2025-04-15 16:45

tags :

- Machine Learning
- Hands on ML - book

Chapter 4 - Hands on

- This chapter dives into how machine learning models work under the hood. It covers:
    - **Linear Regression** using closed-form solutions and **Gradient Descent** (plus its variants).
    - **Polynomial Regression**, overfitting, and how to prevent it with **learning curves** and **regularization**.
    - **Logistic** and **Softmax Regression** for classification tasks.

Linear Regression

- What is Linear Regression?
    - It’s a model that predicts an output by computing a **weighted sum of input features**, plus a **bias** (intercept).
    - The general form is:
      $\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$
    - In vector form:
      $\hat{y} = \theta^T \cdot x$
    - (where θ is the vector of parameters, and x is the vector of input features including a 1 for the bias term)
- How Do We Train It?
    - We adjust θ (the model’s parameters) to best fit the training data.
    - To measure how well it fits, we use **Mean Squared Error (MSE)** as the cost/loss function: $\text{MSE}(\theta) = \frac{1}{m}\sum_{i=1}^{m}(\theta^T x^{(i)} - y^{(i)})^2$
    - This helps us find the values of θ that minimize prediction errors.
- Training loss (like MSE) is optimized during training.
- Evaluation metrics (like precision/recall) may differ from training loss, especially for classifiers.
- Good loss functions are **easy to optimize** and **correlate with real-world goals**.

The Normal Equation

- What is the **Normal Equation**?
  - It's a **direct formula** to compute the best parameters **θ** that minimize the MSE (no iteration needed): $\hat{\theta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T y$
- How It Works:
  1. Generate training data with some noise:
     $y = 4 + 3x + \text{noise}$
  2. Add a column of 1s to `X` (to handle the bias term).
  3. Use NumPy to compute **θ** directly using matrix operations.
  4. Predictions are made by multiplying new input data by the computed **θ**.
- Example Results:
  - The true parameters were 4 and 3.
  - The model estimated them as **4.215** and **2.770** — close, but slightly off due to noise.
- Using Scikit-Learn:
  - You can use `LinearRegression()` from scikit-learn to do the same thing with `.fit()` and `.predict()`.
  - It separates the **intercept_** (bias) and **coef_** (feature weights).
- **Pseudoinverse Alternative:**
  - Instead of using the Normal Equation, you can use the **pseudoinverse**:
    $\hat{\theta} = X^+ y$
    - Computed using `np.linalg.pinv(X)`
    - More stable and handles edge cases where `XᵀX` is not invertible (e.g., redundant features or too few data points).
- **Normal Equation** is a fast, direct way to compute parameters for linear regression.
- **Pseudoinverse** is more robust and always works, even when the Normal Equation fails.
- Libraries like **scikit-learn** and **NumPy** make it easy to use both.

Computational Complexity

- Normal Equation & SVD Complexity:
  - **Normal Equation** has a complexity of about **$O(n^{2.4})$ to $O(n^3)$** → gets **slow with many features**.
  - **SVD** (used by Scikit-Learn) is about **$O(n^2)$** → still slow when **n (features)** is very large.
  - **Both handle large datasets (many rows, m) well**, since they are **linear in m**: **O(m)**.
- Prediction Time:
  - Once the model is trained, making predictions is **very fast**.
  - Time grows **linearly** with the number of instances and features.

Gradient Descent

- What is Gradient Descent?
  - An **optimization algorithm** that finds the best model parameters (like θ) by minimizing a **cost function** (e.g., MSE).
  - It does this by **iteratively moving in the direction of the steepest slope** (i.e., negative gradient).
- How It Works:
  1. **Start** with random parameter values (random initialization).
  2. Repeatedly **adjust parameters** to reduce the cost.
  3. **Stop when** the gradient becomes zero → minimum reached.
- Learning Rate (Step Size):
  - **Too small** → slow convergence (takes many steps).
  - **Too large** → may overshoot or diverge (never settles).
  - Must choose a **balanced learning rate**!
- hallenges:
  - Some cost functions can have **plateaus, local minima**, or **complex shapes**.
  - BUT for **linear regression**, the MSE cost function is:
    - ✅ **Convex** (no local minima)
    - ✅ **Smooth** (no sharp changes)
    - ➡️ So, gradient descent is **guaranteed** to find the global minimum eventually.
- Feature Scaling:
  - If features have **different scales**, gradient descent:
    - Takes weird zigzag paths,
    - Converges **very slowly**.
  - Use tools like `StandardScaler` to **normalize** feature scales.

Batch Gradient Descent

- **What is Batch Gradient Descent?**
  - Batch Gradient Descent is an optimization algorithm used to **minimize the cost function** (like MSE in linear regression) by adjusting model parameters θ\thetaθ.
- **Key Concepts**
  1. **Cost Function (MSE)**
     Measures how bad the model's predictions are: $\text{MSE}(\theta) = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2$
  2. **Gradient (Slope Vector)**
     Tells you how to tweak θ to reduce the error. It's the vector of **partial derivatives** of the cost function with respect to each parameter θ_j .

- manual for each paramete :

$$\frac{\partial}{\partial \theta_j} \mathrm{MSE}(\theta) = \frac{2}{m} \sum_{i=1}^{m} (\theta^\top \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

- vectorized for all parameters :

$$\nabla_\theta \mathrm{MSE}(\theta) = \frac{2}{m} \mathbf{X}^\top (\mathbf{X}\theta - \mathbf{y})$$

3. Gradient Descent Update Rule
   Use the gradient to update θ by taking a step **opposite the slope**:
   $\theta := \theta - \eta \cdot \nabla_\theta \mathrm{MSE}(\theta)$
   - η = learning rate (step size)
4. **Code :-**

```
eta = 0.1                # learning rate
n_epochs = 1000          # number of passes over the dataset
m = len(X_b)             # number of samples

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

for epoch in range(n_epochs):
    gradients = 2 / m * X_b.T @ (X_b @ theta - y)
    theta = theta - eta * gradients
```

5. Learning Rate Matters
   - **Too small** → very slow convergence
   - **Just right** → fast and stable
   - **Too large** → divergence (jumps around, doesn't settle)
6. **When to Stop?**
   - Set a max number of epochs, **or**
   - Stop early when the gradient becomes very small (i.e., close to minimum)
7. **Convergence**
   - Gradient descent will converge for convex functions like MSE
   - The convergence speed depends on the **learning rate** and **tolerance** ϵ

Stochastic Gradient Descent

- **What is SGD?**
  - Unlike **Batch Gradient Descent** (which uses the whole dataset per step), **SGD** uses **only one random data point per update**.

    - This makes each step much **faster** and allows training on very **large datasets**.
- **Pros and Cons**
    - pros
        - Faster updates
        - Can escape local minima
        - Scalable to huge datasets
    - Cons
        - More noisy (bouncy) path to minimum
        - Doesn't settle exactly at the minimum
        - May require careful tuning of learning rate
- **Epochs and Iterations**
    - One **epoch** = one full pass through the dataset.
    - Each **epoch** consists of `m` **updates** (one per training example).
    - If random sampling is used, some examples might be picked multiple times or not at all in one epoch.
- **Learning Rate Schedule**
    - To reduce noise and allow convergence, **learning rate should shrink over time**.
    - This is done using a **learning schedule**, e.g.:

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

    - Starts with larger steps (fast progress), then gets smaller (finer tuning).
- **Important Considerations**
    - Data must be **IID (independent and identically distributed)** — shuffle it!
    - If data is ordered (e.g., by label), SGD can behave badly.
- **SGD in Code**

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

- **Scikit-Learn's SGDRegressor**

```python
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(
    max_iter=1000, tol=1e-5, eta0=0.01, penalty=None,
    n_iter_no_change=100, random_state=42
)
sgd_reg.fit(X, y.ravel())
```

Mini-Batch Gradient Descent

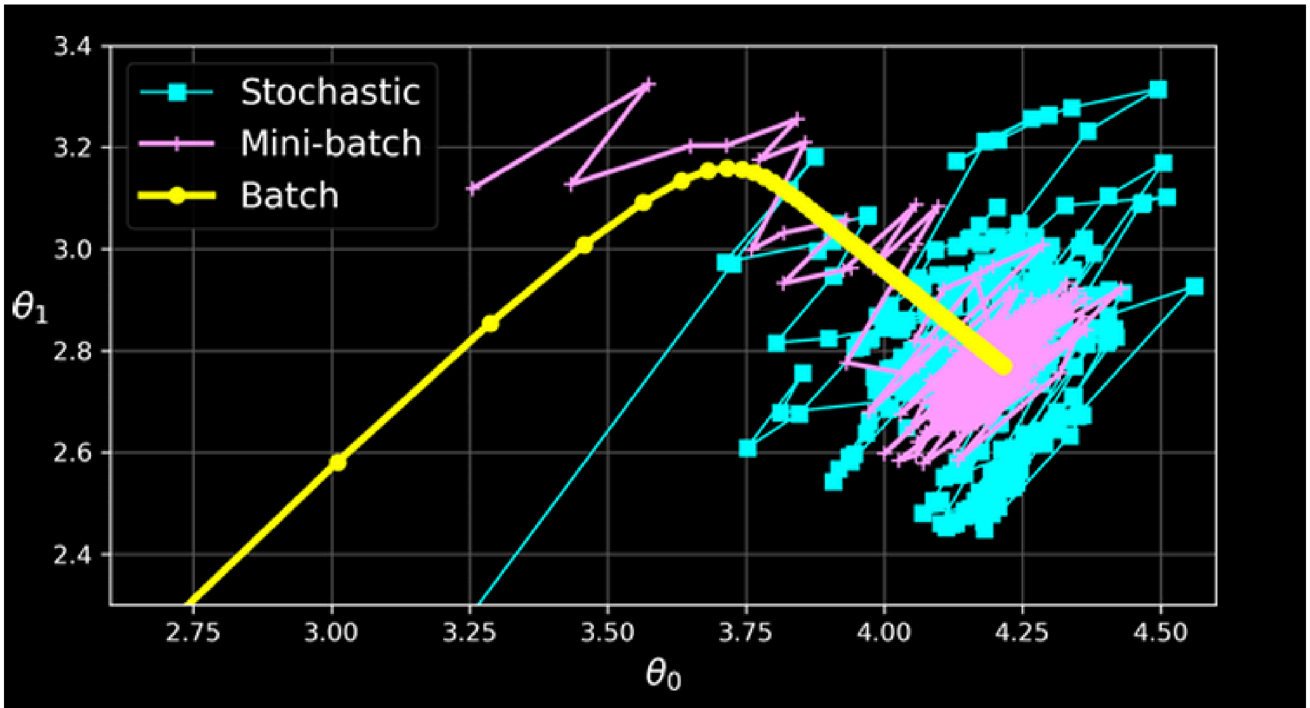
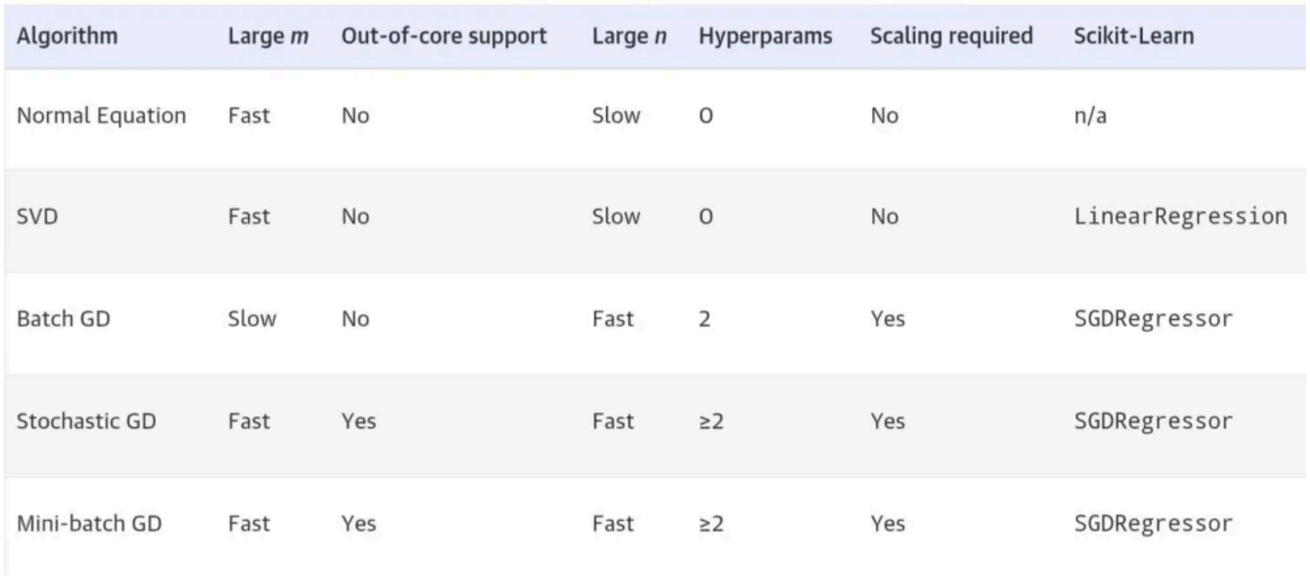
- Mini-batch Gradient Descent (MBGD) is a hybrid between:
  - **Batch GD**: Uses the whole dataset per step (slow but stable).
  - **Stochastic GD (SGD)**: Uses 1 sample per step (fast but noisy).
- **MBGD** uses **small random batches** of data (e.g. 32 or 64 samples).
- **Advantages**
  - Faster than Batch GD.
  - Less noisy than SGD.
  - Can benefit from hardware acceleration (like GPUs).
  - Works well with **large datasets**.
- **Disadvantages**
  - Still bounces around the minimum (but less than SGD).
  - Might get stuck in local minima (especially for non-convex problems).
  - Needs a learning schedule (like SGD).
- 

Figure 4-11. Gradient descent paths in parameter space

- **Batch GD** : smooth, direct to the minimum, stops there.
- **Stochastic GD** : very noisy path, keeps bouncing.
- **Mini-batch GD** : less noisy than SGD, ends near the minimum.

- 

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`

Polynomial Regression

- **Polynomial Regression Overview**
  - **Goal**: Fit **nonlinear data** using a **linear model**.
  - **How**: Add **powers of features** (e.g. x^2, x^3, etc.) as **new features**, then apply **linear regression**.
- **Example Process**:
  1. **Generate nonlinear data**:
     - $y = 0.5x^2 + x + 2 + noise$
  2. **Use** `PolynomialFeatures` from `sklearn` :

     ```
     from sklearn.preprocessing import PolynomialFeatures
     poly_features = PolynomialFeatures(degree=2, include_bias=False)
     X_poly = poly_features.fit_transform(X)
     ```

     - Turns $[x]$ into $[x, x^2]$
  3. **Train Linear Regression** on transformed data:

     ```
     from sklearn.linear_model import LinearRegression
     lin_reg = LinearRegression()
     lin_reg.fit(X_poly, y)
     ```

- **Model Output Example**:

- The model learns:

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

- Close to the original:

$$\hat{y} = 0.5x^2 + 1x + 2 + noise$$

- **Key Points**:
  - **Polynomial regression ≠ nonlinear model** — it's still **linear in parameters**.
  - When multiple features are present, **interactions** (e.g., ab, a^2b) are added automatically.
  - **Warning**: With high degree ddd and many features nnn, the number of features grows **combinatorially**:

$$\text{Features} = \frac{(n+d)!}{d! \cdot n!}$$

Learning Curves

- What Are They?
  - **Plots of:** Training error and Validation error vs. Training set size.
  - Help diagnose:
    - **Underfitting** (high bias)
    - **Overfitting** (high variance)
- Key Observations
  - **Underfitting Example (Linear Model)**:

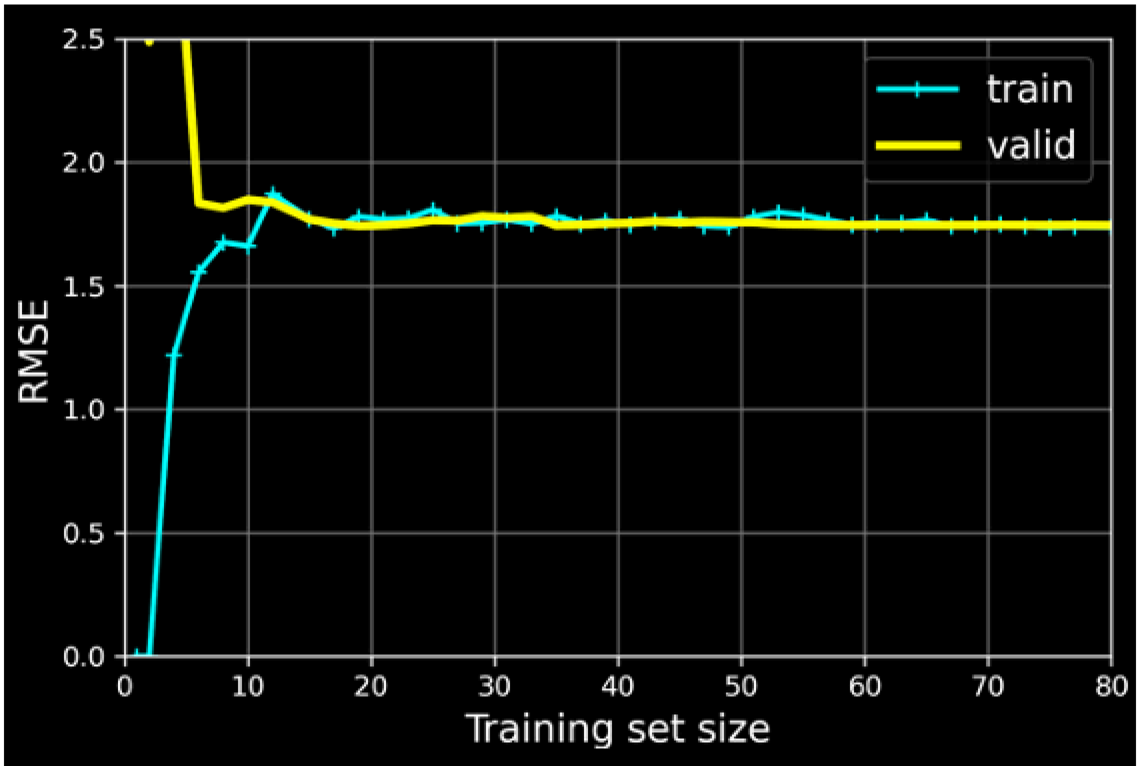
- 

Figure 4-15. Learning curves

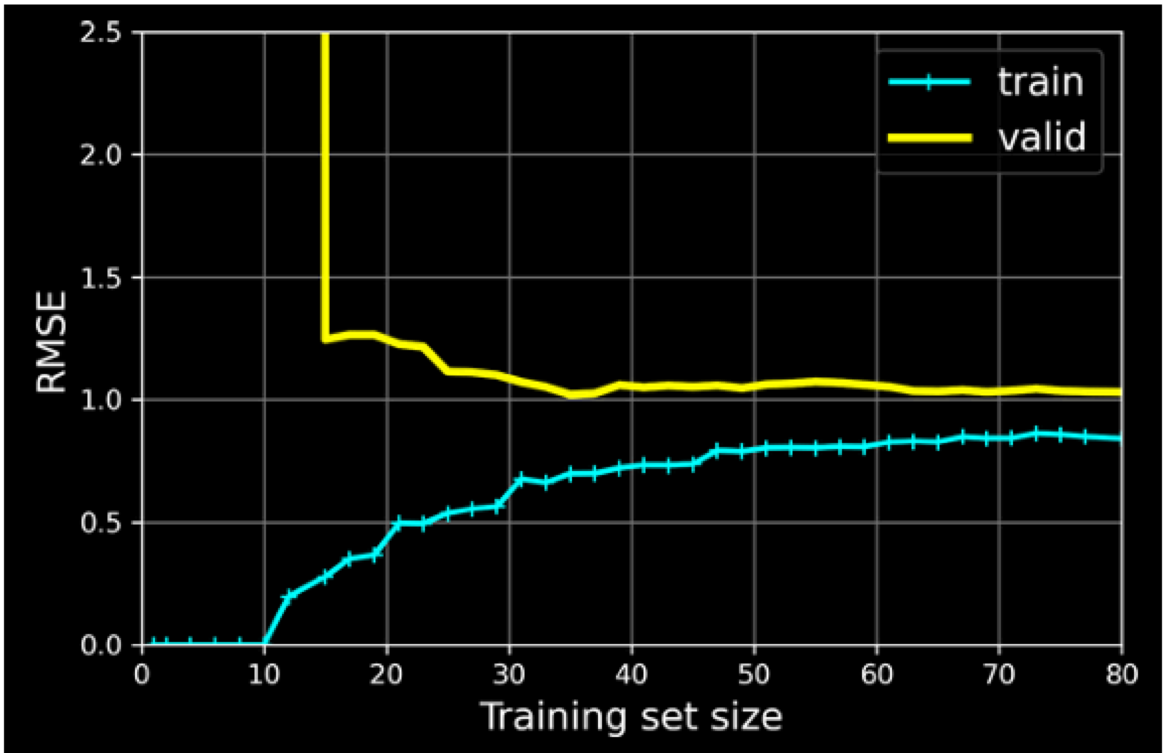
  - **Training error**: Starts low (small data) then rises and plateaus
  - **Validation error**: Starts high, decreases, then plateaus near training error
  - → **Both errors high & close** = Model too simple (high bias)
- **Overfitting Example (10th-degree Polynomial)**:
  - 

Figure 4-16. Learning curves for the 10th-degree polynomial model

  - **Training error**: Very low

        - **Validation error**: Higher and doesn't match training error
        - → **Big gap between curves** = Model too complex (high variance)
        - ✅ Can often fix with **more training data**
- **Tips:**
    - **Underfitting?** → Try a more complex model or better features
    - **Overfitting?** → Try regularization or add more data
    - **Irreducible error?** → Clean up your dataset

Regularized Linear Models

- **Regularization** helps reduce **overfitting** by limiting model complexity.
- For **polynomial models**, reduce the **degree**.
- For **linear models**, **constrain the weights** using:
    1. **Ridge Regression (L2)** – Shrinks all weights.
    2. **Lasso Regression (L1)** – Shrinks some weights to **zero** (feature selection).
    3. **Elastic Net** – Mix of L1 and L2 regularization.

Ridge Regression

- What Is Ridge Regression?
    - **Ridge Regression** (aka **Tikhonov regularization**) is **Linear Regression + L2 regularization**.
    - The goal: Fit the data **while keeping model weights small**, reducing overfitting.
- **Cost Function**
    - $J(\theta) = \text{MSE}(\theta) + \frac{\alpha}{m}\sum_{i=1}^{n}\theta_i^2$
    - Adds a **penalty** for large weights.
    - Bias term θ_0 is **not** regularized.
    - α controls regularization strength:
        - α = 0→ plain Linear Regression.
        - Large α → very flat model.
- **Important: Always Scale Your Features**
    - Use `StandardScaler` before applying Ridge. Regularization is sensitive to feature scales.
- **Intuition**
    - **High α\alphaα** → simple, flat predictions → **low variance**, **high bias**.
    - **Low α\alphaα** → more flexible model → **low bias**, **high variance**.
    - This helps manage the **bias-variance trade-off**.
- **How To Use Ridge in Scikit-Learn**

1. Closed-form (exact solution):

```
from sklearn.linear_model import Ridge
ridge_reg = Ridge(alpha=0.1, solver="cholesky")
ridge_reg.fit(X, y)
ridge_reg.predict([[1.5]])
```

2. Using Stochastic Gradient Descent (SGD):

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(penalty="l2", alpha=0.1/m, tol=None,
                 max_iter=1000, eta0=0.01, random_state=42)
sgd_reg.fit(X, y.ravel())  # y must be 1D
sgd_reg.predict([[1.5]])
```

- **Extra Tip: Automatic Tuning**
  - Use `RidgeCV` for **automatic α tuning** with cross-validation:

```
from sklearn.linear_model import RidgeCV  ridge_cv = RidgeCV(alphas=[0.01,
0.1, 1.0, 10.0])
ridge_cv.fit(X, y)
```

Lasso Regression

- **What is Lasso Regression?**
  - **Lasso** stands for **Least Absolute Shrinkage and Selection Operator**.
  - It's a regularized version of **linear regression** that adds an **ℓ_1 (absolute value) penalty** to the cost function.
- Cost Function:

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

  - ℓ_1 norm = sum of absolute values of the weights (no squares like Ridge).
  - This encourages some weights to become exactly **zero** → **feature selection**!
- **Effect of Lasso:**
  - Can **zero out** unimportant features, making the model **sparse** and easier to interpret.
  - Increasing **α** increases regularization (more weights shrink to zero).
  - Decreasing **α** moves the solution toward plain linear regression (no regularization).
- **Gradient and Subgradient:**

    - Lasso's cost function is **not differentiable at θ = 0**.
    - Gradient descent uses a **subgradient** instead:
        - `sign(θ_i)` is used: returns -1, 0, or 1 based on the sign of each θ.
- You can also use `SGDRegressor(penalty="l1")` to perform lasso-style training with SGD.
- Scikit-Learn Example:

```
from sklearn.linear_model import Lasso

lasso_reg = Lasso(alpha=0.1)
lasso_reg.fit(X, y)
lasso_reg.predict([[1.5]])
# Output: array([1.53788174])
```

Elastic Net Regression

- **What is Elastic Net?**
    - **Elastic Net** combines both **Ridge (ℓ_2)** and **Lasso (ℓ_1)** penalties.
    - It's a **weighted average** of Ridge and Lasso regularization.
- **Cost Function:**
$J(\theta) = MSE(\theta) + r \cdot (2\alpha \sum \mid \theta i \mid) + (1 - r) \cdot (\frac{\alpha}{m} \sum \theta i^2)$
    - `r = 0`: behaves like **Ridge**
    - `r = 1`: behaves like **Lasso**
- **When to Use Elastic Net?**
    - Use **regularization** in general (avoid plain linear regression).
    - **Ridge** is a good default.
    - Use **Lasso/Elastic Net** if you believe:
        - Only a few features are important
        - You want **automatic feature selection**
    - Prefer **Elastic Net** over Lasso when:
        - You have **more features than samples**
        - Some features are **highly correlated**
- Scikit-Learn Example:

```
from sklearn.linear_model import ElasticNet

elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)  # l1_ratio = r
elastic_net.fit(X, y)
elastic_net.predict([[1.5]])
# Output: array([1.54333232])
```

Early Stopping

- **What is Early Stopping?**
  - A **regularization technique** for iterative learning (like gradient descent).
  - You **stop training** when the **validation error stops improving**.
  - Helps **prevent overfitting** by not overtraining the model.
  - Called a “**beautiful free lunch**” by Geoffrey Hinton because it's simple yet powerful.
- **How It Works (Conceptually)**
  - During training:
    - Training error keeps going **down**.
    - Validation error goes **down**, then **up** (indicating overfitting).
  - **Early stopping** halts training **at the lowest validation error**, before overfitting begins.
- **Code**
  1. Data Prep

     ```
     X_train, y_train, X_valid, y_valid = [...]  # split dataset

     preprocessing = make_pipeline(
                         PolynomialFeatures(degree=90, include_bias=False),
                         StandardScaler())
     X_train_prep = preprocessing.fit_transform(X_train)
     X_valid_prep = preprocessing.transform(X_valid)
     ```

  2. Model Setup

     ```
     sgd_reg = SGDRegressor(penalty=None, eta0=0.002, random_state=42)
     ```

     - Uses **Stochastic Gradient Descent** (SGD).
     - **No regularization** (`penalty=None`).
     - Small learning rate `eta0=0.002` .
  3. Training with Early Stopping

     ```
     from copy import deepcopy
     n_epochs = 500
     best_valid_rmse = float('inf')  # start with the worst possible error
     ```

```
for epoch in range(n_epochs):
    sgd_reg.partial_fit(X_train_prep, y_train)  # train one step
    y_valid_predict = sgd_reg.predict(X_valid_prep)
    val_error = mean_squared_error(y_valid, y_valid_predict,
squared=False)  # RMSE

    if val_error < best_valid_rmse:  # if new best
        best_valid_rmse = val_error
        best_model = deepcopy(sgd_reg)  # save current model
```

Logistic Regression

- A classification algorithm (despite its name).
- Predicts the **probability** an instance belongs to class **1** (positive class).
- If probability > 50%, it predicts **1**, else **0** (negative class).
- Used for **binary classification** problems (e.g., spam detection).

Estimating Probabilities

- Logistic regression computes:

$$\hat{p} = \sigma(\theta^T \mathbf{x})$$
$$\text{where } \sigma(t) = \frac{1}{1+e^{-t}} \text{ is the sigmoid function.}$$

- Output is a **probability** between 0 and 1.
- Prediction rule:
    - If $\hat{p}$ ≥ 0.5, predict class **1**.
    - If $\hat{p}$ < 0.5, predict class **0**.

Training and Cost Function

- Train the model so that:
    - High probabilities for **positive class (y = 1)**
    - Low probabilities for **negative class (y = 0)**
- Single Instance Cost
    - $$\text{Cost} = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1-\hat{p}) & \text{if } y = 0 \end{cases}$$
- Total Cost (Log Loss)

- $$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(\hat{p}^{(i)}) + (1-y^{(i)})\log(1-\hat{p}^{(i)})\right]$$
    - Called **log loss**, and it's **convex**, so Gradient Descent can find the **global minimum**.
- Gradient
  - $$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(\sigma(\theta^\top \mathbf{x}^{(i)}) - y^{(i)}\right)x_j^{(i)}$$
  - Same idea as linear regression: error × feature → average over all samples.
  - Used in **batch, stochastic**, or **mini-batch Gradient Descent** to update weights.

Decision Boundaries

- Dataset Used: Iris Dataset
  - 150 iris flowers, 3 species: **Setosa**, **Versicolor**, **Virginica**
  - Features: sepal/petal **length** and **width**
- **Goal:**
  - Classify **whether a flower is Iris Virginica** based on petal width.
- **Steps:**
  1. **Load data** using `load_iris()`
  2. **Use only petal width** as input feature
  3. Create binary labels:
     `y = True` if **Virginica**, `False` otherwise
  4. **Split data** → Train/Test
  5. **Train** logistic regression model with `fit()`
  6. **Predict probabilities** with `predict_proba()`
- **Decision Boundary:**
  - Model predicts **Virginica** if **probability ≥ 0.5**
  - This creates a **threshold (decision boundary)** at **~1.65 cm**
  - If **petal width > 1.65 cm** → **Virginica**, else not

Softmax Regression

- Softmax Regression is a generalization of logistic regression that works for multiple classesdirectly — no need to train separate binary classifiers.
- **How It Works (Prediction)**

1. For an input $\mathbf{x}$, the model calculates a **score** for each class: $s_k(\mathbf{x}) = \theta^{(k)\top}\mathbf{x}$,Each class has its own weight vector $\theta^{(k)}$
2. It then applies the **softmax function** to convert these scores into **probabilities**:

$$\hat{p}_k = \frac{e^{s_k}}{\sum_{j=1}^{K} e^{s_j}}$$

   - K is the number of classes.
   - The output is a probability distribution over the classes.
3. It **predicts** the class with the **highest probability**: $\hat{y} = \arg\max_k \hat{p}_k \quad \text{(predicted class)}$

- Training the Model
  - The model is trained to **maximize the probability** of the correct class.
  - It uses the **cross-entropy loss function**:

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(\hat{p}_k^{(i)})$$

    - $y_k^{(i)}$=1 if example i belongs to class k, else 0.
    - This function penalizes the model when it assigns **low probability** to the correct class.
    - When K=2, this reduces to the familiar **log loss** used in binary logistic regression.
- Key Tip
  - Softmax is good for **mutually exclusive classes** (e.g., flower types), but **not** for multi-label problems (like detecting multiple objects in one image).
- Cross Entropy in Machine Learning
  - Used as the **loss function** for classification tasks.
  - Penalizes wrong class probability predictions.
  - Especially common in **softmax regression** and **neural networks**.
- Gradient of Cross Entropy
  - Helps in **training the model** using gradient descent:

$$\nabla_{\theta^{(k)}} J(\Theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{p}_k^{(i)} - y_k^{(i)}\right)\mathbf{x}^{(i)}$$

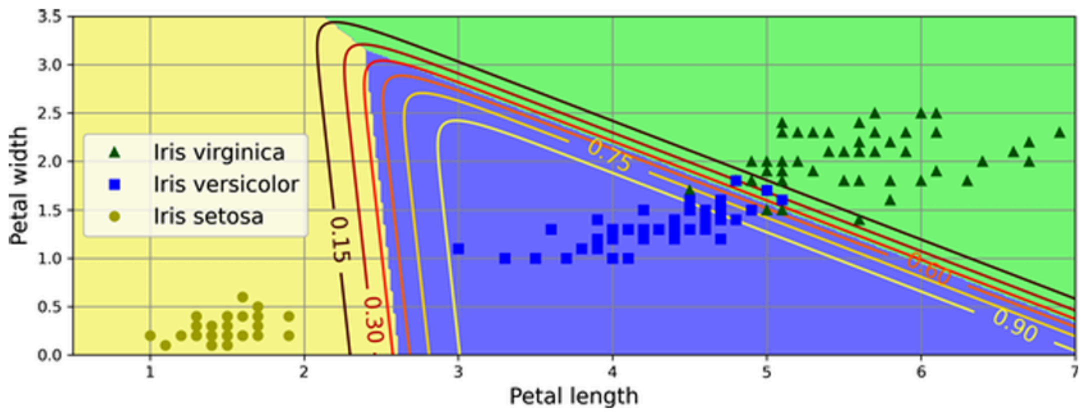
  - Compute gradient per class → update weights → repeat.
- **Softmax Regression with Scikit-Learn**
  - Automatically used when there are **more than 2 classes**.
  - Uses **LogisticRegression** with `solver="lbfgs"` (default).
  - Regularized using L2 by default (controlled by `C`).
  - **Example: Classify Iris Flowers**

```
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = iris["target"]
X_train, X_test, y_train, y_test = train_test_split(X, y,
random_state=42)
softmax_reg = LogisticRegression(C=30, random_state=42)
softmax_reg.fit(X_train, y_train)
```

- Prediction Example:

```
softmax_reg.predict([[5, 2]])  # → array([2])
softmax_reg.predict_proba([[5, 2]])  # → [[0.  , 0.04, 0.96]]
```

- **Visualization**



- *Figure 4-25. Softmax regression decision boundaries*
- Softmax regression creates **linear decision boundaries** between classes.
- Curved lines represent **equal probability contours**.
- At the center where all three classes meet → model gives ~33% to each class.

Resources :

-

Related notes :

-

References :

- **Internal :**
    -
    -
    -
- **External :**
- notebook of the chapter
- hegab videos
- the book