3253 Analytic Techniques and Machine Learning

Module 5: Training Models and Feature Selection

Course Plan

Module Titles
Module 1 – Introduction to Machine Learning
Module 2 – End to End Machine Learning Project
Module 3 – Classification
Module 4 – Clustering and Unsupervised Learning
Current Focus: Module 5 – Training Models and Feature Selection
Module 6 – Support Vector Machines
Module 7 – Decision Trees and Ensemble Learning
Module 8 – Dimensionality Reduction
Module 9 – Introduction to TensorFlow
Module 10 – Introduction to Deep Learning and Deep Neural Networks
Module 11 – Distributing TensorFlow, CNNs and RNNs
Module 12 – Final Assignment and Presentations (no content)

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Learning Outcomes for this Module

- Apply alternative learning strategies to regression
- Evaluate model learning using learning curves
- Employ regularized regression for automatic feature selection
- Learn a logistic regression model for categorical classification problems

Topics for this Module

- **5.1** Linear regression using the normal equations
- **5.2** Linear regression using gradient descent
- **5.3** Non-linear regression
- **5.4** Learning curves
- **5.5** Regularization and ridge regression
- **5.6** Logistic regression
- **5.7** Resources and Wrap-up

Module 5 – Section 1

Linear Regression Using the Normal Equations

Introduction

- So far we have treated Machine Learning models and their training algorithms mostly like black boxes.
    - you optimized a regression system,
    - you improved a digit image classifier,
    - All this without knowing how they actually work

- However, having a good understanding of how things work can help you quickly identify the appropriate model, the right training algorithm to use, and a good set of hyperparameters for your task

Train a Linear Regression Model

- Using a direct "closed-form" equation that directly computes the model parameters that best fit the model to the training set

- Using an iterative optimization approach, called Gradient Descent (GD)
  - Variants:
    - Batch GD
    - Mini-batch GD
    - Stochastic GD

Linear Regression

- The target variable is continuous and expressed as a linear combination of predictors (features)

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

- The goal is to find the parameters -> θ_i which are those minimizing the function:

$$\text{MSE}(\mathbf{X}, h_\theta) = \frac{1}{m} \sum_{i=1}^{m} (\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)})^2$$

The Normal Equation

- To find the value of θ that minimizes the cost function, there is a closed-form solution

$$\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

- $\hat{\theta}$ is the value of theta that minimizes the cost function.

- y is the vector of target values containing y(1) to y(m).
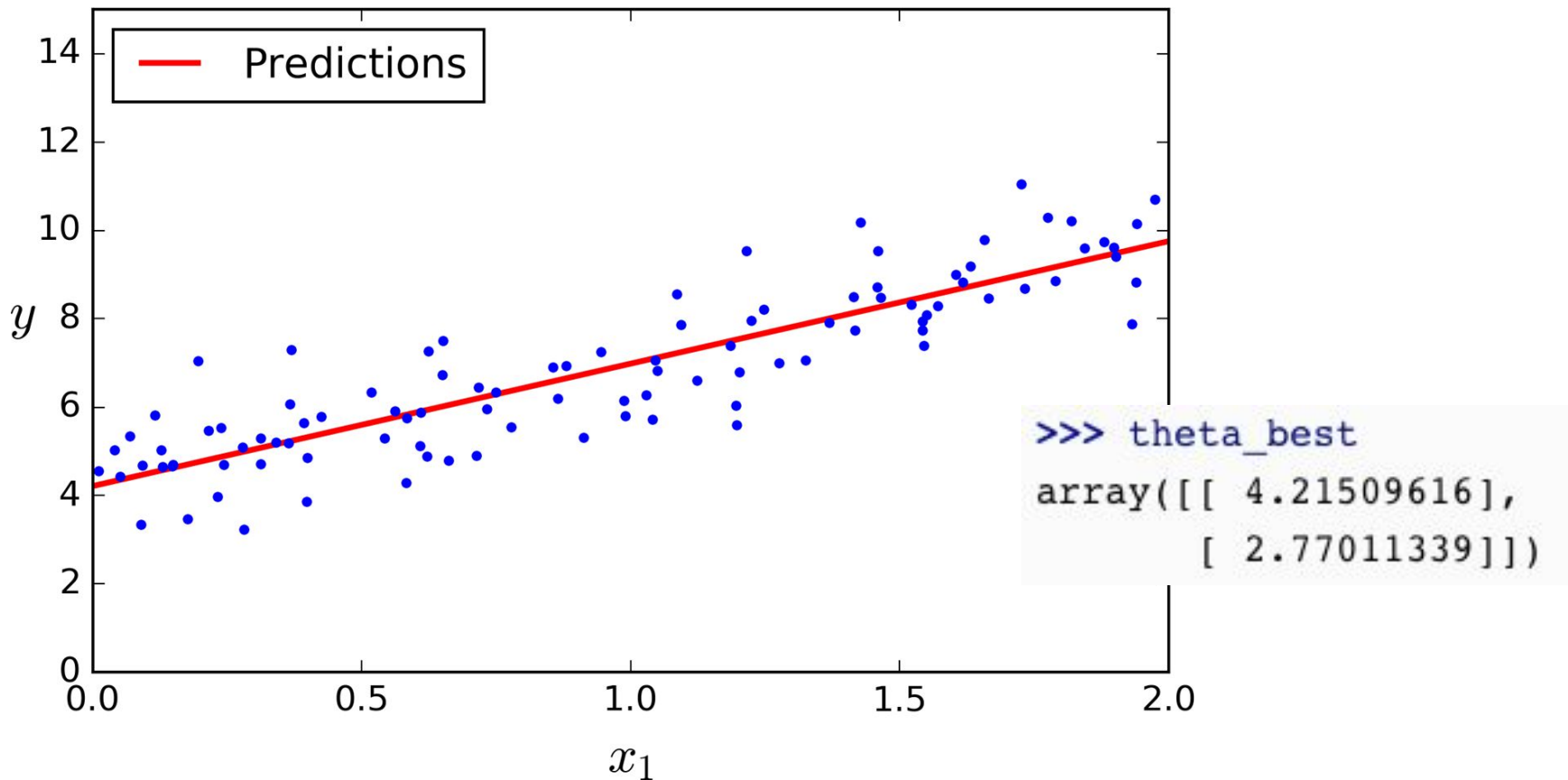
Generated Data

```python
import numpy as np

X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

Solve Using Normal Equation

```python
X_b = np.c_[np.ones((100, 1)), X]   # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```



```
>>> theta_best
array([[ 4.21509616],
       [ 2.77011339]])
```

Scikit-learn

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([ 4.21509616]), array([[ 2.77011339]]))
>>> lin_reg.predict(X_new)
array([[ 4.21509616],
       [ 9.75532293]])
```

<u>Complexity</u>

- The Normal Equation computes the inverse of $X^T \cdot X$, which is an n × n matrix
  - the computational complexity of inverting such a matrix is typically about $O(n^{2.4})$ to $O(n^3)$, depending on the implementation

- What are some other ways of training a Linear Regression model?
  - better suited for cases where there are a large number of features
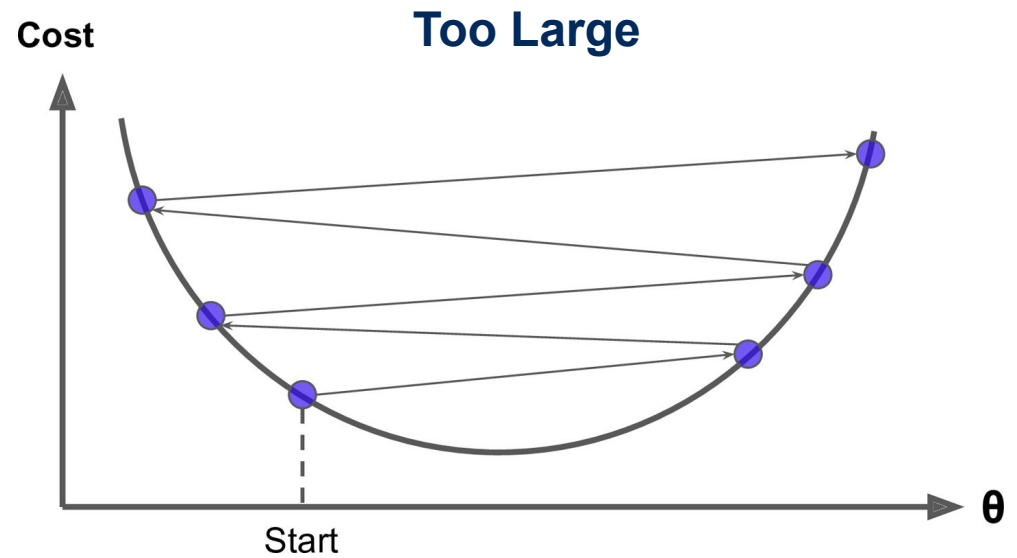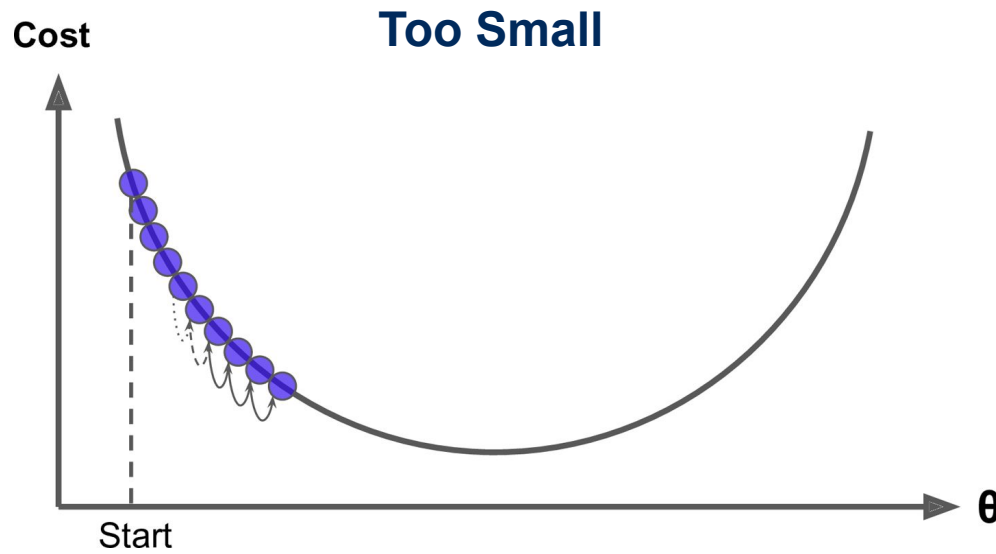 - too many training instances to fit in memory.

Module 5 – Section 2

Linear Regression Using Gradient Descent

Gradient Descent

- Start by filling θ with random values

- Then you improve it gradually, by decreasing the cost function one step at a time (e.g. the MSE)
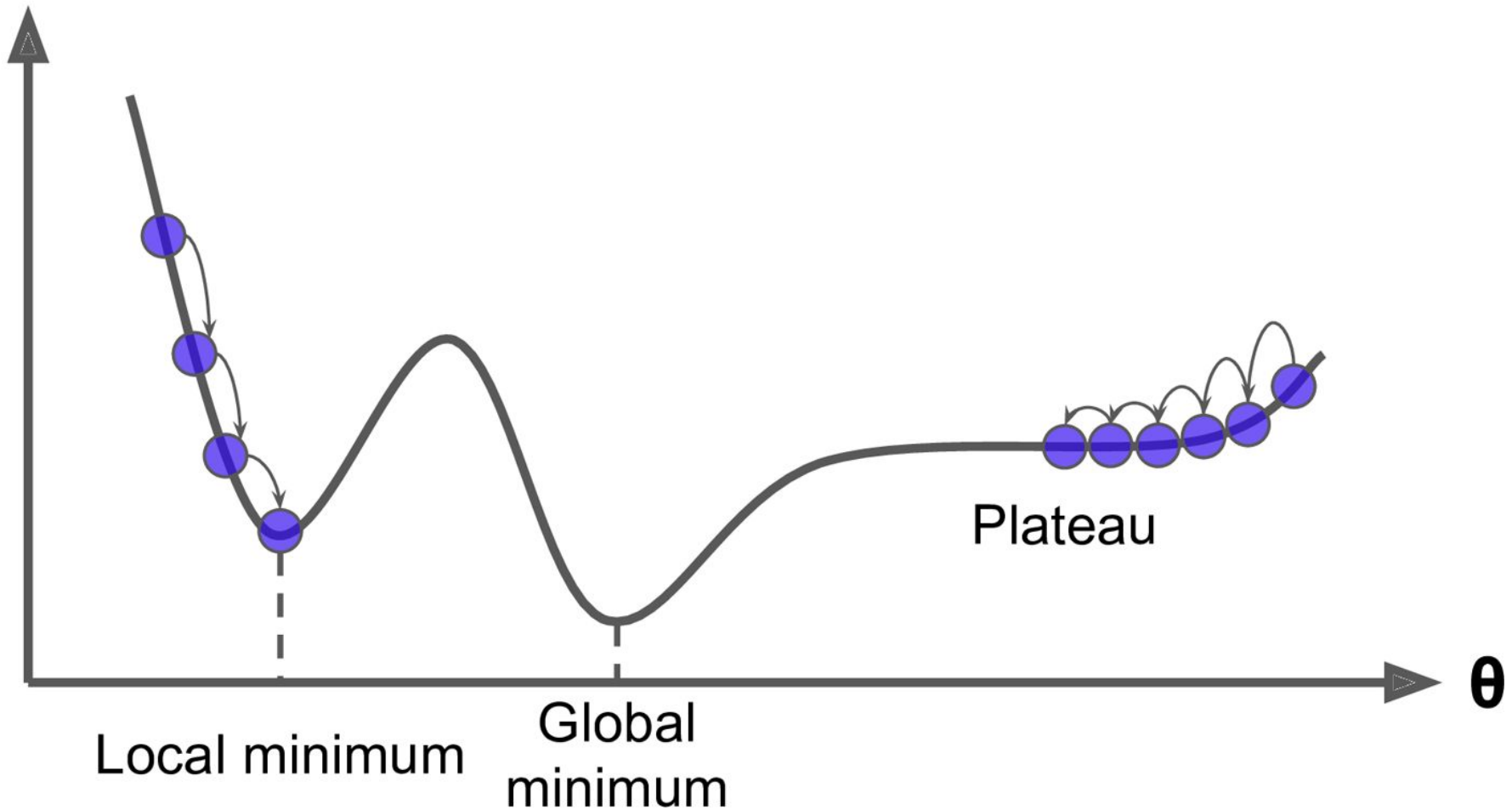
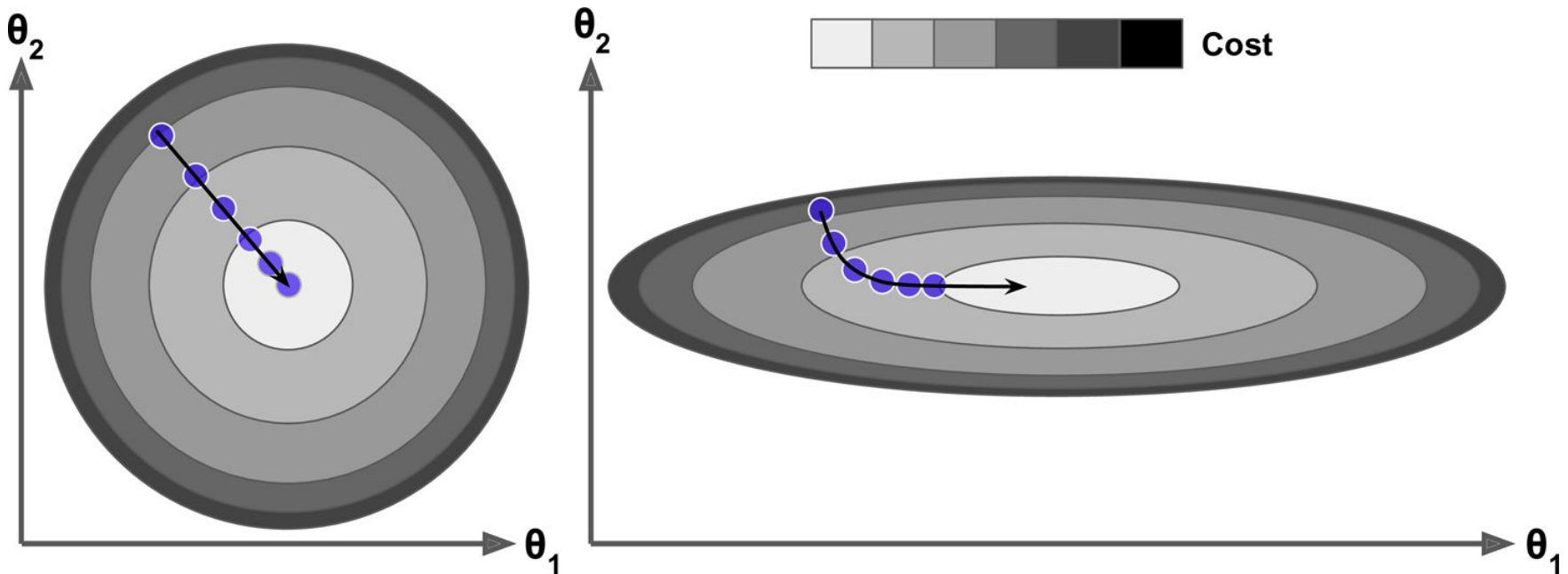- Algorithm converges to a minimum

Step Size

Too Small

Too Large

Local Minimum vs Global Minimum

MSE Cost Function is Convex

Ensure that all features have a similar scale or else it will take much longer to converge.

Batch Gradient Descent

- To implement Gradient Descent, you need to compute the **gradient** of the cost function with regards to each model **parameter** $\boldsymbol{\theta}_j$; m is the size of training data

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\theta) = \frac{2}{m} \sum_{i=1}^{m} (\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}) \, x_j^{(i)}$$

- Once you have the gradient vector, which points uphill, just go in the opposite direction to go downhill. This means subtracting $\nabla \theta$ MSE(θ) from θ

$$\theta^{(\text{next step})} = \theta - \eta \nabla_\theta \text{MSE}(\theta)$$

η Etha is learning rate which is a small number between 0-1

∇_θ Gradient vector is an array of gradient of cost function in respect to each feature

Batch Gradient Descent (cont'd)

$$\nabla_\theta \, \mathrm{MSE}\,(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \mathrm{MSE}\,(\theta) \\ \frac{\partial}{\partial \theta_1} \mathrm{MSE}\,(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \mathrm{MSE}\,(\theta) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T \cdot (\mathbf{X} \cdot \theta - \mathbf{y})$$

It uses the full dataset!

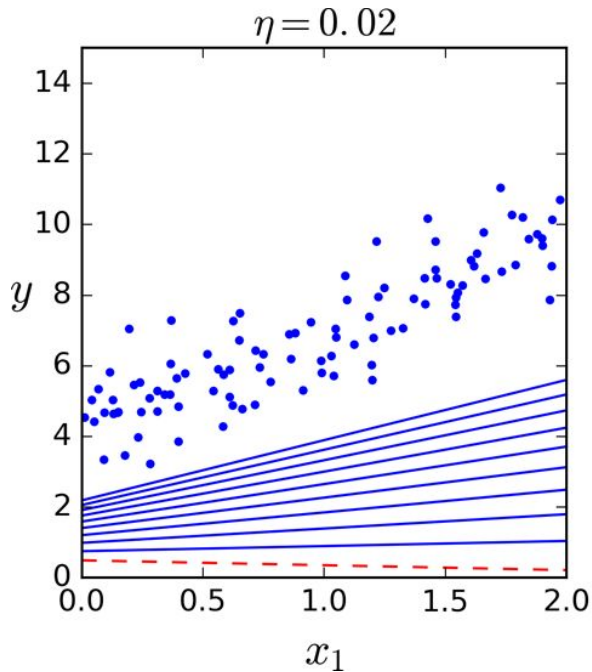```
eta = 0.1   # learning rate
n_iterations = 1000
m = 100

theta = np.random.randn(2,1)   # random initialization

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```
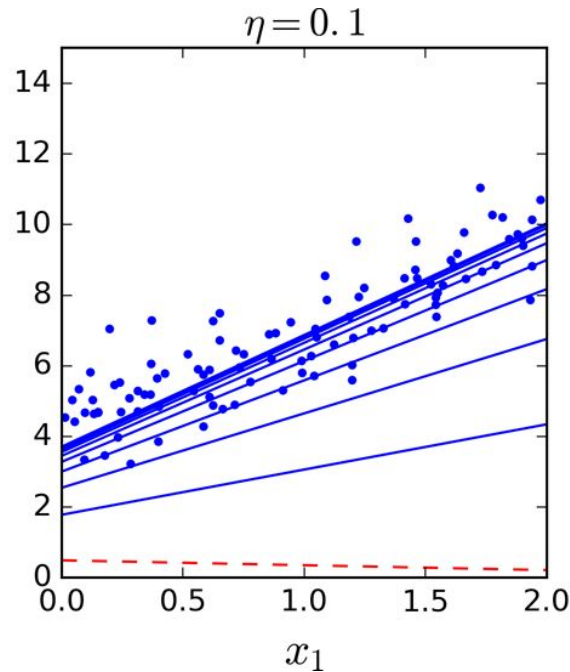
```
>>> theta
array([[4.21509616],
       [2.77011339]])
```
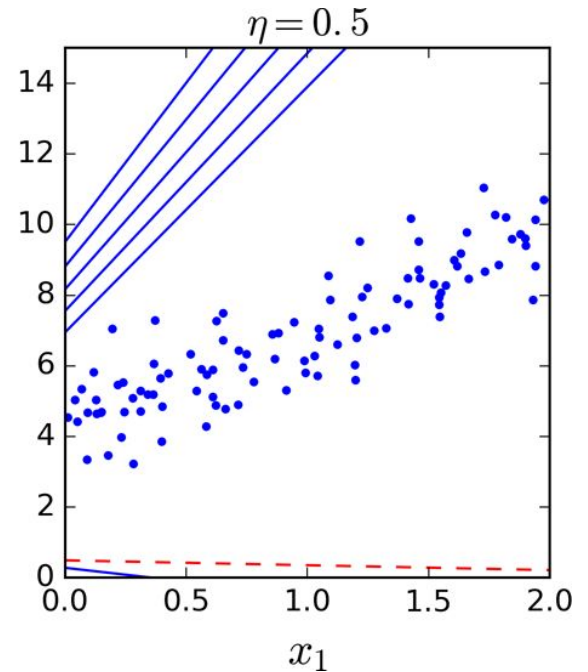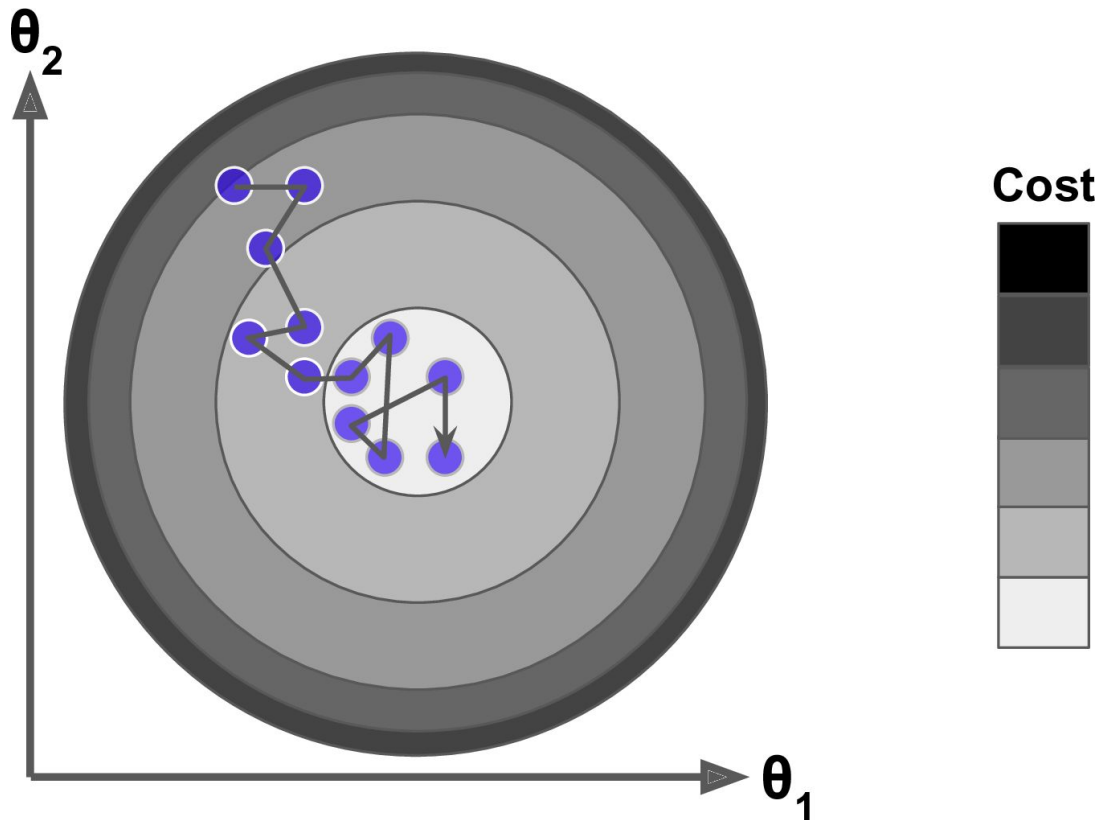
Same result as Normal Eq.

# Learning Rates

# Stochastic Gradient Descent

- Stochastic Gradient Descent picks a random instance in the training set at every step and computes the gradients based only on that single instance.
  - SGD is much **faster** since it has very little data to manipulate -> Makes it possible to train on huge training sets
  - SGD is much **less regular than Batch** Gradient Descent: instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down. Over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down
  - Once the algorithm stops, the final parameter values are **good, but not optimal**.

# Stochastic Gradient Descent (cont'd)



When the cost function is very irregular, Stochastic Gradient Descent has a better chance of finding the global minimum than Batch Gradient Descent does.

# Simulated Annealing

- Randomness is good to escape from local optima, but bad because it means that the algorithm can never settle at the minimum.

- One solution to this dilemma is to **gradually reduce the learning rate.**

  - The steps start out large (which helps make quick progress and escape local minima)

  - Then get smaller and smaller, allowing the algorithm to settle at the global minimum.
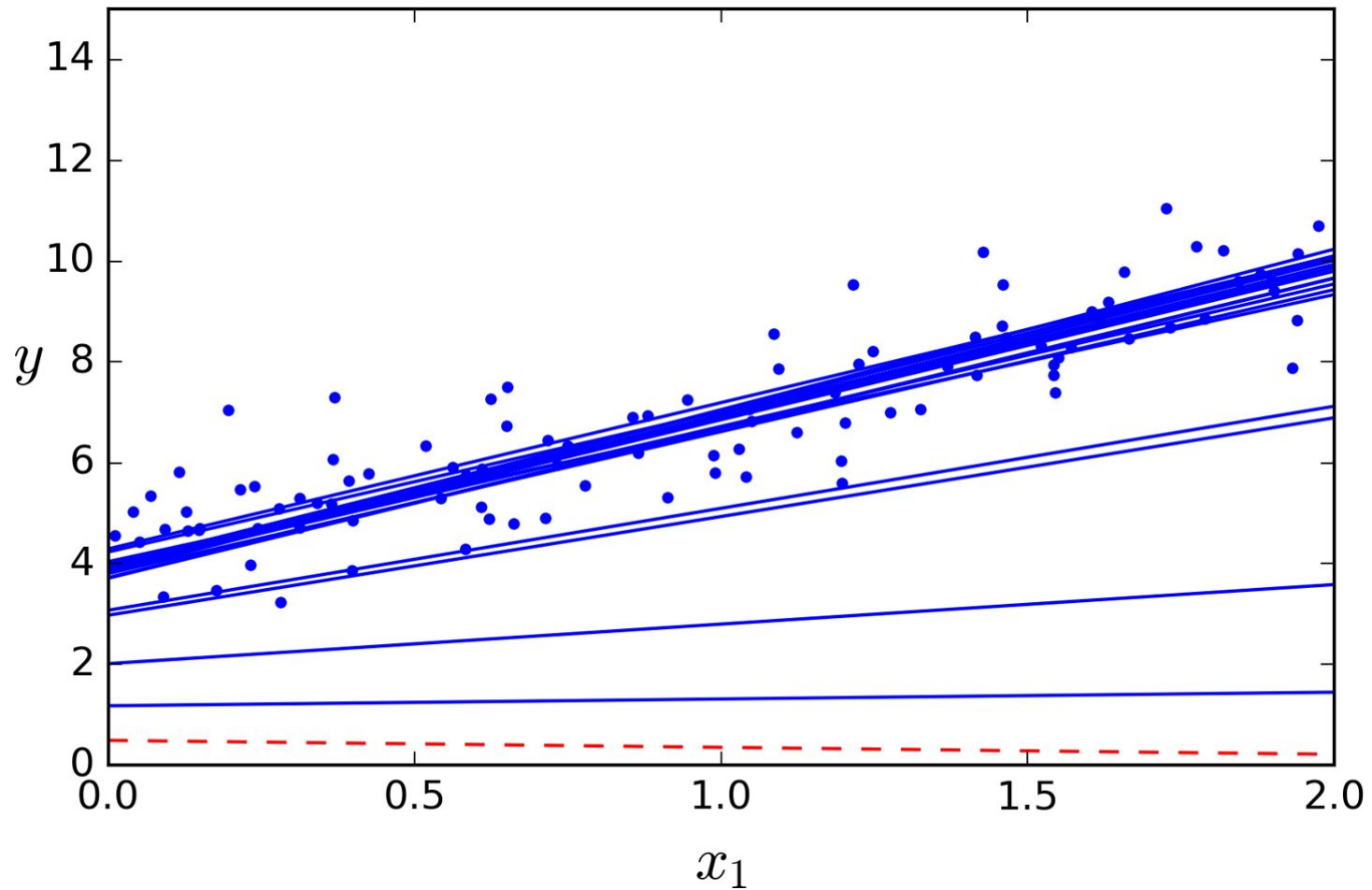
# Simulated Annealing (cont'd)

```python
n_epochs = 50
t0, t1 = 5, 50   # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1)   # random initialization

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
```

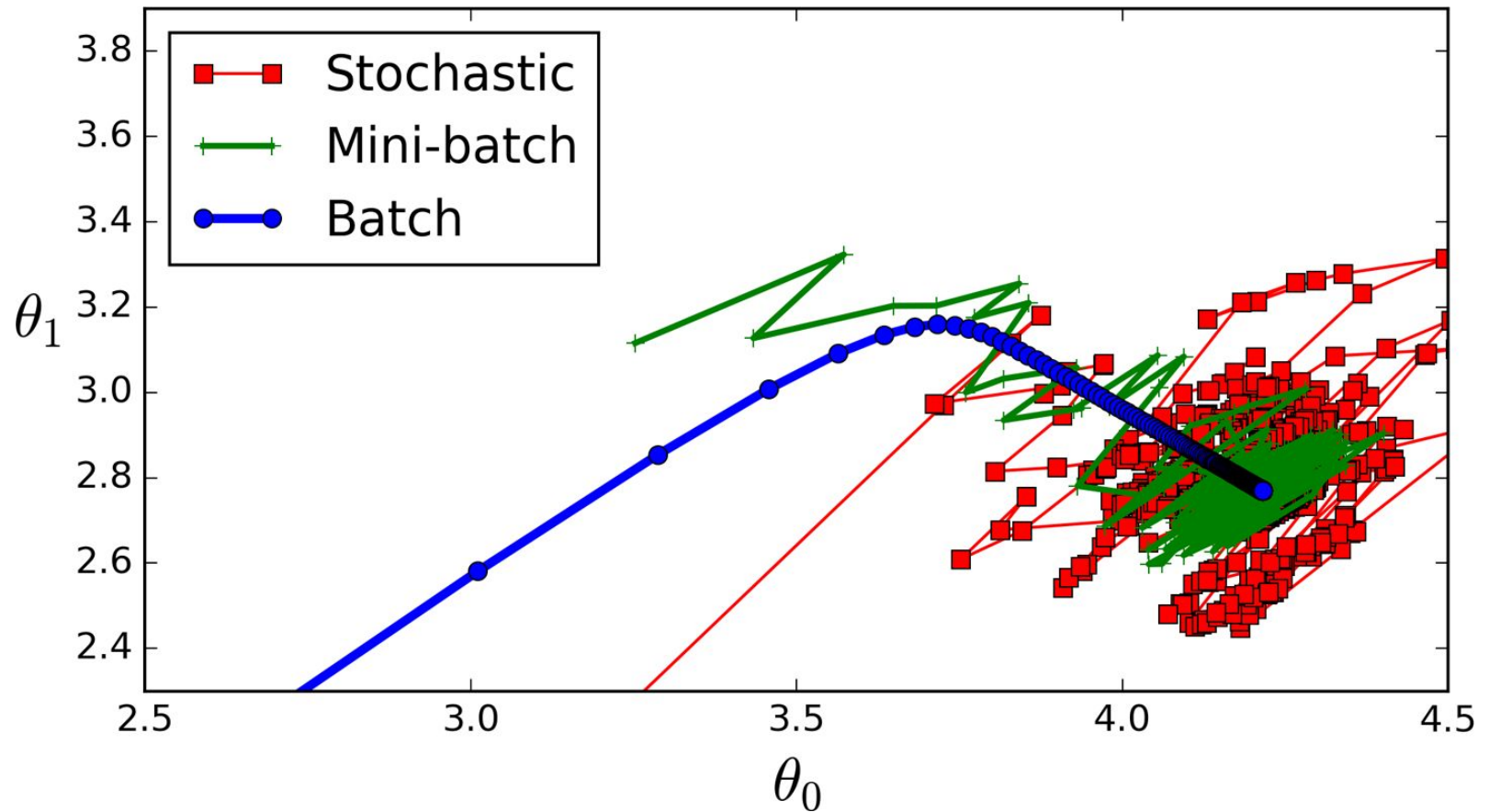# Simulated Annealing (cont'd)

SGD with Scikit-Learn

```python
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(n_iter=50, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
```

```python
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([ 4.16782089]), array([ 2.72603052]))
```

Mini-Batch Gradient Descent

- Mini-batch GD computes the gradients on small random sets of instances called mini-batches.

- The main advantage of Mini-batch GD over Stochastic GD is that you can get a performance boost from hardware optimization of matrix operations.
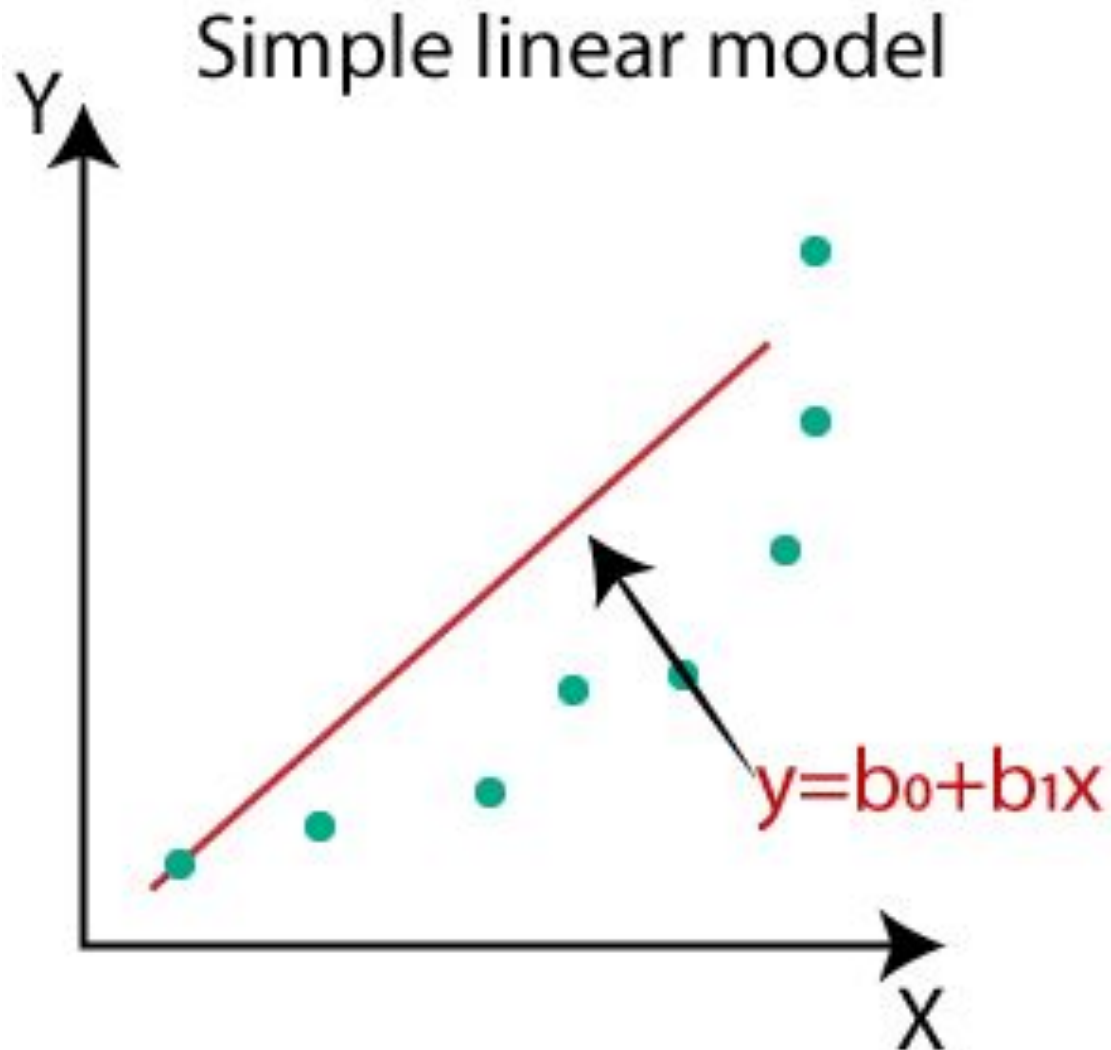
Gradient Descent Comparison

Algorithm	Large m	Out-of-core support	Large n	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	`LinearRegression`
Batch GD	Slow	No	Fast	2	Yes	n/a
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	n/a

Module 5 – Section 3

Non-Linear Regression

Interpret this Chart?

Simple linear model

$y = b_0 + b_1 x$

Polynomial Transformation Reduces Error

Simple linear model

$$y = b_0 + b_1 x$$

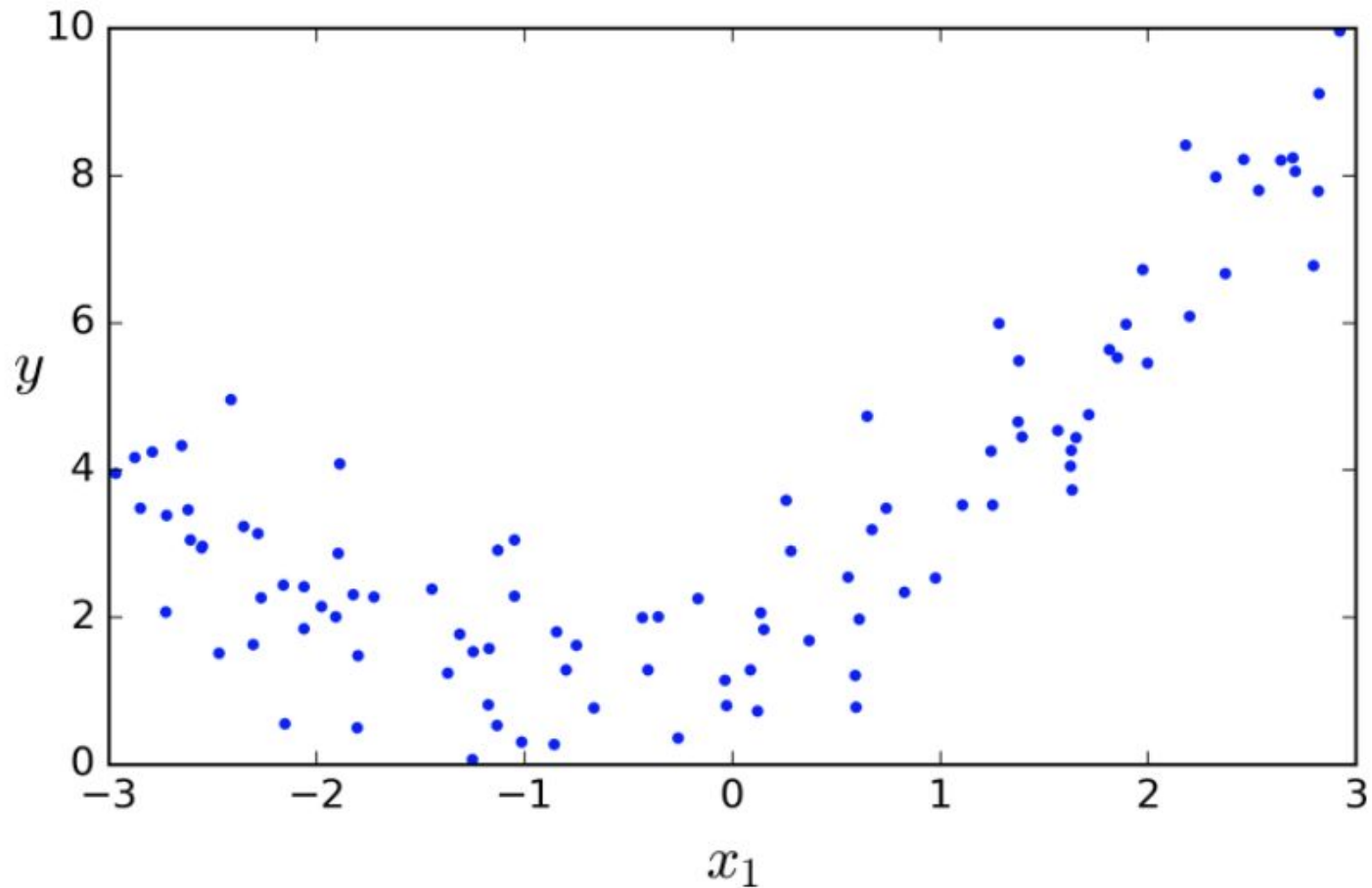### Polynomial model

$$y = b_0 + b_1 x_1 + b_2 x_1^2$$

Polynomial Regression

- What if your data is actually more complex than a simple straight line?
  - You can actually use a linear model to fit nonlinear data by adding powers of each feature as new features, then train a linear model on this extended set of features.

Non-linear Data

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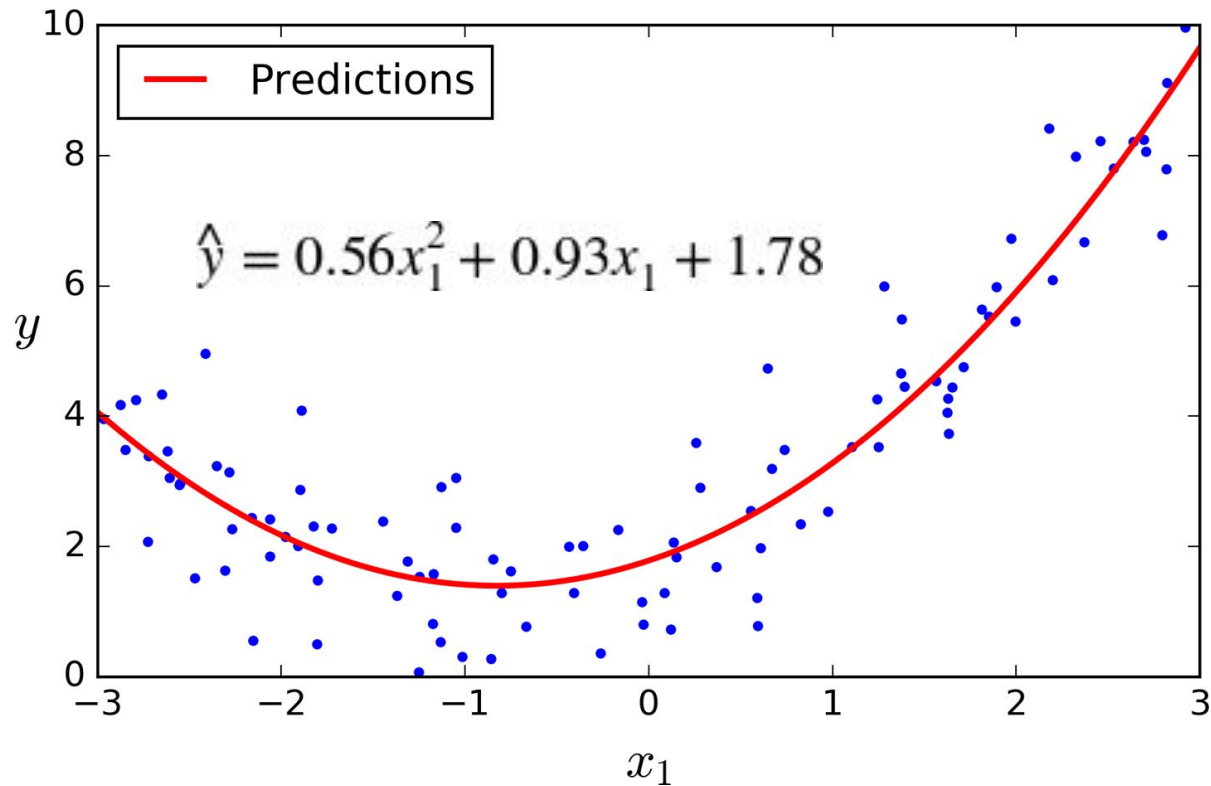
Regression

- $y = \theta_1 x^2 + \theta_2 x + c$

```
>>> 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])
```

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

Non-linear Predictions

$$\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$$

True data $\quad y = 0.5x_1^2 + 1.0x_1 + 2.0 + \text{Gaussian noise}$
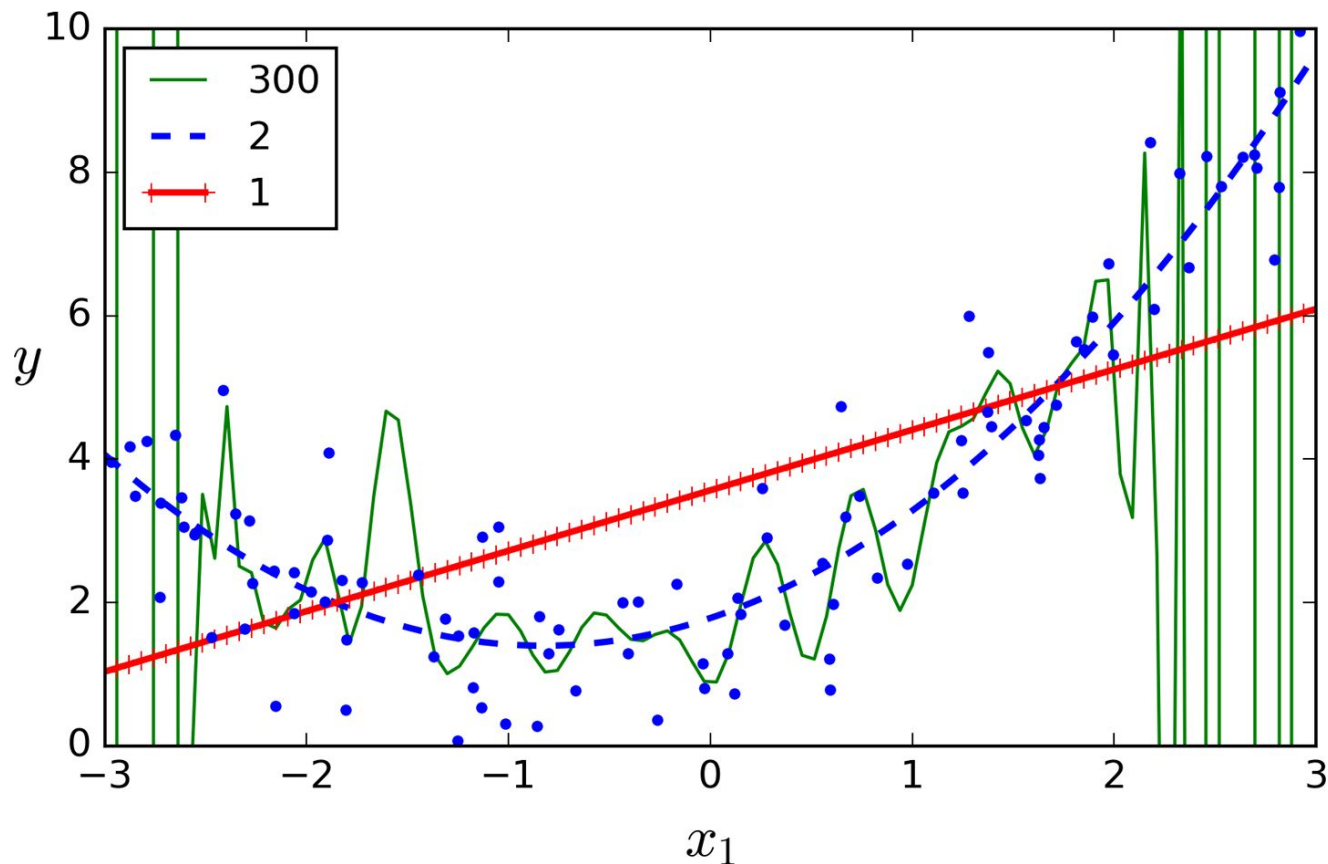
Module 5 – Section 4

Learning Curves

Model complexity & prediction error

- ML models can vary in complexity
  - Linear regression is the simplest model
- Non-linear transformations of the features can increase the complexity
- The generalization error (on test data) of a model depends on:
  - the **model** itself
  - the **noise of the data** (regardless of the model)

Model Complexity and Fitting!

- Look at below chart. Can you interpret the lines?
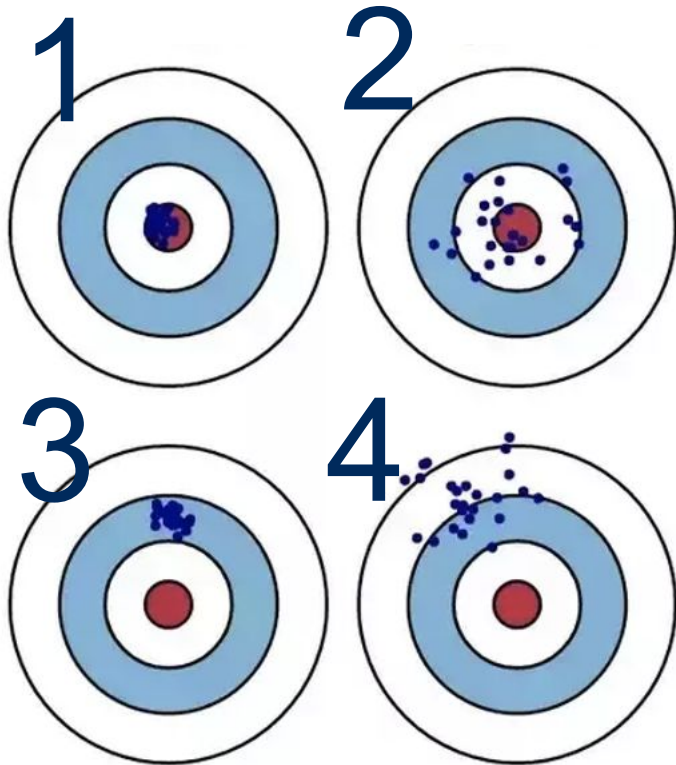
Bias and Variance

- **Bias**: error 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**: error due to the model's excessive sensitivity to small variations in the training data. A model with many degrees of freedom is likely to have high variance, and thus to overfit the training data.

- **Irreducible error**: due to the noisiness of the data itself. The only way to reduce this part of the error is to clean up the data.
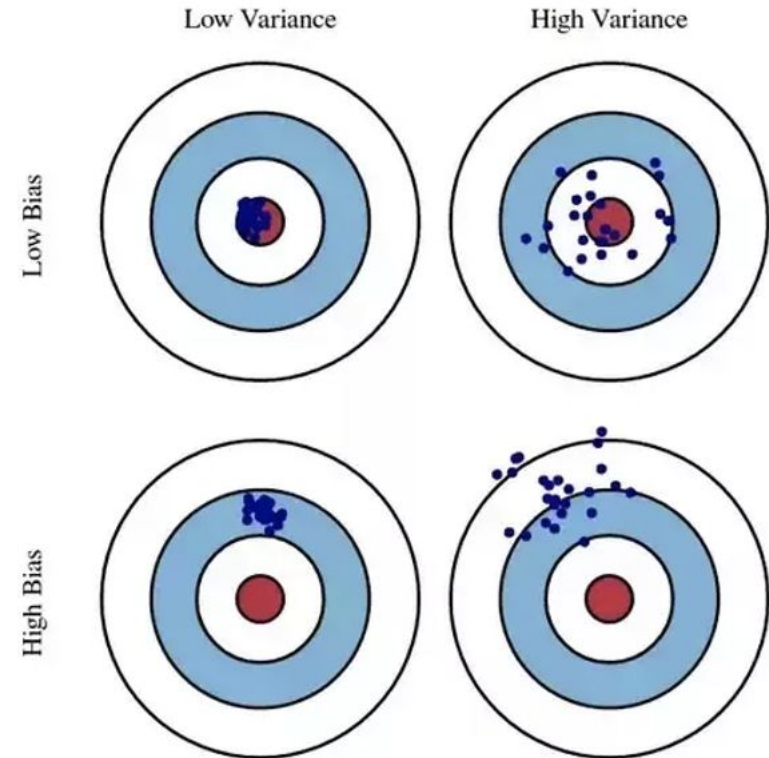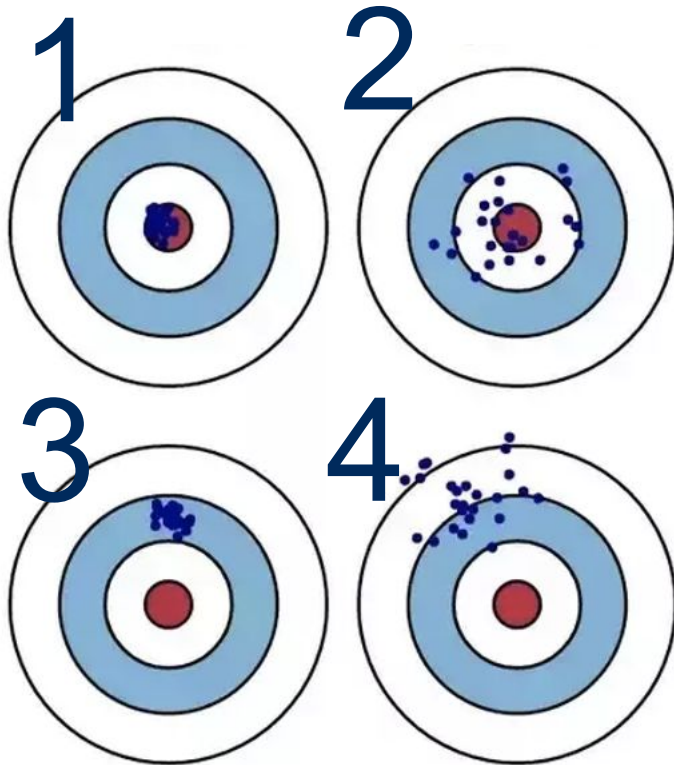
Bias and Variance Tradeoff

- The model's contribution to the error is decomposed into: Bias + Variance
- **Bias** is the error due to a model simpler than the phenomena being predicted
- **Variance** is the error due to a model that learns the nuances/noise of the data

- **Over**-**fitting** relates to high variance models
- **Under**-**fitting** relates to high bias models

Every model is overfit, but some are useful!

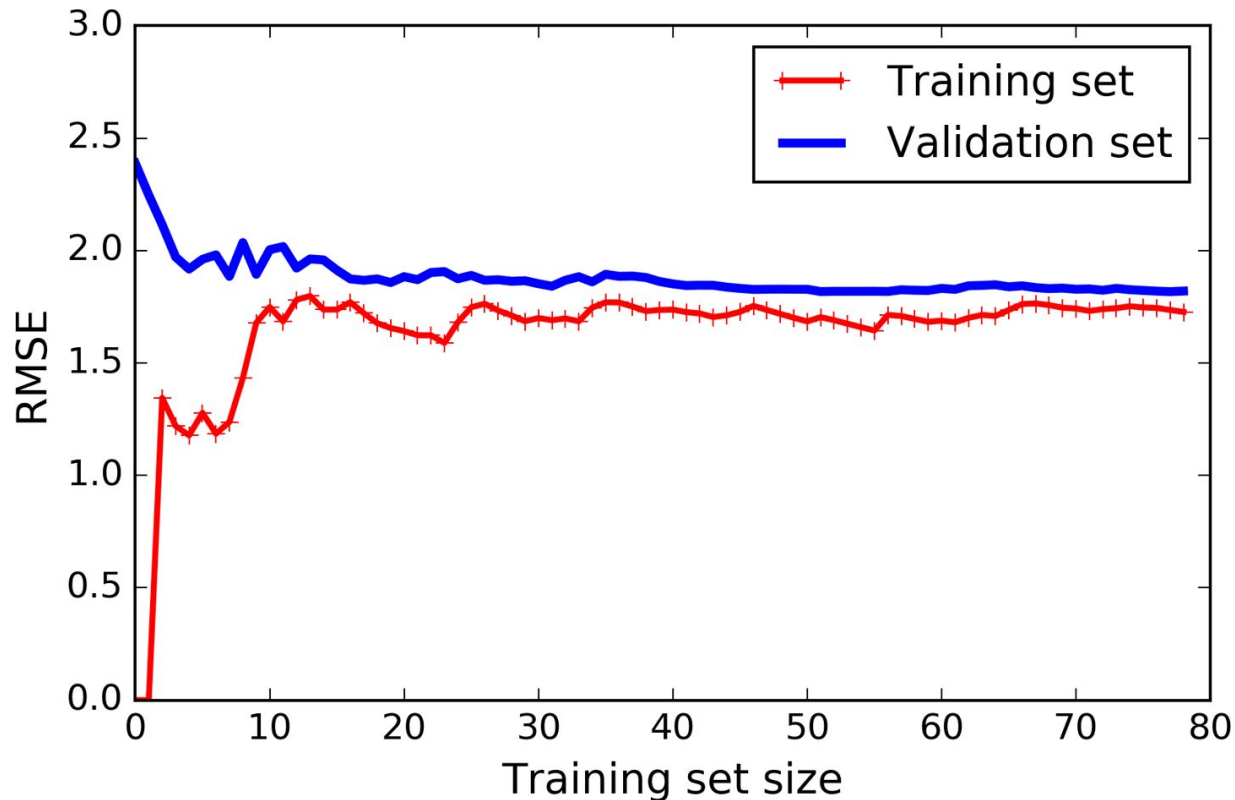# Q: Identify Low/High Bias/Variance

Q: Identify Low/High Bias/Variance

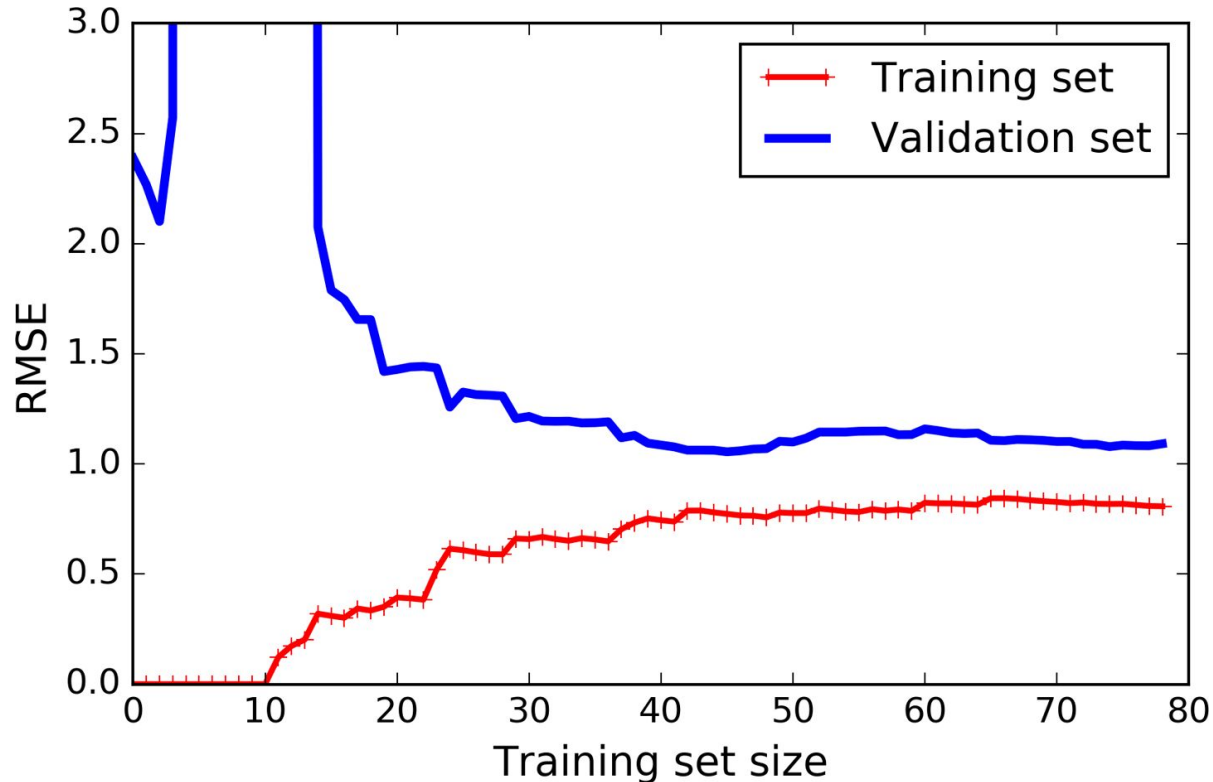Learning Curve can help you diagnose bias/variance

Learning Curves - Underfit

Look at below example and interpret the chart. Ideal RMSE should be less than < 1

These learning curves are typical of an underfitting model. Both curves have reached a plateau; they are close and fairly high.

Learning Curves - Overfit

One way to improve an overfitting model is to feed it more training data until the validation error reaches the training error.

Learning Curves in Python

```python
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_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")
```

Learning Curves (cont'd)

- Now let's look at the learning curves of a 10th-degree polynomial model on the same data

```python
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)
```
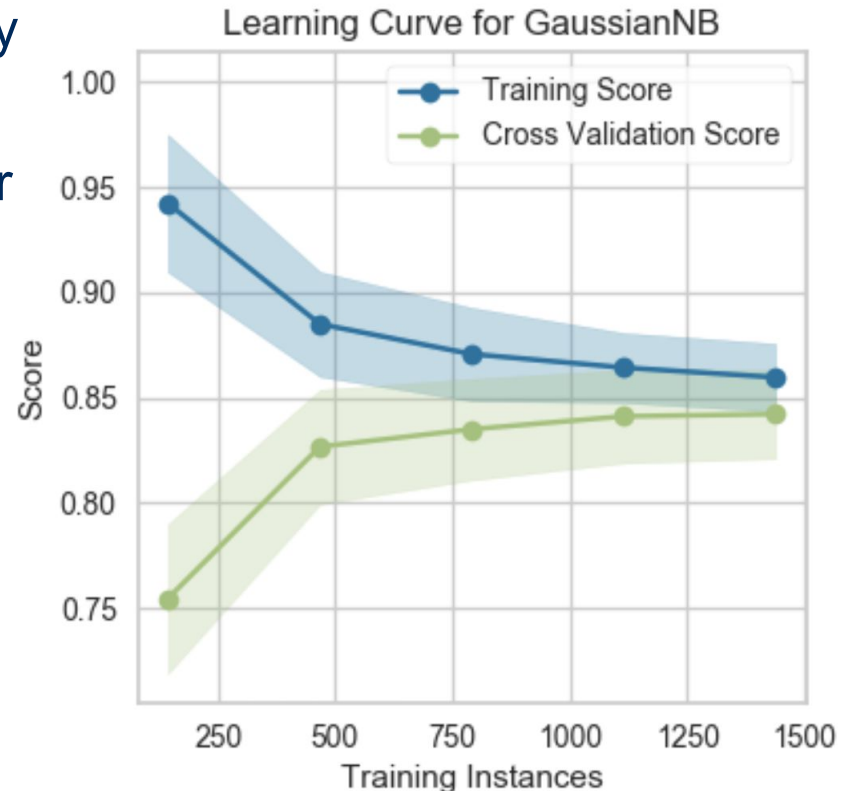
Diagnosing overfitting/underfitting

- Learning Curves helps figuring out if there is over-fit or under-fit
- The training and validation errors are plotted as a function of the training data size
- **Under-fitting models** have both high training and validation error
- Over-fitting models have low training error but high validation error
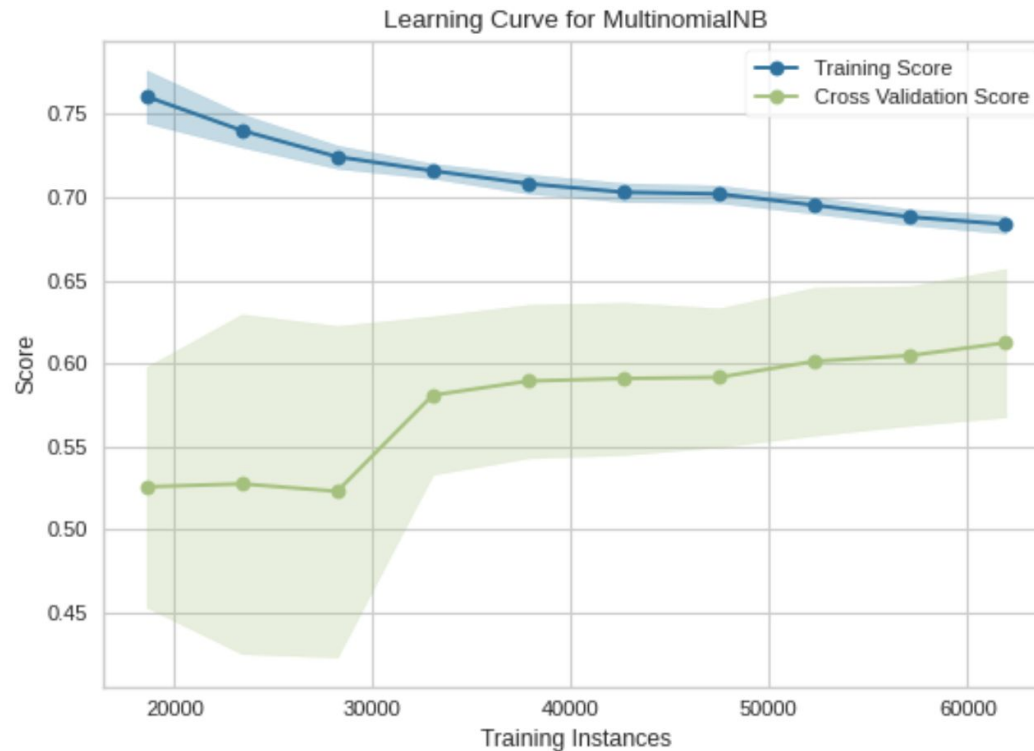
Learning curve and bias/variance diagnostics

- If the training and cross-validation scores converge together as more data is added (shown in the right figure), then the model will probably not benefit from more data.

- If the training score is much greater than the validation score then the model probably requires more training examples in order to generalize more effectively

- This is a good model, and if you need better score, you need **more signal** from either more or better features

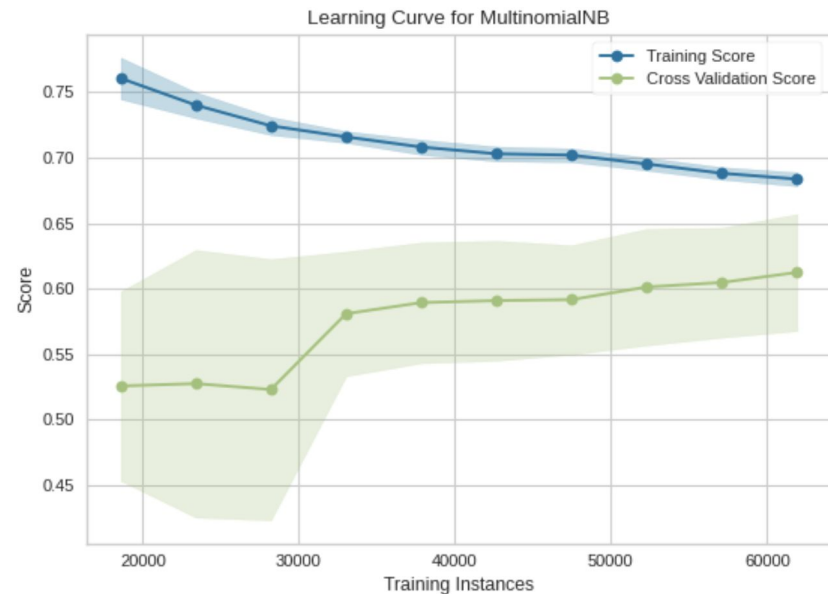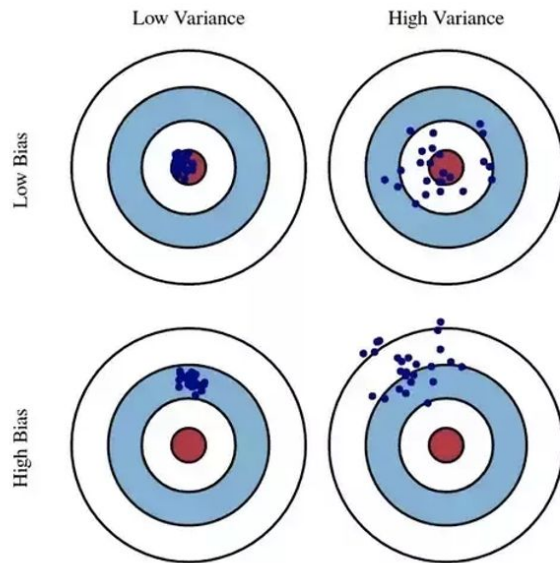
Learning Curve for GaussianNB

Learning curve and bias/variance diagnostics

- Below curves plotted with the MSE (mean scores). Shaded area around the line, show cross-validation variability and represent a standard deviation above and below the mean for all cross-validations.

Learning Curve for MultinomialNB

Learning curve and bias/variance diagnostics

- High variability in training score below, suggest **high bias**.
- High variability in validation score below, suggest **high variance**.
- This model is likely overfit and will benefit from more training samples

Learn more here

Module 5 – Section 5

Regularization and Ridge Regression

Regularization: An Overfitting Solution

- With too many features and their non-linear transformations, overfitting is possible

- Learning Curves can detect overfitting, but it is not a method to fix the problem

- Regularization is a method to fix it

- The idea is to penalize coefficients that do not majorly contribute in reducing the generalization error (cross-validation)

- This helps not using the model to learn just nuances exclusive to the training set

Regularization

- Digression: Learning Curves on other hyperparameter values (instead of the training set size) can be used to fix overfitting

- E.g.: Plot Learning Curves of train and valid error with respect to the number of hidden nodes in a 1-hidden layer NN

Ridge Regression

- The hyperparameter α controls how much you want to regularize the model.
  - If α = 0 then Ridge Regression is just Linear Regression.
  - If α is very large, then all weights end up very close to zero and the result is a flat line going through the data's mean.

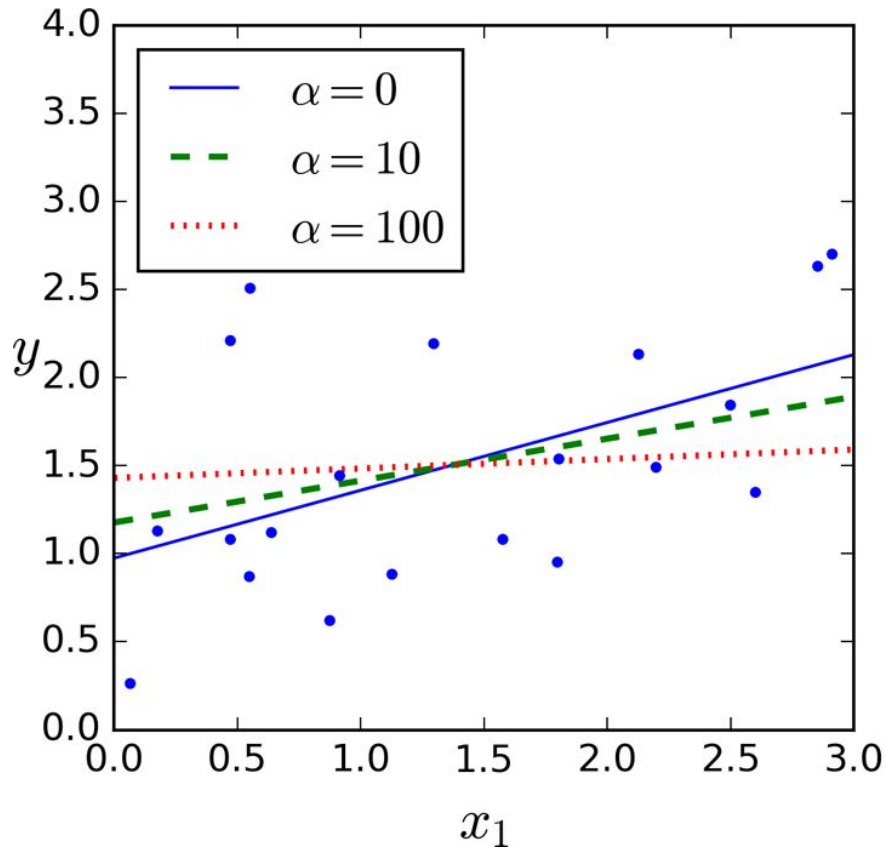$$J(\theta) = \text{MSE}(\theta) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$
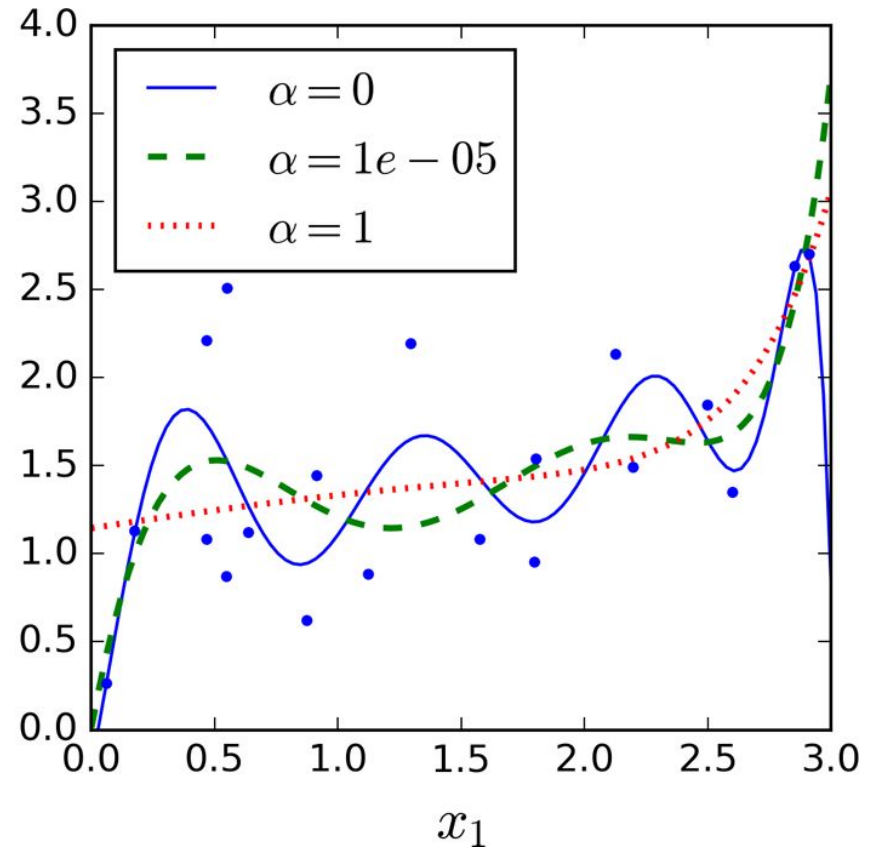
Ridge Regression (cont'd)

- The parameter alpha is calibrated via cross-validation (e.g. k-fold)

- Higher values of alpha increase bias

- Under all regularization methods, **scaling features** is recommended, since the solution is scale dependent

- Normal equation

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

Ridge Regression (cont'd)

Various alpha values. Linear features

Data is first expanded using PolynomialFeatures(degree=10)

Lasso Regression
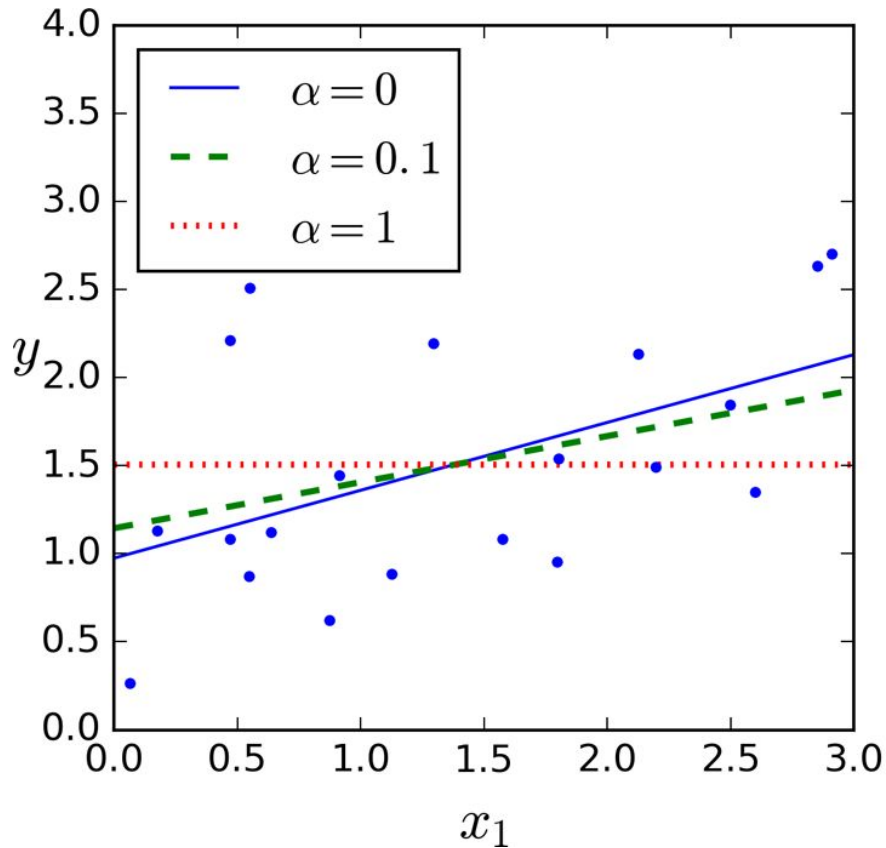
- Least Absolute Shrinkage and Selection Operator:
  - adds a regularization term to the cost function, but it uses the ℓ1 norm of the weight vector

- Cost function

$$J(\theta) = \text{MSE}(\theta) + \alpha \sum_{i=1}^{n} | \theta_i |$$
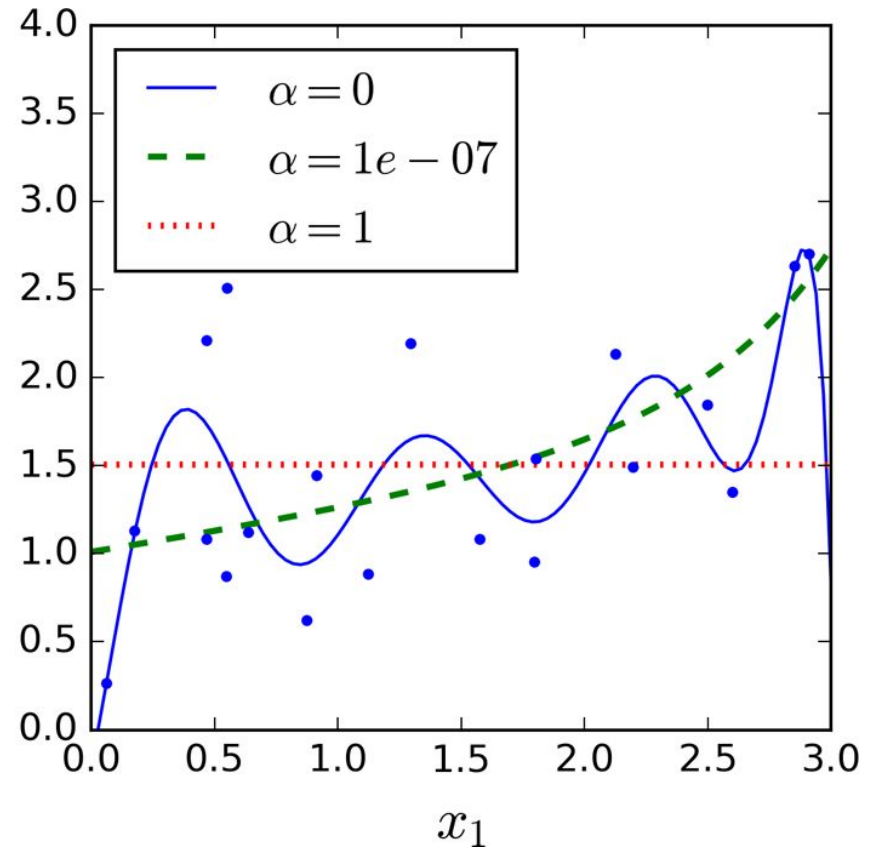
UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Feature Selection with Lasso

- An important characteristic of Lasso Regression is that it tends to completely eliminate the weights of the least important features

  - Lasso Regression automatically performs feature selection and outputs a sparse model
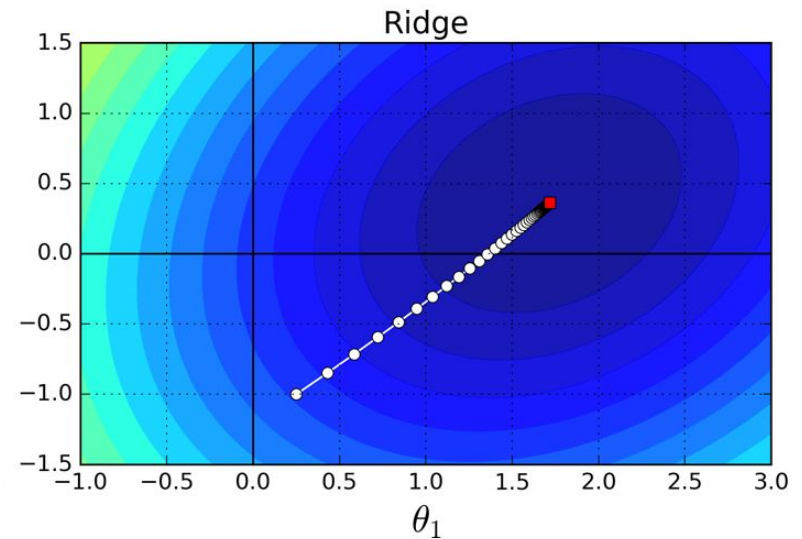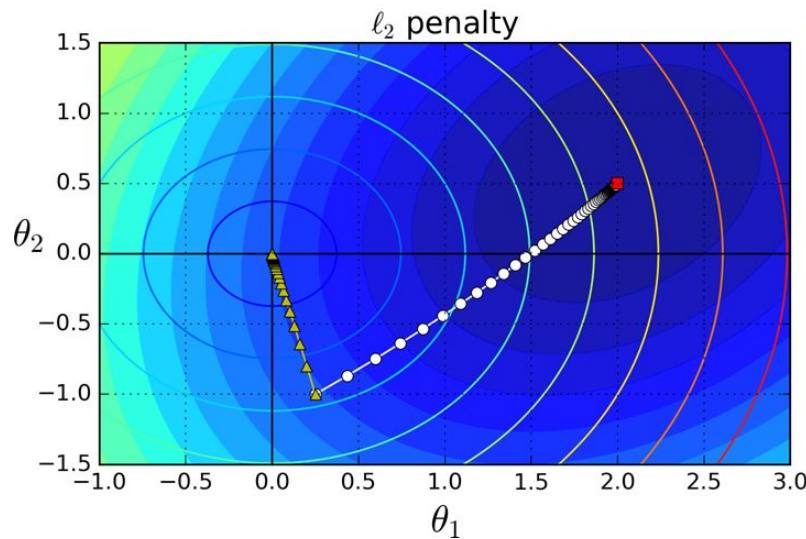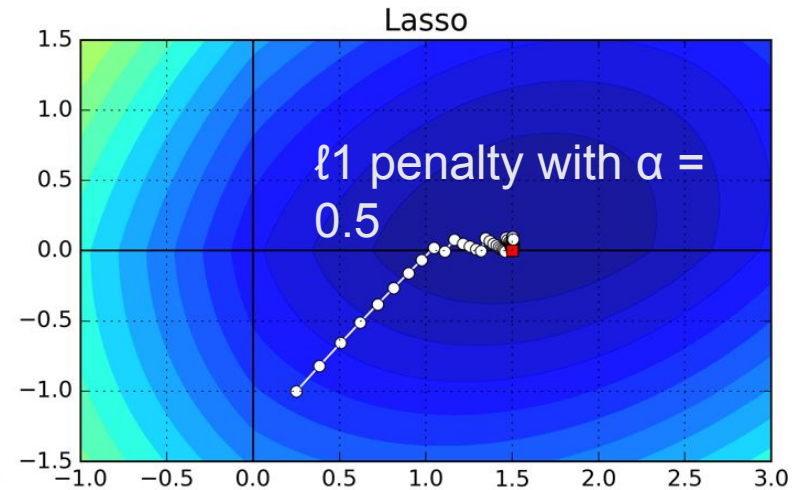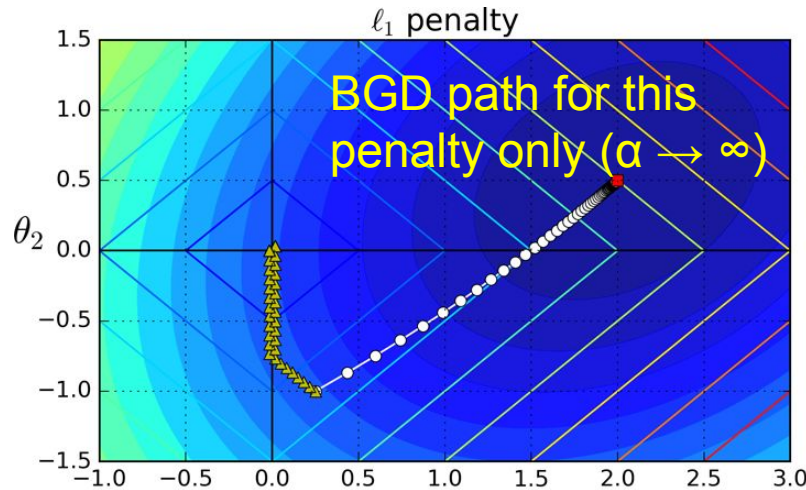
Lasso Regression

Various alpha values. Linear features

Data is first expanded using PolynomialFeatures(degree=10)
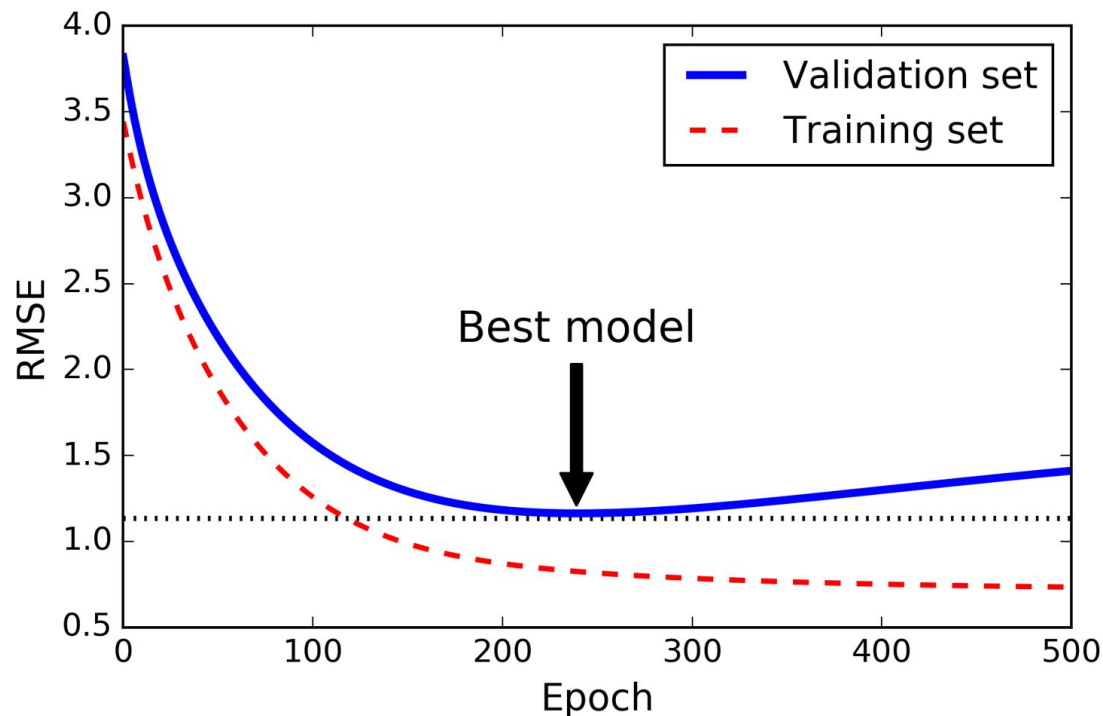
L1 vs. L2 Penalty: Impact of Alpha

Elastic Net

- Middle ground between Ridge Regression and Lasso Regression.
  - The regularization term is a mix of both
  - You can control the mix ratio r.
    - When r = 0, Elastic Net is equivalent to Ridge Regression,
    - When r = 1, it is equivalent to Lasso Regression

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

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Early Stopping

- Stop training as soon as the validation error reaches a minimum
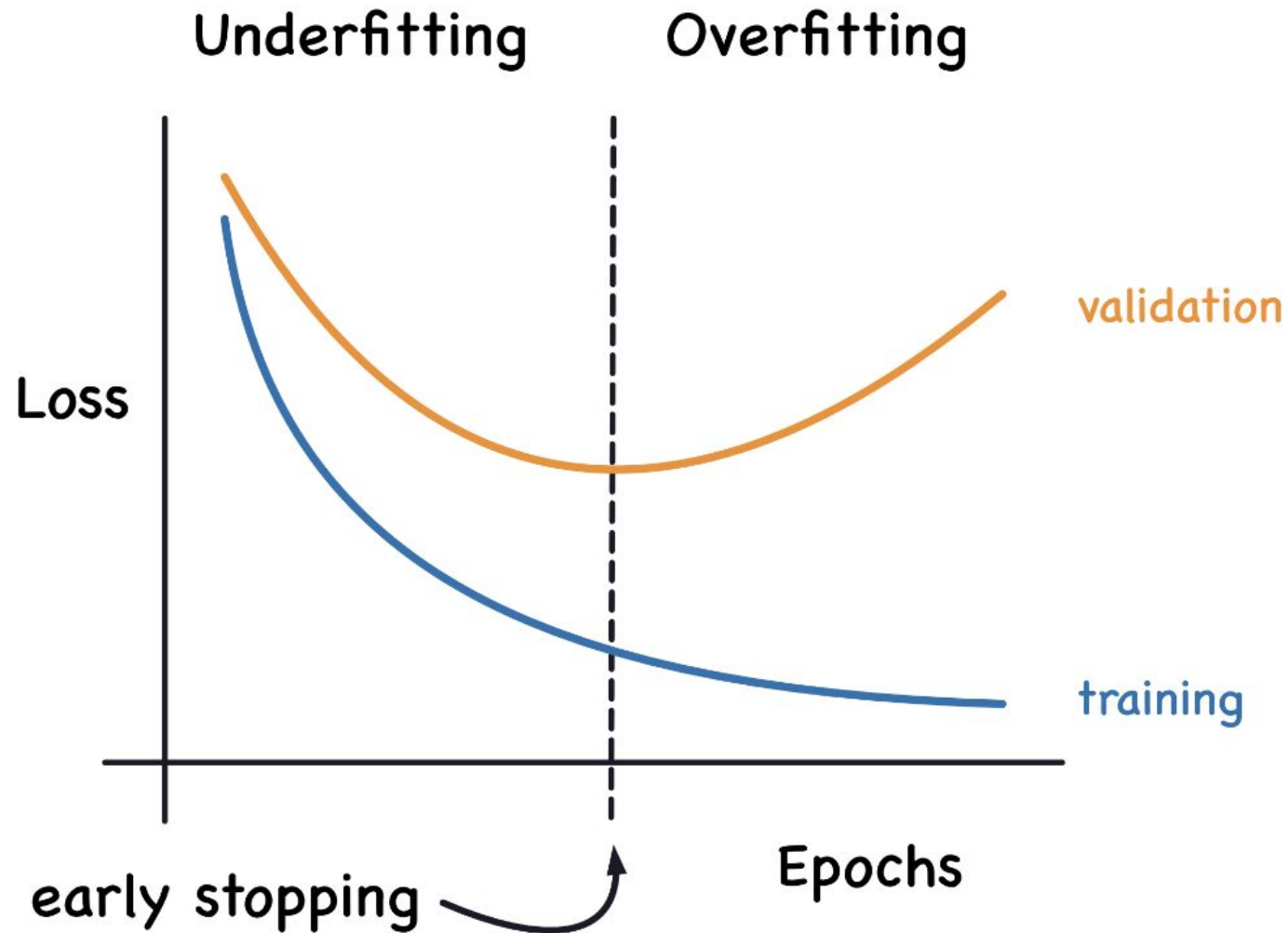
Early Stopping

```python
from sklearn.base import clone

sgd_reg = SGDRegressor(n_iter=1, 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_predict, y_val)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = clone(sgd_reg)
```

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Putting them together!

Module 5 – Section 6

Logistic Regression

Logistic Regression

- A linear model for binary classification
- It learns from examples with labels {0,1}
- Predicts probabilities of new examples having label 1
- A threshold on the probability yields a binary classification
- It is exactly the algorithm for a binary classification Neural Network with no hidden layers
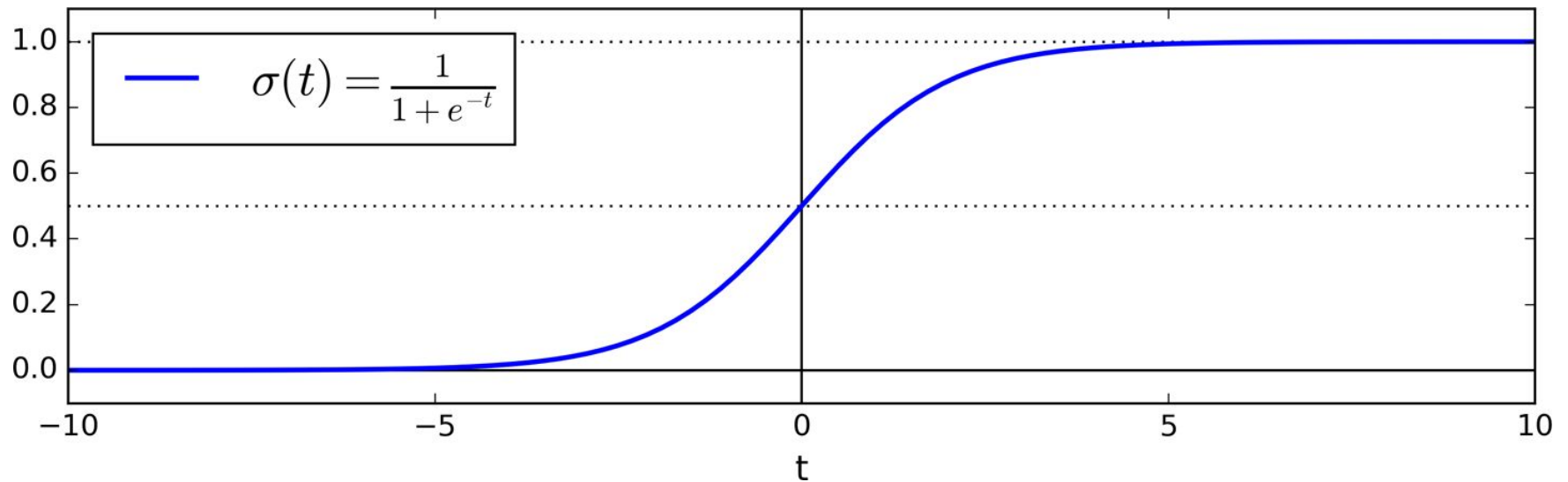
Logistic Regression (cont'd)

- The model computes a weighted sum of the input features (plus a bias term):
  - but instead of outputting the result directly, it outputs the logistic of this result

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

  - where

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

Logistic Function

Cost Function

- The objective of training is to set the parameter vector θ so that the model estimates high probabilities for positive instances (y = 1) and low probabilities for negative instances (y = 0)

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

- over the whole training set :

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

Logistic Regression (cont'd)

- The cost function is convex, so Gradient Descent is guaranteed to find the global minimum (if the learning rate is not too large and you wait long enough).
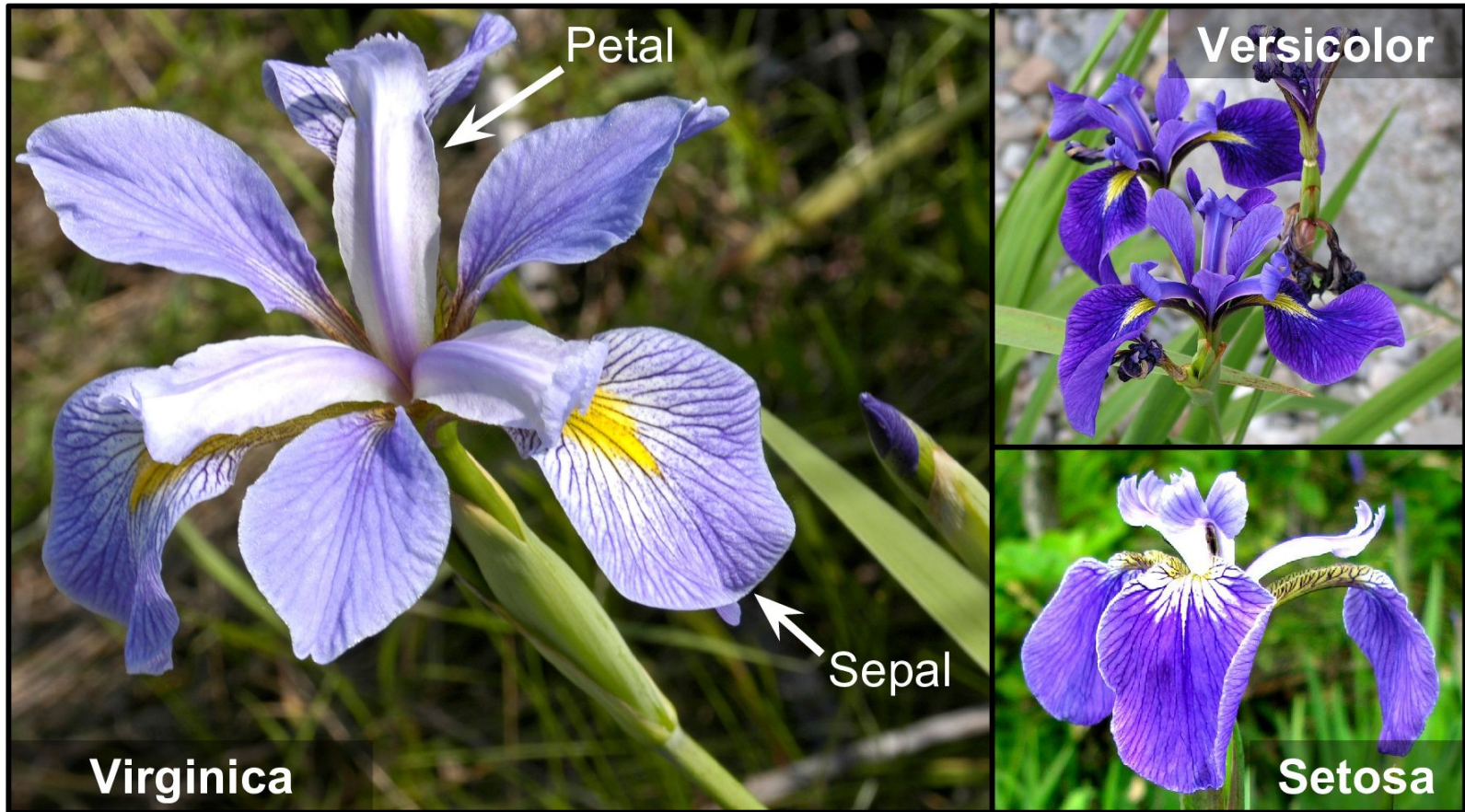
$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(\sigma(\theta^T \cdot \mathbf{x}^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

Logistic Regression (cont'd)

- The solution of logistic regression finds a hyperplane separating both classes
- The hyperplane is the decision boundary
- Points in the decision boundary:
  - If the product is negative, the class is 0, if it positive the class is 1
  - The sigmoid function maps the product to [0,1] (a probability)
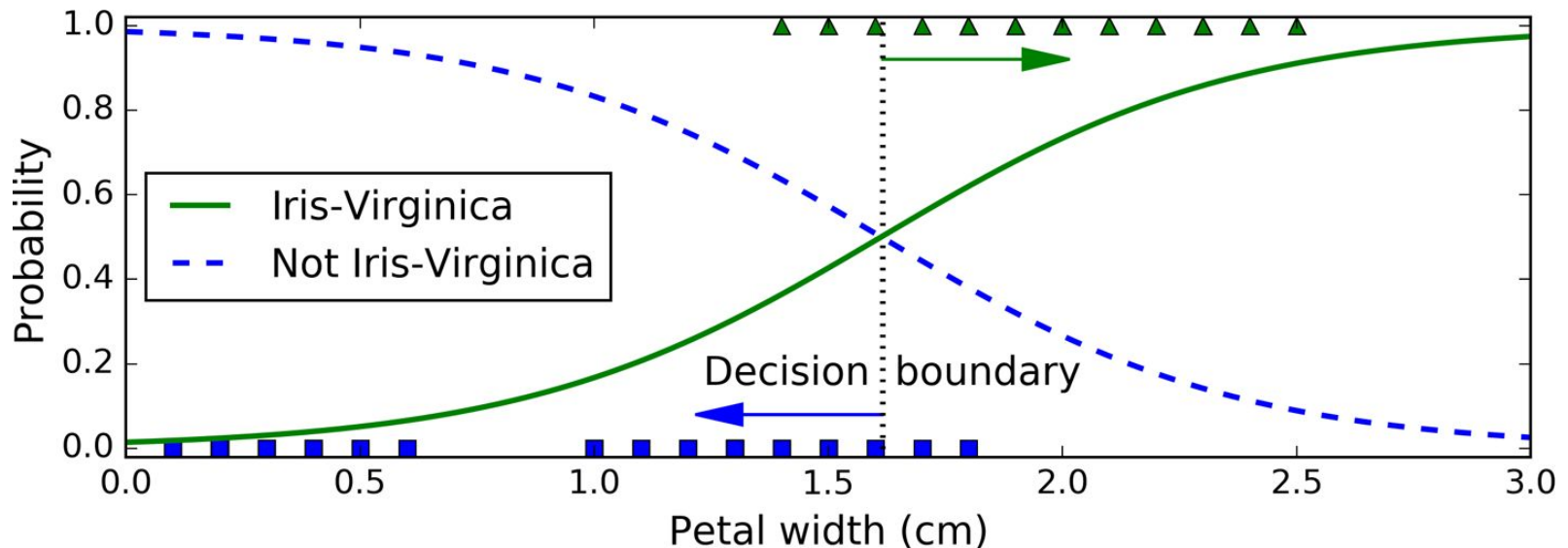
1-d Logistic Example

1-d Logistic Example

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target_names', 'feature_names', 'target', 'DESCR']
>>> 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)
```
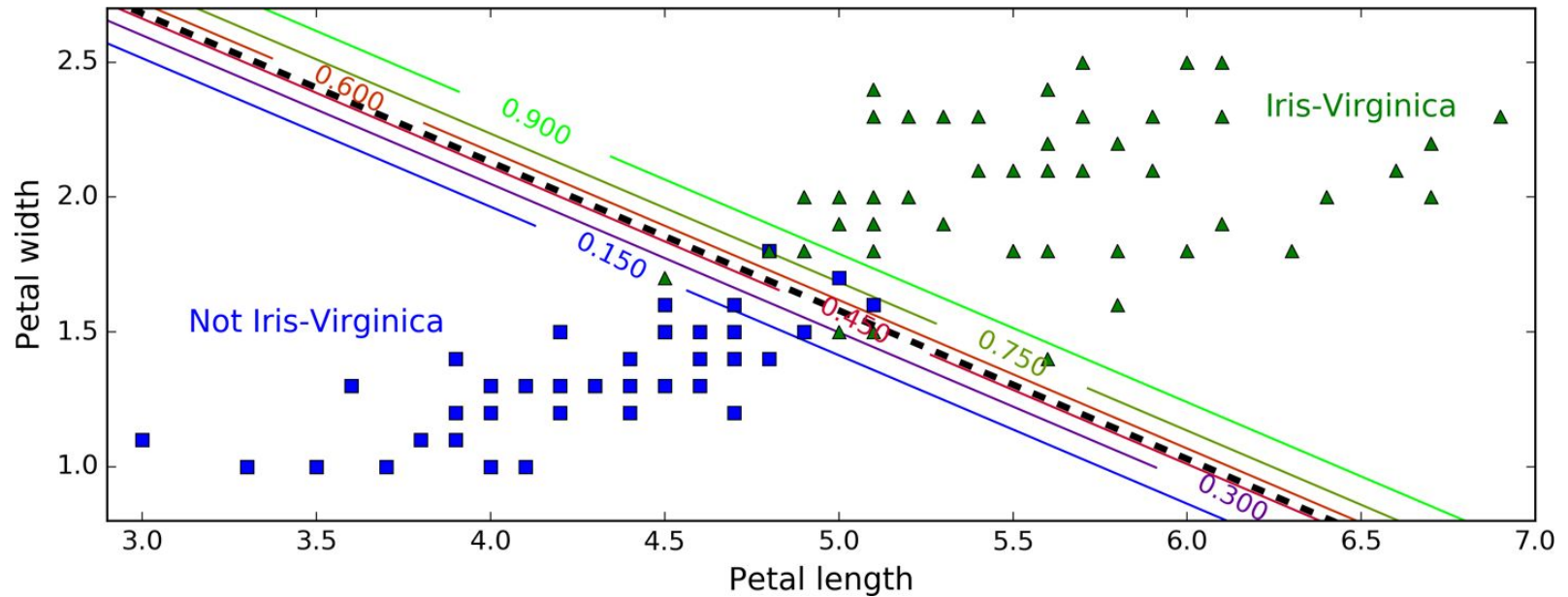
1-d Logistic Example

```python
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")
```
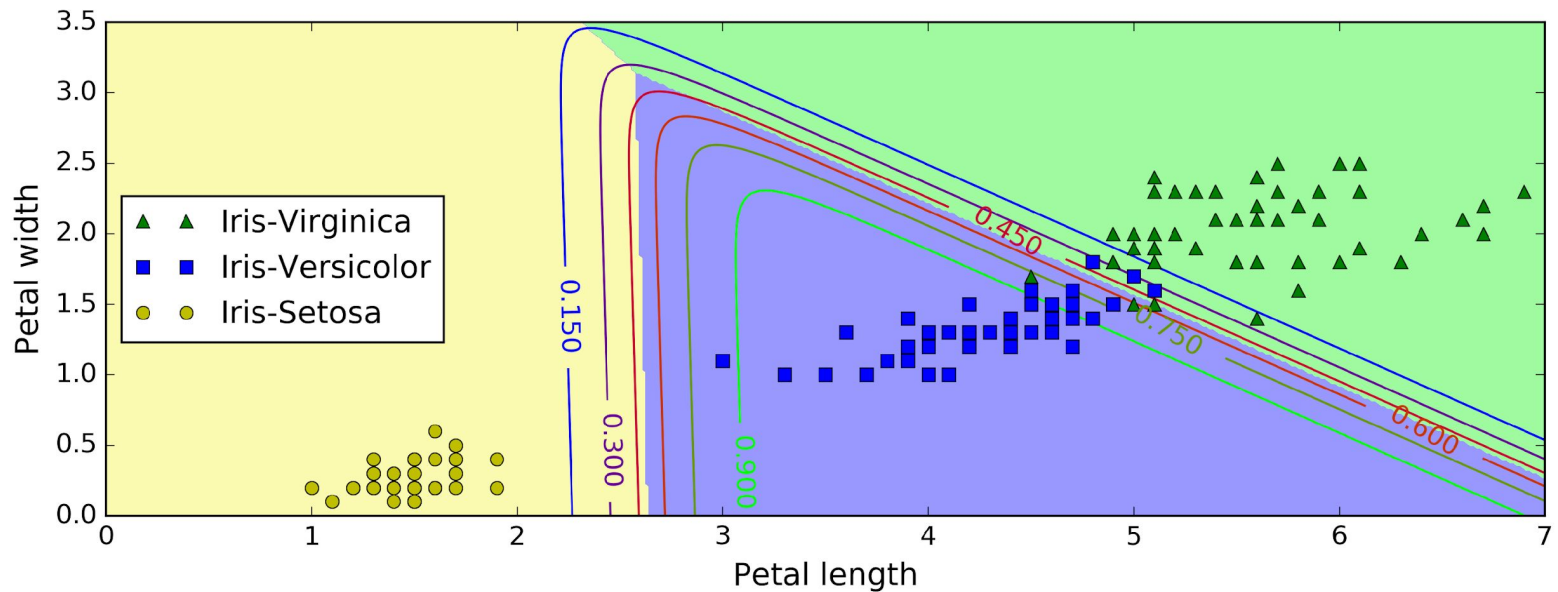
2-d Logistic Example

Softmax Regression

- LR model can be generalized to support multiple classes directly. This is called Softmax Regression, or Multinomial Logistic Regression.

- When given an instance x, the Softmax Regression model first computes a score $s_k(x)$ for each class k, then estimates the probability of each class by applying the softmax

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

Softmax Regression (cont'd)

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

Module 5 – Section 7

Resources and Wrap-up

Resources

- Hands-On Machine Learning with Scikit-Learn and TensorFlow

Homework

- Complete the notebook in the assignments section for this week

- Submit your solution here

  - https://goo.gl/forms/VKD7Zwu54oHjutDc2

- Make sure you rename your notebook to

  - W5_UTORid.ipynb

  - Example: W5_adfasd01.ipynb

Next Class

- Support Vector Machines
  - Chapter 5 textbook

Follow us on social

Join the conversation with us online:

facebook.com/uoftscs

@uoftscs

linkedin.com/company/university-of-toronto-school-of-continuing-studies

@uoftscs

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Any questions?

Thank You

Thank you for choosing the University of Toronto School of Continuing Studies