Data-intensive space engineering

Lecture 2

Carlos Sanmiguel Vila

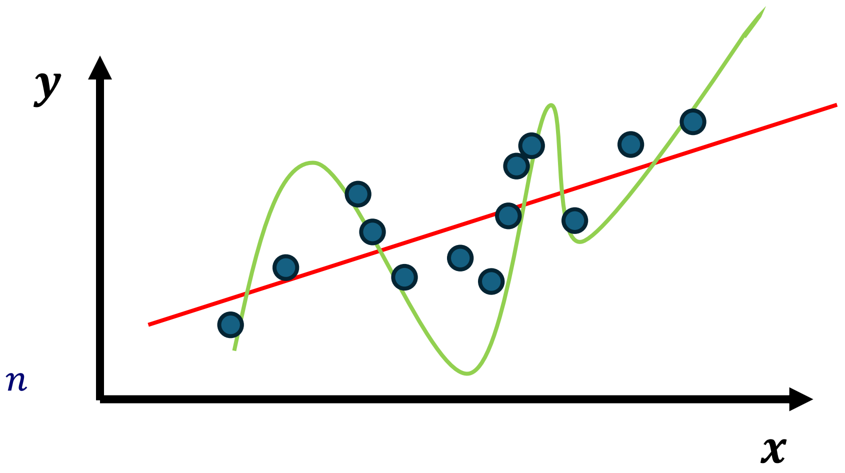
Supervised Learning

Prediction

Given: a training dataset that contains n samples

$$\left(x^{(1)}, y^{(1)}\right), \dots \left(x^{(n)}, y^{(n)}\right)$$

$$x^{(i)} \in X, y^{(i)} \in Y, \text{ being } X \in \mathbb{R}^{n_x\,x\,n} \;\; Y \in \mathbb{R}^{n_y\,x\,n}$$



We want to find a good $h: X \to Y$ (hypothesis)

We care about new values that are not in our training set

Given a set of unseen points x, which is their y?

If y is discrete $\rightarrow$ Classification
If y is continuous $\rightarrow$ Regression

light intensity $\rightarrow$ Is this an exoplanet?
Planet radius $\rightarrow$ Planet Mass

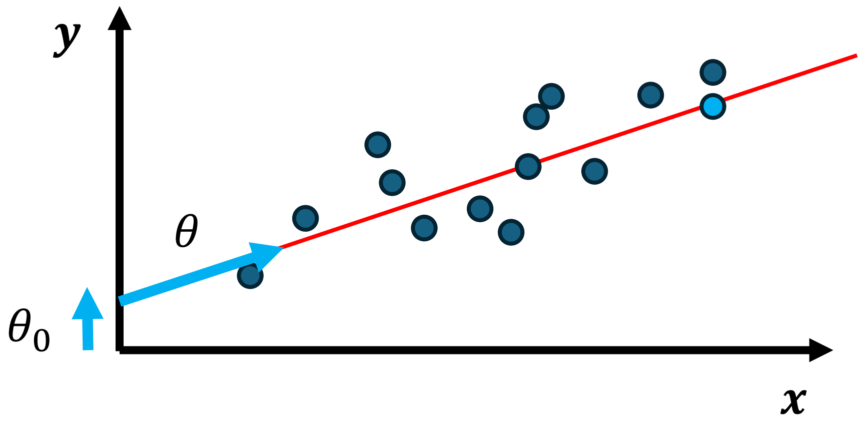
Regression

In regression, we want to quantify the relationship between continuous variables and make predictions.

$h: X \to Y$

How do we represent h?

$$h(x) = \theta_0 + \theta_1 x$$



	$x_1^{(1)}$	$x_2^{(1)}$	....	y
	Planet Radius	**Orbital Period**	....	**Planet Mass**
$x^{(1)}$	1.25	3.101278		0.62
$x^{(2)}$	1.24	3.246740		1.02
	features			target

$h(x) = \theta_0 x_0 + \theta_1 x_1 + \cdots + \theta_d x_d = \sum_{j=0}^{d} \theta_j x_j$ with $x_0 = 1$ $\longrightarrow$ $\hat{y} = h(x) = \theta x$

Regression

$\hat{y} = h(x) = \theta x$

$y = h(x) + \epsilon$
$= \theta x + \epsilon$

- h is the hypothesis function, using the model parameters θ
- θ is the model's parameter vector, containing the bias term θ_0 and the feature weights θ_1 to θ_n
- x is the instance's feature vector, with x_0 always equal to 1.
- $\hat{y}$ is the predicted value.
- ϵ is an error term that captures either unmodeled effects (such as if there are some features very pertinent to predicting the target, but that we'd left out of the regression), or random noise

We have a regression… but how do we train it?

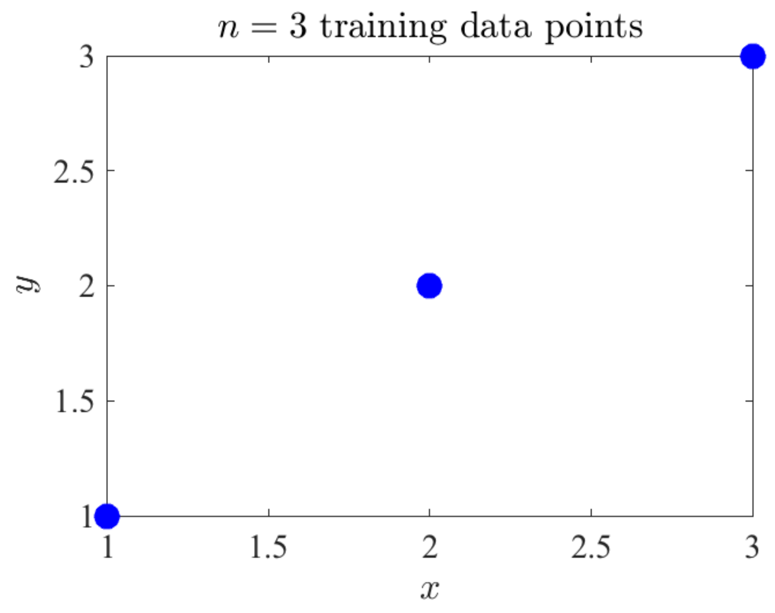
Training a model means setting its parameters so that the model best fits the data. For this purpose, we first need a measure of how well (or poorly) the model fits the training data

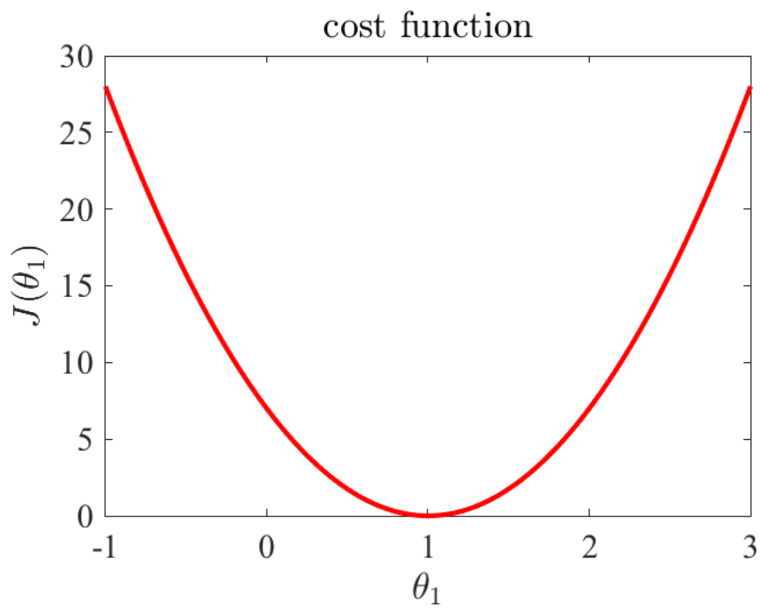
Cost function (Least squares): $J(\theta) = \frac{1}{2}\sum_{i=0}^{n}\left(h\left(x^{(i)}\right) - y^{(i)}\right)^2$ $\min_{\theta} \mathrm{J}(\theta)$

How to choose model parameters?

Cost function (Least squares): $J(\theta) = \frac{1}{2}\sum_{i=0}^{n}\left(h\left(x^{(i)}\right) - y^{(i)}\right)^2$

Goal: Find the value of θ that leads to the minimum value of $J(\theta)$





Mathematical optimization is a central part of machine learning

The normal equation

Given: a training dataset $\left(x^{(i)}, y^{(i)}\right)$ that contains n sample, use dot products

$$\begin{bmatrix} h(x^{(1)}) \\ \vdots \\ h(x^{(n)}) \end{bmatrix} = \begin{bmatrix} \langle x^{(1)}, \theta \rangle \\ \vdots \\ \langle x^{(n)}, \theta \rangle \end{bmatrix} = \begin{bmatrix} x^{(1)} \\ \vdots \\ x^{(n)} \end{bmatrix} \theta = X\theta$$

Thus, we can rewrite the previous cost function as

$$J(\theta) = \frac{1}{2} \|X\theta - y\|^2$$

The partial derivatives are all zero at the minimum value

$$\frac{1}{2}\left(2X^T \cdot \mathrm{X}\theta - 2X^T y\right) = 0$$

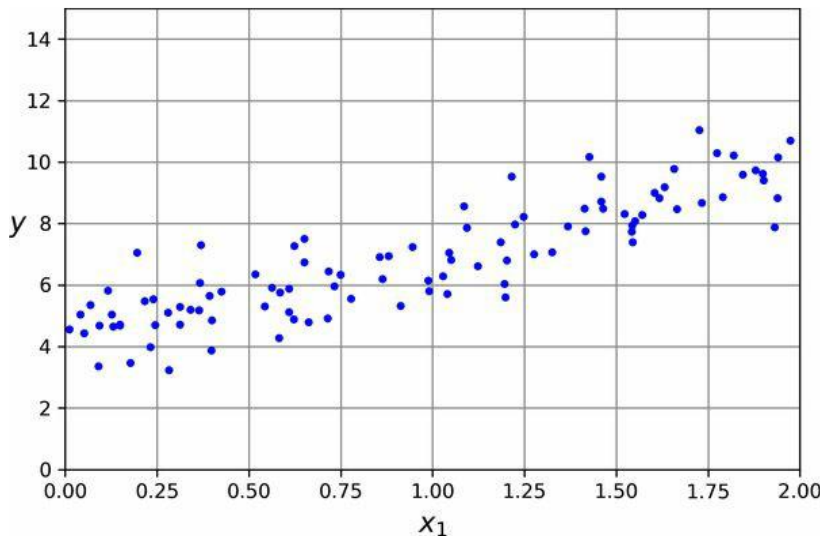
$$\boxed{\hat{\theta} = (X^T \cdot \mathrm{X})^{-1} \cdot X^T \cdot \mathrm{y}}$$

This expression is known as the normal equation solution of the least squares problem

https://en.wikipedia.org/wiki/Least_squares

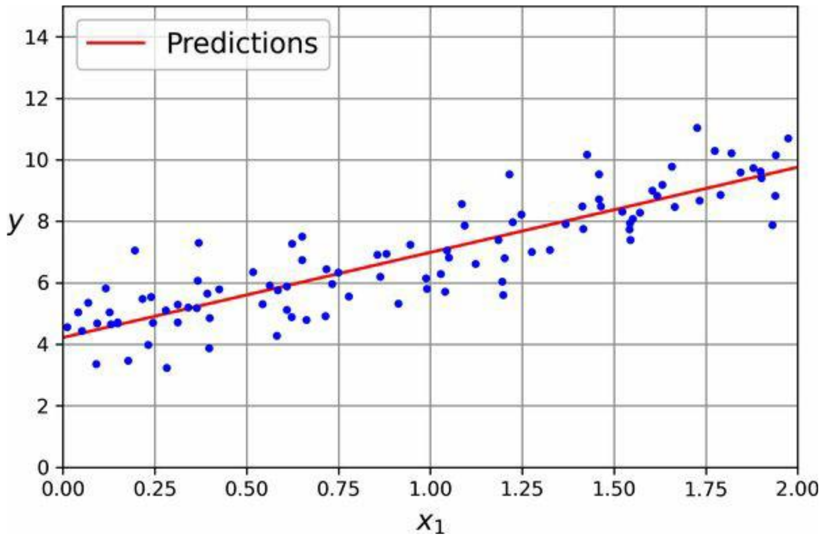
The normal equation

```
import numpy as np
np.random.seed(42) # to make this code example reproducible
m = 100 # number of instances
X = 2 * np.random.rand(m, 1) # column vector
y = 4 + 3 * X + np.random.randn(m, 1) # column vector


from sklearn.preprocessing import add_dummy_feature
X_b = add_dummy_feature(X) # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T @ X_b) @ X_b.T @ y

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



The normal equation

```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = add_dummy_feature(X_new) # add x0 = 1 to each instance
>>> y_predict = X_new_b @ theta_best
>>> y_predict
array([[4.21509616],
[9.75532293]])

import matplotlib.pyplot as plt
plt.plot(X_new, y_predict, "r-", label="Predictions")
plt.plot(X, y, "b.")
[...] # beautify the figure: add labels, axis, grid, and legend
plt.show()
```



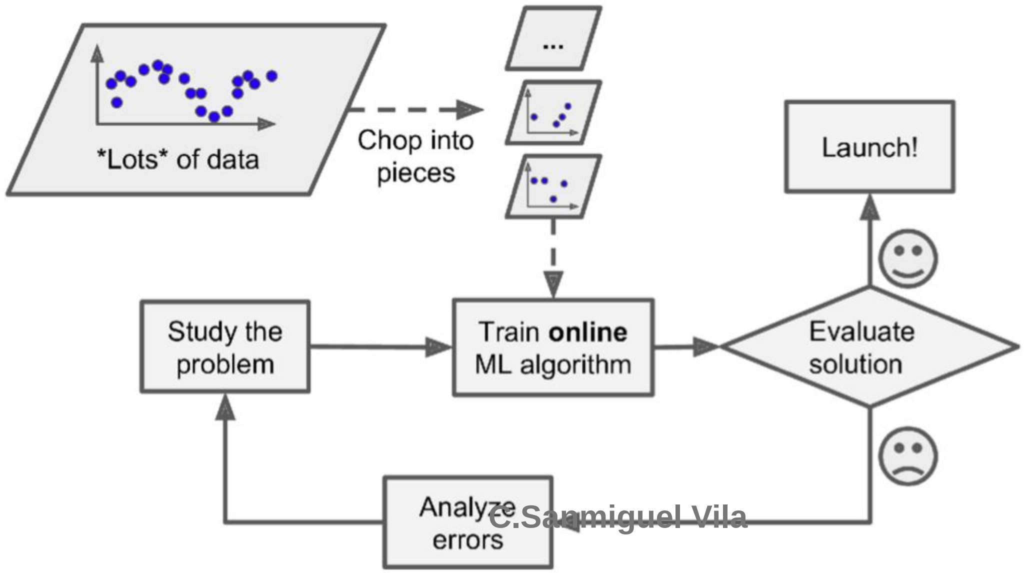
The normal equation

Directly solving it is expensive:

> $O(nd^2)$ for the multiplication *(n=samples, d=features)* and
> $O(d^3)$ for the matrix inverse

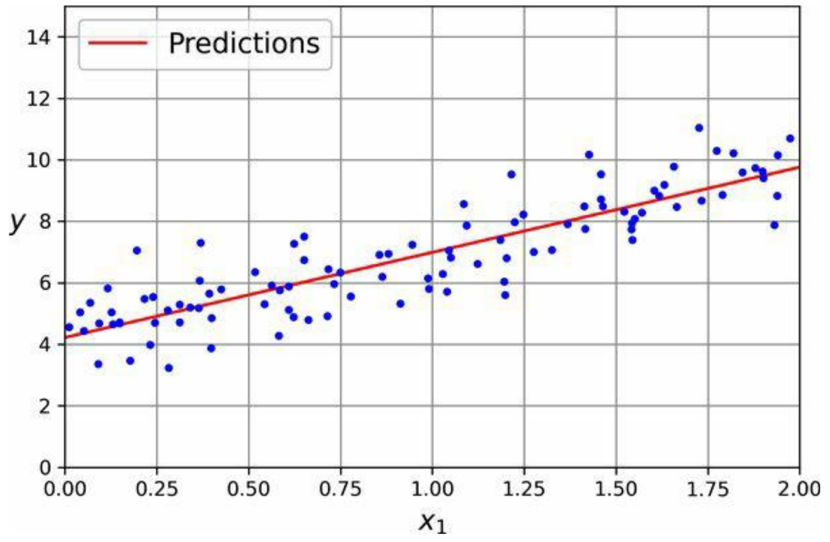
but there are fast methods using SVD or online algorithms.

There are methods (Gradient Descent) that are better suited for cases where there are a large number of features or too many training instances to fit in computer memory.



Scikit-learn

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

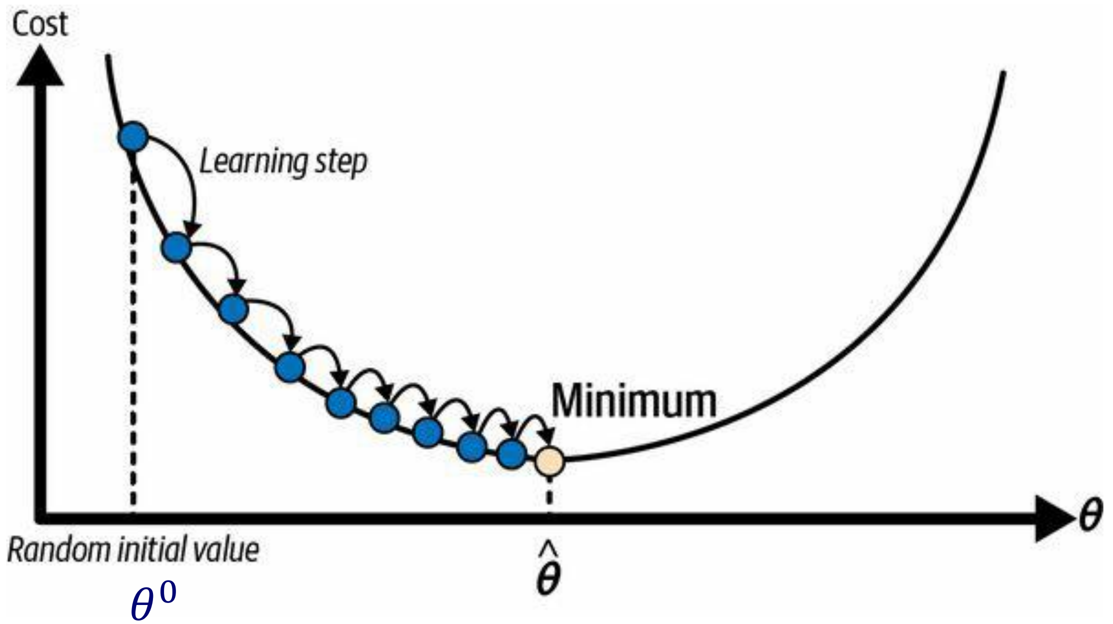


Notice that Scikit-Learn separates the bias term (intercept_) from the feature weights (coef_).

Gradient descent

Gradient descent is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of gradient descent is to tweak parameters iteratively to minimize a cost function

$$\hat{y} = h(x) = \theta x \qquad J(\theta) = \frac{1}{2}\sum_{i=0}^{n}\left(h\left(x^{(i)}\right) - y^{(i)}\right)^2$$



- Start by filling θ with random values
- Take steps to decrease the cost function until convergence

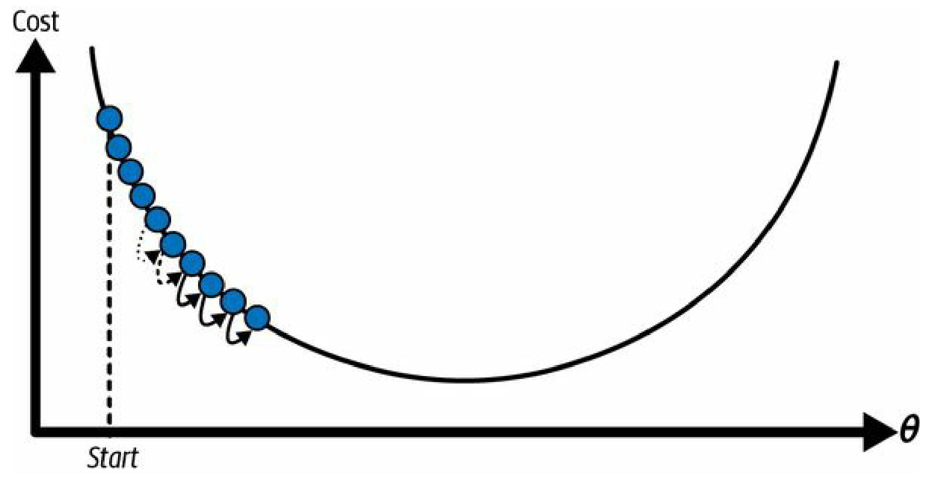
$$\theta^{t+1} = \theta^t - \eta \nabla J(\theta^t)$$

θ^0 is a random initial value

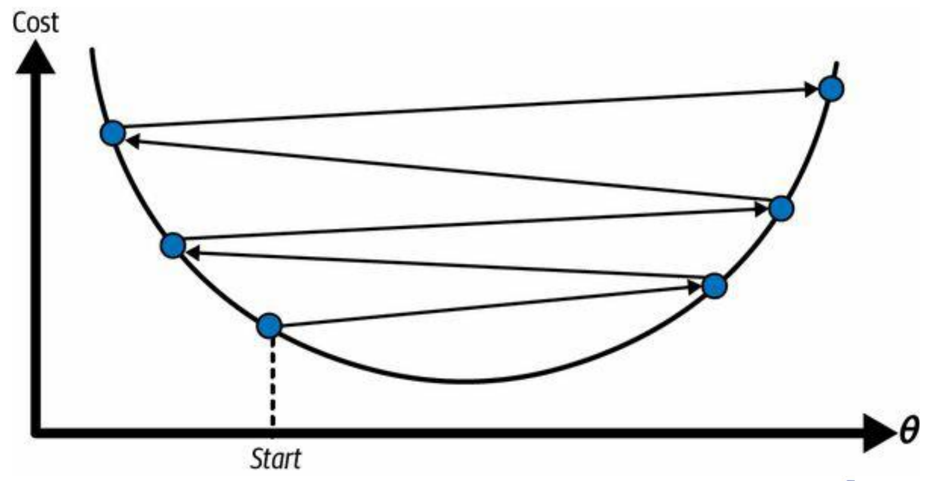
η the learning rate

Gradient descent

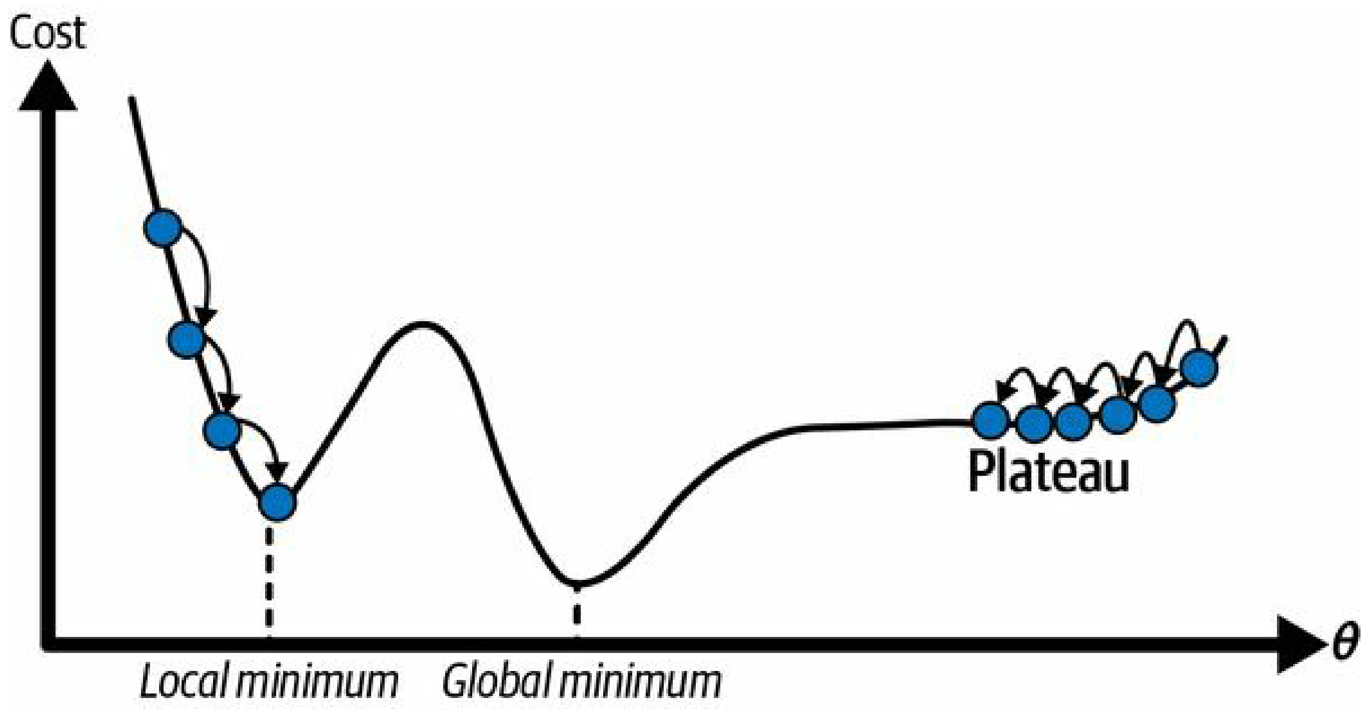
- If the learning rate is too low: many iterations and a long computational time



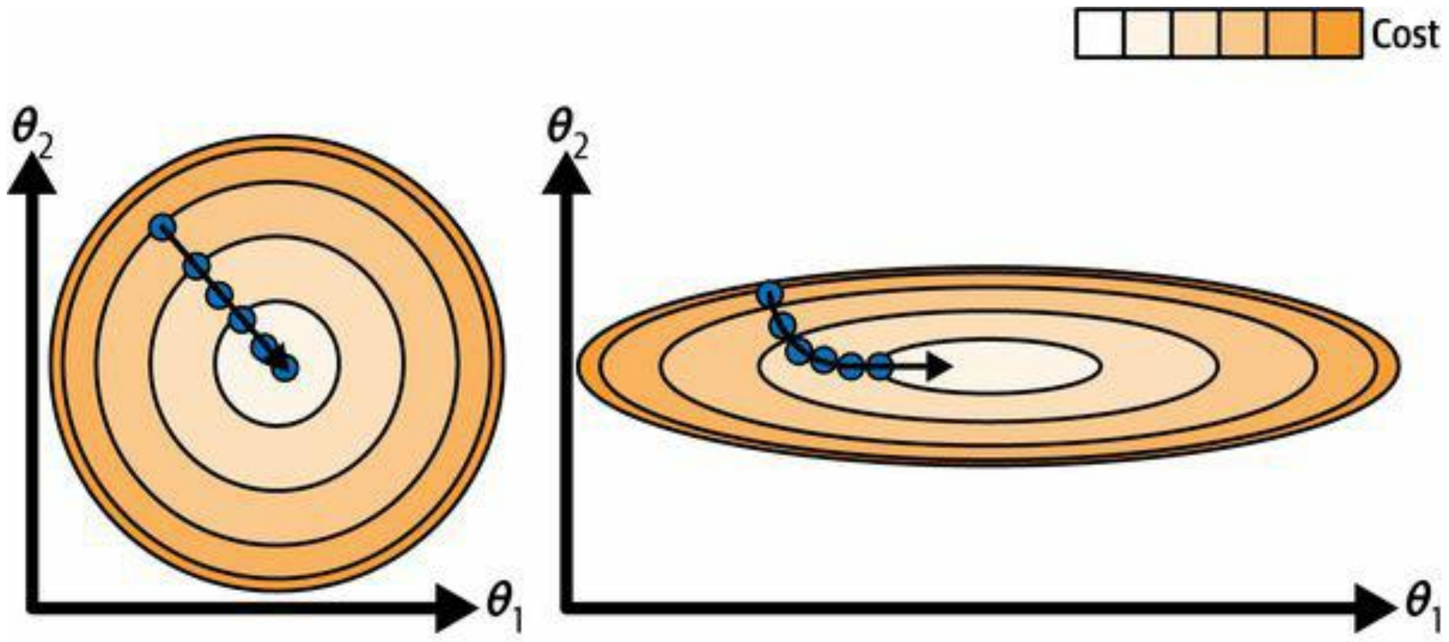
- If the learning rate is too large: fail to find a good solution



Challenges of using gradient descent



Relevance of feature scaling



Gradient descent with (left) and without (right) feature scaling

When using gradient descent, you should ensure that all features have a similar scale (e.g., using Scikit-Learn's StandardScaler class), or else it will take much longer to converge.

Gradient Descent Implementation

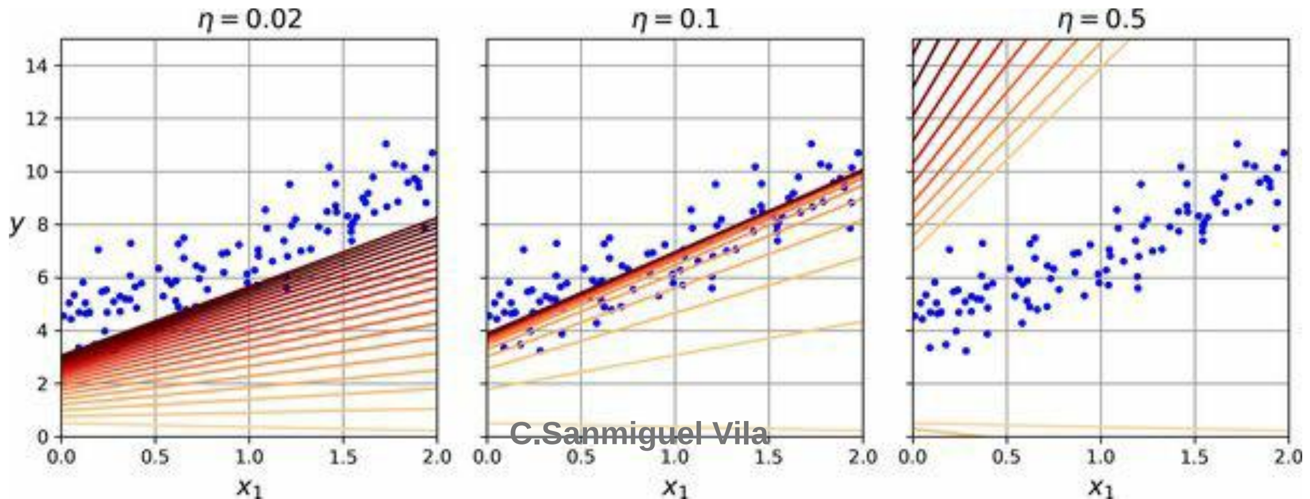
```
eta = 0.1 # learning rate
n_epochs = 1000
m = len(X_b) # number of instances
np.random.seed(42)
theta = np.random.randn(2, 1) # randomly initialized model
parameters
for epoch in range(n_epochs):
        gradients = 2 / m * X_b.T @ (X_b @ theta - y)
        theta = theta - eta * gradients

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

$$\theta^{t+1} = \theta^{t} - \eta \nabla J(\theta^{t})$$

Gradient Descent Implementation

$$\theta^{t+1} = \theta^{t} - \eta \nabla J(\theta^{t})$$

- This formula involves calculations over the full training set X, at each gradient descent step. This is why the algorithm is called batch gradient descent: it uses the whole batch of training data at every step.
- As a result, it is terribly slow on very large training sets.
- However, gradient descent scales well with the number of features; training a linear regression model with hundreds of thousands of features is much faster using gradient descent than using the Normal equation or SVD decomposition.



Batch vs Stochastic Minibatch

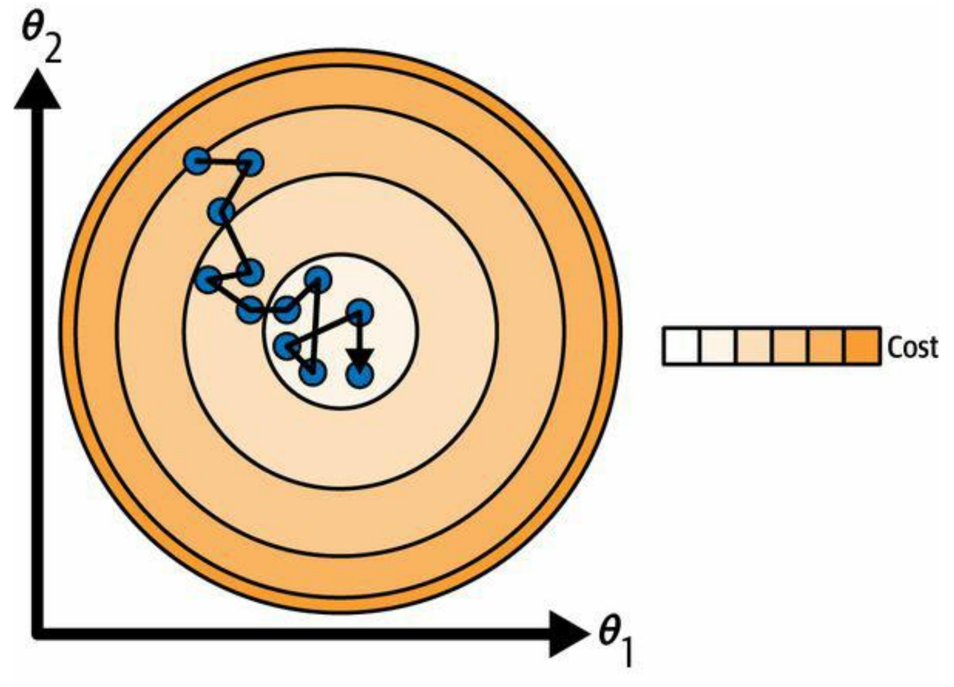
Batch gradient descent: $\theta = \theta^t - \eta \sum_{i=0}^{n} \left(\left(h_\theta\left(x^i\right) - \mathrm{y}^\mathrm{i}\right)x^i\right)$

Stochastic gradient descent picks a random instance in the training set at every step and computes the gradients based only on that single instance.

$$\theta = \theta^t - \eta\left(\left(h_\theta\left(x^i\right) - \mathrm{y}^\mathrm{i}\right)x^i\right.$$

- It is much faster than Batch SD because it has very little data to manipulate at every iteration.

- On the other hand, due to its stochastic (i.e., random) nature, this algorithm is much less regular than batch SD:, the cost function will bounce up and down, decreasing only on average.



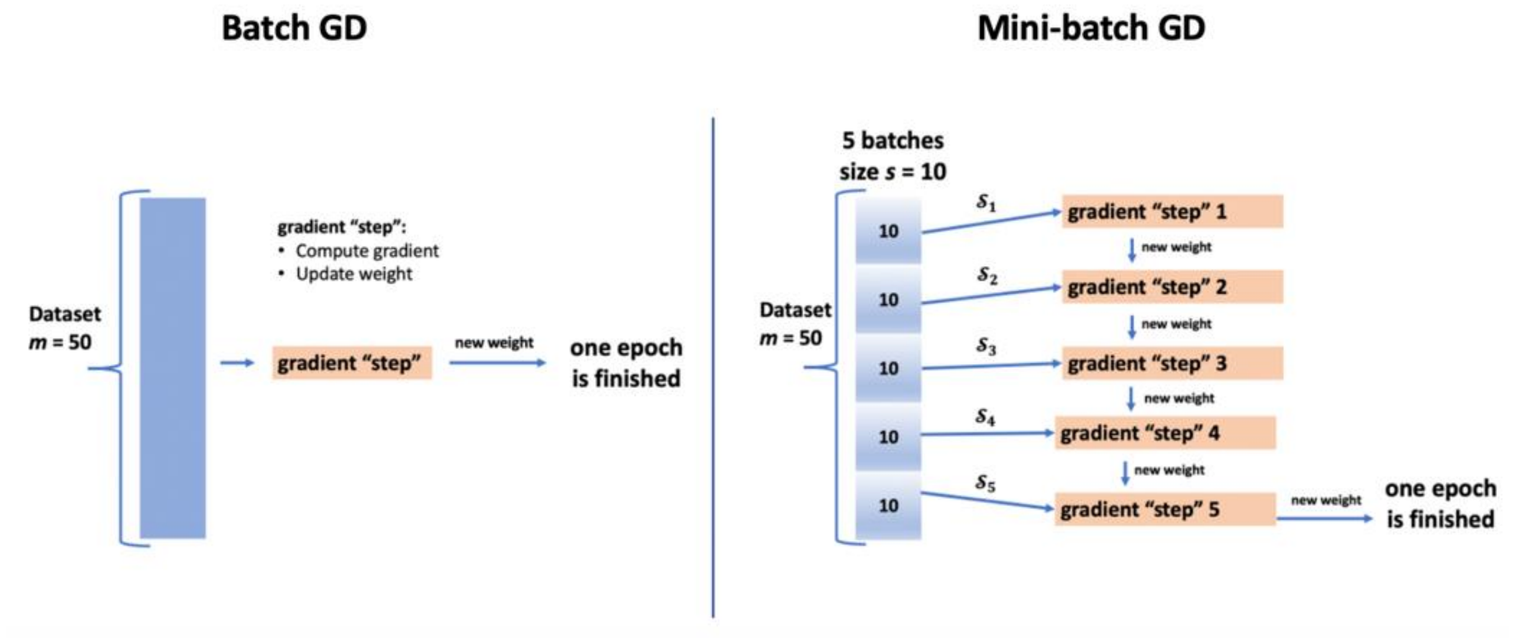
Batch vs Minibatch GD

Batch gradient descent: $\theta = \theta^t - \eta \sum_{i=0}^{n} ((h_\theta(x^i) - \mathrm{y}^\mathrm{i})x^i)$

<u>Minibatch</u>

We take a batch of data (at random) B where $B \ll N$

$$\theta = \theta^t - \eta \sum_{i \in B} ((h_\theta(x^i) - \mathrm{y}^\mathrm{i})x^i$$



Batch vs Minibatch GD

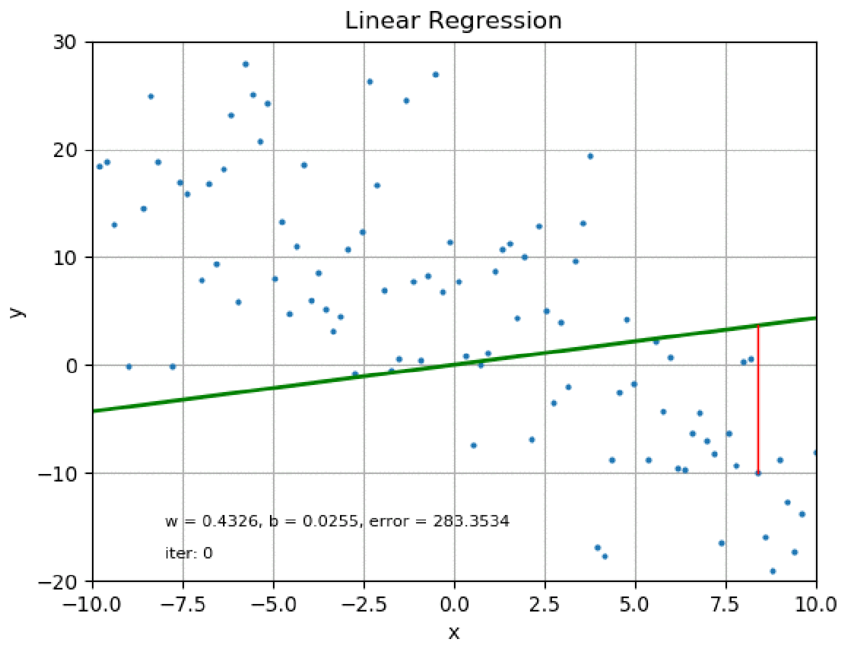
•**Efficient Memory Usage:** Mini-batches allow for better memory utilization, enabling training on large datasets even with limited memory by processing data in smaller chunks.

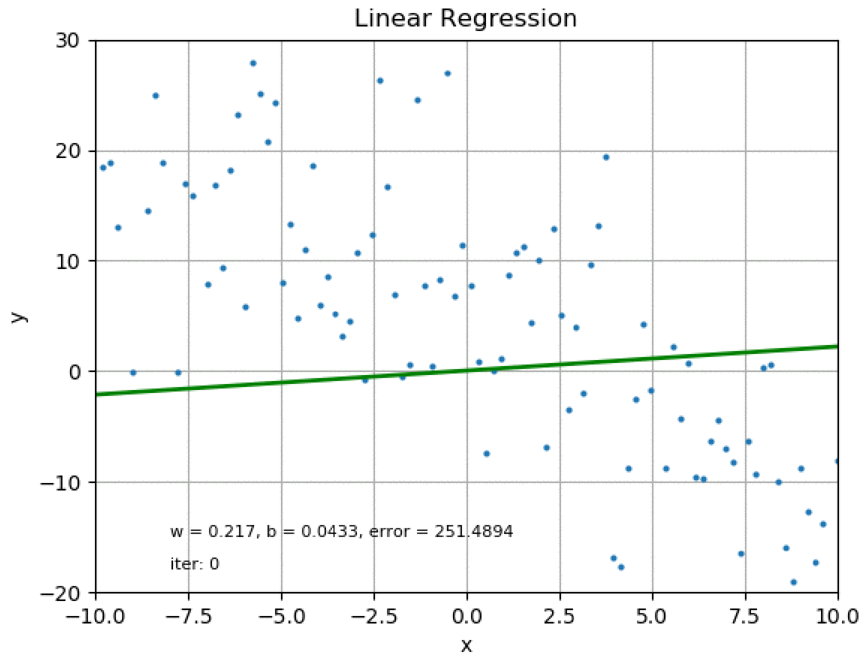
•**Faster Convergence:** Mini-batch gradient descent often converges faster due to more frequent parameter updates, leading to quicker adjustments towards the optimal solution.

•**Improved Generalization:** The noisy gradient estimates from mini-batches help the model escape local minima and saddle points, resulting in better generalization and robustness.

•**Stochasticity:** Introducing stochasticity through mini-batches helps the model explore the solution space more effectively, reducing the risk of getting stuck in local minima.

Optimization through gradient descent



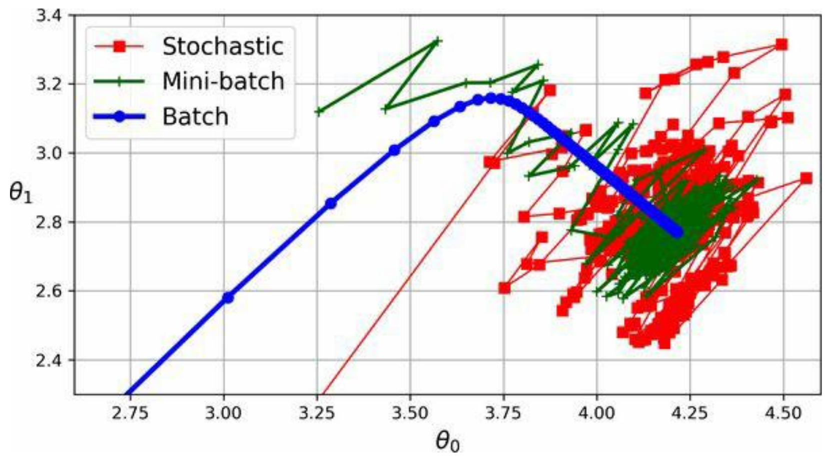


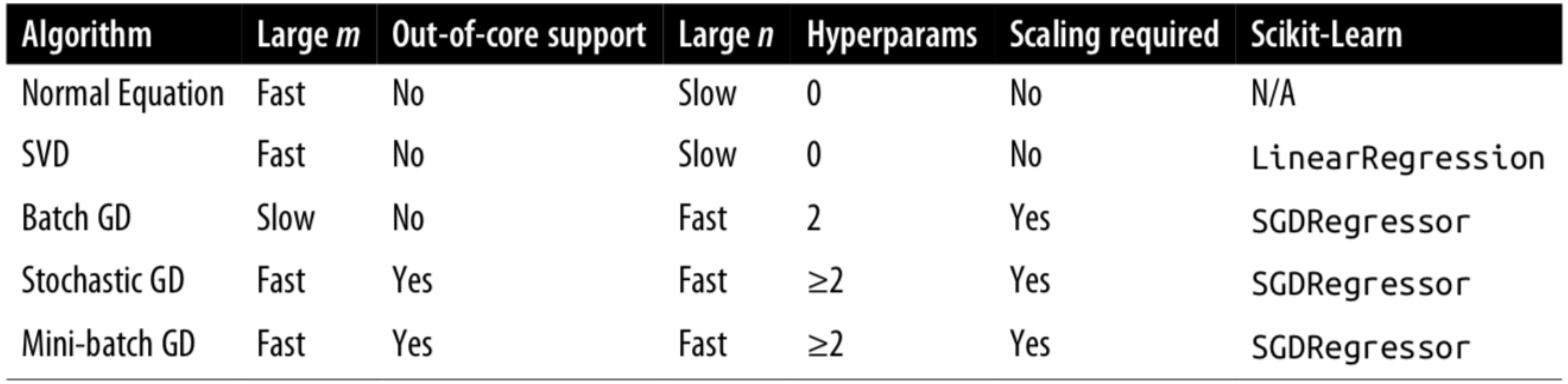
Stochastic gradient descent for 2D linear regression

Batch gradient descent

https://medium.com/swlh/from-animation-to-intuition-linear-regression-and-logistic-regression-f641a31e1caf

Comparison of linear regression implementations





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

Textbook notation: m (number of samples) and n (number of features)

Polynomial regression

A polynomial of degree 1 gives us the linear regression model

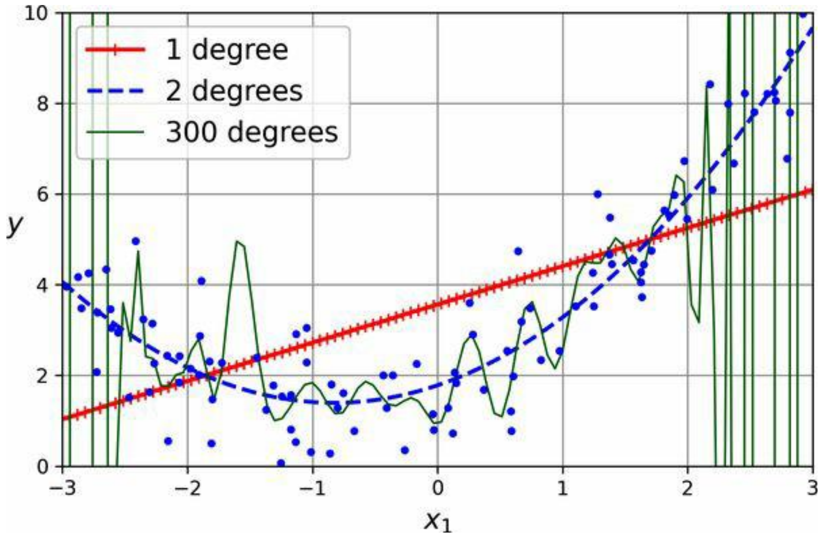
$$h(x) = \theta_0 + \theta_1 x$$

We can add more powers of as new features

$$h(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \cdots$$

- 300-degree polynomial model wiggles around to get as close as possible to the training instances

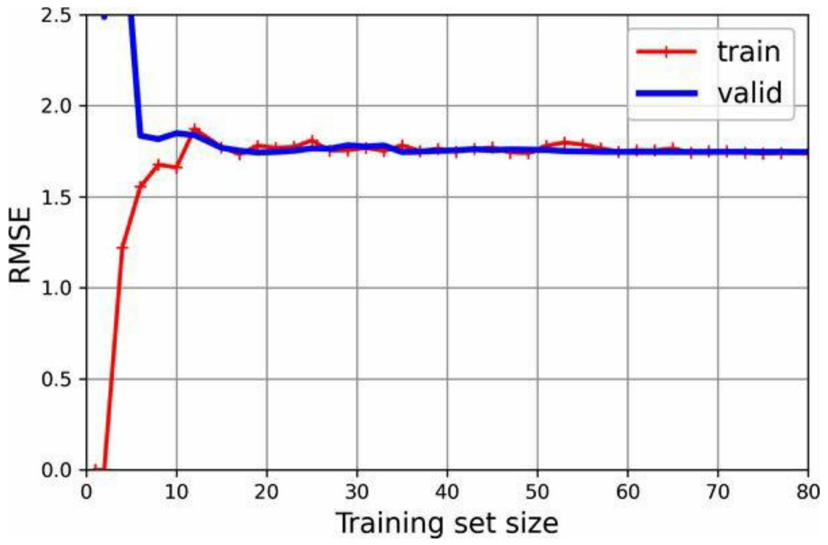
- **Overfitting**: performs well on training data but generalizes poorly



Learning curve

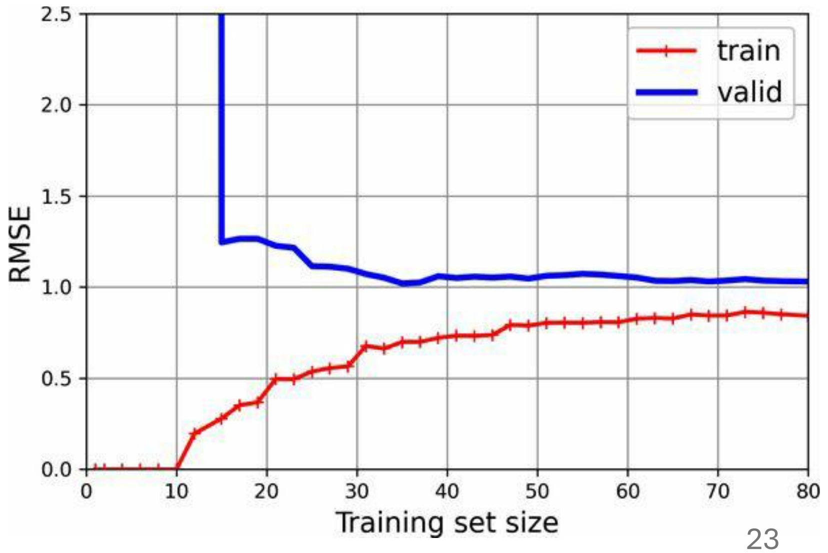
Linear regression

Underfitting: As new instances are added to the training set, it becomes impossible for the model to fit the training data perfectly, both because the data is noisy and because it is not linear at all. So the error on the training data goes up until it reaches a plateau.



10th-degree polynomial model

- The error on the training data is much lower.
- The gap between the curves means that the model performs better on the training data than on the validation data, which is the hallmark of an overfitting model.
- Using a larger training set, however, the two curves would continue to get closer.



Bias/variance tradeoff

- Generalization error can be expressed as the sum of three different errors:

1. **Bias**: This is due to wrong assumptions, such as assuming the data is linear when it is actually quadratic. A high-bias model is most likely to underfit the training data

2. **Variance**: This is due to the model's excessive sensitivity to small variations in the training data (i.e., degrees of freedom). These models are likely to have high variance and thus overfit the training data.

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

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

Ridge (L2) Regression

We can adapt the linear regression to incorporates regularization to prevent overfitting since it tries to prevent a model from becoming too complex.

The cost function is modified to include a penalty term (also called the regularization term) based on the sum of squared weights.

Cost function: $J(\theta) = \frac{1}{2}\sum_{i=0}^{n}\left(\theta x^{(i)} - y^{(i)}\right)^2 + \alpha \sum_{j=0}^{j=d} \theta_j^2$ *n=samples, d=features*

Where α is the regularization parameter (also called the penalty term). $\alpha \sum_{j=0}^{j=d} \theta_j^2$ is the L2 regularization term, penalizing large θ_j.

α shrinks the coefficients, reducing their magnitude, which controls overfitting.
$\alpha = 0$ No regularization (equivalent to ordinary least squares regression).
$\alpha = \infty$ Strong regularization, potentially leading to coefficients being close to zero (though not exactly zero).

Ridge (L2) Regression

Advantages:

- Prevents overfitting by introducing bias to the model.
- Handles multicollinearity well by stabilizing estimates.

Disadvantages:

- Shrinks all coefficients uniformly, even the relevant ones.
- Cannot perform feature selection (i.e., it doesn't zero out coefficients).

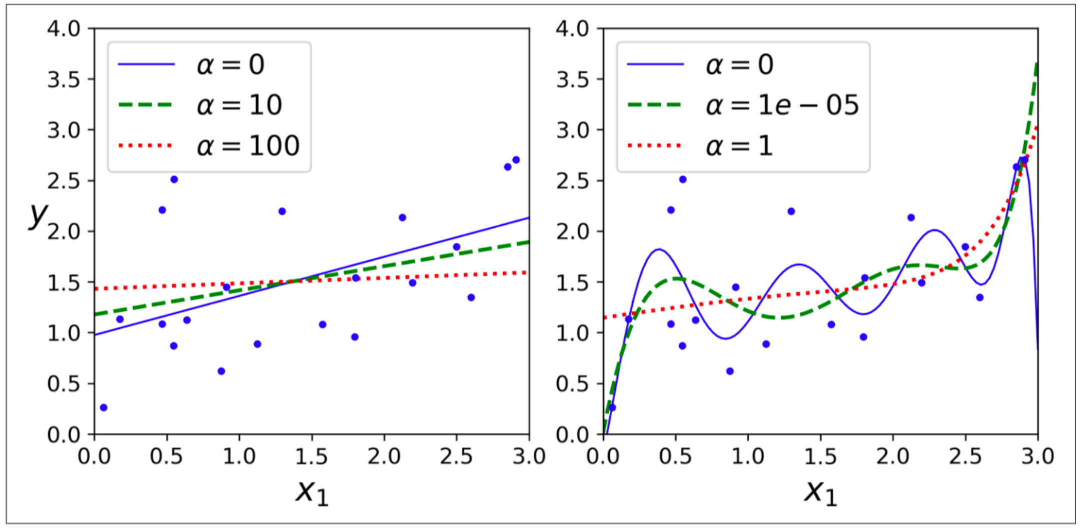


Figure 4-17. A linear model (left) and a polynomial model (right), both with various levels of Ridge regularization

Multicollinearity: some features in your model are **highly correlated**. For least squares (LS) regression, it is complex to estimate the coefficients reliably, as small changes in the data can lead to large fluctuations in the estimates.

Example: Assuming two highly correlated features, x_1 and x_2. In LS regression, small changes in the data might lead to very different coefficients for θ_1 and θ_2, making the model unstable. Ridge shrinks both θ_1 and θ_2 , ensuring that the model does not rely too heavily on either, leading to more reliable predictions.
This ability to **stabilize estimates in the face of multicollinearity** is why Ridge Regression is often preferred when working with highly correlated data.

Encouraging “Sparse” Models: Lasso (L1)

Lasso Regression is another type of linear regression that includes regularization, but unlike Ridge, it uses an L1 penalty. This type of regression is useful for feature selection because it can shrink some coefficients to exactly zero, effectively removing irrelevant features.

Cost function: $J(\theta) = \frac{1}{2}\sum_{i=0}^{n}\left(x^{(i)}\theta - y^{(i)}\right)^2 + \alpha\sum_{j=0}^{j=d}\left|\theta_j\right|$ *n=samples, d=features*

Where α is the regularization parameter (also called the penalty term). $\alpha\sum_{j=0}^{j=d}\left|\theta_j\right|$ is the L1 regularization term, penalizing large absolute values of θ_j.

$\alpha = 0$ No regularization (equivalent to ordinary least squares regression).
$\alpha = \infty$: Strong regularization, where more coefficients shrink to zero.

Encouraging “Sparse” Models: Lasso (L1)

Advantages:

- **Feature selection**: Lasso can set some coefficients to zero, effectively excluding them from the model.
- Helps in dealing with **overfitting** when there are many irrelevant features.

Disadvantages:

- Can struggle with **highly correlated features**, arbitrarily selecting one and shrinking the rest.
- It might underperform when all features are important but only need some shrinkage.

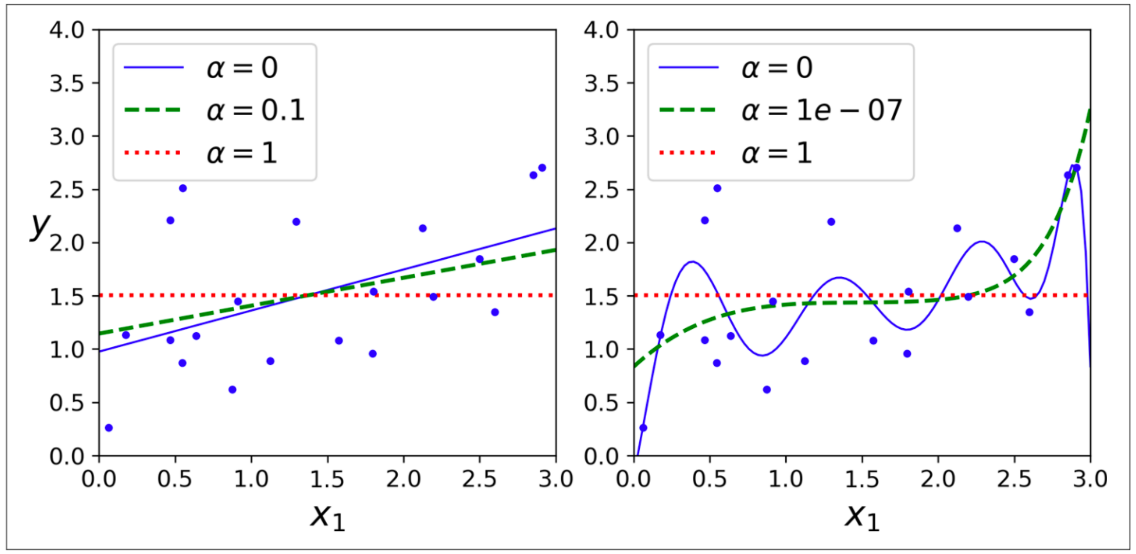


Figure 4-18. A linear model (left) and a polynomial model (right), both using various levels of Lasso regularization

Lasso vs Ridge

Advantages:

- **Feature selection**: Lasso can set some coefficients to zero, effectively excluding them from the model.
- Helps in dealing with **overfitting** when there are many irrelevant features.

Disadvantages:

- Can struggle with **highly correlated features**, arbitrarily selecting one and shrinking the rest.
- It might underperform when all features are important but only need some shrinkage.

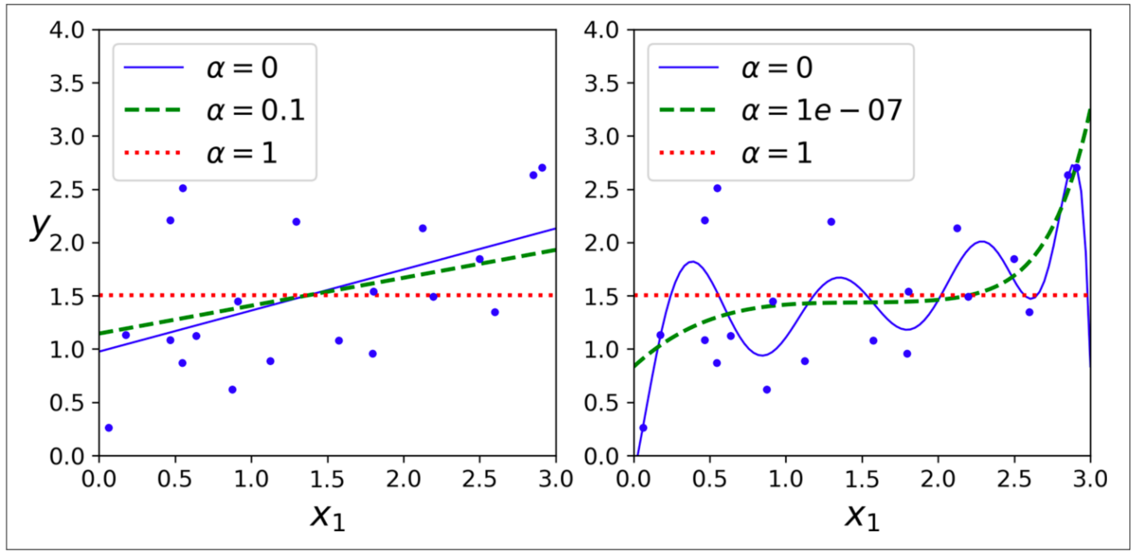


Figure 4-18. A linear model (left) and a polynomial model (right), both using various levels of Lasso regularization

Lasso vs Ridge

- **Ridge**: Useful when you have many small/medium-sized coefficients and when multicollinearity is an issue.

- **Lasso**: Useful when you want feature selection and believe many features are irrelevant.

- Can we put the two together?

Elastic Net Regression:L1 + L2

Elastic Net combines the **L1 penalty** from Lasso and the **L2 penalty** from Ridge to benefit from both:

Cost function: $J(\theta) = \frac{1}{2}\sum_{i=0}^{n}\left(\theta x^{(i)} - y^{(i)}\right)^2 + \alpha_1 \sum_{j=0}^{j=d} \theta_j^2 + \alpha_2 \sum_{j=0}^{j=d} |\theta_j|$

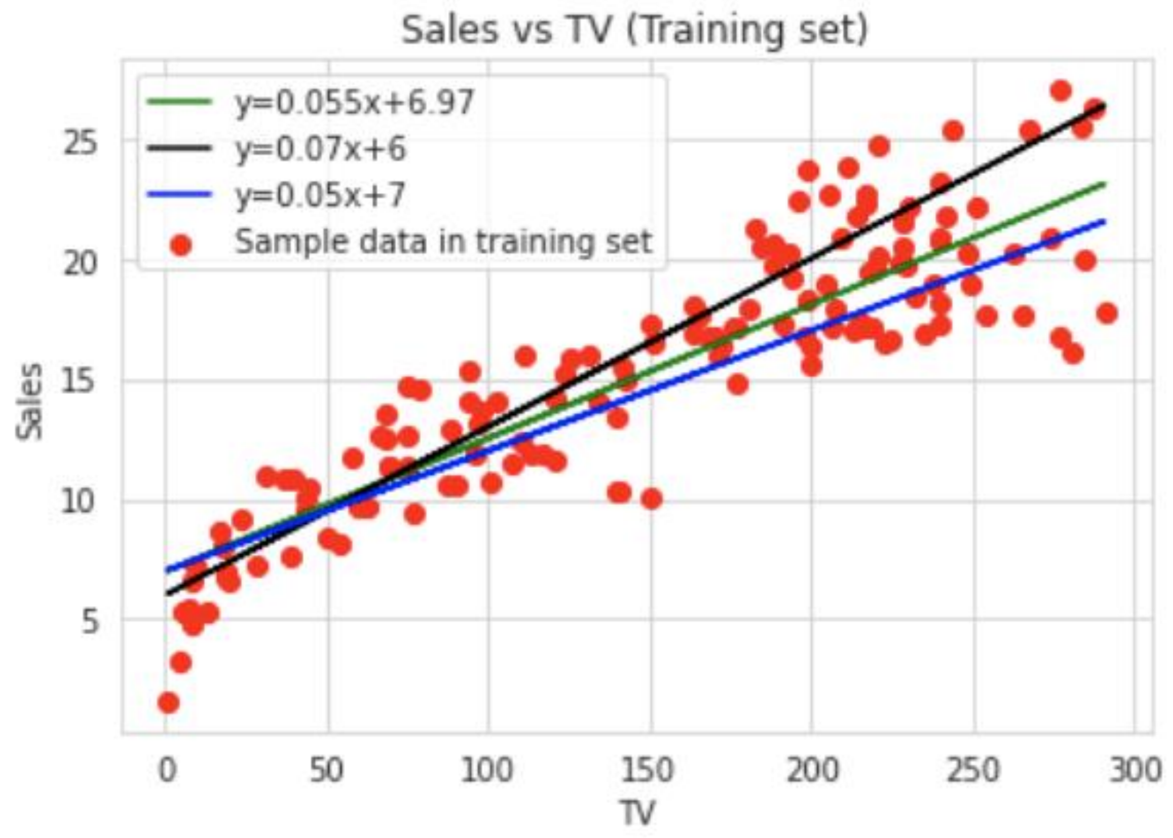
Why Elastic Net? Solves the limitations of Lasso in high-dimensional datasets where multicollinearity is present.

Balances feature selection (L1) with coefficient shrinkage (L2).

Should we always use Elastic Net?

1. **Increased Complexity**:
    1. Elastic Net introduces **two hyperparameters** that need to be tuned. This adds complexity to the model compared to using only Ridge or Lasso, where is only one regularization parameter.
2. **Risk of Over-regularization**:
    1. If the **penalty terms are not chosen carefully**, there’s a risk of over-regularization, where useful features might be shrunk too much, leading to underfitting (where the model is too simple and fails to capture important relationships).
    2. It requires a balance between L1 and L2 penalties, which can be tricky to optimize.
3. **Interpretability**:
    1. Elastic Net might keep more features in the model than Lasso, making it harder to interpret when feature selection is important..
4. **Lasso Limitations with Multicollinearity**:
    1. It might still **struggle with strongly correlated features**. Ridge helps distribute weight across correlated variables, but Elastic Net’s L1 component can still arbitrarily pick one variable over another in highly collinear scenarios.
5. **Requires Sufficient Data**:
    1. Elastic Net is most effective when there are more features than observations or when some features are correlated. However, in smaller datasets, the dual regularization may not provide significant advantages, and tuning can be difficult due to limited data.

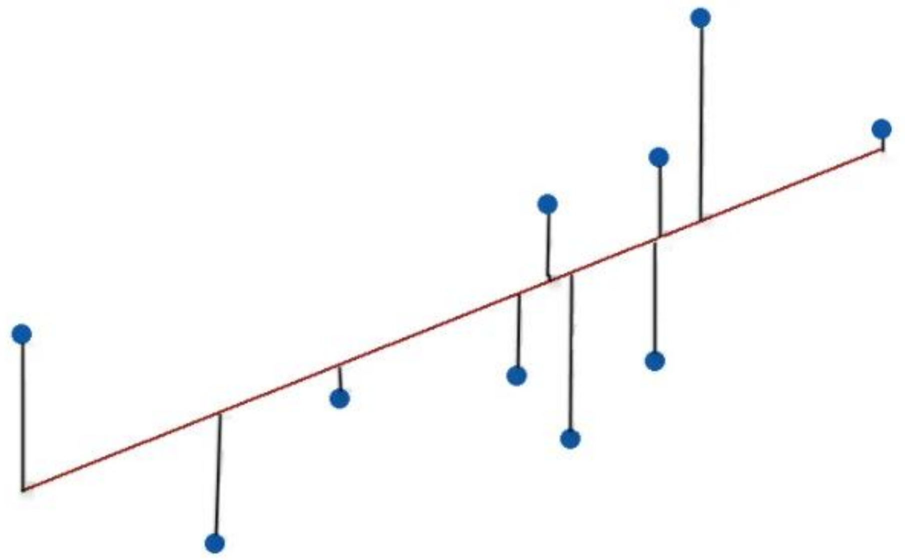
Regression Metrics



Which model is better?

Regression Metrics

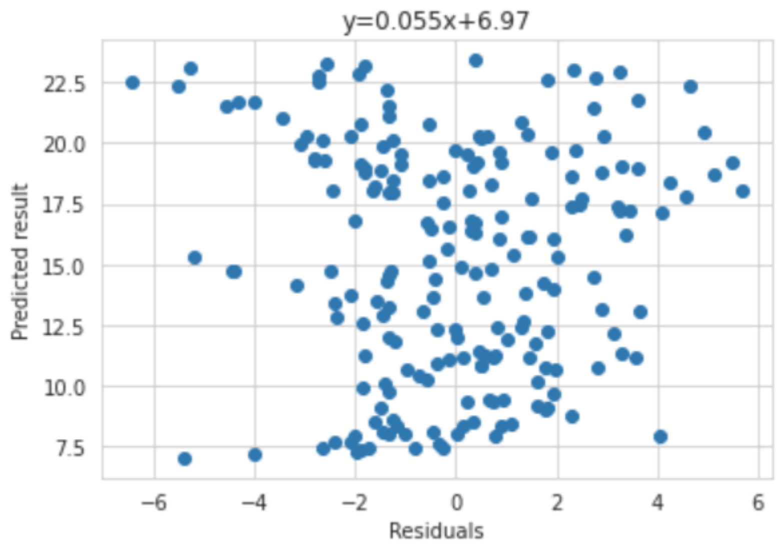
- Residuals refer to the difference between actual values and their predicted values

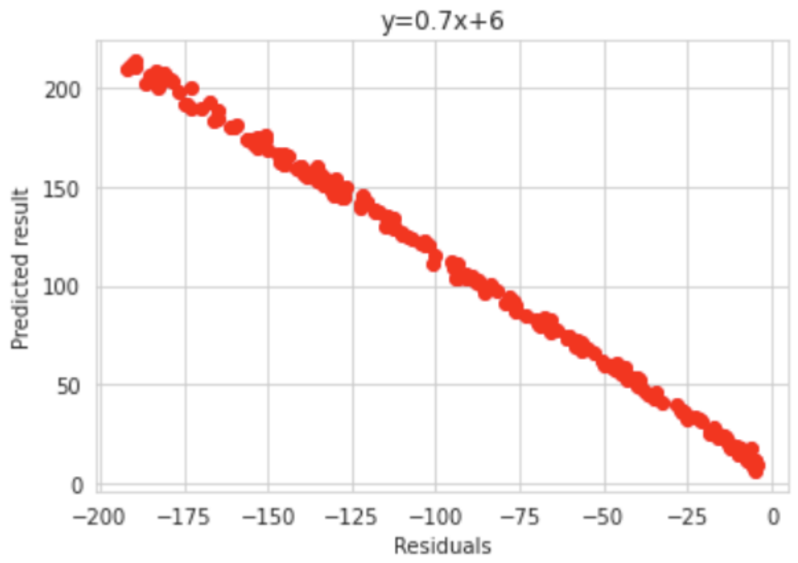


- We want a regression model where residuals are small and unbiased (i.e., random)

Plot the residuals

Which one is better?





You can’t trust a model that is biased

R-squared (R^2) value

- R^2 is always between 0 and 1
  - 0: means a model does not explain any variation in the target variable around its mean (not desirable)
  - 1: means a model that explains all the variation in the target variable around its mean (desirable)

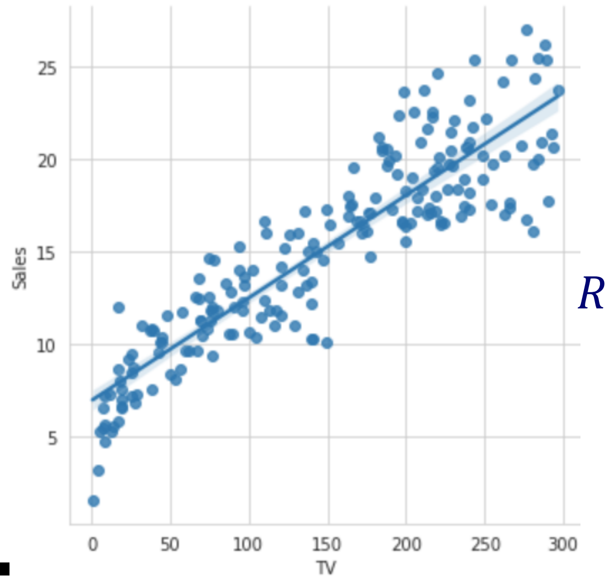
$$R^2 = 1 - \frac{RSS}{TSS}$$

$RSS = \sum_i (y^{(i)} - \hat{y}^{(i)})^2$: sum of squares of residuals
$TSS = \sum_i (y^{(i)} - \bar{y})^2$: total sum of squares (i.e., variance)

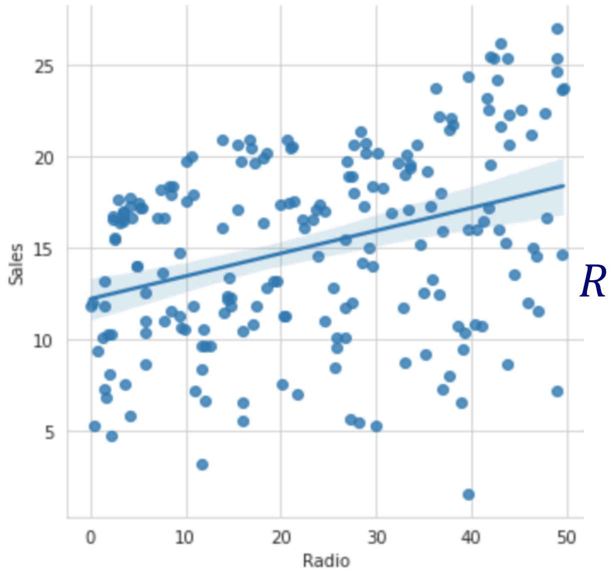
R-squared (R^2) value

- R^2 evaluates the scatter of the data points around the regression line. It is also called as coefficient of determination
- The higher R^2 is, the smaller difference between the observed data and the predicted values

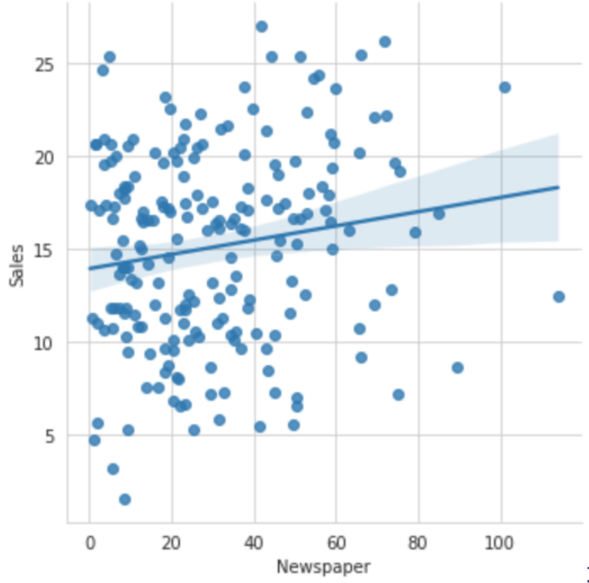
$R^2 = 0.812$



$R^2 = 0.122$



$R^2 = 0.024$



Performance Metrics

- We want to evaluate our regression model on the testing set
- Possible Metrics:
  - Mean Squared Error (MSE)
  - Root Mean Squared Error (RMSE)
  - Mean Absolute Error (MAE)

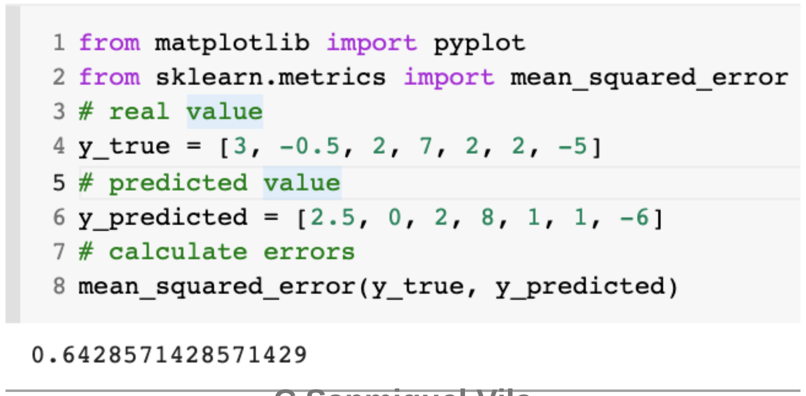
MSE & RMSE

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2$$

$$RMSE = \sqrt{MSE}$$

i	1	2	3	4	5	6	7
Predicted	2.5	0	2	8	1	1	-6
True	3	-0.5	2	7	2	2	-5

$$MSE = \frac{1}{7}\sum_{i=1}^{7}(3-2.5)^2+(-0.5-0)^2+(2-2)^2+(7-8)^2+(2-1)^2+(2-1)^2+(-5-(-6)^2$$



```
from matplotlib import pyplot
from sklearn.metrics import mean_squared_error
# real value
y_true = [3, -0.5, 2, 7, 2, 2, -5]
# predicted value
y_predicted = [2.5, 0, 2, 8, 1, 1, -6]
# calculate errors
mean_squared_error(y_true, y_predicted)
```

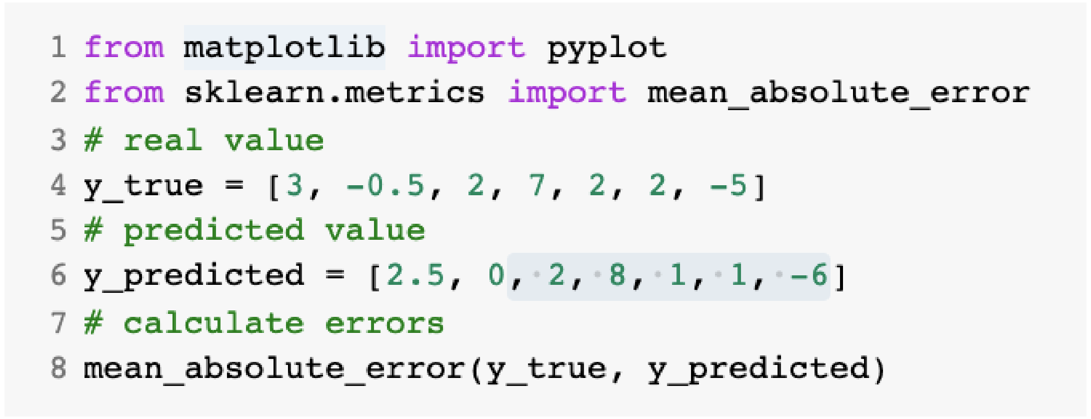
```
0.6428571428571429
```

Mean Absolute Error (MAE)

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |y^{(i)} - \hat{y}^{(i)}|$$

i	1	2	3	4	5	6	7
Predicted	2.5	0	2	8	1	1	-6
True	3	-0.5	2	7	2	2	-5

$$MAE = \frac{1}{7}\sum_{i=1}^{7} |3 - 2.5| + |-0.5 - 0| + |2 - 2| + |7 - 8| + |2 - 1| + |2 - 1| + |-5 - (-6)|$$



```python
from matplotlib import pyplot
from sklearn.metrics import mean_absolute_error
# real value
y_true = [3, -0.5, 2, 7, 2, 2, -5]
# predicted value
y_predicted = [2.5, 0, 2, 8, 1, 1, -6]
# calculate errors
mean_absolute_error(y_true, y_predicted)
```

```
0.7142857142857143
```

MSE vs RMSE vs MAE

- MSE and RMSE penalize large prediction errors via squared function
  - RMSE is more commonly used than MSE as it has the same unit as Y
- MAE is more robust to outliers
- MSE and RMSE are differentiable functions while MAE is not
  - i.e., its derivative exists for all values
  - This makes RMSE used as a default metric for loss function in most ML models
  - The lower the RMSE/MSE is, the better the model is
    - Scoring function: neg_root_mean_squared_error/neg_mean_squared_error
    - https://scikit-learn.org/stable/modules/model_evaluation.html

Negative RMSE & MAE

- In scikit-learn, the metrics for some scoring functions (like cross-validation) are framed so that maximization is preferred. Since RMSE needs to be minimized (lower is better), scikit-learn returns the negative value of RMSE in functions like cross_val_score() or GridSearchCV(), where it expects scores to be maximized. This ensures that a higher (less negative) score is better during optimization.

Negative RMSE & MAE

When to use negative RMSE

- You use negative RMSE when scikit-learn expects scores that need to be maximized, such as Cross-validation with cross_val_score() or other scikit-learn functions such as hyperparameter tuning with GridSearchCV() or RandomizedSearchCV()
- In these cases, scikit-learn requires a metric where higher scores indicate better performance (since these methods are designed to maximize scores). RMSE, on the other hand, is a metric where lower values are better, so scikit-learn returns its negative value to make the scoring consistent with its optimization framework (where higher scores are considered better)
- By using negative errors, scikit-learn turns the problem of minimizing RMSE into one where it's maximizing the negative of RMSE. As a result, a less negative score (closer to 0) indicates better performance

Negative RMSE & MAE

When to use RMSE:

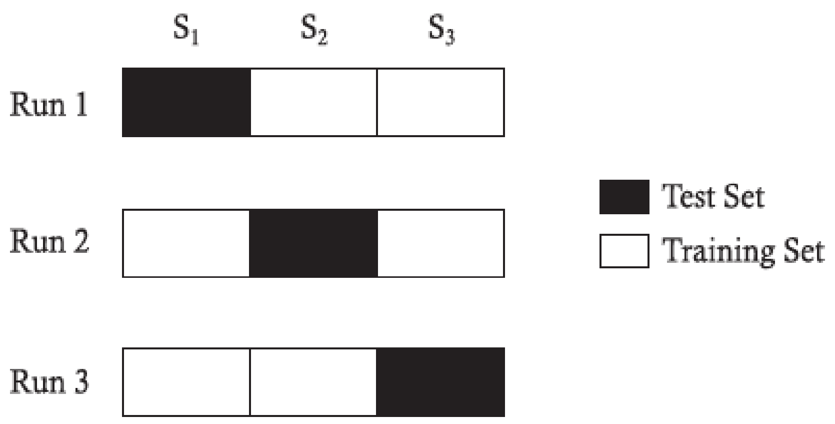
- You use RMSE when manually calculating the RMSE outside of scikit-learn's cross-validation or optimization functions and are interested in the actual RMSE value for interpretation or model comparison.
- For example, you might use RMSE in:
  - Manual evaluation of a model's performance on a test set.
  - Reporting final model performance to give a direct sense of error in the target's units (e.g., predicting house prices or temperature).

Avoid RMSE:

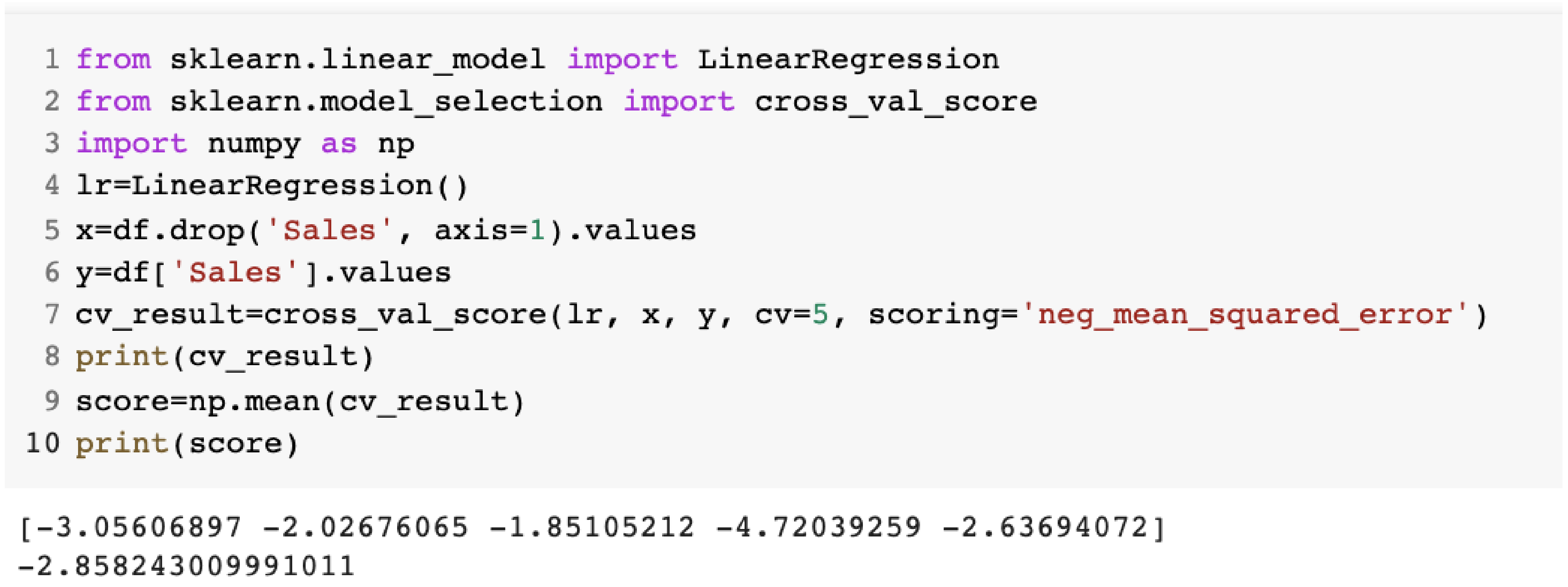
- If you're dealing with data that contains many outliers or non-normally distributed residuals, you might prefer using **MAE** (Mean Absolute Error), which is less sensitive to outliers.

Cross validation for Linear Regression

- **Avoid the performance evaluation being sensitive to the selection of training data**
- **K-fold cross validation:**
  - Divide the whole dataset into K partitions
  - (K-1) partitions for training, 1 partition for testing
  - Rotate the selection of the testing set
  - Average the evaluation



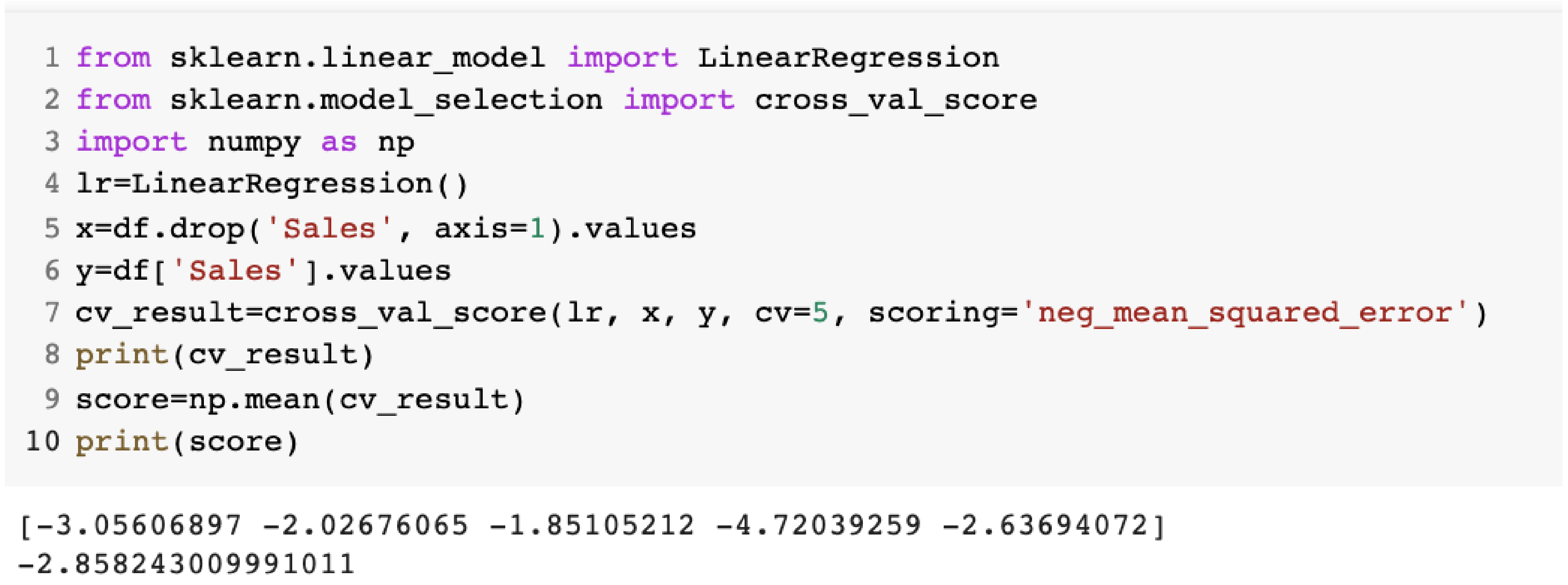
Cross validation for Linear Regression



```python
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score
import numpy as np
lr=LinearRegression()
x=df.drop('Sales', axis=1).values
y=df['Sales'].values
cv_result=cross_val_score(lr, x, y, cv=5, scoring='neg_mean_squared_error')
print(cv_result)
score=np.mean(cv_result)
print(score)
```

```
[-3.05606897 -2.02676065 -1.85105212 -4.72039259 -2.63694072]
-2.858243009991011
```

Cross validation for Linear Regression



```python
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score
import numpy as np
lr=LinearRegression()
x=df.drop('Sales', axis=1).values
y=df['Sales'].values
cv_result=cross_val_score(lr, x, y, cv=5, scoring='neg_mean_squared_error')
print(cv_result)
score=np.mean(cv_result)
print(score)
```

```
[-3.05606897 -2.02676065 -1.85105212 -4.72039259 -2.63694072]
-2.858243009991011
```