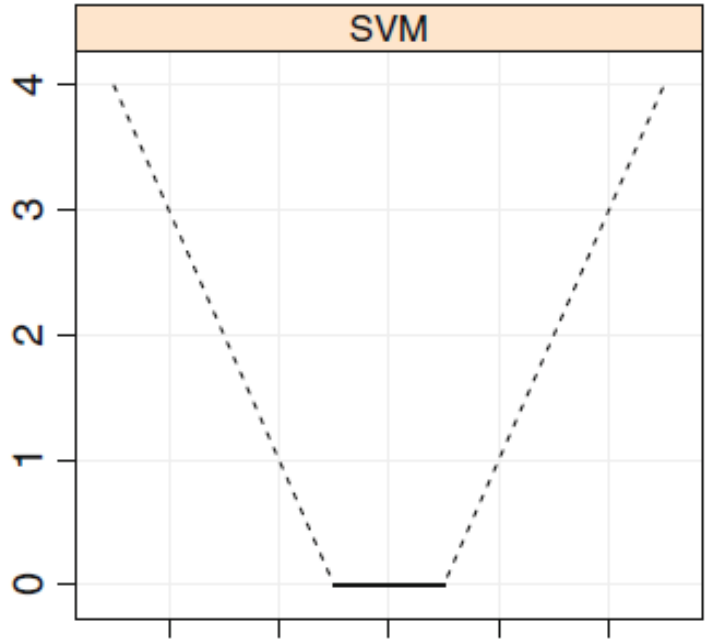




- The Huber function uses the squared residuals when they are “small” and uses the absolute residuals when the residuals are large.

Support Vector Machines

SVMs for regression use a function similar to the Huber function, with an important difference. Given a threshold set by the user (denoted as), data points with residuals within the threshold do not contribute to the regression fit while data points with an absolute difference greater than the threshold contribute a linear-scale amount.



Let us call this function L(.)

Support Vector Machines

- The SVM regression coefficients minimize the following, where $L(\cdot)$ is the epsilon -insensitive function. The *Cost* parameter is the *cost* penalty that is set by the user, which penalizes large residuals.

$$Cost \sum_{i=1}^{n} L_{\epsilon}(y_i - \hat{y}_i) + \sum_{j=1}^{P} \beta_j^2,$$

Support Vector Machines

Recall that the simple linear regression model predicted new samples using linear combinations of the data and parameters. For a new sample, *u*, the prediction equation is

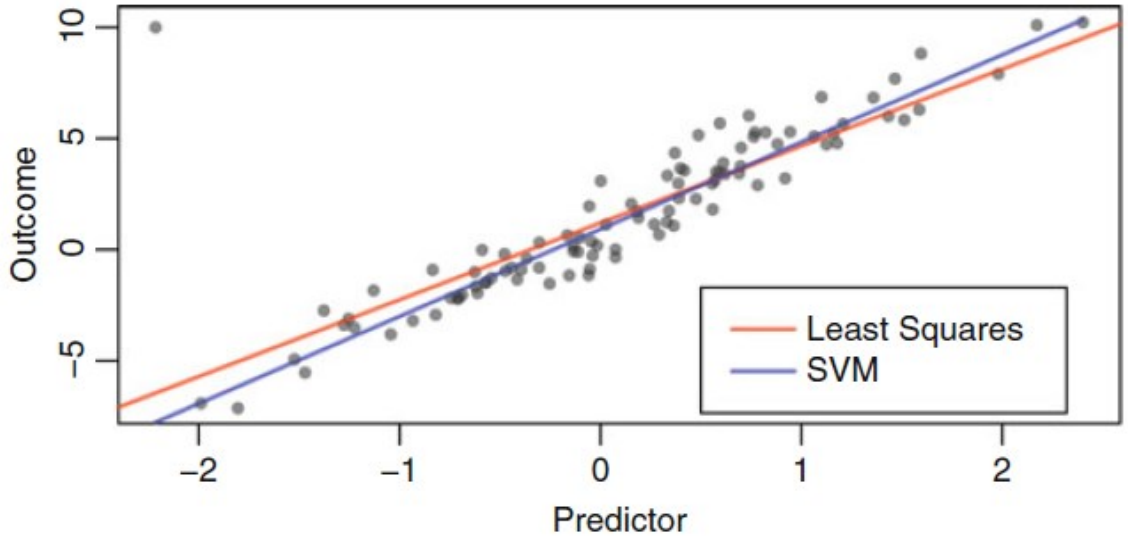
$$\begin{aligned}\hat{y} &= \beta_0 + \beta_1 u_1 + \ldots + \beta_P u_P \\ &= \beta_0 + \sum_{j=1}^{P} \beta_j u_j\end{aligned}$$

Support Vector Machines

$$\begin{aligned}\hat{y} &= \beta_0 + \beta_1 u_1 + \ldots + \beta_P u_P \\ &= \beta_0 + \sum_{j=1}^{P} \beta_j u_j \\ &= \beta_0 + \sum_{j=1}^{P} \sum_{i=1}^{n} \alpha_i x_{ij} u_j \\ &= \beta_0 + \sum_{i=1}^{n} \alpha_i \left(\sum_{j=1}^{P} x_{ij} u_j \right). \end{aligned} \tag{7.2}$$

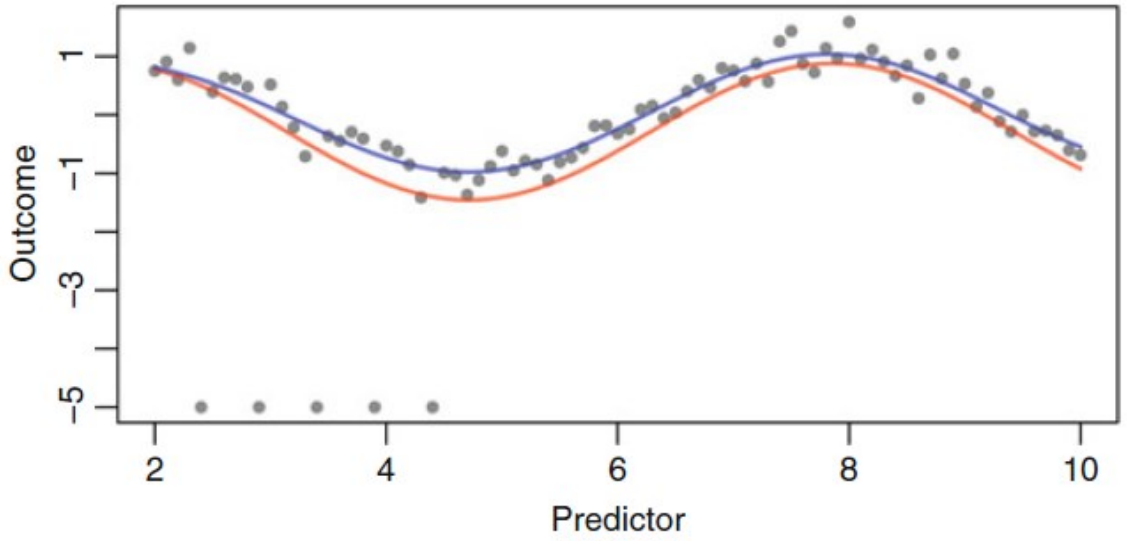
- First, there are as many α parameters as there are data points. From the standpoint of classical regression modeling, this model would be considered *over parameterized*; typically, it is better to estimate fewer parameters than data points.
- However, the use of the cost value effectively regularizes the model to help alleviate this problem. Second, the individual training set data points (i.e., the x_{ij}) are required for new predictions.
- for some percentage of the training set samples, the α_i parameters will be exactly zero, indicating that they have no impact on the prediction equation.

SVM Robustness



A simple linear model was simulated with a slope of 4 and an intercept of 1; one extreme outlier was added to the data. The top panel shows the model fit for a linear regression model (black solid line) and a support vector machine regression model (blue dashed line) with = 0.01. The linear regression line is pulled towards this point, resulting in estimates of the slope and intercept of 3.5 and 1.2, respectively. The support vector regression fit is shown in blue and is much closer to the true regression line with a slope of 3.9 and an intercept of 0.9.

SVM Robustness



To illustrate the ability of this model to adapt to nonlinear relationships, here is simulated data that follow a sin wave. Outliers were also added to these data. Again, the regression line is pulled towards the outlying points. The *red line* is an ordinary regression line (intercept and a term for *sin*(*x*)) and the *blue line* is a radial basis function SVM model

SVM Computation for Classification

class sklearn.svm.SVC(*, *C=1.0*, *kernel='rbf'*, *degree=3*, *gamma='scale'*, *coef0=0.0*, *shrinking=True*, *probability=False*, *tol=0.001*, *cache_size=200*, *class_weight=None*, *verbose=False*, *max_iter=- 1*, *decision_function_shape='ovr'*, *break_ties=False*, *random_state=None*)

class sklearn.svm.LinearSVC(*penalty='l2'*, *loss='squared_hinge'*, *, *dual=True*, *tol=0.0001*, *C=1.0*, *multi_class='ovr'*, *fit_intercept=True*, *intercept_scaling=1*, *class_weight=None*, *verbose=0*, *random_state=None*, *max_iter=1000*)

```
linear_svc = svm.SVC(kernel='linear')
linear_svc.kernel 'linear'
rbf_svc = svm.SVC(kernel='rbf')
rbf_svc.kernel 'rbf'
```

SVM Computation for Classification

The *kernel function* can be any of the following:

- linear: $\langle x, x' \rangle$.
- polynomial: $(\gamma \langle x, x' \rangle + r)^d$, where d is specified by parameter `degree`, r by `coef0`.
- rbf: $\exp(-\gamma \|x - x'\|^2)$, where γ is specified by parameter `gamma`, must be greater than 0.
- sigmoid $\tanh(\gamma \langle x, x' \rangle + r)$, where r is specified by `coef0`.

Different kernels are specified by the `kernel` parameter:

SVM Computation for Regression

There are two different implementations of Support Vector Regression: SVR, and LinearSVR provides a faster implementation

class sklearn.svm.LinearSVR(*, *epsilon=0.0*, *tol=0.0001*, *C=1.0*, *loss='epsilon_insensitive'*, *fit_intercept=True*, *intercept_scaling=1.0*, *dual=True*, *verbose=0*, *random_state=None*, *max_iter=1000*)[source]¶

class sklearn.svm.SVR(*, *kernel='rbf'*, *degree=3*, *gamma='scale'*, *coef0=0.0*, *tol=0.001*, *C=1.0*, *epsilon=0.1*, *shrinking=True*, *cache_size=200*, *verbose=False*, *max_iter=- 1*)

```
from sklearn import svm
X = [[0, 0], [2, 2]]
y = [0.5, 2.5]
regr = svm.SVR()
regr.fit(X, y)
regr.predict([[1, 1]])
```

```
array([1.5])
```

SVM Parameter Tuning

- Which kernel function should be used? This depends on the problem. The radial basis function has been shown to be very effective. However, when the regression line is truly linear, the linear kernel function will be a better choice.
- When training an SVM with the *Radial Basis Function* (RBF) kernel, two parameters must be considered: C and gamma.
- The parameter C, common to all SVM kernels, trades off misclassification of training examples against simplicity of the decision surface. A low C makes the decision surface smooth, while a high C aims at classifying all training examples correctly.
- gamma defines how much influence a single training example has. The larger gamma is, the closer other examples must be to be affected.
- Proper choice of C and gamma is critical to the SVM's performance.

Strengths and Weaknesses

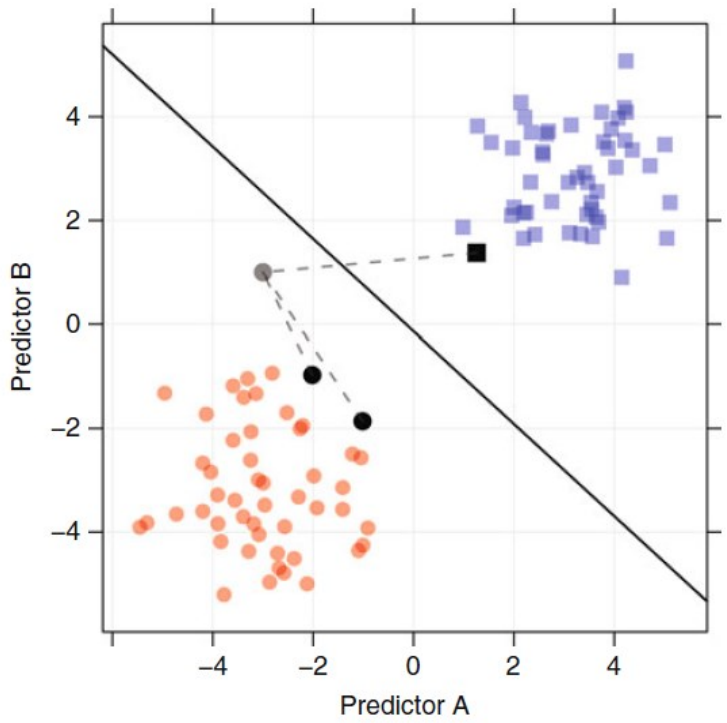
- Kernelized support vector machines are powerful models and perform well on a variety of datasets.
- SVMs allow for complex decision boundaries, even if the data has only a few features.
- They work well on low-dimensional and high-dimensional data (i.e., few and many features, about 10,000 samples is good), but don't scale very well with the number of samples.
- Another downside of SVMs is that they require careful preprocessing of the data and tuning of the parameters
- most people instead use tree-based models such as random forests or gradient boosting (which require little or no preprocessing) in many applications.

Strengths and Weaknesses

- SVM models are hard to inspect; it can be difficult to understand why a particular prediction was made, and it might be tricky to explain the model to a non-expert.
- The important parameters in kernel SVMs are the regularization parameter C, the choice of the kernel, and the kernel-specific parameters.
- we primarily focused on the RBF kernel, other choices are available in scikit-learn
- The RBF kernel has only one parameter, gamma, which is the inverse of the width of the Gaussian kernel. gamma and C both control the complexity of the model, with large values in either resulting in a more complex model. Therefore, good settings for the two parameters are usually strongly correlated, and C and gamma should be adjusted together.

Class work -Classification

True Class	y_i	α_i
Class 2	-1	1.00
Class 1	1	0.34
Class 1	1	0.66



Find the value of D(u) for the points
u = (-4, -2)
u = ((4,0))
And classify them.
Assume that margin points SV1, SV2, SV3 are
SV1 (2,2), SV2 (-1,-2) and SV3 (-2, -1)
B_0 is -4.372

kNN – k-Nearest Neighbor

- kNN is a non-linear regression model that was covered earlier.