Unit 5 Syllabus

Multiple Regression, The Model, Further Assumptions of the Least Squares Model, Fitting the Model, Interpreting the Model, Goodness of Fit, Digression: The Bootstrap, Standard Errors of Regression Coefficients, Regularization, Logistic Regression, The Problem, The Logistic Function, Applying the Model, Goodness of Fit, Support Vector Machines.

Multiple Regression

- Although the VP is pretty impressed with your predictive model, she thinks you can do better. To that end, you’ve collected **additional data: you know how many hours each of your users works each day, and whether they have a PhD.**
- You’d like to use this **additional data to improve your model**.
- $\text{minutes} = \alpha + \beta_1\text{friends} + \beta_2\text{work hours} + \beta_3\text{phd} + \varepsilon$

The Model

- Recall that we fit a model of the form:

$$y_i = \alpha + \beta x_i + \varepsilon_i$$

- Now imagine that each **input xi** is not a single number but rather a vector of k numbers, xi1, ..., xik.
- **The multiple regression model assumes that:**

$$y_i = \alpha + \beta_1 x_{i1} + \ldots + \beta_k x_{ik} + \varepsilon_i$$

- In multiple regression the **vector of parameters is called β**.

```
beta = [alpha, beta_1, ..., beta_k]
```

- Include the constant term as well, which we can achieve by adding a column of 1s to our data:
- `x_i = [1, x_i1, ..., x_ik]`

Further Assumptions of the Least Squares Model

Assumption 1: No multicollinearity

Inputs $x_1, x_2, \ldots, x_k$ must be **linearly independent**

If one column is a combination of others (e.g., `num_acquaintances = num_friends`), you can't uniquely determine the coefficients

Assumption 2: No correlation between inputs and error terms

If an input (say, `friends`) is correlated with an omitted variable (like `work_hours`) that affects y, the model's estimate will be **biased**

For example:

$$\text{Actual model: } y = \alpha + \beta_1 \text{friends} + \beta_2 \text{work hours} + \epsilon$$

But you forget to include `work hours`. If friends and work hours are correlated, then your estimate of β_1 will absorb some of β_2's effect — **biasing the result.**

Fitting the Model: Gradient Descent Approach

Gradient Descent is used in multiple linear regression primarily as a way to efficiently find the optimal model parameters (the weights or coefficients) when dealing with many input variables (features).

```
def predict(x: Vector, beta: Vector) -> float:
    return dot(x, beta)
def error(x, y, beta):
    return predict(x, beta) - y
def sqerror_gradient(x, y, beta):
    err = error(x, y, beta)
    return [2 * err * x_i for x_i in x]
def gradient_step(v: List[float], gradient: List[float], step_size: float) -
> List[float]:
    return [v_i + step_size * grad_i for v_i, grad_i in zip(v, gradient)]
```

Fitting the Model: Gradient Descent Approach

```
def least_squares_fit(xs, ys, learning_rate=0.001, num_steps=1000, batch_size=1):
    guess = [random.random() for _ in xs[0]]
    for _ in range(num_steps):
        for start in range(0, len(xs), batch_size):
            batch_xs = xs[start:start+batch_size]
            batch_ys = ys[start:start+batch_size]
            gradient = vector_mean([sqerror_gradient(x, y, guess)
                for x, y in zip(batch_xs, batch_ys) ])
            guess = gradient_step(guess, gradient, -learning_rate)
    return guess
```

Final Result: Interpreting Coefficients

After running the gradient descent (or using a closed-form solution), you get:

$$\text{minutes} = 30.58 + 0.972\ \text{friends} - 1.87\ \text{work hours} + 0.923\ \text{phd}$$

Interpretations:

- **Intercept (30.58)**: baseline minutes if all inputs are 0
- **0.972 for friends**: Each additional friend adds ~0.97 daily minutes
- **-1.87 for work hours**: More work → less time on site
- **0.923 for PhD (binary)**: Having a PhD increases time spent by ~0.92 minutes

These are **marginal effects** — holding other variables constant.

Goodness of Fit

- R-squared (R^2) measures how well the model explains the variability in the response variable.
- Formula:

$$R^2 = 1 - \frac{\text{Sum of Squared Errors (SSE)}}{\text{Total Sum of Squares (TSS)}}$$

- SSE: Total difference between predicted and actual values.
- TSS: Total variance in the actual values (how far they are from the mean).

```
from scratch.simple_linear_regression import total_sum_of_squares
def multiple_r_squared(xs: List[Vector], ys: Vector, beta: Vector) -> float:
    sum_of_squared_errors = sum(error(x, y, beta) ** 2
    for x, y in zip(xs, ys))
    return 1.0 - sum_of_squared_errors / total_sum_of_squares(ys)
```

which has now increased to 0.68:

```
assert 0.67 < multiple_r_squared(inputs, daily_minutes_good, beta) < 0.68
```

Goodness of Fit

- Adding new variables to a regression will necessarily increase the R-squared. As the simple regression model is just the special case of the multiple regression model where the coefficients on "work hours" and "PhD" both equal 0.
- Any extra variable gives the model **more flexibility**, even if the variable is **irrelevant**.
- This reduces SSE, and since R^2 is inversely related to SSE, **R^2 never decreases** when new variables are added.
- But a **higher R^2 doesn't always mean a better model**, especially if the new variables are noise.

Regularization

What is Regularization?

When we use **multiple linear regression** on datasets with **many features**, we run into two problems:

1. Overfitting

•The model fits the training data **too well**, even capturing noise.

•It performs poorly on new, unseen data (**poor generalization**).

2. Interpretability

If many features have non-zero coefficients, it's hard to **understand** or **explain** the model.

Solution: Regularization

- Add a **penalty** to the cost function (which we normally minimize)

Regularization

Regularized Cost Function

Original Linear Regression Error:

$$y_i = \alpha + \beta_1 x_{i1} + \ldots + \beta_k x_{ik} + \varepsilon_i$$

$$\text{Loss} = \sum (y_i - \hat{y}_i)^2 = \sum (y_i - \beta_0 - \beta_1 x_{i1} - \cdots - \beta_n x_{in})^2$$

Regularized Loss:

Regularized Loss

Regularized Loss=Error Term + Penalty Term

Regularization

Types of Regularization

1. Ridge Regression (L2 Regularization)

Adds a **penalty proportional to the square** of the coefficients:

$$\text{Penalty} = \alpha \sum_{j=1}^{n} \beta_j^2$$

- Does not force coefficients to zero.
- Good when all features are useful but need to reduce magnitude of coefficients.

2. Lasso Regression (L1 Regularization)

Adds a **penalty proportional to the absolute values** of the coefficients:

$$\text{Penalty} = \alpha \sum_{j=1}^{n} |\beta_j|$$

- Can force some coefficients to exactly zero → gives sparse models.
- Useful for feature selection.

Regularization

```
def ridge_penalty(beta: Vector, alpha: float) -> float:
        return alpha * dot(beta[1:], beta[1:])
def squared_error_ridge(x: Vector, y: float, beta: Vector,alpha: float) -> float:
    """estimate error plus ridge penalty on beta"""
    return error(x, y, beta) ** 2 + ridge_penalty(beta, alpha)
```

We can then plug this into gradient descent in the usual way:

```
from scratch.linear_algebra import add
def ridge_penalty_gradient(beta: Vector, alpha: float) -> Vector:
    """gradient of just the ridge penalty"""
    return [0.] + [2 * alpha * beta_j for beta_j in beta[1:]]
def sqerror_ridge_gradient(x: Vector, y: float,beta: Vector,alpha: float) -> Vector:
    """
    the gradient corresponding to the ith squared error term
    including the ridge penalty
    """
    return add(sqerror_gradient(x, y, beta), ridge_penalty_gradient(beta, alpha))
```

Regularization

lasso regression, which uses the penalty:

```
def lasso_penalty(beta, alpha):
        return alpha * sum(abs(beta_i) for beta_i in beta[1:])
```

Regularization

🧪 Effects of Ridge Regularization

Alpha (Penalty Strength)	Coefficients (β)	R^2 (Goodness of Fit)	Interpretation
0.0	[30.51, 0.97, -1.85, 0.91]	~0.68	No regularization
0.1	[30.8, 0.95, -1.83, 0.54]	~0.68	Small shrinkage
1.0	[30.6, 0.90, -1.68, 0.10]	~0.68	"PhD" nearly 0
10.0	[28.3, 0.67, -0.90, -0.01]	~0.55	Strong shrinkage

Why Ridge Helps

- Reduces overfitting by penalizing large weights.
- Controls complexity of the model.
- Especially useful when features are correlated or you have more features than data points.

Logistic Regression

- Data set of about 200 users, containing each user's salary, years of experience as a data scientist, and whether paid for a premium account.
- Represent the dependent variable as either 0 (no premium account) or 1 (premium account).

- A dataset: Each row = `[experience, salary, paid_account]`
  - `experience` = number of years as a data scientist
  - `salary` = user's salary
  - `paid_account` = `1` if paid, `0` otherwise

You're trying to predict `paid_account` from the other two features.

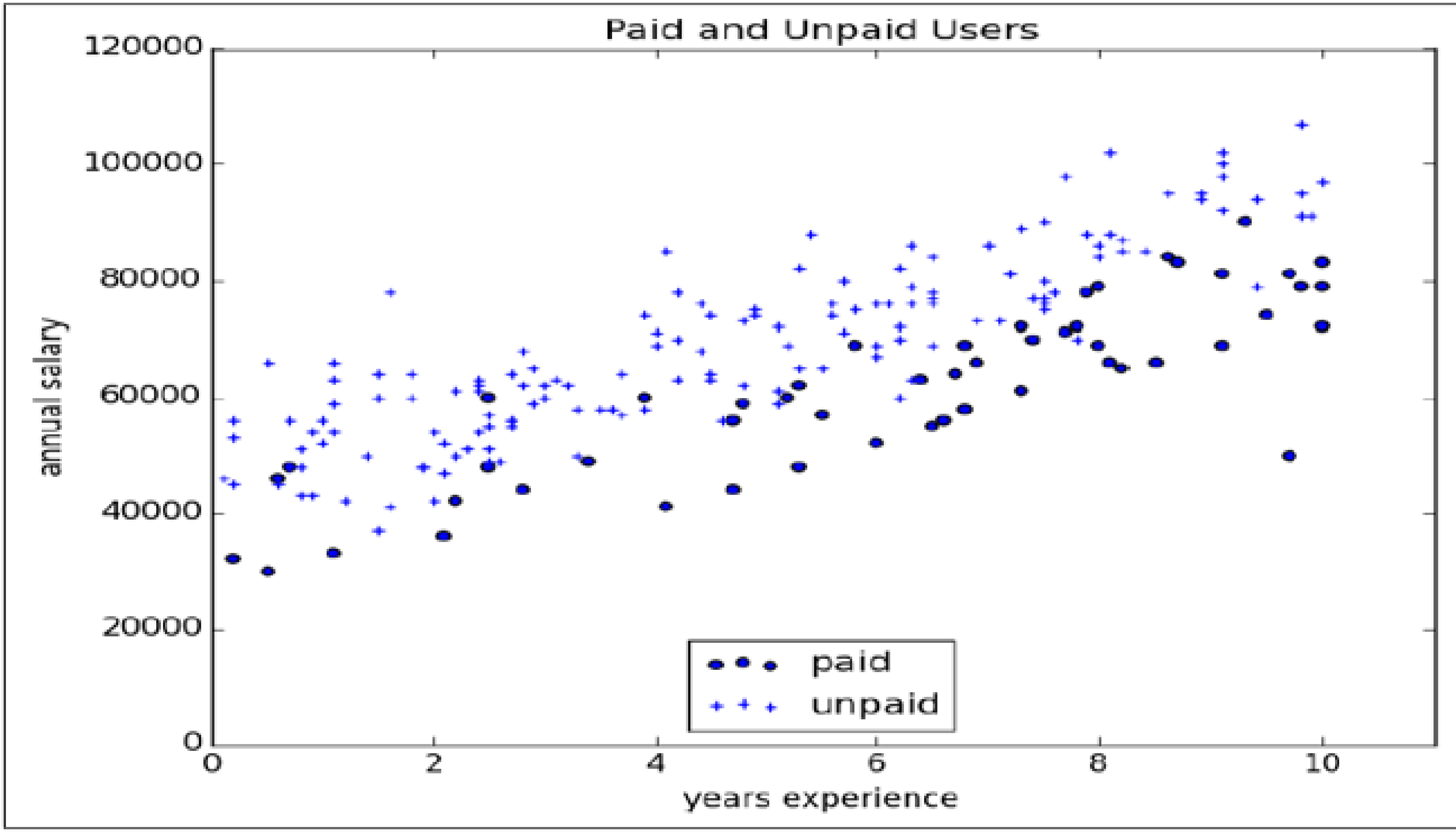


Figure 16-1. Paid and unpaid users

Logistic Regression

First Attempt: Linear Regression:

$$\text{paid\_account} = \beta_0 + \beta_1 \cdot \text{experience} + \beta_2 \cdot \text{salary} + \varepsilon$$

```
xs = [[1.0] + row[:2] for row in data]  # adds intercept term, makes each row [1, experience, salary]
ys = [row[2] for row in data]          # gets 0 or 1 label
```

Program to apply the logistic function to a prediction from the linear model. Show how you convert this linear prediction to a probability

```
from matplotlib import pyplot as plt
from scratch.working_with_data import rescale
from scratch.multiple_regression import least_squares_fit, predict
from scratch.gradient_descent import gradient_step
learning_rate = 0.001
rescaled_xs = rescale(xs)
beta = least_squares_fit(rescaled_xs, ys, learning_rate, 1000, 1)
# [0.26, 0.43, -0.43]
predictions = [predict(x_i, beta) for x_i in rescaled_xs]
plt.scatter(predictions, ys)
plt.xlabel("predicted")
plt.ylabel("actual")
plt.show()
```

Logistic Regression

Predictions Are Not Probabilities

- Linear regression gives predictions like -3, 1.5, 20, etc.
- These don't make sense when predicting a binary outcome (0 or 1):

Linear Regression on Binary Labels:

- Tries to draw a straight line through the 0s and 1s.
- Doesn't respect the idea that outputs should be between 0 and 1.

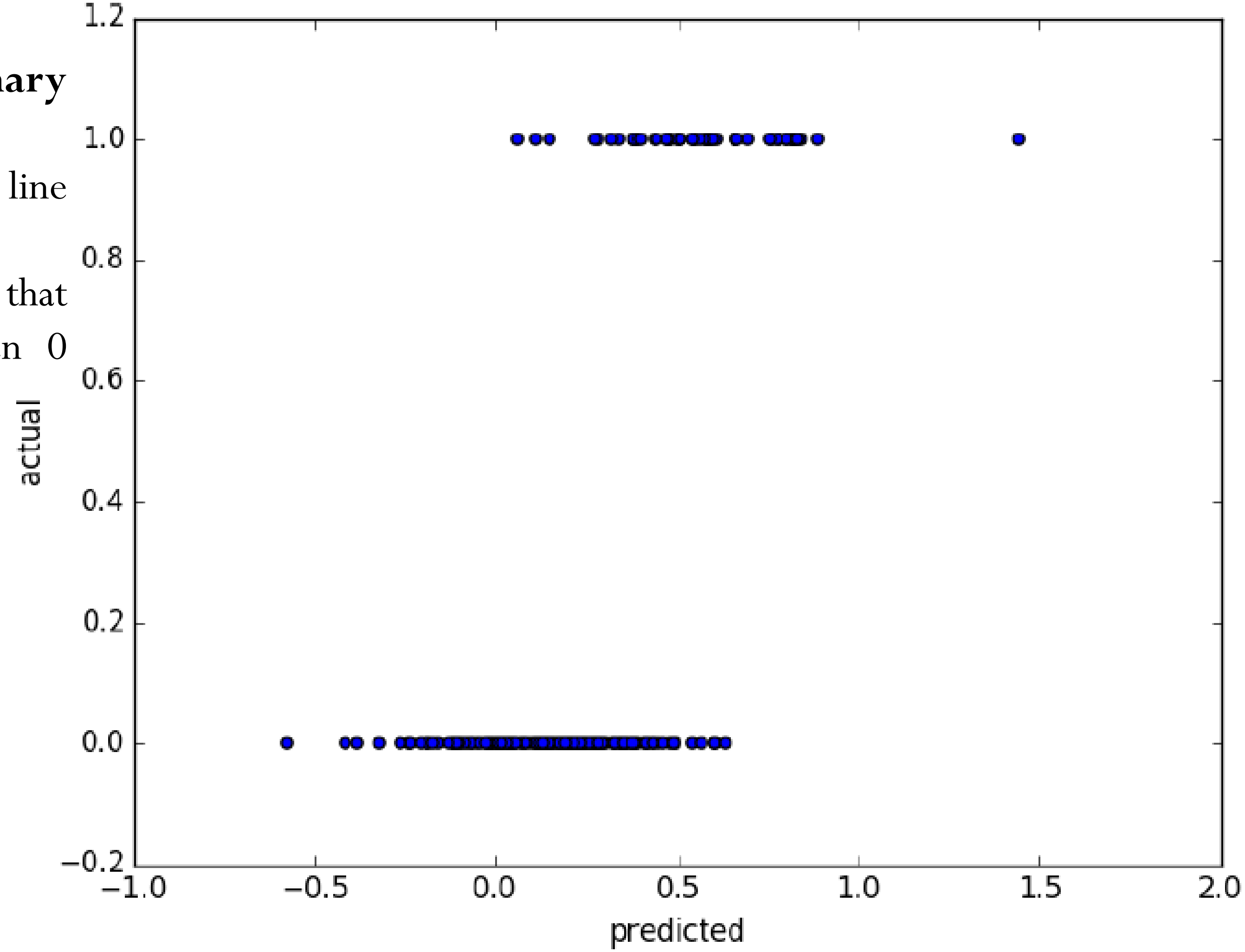


Figure 16-2. Using linear regression to predict premium accounts

What is Logistic Regression?

What is Logistic Regression?

- It's a classification algorithm, typically used when the output variable (y) is binary (e.g., 0 or 1).
- Instead of predicting a continuous value (like in linear regression), it predicts the probability that y = 1.
- The model looks like:

$$y_i = f(x_i \cdot \beta) + \varepsilon_i$$

Where:

- f is the logistic function(Dot Product)
- β are the model parameters
- xi is a vector of input features
- yi is the binary output (0 or 1)

The Logistic Regression Code Snippet

- For large positive values of dot(x_i, beta) to correspond to probabilities close to 1, and for large negative values to correspond to probabilities close to 0. Use **Sigmoid Function**

```
def logistic(x: float) -> float:
        return 1.0 / (1 + math.exp(-x))
```

This function: Converts any real number into the rar

- Large positive inputs → output near 1.
- Large negative inputs → output near 0.

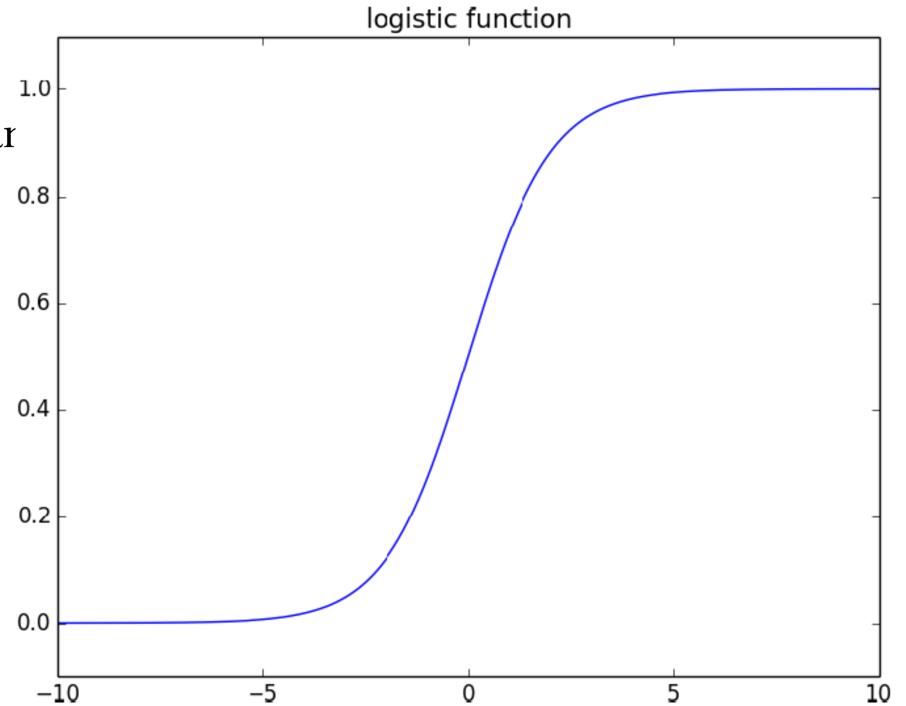


Figure 16-3. The logistic function

The Logistic Regression Code Snippet

- Its derivative is useful for gradient descent:

```
def logistic_prime(x: float) -> float:
    y = logistic(x)
    return y * (1 - y)
```

- Recall that for linear regression we fit the model by **minimizing the sum of squared errors**, which ended up choosing the β that maximized the likelihood of the data.
- In Logistic Regression the two aren't equivalent, so we'll **use gradient descent to maximize the likelihood directly.** This means we need to calculate the likelihood function and its gradient.

The Logistic Function

Given some β, our model says that each y_i should equal 1 with probability $f(x_i\beta)$ and 0 with probability $1 - f(x_i\beta)$.

In particular, the PDF for y_i can be written as:

$$p(y_i|x_i, \beta) = f(x_i\beta)^{y_i}(1 - f(x_i\beta))^{1-y_i}$$

since if y_i is 0, this equals:

$$1 - f(x_i\beta)$$

and if y_i is 1, it equals:

$$f(x_i\beta)$$

It turns out that it's actually simpler to maximize the *log likelihood*:

$$\log L(\beta|x_i, y_i) = y_i \log f(x_i\beta) + (1 - y_i) \log (1 - f(x_i\beta))$$

The Logistic Regression using gradient descent

- **Log is a strictly increasing function**, any beta that **maximizes the log likelihood** also maximizes the likelihood, and vice versa.
- But **gradient descent minimizes** things, so work with the **negative log likelihood**, since maximizing the likelihood is the same as minimizing its negative:

Code snippet to show how gradient descent updates the coefficients in logistic regression using the negative log likelihood gradient

```
import math
from scratch.linear_algebra import Vector, dot
def _negative_log_likelihood(x: Vector, y: float, beta: Vector) -> float:
    """The negative log likelihood for one data point"""
    if y == 1:
        return -math.log(logistic(dot(x, beta)))
    else:
        return -math.log(1 - logistic(dot(x, beta)))
```

Overall log likelihood is the sum of the individual log likelihoods:

```
from typing import List
def negative_log_likelihood(xs: List[Vector], ys: List[float], beta: Vector) -> float:
  return sum(_negative_log_likelihood(x, y, beta) for x, y in zip(xs, ys))
```

The Logistic Regression Code Snippet

We calculate the gradient of the loss function with respect to each β_j:

$$\frac{\partial}{\partial \beta_j} = -(y_i - f(x_i \cdot \beta)) \cdot x_{ij}$$

```
from scratch.linear_algebra import vector_sum
def _negative_log_partial_j(x: Vector, y: float, beta: Vector, j: int) -> float:
    """“ The jth partial derivative for one data point. Here i is the index of the data
    point. """"
    return -(y - logistic(dot(x, beta))) * x[j]

def _negative_log_gradient(x: Vector, y: float, beta: Vector) -> Vector:
    """“ The gradient for one data point. """"
    return [_negative_log_partial_j(x, y, beta, j) for j in range(len(beta))]

def negative_log_gradient(xs: List[Vector],ys: List[float],beta: Vector) -> Vector:
    return vector_sum([_negative_log_gradient(x, y, beta) for x, y in zip(xs, ys)])
```

The Applying Logistic Function

```
from scratch.machine_learning import train_test_split
import random
import tqdm
random.seed(0)
x_train, x_test, y_train, y_test = train_test_split(rescaled_xs, ys, 0.33)
learning_rate = 0.01
# pick a random starting point
beta = [random.random() for _ in range(3)]
with tqdm.trange(5000) as t:
for epoch in t:
    gradient = negative_log_gradient(x_train, y_train, beta)
    beta = gradient_step(beta, gradient, -learning_rate)
    loss = negative_log_likelihood(x_train, y_train, beta)
    t.set_description(f"loss: {loss:.3f} beta: {beta}")
```

1. `negative_log_gradient(...)` computes the gradient of the loss w.r.t. `beta`.
2. `gradient_step(...)` updates `beta` using gradient descent:
   $\beta = \beta - \alpha \cdot \nabla L$
3. `negative_log_likelihood(...)` computes the current loss.
4. `t.set_description(...)` updates the progress bar with the current loss and beta.

after which we find that beta is approximately:

[-2.0, 4.7, -4.5]

Support Vector Machine

What Is an SVM?

- Support Vector Machine (SVM) is a **supervised machine learning algorithm** used for **classification** and sometimes regression. It works by finding a **hyperplane** (decision boundary) that best separates the data into different classes.

Given two classes of data:

- SVM tries to find the **widest possible margin** (i.e., maximum distance) between the **hyperplane** and the **nearest points** of both classes.
- These nearest points are called **support vectors**.
- The larger the margin, the **better the generalization** of the classifier.

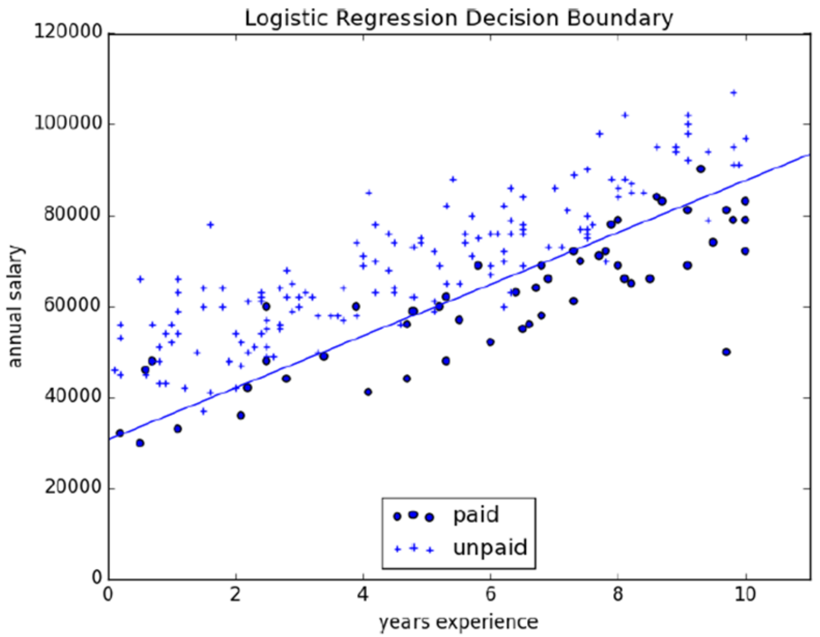


Figure 16-5. Paid and unpaid users with decision boundary

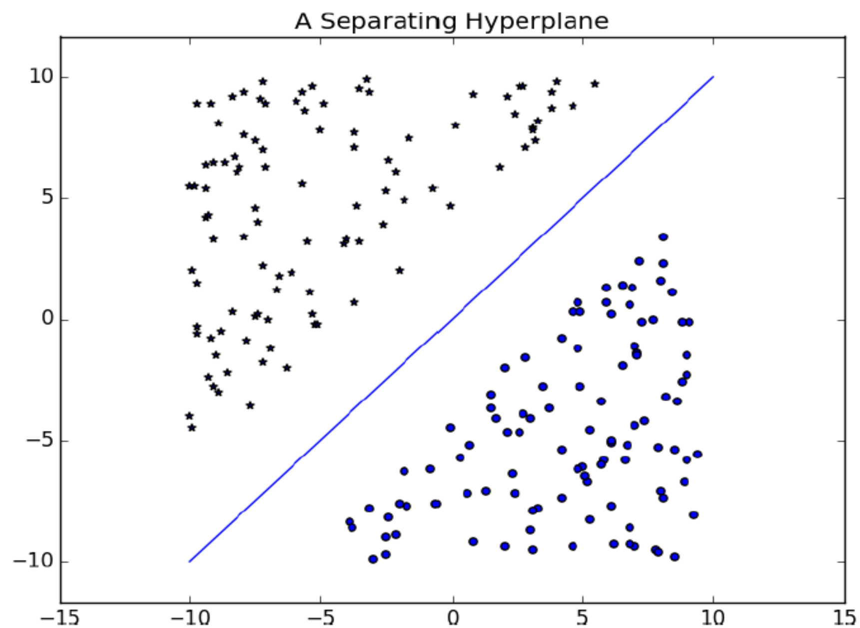


Figure 16-6. A separating hyperplane

Support Vector Machine

- For example, consider the simple one-dimensional dataset. There's no hyperplane that separates the positive examples from the negative ones.

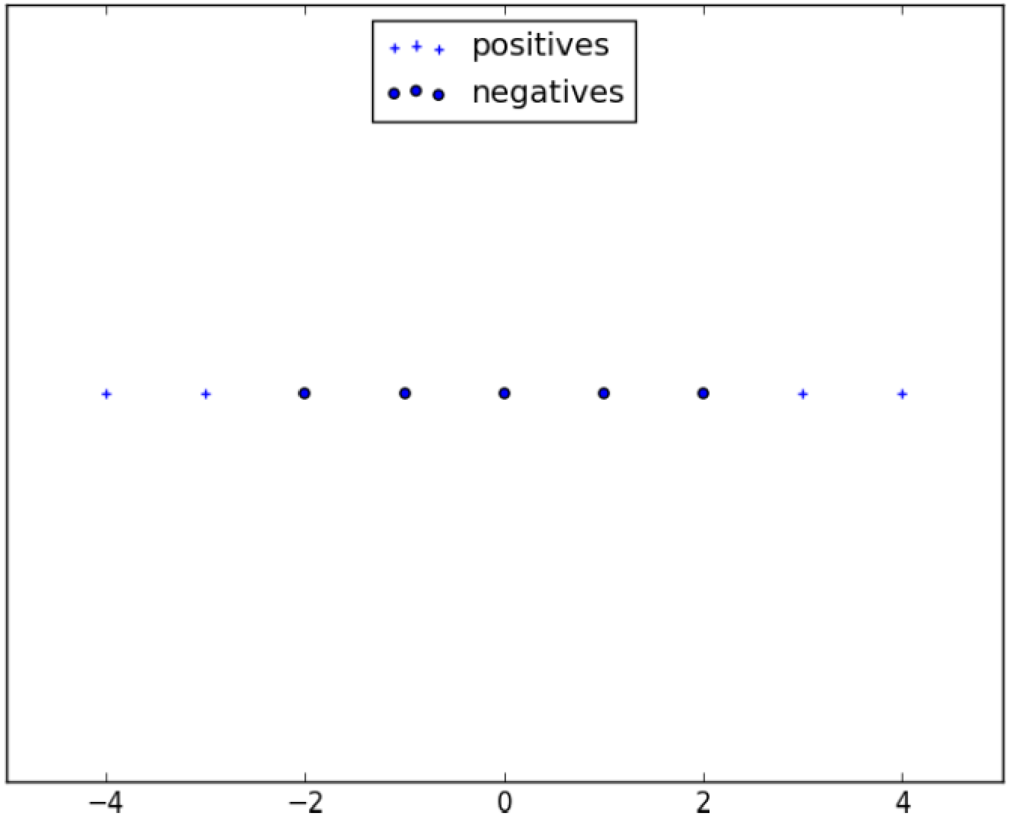
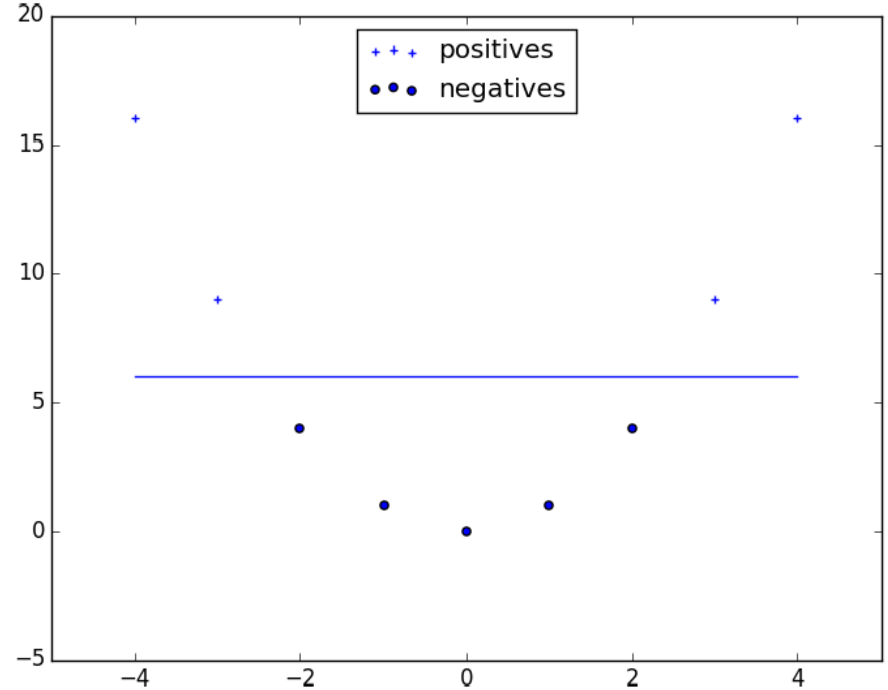


Figure 16-7. A nonseparable one-dimensional dataset

Support Vector Machine

- **Solution: Kernel Trick**
- Transforming the data into a higher dimensional space. Map this dataset to two dimensions by sending the point x to (x, x**2). Now there is a hyperplane that splits the data examples from the negative ones.
- This is usually called the **kernel trick**.
- If there are a lot of points and the mapping is complicated, use a “kernel” function to compute dot products in the higher-dimensional space and use those to find a hyperplane.



Support Vector Machine

SVM to classify a small dataset; plot the separating hyperplane, identify support vectors, and explain how the margin is determined.

BEGIN

1. Create a small dataset:
   - Define feature matrix X with 2D points
   - Define corresponding labels y (0 or 1)

2. Train a linear SVM:
   - Initialize SVM classifier with linear kernel
   - Fit the classifier to X and y

3. Plot data points:
   - For each point in X:
     - Plot with a color based on its label (0 or 1)

4. Plot support vectors:
   - Retrieve support vectors from the classifier
   - Highlight them with larger, outlined markers

Support Vector Machine

6. Compute hyperplane:
   - Get weight vector w (slope/Beta) from classifier
   - Get bias b (Intercept) from classifier(Position hyperplane up/down)
   - Create a range of x-values (x_range)
   - For each x in x_range:
     - Compute y using y = -(w1 * x + b) / w2

7. Compute margin:
   margin = 1 / sqrt(w1^2 + w2^2)
   y_margin_up = y_hyperplane + margin
   y_margin_down = y_hyperplane - margin

$$\text{margin} = \frac{1}{||w||} = \frac{1}{\sqrt{w_1^2 + w_2^2}}$$

8. Plot the hyperplane and margins:
   - Plot the hyperplane using x_range and corresponding y values
   - Plot dashed lines for margins above and below the hyperplane

9. Finalize plot:
   - Add labels, title, legend, and grid
   - Show the plot

END

Support Vector Machine

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm

# Create a small linearly separable dataset
X = np.array([[1, 2], [2, 3], [3, 3], [6, 5], [7, 8], [8, 6]])
y = [0, 0, 0, 1, 1, 1]

# Train a linear SVM
clf = svm.SVC(kernel='linear', C=1.0)
clf.fit(X, y)

# Plotting
plt.figure(figsize=(8, 6))

# Plot data points
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='bwr', label='Data Points')

# Plot support vectors
plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=100, facecolors='none',
edgecolors='k', label='Support Vectors')

# Get the hyperplane
w = clf.coef_[0]
b = clf.intercept_[0]
x_range = np.linspace(0, 10, 100)
y_hyperplane = -(w[0] * x_range + b) / w[1]

# Margins
margin = 1 / np.sqrt(np.sum(w ** 2))
y_margin_up = y_hyperplane + margin
y_margin_down = y_hyperplane - margin
```

Support Vector Machine

```python
# Plot decision boundary and margins
plt.plot(x_range, y_hyperplane, 'k-', label='Hyperplane')
plt.plot(x_range, y_margin_up, 'k--', label='Margins')
plt.plot(x_range, y_margin_down, 'k--')

plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Linear SVM with Margin and Support Vectors')
plt.legend()
plt.grid(True)
plt.show()
```