SUPERVISED MACHINE LEARNING: REGRESSION

MODULE 2:

DATA SPLITS AND POLYNOMIAL REGRESSION

TABLE OF CONTENTS

Data Splitting and Cross-Validation

- The lesson introduces how to **split data into training and testing sets** to assess how a model performs on unseen data.
- **Training data** is used to fit and learn model parameters.
- **Test data** serves as a holdout set to evaluate generalization ability.
- Emphasis is placed on avoiding **data leakage**—when test data unintentionally influences training data.
- **Cross-validation** expands this idea by rotating train-test splits to make performance estimates more reliable.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(

       X, y, test_size=0.3, random_state=42)
```

Data Import and Exploration

- Using **Pandas** to explore data types (`int`, `float`, `object`) and shape.
- The file path is constructed using `os.path.join` to ensure compatibility across operating systems.

```
import pandas as pd, numpy as np, os

file_path = os.path.join("data", "Ames_Housing_Sales.csv")

data = pd.read_csv(file_path)

print(data.shape)

print(data.dtypes.value_counts())
```

One-Hot Encoding (OHE)

- One-hot encoding is used to transform **categorical features** into numerical ones.
- The number of new columns from encoding is **n – 1**, where *n* is the number of unique categories.
- **Sklearn's OneHotEncoder** is preferred over `pd.get_dummies()` for large datasets.
- Sparse matrices are introduced to **save memory** during encoding.
- The concept of **multicollinearity** is explained, and why we sometimes use `drop='first'` to avoid redundant columns.

```
from sklearn.preprocessing import OneHotEncoder

encoder = OneHotEncoder(sparse=True)

encoded = encoder.fit_transform(data[['Neighborhood']])
```

Training and Testing with Encoded Data

- The dataset is split into two versions:
  - Original dataset
  - One-hot encoded dataset
- Both are split using the same random state for consistent evaluation.
- A **Linear Regression** model is trained on each to compare performance.

Findings:

- The **one-hot encoded model** performed better on the training set but worse on the test set → **overfitting.**
- The simpler, non-encoded version generalized better.

```
from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_squared_error

lr = LinearRegression()

lr.fit(X_train, y_train)

y_pred = lr.predict(X_test)

mse = mean_squared_error(y_test, y_pred)
```

Data Scaling

$$z = \frac{x - \mu}{\sigma}$$

Standard Scaler

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Min - Max Scaler

- Demonstrates **StandardScaler** and **MinMaxScaler** from `sklearn.preprocessing`.
- Scaling is applied **only on training data** using `fit_transform`, and **transform only** on test data.
- Scaling **does not affect Linear Regression results**, but it becomes critical for regularized models like Ridge or Lasso.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X_train_scaled = scaler.fit_transform(X_train)

X_test_scaled = scaler.transform(X_test)
```

Model Evaluation and Visualization

- **Scatter plot** of predicted vs actual values helps visualize model accuracy.
- Ideally, all points lie close to the diagonal line, indicating good predictions.

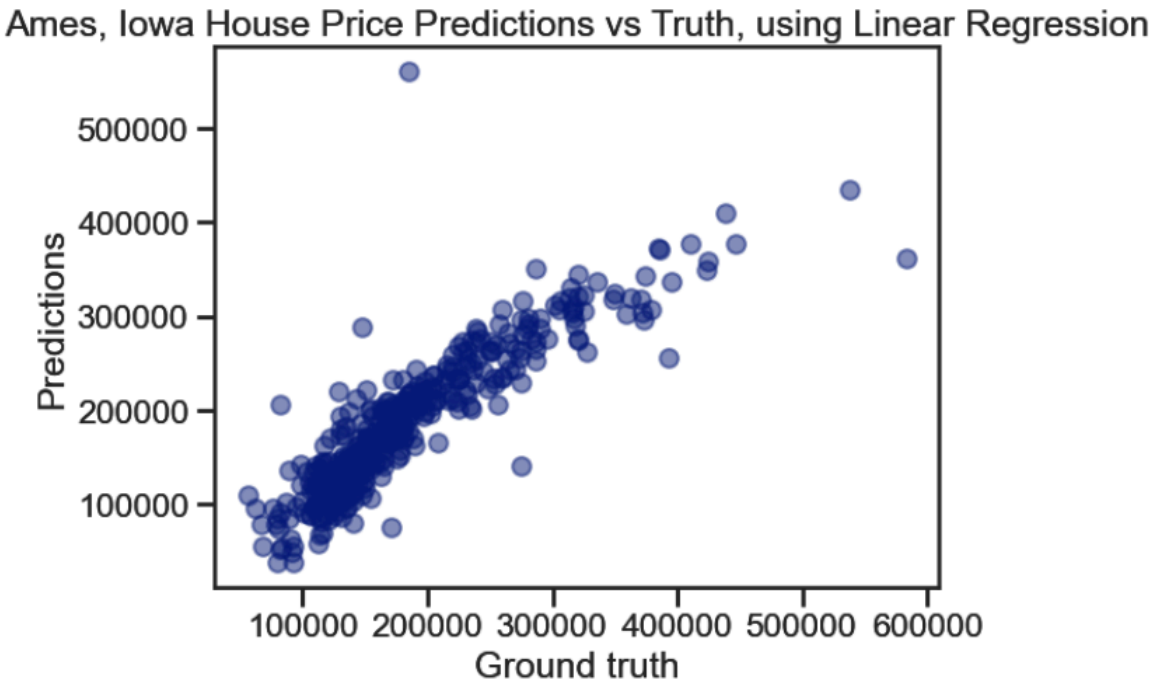
```
plt.scatter(y_test, y_pred, alpha = 0.5)

plt.xlabel('Ground truth')

plt.ylabel('Predictions')

plt.title('Ames, Iowa House Price Predictions vs Truth,
using Linear Regression')

plt.show()
```



Polynomial Regression & Model Complexity

- Extends linear regression by adding **polynomial and interaction terms** to capture nonlinear effects.
- Despite nonlinearity in the data, the model remains **linear in parameters**.

- The section connects this concept to the **bias-variance trade-off** — finding the right complexity to balance **underfitting vs overfitting**.
- Introduces **PolynomialFeatures** for feature expansion.

```
from sklearn.preprocessing import PolynomialFeatures

poly = PolynomialFeatures(degree=2)

X_poly = poly.fit_transform(X)
```

Key Takeaways

1. Data Splitting & Validation
Always split data into training and testing sets to evaluate real-world performance and prevent data leakage.

2. Model Generalization
The goal is not just accuracy on known data but strong performance on unseen data.

3. Cross-Validation
Use multiple train-test splits to ensure stable and reliable model evaluation.

4. Feature Encoding

- Convert categorical variables using One-Hot Encoding.
- Remember to use `drop='first'` when interpretability matters to avoid multicollinearity.

5. Feature Scaling

- Apply scaling only on training data (using `fit_transform`).
- Transform test data to prevent information leakage.

6. Overfitting Awareness
More features or higher complexity can lead to overfitting—high training accuracy but poor generalization.

7. Polynomial Features
Use polynomial or interaction terms to model nonlinear relationships while keeping the model linear in parameters.

8. Bias–Variance Trade-off
Always aim for the right balance between model complexity (low bias) and generalization (low variance).