

2 Regression

Regression is a fundamental tool in machine learning used to model the relationship between an independent variable (called data and denoted x) and a dependent variable (called target and denoted y). The goal is to learn a function that best predicts the target value from the input data. This relationship is shown in figure 2.1 .

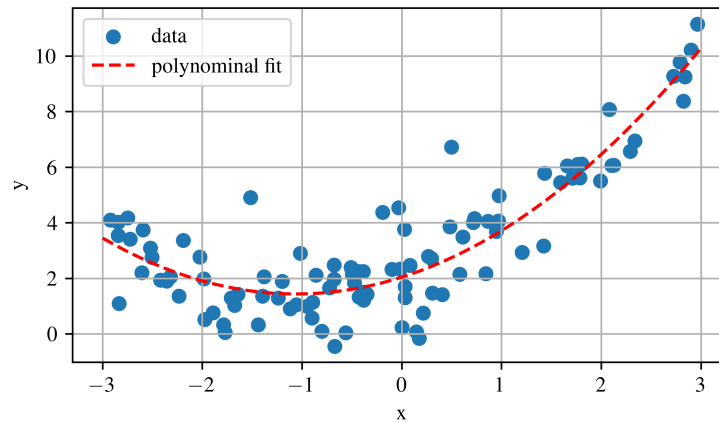


Figure 2.1: Polynomial model fit via regression to a generic dataset.

In this chapter, we first examine Linear Regression and contrast two ways to estimate its parameters:

1. **Closed-form solution.** Solve for the parameter vector in one step by minimizing the cost function on the entire training set.
2. **Gradient Descent .** Apply an iterative optimizer that repeatedly updates the parameters in small steps that lower the cost until it reaches the same optimum found by the closed-form method. We will look at three common Gradient Descent variants: Batch Gradient Descent, Mini-batch Gradient Descent, and Stochastic Gradient Descent.

The first approach gives an exact answer immediately, whereas the second reaches that answer through successive refinements.

Review 2.1 Ames Housing Dataset

The Ames housing dataset provides details on individual residential property sales in Ames, Iowa, from 2006 to 2010 ^a. It includes 2,930 entries and numerous variables (23 nominal, 23 ordinal, 14 discrete, and 20 continuous) that are used to evaluate home values ^b.

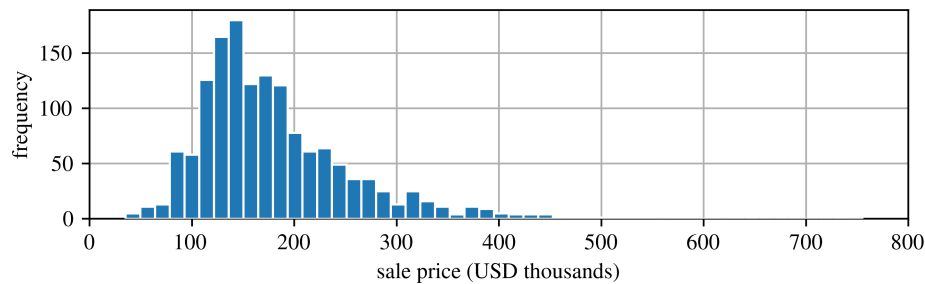


Figure 2.2: Histogram of sale prices from 2006 to 2010 for the 2,930 entries.

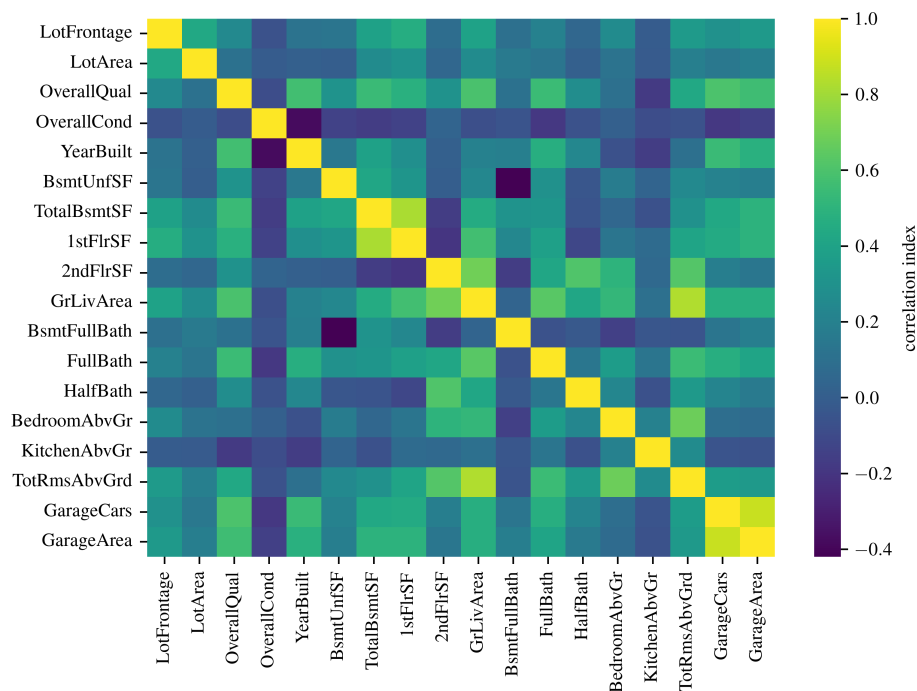


Figure 2.3: Correlation matrix for the 13 input parameters.

^aThe author of this text bought a home in Ames Iowa during this time right as the 2008 housing crash unfolded and on the last day they allowed home loans with no down payment.

^bDe Cock, Dean. "Ames, Iowa: Alternative to the Boston housing data as an end of semester regression project." Journal of Statistics Education 19.3 (2011).

2.1 Linear Regression

Consider the scenario where you wish to determine the value of a house in Ames using the Ames dataset included in sklearn. To illustrate this, let us plot the relationship between the above ground living area and the housing price.

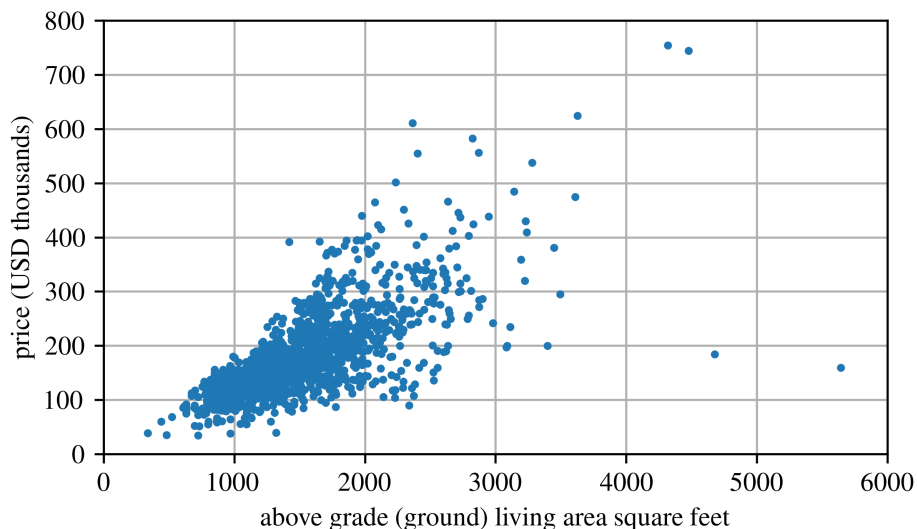


Figure 2.4: Ames housing data.

A clear trend is observable, despite the data's noise (i.e., partial randomness): the value of a house increases with the above ground living area. Therefore, the house value can be modeled as a linear function of the above ground living area:

$$\text{price_model} = \theta_1 + \theta_2 \times \text{above_ground_living_area} \quad (2.1)$$

This model comprises two parameters, θ_1 and θ_2 , which can be adjusted to represent any linear relationship.

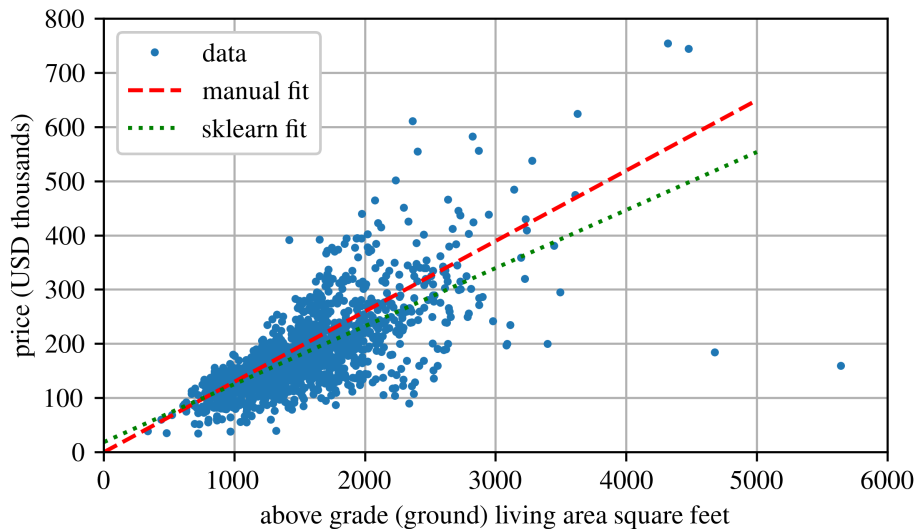


Figure 2.5: Ames housing data with trend lines.

Notations

- **x**: a lowercase, bolded variable that denotes a vector.
- **X**: an uppercase, bolded variable that represents a matrix.
- **X**: this denotes the feature matrix, also known as the input matrix.
- **y**: this is the label vector corresponding to the feature matrix, representing the target outputs.
- $\mathbf{x}^{(i)}$ represents the i^{th} instance in the dataset **X**. For instance, a dataset about cats might include an entry for a specific cat named Mittens. This entry (labelled y) could detail several attributes such as age, weight, and length.
- h : the hypothesis or prediction function that, given a feature vector **X**, outputs the predicted values $\hat{y} = h(\mathbf{X})$.
- n : the total number of features in the dataset, which defines the complexity or degree of the model.
- m : the total number of instances in the dataset.
- \hat{x} : an estimated value, indicated by a hat, and pronounced as “x-hat”.

2.1.1 Closed Form Solution

Before you can tune a model’s parameters, you need a clear way to judge how well it fits the data. The standard approach is to define a cost function that assigns a numerical penalty to poor fits; lower values indicate better performance. For a linear model, this cost function measures the difference between the model’s predictions and the actual training targets, and you minimize that difference using Ordinary Least Squares linear regression. Practically, you feed your training data into the algorithm, which then solves for the parameter values that align the linear relationship most closely to the data. In Python, this procedure is implemented in `scikit-learn` as `sklearn.linear_model.LinearRegression`.

In the context of the Ames housing data, a straightforward cost function used is the Mean Squared Error (MSE), defined as

$$\text{MSE} = J = \frac{1}{m} \sum_{i=1}^m (\hat{y}_i - y_i)^2 \quad (2.2)$$

where m is the number of dataset instances, and J serves as a general representation of the cost function in machine learning contexts.

The MSE is also applied to minimize the error in a linear regression model, with the hypothesis h_{θ} , trained on dataset **X**. This is expressed as:

$$\text{MSE}(\mathbf{X}, h_{\theta}) = J = \frac{1}{m} \sum_{i=1}^m (\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)})^2 \quad (2.3)$$

where the objective is to minimize the cost function by iteratively refining the values of $\boldsymbol{\theta}$.

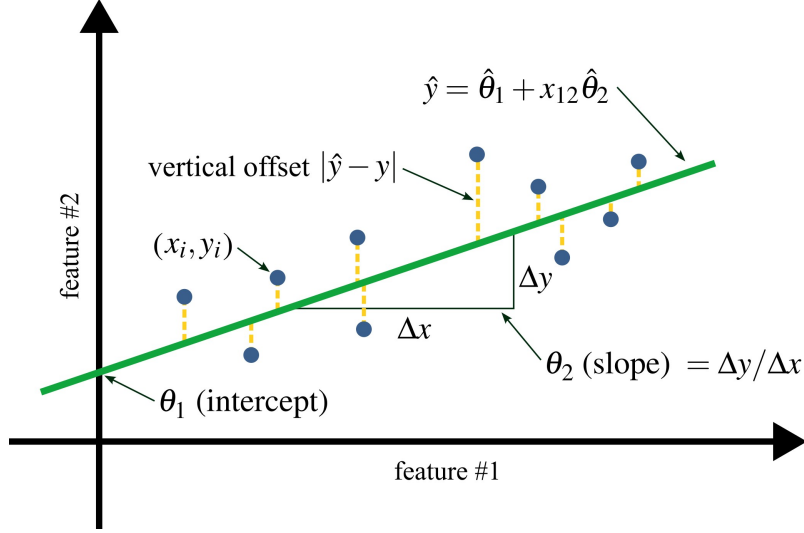


Figure 2.6: Linear regression of a two-feature dataset obtained with least squares.

Consider a dataset containing n observations, denoted as $(y_i, x_i)_{i=1}^n$. Each observation i consists of a scalar response y_i and a column vector x_i that holds the values for p predictors (regressors), represented as x_{ij} where $j = 1, \dots, p$. In the context of a linear regression model, the response variable y_i is modeled as a linear combination of the regressors, influenced by an error term ϵ_i :

$$y_i = \theta_1 x_{i1} + \theta_2 x_{i2} + \dots + \theta_m x_{im} + \epsilon_i \quad (2.4)$$

This model can also be written in matrix notation as

$$\mathbf{y} = \mathbf{X}\boldsymbol{\theta} + \boldsymbol{\epsilon} \quad (2.5)$$

Where:

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1m} \\ x_{21} & x_{22} & \dots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{nm} \end{bmatrix}, \quad \boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_p \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \quad (2.6)$$

However, the best-fit model \hat{y} will not be able to account for the unmodeled error, therefore:

$$\hat{y}_i = \hat{\theta}_1 x_{i1} + \hat{\theta}_2 x_{i2} + \dots + \hat{\theta}_m x_{im} \quad (2.7)$$

or

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\boldsymbol{\theta}} \quad (2.8)$$

This goal is to find $\hat{\boldsymbol{\theta}}$ such that the error is minimized. Mathematically, this is simple enough as the $\hat{\boldsymbol{\theta}}$ is the minimization of the least-squares hyperplane, or:

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} \quad (2.9)$$

Equation 2.9 is referred to as the normal equation. Consider a simple linear model with $p = 2$, where $x_{i1} = 1$. In this scenario, x_{i1} acts as the bias term of the equation, whereas the remaining

elements of \mathbf{X} represent the data. Without this bias term, the solution would only address the slope of the line. Subsequently, solving for $\hat{\theta}_1$ and $\hat{\theta}_2$ yields:

$$[x_{11} x_{12}] [\hat{\theta}_1 \hat{\theta}_2] = \hat{y} \quad (2.10)$$

As $x_{11} = 1$ (again, called the bias term) this becomes:

$$\hat{y} = \hat{\theta}_1 + x_{12} \hat{\theta}_2 \quad (2.11)$$

This is simply another expression for the equation of a liner line:

$$y = mx + b \quad (2.12)$$

Solving the model output for a input or a series of inputs can than be done simply enough.

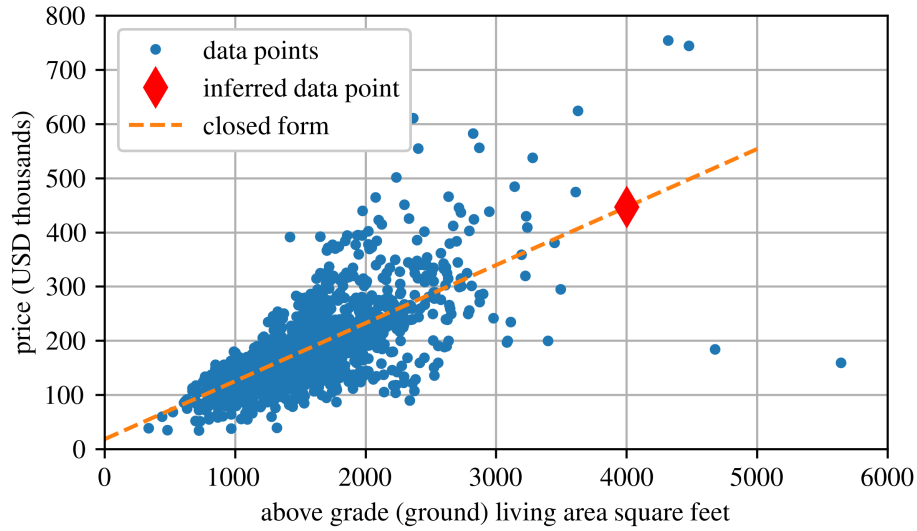


Figure 2.7: Ames housing data inferred.

2.1.2 Computational Complexity

The Normal Equation calculates the inverse of $\mathbf{X}^T \mathbf{X}$, an $n \times n$ matrix, where n represents the number of features. The computational complexity of matrix inversion generally ranges from $O(n^{2.4})$ to $O(n^3)$, depending on the specific algorithm used. Consequently, doubling the number of features would increase the computational effort by approximately $2^{2.4} \approx 5.3$ to $2^3 = 8$ times.

WARNING

When the number of features becomes very large, for example around 100,000, solving the Normal Equation becomes extremely slow.

On the positive side, the computational complexity of this equation is linear with respect to the number of instances in the training set, denoted as $O(m)$. This allows it to efficiently handle large training sets, as long as they fit within memory.

Furthermore, once your Linear Regression model is trained (whether through the Normal Equation or another method), making predictions is very quick. The computational effort is linear in relation to both the number of instances you wish to predict and the number of features. Essentially, predicting for twice as many instances or features will approximately double the computation time.

Next, we will explore different methods for training a Linear Regression model that are more appropriate for scenarios with a large number of features, or when the training data exceeds memory capacity.

Example 2.1 Linear Regression - Ames Housing Dataset

This example uses the Ames housing dataset to model the relationship between above-ground living area and house price using linear regression. It compares a manual fit, a closed-form solution, and gradient descent, alongside Scikit-Learn's `LinearRegression` and `SGDRegressor`, to illustrate how each method fits a line to the data.

Gradient Descent is a versatile optimization algorithm capable of finding optimal solutions across a broad spectrum of problems. The core concept of Gradient Descent involves iteratively adjusting parameters to minimize a cost function^a. Consider the analogy of being lost in a dense fog in the mountains and seeking to descend:

1. You sense the ground's slope under your feet (the local gradient).
2. You descend in the direction of the steepest descent.
3. Reaching a point where the local gradient is zero indicates that you have found a minimum.

The Gradient Descent process starts with random initialization of the parameter θ , followed by gradual improvements. Each incremental step aims to reduce the cost function until the algorithm converges to a minimum.

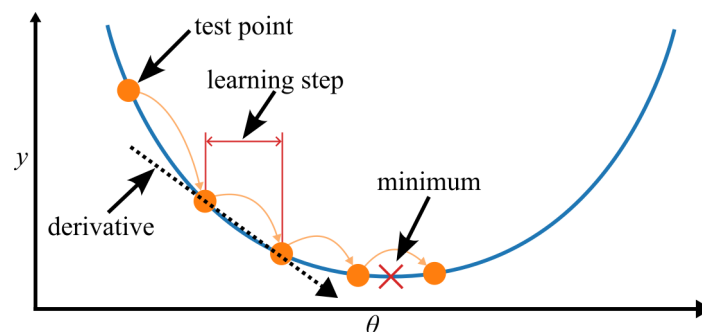


Figure 2.8: Illustration of gradient descent.

A critical aspect of Gradient Descent is the step size, controlled by the learning rate hyperparameter. A small learning rate leads to a slow convergence, requiring many iterations. Conversely, a high learning rate might cause the algorithm to overshoot the minimum, potentially causing divergence and failing to find an optimal solution.

^aRuder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

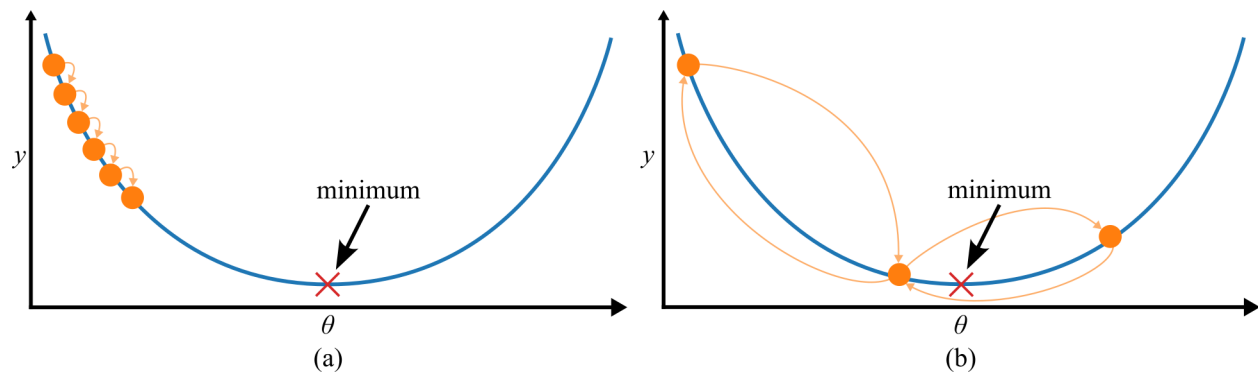


Figure 2.9: Effect of learning rate in gradient descent, showing: (a) a slow search with a low step size, and (b) a search with a large step size that oscillates around the minimum without ever finding the minimum.

Cost functions do not always present themselves as neat, concave shapes. They can feature obstacles such as holes, ridges, plateaus, and various complex topographies that complicate the convergence to the minimum. The challenges associated with Gradient Descent include:

- Starting the algorithm from a random point on the left may lead to convergence at a local minimum rather than the more optimal global minimum.
- Initiating on the right might result in a prolonged journey across a plateau, and terminating the process too soon could prevent reaching the global minimum.

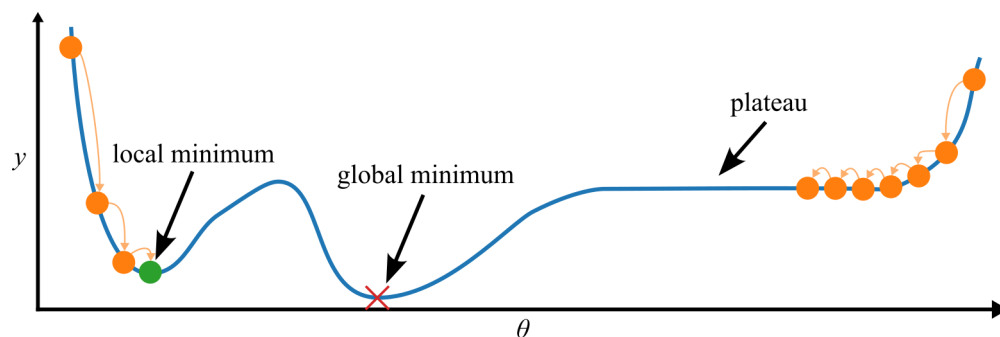


Figure 2.10: Gradient features and descent challenges.

Fortunately, the Mean Squared Error (MSE) cost function for Linear Regression models is convex. This characteristic ensures that for any two points on the curve, the line segment joining them does not intersect the curve itself. Consequently:

- There are no local minima, only one global minimum exists.
- It is a continuous function, and its slope changes smoothly.

These properties significantly benefit the optimization process: Gradient Descent can reliably approximate the global minimum given sufficient time and an appropriate learning rate.

2.1.3 Comparison of Gradient Descent Methods

Before looking into the mathematical details, let us examine how various gradient descent methods achieve the optimal solution, as illustrated in Figure 2.11. Additionally, Table 1 provides a comparative analysis of different gradient descent methods, highlighting their respective strengths and weaknesses.

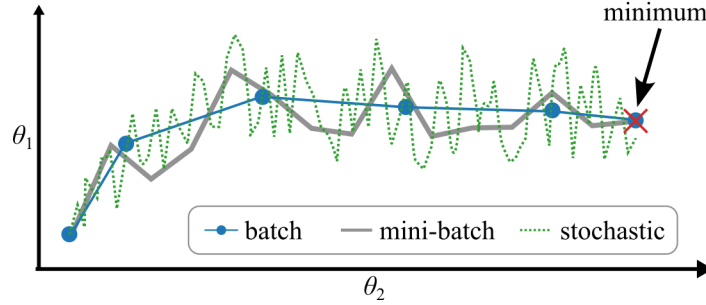


Figure 2.11: Comparing gradient descent methods.

Table 1: Comparison of various linear regression solutions.

algorithm	large number of instances (m)	data must fit in memory	large number of features (n)	hyperparams	scaling required	Scikit-Learn
normal equation	fast	yes	slow	0	no	LinearRegression
batch GD	slow	yes	fast	2	yes	n/a
stochastic GD	fast	no	fast	≥ 2	yes	SGDRegressor
mini-batch GD	fast	no	fast	≥ 2	yes	n/a

NOTE

After training, there is minimal difference between the models: these algorithms converge on similar solutions and make predictions in a nearly identical manner.

2.1.4 Batch Gradient Descent

Batch Gradient Descent (shown in figure 2.12) processes the entire batch of training data \mathbf{X} at every step, which is the origin of its name, “batch.” Consider again our Mean Squared Error (MSE) cost function:

$$J = \frac{1}{m} \sum_{i=1}^m (\hat{y}_i - y_i)^2 \quad (2.13)$$

For a hypothesis h_{θ} , the MSE for the dataset \mathbf{X} can be computed for each instance $\mathbf{x}^{(i)}$ as follows:

$$J(\mathbf{X}, h_{\theta}) = J = \frac{1}{m} \sum_{i=1}^m (\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)})^2 \quad (2.14)$$

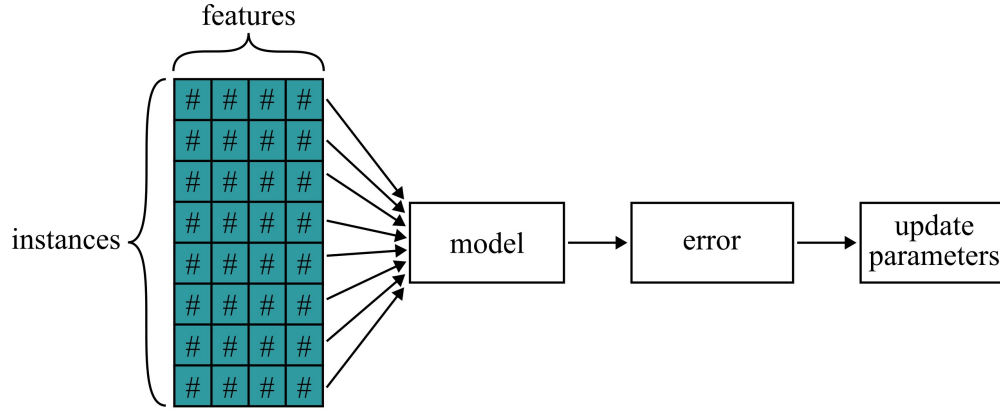


Figure 2.12: Flowchart of Batch Gradient Descent.

With this foundation, we can implement gradient descent by computing the gradient of the cost function with respect to each parameter θ_j . Specifically, this involves calculating how much the cost function changes when θ_j is altered slightly. This calculation is known as a partial derivative. Conceptually, it is akin to determining “the slope of the mountain under my feet if I face east,” then repeating the question facing north, and similarly for any other direction in a higher-dimensional space. The partial derivative of the cost function with respect to parameter θ_j is computed as:

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{2}{m} \sum_{i=1}^m (\theta^T \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)} \quad (2.15)$$

Rather than computing each partial derivative individually, all partial derivatives can be calculated simultaneously using the gradient vector, $\nabla_{\theta} J(\theta)$, which encompasses all partial derivatives of the cost function for each model parameter:

$$\nabla_{\theta} J(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} J(\theta) \\ \frac{\partial}{\partial \theta_1} J(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} J(\theta) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T \cdot (\mathbf{X}\theta - \mathbf{y}) \quad (2.16)$$

WARNING

Be aware that this method computes using the entire training set \mathbf{X} at every step of Gradient Descent, which is why the approach is named Batch Gradient Descent. Although this method can be exceedingly slow with large training sets, it performs well with numerous features, outpacing the Normal Equation in training Linear Regression models with extensive features.

Once you obtain the gradient vector, which indicates the direction of ascent, the next step involves moving in the opposite direction, effectively going downhill. This is achieved by subtracting $\nabla_{\theta} J(\theta)$ from θ , incorporating the learning rate η to scale the step size:

$$\theta^{(\text{next step})} = \theta - \eta \nabla_{\theta} J(\theta) \quad (2.17)$$

Interestingly, this process aligns perfectly with the results from the Normal Equation. However, variations in the learning rate η can significantly influence the outcomes. Figure 2.13 illustrates the first 10 steps of Gradient Descent with three different learning rates, with the dashed line marking the starting point.

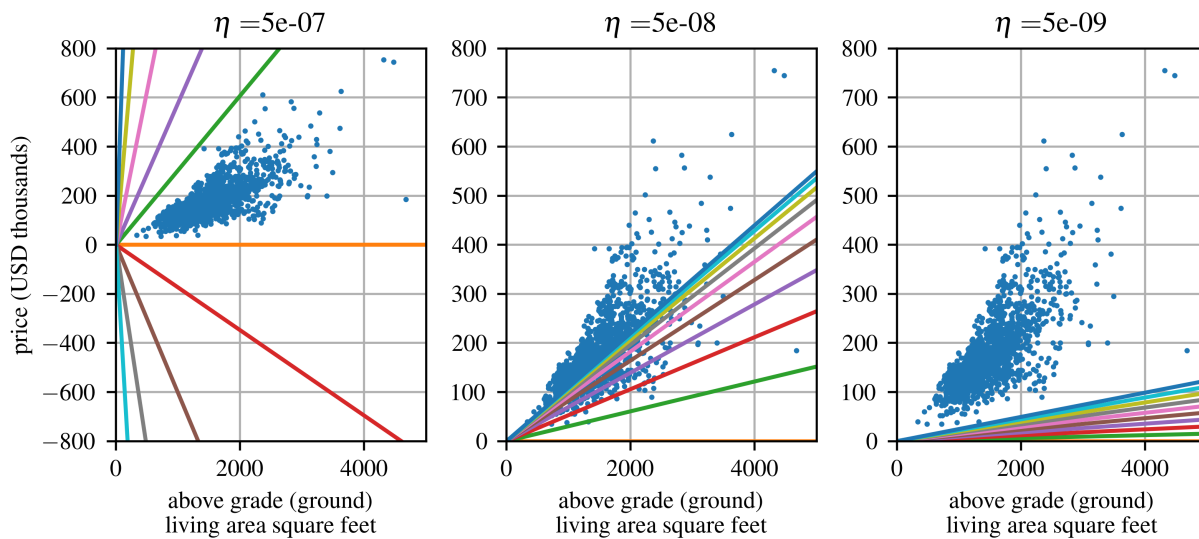


Figure 2.13: The effects of different learning rates on the gradient descent algorithms, showing: (a) too high ($\eta = 5e - 07$); (b) about right ($\eta = 5e - 08$), and; (c) too low ($\eta = 5e - 09$).

Figure 2.13 highlights the impact of the learning rate on gradient descent. Figure 2.13(a) With a very large step size, $\eta = 5 \times 10^{-7}$, each update overshoots the minimum so the algorithm diverges rather than converging. Figure 2.13(b) Using $\eta = 5 \times 10^{-8}$ provides a balanced step size; the cost decreases just right ^a and the algorithm reaches the optimum in only a few iterations. Figure 2.13(c) A small step size, $\eta = 5 \times 10^{-9}$, keeps the algorithm moving toward the minimum yet progress is slow, requiring many more iterations to achieve the same result.

To identify an appropriate learning rate, a grid search can be employed. It is advisable to limit the number of iterations during grid search to avoid models that are too slow to converge.

Determining the correct number of iterations is also crucial. If set too low, the algorithm may stop far from the optimal solution. Conversely, overly high settings lead to unnecessary computations after convergence. A practical approach is to allow a large number of iterations but to halt the algorithm when the gradient vector's norm shrinks to below a small threshold ϵ (known as the tolerance), indicating proximity to the minimum.

^aPyle, K. "Goldilocks and the three bears." Mother's Nursery Tales (1918): 207-213.

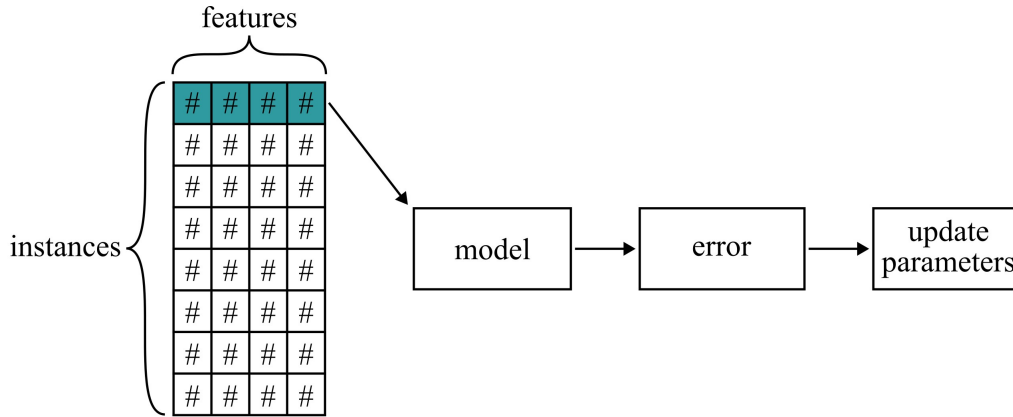


Figure 2.14: Flowchart of Stochastic Gradient Descent.

2.1.5 Stochastic Gradient Descent

Stochastic Gradient Descent (shown in figure 2.14) updates the model parameters after evaluating each individual training sample $x^{(i)}$ and its corresponding label $y^{(i)}$, which stands in contrast to batch gradient descent that uses the entire dataset for each update. The update rule for SGD can be expressed as

$$\theta^{(\text{next step})} = \theta - \eta \nabla_{\theta} J(\theta; x^{(i)}; y^{(i)}). \quad (2.18)$$

While batch gradient descent tends to perform unnecessary recalculations for large datasets by reevaluating gradients for similar examples with each update, SGD eliminates this inefficiency by updating parameters incrementally. Consequently, SGD is generally faster and adaptable for online learning. Due to its frequent and individual updates, SGD exhibits high variance in the objective function, causing significant fluctuations as illustrated in Fig. 2.15(b).

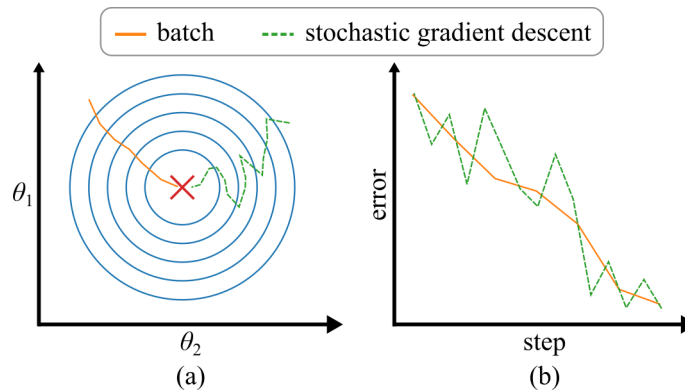


Figure 2.15: Comparison of the iterative performance of Batch and Stochastic Gradient Descent, showing: (a) how they move towards their target, and (b) the error at comparable steps.

2.1.6 Mini-batch Gradient Descent

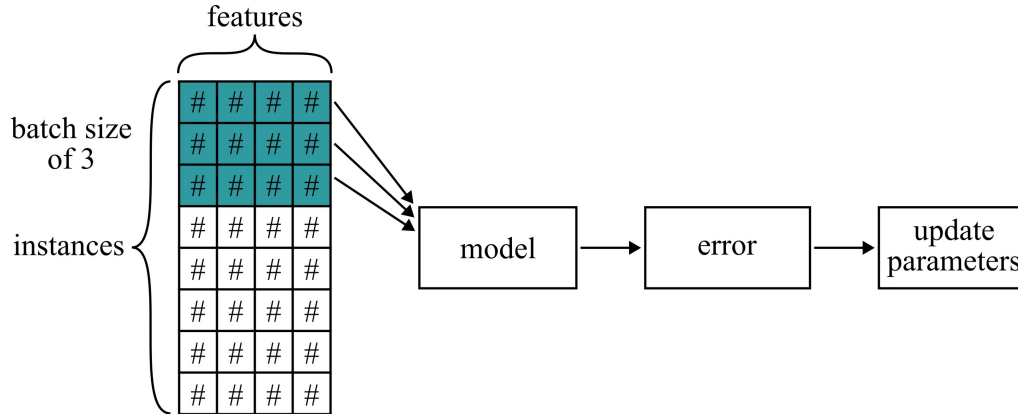


Figure 2.16: Flowchart of Mini-batch Gradient Descent.

$$\theta^{(\text{next step})} = \theta - \eta \nabla_{\theta} J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \quad (2.19)$$

The final Gradient Descent algorithm we will discuss is Mini-batch Gradient Descent. This method combines elements of both Batch and Stochastic Gradient Descent: rather than calculating gradients using the entire training set (as in Batch GD) or a single instance (as in Stochastic GD), Mini-batch GD computes gradients using small, randomly selected subsets of instances known as minibatches. One significant benefit of Mini-batch GD over Stochastic GD is the improved performance from hardware-optimized matrix operations, particularly on GPUs.

Mini-batch GD's trajectory through parameter space tends to be more stable compared to the often erratic path of SGD, especially with larger minibatches. This stability typically brings Mini-batch GD closer to the minimum than SGD. However, Mini-batch GD might struggle more with escaping local minima in scenarios prone to such issues, which is less of a concern in Linear Regression. Figure 2.11 illustrates the trajectories of these three Gradient Descent techniques during training. While Batch GD precisely reaches the minimum, Stochastic GD and Mini-batch GD tend to oscillate nearby. It is important to note, however, that while Batch GD is slow in taking steps, both Stochastic GD and Mini-batch GD could also effectively reach the minimum with an appropriate learning rate schedule.

2.1.7 Feature Scaling

The cost function can resemble an elongated bowl if the feature scales vary significantly. Figure 2.17 illustrates the effect of feature scaling on Gradient Descent. In figure 2.17(a), Gradient Descent progresses directly towards the minimum, reaching it swiftly. However, in figure 2.17(b) it initially moves almost orthogonally to the direction of the global minimum, followed by a lengthy traverse down a nearly flat valley, significantly delaying the convergence.

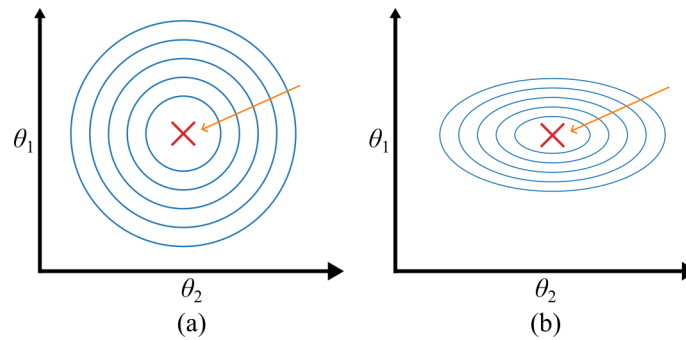


Figure 2.17: Gradient descent on a 2D surface for parameters that are: (a) equally scaled, and (b) unequally scaled.

Training a model can be viewed as a search through the parameter space for the combination that minimizes the cost function. When additional parameters are introduced, this space gains extra dimensions, making the optimization task more challenging. In Ordinary Least Squares Linear Regression, however, the cost surface is convex and bowl-shaped, so any descent method is guaranteed to reach the global minimum. To normalize feature scales, two common methods are employed. They are Min-max scaling and Standardization.

- **Min-max scaling**, often referred to as normalization, is straightforward: it rescales the data to a range of 0 to 1 by subtracting the minimum value and dividing by the range (max minus min).

$$\mathbf{X}' = \frac{\mathbf{X} - \mathbf{X}_{\min}}{\mathbf{X}_{\max} - \mathbf{X}_{\min}} \quad (2.20)$$

Scikit-Learn offers `sklearn.preprocessing.MinMaxScaler` for this purpose.

- **Standardization** differs significantly as it first subtracts the mean (resulting in a zero mean) and then divides by the standard deviation to achieve unit variance. This method does not limit values to a specific range, which can be problematic for certain algorithms, such as neural networks which often expect inputs between 0 and 1. However, standardization is less sensitive to outliers. For instance, if a median income is mistakenly recorded as 100, min-max scaling would compress all other values between 0 and 15 to between 0 and 0.15, whereas standardization would be minimally affected.

$$\mathbf{X}' = \frac{\mathbf{X} - \bar{\mathbf{X}}}{\sigma} \quad (2.21)$$

`sklearn.preprocessing.StandardScaler` is provided by Scikit-Learn for standardization.

WARNING

It is crucial to apply scalers exclusively to the training data and not to the entire dataset, which includes the test set. This practice ensures that the model is not inadvertently exposed to test data during training. Once the scalers are fitted to the training data, they can then be used to transform the training set, the test set, and any new data subsequently encountered.

2.2 Polynomial Regression

What if the underlying pattern of your data is more complex than a simple linear relationship? Consider the dataset shown in figure 2.18 with non-linear characteristics (one feature across multiple instances), which typically can be modeled using a second-order polynomial

$$\hat{y} = ax^2 + bx + c. \quad (2.22)$$

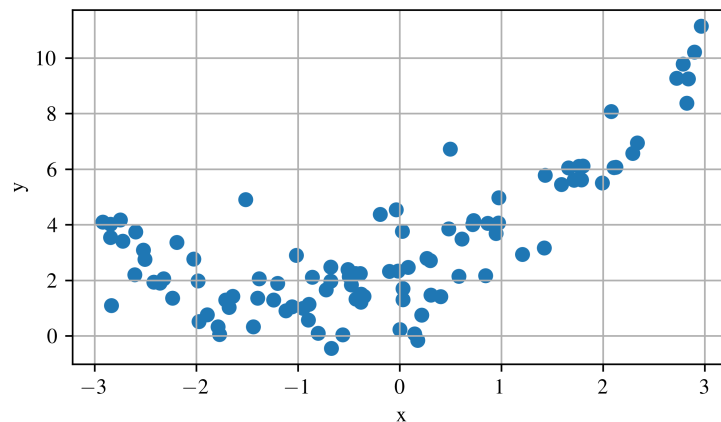


Figure 2.18: Dataset derived from a 2nd-order polynomial.

A simple linear fit will under-perform, so you first enrich the training set by adding the squared term of each feature as an extra column; shown in figure 2.19. Although the model you train is still linear with respect to its parameters, it now operates on an augmented feature space and can represent the desired quadratic curve. This approach is called Polynomial Regression, and it extends naturally by including higher powers whenever more complex nonlinearities are present.

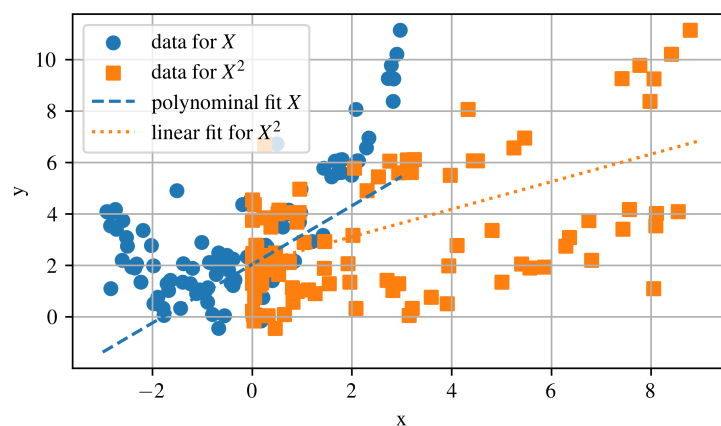


Figure 2.19: Linear models fit to the individual features of a polynomial dataset.

This data can then be incorporated into two linear models, where the slopes of the feature sets serve as the parameters for the base-line polynomial expression (a and b in Equation 2.22) and the bias term is the offset (c in Equation 2.22). The results of such a polynomial fit is shown in figure 2.20.

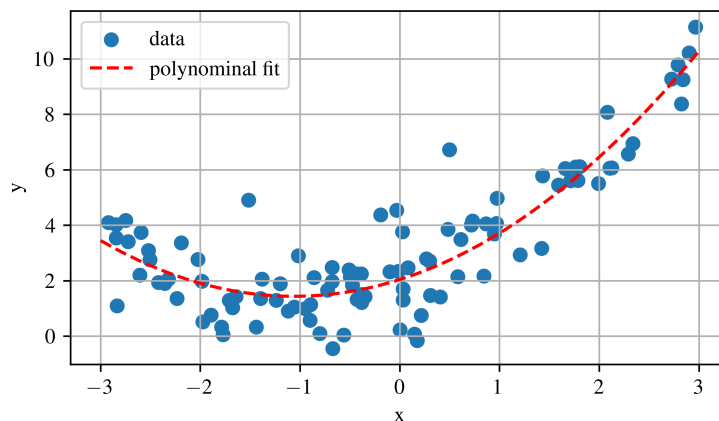


Figure 2.20: Polynomial model fit to a dataset.

It is important to note that Polynomial Regression can identify interrelationships between features in cases where multiple features exist, a capability beyond the scope of simple Linear Regression. This enhancement is enabled by `PolynomialFeatures`, which includes all possible combinations of features up to the specified degree. For instance, with two features a and b , and a degree of 3, `PolynomialFeatures` would add not only a^2 , a^3 , b^2 , and b^3 but also the combined terms ab , a^2b , and ab^2 .

WARNING

`PolynomialFeatures(degree=d)` expands an array with n original features into one that includes $\frac{(n+d)!}{d!n!}$ features, accounting for all combinations of features up to the d -th degree. Here, $n!$ represents the factorial of n , calculated as $1 \times 2 \times 3 \times \dots \times n$. Be cautious of the rapid increase in the number of features, known as combinatorial explosion!

Example 2.2 Polynomial Regression

This example fits a non-linear dataset using polynomial regression by adding x^2 as an extra feature. It demonstrates how linear models can approximate curved relationships when polynomial terms are included, and shows the resulting fit compared to the original data.

2.3 Training and Testing Data

Training a predictive model on a dataset and then testing it with the same data is a fundamental error in methodology. Such a model could simply memorize the labels of the training samples, achieving perfect performance during training but failing to make any useful predictions on new, unseen data. This phenomenon is known as overfitting. To prevent this, it is standard practice in supervised machine learning to reserve a portion of the available data as a test set, denoted as \mathbf{X}_{test} and \mathbf{y}_{test} .

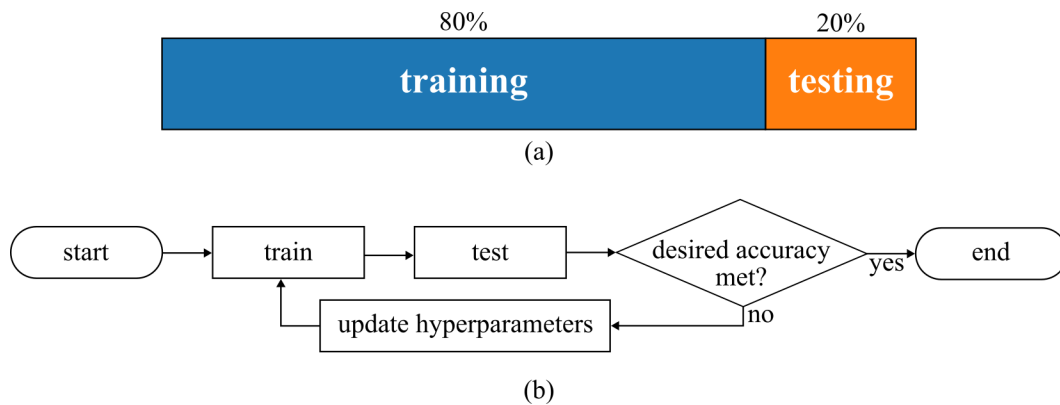


Figure 2.21: Splitting data up into training and testing subsets.

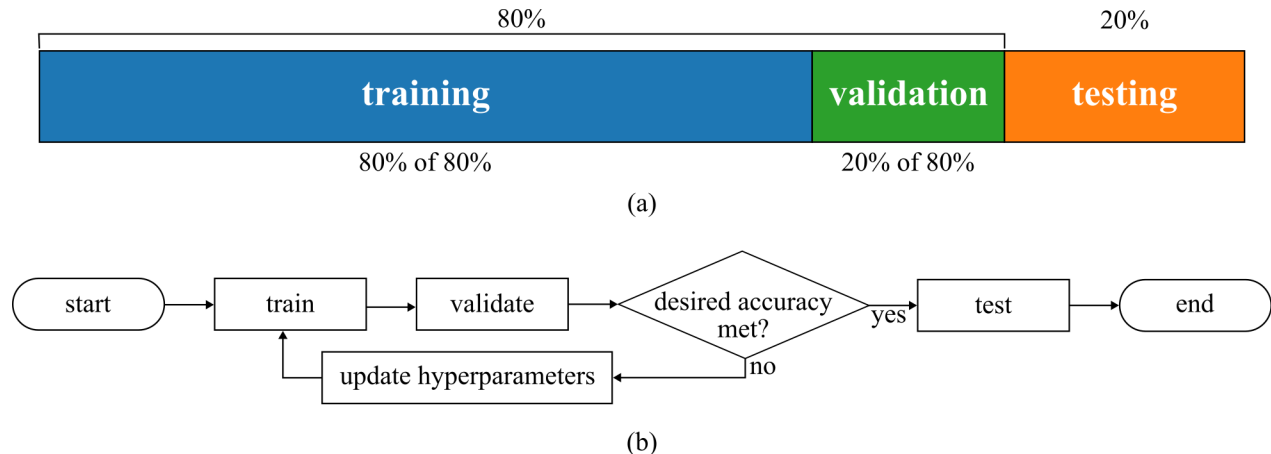


Figure 2.22: Splitting data up into training, validation, and testing subsets.

The function `sklearn.model_selection.train_test_split` in scikit-learn randomly divides arrays or matrices into training and testing subsets.

2.4 Pipelines

Pipelines in scikit-learn streamline the process by sequentially applying a list of transformations followed by a final estimator to a dataset. Employing pipelines allows for the integration of multiple processing steps, which can then be cross-validated together while experimenting with various parameters. Fig. 2.23 illustrates a typical pipeline configuration in scikit-learn.

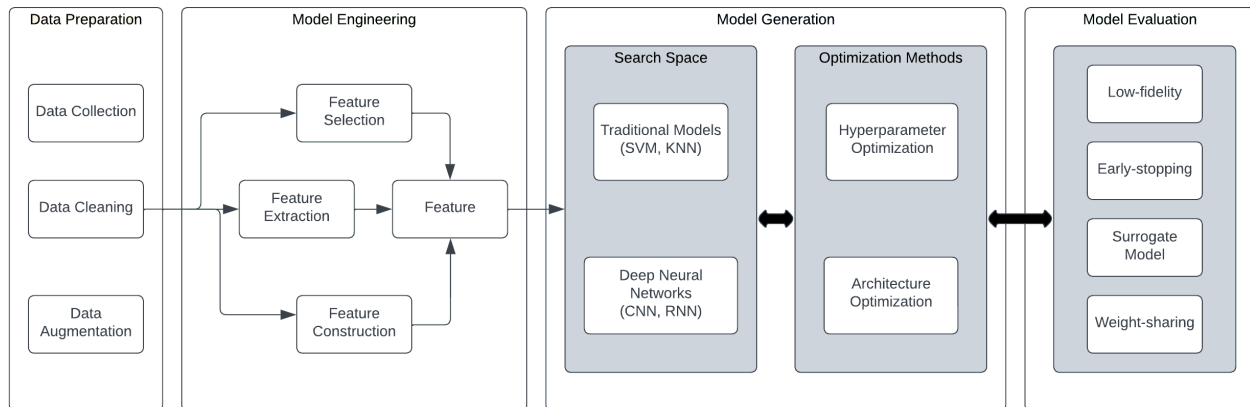


Figure 2.23: Pipeline setup for the automated deployment of pre-processing and modeling steps.^a

2.5 Learning Curves

Performing Polynomial Regression with a high degree typically allows for a closer fit to the training data compared to simple Linear Regression. For instance, Figure 2.24 demonstrates the application of a 30-degree polynomial model to a set of training data, contrasting it with both a linear model and a quadratic model (2nd-degree polynomial). Observe how the 30-degree polynomial model contorts to closely match the training instances.

- The linear model exhibits underfitting.
- The high-degree Polynomial Regression model drastically overfits the training data.
- Among these, the quadratic model is likely to generalize best.

In this instance, the appropriateness of the quadratic model is clear since the data was initially generated using such a model. However, in real-world scenarios where the underlying function of the data is unknown, determining the optimal complexity for your model can be challenging. How can you ascertain whether your model is overfitting or underfitting the data?

^aModified from PopovaZhuhadar, CC BY-SA 4.0 <<https://creativecommons.org/licenses/by-sa/4.0/>>, via Wikimedia Commons

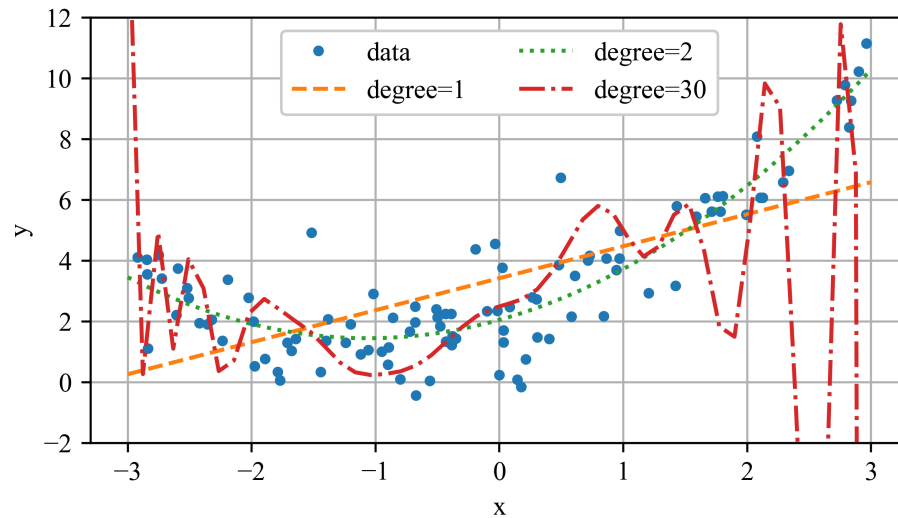


Figure 2.24: Polynomial regression showing underfitting (degree=1), a respectable model fit (degree=2), and overfitting (degree=30).

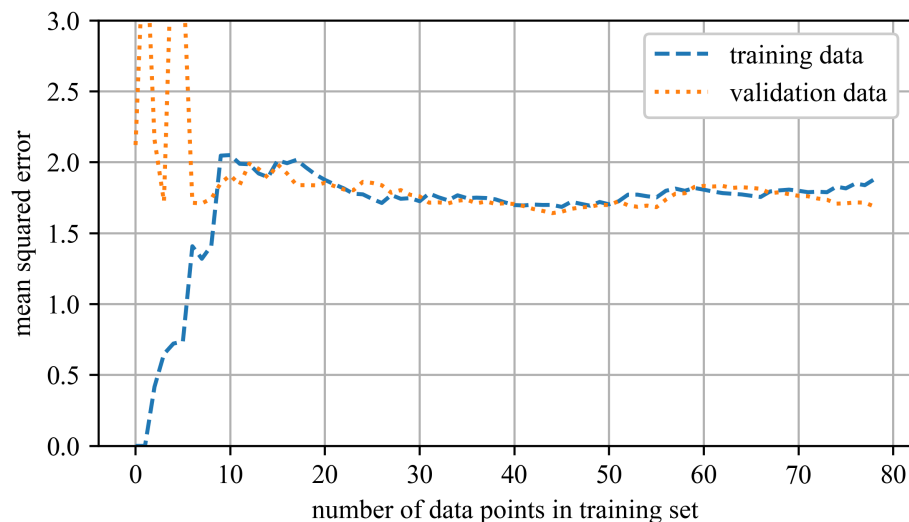


Figure 2.25: Learning curves for the underfitting linear model (degree=1) where underfitting is evident because both curves have plateaued; they are close together and relatively high.

Examining the model's behavior in figures 2.25 and 2.26 on the training set shows a clear trend. With only one or two samples the fit is perfect, so the error curve begins at zero. As additional observations are introduced the model can no longer capture every point exactly, partly because of measurement noise and partly because the underlying relationship is nonlinear, and consequently the training error rises. After a sufficient number of samples the curve flattens out; beyond this plateau adding further data neither markedly lowers nor raises the average error.

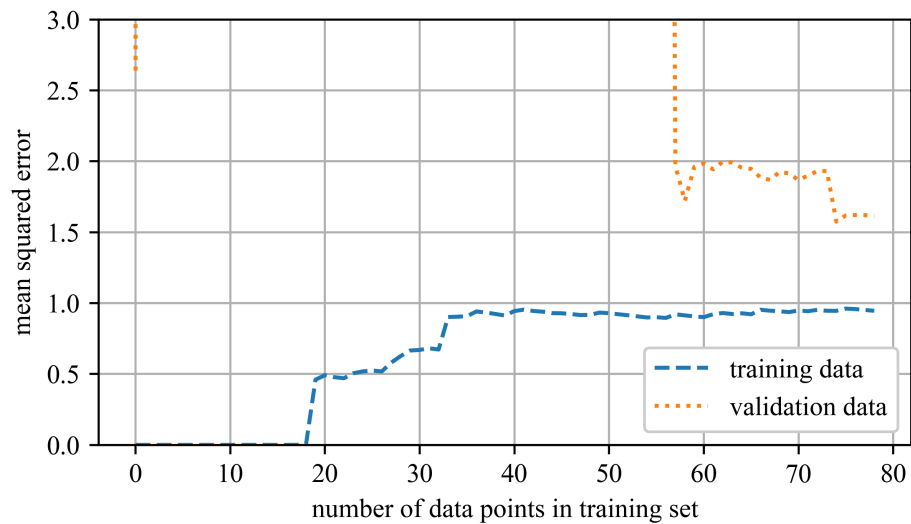


Figure 2.26: Learning curves for the overfitting polynomial model (degree=20) showing a significant gap between the curves, which indicates better performance on the training data than on the validation data.

Inspection of the validation curve in figure 2.26 tells a complementary story. With only a handful of training samples the model cannot generalize, so the validation error starts high. As more examples become available it learns progressively better patterns and the error falls. Eventually, though, the simple linear hypothesis is unable to capture the data's full complexity, causing the decline to flatten out; the validation curve plateaus at nearly the same level as the training curve. Their close proximity and uniformly large values are characteristic of an underfitting model.

Figure 2.26 displays the learning curves for a model fitted with a 20th-degree polynomial. The overall shape resembles the curves seen earlier, yet two key differences stand out. First, the error on the training set is far lower than that achieved by the Linear Regression model, demonstrating the polynomial's extra flexibility. Second, a pronounced gap separates the training and validation curves, which means the model performs much better on the data it has already seen; this divergence is the hallmark of overfitting. Expanding the training set could reduce the gap, but without additional regularization the risk of overfitting would remain.

Next, let's apply the code using a 2nd order polynomial, which accurately captures the essence of the data without underfitting or overfitting. These results are shown in Figure 2.27. Here it can be seen that the training and validation curves again meet after converging, showing a well-fit model.

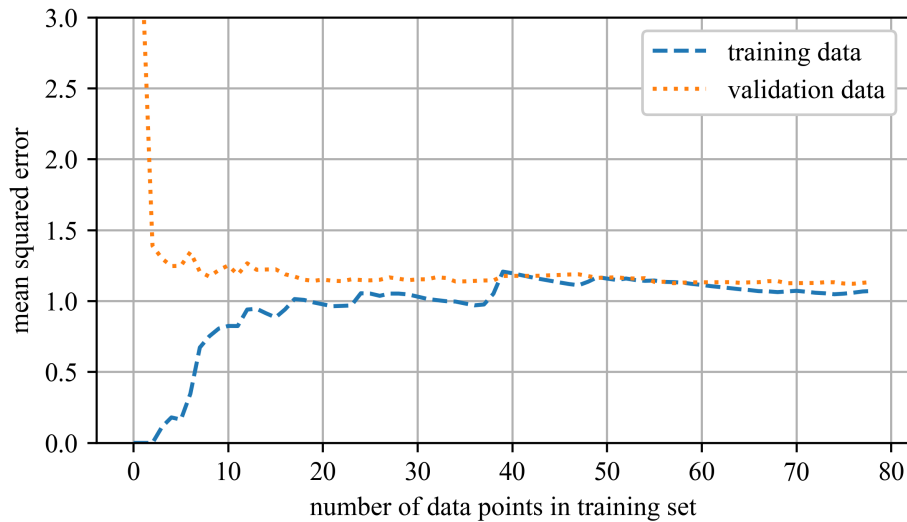


Figure 2.27: Learning curves for the optimal polynomial model (degree=2) showing well-aligned curves that plateau at a relatively low error level.

Example 2.3 Learning Curves

This example compares learning curves for linear and polynomial regression models. By plotting training and validation errors as the training set grows, it illustrates how model complexity impacts bias and variance, and helps identify underfitting and overfitting behavior.

2.6 Regularized Linear Models

Overfitting is frequently curbed by regularizing the model, meaning you introduce constraints that reduce its effective degrees of freedom and make it harder to memorize noise. In a polynomial setting you can achieve this simply by choosing a lower-degree polynomial, while in a linear model you typically add a penalty that discourages large coefficient values. This naturally raises the question: how is regularization expressed mathematically?

2.6.1 Ridge Regression

Ridge Regression modifies Linear Regression by adding a regularization term to the cost function, which compels the learning algorithm to fit the data while keeping the model weights as small as possible. The updated cost-function is

$$J(\theta) = \text{MSE}(\theta) + \alpha \frac{1}{2} \sum_{i=1}^n \theta_i^2. \quad (2.23)$$

The hyperparameter α dictates the degree of regularization. If $\alpha = 0$, Ridge Regression reduces to plain Linear Regression. If α is very high, the model weights are significantly shrunk, resulting in a nearly flat line through the mean of the data. Equation 2.23 outlines the cost function for Ridge Regression. The bias term θ_0 is not regularized, as indicated by the summation starting from $i = 1$.

NOTE

The regularization term is only included during the training phase. After training, the model's performance should be evaluated based on the unregularized performance measure.

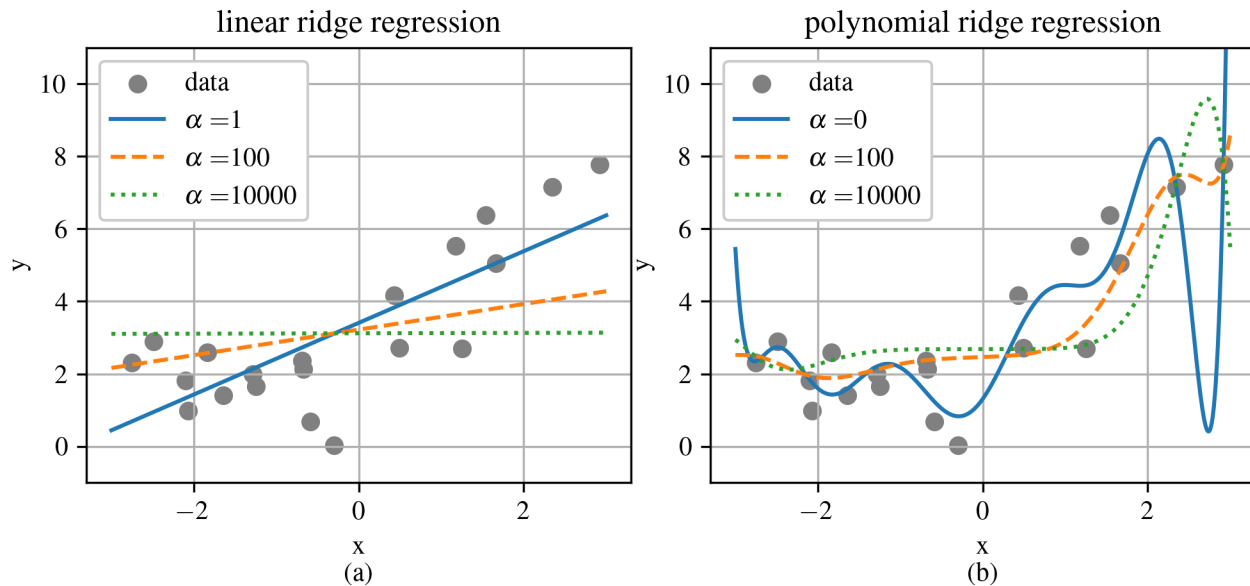


Figure 2.28: Visualization of Ridge Regression models with varying α values.

Figure 2.28 illustrates various Ridge models trained on linear data, showcasing different α value. In Figure 2.28(a), models are purely Ridge Regression, leading to linear predictions. In Figure 2.28(b), the data undergoes Polynomial Regression with Ridge regularization. To do this, the data is first expanded using `PolynomialFeatures(degree=10)`, then scaled with `StandardScaler`, and finally, Ridge models are applied to these transformed features.

Increasing α results in flatter (i.e., more reasonable and less extreme) predictions, thereby reducing the model's variance but increasing its bias. Ridge Regression can be performed using a closed-form solution or through Gradient Descent, similar to Linear Regression. The pros and cons for each method mirror those discussed previously. The closed-form solution, an extension of the normal equation, is given by:

$$\hat{\theta} = (X^T \cdot X + \alpha A)^{-1} \cdot X^T \cdot y \quad (2.24)$$

where α influences the regularization and A is an $n \times n$ identity matrix with the top-left value replaced by 0 to exclude the bias term. When using Gradient Descent, the derivative of the cost function guides the adjustment toward optimal model parameters.

Example 2.4 Ridge Regression

This example demonstrates Ridge Regression as a regularized alternative to linear regression. A high-degree polynomial model is fit to noisy data using a pipeline with Ridge regularization, showing how the regularization term reduces overfitting and stabilizes the model.

2.7 Early Stopping

An alternative approach to regularizing iterative learning methods (such as Gradient Descent) is to halt training as soon as the validation error hits its lowest point, a strategy known as early stopping. Figure 2.29 illustrates a complex model, specifically a high-degree Polynomial Regression model, being refined through Batch Gradient Descent. Over the course of training, as epochs progress, the model's prediction error (RMSE) on both the training and validation sets decreases. However, after some time, the validation error ceases to decline and begins to increase, signaling that the model is beginning to overfit the training data. By implementing early stopping, training is terminated the moment the validation error reaches its minimum. Geoffrey Hinton praised this method stating, “Early stopping (is) beautiful free lunch” due to its simplicity^a.

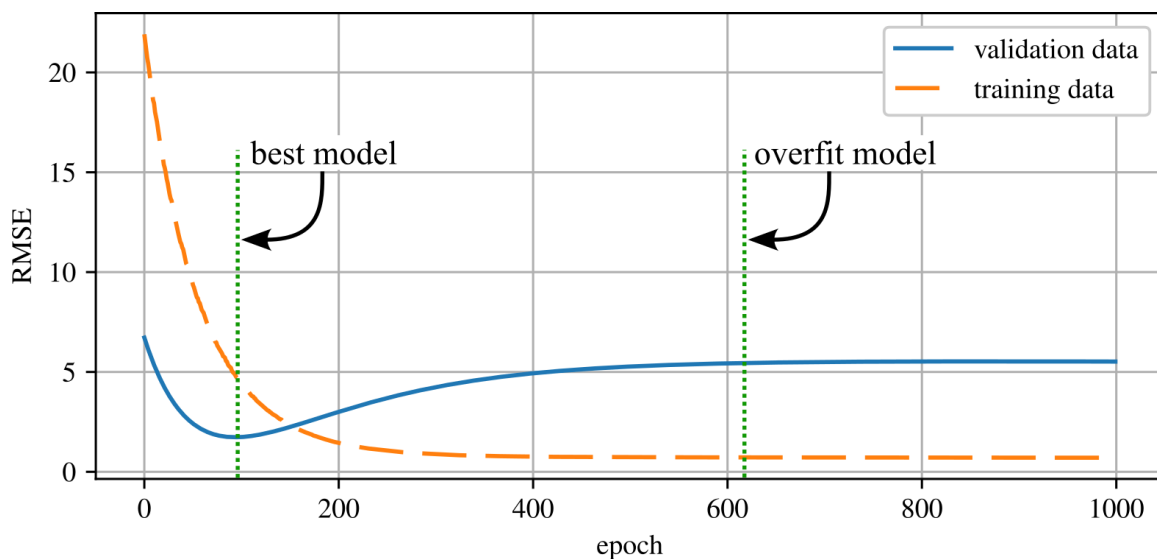


Figure 2.29: An example showing how early stopping can result in a better model as it does not allow for the over-fitting of the training set.

Example 2.5 Early Stopping

This example uses a high-degree polynomial model trained with stochastic gradient descent to demonstrate early stopping. By tracking training and validation errors across epochs, it highlights how halting training early can prevent overfitting and improve generalization.

^aPennington, Jeffrey, Richard Socher, and Christopher D. Manning. “Glove: Global vectors for word representation.” Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). 2014.

2.8 Examples

Example 2.1

```

1  """
2  Example 2.1 Linear Regression
3  @author: Austin Downey
4  """
5
6  import IPython as IP
7  IP.get_ipython().run_line_magic('reset', '-sf')
8
9  import numpy as np
10 import matplotlib.pyplot as plt
11 import sklearn as sk
12 from sklearn import datasets
13 from sklearn.linear_model import LinearRegression
14
15 plt.close('all')
16
17 %% load data
18
19 ames = sk.datasets.fetch_openml(name="house_prices", as_frame=True, parser='auto')
20 target = ames['target'].values
21 data = ames['data']
22 YrSold = data['YrSold'].values # Year Sold (YYYY)
23 MoSold = data['MoSold'].values # Month Sold (MM)
24 OverallCond = data['OverallCond'].values # OverallCond: Rates the overall condition of
    the house
25 GrLivArea = data['GrLivArea'].values # Above grade (ground) living area square feet
26 BedroomAbvGr = data['BedroomAbvGr'].values # Bedrooms above grade (does NOT include
    basement bedrooms)
27
28 # Ask a home buyer to describe their dream house, and they probably won't begin
29 # with the height of the basement ceiling or the proximity to an east-west railroad.
30 # But this playground competition's dataset proves that much more influences price
31 # negotiations than the number of bedrooms or a white-picket fence.
32
33 # With 79 explanatory variables describing (almost) every aspect of residential
34 # homes in Ames, Iowa, this competition challenges you to predict the final price
35 # of each home.
36
37 # Plot a few of the interesting features vs the target (price). In particular,
38 # let's plot the number of rooms vs. the price.
39 plt.figure()
40 plt.plot(GrLivArea, target, 'o', markersize=2)
41 plt.xlabel('Above grade (ground) living area square feet')
42 plt.ylabel('price (USD)')
43 plt.grid(True)
44 plt.tight_layout()
45
46 %% Build a model for the data
47 X = GrLivArea
48 Y = target
49 model_X = np.linspace(0, 5000)
50
51 theta_1 = 0
52 theta_2 = 100
53 model_Y_manual = theta_1 + theta_2 * model_X
54
55 plt.figure()
56 plt.plot(X, Y, 'o', markersize=2, label='data')
57 plt.plot(model_X, model_Y_manual, '--', label='manual fit')
58 plt.xlabel('Above grade (ground) living area square feet')
59 plt.ylabel('price (USD)')
60 plt.grid(True)
61 #plt.xlim([3.5, 9])
62 #plt.ylim([0, 50000])
63 plt.legend(framealpha=1)
64 plt.tight_layout()
65

```



```

66 # add a dimension to the data as math is easier in 2d arrays and sk learn only
67 # takes 2d arrays
68 X = np.expand_dims(X,axis=1)
69 Y = np.expand_dims(Y,axis=1)
70 model_X = np.expand_dims(model_X,axis=1)
71
72 %% compute the linear regression solution using the closed form solution
73
74 # compute
75 X_b = np.ones((X.shape[0],2))
76 X_b[:,1] = X.T # add x0 = 1 to each instance
77
78 theta_closed_form = np.linalg.inv(X_b.T@X_b)@X_b.T@Y
79
80 model_Y_closed_form = theta_closed_form[0] + theta_closed_form[1]*3000
81 model_Y_closed_form = theta_closed_form[0] + theta_closed_form[1]*model_X
82
83 plt.figure()
84 plt.plot(X,Y,'o',markersize=3,label='data points')
85 plt.xlabel('Above grade (ground) living area square feet')
86 plt.ylabel('price (USD)')
87 plt.plot(3000,model_Y_closed_form,'dr',markersize=10,zorder=10,
88         label='inferred data point')
89 plt.plot(model_X,model_Y_closed_form,'-',label='normal equation')
90 plt.grid(True)
91 plt.legend()
92 plt.tight_layout()
93
94 %% compute the linear regression solution using gradient descent
95
96 eta = 0.00000001 # learning rate
97 n_iterations = 100
98 m = X.shape[0]
99 theta_gradient_descent = np.random.randn(2,1) # random initialization
100 for iteration in range(n_iterations):
101     gradients = 2/m * X_b.T.dot(X_b.dot(theta_gradient_descent) - Y)
102     theta_gradient_descent = theta_gradient_descent - eta * gradients
103
104 print(theta_gradient_descent)
105
106 model_Y_gradient_descent = theta_gradient_descent[0] \
107     + theta_gradient_descent[1]*model_X
108
109 plt.figure()
110 plt.plot(X,Y,'o',markersize=3,label='data points')
111 plt.xlabel('Above grade (ground) living area square feet')
112 plt.ylabel('price (USD)')
113 plt.plot(model_X,model_Y_closed_form,'-',label='normal equation')
114 plt.plot(model_X,model_Y_gradient_descent,':',label='gradient descent')
115 plt.grid(True)
116 plt.legend()
117 plt.tight_layout()
118
119 %% compute the linear regression solution using sk-learn
120
121 # build and train a closed from linear regression model in sk-learn
122 model_LR = sk.linear_model.LinearRegression()
123 model_LR.fit(X,Y[:,0])
124 model_Y_sk_LR = model_LR.predict(model_X)
125
126 # build and train a Stochastic Gradient Descent linear regression model in sk-learn.
127 # Note that in running the model, the best way to do this would be to use a pipeline
128 # =with feature scaling. However, here we just set 'eta0' to a low value, this
129 # is done only for educational # purposes and is not the ideal methodology in
130 # terms of system robustness.
131 model_SGD = sk.linear_model.SGDRegressor(learning_rate='constant',eta0=0.00000001)
132 model_SGD.fit(X,Y[:,0])

```

```
133 model_Y_sk_SGD = model_SGD.predict(model_X)
134
135 # plot the modeled results
136 plt.figure()
137 plt.plot(X,Y,'o',markersize=2,label='data')
138 plt.plot(model_X,model_Y_closed_form,'-',label='normal equation')
139 plt.plot(model_X,model_Y_gradient_descent,'--',label='gradient descent')
140 plt.plot(model_X,model_Y_sk_LR,':',label='sklearn normal equation')
141 plt.plot(model_X,model_Y_sk_SGD,'-.',label='sklearn stochastic gradient descent')
142
143 plt.xlabel('Above grade (ground) living area square feet')
144 plt.ylabel('price (USD)')
145 plt.grid(True)
146 plt.legend(framealpha=1)
147 plt.tight_layout()
```

Example 2.2

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Example 2.2
5  Polynomial regression
6  Machine Learning for Engineering Problem Solving
7  @author: Austin Downey
8  """
9
10 import IPython as IP
11 IP.get_ipython().run_line_magic('reset', '-sf')
12
13 import numpy as np
14 import scipy as sp
15 import matplotlib as mpl
16 import matplotlib.pyplot as plt
17 import sklearn as sk
18 from sklearn import linear_model
19
20 plt.close('all')
21
22 %% build the data sets
23 np.random.seed(2) # 2 and 6 are pretty good
24 m = 100
25 X = 6 * np.random.rand(m,1) - 3
26 Y = 0.5 * X**2 + X + 2 + np.random.randn(m,1)
27
28 # plot the data
29 plt.figure()
30 plt.grid(True)
31 plt.scatter(X,Y)
32 plt.xlabel('x')
33 plt.ylabel('y')
34
35 %% perform polynomial regression
36
37 # generate x^2 as we use the model y = a*x^2 + b*x + c
38 X_poly_manual = np.hstack((X,X**2))
39
40 # or use the code as this does lots of features for multi-feature data sets.
41 poly_features = sk.preprocessing.PolynomialFeatures(degree=2, include_bias=False)
42 X_poly_sk = poly_features.fit_transform(X)
43
44 # they do do same thing as shown below, so select one to carry forward.
45 print(X_poly_manual == X_poly_sk)
46 X_poly = X_poly_manual
47
48 # In essence, we now have two data sets. We can plot that here
49 plt.figure()
50 plt.grid(True)
51 plt.scatter(X_poly[:,0],Y,label = 'data for x')
52 plt.scatter(X_poly[:,1],Y,marker='s',label = 'data for $x^2$')
53 plt.legend()
54 plt.xlabel('x')
55 plt.ylabel('y')
56
57 # and fit linear models to these data sets
58 model = sk.linear_model.LinearRegression() # Select a linear model
59 model.fit(X_poly,Y) # Train the model
60 X_model_1 = np.linspace(-3,3)
61 X_model_2 = np.linspace(0,9)
62
63 # the model parameters are:
64 model_coefficients = model.coef_
65 model_intercept = model.intercept_
66 print(model_coefficients)
67 print(model_intercept)

```

```
68
69 Y_X1 = model_coefficients[0][0]*X_model_1 + model_intercept
70 Y_X2 = model_coefficients[0][1]*X_model_2 + model_intercept
71
72 # now if we plot the linear models on the extended set of features.
73 plt.figure()
74 plt.grid(True)
75 plt.scatter(X_poly[:,0],Y,label = 'data for x')
76 plt.scatter(X_poly[:,1],Y,marker='s',label = 'data for $x^2$')
77 plt.plot(X_model_1,Y_X1,'--',label='linear fit $x$')
78 plt.plot(X_model_2,Y_X2,':',label='linear fit for $x^2$',)
79 plt.legend()
80 plt.xlabel('x')
81 plt.ylabel('y')
82 plt.savefig('example_6_fig_1',dpi=300)
83
84 # now that we have a parameter for x and x^2, these can be recombined into a single
85 # model, y = x^2*a + x*b + c.
86 Y_polynomial = model_coefficients[0][1]*X_model_1**2 + model_coefficients[0][0]*\
87     X_model_1 + model_intercept
88
89 plt.figure()
90 plt.grid(True)
91 plt.scatter(X,Y,label='data')
92 plt.plot(X_model_1,Y_polynomial,'r--',label='polynomial fit')
93 plt.xlabel('x')
94 plt.ylabel('y')
95 plt.legend()
96 plt.savefig('example_6_fig_2',dpi=300)
97
98
99
100
101
```

Example 2.3

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Example 2.3
5  Learning curves
6  Machine Learning for Engineering Problem Solving
7  @author: Austin Downey
8  """
9
10 import IPython as IP
11 IP.get_ipython().run_line_magic('reset', '-sf')
12
13 import numpy as np
14 import matplotlib.pyplot as plt
15 import sklearn as sk
16 from sklearn import linear_model
17 from sklearn import pipeline
18
19 plt.close('all')
20
21 %% build the data sets
22 np.random.seed(2) # 2 and 6 are pretty good
23 m = 100
24 X = 6 * np.random.rand(m,1) - 3
25 Y = 0.5 * X**2 + X + 2 + np.random.randn(m,1)
26 X_model = np.linspace(-3,3)
27
28 # plot the data
29 plt.figure()
30 plt.grid(True)
31 plt.scatter(X,Y)
32 plt.xlabel('x')
33 plt.ylabel('y')
34
35 %% generate learning curves for a linear model
36
37 # build the linear model in SK learn
38 model = sk.linear_model.LinearRegression()
39
40 # split the data into training and validation data sets
41 # Split arrays or matrices into random train and test subsets
42 X_train, X_val, y_train, y_val = sk.model_selection.train_test_split(X, Y, test_size=0.2)
43
44 train_errors, val_errors = [], []
45 for i in range(1, len(X_train)):
46     model.fit(X_train[:i], y_train[:i])
47     y_train_predict = model.predict(X_train[:i])
48     y_val_predict = model.predict(X_val)
49
50     # compute the error for the trained model
51     mse_train = sk.metrics.mean_squared_error(y_train[:i], y_train_predict)
52     train_errors.append(mse_train)
53
54     # compute the error for the validation model
55     mse_val = sk.metrics.mean_squared_error(y_val, y_val_predict)
56     val_errors.append(mse_val)
57
58 # predict model
59 y_model = model.predict(np.expand_dims(X_model,axis=1))
60
61 plt.figure('test model')
62 plt.scatter(X,Y,s=2, label='data')
63 plt.scatter(X_train[:i],y_train[:i], label='data in training set')
64 plt.scatter(X_val,y_val, marker='s', label='validation data')
65 plt.plot(X_model,y_model,'r--',label='model')
66 plt.xlabel('x')
67 plt.ylabel('y')

```

```

68     plt.legend(loc=2)
69     plt.grid(True)
70     plt.savefig('test_plots/linear_model_'+str(i))
71     plt.close('test model')
72
73 plt.figure()
74 plt.grid(True)
75 plt.plot(train_errors, "--", label="train")
76 plt.plot(val_errors, ":", label="val")
77 plt.xlabel('number of data points in training set')
78 plt.ylabel('mean squared error')
79 plt.legend(framealpha=1)
80 plt.ylim(0,6)
81 ### generate learning curves for a polynomial model
82
83 model = sk.pipeline.Pipeline((
84     ("poly_features", sk.preprocessing.PolynomialFeatures(degree=20, include_bias=False)),
85     ("lin_reg", sk.linear_model.LinearRegression()),
86 ))
87
88 # split the data into training and validation data sets
89 # Split arrays or matrices into random train and test subsets
90 X_train, X_val, y_train, y_val = sk.model_selection.train_test_split(X, Y, test_size=0.2)
91
92 train_errors = []
93 val_errors = []
94 for i in range(1, len(X_train)):
95     model.fit(X_train[:i], y_train[:i])
96     y_train_predict = model.predict(X_train[:i])
97     y_val_predict = model.predict(X_val)
98
99     # compute the error for the trained model
100     mse_train = sk.metrics.mean_squared_error(y_train[:i], y_train_predict)
101     train_errors.append(mse_train)
102
103     # compute the error for the validation model
104     mse_val = sk.metrics.mean_squared_error(y_val, y_val_predict)
105     val_errors.append(mse_val)
106
107 plt.figure('test model')
108 plt.scatter(X, Y, s=2, label='data')
109 plt.scatter(X_train[:i], y_train[:i], label='data in training set')
110 plt.scatter(X_val, y_val, marker='s', label='validation data')
111 y_model = model.predict(np.expand_dims(X_val, axis=1))
112 plt.plot(X_val, y_model, 'r--', label='model')
113 plt.xlabel('x')
114 plt.ylabel('y')
115 plt.legend(loc=2)
116 plt.grid(True)
117 plt.savefig('test_plots/polynominal_model_'+str(i))
118 plt.close('test model')
119
120 plt.figure()
121 plt.grid(True)
122 plt.plot(train_errors, "--", label="train")
123 plt.plot(val_errors, ":", label="val")
124 plt.xlabel('number of data points in training set')
125 plt.ylabel('mean squared error')
126 plt.legend(framealpha=1)
127 plt.ylim(0,6)
128
129
130
131

```

Example 2.4

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Example 2.4
5  Ridge Regression
6  Machine Learning for Engineering Problem Solving
7  @author: Austin Downey
8  """
9
10 import IPython as IP
11 IP.get_ipython().run_line_magic('reset', '-sf')
12
13 import numpy as np
14 import matplotlib.pyplot as plt
15 import sklearn as sk
16 from sklearn import linear_model
17 from sklearn import pipeline
18
19 plt.close('all')
20
21 """ build the data sets
22 m = 20
23 X = 6 * np.random.rand(m, 1) - 3
24 Y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
25
26 X_model = np.linspace(-3,3,num=1000)
27 X_model = np.expand_dims(X_model,axis=1)
28
29
30 """ Perform Ridge Regression
31
32 # plot the data
33 plt.figure()
34 plt.grid(True)
35 plt.scatter(X,Y,color='gray')
36 plt.xlabel('x')
37 plt.ylabel('y')
38
39 # build and plot a linear model
40 model_linear = sk.linear_model.Ridge(alpha=100, solver="cholesky")
41 model_linear.fit(X, Y)
42 y_model_linear = model_linear.predict(X_model)
43 plt.plot(X_model,y_model_linear,'-',label='linear model')
44
45 # build and plot a polynomial model
46 model_poly = sk.pipeline.make_pipeline(sk.preprocessing.PolynomialFeatures(10),
47                                       sk.linear_model.Ridge(alpha=100, solver="cholesky"))
48 model_poly.fit(X, Y)
49 y_model_poly = model_poly.predict(X_model)
50 plt.plot(X_model,y_model_poly,'-',label='polynomial model')
51
52 plt.legend()
53
54
55
56
57
58

```

Example 2.5

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Example 2.5
5  Early Stopping
6  Machine Learning for Engineering Problem Solving
7  @author: Austin Downey
8  """
9
10 import IPython as IP
11 IP.get_ipython().run_line_magic('reset', '-sf')
12
13 import numpy as np
14 import matplotlib.pyplot as plt
15 import sklearn as sk
16
17
18 plt.close('all')
19
20 %% build the data sets
21
22 # use 6 to help give a smooth curve that makes the case for early stopping
23 np.random.seed(6)
24
25 m = 20
26 X = 6 * np.random.rand(m, 1) - 3
27 Y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
28
29 # plot the data
30 plt.figure()
31 plt.grid(True)
32 plt.scatter(X,Y,color='gray')
33 plt.xlabel('x')
34 plt.ylabel('y')
35
36 X_model = np.linspace(-3,3,num=1000)
37 X_model = np.expand_dims(X_model,axis=1)
38
39 X_train, X_val, y_train, y_val = sk.model_selection.train_test_split(X, Y, test_size=0.2)
40
41 %% perform early stopping
42
43
44 # prepare the data
45 poly_scaler = sk.pipeline.Pipeline([("poly_features", sk.preprocessing.PolynomialFeatures
46 (
47     degree=90, include_bias=False)), ("std_scaler", sk.preprocessing.StandardScaler())])
48 X_train_poly_scaled = poly_scaler.fit_transform(X_train)
49 X_val_poly_scaled = poly_scaler.fit_transform(X_val)
50
51 # set up the model, not that by setting max_iter=1 it will only train one epoch
52 model = sk.linear_model.SGDRegressor(max_iter=1, tol=0, learning_rate="constant"
53     ,eta0=0.0005,penalty=None,warm_start=True)
54
55 # Train the model in a loop to build the data set to investigate the benefit of early
56 # stopping
57 val_errors = []
58 train_errors = []
59 for epoch in range(1000):
60     model.fit(X_train_poly_scaled, y_train.ravel()) # continues where it left off
61     y_val_predict = model.predict(X_val_poly_scaled) # Predict the target values
62     y_train_predict = model.predict(X_train_poly_scaled) # Predict the target values
63     val_error = sk.metrics.mean_squared_error(y_val, y_val_predict) # Calculate error
64     train_error = sk.metrics.mean_squared_error(y_train.ravel(), y_train_predict) #
65     Calculate error
66     val_errors.append(val_error)

```



```
65     train_errors.append(train_error)
66
67     # plot the early learning curves, you may have to plot this a few times to get
68     # a set of curves that shows strong results
69     plt.figure()
70     plt.grid(True)
71     plt.plot(val_errors, label='validation data')
72     plt.plot(train_errors, '--', label='training data')
73     plt.ylabel('RMSE')
74     plt.xlabel('epoch')
75     plt.legend()
76
77
78
79
80
81
```