

Machine Learning Lab

Session 2: Polynomial Regression & Classification

BFA3 – Université Paris-Dauphine

M2 Level

Duration: 2h30 (30 min lecture + 2h lab)

Date: _____

Learning Objectives

By the end of this lab, you will be able to:

1. Extend linear regression to polynomial regression for non-linear patterns
2. Implement the quadratic OLS objective function and its gradient
3. Understand classification as a fundamentally different problem from regression
4. Implement the logistic regression objective function (binary cross-entropy)
5. Apply these techniques to financial market data analysis

Prerequisites

This lab builds upon Session 1. You should be comfortable with:

- Linear regression and OLS objective function
- Gradient descent algorithm (implemented in Session 1)
- Computing partial derivatives
- Basic probability (for logistic regression)

Files Provided

- `ml_lab2_gui.py` – The GUI application with gaps to complete

To run the application after completing the TODOs:

```
python ml_lab2_gui.py
```

Part I: Polynomial Regression

1 Beyond Linear Models

In Session 1, we fitted linear models of the form $\hat{y} = ax + b$. However, many real-world phenomena exhibit **non-linear** relationships that cannot be captured by a straight line.

Financial Example: Non-Linear Patterns in Finance

Consider modeling the relationship between:

- **Implied volatility vs. strike price:** The “volatility smile” is a well-known U-shaped (quadratic) pattern in options markets
- **Bond price vs. yield:** The convexity of bond prices creates a curved relationship
- **Trading costs vs. order size:** Market impact often follows a square-root or polynomial law

A linear model would systematically miss these patterns.

2 Quadratic Regression Model

Definition: Quadratic Regression

A quadratic regression model has the form:

$$\hat{y} = ax^2 + bx + c$$

where:

- a is the **quadratic coefficient** (controls curvature)
- b is the **linear coefficient** (controls slope at $x = 0$)
- c is the **intercept**

The sign of a determines whether the parabola opens upward ($a > 0$) or downward ($a < 0$).

Financial Example: Volatility Smile Modeling

In options pricing, the implied volatility σ often varies with the strike price K in a parabolic pattern:

$$\sigma(K) = a(K - K_{ATM})^2 + b(K - K_{ATM}) + \sigma_{ATM}$$

where K_{ATM} is the at-the-money strike. This “smile” or “smirk” shape is crucial for:

- Pricing exotic options
- Risk management (Greeks computation)
- Detecting arbitrage opportunities

3 Quadratic OLS Objective Function

To fit a quadratic model, we minimize the sum of squared errors:

Definition: Quadratic OLS Objective

Given data points $\{(x_i, y_i)\}_{i=1}^n$, the objective function is:

$$\mathcal{L}(a, b, c) = \frac{1}{n} \sum_{i=1}^n (y_i - (ax_i^2 + bx_i + c))^2$$

This is the **Mean Squared Error (MSE)** between observations and predictions.

3.1 Simplification: Optimal Intercept

As in Session 1, we can reduce the number of parameters by computing c optimally as a function of a and b :

Theorem: Optimal Intercept

Setting $\frac{\partial \mathcal{L}}{\partial c} = 0$ gives:

$$c^*(a, b) = \bar{y} - a\bar{x}^2 - b\bar{x}$$

where $\bar{y} = \frac{1}{n} \sum y_i$, $\bar{x} = \frac{1}{n} \sum x_i$, and $\bar{x}^2 = \frac{1}{n} \sum x_i^2$.

Substituting c^* back, we optimize over only two parameters (a, b) :

$$\mathcal{L}(a, b) = \frac{1}{n} \sum_{i=1}^n (y_i - ax_i^2 - bx_i - c^*(a, b))^2$$

3.2 Gradient of the Quadratic OLS Objective

To apply gradient descent, we need the partial derivatives:

Theorem: Gradient Components

Let $r_i = y_i - ax_i^2 - bx_i - c^*(a, b)$ be the residual for point i . Then:

$$\frac{\partial \mathcal{L}}{\partial a} = -\frac{2}{n} \sum_{i=1}^n r_i \cdot (x_i^2 - \bar{x}^2)$$

$$\frac{\partial \mathcal{L}}{\partial b} = -\frac{2}{n} \sum_{i=1}^n r_i \cdot (x_i - \bar{x})$$

Derivation sketch: The chain rule accounts for how c^* depends on a and b . The terms $(x_i^2 - \bar{x}^2)$ and $(x_i - \bar{x})$ appear because changing a or b also changes c^* .

Part II: Classification

4 From Regression to Classification

Regression predicts a **continuous** value. Classification predicts a **discrete category**.

Financial Example: Classification in Finance

Many financial decisions are binary:

- **Credit scoring:** Will this borrower default? (Yes/No)
- **Trading signals:** Should we buy or sell? (Buy/Sell)
- **Fraud detection:** Is this transaction fraudulent? (Fraud/Legitimate)
- **Market regime:** Are we in a bull or bear market? (Bull/Bear)

5 Binary Classification Setup

Definition: Binary Classification Problem

Given data points $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ where:

- $\mathbf{x}_i = (x_{i,1}, x_{i,2}) \in \mathbb{R}^2$ are the features
- $y_i \in \{0, 1\}$ is the binary label (class)

The goal is to find a **decision boundary** that separates the two classes.

Financial Example: Market Regime Classification

Consider classifying trading days as “high volatility” ($y = 1$) or “low volatility” ($y = 0$) based on:

- x_1 : Previous day’s return
- x_2 : Trading volume relative to average

A linear decision boundary would be: $w_1x_1 + w_2x_2 + b = 0$

Days with $w_1x_1 + w_2x_2 + b > 0$ are classified as high volatility.

6 Logistic Regression Model

Why not use linear regression for classification? Because:

- Linear regression outputs can be < 0 or > 1 (not valid probabilities)
- MSE is not appropriate for binary outcomes

Definition: Sigmoid Function

The sigmoid (logistic) function maps any real number to $(0, 1)$:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Properties:

- $\sigma(0) = 0.5$
- $\lim_{z \rightarrow +\infty} \sigma(z) = 1$
- $\lim_{z \rightarrow -\infty} \sigma(z) = 0$
- $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ (useful for gradients)

Definition: Logistic Regression Model

For a 2D input $\mathbf{x} = (x_1, x_2)$, the model predicts:

$$P(y = 1|\mathbf{x}) = \sigma(w_1x_1 + w_2x_2 + b) = \frac{1}{1 + e^{-(w_1x_1 + w_2x_2 + b)}}$$

The **decision boundary** is where $P(y = 1) = 0.5$, i.e., where:

$$w_1x_1 + w_2x_2 + b = 0$$

This is a straight line in 2D space.

7 Binary Cross-Entropy Loss

For classification, we use **cross-entropy** instead of MSE:

Definition: Binary Cross-Entropy (BCE)

$$\mathcal{L}(w_1, w_2, b) = -\frac{1}{n} \sum_{i=1}^n [y_i \log(p_i) + (1 - y_i) \log(1 - p_i)]$$

where $p_i = \sigma(w_1x_{i,1} + w_2x_{i,2} + b)$ is the predicted probability.

Intuition:

- When $y_i = 1$: Loss is $-\log(p_i)$. We want $p_i \rightarrow 1$, so $-\log(p_i) \rightarrow 0$.
- When $y_i = 0$: Loss is $-\log(1 - p_i)$. We want $p_i \rightarrow 0$, so $-\log(1 - p_i) \rightarrow 0$.
- Wrong predictions lead to high loss (approaching $+\infty$).

7.1 Gradient of the Cross-Entropy Loss

Theorem: Gradient of BCE

The gradient with respect to w_1 is:

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{1}{n} \sum_{i=1}^n (p_i - y_i) \cdot x_{i,1}$$

Similarly for w_2 and b . The term $(p_i - y_i)$ is the **prediction error**.

This elegant form comes from the derivative of the sigmoid: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

Part III: Implementation

8 Task 1: Implement the Quadratic OLS Objective

Open `ml_lab2_gui.py` and find the function `create_quadratic_ols_objective`.

8.1 Function Specification

```

1 def create_quadratic_ols_objective(X, y):
2     """
3     Create Quadratic OLS objective:  $L(a, b)$  with  $c$  computed optimally.
4      $y = a \cdot x^2 + b \cdot x + c$ 
5
6     Args:
7         X: Input features (1D numpy array of x values)
8         y: Target values (1D numpy array)
9
10    Returns:
11        func: Function that computes loss given params [a, b]
12        gradient: Function that computes gradient given params [a, b]
13        get_params: Function that returns (a, b, c) given [a, b]
14    """

```

Listing 1: Quadratic OLS objective function structure

8.2 Student Template

```

1 def create_quadratic_ols_objective(X, y):
2     n = len(X)
3     X2 = X**2
4     y_mean = np.mean(y)
5     x_mean = np.mean(X)
6     x2_mean = np.mean(X2)
7
8     def func(params):
9         a, b = params
10         # =====
11         # TODO 1: Compute optimal c
12         # =====
13         # Formula:  $c = y\_mean - a * x2\_mean - b * x\_mean$ 
14         c = _____ # YOUR CODE HERE
15
16         # =====
17         # TODO 2: Compute residuals
18         # =====
19         # Residual:  $r_i = y_i - (a * x_i^2 + b * x_i + c)$ 
20         residuals = _____ # YOUR CODE HERE
21
22         # =====
23         # TODO 3: Compute and return MSE loss
24         # =====
25         # MSE =  $(1/n) * \sum(residuals^2)$ 
26         return _____ # YOUR CODE HERE
27
28     def gradient(params):
29         a, b = params

```

```

30     c = y_mean - a * x2_mean - b * x_mean
31     residuals = y - (a * X2 + b * X + c)
32
33     # =====
34     # TODO 4: Compute partial derivative with respect to a
35     # =====
36     # da = -2/n * sum(residuals * (X2 - x2_mean))
37     da = _____ # YOUR CODE HERE
38
39     # =====
40     # TODO 5: Compute partial derivative with respect to b
41     # =====
42     # db = -2/n * sum(residuals * (X - x_mean))
43     db = _____ # YOUR CODE HERE
44
45     return np.array([da, db])
46
47 def get_params(params):
48     a, b = params
49     c = y_mean - a * x2_mean - b * x_mean
50     return a, b, c
51
52 return func, gradient, get_params

```

Listing 2: TODO: Complete the quadratic OLS objective

Hint

TODO 1: Use the formula $c = \bar{y} - a \cdot \bar{x^2} - b \cdot \bar{x}$

TODO 2: Compute $r_i = y_i - (ax_i^2 + bx_i + c)$ using vectorized operations: $y - (a * X2 + b * X + c)$

TODO 3: Return the mean of squared residuals: `np.sum(residuals**2) / n`

TODO 4 & 5: Apply the gradient formulas. Remember X2 contains x_i^2 values.

9 Task 2: Implement the Logistic Regression Objective

Find the function `create_logistic_objective` in `ml_lab2_gui.py`.

9.1 Function Specification

```

1 def create_logistic_objective(X, y):
2     """
3     Create Logistic Regression objective (binary cross-entropy).
4
5     Args:
6         X: Input features (n x 2 numpy array)
7         y: Binary labels (numpy array of 0s and 1s)
8
9     Returns:
10        func: Function that computes BCE loss given w1
11        derivative: Function that computes gradient given w1
12        get_params: Function that returns (w1, w2, b) given w1
13    """

```

Listing 3: Logistic regression objective function structure

9.2 Student Template

```

1 def create_logistic_objective(X, y):
2     n = len(y)
3     x1 = X[:, 0]
4     x2 = X[:, 1]
5     x1_mean = np.mean(x1)
6     x2_mean = np.mean(x2)
7     var_x1 = np.var(x1)
8     var_x2 = np.var(x2)
9     var_ratio = var_x1 / var_x2 if var_x2 > 1e-10 else 1.0
10
11     def sigmoid(z):
12         z = np.clip(z, -500, 500) # Prevent overflow
13         # =====
14         # TODO 1: Implement the sigmoid function
15         # =====
16         # Formula: sigma(z) = 1 / (1 + exp(-z))
17         return _____ # YOUR CODE HERE
18
19     def func(w1):
20         w2 = w1 * var_ratio
21         b = -w1 * x1_mean - w2 * x2_mean
22
23         # =====
24         # TODO 2: Compute z = w1*x1 + w2*x2 + b
25         # =====
26         z = _____ # YOUR CODE HERE
27
28         # =====
29         # TODO 3: Compute predicted probabilities using sigmoid
30         # =====
31         p = _____ # YOUR CODE HERE
32
33         # Clip probabilities to avoid log(0)

```

```

34     eps = 1e-15
35     p = np.clip(p, eps, 1 - eps)
36
37     # =====
38     # TODO 4: Compute and return binary cross-entropy loss
39     # =====
40     # BCE = -mean(y * log(p) + (1-y) * log(1-p))
41     loss = _____ # YOUR CODE HERE
42     return loss
43
44     def derivative(w1):
45         w2 = w1 * var_ratio
46         b = -w1 * x1_mean - w2 * x2_mean
47         z = w1 * x1 + w2 * x2 + b
48         p = sigmoid(z)
49
50         # =====
51         # TODO 5: Compute prediction error (p - y)
52         # =====
53         error = _____ # YOUR CODE HERE
54
55         # =====
56         # TODO 6: Compute derivative with respect to w1
57         # =====
58         # Account for chain rule with w2 and b depending on w1
59         dw1 = np.mean(error * (x1 - x1_mean + var_ratio * (x2 - x2_mean)
60     ))
61
62     return dw1
63
64     def get_params(w1):
65         w2 = w1 * var_ratio
66         b = -w1 * x1_mean - w2 * x2_mean
67         return w1, w2, b
68
69     return func, derivative, get_params

```

Listing 4: TODO: Complete the logistic regression objective

Hint**TODO 1:** Sigmoid formula: $1 / (1 + \text{np.exp}(-z))$ **TODO 2:** Linear combination: $w1 * x1 + w2 * x2 + b$ **TODO 3:** Apply sigmoid to z: `sigmoid(z)`**TODO 4:** Cross-entropy: $-\text{np.mean}(y * \text{np.log}(p) + (1 - y) * \text{np.log}(1 - p))$ **TODO 5:** Simple subtraction: $p - y$

Part IV: Experimentation & Discussion

Once your implementations are complete, run the GUI and experiment with different settings.

10 Experiment 1: Quadratic Regression

1. Generate a **Quadratic** dataset
2. Select **Quadratic OLS** model
3. Set $\alpha = 0.1$ and click “Run All”
4. Observe the 3D loss surface and the fitted parabola

Discussion Questions

Q1. Examine the 3D loss surface. Is it convex? What does this imply about the uniqueness of the optimal solution?

Q2. Try fitting a **Linear OLS** model to the Quadratic dataset. Compare the final loss values. What do you observe?

Q3. In the context of the volatility smile, why might we prefer a quadratic model even if the improvement in fit seems small?

11 Experiment 2: Model Mismatch

1. Generate a **Linear** dataset
2. Fit both **Linear OLS** and **Quadratic OLS** models
3. Compare the fitted curves and loss values

Discussion Questions

Q4. When fitting a quadratic model to linear data, what do you expect for the coefficient a (the x^2 term)?

Q5. Is there any danger in always using a more complex model (e.g., quadratic instead of linear)? Relate this to the concept of **overfitting**.

12 Experiment 3: Classification with Logistic Regression

1. Generate a **Two Clusters** dataset
2. Select **Logistic Regression** model
3. Set $\alpha = 0.5$ and click “Run Step” multiple times
4. Watch the decision boundary evolve

Discussion Questions

Q6. As gradient descent progresses, how does the decision boundary change? Does it stabilize?

Q7. What happens if you set α too high (e.g., $\alpha = 5$)? Explain in terms of the loss function.

Q8. Generate a **White Noise** dataset and try to fit a Logistic Regression model. What happens? What does this tell you about trying to classify random data?

13 Experiment 4: Financial Interpretation

Discussion Questions

Q9. In credit scoring, false negatives (predicting “no default” when the borrower defaults) and false positives (predicting “default” when the borrower repays) have different costs. How might you adjust the classification threshold (default is 0.5) to account for this asymmetry?

Q10. For the volatility smile fitting problem:

- What would happen if you used a higher-degree polynomial (cubic, quartic)?
- What are the trade-offs between model complexity and out-of-sample performance?

Q11. In a market regime classification problem, would you expect the decision boundary to remain stable over time? What implications does this have for model retraining frequency?

A Complete Solutions (For Instructors Only)

A.1 Quadratic OLS Objective

```

1 def create_quadratic_ols_objective(X, y):
2     n = len(X)
3     X2 = X**2
4     y_mean = np.mean(y)
5     x_mean = np.mean(X)
6     x2_mean = np.mean(X2)
7
8     def func(params):
9         a, b = params
10        c = y_mean - a * x2_mean - b * x_mean
11        residuals = y - (a * X2 + b * X + c)
12        return np.sum(residuals**2) / n
13
14    def gradient(params):
15        a, b = params
16        c = y_mean - a * x2_mean - b * x_mean
17        residuals = y - (a * X2 + b * X + c)
18        da = -2 * np.sum(residuals * (X2 - x2_mean)) / n
19        db = -2 * np.sum(residuals * (X - x_mean)) / n
20        return np.array([da, db])
21
22    def get_params(params):
23        a, b = params
24        c = y_mean - a * x2_mean - b * x_mean
25        return a, b, c
26
27    return func, gradient, get_params

```

Listing 5: Complete implementation

A.2 Logistic Regression Objective

```

1 def create_logistic_objective(X, y):
2     n = len(y)
3     x1, x2 = X[:, 0], X[:, 1]
4     x1_mean, x2_mean = np.mean(x1), np.mean(x2)
5     var_ratio = np.var(x1) / np.var(x2) if np.var(x2) > 1e-10 else 1.0
6
7     def sigmoid(z):
8         z = np.clip(z, -500, 500)
9         return 1 / (1 + np.exp(-z))
10
11    def func(w1):
12        w2 = w1 * var_ratio
13        b = -w1 * x1_mean - w2 * x2_mean
14        z = w1 * x1 + w2 * x2 + b
15        p = sigmoid(z)
16        eps = 1e-15
17        p = np.clip(p, eps, 1 - eps)
18        return -np.mean(y * np.log(p) + (1 - y) * np.log(1 - p))
19
20    def derivative(w1):
21        w2 = w1 * var_ratio

```

```
22     b = -w1 * x1_mean - w2 * x2_mean
23     p = sigmoid(w1 * x1 + w2 * x2 + b)
24     error = p - y
25     return np.mean(error * (x1 - x1_mean + var_ratio * (x2 - x2_mean
26 )))
27
28 def get_params(w1):
29     w2 = w1 * var_ratio
30     return w1, w2, -w1 * x1_mean - w2 * x2_mean
31
32 return func, derivative, get_params
```

Listing 6: Complete implementation