Lecture Four: Model Selection

COMP3032 Machine Learning

Polynomial Regression

Training Data: underfitting and overfitting

- The Bias-Variance trade-off

Cross-validation

Learning Curves and Expected Error

- Learning curves
- Expected prediction error

Regularisation

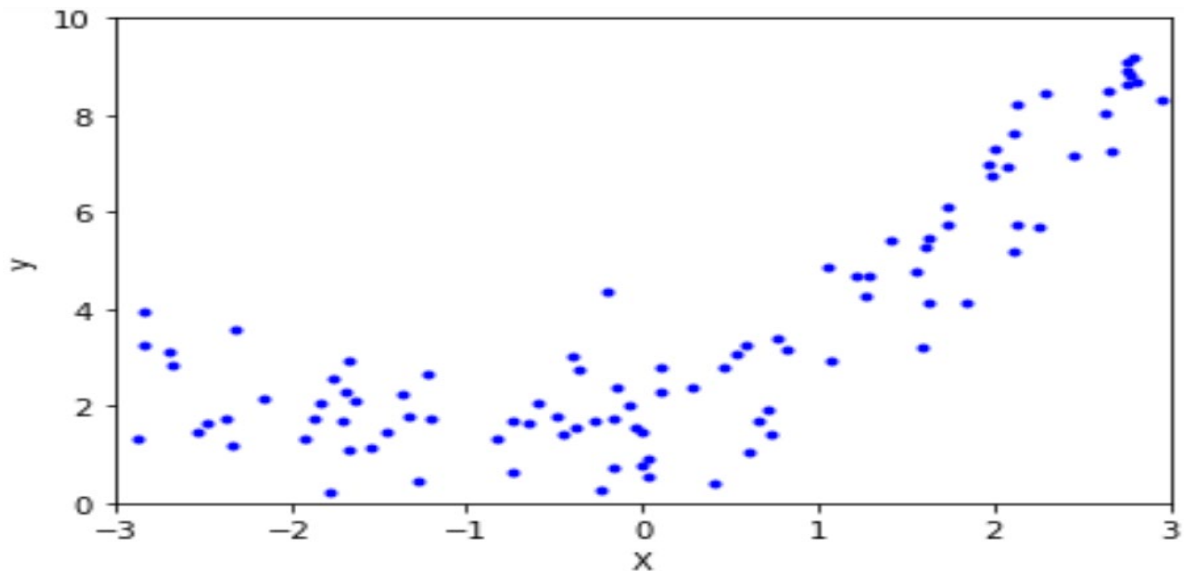
Polynomial Regression

- Polynomial regression generalises linear regression by allowing exponents to occur in each feature
- $\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2^2 + ... + \theta_n x_n^n$
- Polynomial regression is used when we have a non-linear looking data

An Example: generate some nonlinear data

Generate some nonlinear data, based on a simple quadratic equation (plus some noise)

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



An example: extend the training data with polynomial 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])
```

- PolynomialFeatures adds all combinations of features up to the given degree.
- X_poly contains the original feature of X plus the square of this feature.

An Example: Fit a linear regression model

Now you can fit a LinearRegression model to this extended training data

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

array([1.78134581]), array([[0.93366893, 0.56456263]])
```

- The model estimates $\hat{y} = 1.78 + 0.93x_1 + 0.56x_2^2$
- Not bad, the original was
  $\hat{y} = 2.0 + 1.0x_1 + 0.5x_2^2 + \mathit{Gaussiannoise}$

An Example: the graph

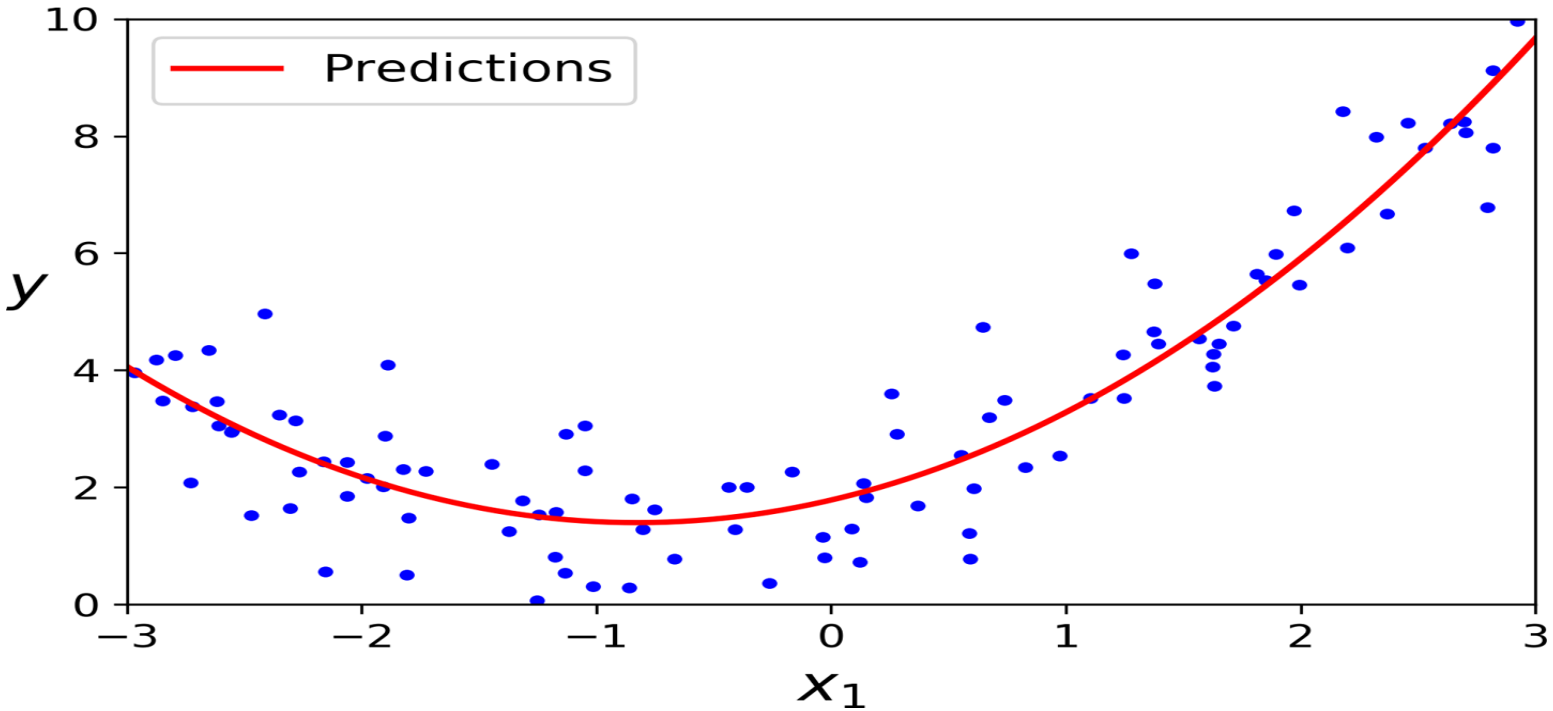


Figure 4-13. Polynomial Regression model predictions

Underfitting and Overtting

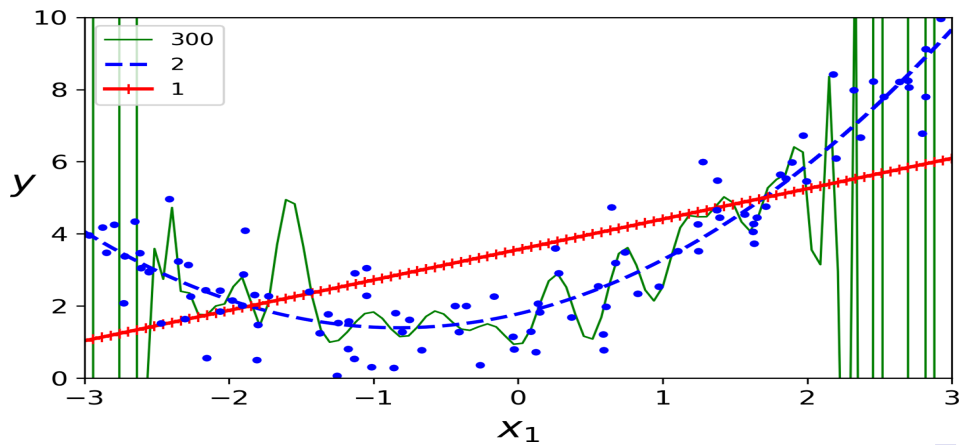
- Underfitting or high bias occurs when the model is too simple to learn the underlying structure of the data
- Overfitting or high variance is when the model performs well on the training data but it doesn't generalise well to predict new data

Overfitting the Training Data

- Overfitting usually occurs when the model is too complex relative to the noisiness of the data
- Complex models can fit training data really well, but if the data is noisy, then the model is likely to detect patterns in the noise
- Overgenralisation

Model selection

- High-degree Polynomial Regression is likely fit the training data much better than with plain Linear Regression
- Perform a 300-degree polynomial model to the preceding training data, and compares the result with a pure linear model and a quadratic model (second-degree polynomial)
- This high-degree Polynomial Regression model is severely overfitting it
- The linear model is underfitting it
- The model that generalizes best is the quadratic model



The Bias/Variance trade-off

A model's generalization error can come from three different errors

- Bias: due to overly simplistic assumptions
  - Such as assuming that the data is linear when it is actually quadratic
  - A high-bias model indicates not capturing the underlying patterns in the data
  - Lead to underfit the training data
- Variance: due to the model's excessive sensitivity to small variations in the training data
  - A high-variance model suggests that the model is too complex and captures noise in the training data
  - Lead to overfit the training data
  - Such as a high-degree polynomial model
- Irreducible error: due to the noisiness of the data itself
  - The only way to reduce this error is to clean up the data

The Bias/Variance trade-off

- The trade-off arises as when you try to reduce bias, you often increase variance, and vice versa
- Increase a model's complexity will typically increase its variance and reduce its bias
- Reduce a model?s complexity increases its bias and reduces its variance
- Find a balance between creating models that are too simplistic and overly complex

Possible solutions to Underfitting

Ideally, we want a model that neither underfits nor overfits

- Selec a more powerful model with more parameters
- Feed better features to the algorithm
- Reduce the constraints on the model

Possible solutions to Overfitting

Simplify the model

- Reduce the number of features (attributes)
  - Manually choosing fewer features
  - Use a model selection algorithm
- Regularization: constraining the model
  - Keep all the features, but reduce the parameters θ_j
  - e.g. linear vs higher-degree polynomial models
- Reduce the noise in the training data (e.g. fix errors, remove outliers)
- Gather more training data

Training set and validation set

- Use part of the training set for training and part of it for model validation
- i.e. split the training set into a smaller training set and a validation set
- Train your models against the smaller training set and evaluate them against the validation set
- Don't touch the test set until you are ready to launch a model

K-fold cross-validation

The idea:

- use many small validation sets
- Each model is evaluated once per validation set after it is trained on the rest of the data
- averaging out all the evaluations of a model to get a much more accurate measure of its performance
- drawback, the training time is multiplied by the number of validation sets

K-fold cross-validation

K-fold cross-validation in more detail:

- split the training set into k(usually 10) distinct subsets called folds
- picking a different fold for evaluation every time and training on the other k-1(9) folds
- train and evaluate a model k(10) times
- The result is an array containing the k evaluation scores, then calculate the average

Typical splits

- Training set, 80%, validation set, 10%, Test set, 10%
- Training set, 50%, validation set, 25%, Test set, 25%

Cross-validation

10-fold cross-validation on models

```
>>>from sklearn.model_selection import cross_val_score
>>>lin_scores = cross_val_score(lin_reg, housing_prepared,
   housing_labels,scoring="neg_mean_squared_error", cv=10)
>>> lin_rmse_scores = np.sqrt(-lin_scores)
>>> display_scores(lin_rmse_scores)
Scores: [66782.73843989 66960.118071
         70347.95244419 74739.57052552
         68031.13388938 71193.84183426
         64969.63056405 68281.61137997
         71552.91566558 67665.10082067]
Mean: 69052.46136345083
Standard deviation: 2731.674001798348
```

Model selection

- How to decide how complex your model should be?
- How to tell that your model is overfitting or underfitting the data?
- Using cross-validation to get an estimate of a model's generalization performance.
  - If a model performs well on the training data but generalizes poorly according to the cross-validation metrics, then your model is overfitting.
  - If it performs poorly on both, then it is underfitting.
  - This is one way to tell when a model is too simple or too complex.

What is a Learning curve

- Another way to select a model is to look at the learning curves
- They are plots of the model's performance on the training set and the validation set as a function of the training set size (or the training iteration).
- To generate the plots, train the model several times on different sized subsets of the training set.

An example

The following code defines a function that, given some training data, plots the learning curves of a model

```
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)
```

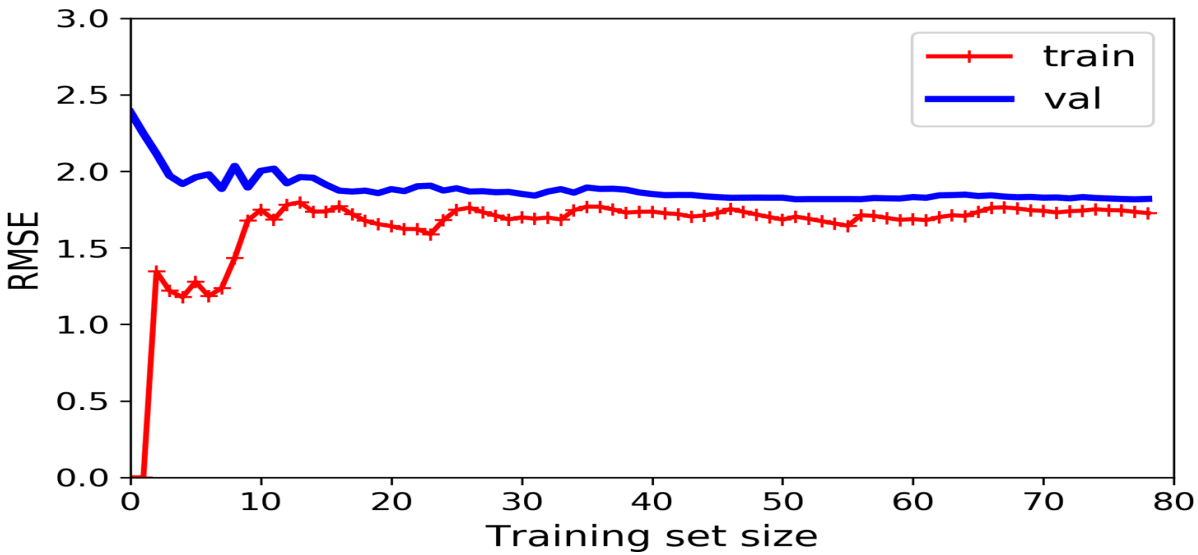
An example continued

```
        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")
```

learning curves of the linear regression model

First look at the learning curves of the plain Linear Regression model (a straight line)

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



The figure: on training data

- It is underfitting
- on training data: zero error for initially, then error goes up until it reaches a plateau
  - When there are just one or two instances in the training set, the model can fit them perfectly
  - As new instances are added, it becomes impossible for the model to fit the training data perfectly, because the data is noisy or not linear at all
  - At some point adding new instances doesn't make the average error much better or worse

The figure: on validation data

- on validation data: big error initially, then error goes down until it reaches a plateau
  - When the model is trained on very few training instances, it is incapable of generalizing properly
  - As the model is shown more training examples, it learns, the error slowly goes down
  - A straight line cannot do a good job modeling the data, so the error ends up at a plateau, very close to the other curve
- Both curves have reached a plateau, close and fairly high.
- This is the feature of an underfitting model

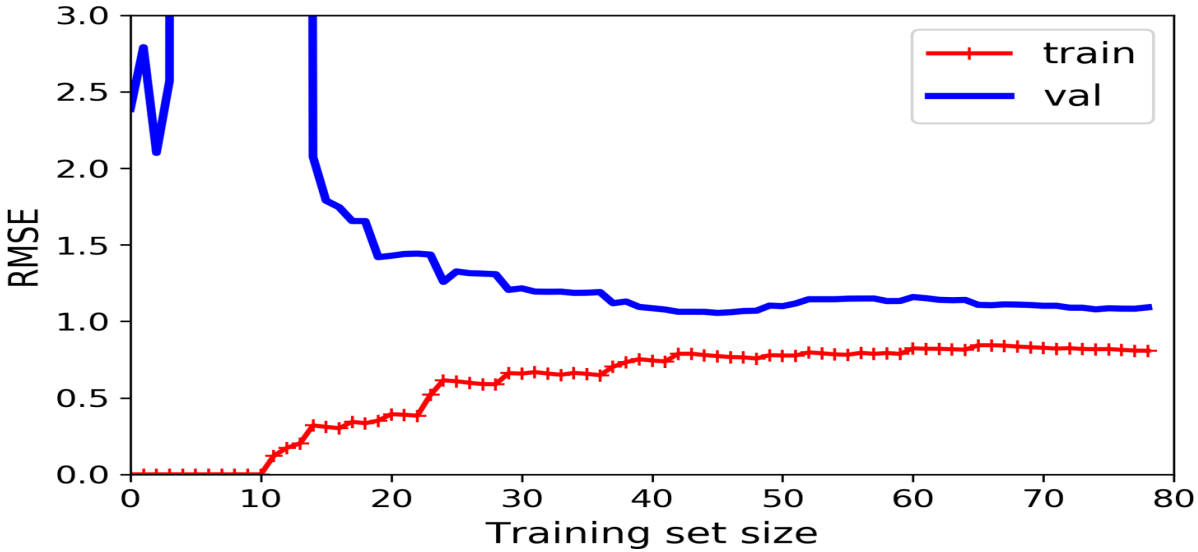
learning curves of the polynomial regression model

Then look at the learning curves of a 10th-degree polynomial model on the same data

```
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)
```

The figure

Learning curves for the 10th-degree polynomial model



The figure: overfitting

- It is overfitting
- The error on the training data is much lower than with the Linear Regression model
- There is a gap between the curves. This means that the model performs a lot better on the training data than on the validation data
- This is the feature of an overfitting model

Expected Prediction Error (EPE)

How accurate was our prediction?

- Quantify the average prediction error of a model on NEW, unseen data points
  - A measure of how well the model's predictions match the actual outcomes
  - calculate the difference between the actual target value and the predicted value for each data point, then averaging these errors across all data points
- EPE is the true error of prediction
- EPE provides insight into how well the model generalizes to new data
- Can help in model selection and hyperparameter tuning
- EPE is often used to evaluate the performance of regression models

Expected Prediction Error (EPE)

How accurate was our prediction?

- True model: $y = f_\theta(x) + \delta$
  - θ is the model parameters, and δ is the random noise that is independent of x and has mean zero
- Estimated model: $\hat{y} = \hat{f}_{\hat{\theta}}(x)$
- Linear regression model: $y = \theta_0 + \theta_1 x_1 + \cdots + \theta_p x_p + \delta$
- Linear regression estimator: $\hat{y} = \hat{\theta}_0 + \hat{\theta}_1 x_1 + \cdots + \hat{\theta}_k x_k$
- $EPE = E[L(y, \hat{f}_{\hat{\theta}}(x))]$, $L(y, \hat{f})$is the loss function, which is always non-negative
  - EPE of regressor: $E[(y - \hat{y})^2]$

What is Regularisation

Constraining a model to make it simpler and reduce the risk of overfitting

- With increasing number of features, or growing model complexity, the chances for overfitting increase
- Regularisation adds a penalty to the cost function
- The regularisation term puts a constraint on the coefficients and weights of our model
- The idea is to make more complex models more expensive
  - For polynomial models, a simple regularisation is to reduce the number of degrees
  - For linear models, we can archive regularisation by constraining the weights of models

Regularisation by penalising complex models

- Penalising the magnitude of coefficients of features,θ_i , while minimising error
- Add a penalty term to the loss function that consists of
  - a complexity parameter, λ
  - a function with all the coefficients of features, $P(\theta)$
  - i.e. $L(y, \hat{f}) + \lambda P(\theta)$
- Optimise for small cost AND also low complexity model

Regularisation reduces the risk of overfitting

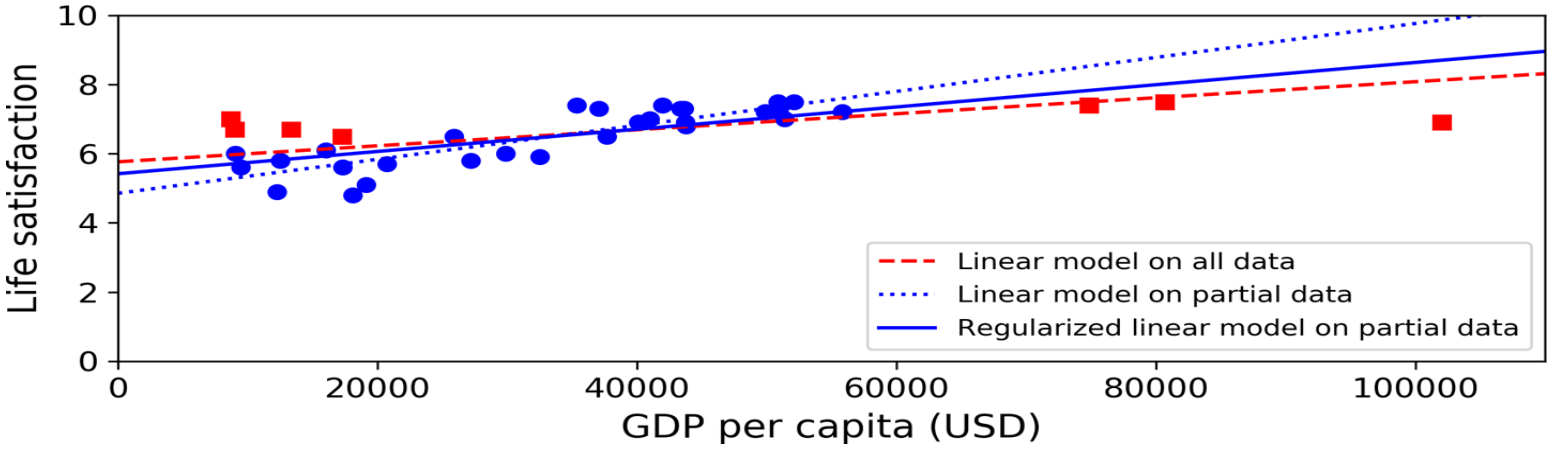
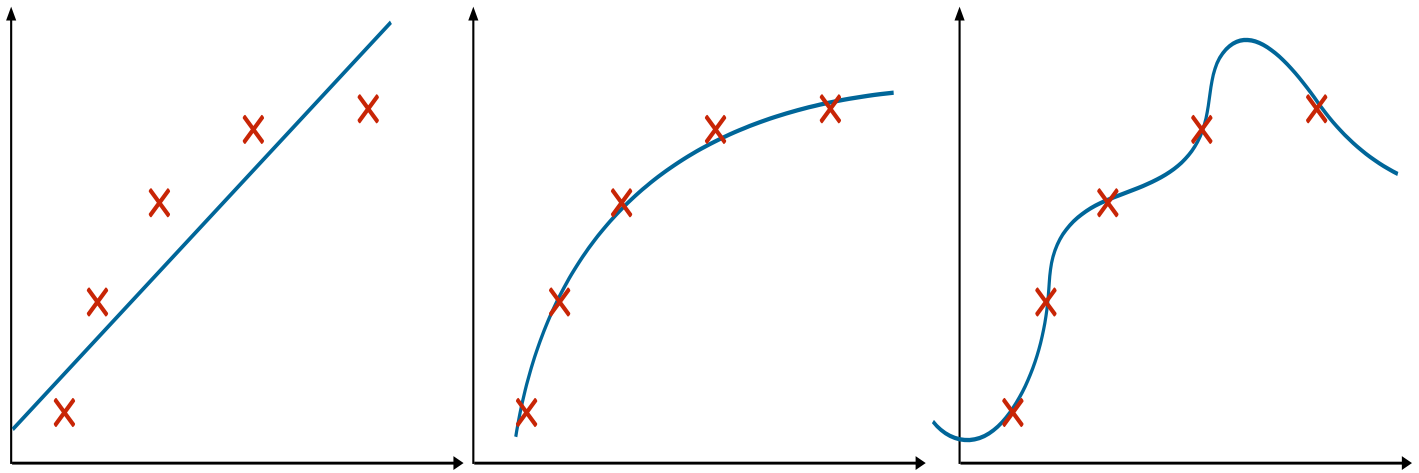


Figure 1-23. Regularization reduces the risk of overfitting

Regularisation reduces the risk of overfitting



Regularized Linear Models

A good way to reduce overfitting is to regularize the model (i.e., to constrain it). The fewer degrees of freedom a model has, the harder for it to overfit the data.

- A simple way to regularize a polynomial model is to reduce the number of polynomial degrees
- Regularization is typically achieved by constraining the weights of the model
- Ridge Regression is a regularized version of Linear Regression

Add a regularization term to cost function

- A regularization term equal to $\alpha \sum_1^n \theta_i^2$ is added to the cost function
- This forces the learning algorithm to not only fit the data but also keep the model weights as small as possible
- The regularization term should only be added to the cost function during training
- Once the model is trained, use the unregularized performance measure to evaluate the model's performance

- Ridge Regression cost function: $J(\theta) = MSE(\theta) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$
- The hyperparameter α controls how much you want to regularize the model.
- If $\alpha = 0$, then Ridge Regression is just Linear Regression.
- If α is very large, then all weights would be very close to zero and the result is a flat line going through the data's mean.
- Note that the bias term θ_0 is not regularized
- Closed form equation: $\hat{\Theta} = (X^T \cdot X + \alpha A)^{-1} \cdot X^T \cdot y$
  - A is the identity matrix
  - except with a 0 in the top-left cell, corresponding to the bias term.

Ridge Regression with Scikit-Learn

Here is how to perform Ridge Regression with Scikit-Learn using a closed-form solution (a matrix factorization technique)

```
from sklearn.linear_model import Ridge

ridge_reg = Ridge(alpha=1, solver="cholesky")
ridge_reg.fit(X, y)
ridge_reg.predict([[1.5]])

# array([[4.38954632]])
```

Ridge Regression with Scikit-Learn

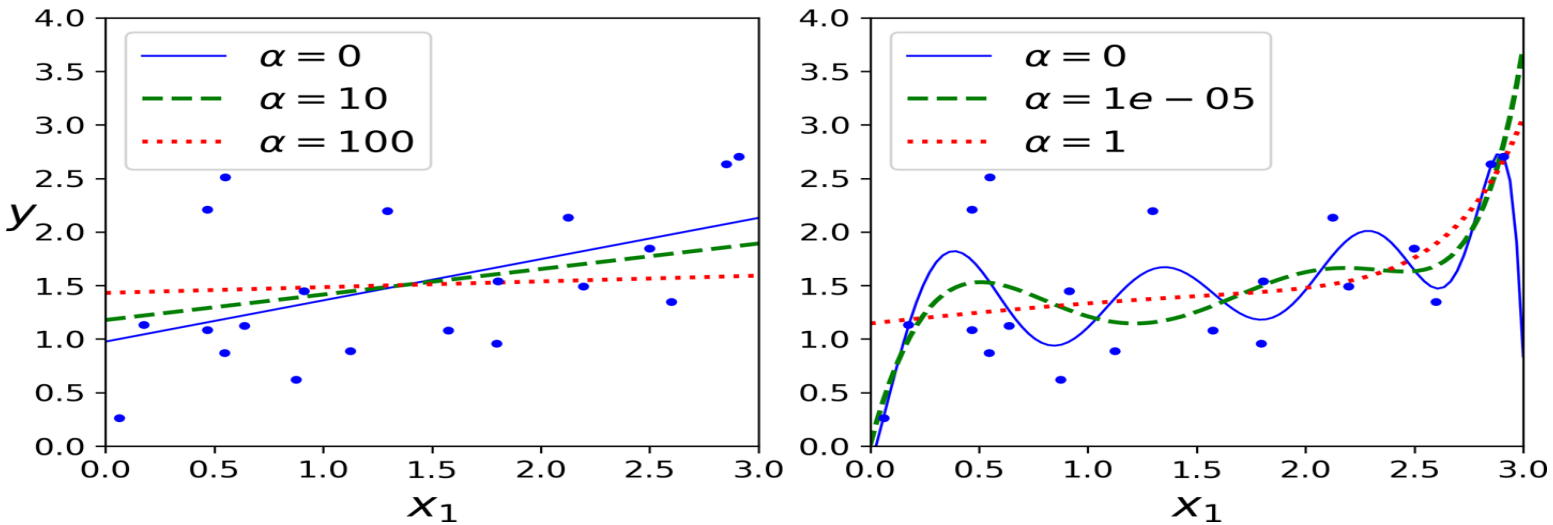
Here is how to perform Ridge Regression with Scikit-Learn using Stochastic Gradient Descent

```
from sklearn.linear_model import SGDRegressor

sgd_reg = SGDRegressor(penalty="l2")
sgd_reg.fit(X, y.ravel())
sgd_reg.predict([[1.5]])

# array([4.354387])
```

- The penalty hyperparameter sets the type of regularization term to use.
- Specifying "*l2*" indicates that the regularization term added to the cost function is the same as the Ridge regression.

Ridge Regression with Scikit-Learn

A linear model (left) and a polynomial model (right), both with various levels of Ridge regularization



Lasso Regression

lasso regression is another regularized version of Linear Regression. Just like ridge regression, it also adds a regularization term to the cost function.

- Regularisation term is an *absolute value*
- Cost function

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

- Main feature: tend to *eliminate* the weights of the *least important* features

Lasso Regression with Scikit-Learn

Here is how to perform Ridge Regression with Scikit-Learn:

```
from sklearn.linear_model import Lasso

lasso_reg = Lasso(alpha=0.1)
lasso_reg.fit(X, y)
lasso_reg.predict([[1.5]])

#  array([4.61287497])
```

Note: Can also use the SGDRegressor() function for Lasso regression, with "l1" for the penalty parameter.

Ridge vs. Lasso Regression

Ridge:

- includes all (or none) of the features
- models perform better in prediction
- prevent overfitting, but not very useful for data with huge dimensions
- For Data with highly correlated features, generally works well, coefficients will be distributed among them

Lasso:

- performs feature selection, some coefficients become zero
- solution is sparse, useful for data with huge dimension
- arbitrarily selects any one feature among the correlated ones, chosen variable changes with model parameter

Early stopping

A different way to regularize iterative learning algorithms such as Gradient Descent is to stop training as soon as the validation error reaches a minimum.

- This is called early stopping.
- The figure shows a complex model (high-degree Polynomial Regression model) being trained with Batch Gradient Descent.
- After a while the validation error stops decreasing and starts to go back up.
- This indicates that the model has started to overfit the training data.
- With early stopping you just stop training as soon as the validation error reaches the minimum.
- Simple and efficient

Early stopping