Training Models: Polynomial Regression

Hands-on Machine Learning: Chapter 4

Polynomial Regression

Train linear model on extended set of features.

Add powers of each feature as new features.

Polynomial Regression

Train linear model on extended set of features.

Add powers of each feature as new features.

Simulate quadratic $y = ½ x^2 + x + 2 + \text{noise}$

```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

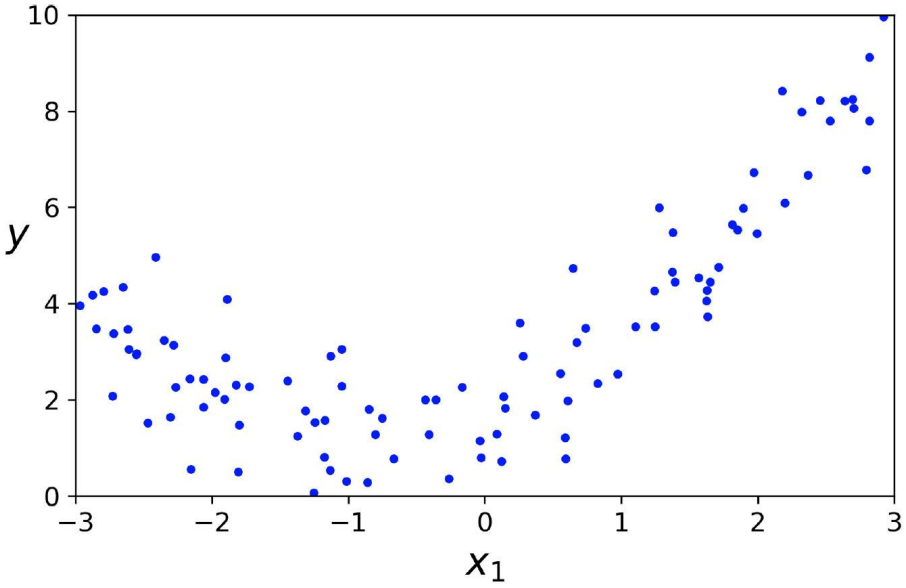


Figure 4-12. Generated nonlinear and noisy dataset

Polynomial Regression

Train linear model on extended set of features.

Add powers of each feature as new features.

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929, 0.56664654])
```

Features a & b with degree=3. Adds features a^2, a^3, b^2, and b^3, but also combinations ab, a^2b, ab^2.

Beware of combinatorial explosion of number of features + degrees factorial!

Polynomial Regression

$y = \frac{1}{2} x^2 + 1x + 2 + \text{noise}$

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.coef_
array([[0.93366893, 0.56456263]])
>>> lin_reg.intercept_
(array([1.78134581])
```

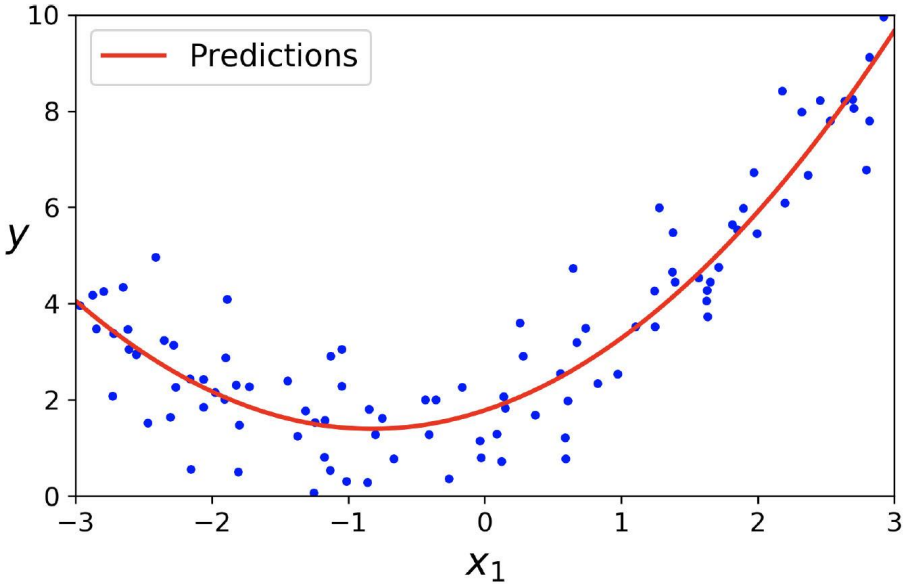


Figure 4-13. Polynomial Regression model predictions

Polynomial Regression

Compare 1, 2, 300-degree polynomials

Overfitting
performs well on training data but
generalizes poorly on cross-validation

Underfitting
performs poorly on training and cross-validation

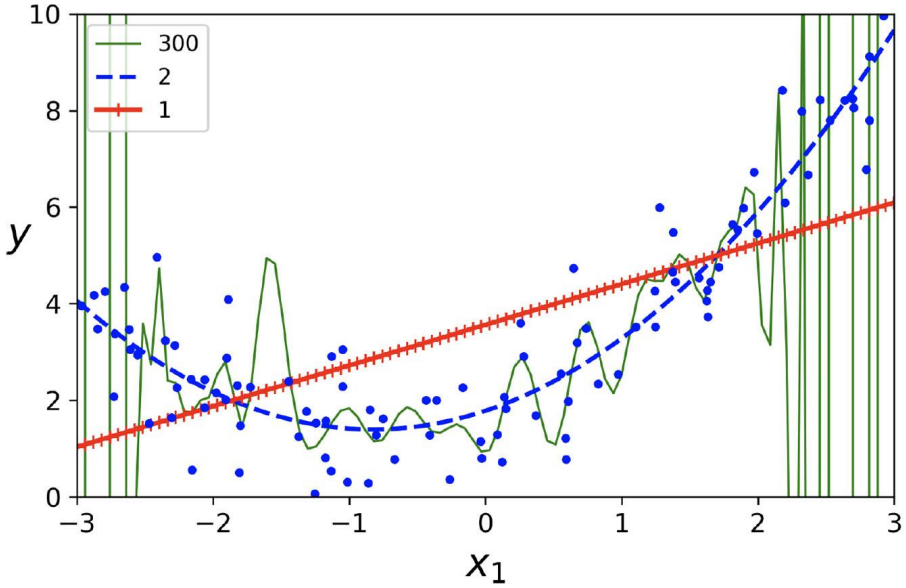


Figure 4-14. High-degree Polynomial Regression

Learning Curves

Plot training set size (or training iteration) against model's performance training & validation errors

```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
```

If underfitting, adding more data will not help. Need a more complex model or better features.

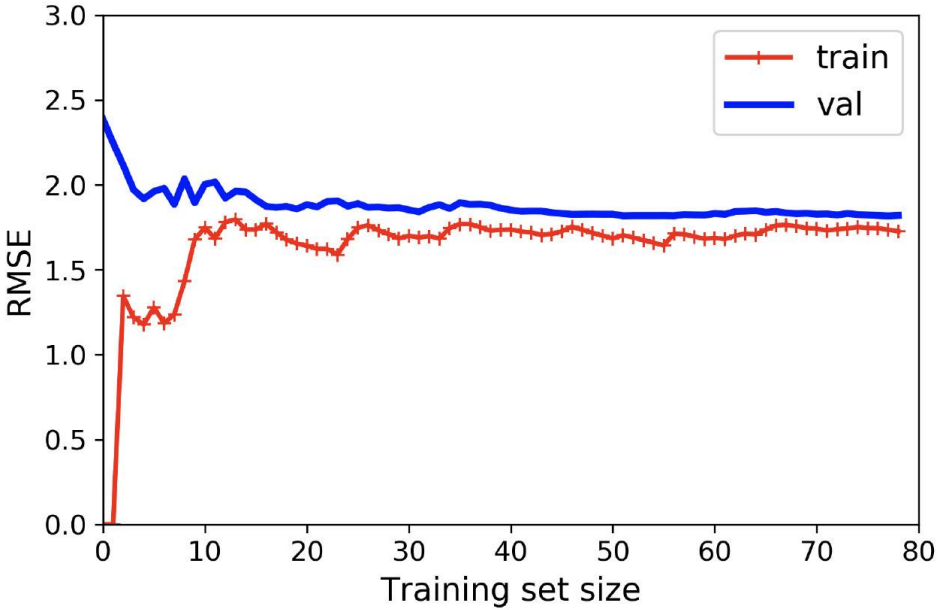


Figure 4-15. Learning curves

Learning Curves

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
  X_train, X_val, y_train, y_val = train_test_split(
        X, y, test_size=0.2)
  train_errors, val_errors = [], []
  for m in range(1, len(X_train)):
    model.fit(X_train[:m], y_train[:m])
    y_train_predict = model.predict(X_train[:m])
    y_val_predict = model.predict(X_val)
    train_errors.append(mean_squared_error(
        y_train[:m], y_train_predict))
    val_errors.append(mean_squared_error(
        y_val, y_val_predict))
  plt.plot(np.sqrt(train_errors),
        "r-+", linewidth=2, label="train")
  plt.plot(np.sqrt(val_errors),
        "b-", linewidth=3, label="val")
```

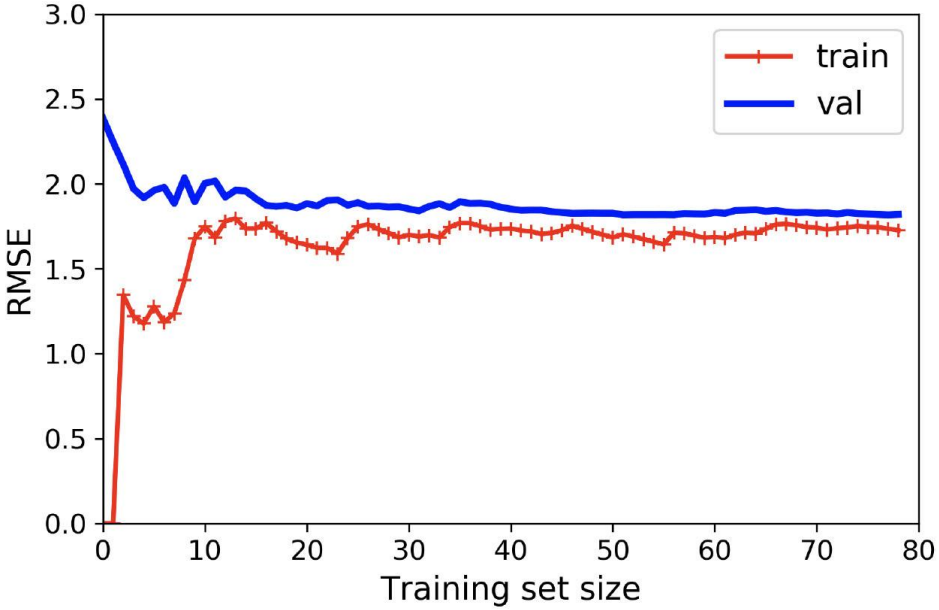


Figure 4-15. Learning curves

Learning Curves

```
from sklearn.pipeline import Pipeline
polynomial_regression = Pipeline([
    ("poly_features", PolynomialFeatures(degree=10,
                      include_bias=False)),
    ("lin_reg", LinearRegression()),
  ])
plot_learning_curves(polynomial_regression, X, y)
```

Error on training data much lower.
Gap between curves shows overfitting.
User larger training set until train = val errors.

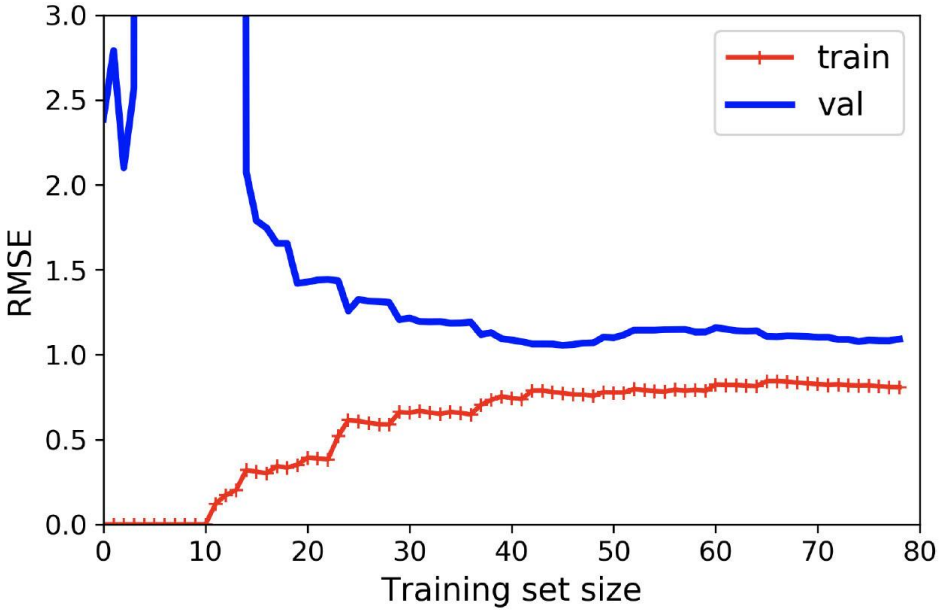


Figure 4-16. Learning curves for the polynomial model

Bias/Variance Tradeoff

Model's generalization error is sum of 3 different types of errors:

Bias
Wrong assumptions, such as assuming that the data is linear when it is actually quadratic
A *high-bias* model is most likely to *underfit* the training data.

Variance
Excessive sensitivity to small variations in the training data.
Many degrees of freedom (high-degree polynomial) likely to have
high variance and *overfit* the training data.

Irreducible error
Noisiness of the data itself.
Clean up the data (fix data sources, broken sensors, detect/remove outliers)

Training Models: Regularization

Hands-on Machine Learning: Chapter 4

Regularized Linear Models

Reduce variance or overfitting by constraining it

Polynomial models fewer degrees of freedom.

Linear Models constrain weights:

- **Ridge Regression**
- **Lasso Regression**
- **Elastic Net**

Early Stopping

Ridge Regression

Tikhonov regularization

Penalty ½ (ℓ_2 norm of $\mathbf{w}$)^2 aka Euclidean norm

$\|\mathbf{w}\|_2 = \sqrt{\sum \mathbf{w}^2}$

$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$

Ridge Regression

Regularization hyperparameter **α**
α = 0 same as unregularized
α large, weights close to 0, flat line through data mean

Don't add bias term θ_0

Only regularize during **training** not when evaluating performance

Important to **scale** input features before performing regularization

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$

Ridge Regression

Closed-form with Cholesky matrix factorization

```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=1, solver="cholesky")
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
array([[1.55071465]])
```

Stochastic Gradient Descent, can spec alpha

```
>>> sgd_reg = SGDRegressor(penalty="12")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([1.47012588])
```

$$\hat{\theta} = \left(\mathbf{X}^T\mathbf{X} + \alpha\mathbf{A}\right)^{-1} \quad \mathbf{X}^T \quad \mathbf{y}$$

Ridge Regression

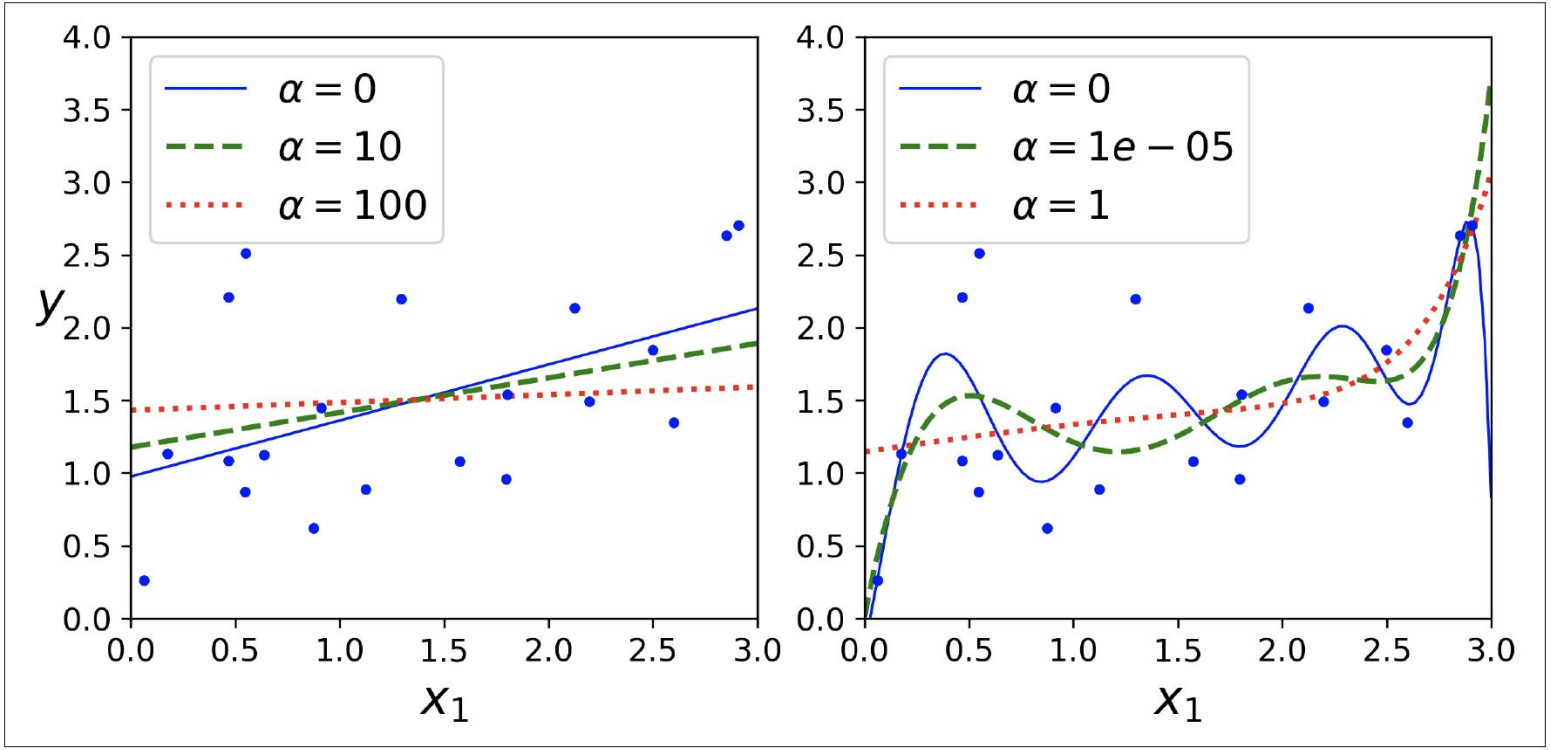


Figure 4-17. Ridge Regression

Lasso Regression

Least Absolute Shrinkage and Selection Operator Regression

Penalty ℓ_1 norm aka Manhattan distance

$\|\mathbf{w}\|_1 = \sum |\mathbf{w}|$

Automatic feature selection. Sparse model with few nonzero feature weights

$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + \alpha \sum_{i=1}^{n} |\theta_i|$$

Lasso Regression

```
>>> from sklearn.linear_model import Lasso
>>> lasso_reg = Lasso(alpha=0.1)
>>> lasso_reg.fit(X, y)
>>> lasso_reg.predict([[1.5]])
array([1.53788174])
```

```
>>> sgd_reg = SGDRegressor(penalty="l1", alpha=0.1)
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([1.4656962])
```

$$g(\boldsymbol{\theta}, J) = \nabla_{\boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta}) + \alpha \begin{pmatrix} \operatorname{sign}(\theta_1) \\ \operatorname{sign}(\theta_2) \\ \vdots \\ \operatorname{sign}(\theta_n) \end{pmatrix} \quad \text{where} \quad \operatorname{sign}(\theta_i) = \begin{cases} -1 & \text{if } \theta_i < 0 \\ 0 & \text{if } \theta_i = 0 \\ +1 & \text{if } \theta_i > 0 \end{cases}$$

Lasso Regression

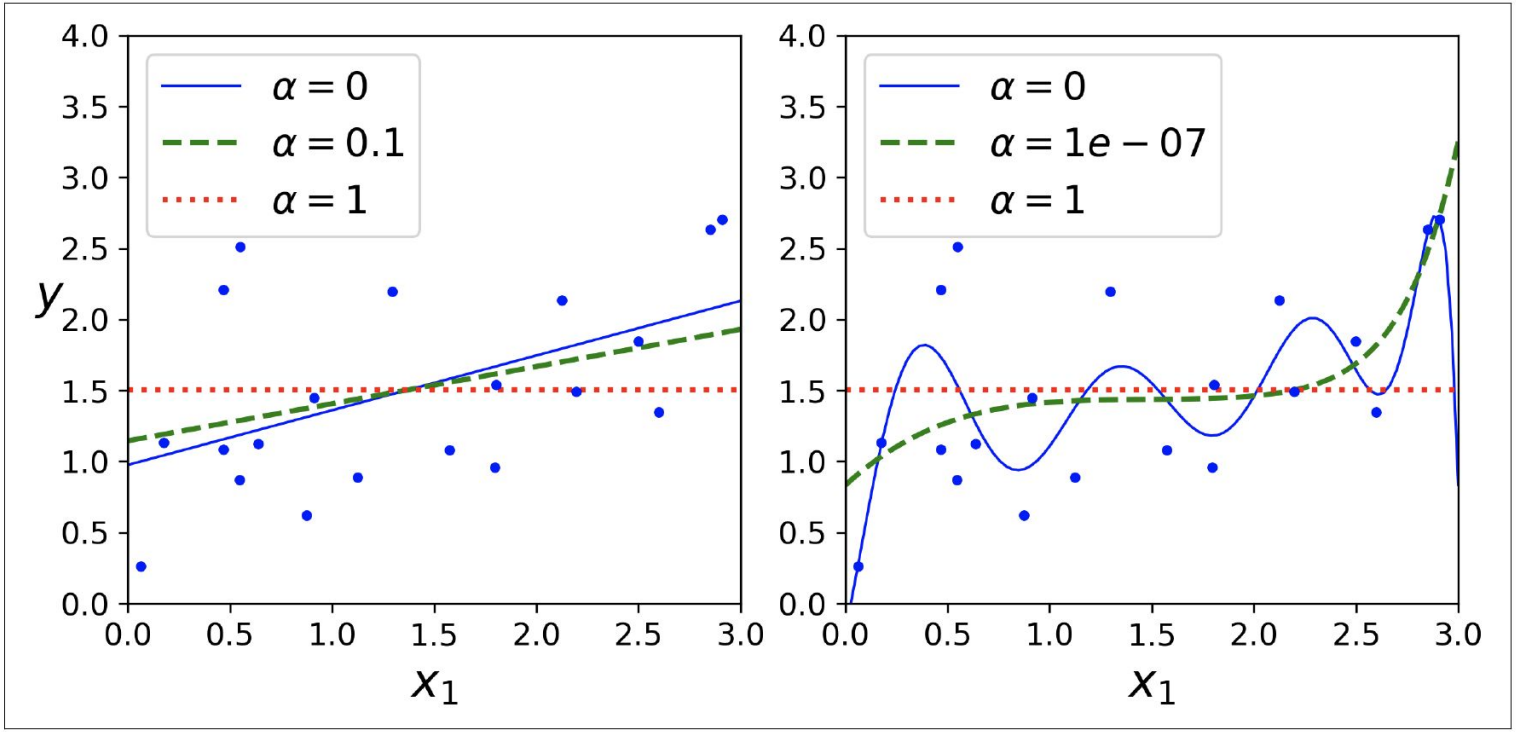


Figure 4-18. Lasso Regression

Ridge vs Lasso Regression

Contours penalty & cost

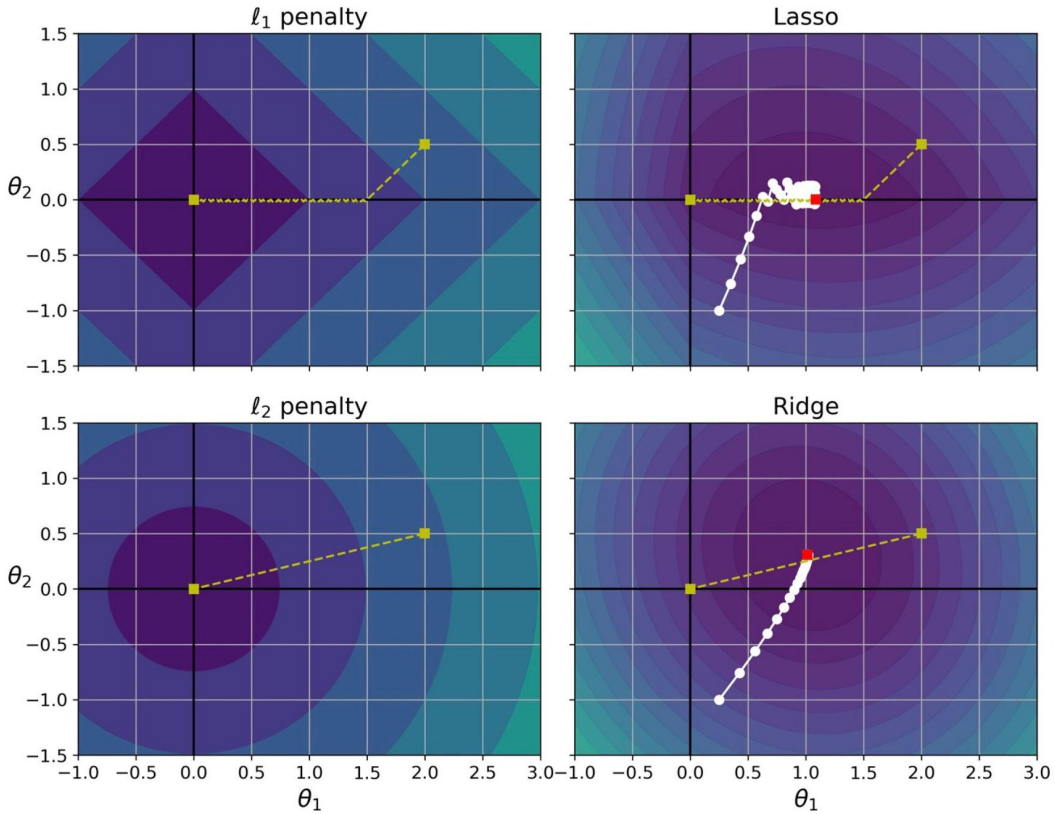
Yellow: regularized params

Red: global optimum
α↑ optimum left
α↓ optimum right

White: gradient descent path

Lasso roll into gutter, bounce around
Can reduce learning rate

Ridge natural slows toward converge



Ridge vs Lasso Regression

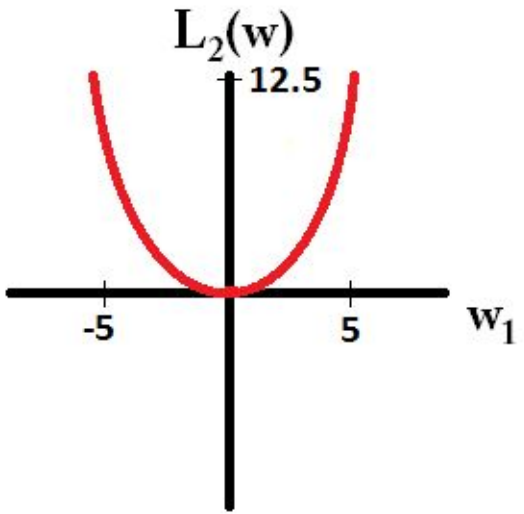
Linear Algebra

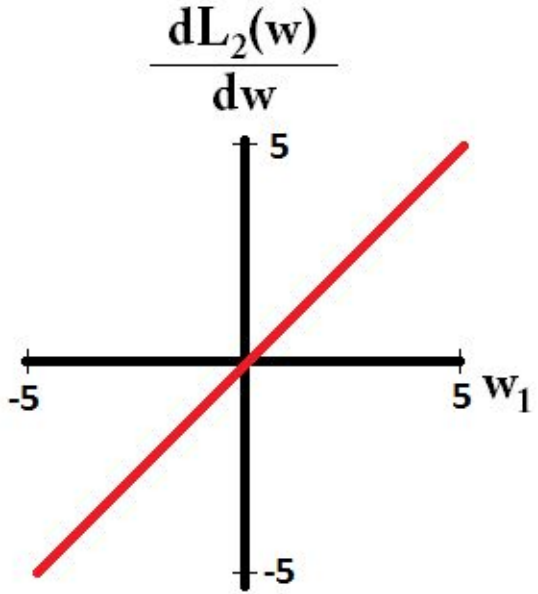
The norm of a vector **w** denoted $\|\mathbf{w}\|$ is a measure of the length or magnitude of **w**.

Multiple possible norms.

Most common is ℓ_2 norm aka Euclidean norm

$\|\mathbf{w}\|_2 = \sqrt{\sum \mathbf{w}^2}$



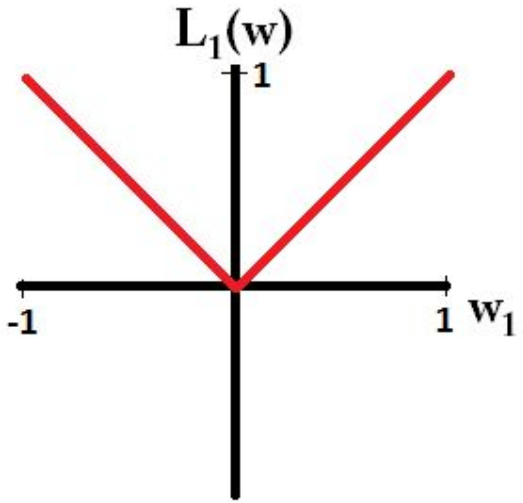


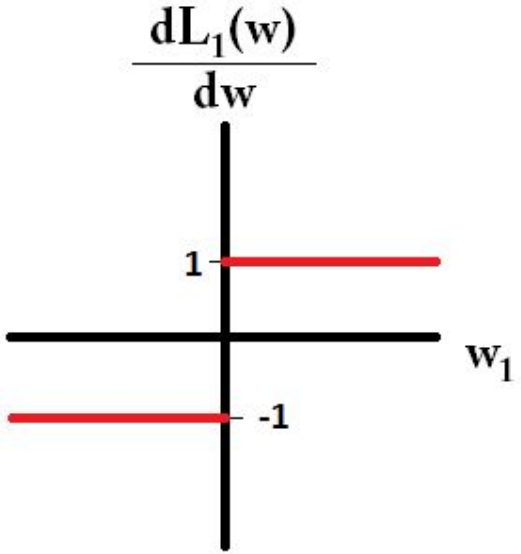
Ridge vs Lasso Regression

Linear Algebra

ℓ_1 norm
aka Manhattan distance

$\|\mathbf{w}\|_1 = \sum |\mathbf{w}|$





Alternate explanations?

Elastic Net

Middle ground between Ridge Regression and Lasso Regression

Control mix ratio r
r=0 Ridge Regression
r=1 Lasso Regression

Lasso Regression with ℓ_1 penalty term

Ridge Regression with ℓ_2 penalty term

$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + r\alpha\sum_{i=1}^{n}\left|\theta_i\right| + \frac{1-r}{2}\alpha\sum_{i=1}^{n}\theta_i^2$$

Elastic Net

```
>>> from sklearn.linear_model import ElasticNet
>>> elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
>>> elastic_net.fit(X, y)
>>> elastic_net.predict([[1.5]])
array([1.54333232])
```

```
>>> sgd_reg = SGDRegressor(penalty="elasticnet",
              l1_ratio=0.5)
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([1.47012588])
```

Lasso Regression with ℓ_1 penalty term

Ridge Regression with ℓ_2 penalty term

$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + r\alpha\sum_{i=1}^{n}|\theta_i| + \frac{1-r}{2}\alpha\sum_{i=1}^{n}\theta_i^2$$

Elastic Net vs Ridge vs Lasso?

Avoid plain Linear Regression. Preferable have a bit of regularization.

Ridge is good default.

Lasso and Elastic Net reduce useless feature weights. Good when only few features useful.

Elastic Net preferred. Lasso sometimes erratic.

Share experiences with regularization? How much time selecting & tuning?

Early Stopping

Stop training when validation error reaches minimum

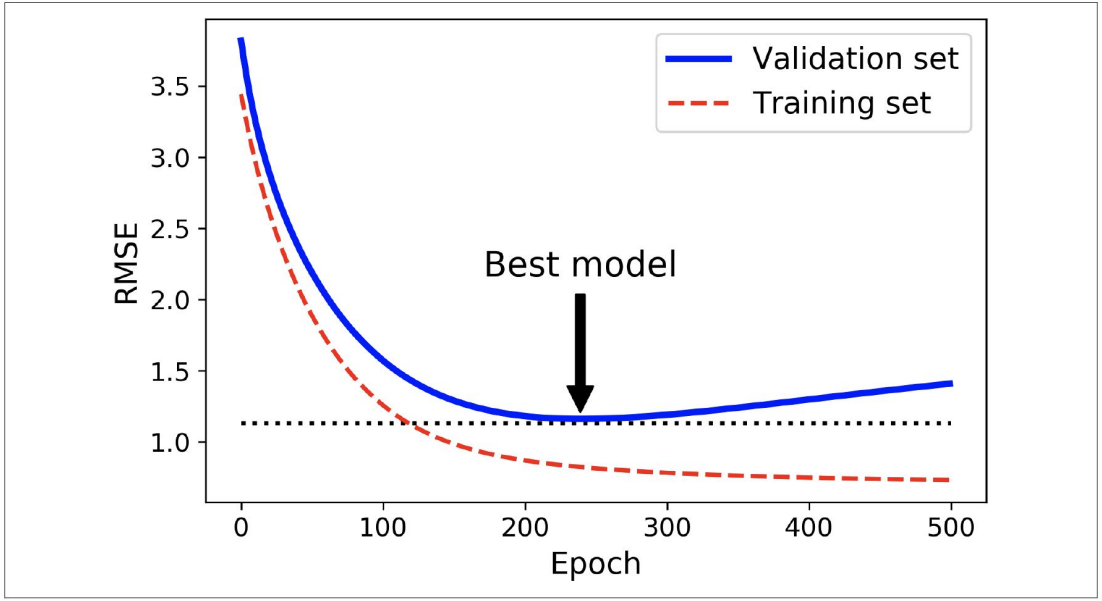


Figure 4-20. Early stopping regularization

Early Stopping

```
from copy import deepcopy
# prepare the data
poly_scaler = Pipeline([
        ("poly_features", PolynomialFeatures(degree=90, include_bias=False)),
        ("std_scaler", StandardScaler())
])
X_train_poly_scaled = poly_scaler.fit_transform(X_train)
X_val_poly_scaled = poly_scaler.transform(X_val)
sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True,
        penalty=None, learning_rate="constant", eta0=0.0005)
minimum_val_error = float("inf")
best_epoch = None
best_model = None
for epoch in range(1000):
        sgd_reg.fit(X_train_poly_scaled, y_train) # continues where it left off
        y_val_predict = sgd_reg.predict(X_val_poly_scaled)
        val_error = mean_squared_error(y_val, y_val_predict)
        if val_error < minimum_val_error:
                minimum_val_error = val_error
                best_epoch = epoch
                best_model = deepcopy(sgd_reg)          # previously sklearn.base.clone
```

Training Models: Logistic Regression

Hands-on Machine Learning: Chapter 4

Logistic Regression

Logit Regression estimates probability

Binary classifier

Logit function, log-odds

$$\text{logit}(p) = \log(\frac{p}{1-p})$$

Logistic Regression

Estimating probabilities

$$\hat{p} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \sigma\left(\mathbf{x}^T\boldsymbol{\theta}\right)$$

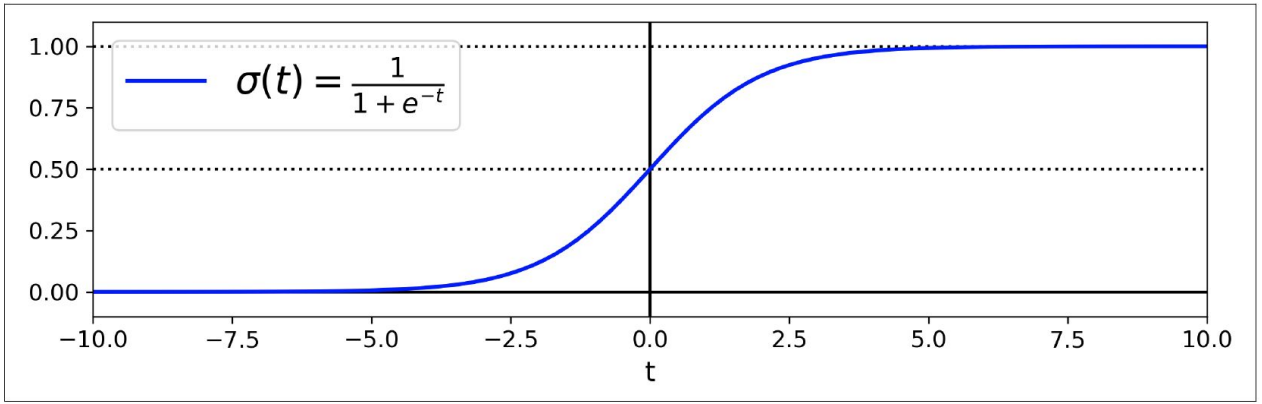
Logistic Regression

Estimating probabilities

$$\hat{p} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \sigma\left(\mathbf{x}^T\boldsymbol{\theta}\right)$$

Sigmoid logistic function

$$\sigma(t) = \frac{1}{1 + \exp\left(-t\right)}$$



Logistic Regression

Estimating probabilities

$$\hat{p} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \sigma\left(\mathbf{x}^T\boldsymbol{\theta}\right)$$

Sigmoid logistic function

$$\sigma(t) = \frac{1}{1 + \exp\left(-t\right)}$$

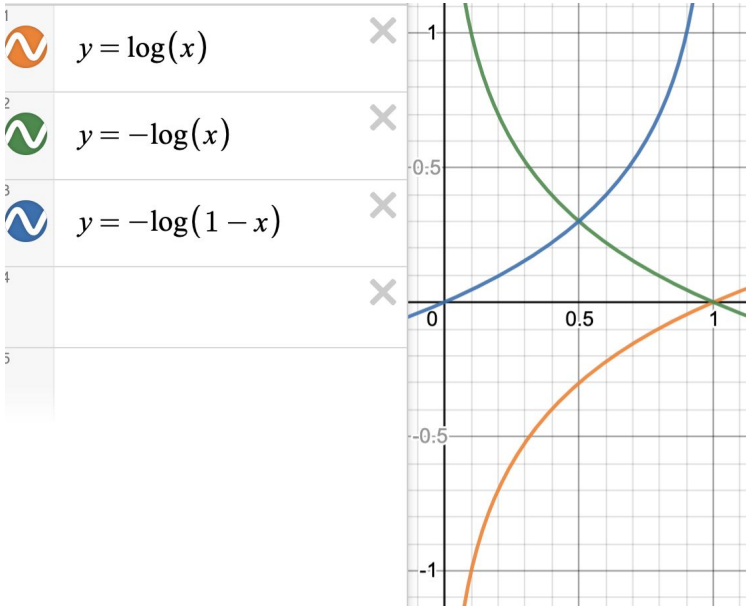
Prediction

$$\hat{y} = \begin{cases} 0 \text{ if } \hat{p} < 0.5 \\ 1 \text{ if } \hat{p} \geq 0.5 \end{cases}$$

Training and Cost Function

High probabilities for positive instances (y=1)
Low probabilities for negative instances (y=0)

$$c(\boldsymbol{\theta}) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1 - \hat{p}) & \text{if } y = 0 \end{cases}$$



Training and Cost Function

High probabilities for positive instances (y=1)
Low probabilities for negative instances (y=0)

$$c(\boldsymbol{\theta}) = \begin{cases} -\log\left(\hat{p}\right) & \text{if } y = 1 \\ -\log\left(1 - \hat{p}\right) & \text{if } y = 0 \end{cases}$$

Log loss average cost over all training instances

$$J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(\hat{p}^{(i)}\right) + \left(1 - y^{(i)}\right)\log\left(1 - \hat{p}^{(i)}\right)\right]$$

Logistic cost partial derivatives
Convex but not closed-form

$$\frac{\partial}{\partial\theta_j}\mathrm{J}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}\left(\sigma\left(\boldsymbol{\theta}^T\mathbf{x}^{(i)}\right) - y^{(i)}\right)x_j^{(i)}$$

Decision Boundaries

Iris dataset 150 flowers

3 species:

- Iris-Setosa
- Iris-Versicolor
- Iris-Virginica

Sepal & petal length & width

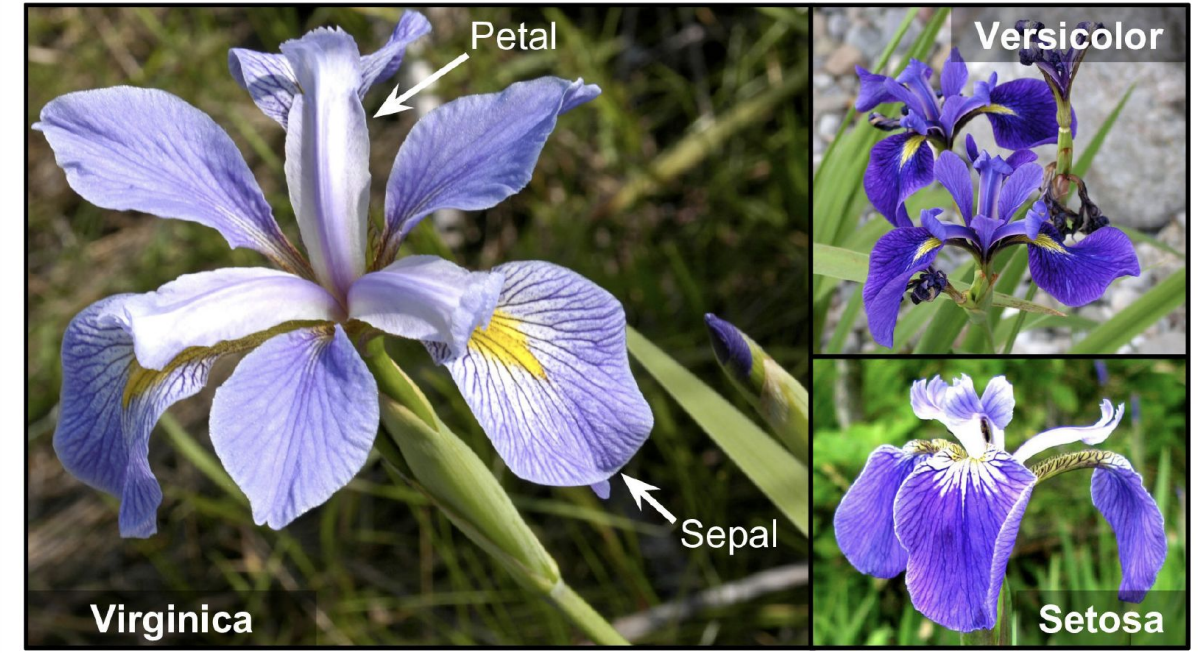


Figure 4-22. Flowers of three iris plant species[16]

Decision Boundaries

Detect Iris-Virginica based only on petal width feature

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename']
>>> X = iris["data"][:, 3:] # petal width
>>> y = (iris["target"] == 2).astype(np.int) # 1 if Iris-Virginica, else 0
```

```
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(X, y)
```

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
plt.plot(X_new, y_proba[:, 1], "g-", label="Iris-Virginica")
plt.plot(X_new, y_proba[:, 0], "b--", label="Not Iris-Virginica")
```

Decision Boundaries

Iris-Virginica 1.4-2.5 cm. Not Iris-Virginica 0.1-1.8 cm. Decision boundary ~1.6 cm

```
>>> log_reg.predict([[1.7], [1.5]])
array([1, 0])
```

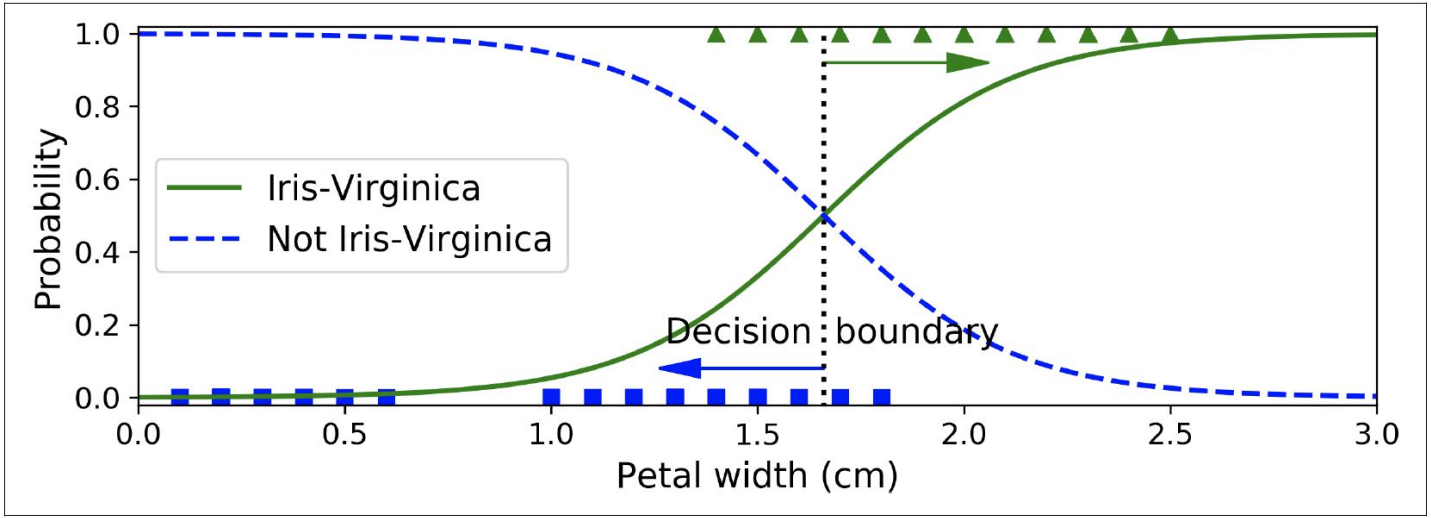


Figure 4-23. Estimated probabilities and decision boundary

Decision Boundaries

Linear boundaries probabilities based on petal width and length.
Logistic Regression can be regularized using ℓ_1 or ℓ_2 penalties, parameter C = inverse of α

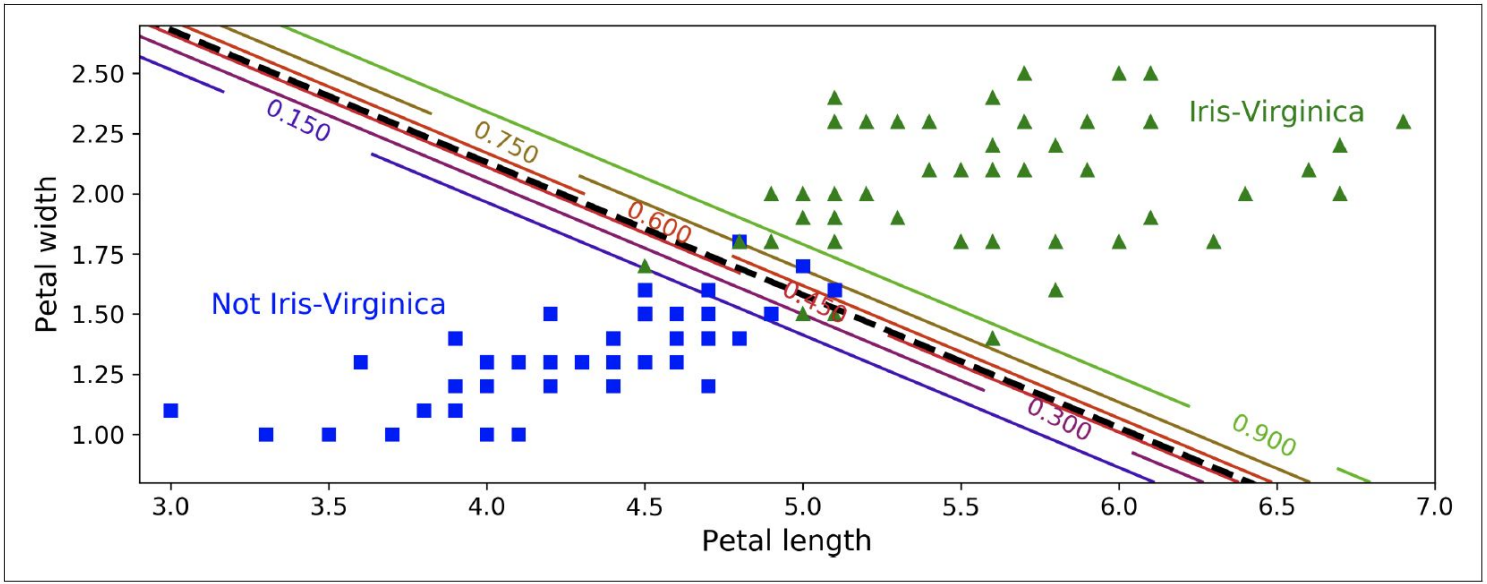


Figure 4-24. Linear decision boundary

Softmax Regression

Multinomial Logistic Regression generalized to support multiple classes

Softmax score for class k

$$s_k(\mathbf{x}) = \mathbf{x}^T \boldsymbol{\theta}^{(k)}$$

Softmax Regression

Multinomial Logistic Regression generalized to support multiple classes

Softmax score for class k

$$s_k(\mathbf{x}) = \mathbf{x}^T \boldsymbol{\theta}^{(k)}$$

Softmax function aka normalized exponential $\sigma(s(x))_k$ probability x in class k given the scores

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} \exp\left(s_j(\mathbf{x})\right)}$$

Softmax Regression

Multinomial Logistic Regression generalized to support multiple classes

Softmax score for class k

$$s_k(\mathbf{x}) = \mathbf{x}^T \boldsymbol{\theta}^{(k)}$$

Softmax function aka normalized exponential $\sigma(s(x))_k$ probability x in class k given the scores

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} \exp\left(s_j(\mathbf{x})\right)}$$

Softmax regression classifier prediction with highest estimated probability

$$\hat{y} = \underset{k}{\operatorname{argmax}}\ \sigma(\mathbf{s}(\mathbf{x}))_k = \underset{k}{\operatorname{argmax}}\ s_k(\mathbf{x}) = \underset{k}{\operatorname{argmax}}\ \left(\left(\boldsymbol{\theta}^{(k)}\right)^T \mathbf{x}\right)$$

Only predicts 1 class at a time.
Mutually exclusive classes not multi-output.

Cross Entropy

Measure how well estimated class probabilities match target classes.

Cross entry cost function
$y_k^{(i)}$ is target probability i^{th} instance belongs to class k. k=2 binary classification.

$$J(\mathbf{\Theta}) = -\frac{1}{m}\Sigma_{i=1}^{m}\Sigma_{k=1}^{K} y_k^{(i)}\log\left(\hat{p}_k^{(i)}\right)$$

Cross entry gradient vector for class k

$$\nabla_{\boldsymbol{\theta}^{(k)}} J(\mathbf{\Theta}) = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{p}_k^{(i)} - y_k^{(i)}\right)\mathbf{x}^{(i)}$$

Cross Entropy

Use Softmax Regression to classify iris flowers into all 3 classes

```
X = iris["data"][:, (2, 3)] # petal length, petal width
y = iris["target"]
softmax_reg = LogisticRegression(multi_class="multinomial",solver="lbfgs", C=10)
softmax_reg.fit(X, y)
```

Predict petal 5 cm long & 2 cm wide

- class 0: Iris-Setosa
- class 1: Iris-Versicolor 5.8%
- class 2: Iris-Virginica 94.2%

```
>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]])
array([[6.38014896e-07, 5.74929995e-02, 9.42506362e-01]])
```

Cross Entropy

Linear decision boundary probabilities for Iris-Versicolor.
All meet at 33%

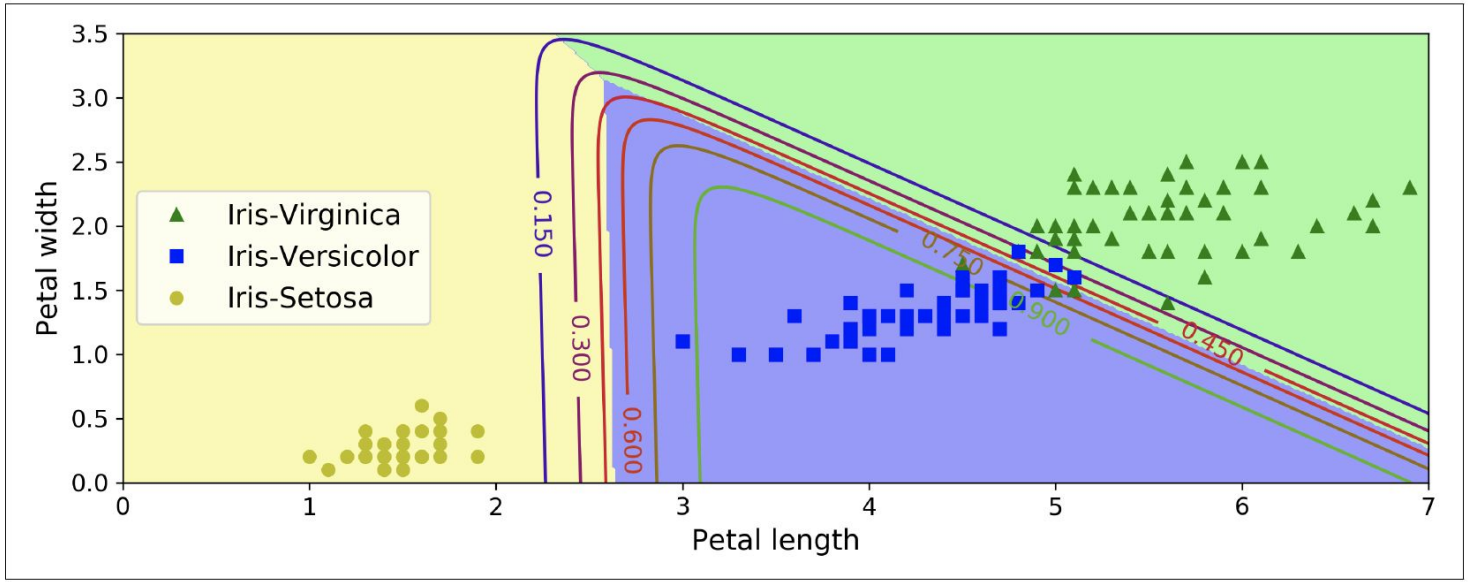


Figure 4-25. Softmax Regression decision boundaries