# Regression and Linear Classification

## Task 1 Predict a house price and measure its error

Setup 5 minutes + Task 1 12 minutes

Work from small calculations to fitted curves, then train a two-feature linear classifier. Use one script named **STUDENTID_CV_Lab6.py**. Complete the TODOs, do the requested calculations on paper first, and keep short explanations as **# Task N** comments. All datasets are supplied here.

**Timing:** setup 5; Tasks 1-4: 50; break 10; Tasks 5-8: 55. Total 120 minutes. You need NumPy and Matplotlib. House prices and image features are invented teaching data.

```
import numpy as np
import matplotlib.pyplot as plt
from numpy.polynomial import Polynomial
def mse(prediction, target):
    return np.mean((prediction - target) ** 2)
```

### 1. Define the input, target and three candidate rules

The input is house area. Define x = area / 20 so that a 40-square-metre home has x = 2. The target y is price in **thousand USD**: 84 means USD 84,000. Each row is one different training house.

| Area in square metres | Scaled x | Observed price y |
|---|---|---|
| 40 | 2 | 84 |
| 60 | 3 | 92 |
| 80 | 4 | 128 |
| 100 | 5 | 144 |
| 120 | 6 | 188 |

**Constant:** y_hat = 126.
**Linear:** y_hat = 26x + 22.
**Quadratic:** y_hat = 50 + 10x + 2x^2. The x^2 term lets the graph bend. These are initially supplied rules; you will fit their parameters in Task 2.

### 2. Complete an error table for each rule

For every house, calculate y_hat, error e = y_hat - y and e^2. Then average the five squared errors. Keep the errors signed until you square them.

| Worked row | Prediction | Error | Squared error |
|---|---|---|---|
| Quadratic at x = 2 | 50 + 20 + 8 = 78 | 78 - 84 = -6 | 36 |

**Check:** linear MSE = 66.4; quadratic MSE = 36. The units are (thousand USD)^2. A negative error means underprediction. Squaring makes both error directions contribute positively.

**Show the TA:** all three error tables and MSEs. Which rule fits these observed prices best so far? Why is checking only one house insufficient?

Computer Vision, NU

# Task 2 Fit models from simple to more complex

10 minutes | Constant, linear, quadratic and fourth-degree curves

Now estimate parameters from the same five training houses. A degree-0 polynomial is constant; degree 1 is a straight line; degree 2 adds x^2; degree 4 also allows x^3 and x^4. More degrees mean more adjustable coefficients.

```
area = np.array([40.,60.,80.,100.,120.])
x = area / 20
y = np.array([84.,92.,128.,144.,188.])
models = {}
for degree in [0,1,2,4]:
    models[degree] = Polynomial.fit(x, y, degree)
    prediction = models[degree](x)
    # TODO: print degree, predictions and mse(prediction, y)
    # TODO: print models[degree].convert().coef
```

**Supplied fitting tool:** Polynomial.fit chooses coefficients to minimize the training sum of squared errors. For a fixed number of rows, minimizing that sum also minimizes MSE. You do not need to implement this solver. The converted coefficient array is ordered [b, w1, w2, ...].

## 1. Write the fitted linear and quadratic equations

Use the printed coefficients to write y_hat = b + w1*x and y_hat = b + w1*x + w2*x^2. Compare them with the supplied rules on page 1. Explain why they differ: the observed prices do not lie exactly on the supplied quadratic.

## 2. Evaluate on houses that were not used for fitting

| Validation area | Scaled x | Observed validation price |
|---|---|---|
| 50 | 2.5 | 87.5 |
| 70 | 3.5 | 109.5 |
| 90 | 4.5 | 135.5 |
| 110 | 5.5 | 165.5 |

```
xval = np.array([50.,70.,90.,110.]) / 20
yval = np.array([87.5,109.5,135.5,165.5])
for degree, model in models.items():
    # TODO: predict model(xval)
    # TODO: print training MSE and validation MSE
    pass
```

Keep the validation houses out of Polynomial.fit. Use them only to compare the already fitted choices. For this constructed example the validation values follow a smooth relation, while the training observations include small price deviations; this makes the effect of fitting those deviations visible.

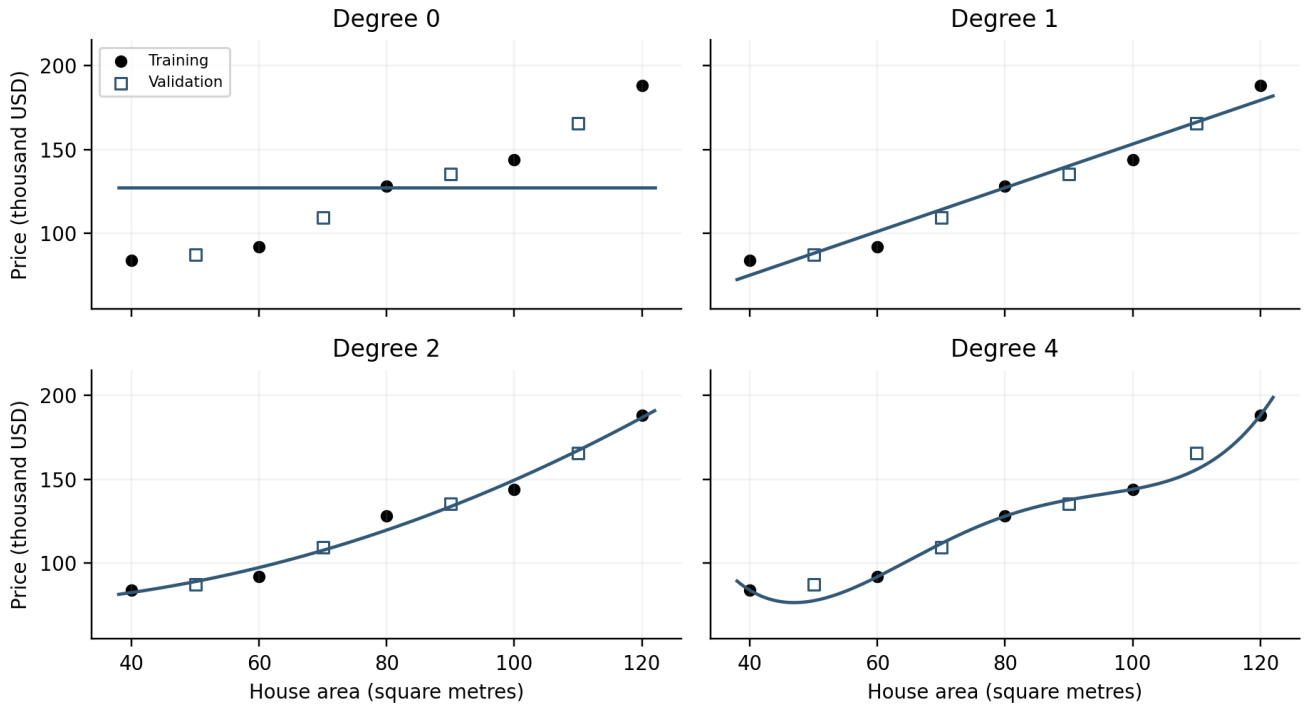
**Check:** the fitted linear model is approximately 23.2 + 26x. A fourth-degree polynomial can pass through all five training points. A training MSE near zero does not yet tell you its error on the validation houses.

**Show the TA:** the two fitted equations, the parameter counts (degree + 1), and all training/validation MSEs.

# Task 3 Decide which curve fits usefully

13 minutes | Underfitting, useful fit and overfitting

The four panels use identical axes and the same houses. Black circles are training prices; outlined squares are validation prices. Notice both the broad trend and how each fitted curve behaves between training houses.




```
grid_area = np.linspace(40,120,300)
for degree, model in models.items():
    plt.plot(grid_area, model(grid_area/20), label=f"degree {degree}")
plt.scatter(area, y, color="black", label="training")
plt.scatter(xval*20, yval, marker="s", label="validation")
# TODO: label area and price units, add a legend, and show the plot
```

| Degree | Training MSE | Validation MSE | Your interpretation |
|---|---|---|---|
| 0 | ___ | ___ | ___ |
| 1 | ___ | ___ | ___ |
| 2 | ___ | ___ | ___ |
| 4 | ___ | ___ | ___ |

**Check:** the fitted quadratic has training MSE about 26.3314 and validation MSE about 2.9461. The degree-4 curve has essentially zero training error but validation MSE about 50.0625. Calculate the other rows yourself.

**Interpret:** the constant underfits the strong area-price trend. A line captures the rise but misses some curvature. The quadratic gives the lowest validation error here. The degree-4 model follows the training deviations too closely: lower training error is accompanied by worse validation error, evidence of overfitting in this example.

**Show the TA:** your table and selected degree. Explain why a more complicated curve is not always better, and why the validation error of one small synthetic set is not a guarantee for real houses. Do not conclude that degree 2 is always best.

# Task 4 Learn a straight line by gradient descent

15 minutes, then a 10-minute break | Update both w and b

Use the same houses, with new scaling for this optimization task: **u = area / 100** and **v = y / 100**, where y is price in thousand USD. Thus v = 0.84 represents USD 84,000. The model is v_hat = w*u + b. Keep this scaling separate from x = area/20 used in Tasks 1-3.

**Mean loss:** J = mean(e^2), where e = w*u + b - v.
**Gradients:** dw = 2*mean(e*u); db = 2*mean(e).
**Update:** w_new = w - eta*dw; b_new = b - eta*db. Compute both gradients using the same old w and b.

## 1. Calculate the first update by hand

Start at w = 1, b = 0 and eta = 0.1. Use the five u values [0.4, 0.6, 0.8, 1.0, 1.2] and v values [0.84, 0.92, 1.28, 1.44, 1.88]. Calculate each prediction and error, then dw and db.

| Quantity | Worked result |
|---|---|
| Initial MSE | 0.23648 |
| dw and db | -0.8032 and -0.944 |
| w_new and b_new | 1.08032 and 0.0944 |
| New MSE | Calculate it, then compare with about 0.10854 |

## 2. Complete and run the learning loop

```
u = area / 100
v = y / 100
def fit_line(eta, steps):
    w, b = 1.0, 0.0
    history = []
    for step in range(steps + 1):
        e = w*u + b - v
        history.append(np.mean(e**2))
        if step == steps:
            break
        # TODO: calculate dw and db from this same e
        # TODO: update w and b by subtracting eta * gradient
    return w, b, np.array(history)
# TODO: call fit_line(0.1, 500), print parameters, and plot history
```

**Check:** after 500 updates, w is about 1.2995 and b about 0.2324. In the original price units this is close to y_hat = 26x + 23.2 from Task 2. Dividing price by 100 also divides its MSE by 10,000; compare losses only after accounting for units.

**Scenario:** restart from w = 1, b = 0 and compare eta = 0.01, 0.1 and 1.0 for ten updates each. Report the final MSE. Which rate is slow, which makes useful progress and which overshoots? These numerical rates depend on scaling.

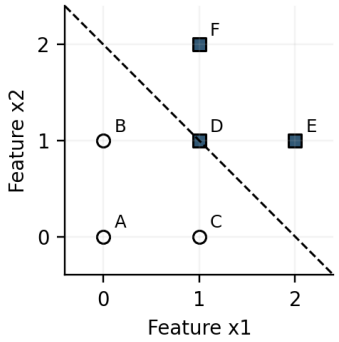
**Show the TA:** the first update, the learned line, the MSE curve and the rate comparison. Take a 10-minute break.

# Task 5 Use a straight boundary for two classes

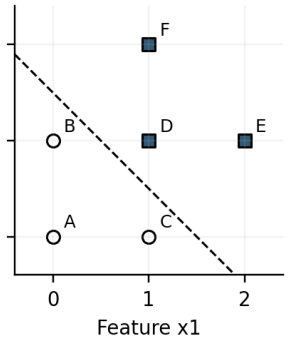
12 minutes | Two input features, one linear score

Suppose each image has two numerical feature measurements, x1 and x2. The target is class 0 or 1. The six examples are printed in the code below, in order A-F. Circles mark true class 0; squares mark true class 1. The figures preview the candidate boundaries you will calculate.

Model B: boundary x1 + x2 = 2




Model C: boundary x1 + x2 = 1.5




**Linear score:** z = w1*x1 + w2*x2 + b.
**Class decision:** predict 1 if z >= 0.5, otherwise 0.
**Training error:** J = mean((z - y)^2). This section fits the numeric 0/1 targets directly. z is a **score**, not a probability; it may fall below 0 or above 1.

## 1. Compare three manually chosen classifiers

| Model | w1 | w2 | b | MSE | Accuracy |
|---|---|---|---|---|---|
| A | 0 | 0 | 0 | ___ | ___ |
| B | 0.25 | 0.25 | 0 | ___ | ___ |
| C | 0.5 | 0.5 | -0.25 | ___ | ___ |

```
# Rows A, B, C, D, E, F in this order:
X = np.array([[0,0],[0,1],[1,0],[1,1],[2,1],[1,2]], dtype=float)
yclass = np.array([0.,0.,0.,1.,1.,1.])
w = np.array([0.25,0.25])
b = 0.0
z = X @ w + b
# TODO: compute MSE using z and yclass
# TODO: predicted = (z >= 0.5).astype(int)
# TODO: compute accuracy and repeat for the three models
```

**Worked row:** Model B gives z = 0.25*1 + 0.25*1 = 0.5 for D. Its class prediction is 1, while its squared error is (0.5 - 1)^2 = 0.25. Correct classification does not require zero squared error.

**Check:** Models B and C both classify all six examples correctly, but their MSEs are about 0.08333 and 0.0625. Model A predicts only class 0 and has MSE 0.5.

**Show the TA:** your per-example scores, completed table and one explanation of the difference between classification accuracy and the numeric training error.

# Task 6 Update both weights and the bias with MSE

15 minutes | Gradient descent for the 2D linear classifier

Use X and yclass from Task 5. Let N = 6 and e = X @ w + b - yclass. The mean squared error gradients are **dw = (2/N)*X.T @ e** and **db = 2*mean(e)**. dw contains one derivative per feature.

## 1. Calculate the first update from the zero model

Start with w = [0, 0], b = 0 and eta = 0.1. All six scores are zero. Complete the two contribution columns before summing them.

| Example | Error e | e*x1 | e*x2 |
|---|---|---|---|
| A | 0 | 0 | 0 |
| B | 0 | ___ | ___ |
| C | 0 | ___ | ___ |
| D | -1 | ___ | ___ |
| E | -1 | ___ | ___ |
| F | -1 | ___ | ___ |
| Sum | -3 | ___ | ___ |

**Check:** each feature-contribution sum is -4. Multiply by 2/6 to get dw = [-1.3333, -1.3333]. The bias gradient is -1. After one update, w = [0.1333, 0.1333] and b = 0.1. Recalculate all six scores: the new MSE is 0.17.

## 2. Complete the learning function

```
def train_classifier(X, y, eta=0.1, steps=300):
    w = np.zeros(X.shape[1])
    b = 0.0
    history = []
    for step in range(steps + 1):
        z = X @ w + b
        e = z - y
        history.append(np.mean(e**2))
        if step == steps:
            break
        # TODO: dw = 2 * (X.T @ e) / len(y)
        # TODO: db = 2 * e.mean()
        # TODO: update w and b using these gradients
    return w, b, np.array(history)
# TODO: train on X, yclass; print w, b and the final MSE
```

**Explain:** why is the bias gradient not multiplied by x1 or x2? Why do negative gradients increase a parameter under the subtraction rule? Why must both weight gradients and db be computed before changing any parameter?

**Show the TA:** the hand calculation, the first updated scores and your completed gradient/update lines. The factor 2 comes from differentiating e^2; the division by N comes from the mean.

Computer Vision · NUU

# Task 7 Draw the boundary and examine model limits

13 minutes | How w and b change a classifier

Run the completed function for 300 updates at eta = 0.1. Plot the MSE history. At initialization, after one update and after 300 updates, report w1, w2, b, MSE and classification accuracy using the fixed 0.5 threshold.

**Check:** the final weights are about [0.4091, 0.4091], b about -0.1818 and MSE about 0.04545. All six examples are classified correctly. MSE need not reach zero even when accuracy is 100%.

## 1. Plot the line where the predicted class changes

The decision boundary is w1*x1 + w2*x2 + b = 0.5. If w2 is nonzero, solve for x2 = (0.5 - b - w1*x1)/w2. Points on opposite sides receive different class predictions.

```
def draw_boundary(X, y, w, b, title):
    for label, marker in [(0,"o"),(1,"s")]:
        rows = (y == label)
        plt.scatter(X[rows,0], X[rows,1], marker=marker,
                    label=f"class {label}", edgecolors="black")
    grid = np.linspace(-0.5,2.5,100)
    if abs(w[1]) > 1e-10:
        plt.plot(grid, (0.5-b-w[0]*grid)/w[1], color="black")
    elif abs(w[0]) > 1e-10:
        plt.axvline((0.5-b)/w[0], color="black")
    plt.xlabel("Feature x1"); plt.ylabel("Feature x2")
    plt.xlim(-0.5,2.5); plt.ylim(-0.5,2.5)
    plt.title(title); plt.legend(); plt.show()
# TODO: draw the trained boundary using the returned w and b
```

Compare three separate choices using fresh values each time:
**I:** w = [0.5, 0.5], b = -0.25.
**II:** keep those weights but change b to +0.25.
**III:** w = [0.5, 0], b = 0.
For each, draw the line, calculate all six scores, MSE and accuracy. Which change shifts a parallel boundary? Which changes its orientation?

## 2. Recognize an underfitting limitation

New scenario: Xxor = [[0,0],[0,1],[1,0],[1,1]] and yxor = [0,1,1,0]. Sketch the labelled points. Can one straight line separate the two diagonal pairs?

**Worked check:** w = [0,0], b = 0.5 gives four scores of 0.5 and MSE 0.25. Both dw and db are zero, yet this does not solve the classification task. Verify the gradients. A more suitable model would be needed to represent the pattern; extra iterations alone cannot make a straight boundary separate it.

**Show the TA:** the learning curve, the three boundary trials and the XOR sketch. Distinguish failure to optimize from a model that cannot represent the required pattern.

# Task 8 Why use the negative log of probability?

15 minutes | A final loss comparison with a sigmoid output

This is a **separate final experiment**. Convert a single score z into a probability p = 1/(1 + exp(-z)). The observed label is y = 1, so p is also the probability of the true class. Compare probability MSE = (p - 1)^2 with cross-entropy = -ln(p). For y = 0 the cross-entropy would be -ln(1 - p).

| True-class p | MSE (p - 1)^2 | Cross-entropy -ln(p) |
|---|---|---|
| 0.90 | 0.01 | 0.1054 |
| 0.50 | ___ | ___ |
| 0.10 | ___ | ___ |
| 0.01 | ___ | ___ |

Complete the table with a calculator. When a prediction becomes confidently wrong, probability MSE approaches 1, whereas -ln(p) grows without a finite bound. A larger penalty alone does not explain learning speed; examine the gradients too.

## 1. Calculate one update for the same wrong starting score

Use z = -5, p about 0.006693, y = 1 and eta = 1. Treat z as one adjustable bias. The derivatives with respect to z are:
**Probability MSE:** 2*(p - 1)*p*(1 - p), about -0.013207.
**Cross-entropy:** p - 1, about -0.993307.
Calculate z_new = z - eta*gradient for each loss, then recalculate sigmoid(z_new).

**Check:** after one update, probability MSE gives p about 0.006781; cross-entropy gives p about 0.017868. The extra sigmoid factor p*(1 - p) makes the MSE gradient small at this wrong saturated output. These derivatives differ from the raw-score MSE used in Tasks 5-7.

## 2. Repeat the calculation as an experiment

```
for loss_name in ["MSE", "cross-entropy"]:
    z = -5.0                      # restart identically
    for step in range(21):
        p = 1 / (1 + np.exp(-z))
        if step in [0,1,5,20]:
            print(loss_name, step, z, p)
        if step == 20:
            break
        if loss_name == "MSE":
            # TODO: gradient = 2*(p-1)*p*(1-p)
            pass
        else:
            # TODO: gradient = p-1
            pass
        # TODO: z = z - gradient   (eta = 1)
```

**Check:** after 20 updates, p is about 0.00904 for MSE and 0.93226 for cross-entropy. Compare the same output probability, not the absolute sizes of two differently defined losses. Explain which loss corrects this confident mistake faster and why.

**Conclude:** cross-entropy often makes sigmoid classification easier to optimize because of this gradient behaviour. MSE remains useful for numerical regression and is also a valid probability score. This single experiment does not prove that cross-entropy wins in every setting.

**Final demonstration:** show the house-price errors and fitted curves, the underfitting/overfitting comparison, your 2D gradients and boundaries, and the final loss experiment. Keep the script and figures for revision.

tel Vision - NUU