Lecture 4: Training Models II

Dr.-Ing. Sudchai Boonto, Assistant Professor

February 9, 2018

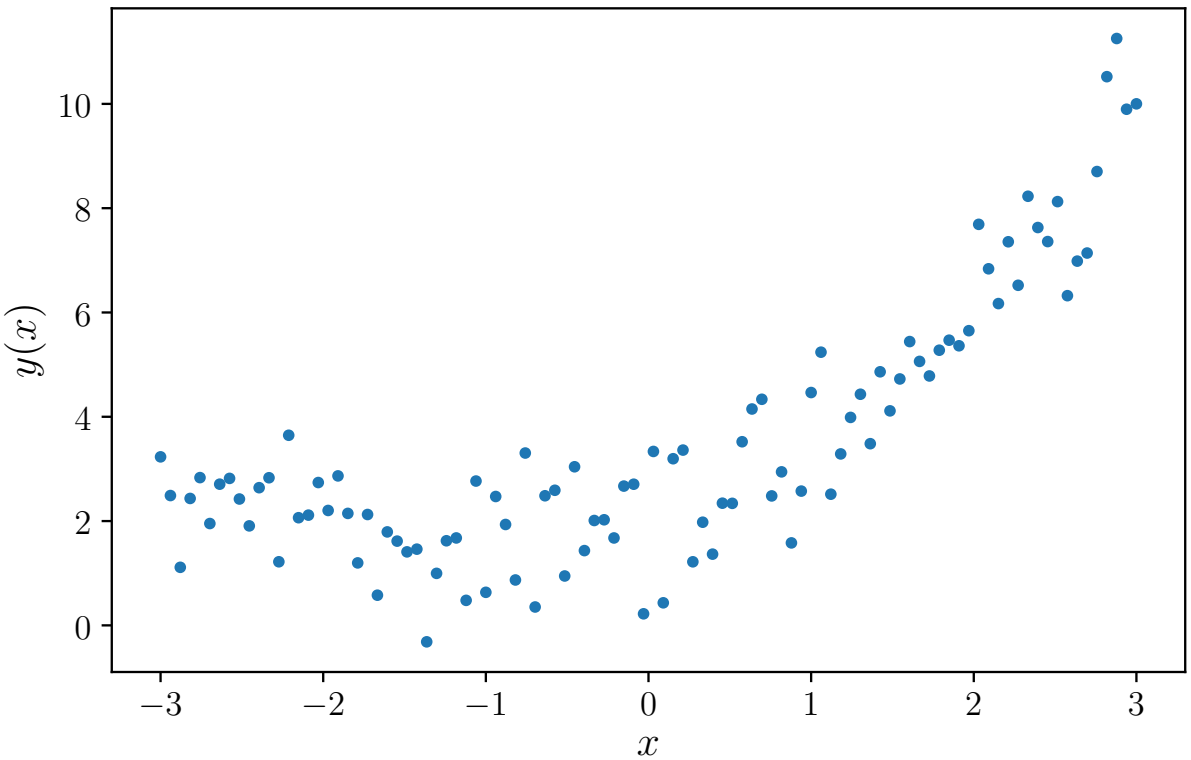
Department of Control System and Instrument Engineering, KMUTT

Polynomial Regression

Using Batch Gradient Descent

Preparing data:

```
m = 100
X = np.linspace(-3, 3, 100)
X = np.c_[X]
y = 2 + X + 0.5 * X**2 + np.random.randn(m,1)
```



Using Batch Gradient Descent

Using normal Batch Gradient Descent:

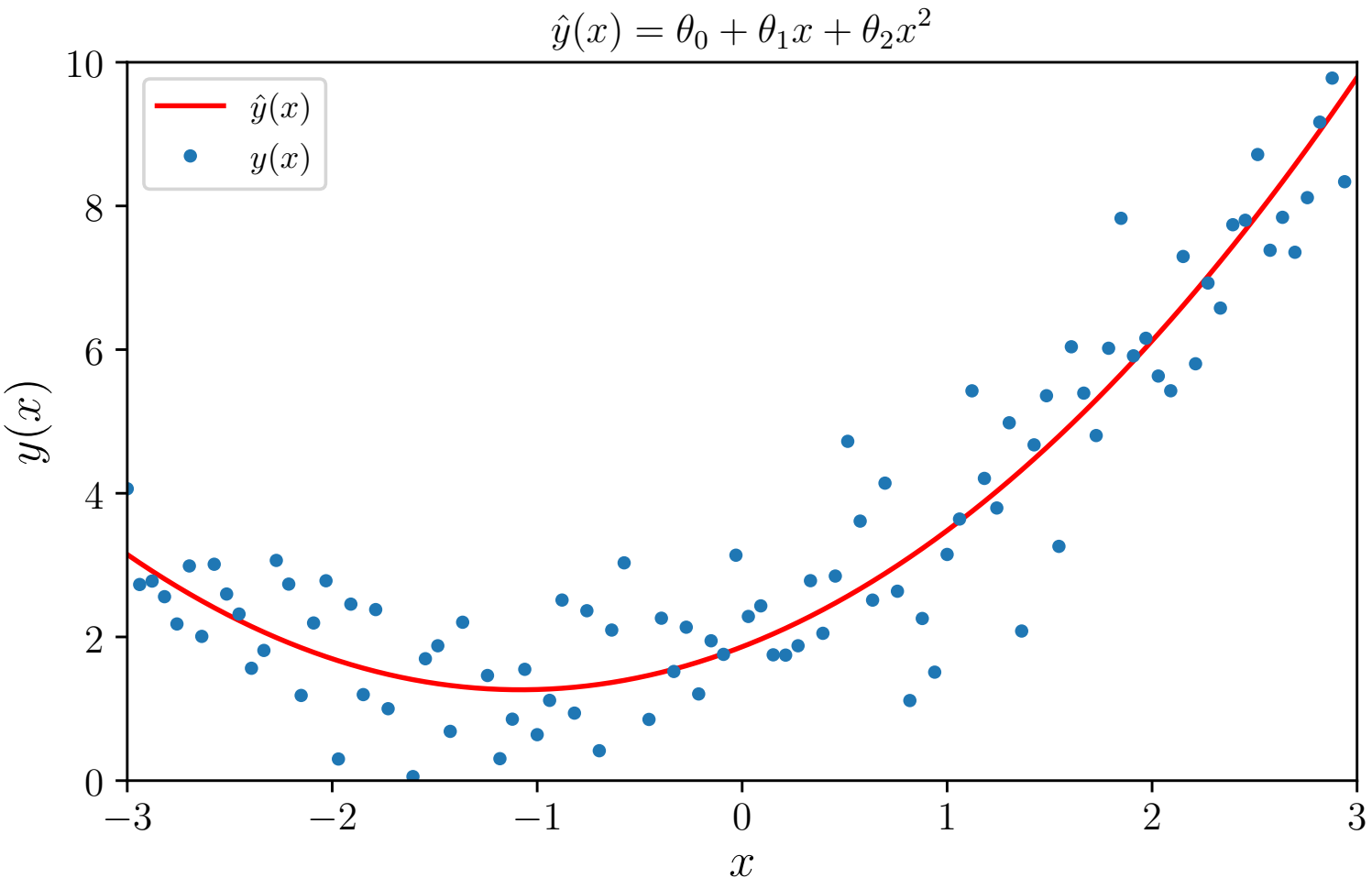
$$\theta_{(k+1)} = \theta_k - \eta \nabla_\theta V(\theta),$$

where $V(\theta)$ is a cost function (MSE).

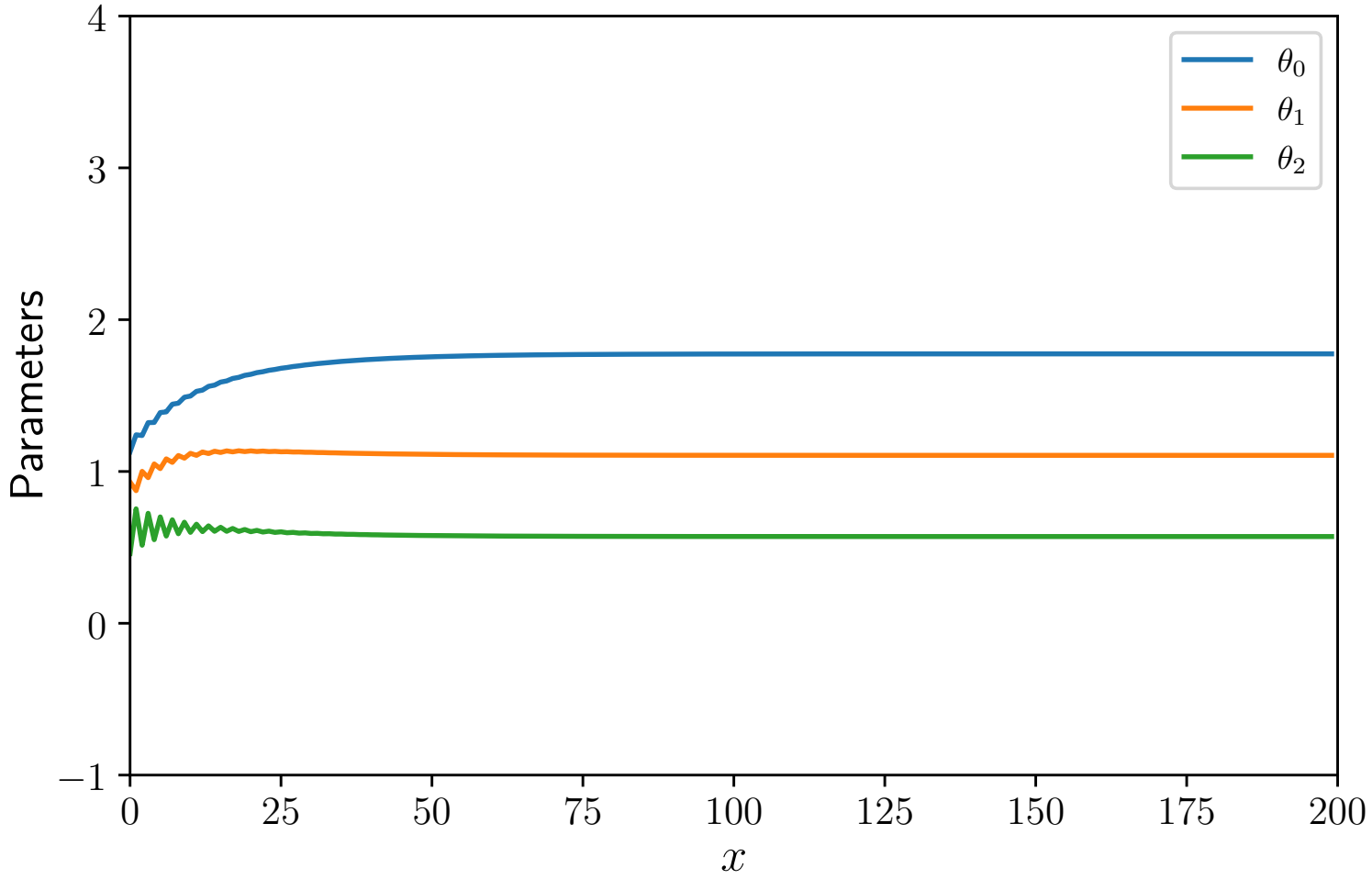
```
N_train = 80
X_b = np.c_[np.ones((N_train, 1)), x_train, x_train**2]

eta = 0.07
n_iterations = 200
m = N_train
theta = np.random.randn(3,1)

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y_train)
    theta = theta - eta * gradients
```

η is a learning rate. It should be small enough. The necessary condition is $\eta < 2/(\lambda_{\max}(H(V(\theta))))$.

Using Batch Gradient Descent



Using Batch Gradient Descent



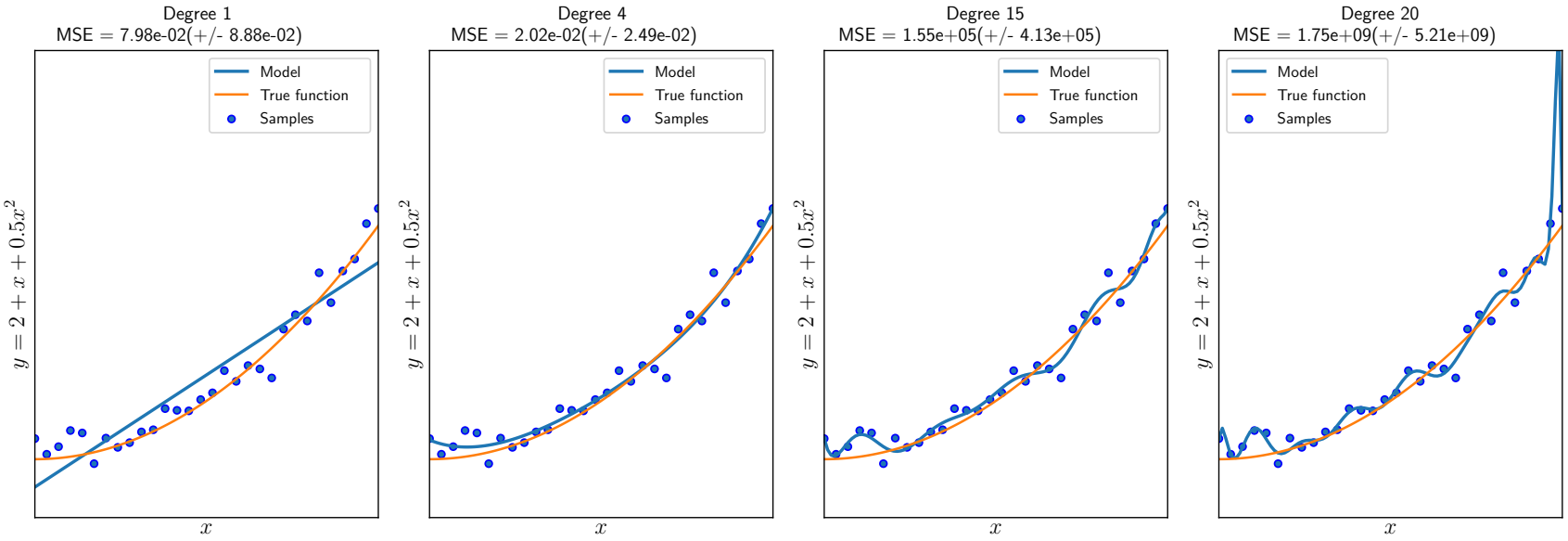
$\theta_0 = 1.8632, \theta_1 = 1.1067, \theta_2 = 0.5116$

Using Scikit-Learn

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

m = 100
X = np.linspace(-3,3, 100)
X = np.c_[X]
y = 2 + X + 0.5 * X**2 + np.random.randn(m,1)
N_train = 80
x_train = X[0:80]
y_train = y[0:80]

poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(x_train)

lin_reg = LinearRegression()
lin_reg.fit(X_poly,y_train)
lin_reg.intercept_, lin_reg.coef_
```

$\theta_0 = 2.0072, \theta_1 = 0.9621, \theta_2 = 0.4638$

Learning Curves



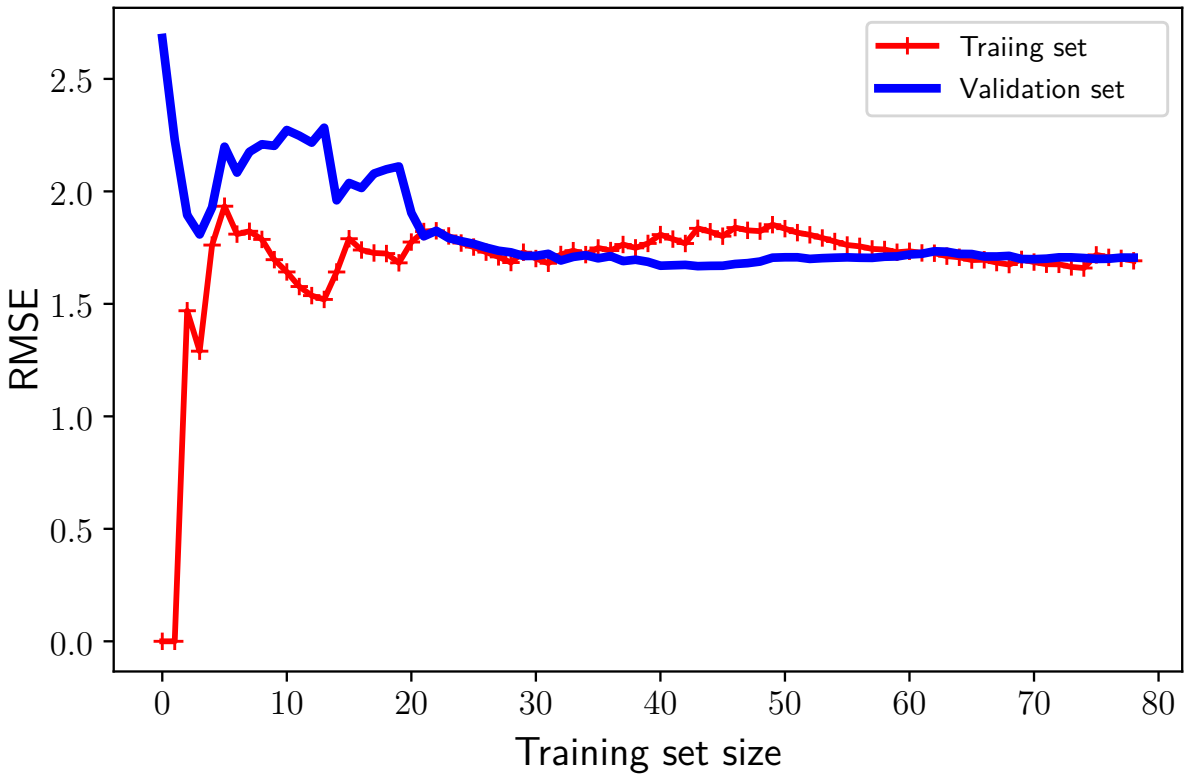
- High-degree Polynomial Regression model is severely overfitting the training data
- Linear model is underfitting the training data, the quadratic model is perfect fit.

Learning Curves

- **Learning curves:** these are plots of the model's performance on the training set and the validation set as a function of the training set size.

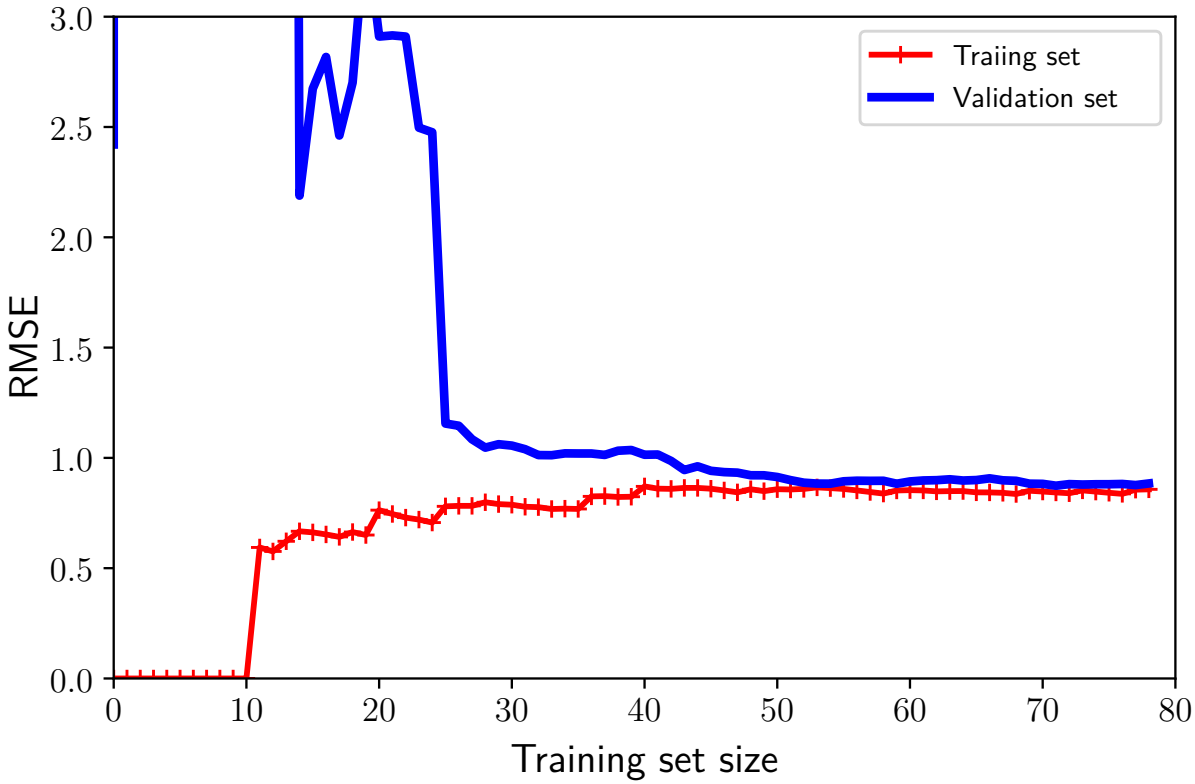
```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)  # 20 percent
    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_predict,
                                               y_train[:m]))
        val_errors.append(mean_squared_error(y_val_predict, y_val))
    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")

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

Learning Curves



- From the learning curves of linear model show it is an underfitting model. Both curves have reached a plateau at quite high value.

Learning Curves



- The error on the training data is much lower than with the Linear Regression model.
- If there is a gap between the curves. This means that the model performs significantly better on the training data than on the validation data, which is the hallmark of an overfitting model. If we have larger training set, the two curves would continue to get closer.

Tradeoff

The Bias/Variance Tradeoff

Bias This part of generalization error is due to 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 This part is due to the model's excessive sensitivity to small variations in the training data. A model with many degrees of freedom (such as a high-degree polynomial model) is likely to have high variance, and thus to overfit the training data.

Irreducible error This part is due to the noisiness of the data itself. The only way to reduce this part of the error is to clean up the data (e.g. fix the data sources, such as broken sensors, or detect and remove outliners).

Increasing a model's complexity will typically increase its variance and reduce its bias. Conversely, reducing a model's complexity increases its bias and reduces its variance. This is why it is called a tradeoff.

Regularization

Regularized Linear Models

For linear model (linear-in-parameters model), regularization is typically achieved by constraining the weights of the model. Here we consider:

- Ridge Regression (Tikhonov Regularization)
- Lasso Regression
- Elastic Net

Ridge Regression (Tikhonov Regularization)

The **Ridge Regression** addresses some of the problem of **Ordinary Least Squares** is defined by imposing a penalty on the size of coefficients. The ridge coefficients minimize a penalized residual sum of squares

$$\min_{\theta} V(\theta) = \|\theta^T x - y\|^2 + \alpha\|\theta\|^2 = \theta^T(X^T X + \alpha I)\theta - 2y^T X\theta + y^T y$$

The analytic solution is

$$\hat{\theta} = (X^T X + \alpha I)^{-1} X^T y$$

Here, $\alpha \geq 0$ is a complexity parameter that controls the amount of shrinkage: the larger the value of α, the greater the amount of shrinkage and thus the coefficients become more robust to colinearity.

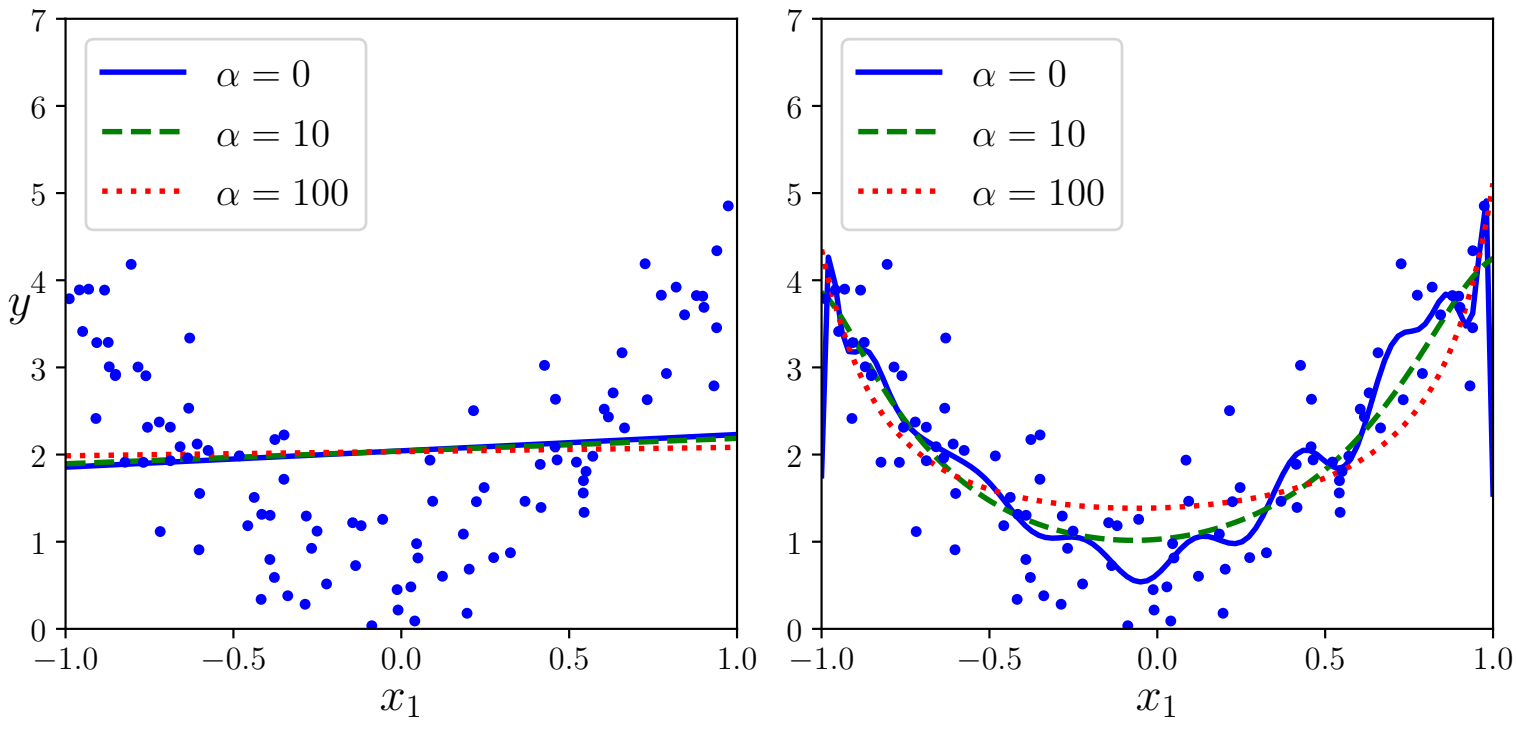
```
from sklear.linear_model import Ridge
ridge_reg = Ridge(alpha=1, solvers"cholesky")
ridge_reg.fit(X,y)
ridge_reg.predict([[1.5]])
```

you will get `array([[ 1.55071465]])`

Ridge Regression (Tikhonov Regularization)

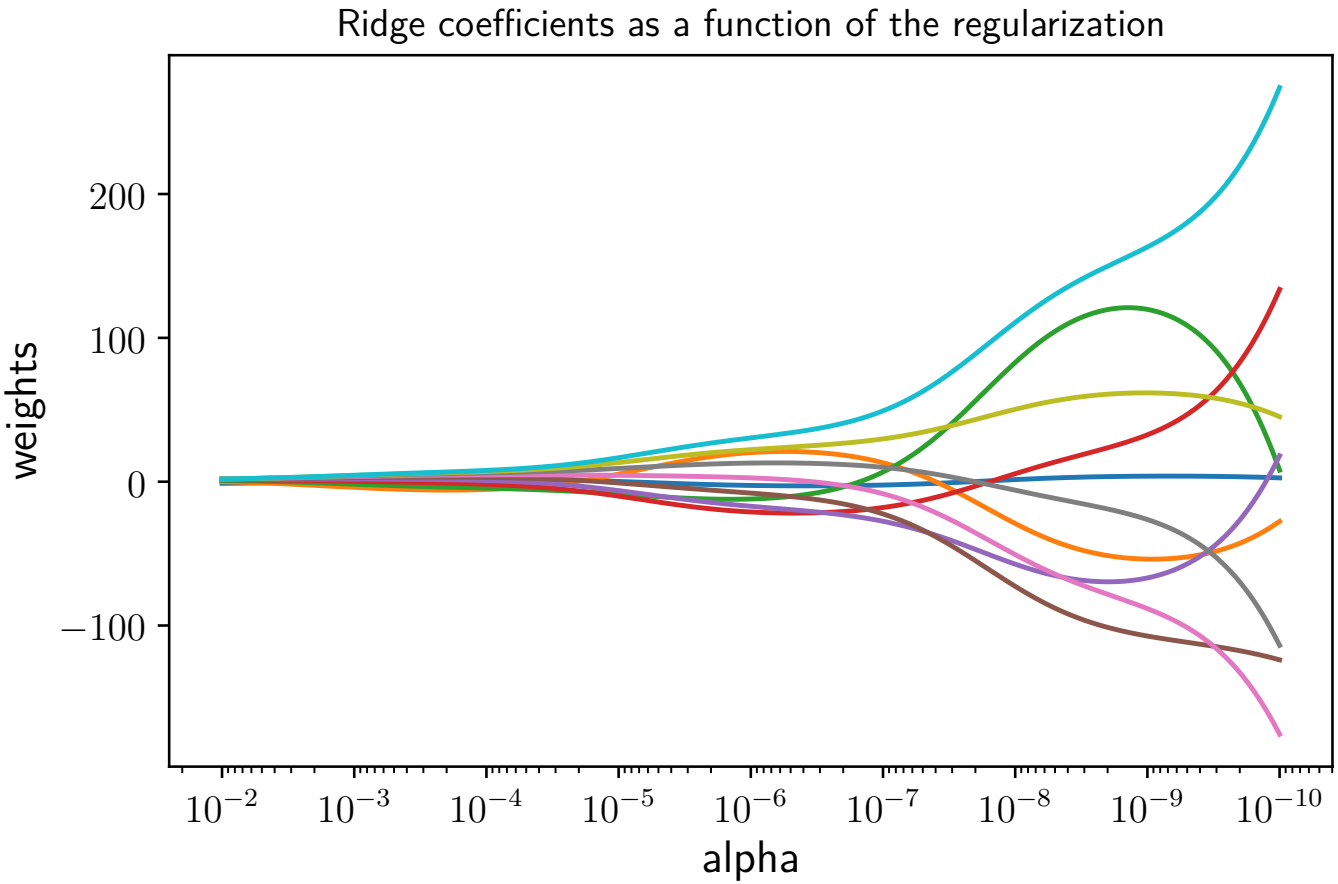
```
from sklear.linear_model import Ridge
ridge_reg = Ridge(alpha=1, solvers"cholesky")
ridge_reg.fit(X,y)
ridge_reg.predict([[1.5]])
```

you will get `array([[ 1.55071465]])`



The left hand side is underfit.

Ridge Regression (Tikhonov Regularization)

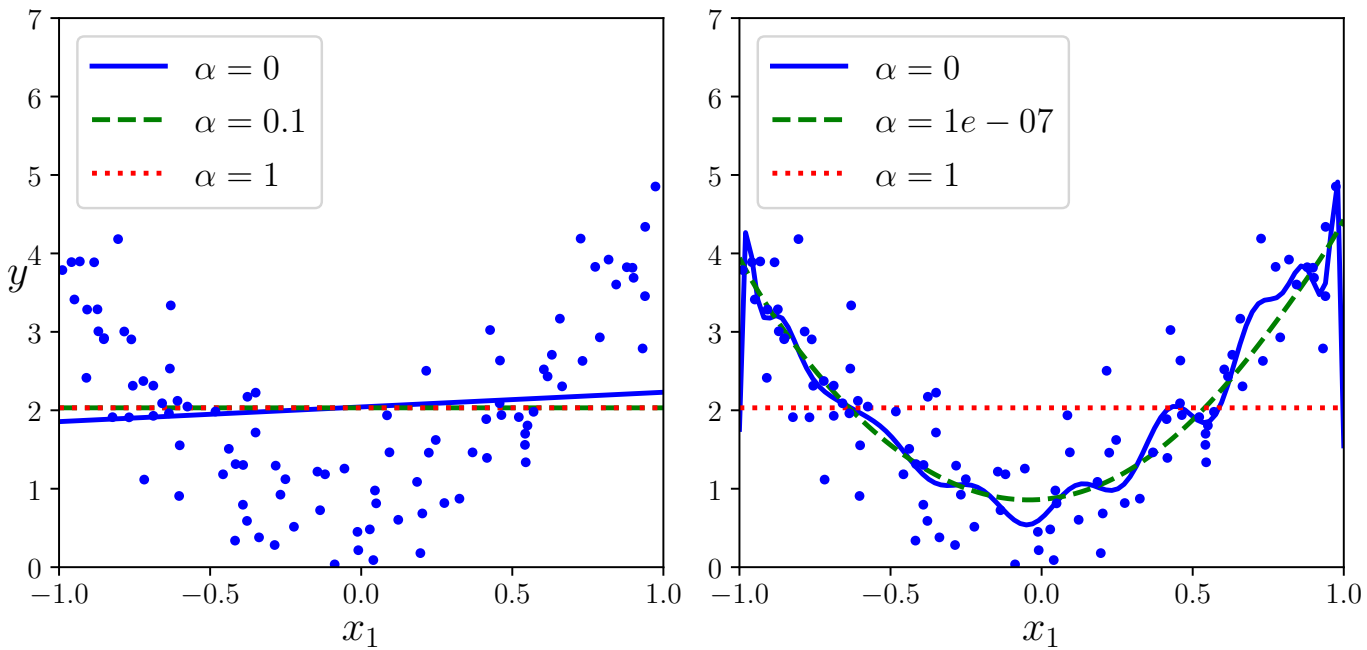


When alpha is very large, the regularization effect dominates the squared loss function and the coefficients tend to zero. At the end of the path, as alpha tends toward zero and the solution tends towards the ordinary least squares, coefficients exhibit big oscillations. In practise it is necessary to tune alpha in such a way that a balance is maintained between both.

Lasso Regression

Least Absolute Shrinkage and Selection Operator Regression: it is just like Ridge Regression, it adds a regularization term to the cost function, but it uses the l_1 norm of the weight vector instead of half the square of the l_2 norm

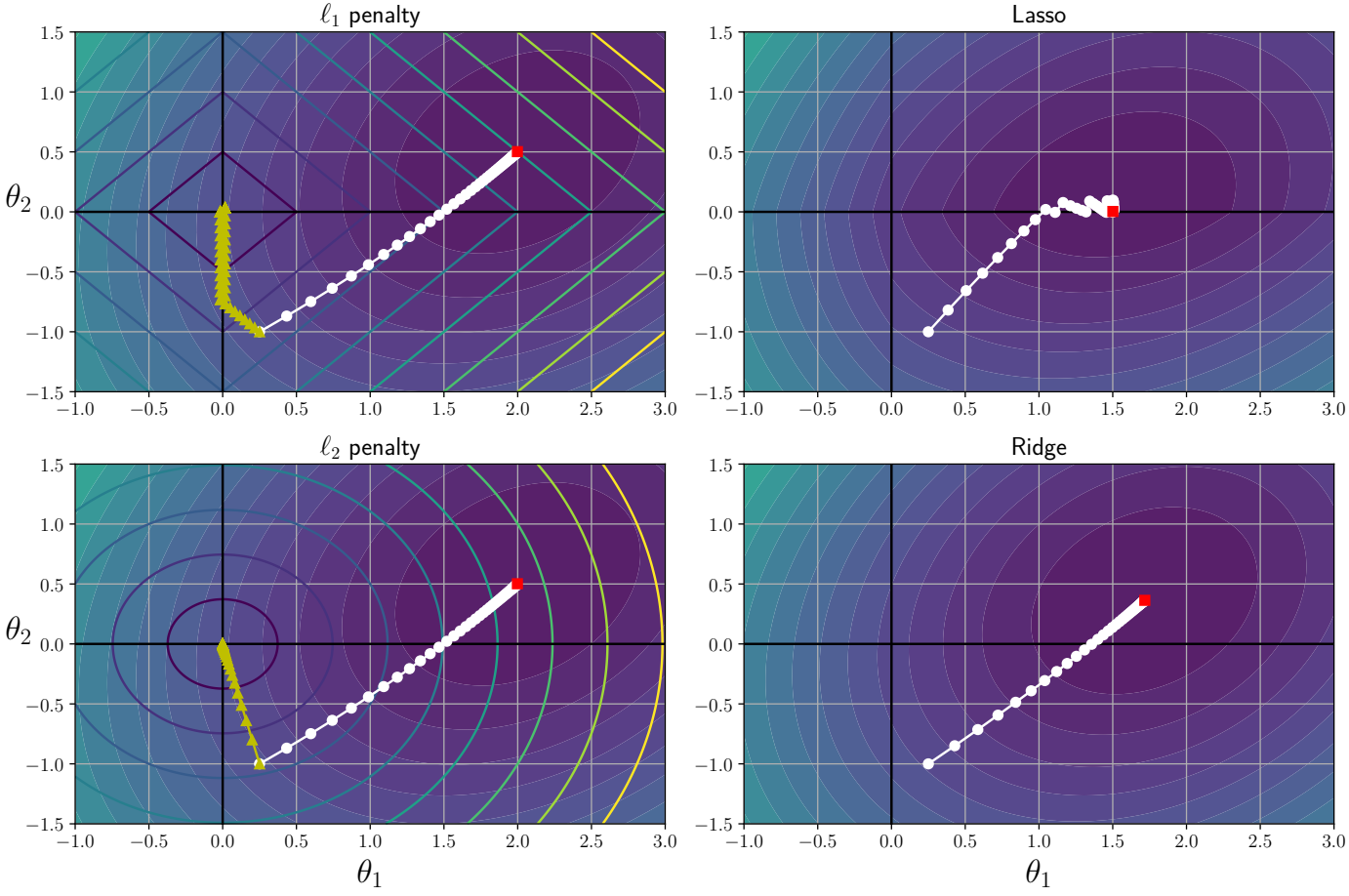
$$\min_{\theta} V(\theta) = \|\theta^T x - y\|^2 + \alpha\|\theta\|_1$$



Lasso Regression

- An important characteristic of Lasso Regression is that it tends to completely eliminate the weights of the least important features (i.e., set them to zero)
- In the previous picture the dashed line in the right plot (with $\alpha = 10^{-7}$ looks quadratic, almost linear: all the wights for the high-degree polynomial features are equal to zero.
- Lasso Regression automatically performs feature selection and outputs a *sparse model* (ie., with few nonzero feature weights).

Lasso Regression



Elastic Net

Elastic Net is a middle ground between Ridge Regression and Lasso Regression. The regularization term is a simple mix of both Ridge and Lasso's regularization terms, and you can control the mix ratio r. When $r = 0$, Elastic Net is equivalent to Ridge Regression, and when $r = 1$ it is equivalent to Lasso Regression

$$\min_{\theta} V(\theta) = \|\theta^T x - y\|^2 + r\alpha\|\theta\|_1 + \frac{1-r}{2}\alpha\|\theta\|$$

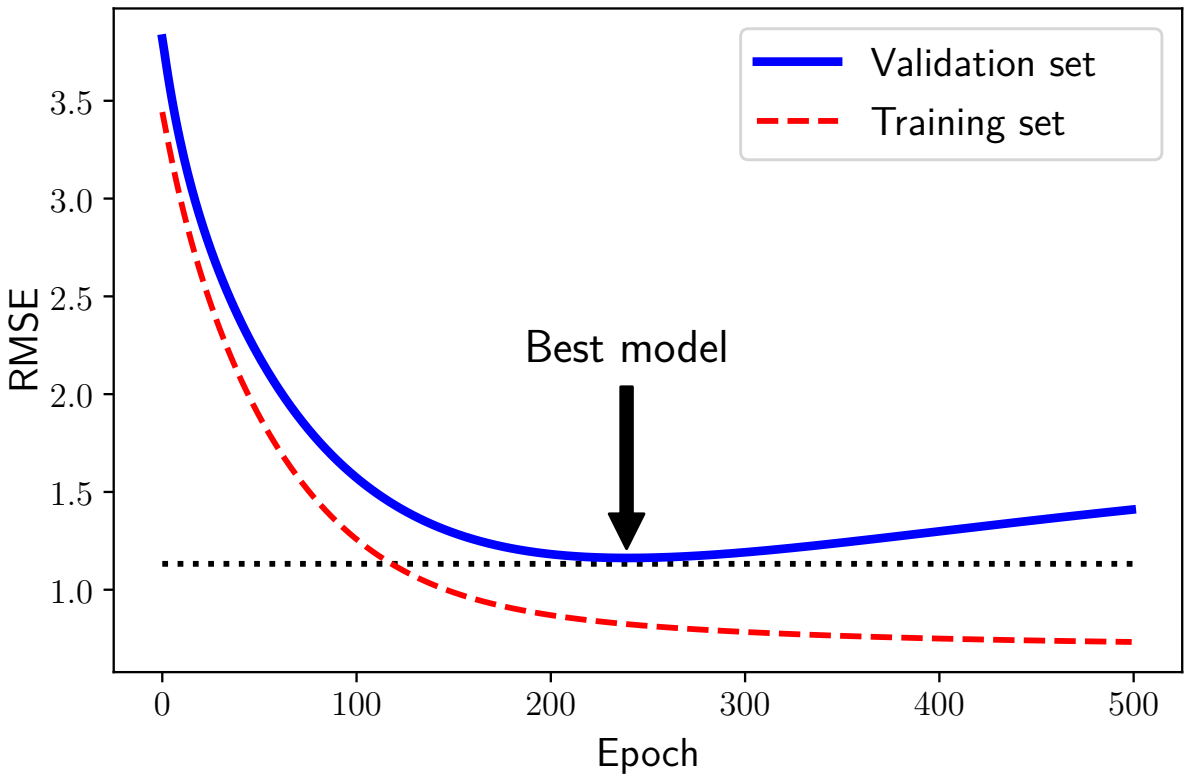
- You should use one of the regularization, at least.
- Ridge is a good default. If you want only a few features, you should go to Lasso or Elastic Net.
- Elastic Net is in general better than Lasso, since Lasso may behave erratically when the number of features is greater than the number of traning instances or when several features are strongly correlated.

Elastic Net

```
from sklearn.linear_model import ElasticNet
from sklearn.datasets import make_regression

X, y = make_regression(n_features=2, random_state=0)
elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
elastic_net.fit(X,y)
print(elastic_net.coef_)
print(elastic_net.intercept_)
print(elastic_net.predict([[0, 0]]))
```

Early Stopping

- A very different way to regularize iterative learning algorithms such as Gradient Descent is to stop training as soon as the validation error reaches a minimum.
- the technique is called **early stopping**



Logistic Regression

Logistic Regression

Logistic Regression or **Logit Regression** is commonly used to estimate the probability that an instance belongs to a particular class. If the estimated probability is greater than 50%, then the model predicts that the instance belongs to that class (called the positive class, labeled 1), or else it predicts that it does not (ie. it belongs to the negative class, labeled 0). This makes it a binary classifier

Estimating Probabilities

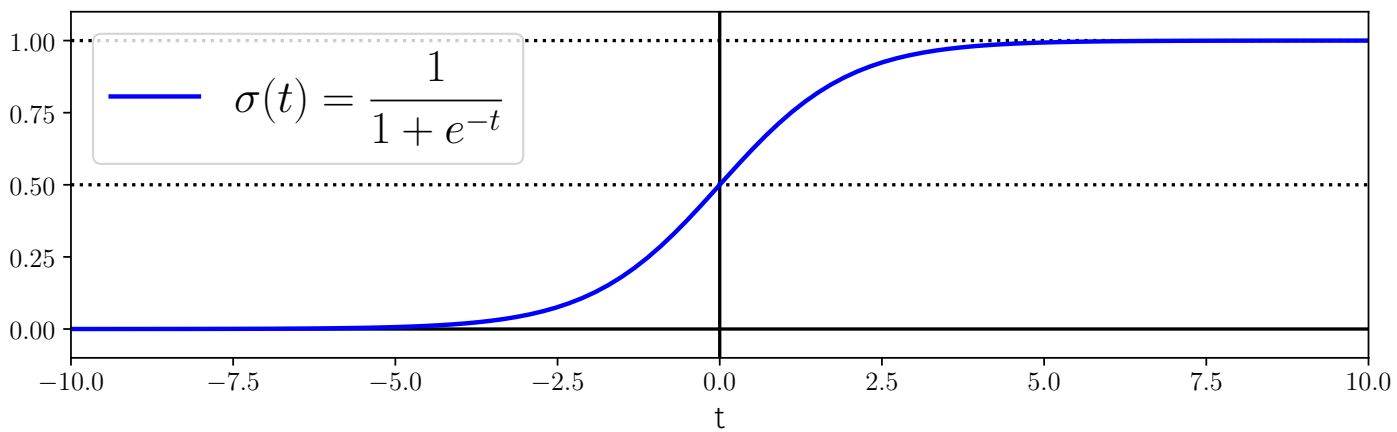
The output of logistic is the output of the closed-form formula:

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

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

The Logistic Regression model prediction is

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



Training and Cost Function

For a single training instance x the cost function is

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

- the $-\log(t)$ grows very large when t approaches 0
- the cost will be large if the model estimates a probability close to 0 for a positive instance, and it will also be very large if the model estimates a probability close to 1 for a negative instance.
- the $-\log(t)$ is close to 0 when t is close to 1, so the cost will be close to 0 if the estimated probability is close to 0 for a negative instance or close to 1 for a positive instance.

Training and Cost Function

the cost function over the whole training set is simply the average cost over all training instances. It can be written in a single expression, called the **log loss**.

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

- There is no closed-form equation to compute the value of θ
- It is a convex function, so Gradient Descent (or any other optimization algorithm) is guaranteed to find the global minimum.

The partial derivatives of the cost function with regards to the j^{th} model parameter θ_j is given by

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

We can train the model by using the Gradient based optimization.

Softmax Regression

The Logistic Regression model can be generalized to support multipleclasses directly, whithout having to train an combine multiple binary classifiers. This is called **Softmax Regression** or **Multinominal Logistic Regression**. The idea is very simple:

- when given an instance x, the Softmax Regression model first computes a score $s_k(x)$ for each class k.
- estimates the probability of each class by applying the **softmax function** to the scores
- The equation to compute $s_k(x)$ is

$$s_k(x) = \left(\theta^{(k)}\right)^T x$$

The each class has its own dedicated parameter vector $\theta^{(k)}$. All these vectors are typically stored as rows in a **parameter matrix** Θ.

Softmax Regression

- Once you have computed the score of every class for the instance x, you can estimate the probability $\hat{p}_k$ that the instance belongs to class k by running the scores through the softmax function:

$$\hat{p}_k = \sigma(s(x))_k = \frac{e^{s_k(x)}}{\sum_{j=1}^{K} e^{(s_j(x))}},$$

  - K is the number of classes
  - $s(x)$ is a vector containing the scores of each class for the instance x
  - $\sigma(s(x))_k$ is the estimated probability that the instance x belongs to class k given the scores of each class for that instance.

- The softmax Regression classifier predicts the class with the highest estimated probability

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

  The argmax operator returns the value of a variable that maximizes a function. Here it returns the value of k that maximizes the estimated probability $\sigma(s(x))_k$.

Softmax Regression: Training

The objective is to have a model that estimates a high probability for the target class (and consequently a low probability for the other classes).

- Minimizing the cost function, which is called **Cross Entropy**

$$V(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right)$$

  $y_k^{(i)}$ is equal to 1 if the target class for the i^{th} instance is k otherwise, it is equal to 0

- The gradient vector of this cost function with regards to $\theta^{(k)}$ is given by

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

  We can use Gradient based optimization to find the parameter matrix Θ that minimize the cost function.

Reference

1. Sebatian Ruder, "*An Overview of Gradient Descent Optimization Algorithms*", arXiv:1609.04747v2, 2017
2. Aurélien, Géron, "*Hands-On Machine Learning with Scikit-Learn & TensorFlow*", O'reilly, 2017
3. S.P.K. Spielberg, R. B. Gopaluni, P. D. Loewen, "Deep Reinforcement Learning Approaches for Process Control", 2017 6th International Symposium on Advanced control of Industrial Processes (AdCONIP), May 28-31, 2017, Taipei, Taiwan