Supervised Learning Deep Dive

From Linear Regression to Decision Trees

Nipun Batra | IIT Gandhinagar

Recap: Supervised Learning

Input: Features (X) + Labels (y)

Goal: Learn function f where f(X) ≈ y

If y is...	Task	Example
Category	Classification	Spam or Not
Number	Regression	House Price

Today's Agenda

1. **Linear Regression** - Predicting numbers
2. **Logistic Regression** - Predicting categories
3. **Decision Trees** - Rule-based learning
4. **K-Nearest Neighbors** - Learning from similarity
5. **Evaluation Metrics** - How well did we do?

Part 1: Linear Regression

Predicting Continuous Values

The Simplest ML Model

Question: Given house size, predict price?

Size (sqft)	Price ($)
1000	200,000
1500	300,000
2000	400,000

What would a 1750 sqft house cost?

Linear Regression: The Idea

Find the **best fitting line** through the data.

$$\hat{y} = wx + b$$

Symbol	Meaning
x	Input feature (size)
$\hat{y}$	Predicted output (price)
w	Weight (slope)
b	Bias (intercept)

What is "Best Fitting"?

Goal: Minimize the error between predictions and actual values

$$\text{Error} = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

This is called **Mean Squared Error (MSE)** or **Sum of Squared Errors (SSE)**.

Why Squared Errors?

Property	Benefit
Always positive	Errors don't cancel out
Penalizes big errors	Large mistakes cost more
Differentiable	Can use calculus to optimize
Has closed-form solution	Can solve directly

Finding the Best Line

The math (closed form):

$$w = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2}$$

$$b = \bar{y} - w\bar{x}$$

Don't memorize this - sklearn does it for you!

Linear Regression in sklearn

```
from sklearn.linear_model import LinearRegression
import numpy as np

# Data
X = np.array([[1000], [1500], [2000], [2500]])  # sqft
y = np.array([200000, 300000, 400000, 500000])   # price

# Train
model = LinearRegression()
model.fit(X, y)

# Predict
model.predict([[1750]])  # → $350,000
```

Understanding the Model

```
print(f"Weight (w): {model.coef_[0]:.0f}")   # 200
print(f"Bias (b): {model.intercept_:.0f}")    # 0

# The learned equation:
# price = 200 * sqft + 0
# For 1750 sqft: 200 * 1750 = $350,000
```

Interpretation: Each sqft adds $200 to price.

Multiple Features

What if we have more than one feature?

$$\hat{y} = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$$

```
X = [[1500, 3, 2],    # sqft, beds, baths
     [2000, 4, 3],
     [1200, 2, 1]]

model.fit(X, y)

# model.coef_ → [w_sqft, w_beds, w_baths]
```

When Linear Regression Fails

Problem: Not all relationships are linear!

Scenario	Issue
Curved patterns	Line can't capture curve
Outliers	Line gets pulled toward them
Discrete targets	Wrong tool (use classification)

Solution: Use polynomial features or other models.

Part 2: Logistic Regression

Predicting Categories

The Classification Problem

Question: Given email features, is it spam?

num_exclamations	has_FREE	is_spam
5	1	Yes
0	0	No
3	1	Yes
1	0	No

Output is a **category**, not a number!

Why Not Linear Regression?

Linear regression predicts any number (-∞ to +∞).

But for classification, we need:

- Probability between 0 and 1
- A decision boundary

The Sigmoid Function

Maps any number to range (0, 1):

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

z value	σ(z)
-10	≈ 0
0	0.5
+10	≈ 1

Logistic Regression Model

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

Output: Probability of belonging to class 1

Decision:

- If P > 0.5 → Predict class 1
- If P ≤ 0.5 → Predict class 0

Logistic Regression in sklearn

```
from sklearn.linear_model import LogisticRegression

# Data
X = [[5, 1], [0, 0], [3, 1], [1, 0]]  # features
y = [1, 0, 1, 0]  # 1=spam, 0=not spam

# Train
model = LogisticRegression()
model.fit(X, y)

# Predict
model.predict([[4, 1]])  # → [1] (spam)
model.predict_proba([[4, 1]])  # → [0.1, 0.9] (probabilities)
```

Decision Boundary

Logistic regression learns a **linear decision boundary**:

Multi-class Classification

What if we have more than 2 classes?

```
from sklearn.linear_model import LogisticRegression

# 3 classes: 0, 1, 2
model = LogisticRegression(multi_class='multinomial')
model.fit(X, y)

# predict_proba gives probability for each class
model.predict_proba([[2, 3]])  # → [0.1, 0.7, 0.2]
```

Part 3: Decision Trees

Rule-Based Learning

The Most Intuitive Model

Decision Trees make decisions like humans do!

Decision Tree for Classification

How Trees Learn

At each node, find the **best split**:

Criterion	What It Measures
Gini Impurity	How mixed are the classes?
Information Gain	How much uncertainty is reduced?

Goal: Create pure groups (all same class).

Decision Tree in sklearn

```
from sklearn.tree import DecisionTreeClassifier

# Data
X = [[5, 1], [0, 0], [3, 1], [1, 0]]  # features
y = [1, 0, 1, 0]  # spam or not

# Train
model = DecisionTreeClassifier(max_depth=3)
model.fit(X, y)

# Predict
model.predict([[4, 1]])  # → [1] (spam)
```

Visualizing the Tree

```
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 8))
plot_tree(model,
          feature_names=['num_exclaim', 'has_FREE'],
          class_names=['Not Spam', 'Spam'],
          filled=True)
plt.show()
```

Trees for Regression Too!

```
from sklearn.tree import DecisionTreeRegressor

# Predict house prices
model = DecisionTreeRegressor(max_depth=4)
model.fit(X_train, y_train)

# Prediction = average of leaf node examples
model.predict(X_test)
```

Tree Pros and Cons

Pros

- Easy to understand
- No feature scaling needed
- Handles non-linear patterns
- Can show feature importance

Cons

- Prone to overfitting
- Unstable (small changes → big changes)
- Not smooth predictions
- Can be biased if classes imbalanced

Controlling Tree Complexity

```
model = DecisionTreeClassifier(
    max_depth=5,        # Limit depth
    min_samples_leaf=5, # Min samples per leaf
    min_samples_split=10 # Min samples to split
)
```

Deeper tree = More complex, risk of overfitting

Shallower tree = Simpler, might underfit

Part 4: K-Nearest Neighbors

Learning by Similarity

The Simplest Idea

To classify a new point:

1. Find the K closest training examples
2. Take a vote
3. Majority wins!

No explicit training - just store the data!

K-NN Example (K=3)

K-NN in sklearn

```
from sklearn.neighbors import KNeighborsClassifier

# Train (just stores the data!)
model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, y_train)

# Predict (finds nearest neighbors)
model.predict(X_test)
```

Choosing K

K Value	Effect
K=1	Very sensitive, captures noise
K=3-5	Often works well
K=large	Smoother, might miss local patterns

Rule of thumb: Try odd K to avoid ties.

K-NN for Regression

Instead of voting, **average** the neighbors:

```
from sklearn.neighbors import KNeighborsRegressor

model = KNeighborsRegressor(n_neighbors=5)
model.fit(X_train, y_train)

# Prediction = average of 5 nearest prices
model.predict(X_test)
```

K-NN Pros and Cons

Pros

- No training time!
- Simple to understand
- Works for any number of classes
- Natural for some problems

Cons

- Slow prediction (search all points)
- Sensitive to feature scaling
- Curse of dimensionality
- Memory intensive (store all data)

Feature Scaling Matters!

```
from sklearn.preprocessing import StandardScaler

# Scale features to have mean=0, std=1
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Then use K-NN
model.fit(X_train_scaled, y_train)
```

Without scaling: Feature with larger range dominates distance!

Part 5: Evaluation Metrics

How Well Did We Do?

The Fundamental Split

Remember: Always evaluate on **test data**!

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

Training accuracy is **not** the goal!

Classification Metrics: Accuracy

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}}$$

```
from sklearn.metrics import accuracy_score

accuracy = accuracy_score(y_test, predictions)
print(f"Accuracy: {accuracy:.1%}")  # e.g., 95.0%
```

The Accuracy Trap

Scenario: Detecting rare disease (1% of population)

Model	Prediction	Accuracy
Dumb model	Always "Healthy"	99%!
Smart model	Tries to detect	95%

99% accuracy but misses ALL sick patients!

The Confusion Matrix

	Predicted NO	Predicted YES
Actual NO	True Negative (TN)	False Positive (FP)
Actual YES	False Negative (FN)	True Positive (TP)

```
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test, predictions)
print(cm)
```

Precision and Recall

$$\text{Precision} = \frac{TP}{TP + FP}$$

"Of those I predicted positive, how many were correct?"

$$\text{Recall} = \frac{TP}{TP + FN}$$

"Of all actual positives, how many did I find?"

When to Use Which?

Scenario	Priority	Metric
Spam filter	Don't lose good emails	Precision
Cancer detection	Don't miss any cancer	Recall
Balanced need	Both matter equally	F1 Score

F1 Score

Harmonic mean of precision and recall:

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

```
from sklearn.metrics import precision_score, recall_score, f1_score

print(f"Precision: {precision_score(y_test, pred):.2f}")
print(f"Recall: {recall_score(y_test, pred):.2f}")
print(f"F1: {f1_score(y_test, pred):.2f}")
```

Classification Report

```
from sklearn.metrics import classification_report

print(classification_report(y_test, predictions))
```

Output:

```
              precision    recall  f1-score   support
           0       0.95      0.98      0.97       100
           1       0.97      0.93      0.95        80
    accuracy                           0.96       180
   macro avg       0.96      0.96      0.96       180
```

Regression Metrics: MSE and RMSE

Mean Squared Error:

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

Root Mean Squared Error:

$$RMSE = \sqrt{MSE}$$

```
from sklearn.metrics import mean_squared_error
import numpy as np

mse = mean_squared_error(y_test, predictions)
rmse = np.sqrt(mse)
```

Regression Metrics: R^2 Score

$$R^2 = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y})^2}$$

R^2 Value	Interpretation
1.0	Perfect predictions
0.7 - 0.9	Good model
0.0	Same as predicting mean
< 0	Worse than predicting mean

```
from sklearn.metrics import r2_score
r2 = r2_score(y_test, predictions)
```

Metrics Summary

Task	Metric	When to Use
Classification	Accuracy	Balanced classes
Classification	Precision	False positives costly
Classification	Recall	False negatives costly
Classification	F1	Balance precision/recall
Regression	MSE/RMSE	General use
Regression	R^2	Compare models

Putting It All Together

```
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report

# Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Train
model = DecisionTreeClassifier(max_depth=5)
model.fit(X_train, y_train)

# Evaluate
predictions = model.predict(X_test)
print(classification_report(y_test, predictions))
```

Key Takeaways

1. **Linear Regression:** Fit a line, minimize squared error
2. **Logistic Regression:** Classification with probabilities
3. **Decision Trees:** Human-readable rules, prone to overfitting
4. **K-NN:** Classify by voting among neighbors
5. **Metrics matter:** Accuracy isn't everything!

You Now Know 4 Algorithms!

Next: Model Selection & Ensembles

Lab: Implement these algorithms on real datasets

"All models are wrong, but some are useful."
— George Box

Questions?