UEMH3163/UEECS2053/UECS2153 – Artificial Intelligence

# Regularization And Gradient Descent

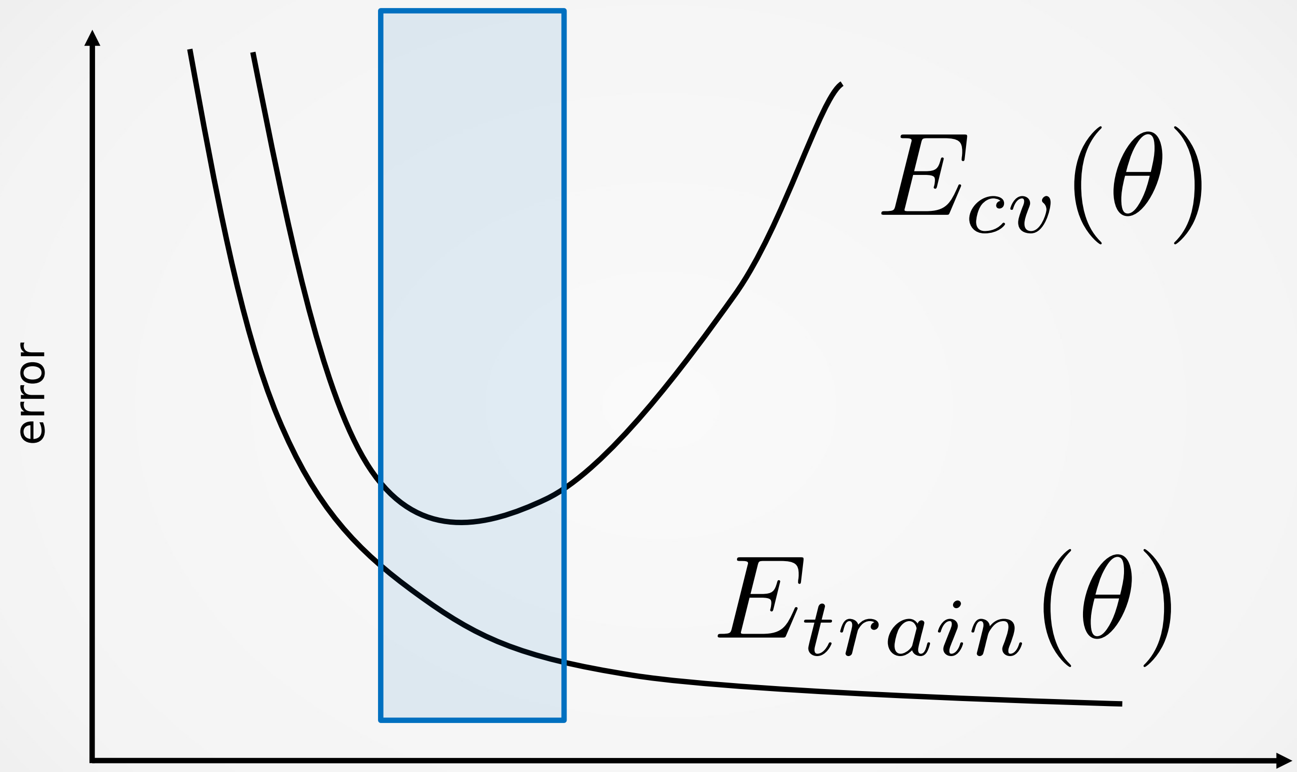
UTAR
UNIVERSITI TUNKU ABDUL RAHMAN

# Learning Objectives

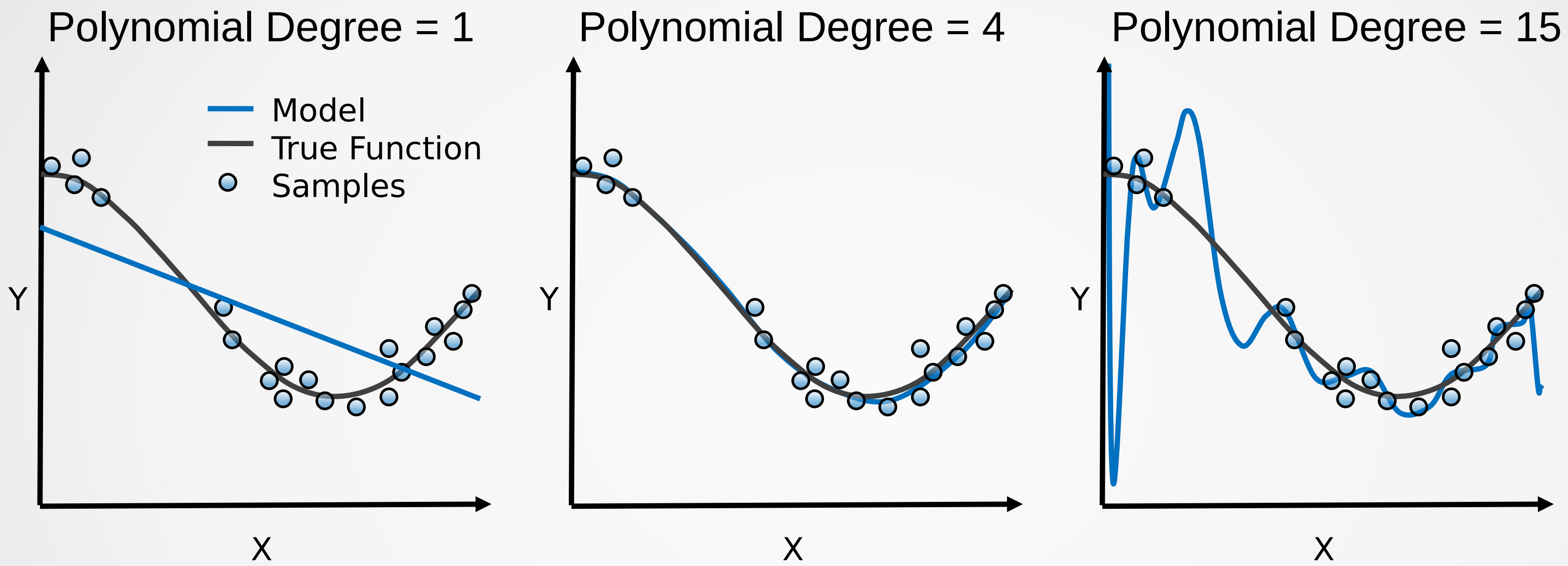
After completing this lecture, you will be able to:-

- Describe the function and purpose of regularization
- Apply L1, L2, and a combination of both regularizations to a Linear Regression problem
- Optimize hyperparameters using validation sets
- Use Feature Selection to simplify models
- Describe the gradient descent optimizer
- Use stochastic gradient descent and mini-batch gradient descent to speed up the gradient descent optimizer

# Regularization

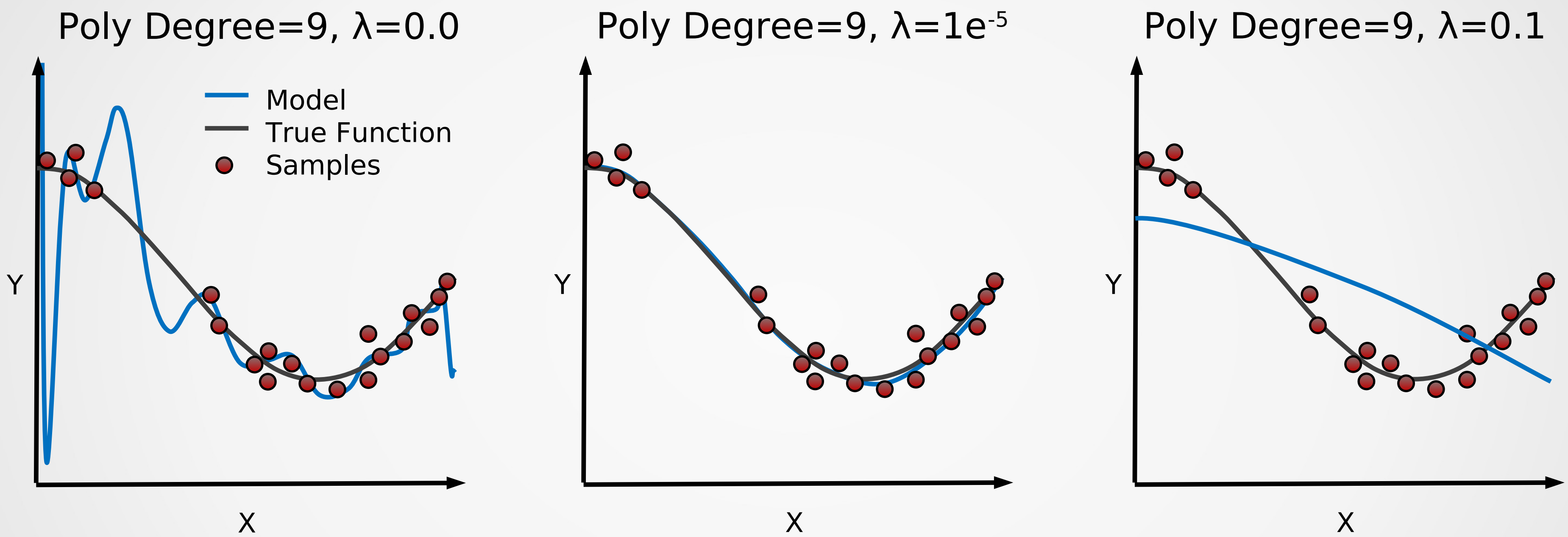


# Regularization




$$J(\beta_0, \beta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left( \left( \beta_0 + \beta_1 x^i_{obs} \right) - y^i_{obs} \right)^2$$

# Ridge Regression




$$J(\beta_0, \beta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left( \left( \beta_0 + \beta_1 x_{obs}^{i} \right) - y_{obs}^{i} \right)^2 + \lambda \sum_{j=1}^{k} \beta_j^2$$

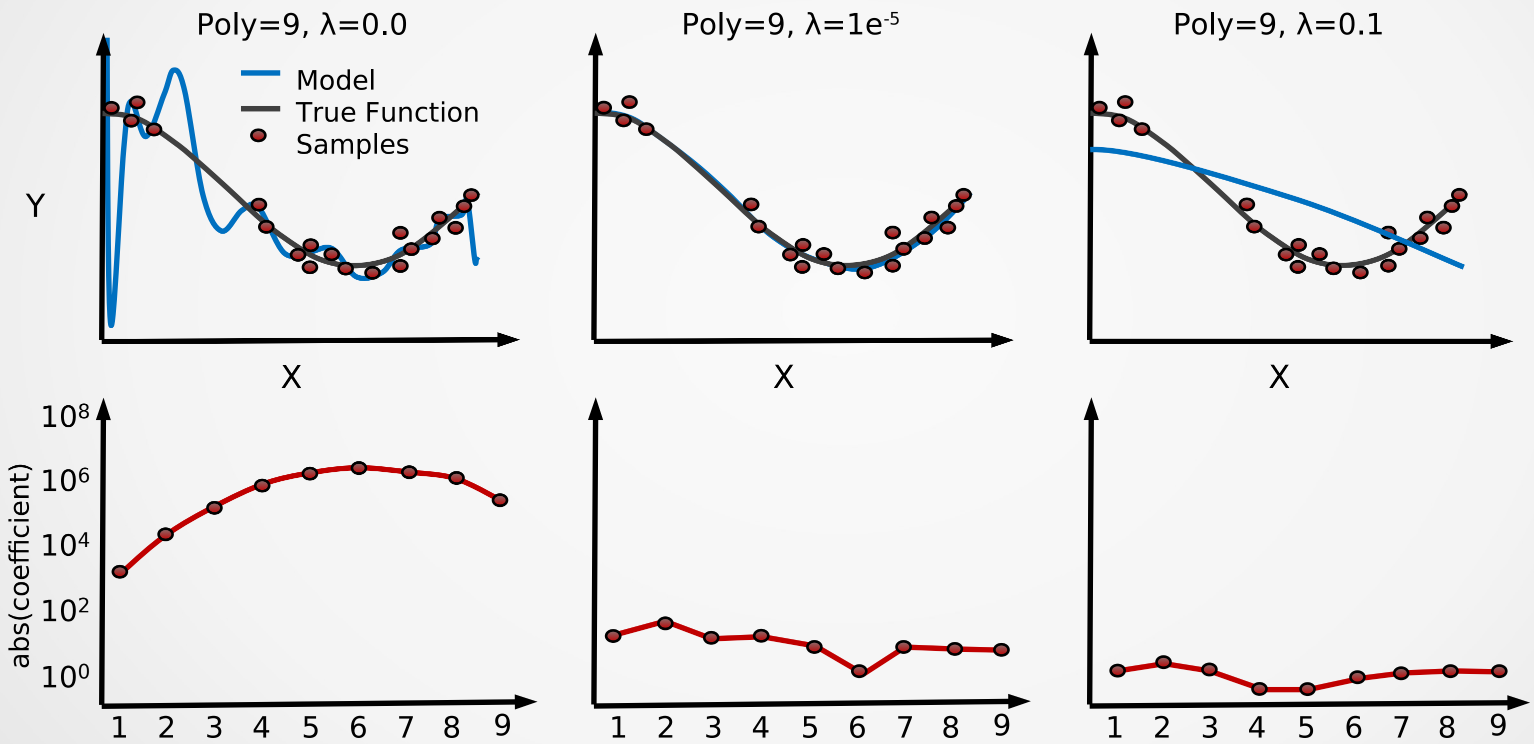
# Ridge Regression

$$J(\beta_0, \beta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left( \left( \beta_0 + \beta_1 x^i_{obs} \right) - y^i_{obs} \right)^2 + \lambda \sum_{j=1}^{k} \beta_j^2$$

- Penalty shrinks magnitude of all coefficients
- Larger coefficients strongly penalized because of the squaring
- $\lambda$ controls how strongly we want to regularize

# Ridge Regression

## Effect of ridge regression on parameters




$$J(\beta_0, \beta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left( \left( \beta_0 + \beta_1 x_{obs}^{i} \right) - y_{obs}^{i} \right)^2 + \lambda \sum_{j=1}^{k} \beta_j^2$$

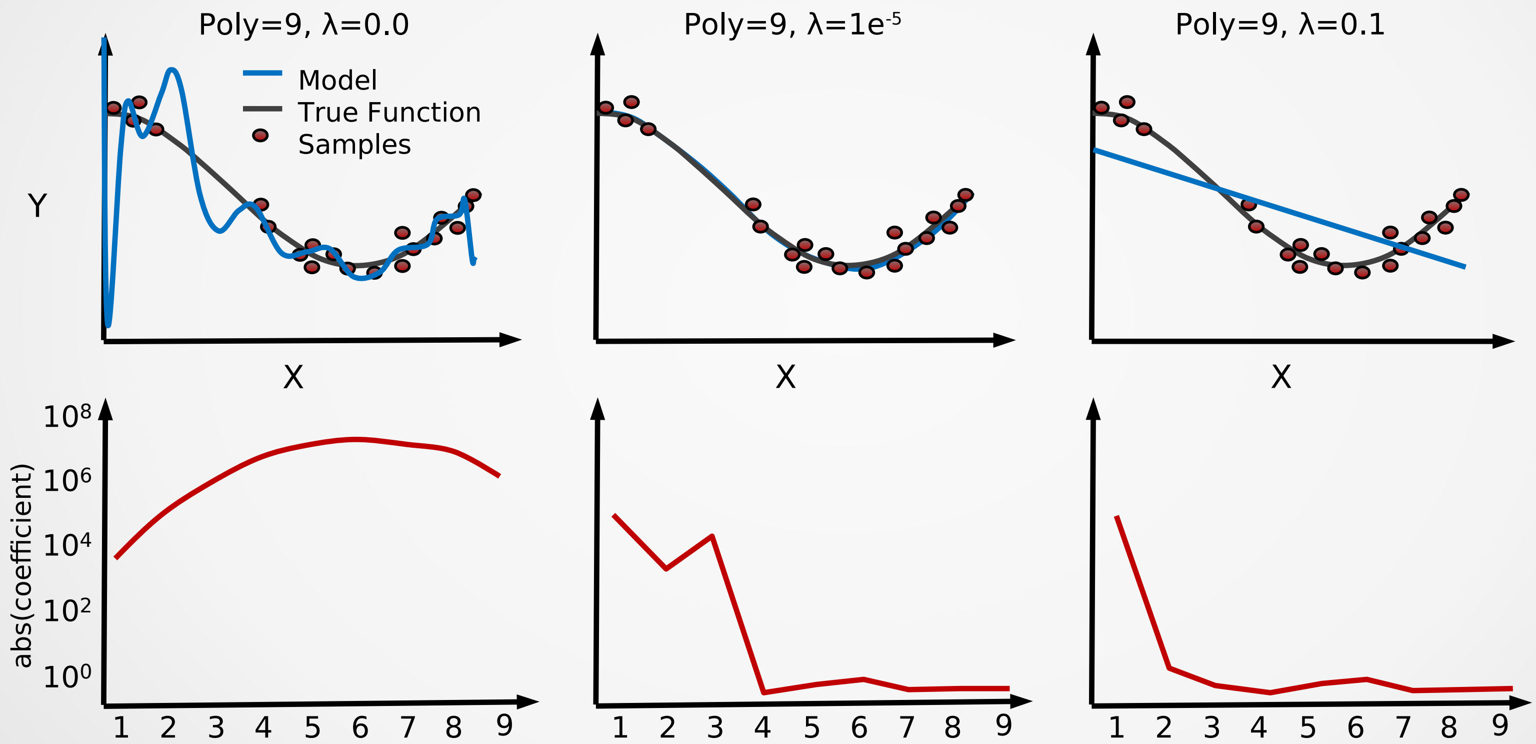
# Lasso Regression (L1)

$$J(\beta_0, \beta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left( \left( \beta_0 + \beta_1 x^i_{obs} \right) - y^i_{obs} \right)^2 + \lambda \sum_{j=1}^{k} |\beta_j|$$

- Penalty selectively shrinks some coefficients
- Can be used for feature selection
- Slower to converge than Ridge Regression

# Lasso Regression (L1)

## Effect of lasso regression on parameters



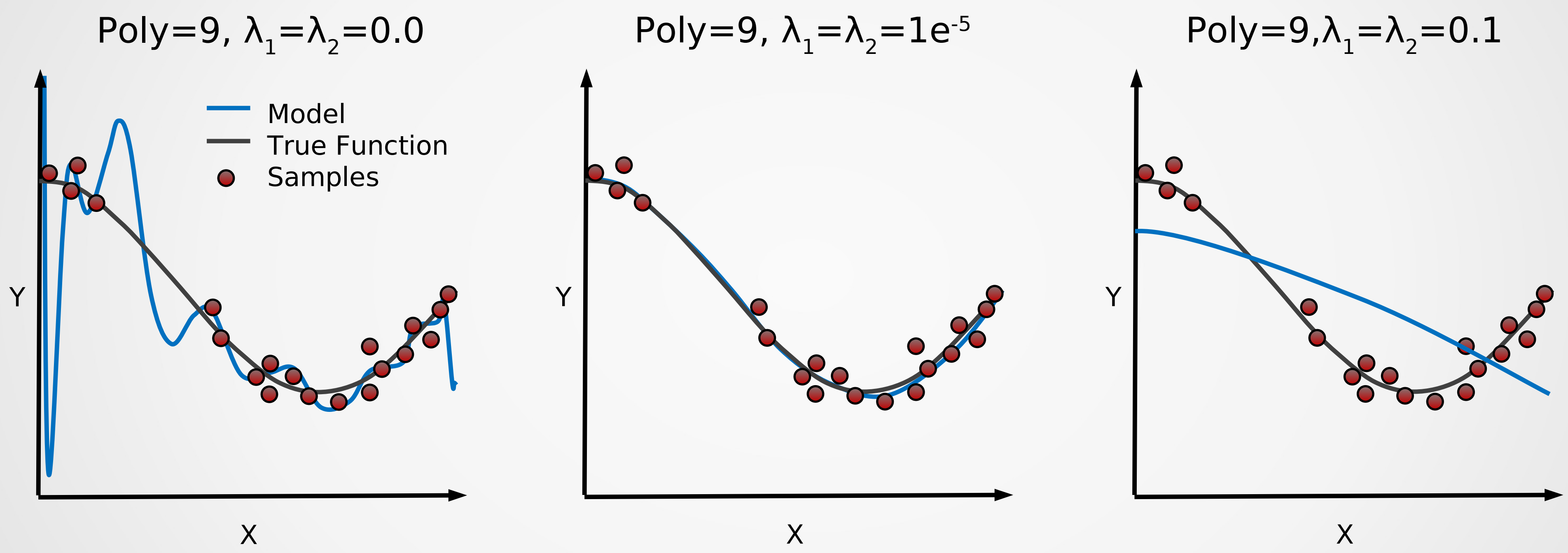

$$J(\beta_0, \beta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left( (\beta_0 + \beta_1 x_{obs}^{i}) - y_{obs}^{i} \right)^2 + \lambda \sum_{j=1}^{k} |\beta_j|$$

# Elastic Net Regularization

$$J(\beta_0, \beta_1) = \frac{1}{2m}\sum_{i=1}^{m}\left(\left(\beta_0 + \beta_1 x_{obs}^{i}\right) - y_{obs}^{i}\right)^2 + \lambda_1 \sum_{j=1}^{k} |\beta_j| + \lambda_2 \sum_{j=1}^{k} \beta_j^2$$

- Compromise of both Ridge and Lasso regression
- Requires tuning of additional parameter that distributes regularization penalty between L1 and L2

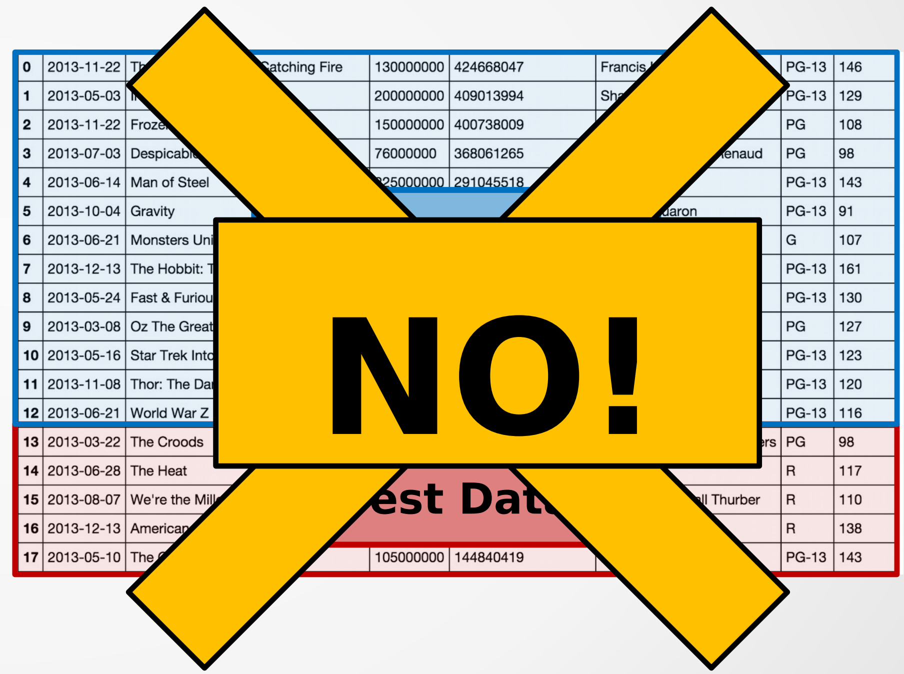
# Elastic Net Regularization



# Optimizing Hyperparameters

- Regularization coefficients ($\lambda_1$ and $\lambda_2$) are empirically determined
- Need values that generalize, never use test data for tuning of the training process

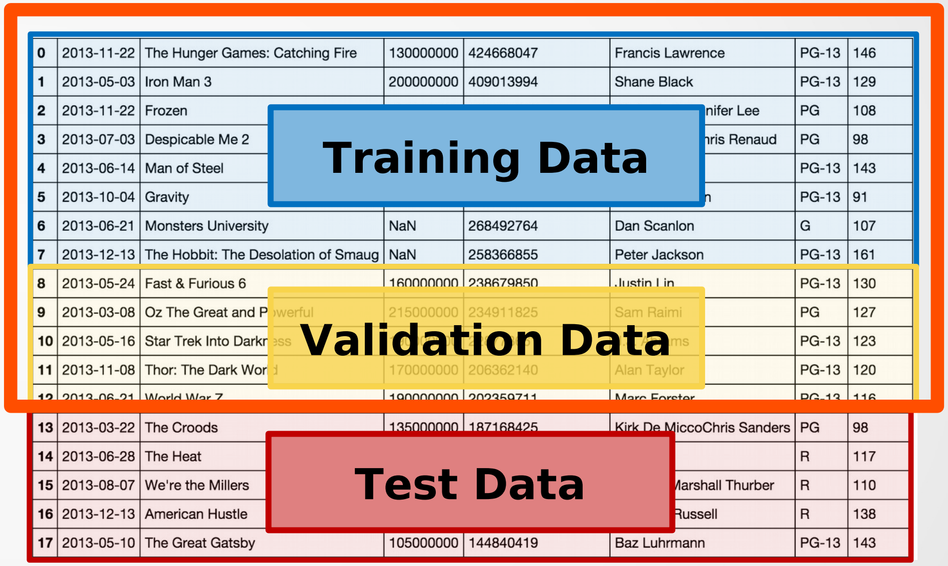
Why not use cross-validation with test data?



# Optimizing Hyperparameters

- Regularization coefficients ($\lambda_1$ and $\lambda_2$) are empirically determined
- Need values that generalize, never use test data for tuning of the training process
- Create additional split to tune hyperparameters – validation set

**Tune λ with Cross Validation**



# Ridge Regression Syntax

- Import the class containing the regression method

```
from sklearn.linear_model import Ridge
```

- Create an instance of the class

```
RR = Ridge(alpha=1.0)
```

- Fit the instance on the data and then predict the expected value

```
RR = RR.fit(x_train, y_train)
y_predict = RR.predict(x_test)
```

- The `RidgeCV` class will perform cross validation on a set of values for alpha

# Lasso Regression Syntax

- Import the class containing the regression method

```
from sklearn.linear_model import Lasso
```

- Create an instance of the class

```
LR = Lasso(alpha=1.0)
```

- Fit the instance on the data and then predict the expected value

```
LR = LR.fit(x_train, y_train)
y_predict = LR.predict(x_test)
```

- The `LassoCV` class will perform cross validation on a set of values for alpha

# ElasticNet Regression Syntax

- Import the class containing the regression method

```
from sklearn.linear_model import ElasticNet
```

- Create an instance of the class

```
EN = ElasticNet(alpha=1.0, l1_ratio=0.5)
```

- Fit the instance on the data and then predict the expected value

```
EN = EN.fit(x_train, y_train)
y_predict = EN.predict(x_test)
```

- The `ElasticNetCV` class will perform cross validation on a set of values for alpha

# Feature Selection

- Regularization performs feature selection by shrinking the contribution of features
- For L1-regularization, this is accomplished by driving some coefficients to zero
- Feature selection can also be performed by removing features

# Feature Selection

Why is this important?

- Reducing the number of features is another way to prevent overfitting (similar to regularization)
- For some models, fewer features can improve fitting time and/or results
- Identifying most critical features can improve model interpretability

# Recursive Feature Elimination Syntax

- Import the class containing the feature selection method

```
from sklearn.feature_selection import RFE
```

- Create an instance of the class

```
rfeMod = RFE(est, n_features_to_select=5)
```

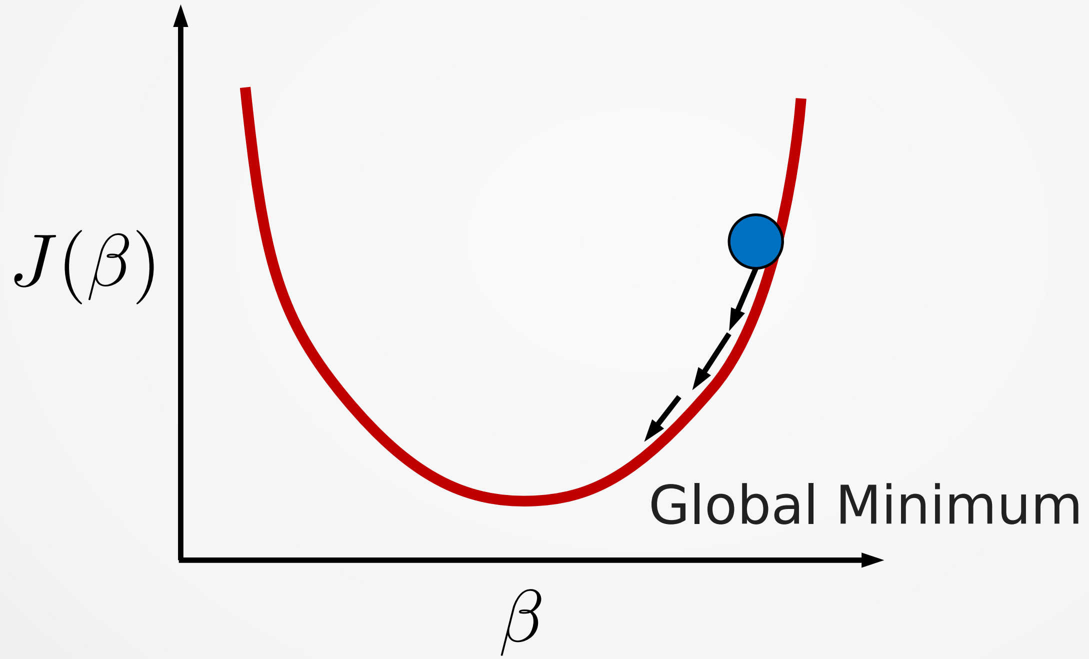
- Fit the instance on the data and then predict the expected value

```
rfeMod = rfeMod.fit(x_train, y_train)
y_predict = rfeMod.predict(x_test)
```

- The RFECV class will perform feature elimination using cross validation

# Gradient Descent

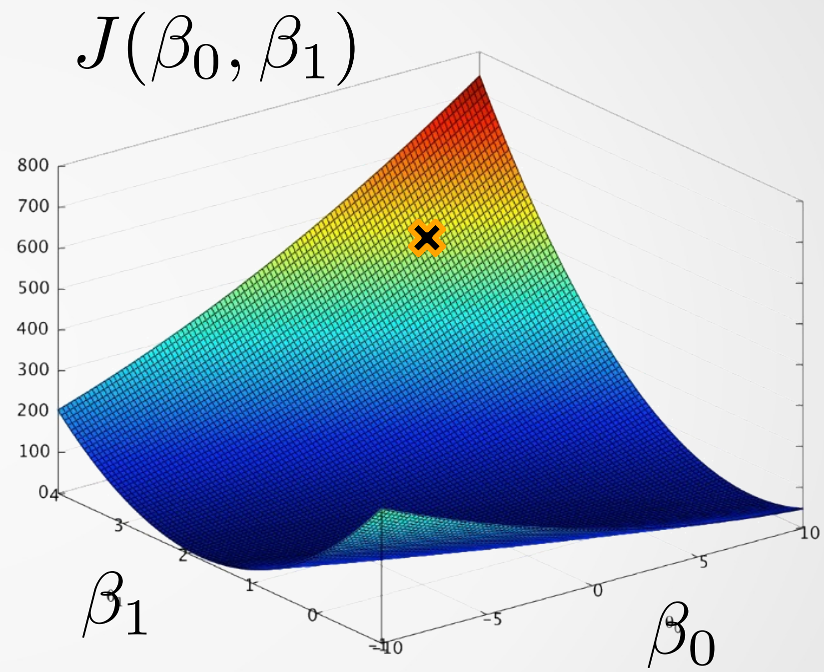
Start with a cost function $J(\beta)$




Gradually move towards the minimum

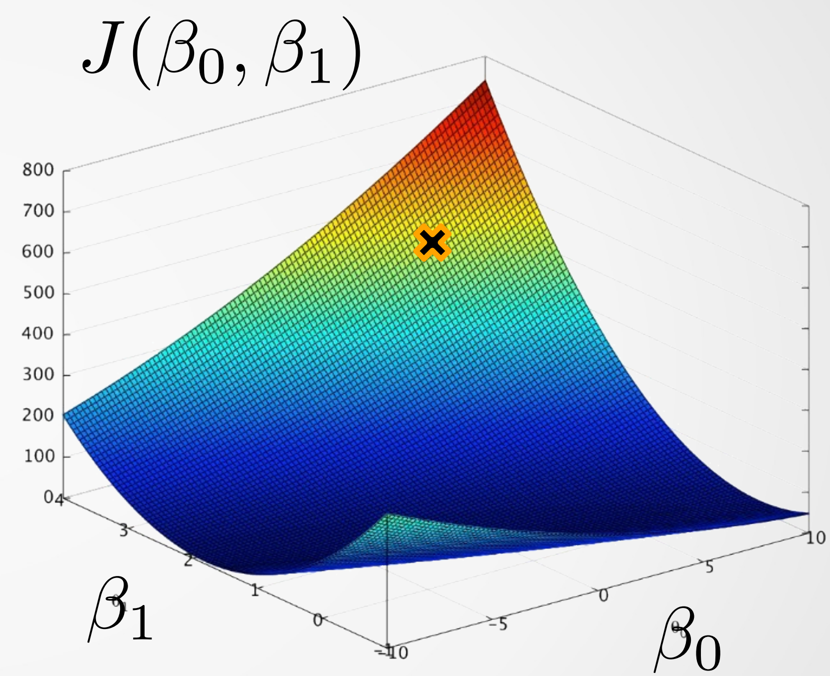
# Gradient Descent

- Two-parameter case $(\beta_0, \beta_1)$
- Surface is more complicated than curve, meaning finding the minimum is also more complicated
- How to do this without knowing what the function $J(\beta_0, \beta_1)$ looks like?



# Gradient Descent

- Compute the gradient $\nabla J(\beta_0, \beta_1)$ which points in the direction of the biggest increase
- The negative $-\nabla J(\beta_0, \beta_1)$ points to the biggest decrease at that point
- The gradient vector's coordinates consist of the partial derivative of the parameters $\nabla J(\beta_0, \ldots, \beta_n) = \left\langle \frac{\delta J}{\delta \beta_0}, \ldots, \frac{\delta J}{\delta \beta_n} \right\rangle$

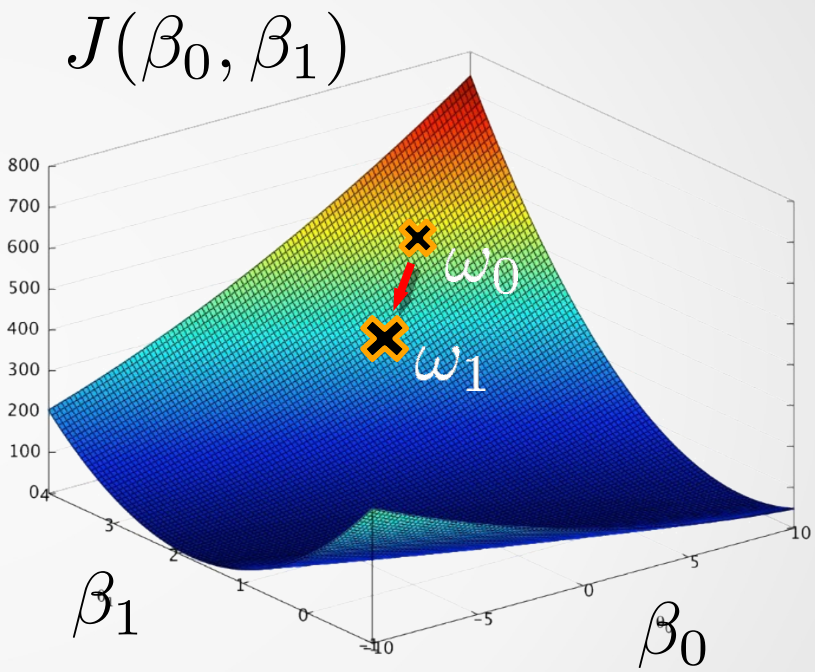


# Gradient Descent

- Use the gradient $\nabla$ and the cost function to calculate the next point from the current

$$\omega_1 = \omega_0 - \alpha \nabla \frac{1}{2} \sum_{i=1}^{m} \left( \left( \beta_0 + \beta_1 x_{obs}^{i} \right) - y_{obs}^{i} \right)^2$$

- The learning rate $\alpha$ is a tunable parameter that determines step size

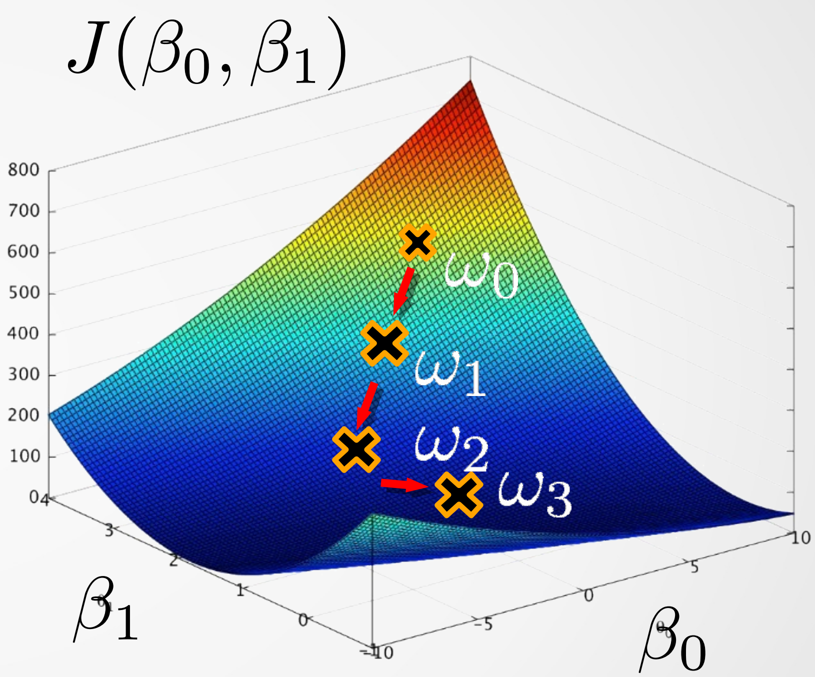


# Gradient Descent

- Each point can be iteratively calculated from the previous one

$$\omega_2 = \omega_1 - \alpha \nabla \frac{1}{2} \sum_{i=1}^{m} \left( \left( \beta_0 + \beta_1 x^i_{obs} \right) - y^i_{obs} \right)^2$$

$$\omega_3 = \omega_2 - \alpha \nabla \frac{1}{2} \sum_{i=1}^{m} \left( \left( \beta_0 + \beta_1 x^i_{obs} \right) - y^i_{obs} \right)^2$$



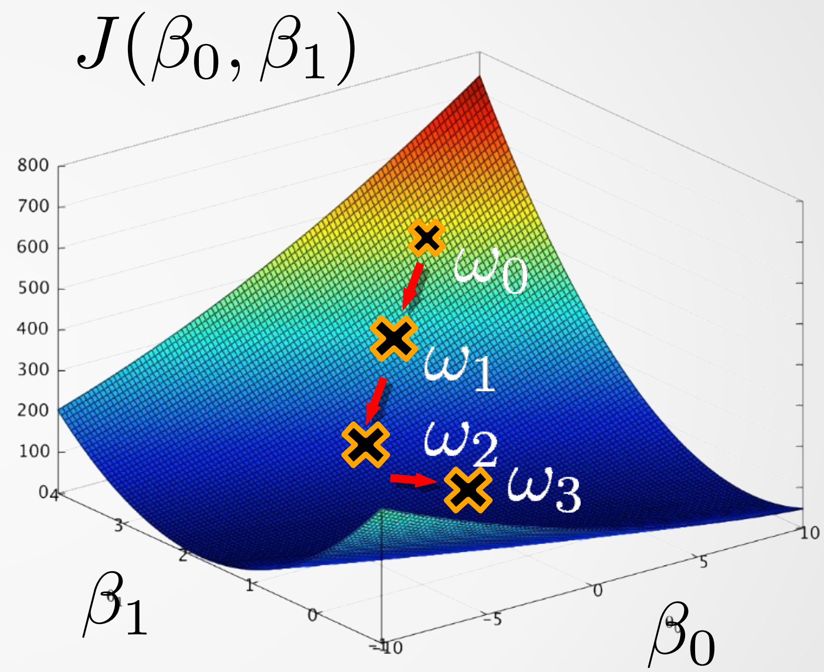
# Stochastic Gradient Descent

- Use a single data point to determine the gradient and cost function instead of all the data

$$\omega_1 = \omega_0 - \alpha\nabla\frac{1}{2}\cancel{\sum_{i=1}^{m}}\left(\left(\beta_0 + \beta_1 x_{obs}^{i}\right) - y_{obs}^{i}\right)^2$$

$$\Downarrow$$

$$\boxed{\omega_1 = \omega_0 - \alpha\nabla\frac{1}{2}\left(\left(\beta_0 + \beta_1 x_{obs}^{0}\right) - y_{obs}^{0}\right)^2}$$



# Stochastic Gradient Descent

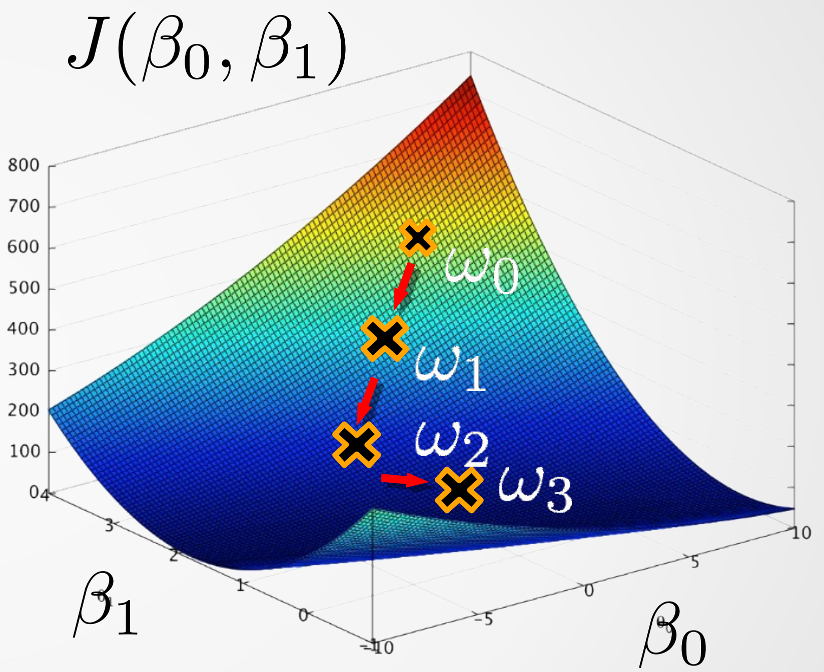
- Use a single data point to determine the gradient and cost function instead of all the data

$$\omega_1 = \omega_0 - \alpha \nabla \frac{1}{2} \left( (\beta_0 + \beta_1 x^0_{obs}) - y^0_{obs} \right)^2$$

$$\cdots$$

$$\omega_n = \omega_{n-1} - \alpha \nabla \frac{1}{2} \left( (\beta_0 + \beta_1 x^0_{obs}) - y^0_{obs} \right)^2$$

- Path is a less direct due to noise in single data point - “stochastic”

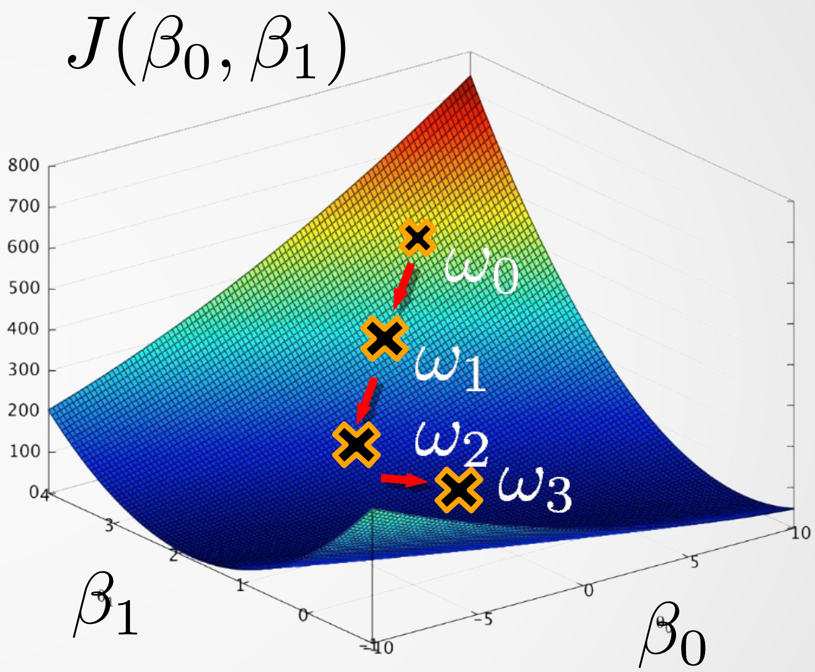


# Mini Batch Gradient Descent

- Perform an update for every *n* training examples

$$\omega_1 = \omega_0 - \alpha \nabla \frac{1}{2} \boxed{\sum_{i=1}^{n}} ((\beta_0 + \beta_1 x_{obs}^0) - y_{obs}^0)^2$$

- Best of both worlds:
  - Reduced memory relative to “vanilla” gradient descent
  - Less noisy than stochastic gradient descent



# Mini Batch Gradient Descent

- Mini batch implementation typically used for neural nets
- Batch sizes range from 50–256 points
- Trade off between batch size and learning rate ()
- Tailor learning rate schedule: gradually reduce learning rate during a given epoch

# Stochastic Gradient Descent Regression Syntax

- Import the class containing the regression model

```
from sklearn.linear_model import SGDRegressor
```

- Create an instance of the class

```
SGDreg = SGDRegressor(loss='squared loss',
                      alpha=0.1, penalty='l2')
```

- Fit the instance on the data and then transform the data

```
SGDreg = SGDRreg.fit(x_train, y_train)
y_pred = SGDreg.predict(x_test)
```

- Other loss methods exist e.g. `epsilon_insensitive`, `huber`

## Stochastic Gradient Descent Regression Syntax

- Import the class containing the regression model

```
from sklearn.linear_model import SGDRegressor
```

- Create an instance of the class

```
SGDreg = SGDRegressor(loss='squared loss',
                      alpha=0.1, penalty='l2')
```

- Fit the instance on the data and then transform the data

```
SGDreg = SGDRreg.partial_fit(x_train, y_train)
y_pred = SGDreg.predict(x_test)
```

# Stochastic Gradient Descent Classification Syntax

- Import the class containing the regression model

```
from sklearn.linear_model import SGDClassifier
```

- Create an instance of the class

```
SGDclass = SGDClassifier(loss='squared loss',
                         alpha=0.1, penalty='l2')
```

- Fit the instance on the data and then transform the data

```
SGDclass = SGDRclass.fit(x_train, y_train)
y_pred = SGDclass.predict(x_test)
```

- Other loss methods exist e.g. `hinge`, `squared_hinge`

# Stochastic Gradient Descent Classification Syntax

- Import the class containing the regression model

```
from sklearn.linear_model import SGDClassifier
```

- Create an instance of the class

```
SGDclass = SGDClassifier(loss='squared loss',
                         alpha=0.1, penalty='l2')
```

- Fit the instance on the data and then transform the data

```
SGDclass = SGDRclass.partial_fit(x_train, y_train)
y_pred = SGDclass.predict(x_test)
```

# End of Lecture

Many thanks to Intel Software for providing a variety of resources for this lecture series