Data Preprocessing and Feature Engineering

Module 5, Lab 3: Preparing Data for Machine Learning

Raw data is rarely ready for machine learning algorithms. This lab teaches you how to clean, transform, and engineer features to create high-quality datasets that lead to better model performance.

Learning Objectives

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

- Handle missing values using various strategies
- Encode categorical variables for machine learning
- Scale and normalize numerical features
- Detect and handle outliers appropriately
- Create new features through feature engineering
- Build preprocessing pipelines for reproducibility

Why This Matters

Data preprocessing often takes 80% of a data scientist's time, but it's crucial for model success. Poor data preparation leads to poor models, regardless of the algorithm used.

Setup and Data Loading

```
# Install required packages
!pip install --upgrade pip
!pip install pandas numpy matplotlib seaborn scikit-learn
```

```
Requirement already satisfied: pip in
/Users/martin.demel/myenv3.10/lib/python3.10/site-packages (25.2)
Requirement already satisfied: pandas in
/Users/martin.demel/myenv3.10/lib/python3.10/site-packages (2.3.1)
Requirement already satisfied: numpy in
/Users/martin.demel/myenv3.10/lib/python3.10/site-packages (1.26.4)
Requirement already satisfied: matplotlib in
/Users/martin.demel/myenv3.10/lib/python3.10/site-packages (3.10.5)
Requirement already satisfied: seaborn in
/Users/martin.demel/myenv3.10/lib/python3.10/site-packages (0.13.2)
Requirement already satisfied: scikit-learn in
/Users/martin.demel/myenv3.10/lib/python3.10/site-packages (1.7.1)
Requirement already satisfied: python-dateutil>=2.8.2 in
/Users/martin.demel/myenv3.10/lib/python3.10/site-packages (from
pandas) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in
```

```
/Users/martin.demel/myenv3.10/lib/python3.10/site-packages (from
pandas) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in
/Users/martin.demel/myenv3.10/lib/python3.10/site-packages (from
pandas) (2025.2)
Requirement already satisfied: contourpy>=1.0.1 in
/Users/martin.demel/myenv3.10/lib/python3.10/site-packages (from
matplotlib) (1.3.1)
Requirement already satisfied: cycler>=0.10 in
/Users/martin.demel/myenv3.10/lib/python3.10/site-packages (from
matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
/Users/martin.demel/myenv3.10/lib/python3.10/site-packages (from
matplotlib) (4.55.0)
Requirement already satisfied: kiwisolver>=1.3.1 in
/Users/martin.demel/myenv3.10/lib/python3.10/site-packages (from
matplotlib) (1.4.7)
Requirement already satisfied: packaging>=20.0 in
/Users/martin.demel/myenv3.10/lib/python3.10/site-packages (from
matplotlib) (25.0)
Requirement already satisfied: pillow>=8 in
/Users/martin.demel/myenv3.10/lib/python3.10/site-packages (from
matplotlib) (11.0.0)
Requirement already satisfied: pyparsing>=2.3.1 in
/Users/martin.demel/myenv3.10/lib/python3.10/site-packages (from
matplotlib) (3.0.9)
Requirement already satisfied: scipy>=1.8.0 in
/Users/martin.demel/myenv3.10/lib/python3.10/site-packages (from
scikit-learn) (1.13.1)
Requirement already satisfied: joblib>=1.2.0 in
/Users/martin.demel/myenv3.10/lib/python3.10/site-packages (from
scikit-learn) (1.5.1)
Requirement already satisfied: threadpoolctl>=3.1.0 in
/Users/martin.demel/myenv3.10/lib/python3.10/site-packages (from
scikit-learn) (3.5.0)
Requirement already satisfied: six>=1.5 in
/Users/martin.demel/myenv3.10/lib/python3.10/site-packages (from
python-dateutil>=2.8.2->pandas) (1.16.0)
```

```python
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler, MinMaxScaler,
LabelEncoder, OneHotEncoder
from sklearn.impute import SimpleImputer, KNNImputer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
import warnings
warnings.filterwarnings('ignore')

# Set plotting style
plt.style.use('seaborn-v0_8')
%matplotlib inline

print("Libraries imported successfully!")
```

```
Libraries imported successfully!
```

Creating a Realistic Dataset with Data Quality Issues

We'll create a dataset that mimics real-world data problems you'll encounter.

```
# Create a realistic employee dataset with various data quality issues
np.random.seed(42)
n_employees = 1000

# Generate base employee data
employee_data = {
    'employee_id': range(1, n_employees + 1),
    'age': np.random.normal(35, 10, n_employees),
    'years_experience': np.random.exponential(5, n_employees),
    'education_level': np.random.choice(['High School', 'Bachelor',
'Master', 'PhD'],
                                       n_employees, p=[0.2, 0.5, 0.25,
0.05]),
    'department': np.random.choice(['Engineering', 'Sales',
'Marketing', 'HR', 'Finance'],
                                  n_employees, p=[0.3, 0.25, 0.2,
0.15, 0.1]),
    'job_level': np.random.choice(['Junior', 'Mid', 'Senior', 'Lead'],

                                 n_employees, p=[0.3, 0.4, 0.25,
0.05]),
    'location': np.random.choice(['New York', 'San Francisco',
'Chicago', 'Austin', 'Remote'],
                                n_employees, p=[0.25, 0.2, 0.15, 0.15,
0.25]),
    'performance_score': np.random.normal(7.5, 1.5, n_employees),
    'hours_per_week': np.random.normal(42, 8, n_employees),
    'projects_completed': np.random.poisson(8, n_employees),
    'training_hours': np.random.gamma(2, 10, n_employees)
}

# Create DataFrame
df = pd.DataFrame(employee_data)
```

```
# Add realistic constraints
df['age'] = np.clip(df['age'], 22, 65)
df['years_experience'] = np.clip(df['years_experience'], 0, df['age']
- 22)
df['performance_score'] = np.clip(df['performance_score'], 1, 10)
df['hours_per_week'] = np.clip(df['hours_per_week'], 20, 60)

# Create salary based on realistic factors (this will be our target
variable)
base_salary = 50000
education_bonus = {'High School': 0, 'Bachelor': 15000, 'Master':
25000, 'PhD': 40000}
level_bonus = {'Junior': 0, 'Mid': 20000, 'Senior': 40000, 'Lead':
70000}
dept_bonus = {'Engineering': 15000, 'Sales': 10000, 'Marketing': 5000,
'HR': 0, 'Finance': 8000}
location_bonus = {'New York': 20000, 'San Francisco': 25000,
'Chicago': 5000, 'Austin': 8000, 'Remote': 0}

df['salary'] = (base_salary +
                df['education_level'].map(education_bonus) +
                df['job_level'].map(level_bonus) +
                df['department'].map(dept_bonus) +
                df['location'].map(location_bonus) +
                df['years_experience'] * 2000 +
                df['performance_score'] * 3000 +
                np.random.normal(0, 10000, n_employees))

df['salary'] = np.maximum(df['salary'], 35000)  # Minimum salary

print(f"Dataset created with {len(df)} employees")
print(f"Dataset shape: {df.shape}")
df.head()
```

```
Dataset created with 1000 employees
Dataset shape: (1000, 12)

   employee_id        age  years_experience education_level
department  \
0            1  39.967142          0.916506       Bachelor
Marketing
1            2  33.617357          0.552244     High School
Engineering
2            3  41.476885          5.058921     High School
Sales
3            4  50.230299          6.128975       Bachelor
HR
4            5  32.658466          0.160479         Master
Sales
```

```
  job_level       location  performance_score  hours_per_week  \
0    Junior        Chicago           6.516468       52.944635   
1    Junior         Austin           7.146433       43.007211   
2       Mid  San Francisco           6.090805       48.808993   
3       Mid       New York           9.178867       51.790847   
4    Junior         Remote           7.058316       44.703644   

   projects_completed  training_hours         salary  
0                   5       15.103473   98155.118139  
1                   6       10.621985   98464.144788  
2                   7       81.025460  160878.080717  
3                   8       26.356775  145460.439793  
4                   5       26.094782  107113.789261  
```

Introducing Realistic Data Quality Issues

```python
# Introduce missing values (realistic patterns)
# Performance scores might be missing for new employees
new_employee_mask = df['years_experience'] < 0.5
df.loc[new_employee_mask & (np.random.random(len(df)) < 0.3),
'performance_score'] = np.nan

# Training hours might be missing randomly
missing_training = np.random.choice(df.index, size=80, replace=False)
df.loc[missing_training, 'training_hours'] = np.nan

# Some education levels might be missing
missing_education = np.random.choice(df.index, size=30, replace=False)
df.loc[missing_education, 'education_level'] = np.nan

# Add some outliers
# Extremely high performers
outlier_indices = np.random.choice(df.index, size=10, replace=False)
df.loc[outlier_indices, 'hours_per_week'] = np.random.uniform(70, 80,
10)
df.loc[outlier_indices, 'projects_completed'] = np.random.uniform(25,
35, 10)

# Add some inconsistent data
# Some employees with PhD but very low experience (career changers)
career_changer_indices = np.random.choice(df[df['education_level'] ==
'PhD'].index, size=5, replace=False)
df.loc[career_changer_indices, 'years_experience'] =
np.random.uniform(0, 2, 5)

# Add some duplicate-like entries (same person, different records)
duplicate_base = df.sample(3).copy()
duplicate_base['employee_id'] = range(n_employees + 1, n_employees +
4)
# Slightly modify some values to simulate data entry errors
```

```
duplicate_base['age'] += np.random.randint(-1, 2, 3)
duplicate_base['salary'] += np.random.randint(-5000, 5000, 3)
df = pd.concat([df, duplicate_base], ignore_index=True)

print("Data quality issues introduced:")
print(f"Missing values: {df.isnull().sum().sum()}")
print(f"Total records: {len(df)}")
print("\nMissing values by column:")
print(df.isnull().sum()[df.isnull().sum() > 0])
```

```
Data quality issues introduced:
Missing values: 150
Total records: 1003

Missing values by column:
education_level      30
performance_score    40
training_hours       80
dtype: int64
```

Step 1: Data Quality Assessment

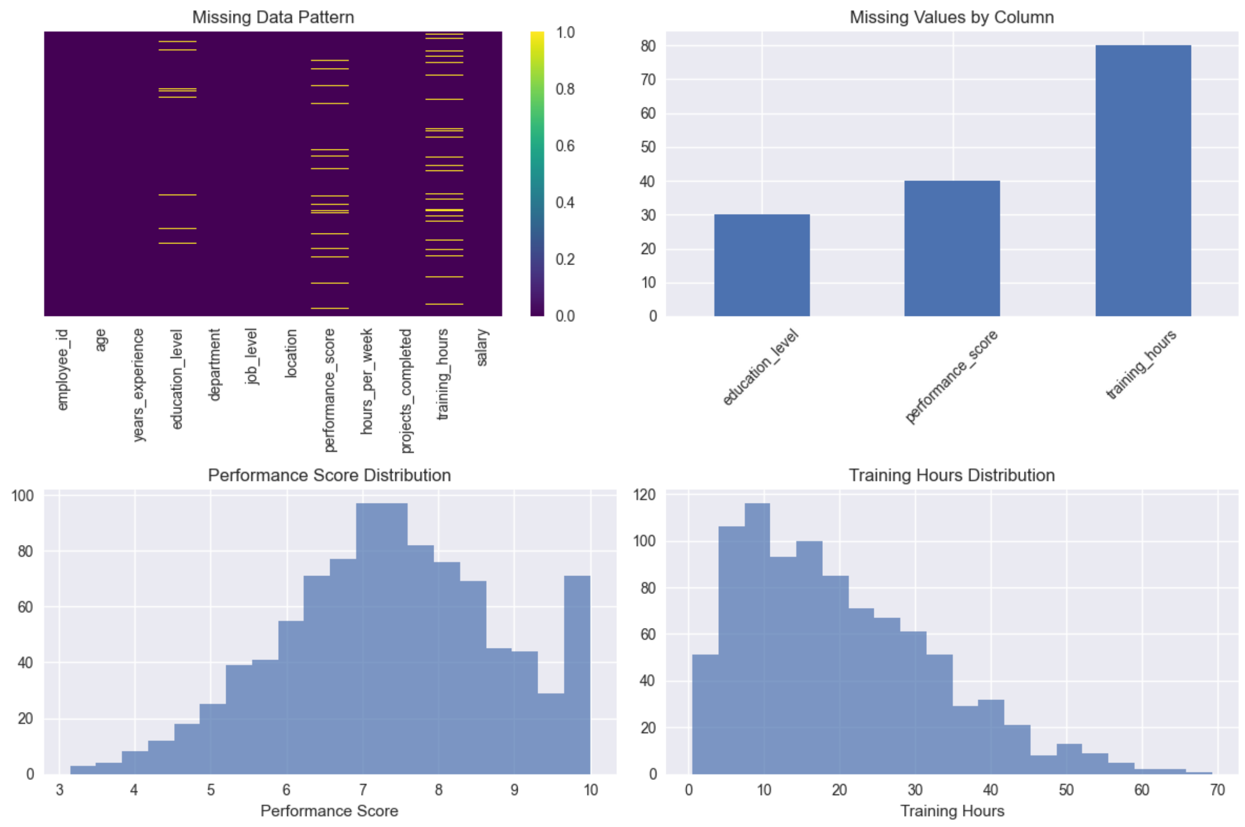
Before preprocessing, let's understand what we're working with.

```
# Comprehensive data quality report
def data_quality_report(df):
    print("=== DATA QUALITY REPORT ===")
    print(f"\n□ Dataset Overview:")
    print(f"   • Shape: {df.shape}")
    print(f"   • Memory usage: {df.memory_usage(deep=True).sum() / 
1024:.1f} KB")

    print(f"\n□ Missing Values:")
    missing_data = df.isnull().sum()
    missing_percent = (missing_data / len(df)) * 100

    for col in missing_data[missing_data > 0].index:
        print(f"   • {col}: {missing_data[col]} 
({missing_percent[col]:.1f}%)")

    print(f"\n□ Data Types:")
    print(f"   • Numerical columns: 
{len(df.select_dtypes(include=[np.number]).columns)}")
    print(f"   • Categorical columns: 
{len(df.select_dtypes(include=['object']).columns)}")

    print(f"\n□ Potential Issues:")
    # Check for duplicates
    duplicates = len(df) - len(df.drop_duplicates())
    if duplicates > 0:
        print(f"   • Duplicate rows: {duplicates}")

    # Check for outliers in numerical columns
    numerical_cols = df.select_dtypes(include=[np.number]).columns
    for col in numerical_cols:
        Q1 = df[col].quantile(0.25)
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1
        outliers = len(df[(df[col] < Q1 - 1.5*IQR) | (df[col] > Q3 +
1.5*IQR)])
        if outliers > 0:
            print(f"   • {col} outliers: {outliers}
({outliers/len(df)*100:.1f}%)")

data_quality_report(df)
```

```
=== DATA QUALITY REPORT ===

□ Dataset Overview:
   • Shape: (1003, 12)
   • Memory usage: 312.6 KB

□ Missing Values:
   • education_level: 30 (3.0%)
   • performance_score: 40 (4.0%)
   • training_hours: 80 (8.0%)

□ Data Types:
   • Numerical columns: 8
   • Categorical columns: 4

□ Potential Issues:
   • age outliers: 4 (0.4%)
   • years_experience outliers: 34 (3.4%)
   • performance_score outliers: 3 (0.3%)
   • hours_per_week outliers: 10 (1.0%)
   • projects_completed outliers: 11 (1.1%)
   • training_hours outliers: 10 (1.0%)
   • salary outliers: 6 (0.6%)
```

```
# Visualize missing data patterns
plt.figure(figsize=(12, 8))

# Missing data heatmap
plt.subplot(2, 2, 1)
sns.heatmap(df.isnull(), cbar=True, yticklabels=False, cmap='viridis')
plt.title('Missing Data Pattern')

# Missing data bar plot
```

```
plt.subplot(2, 2, 2)
missing_counts = df.isnull().sum()
missing_counts = missing_counts[missing_counts > 0]
missing_counts.plot(kind='bar')
plt.title('Missing Values by Column')
plt.xticks(rotation=45)

# Distribution of numerical variables with missing values
plt.subplot(2, 2, 3)
df['performance_score'].hist(bins=20, alpha=0.7)
plt.title('Performance Score Distribution')
plt.xlabel('Performance Score')

plt.subplot(2, 2, 4)
df['training_hours'].hist(bins=20, alpha=0.7)
plt.title('Training Hours Distribution')
plt.xlabel('Training Hours')

plt.tight_layout()
plt.show()
```



Step 2: Handling Missing Values

Different strategies work better for different types of missing data.

2.1 Understanding Missing Data Patterns

```
# Analyze missing data patterns
print("Missing Data Analysis:")
print("\n1. Performance Score Missing Pattern:")
missing_perf = df[df['performance_score'].isnull()]
print(f"   • Average years of experience: 
{missing_perf['years_experience'].mean():.2f}")
print(f"   • Most common job level: 
{missing_perf['job_level'].mode().iloc[0]}")

print("\n2. Training Hours Missing Pattern:")
missing_training = df[df['training_hours'].isnull()]
print(f"   • Average age: {missing_training['age'].mean():.2f}")
print(f"   • Department distribution:")
print(missing_training['department'].value_counts())

print("\n3. Education Level Missing Pattern:")
missing_edu = df[df['education_level'].isnull()]
print(f"   • Average salary: ${missing_edu['salary'].mean():.0f}")
print(f"   • Average years experience: 
{missing_edu['years_experience'].mean():.2f}")
```

```
Missing Data Analysis:

1. Performance Score Missing Pattern:
   • Average years of experience: 0.17
   • Most common job level: Junior

2. Training Hours Missing Pattern:
   • Average age: 35.49
   • Department distribution:
department
Engineering    21
Sales          20
Marketing      15
HR             13
Finance        11
Name: count, dtype: int64

3. Education Level Missing Pattern:
   • Average salary: $137495
   • Average years experience: 3.90
```

2.2 Imputation Strategies

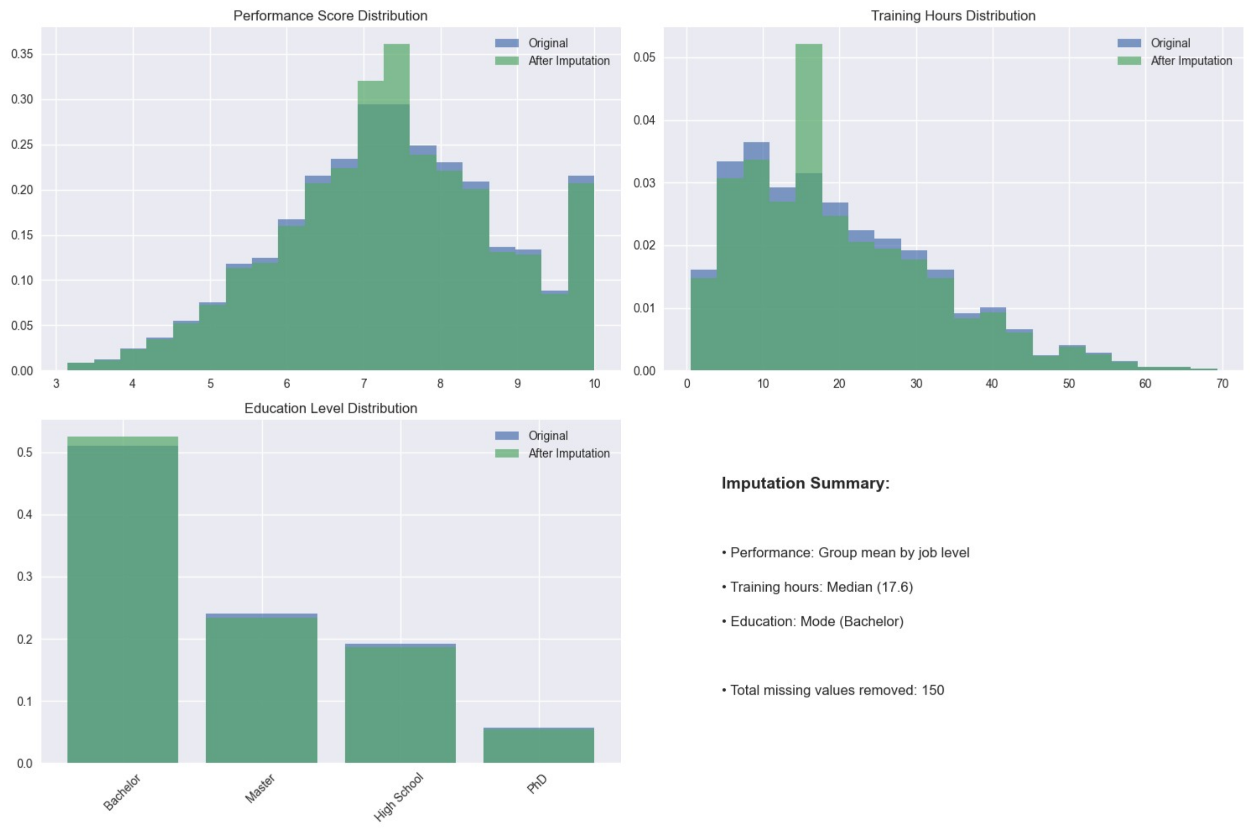
```
# Create a copy for preprocessing
df_processed = df.copy()

print("Applying different imputation strategies...")
```

```python
# Strategy 1: Mean imputation for performance_score (numerical)
# But let's be smarter - use group mean based on job level
performance_means = df_processed.groupby('job_level')
['performance_score'].mean()
print("\nPerformance score means by job level:")
print(performance_means)

for level in performance_means.index:
    mask = (df_processed['job_level'] == level) &
(df_processed['performance_score'].isnull())
    df_processed.loc[mask, 'performance_score'] =
performance_means[level]

# Strategy 2: Median imputation for training_hours (skewed
distribution)
training_median = df_processed['training_hours'].median()
df_processed['training_hours'].fillna(training_median, inplace=True)
print(f"\nFilled training hours with median: {training_median:.1f}")

# Strategy 3: Mode imputation for education_level (categorical)
education_mode = df_processed['education_level'].mode().iloc[0]
df_processed['education_level'].fillna(education_mode, inplace=True)
print(f"Filled education level with mode: {education_mode}")

print(f"\nMissing values after imputation:
{df_processed.isnull().sum().sum()}")
```

```
Applying different imputation strategies...

Performance score means by job level:
job_level
Junior    7.483550
Lead      7.326790
Mid       7.256741
Senior    7.435315
Name: performance_score, dtype: float64

Filled training hours with median: 17.6
Filled education level with mode: Bachelor

Missing values after imputation: 0
```

```python
# Compare distributions before and after imputation
fig, axes = plt.subplots(2, 2, figsize=(15, 10))

# Performance score
axes[0, 0].hist(df['performance_score'].dropna(), bins=20, alpha=0.7,
label='Original', density=True)
axes[0, 0].hist(df_processed['performance_score'], bins=20, alpha=0.7,
label='After Imputation', density=True)
```

```python
axes[0, 0].set_title('Performance Score Distribution')
axes[0, 0].legend()

# Training hours
axes[0, 1].hist(df['training_hours'].dropna(), bins=20, alpha=0.7,
label='Original', density=True)
axes[0, 1].hist(df_processed['training_hours'], bins=20, alpha=0.7,
label='After Imputation', density=True)
axes[0, 1].set_title('Training Hours Distribution')
axes[0, 1].legend()

# Education level
original_edu = df['education_level'].value_counts(normalize=True)
imputed_edu =
df_processed['education_level'].value_counts(normalize=True)
axes[1, 0].bar(range(len(original_edu)), original_edu.values,
alpha=0.7, label='Original')
axes[1, 0].bar(range(len(imputed_edu)), imputed_edu.values, alpha=0.7,
label='After Imputation')
axes[1, 0].set_xticks(range(len(original_edu)))
axes[1, 0].set_xticklabels(original_edu.index, rotation=45)
axes[1, 0].set_title('Education Level Distribution')
axes[1, 0].legend()

# Summary
axes[1, 1].text(0.1, 0.8, 'Imputation Summary:', fontsize=14,
fontweight='bold')
axes[1, 1].text(0.1, 0.6, f'• Performance: Group mean by job level',
fontsize=12)
axes[1, 1].text(0.1, 0.5, f'• Training hours: Median
({training_median:.1f})', fontsize=12)
axes[1, 1].text(0.1, 0.4, f'• Education: Mode ({education_mode})',
fontsize=12)
axes[1, 1].text(0.1, 0.2, f'• Total missing values removed:
{df.isnull().sum().sum()}', fontsize=12)
axes[1, 1].set_xlim(0, 1)
axes[1, 1].set_ylim(0, 1)
axes[1, 1].axis('off')

plt.tight_layout()
plt.show()
```



Imputation Summary:

• Performance: Group mean by job level

• Training hours: Median (17.6)

• Education: Mode (Bachelor)

• Total missing values removed: 150

Step 3: Encoding Categorical Variables

Machine learning algorithms work with numbers, so we need to convert categorical data.

```
# Identify categorical columns
categorical_columns = ['education_level', 'department', 'job_level',
'location']
print("Categorical columns to encode:")
for col in categorical_columns:
    print(f"• {col}: {df_processed[col].nunique()} unique values")
    print(f"  Values: {list(df_processed[col].unique())}")
    print()
```

```
Categorical columns to encode:
• education_level: 4 unique values
  Values: ['Bachelor', 'High School', 'Master', 'PhD']

• department: 5 unique values
  Values: ['Marketing', 'Engineering', 'Sales', 'HR', 'Finance']

• job_level: 4 unique values
  Values: ['Junior', 'Mid', 'Senior', 'Lead']

• location: 5 unique values
```

```
  Values: ['Chicago', 'Austin', 'San Francisco', 'New York', 'Remote']
```

3.1 Ordinal Encoding (for ordered categories)

```python
# Education level has a natural order
education_order = ['High School', 'Bachelor', 'Master', 'PhD']
education_mapping = {level: i for i, level in
enumerate(education_order)}
df_processed['education_level_encoded'] =
df_processed['education_level'].map(education_mapping)

# Job level also has a natural order
job_order = ['Junior', 'Mid', 'Senior', 'Lead']
job_mapping = {level: i for i, level in enumerate(job_order)}
df_processed['job_level_encoded'] =
df_processed['job_level'].map(job_mapping)

print("Ordinal Encoding Applied:")
print("\nEducation Level Mapping:")
for original, encoded in education_mapping.items():
    print(f"  {original} → {encoded}")

print("\nJob Level Mapping:")
for original, encoded in job_mapping.items():
    print(f"  {original} → {encoded}")
```

```
Ordinal Encoding Applied:

Education Level Mapping:
  High School → 0
  Bachelor → 1
  Master → 2
  PhD → 3

Job Level Mapping:
  Junior → 0
  Mid → 1
  Senior → 2
  Lead → 3
```

3.2 One-Hot Encoding (for nominal categories)

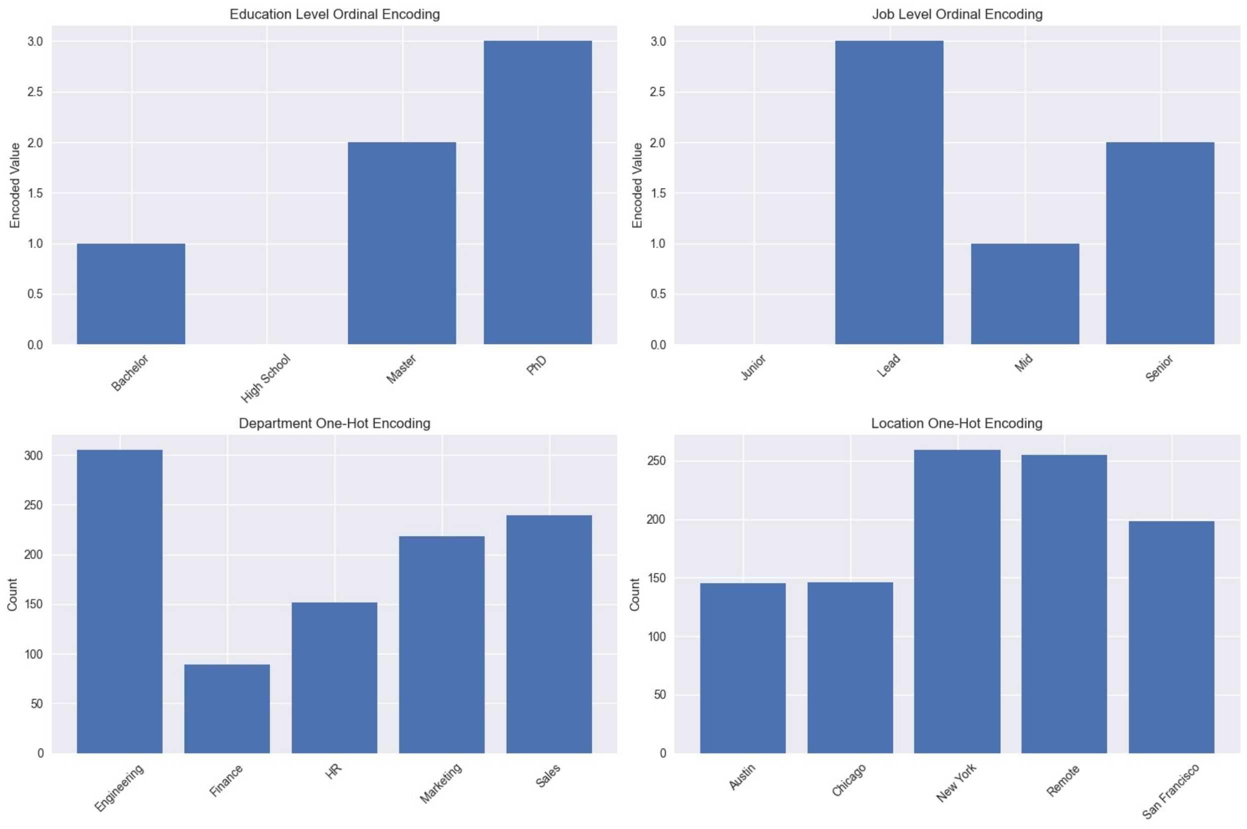
```python
# Department and location don't have natural order - use one-hot
encoding
# Create dummy variables
department_dummies = pd.get_dummies(df_processed['department'],
prefix='dept')
location_dummies = pd.get_dummies(df_processed['location'],
prefix='loc')
```

```
print("One-Hot Encoding Applied:")
print(f"\nDepartment columns created:
{list(department_dummies.columns)}")
print(f"Location columns created: {list(location_dummies.columns)}")

# Add to dataframe
df_processed = pd.concat([df_processed, department_dummies,
location_dummies], axis=1)

print(f"\nDataset shape after encoding: {df_processed.shape}")
```

```
One-Hot Encoding Applied:

Department columns created: ['dept_Engineering', 'dept_Finance',
'dept_HR', 'dept_Marketing', 'dept_Sales']
Location columns created: ['loc_Austin', 'loc_Chicago', 'loc_New
York', 'loc_Remote', 'loc_San Francisco']

Dataset shape after encoding: (1003, 24)
```

```
# Visualize the encoding results
fig, axes = plt.subplots(2, 2, figsize=(15, 10))

# Education level encoding
education_comparison = pd.DataFrame({
    'Original': df_processed['education_level'].value_counts(),
    'Encoded': df_processed.groupby('education_level')
['education_level_encoded'].first()
})
axes[0, 0].bar(education_comparison.index,
education_comparison['Encoded'])
axes[0, 0].set_title('Education Level Ordinal Encoding')
axes[0, 0].set_ylabel('Encoded Value')
axes[0, 0].tick_params(axis='x', rotation=45)

# Job level encoding
job_comparison = pd.DataFrame({
    'Original': df_processed['job_level'].value_counts(),
    'Encoded': df_processed.groupby('job_level')
['job_level_encoded'].first()
})
axes[0, 1].bar(job_comparison.index, job_comparison['Encoded'])
axes[0, 1].set_title('Job Level Ordinal Encoding')
axes[0, 1].set_ylabel('Encoded Value')
axes[0, 1].tick_params(axis='x', rotation=45)

# Department one-hot encoding
dept_cols = [col for col in df_processed.columns if
col.startswith('dept_')]
```

```python
dept_sums = df_processed[dept_cols].sum()
axes[1, 0].bar(range(len(dept_sums)), dept_sums.values)
axes[1, 0].set_xticks(range(len(dept_sums)))
axes[1, 0].set_xticklabels([col.replace('dept_', '') for col in
dept_sums.index], rotation=45)
axes[1, 0].set_title('Department One-Hot Encoding')
axes[1, 0].set_ylabel('Count')

# Location one-hot encoding
loc_cols = [col for col in df_processed.columns if
col.startswith('loc_')]
loc_sums = df_processed[loc_cols].sum()
axes[1, 1].bar(range(len(loc_sums)), loc_sums.values)
axes[1, 1].set_xticks(range(len(loc_sums)))
axes[1, 1].set_xticklabels([col.replace('loc_', '') for col in
loc_sums.index], rotation=45)
axes[1, 1].set_title('Location One-Hot Encoding')
axes[1, 1].set_ylabel('Count')

plt.tight_layout()
plt.show()
```



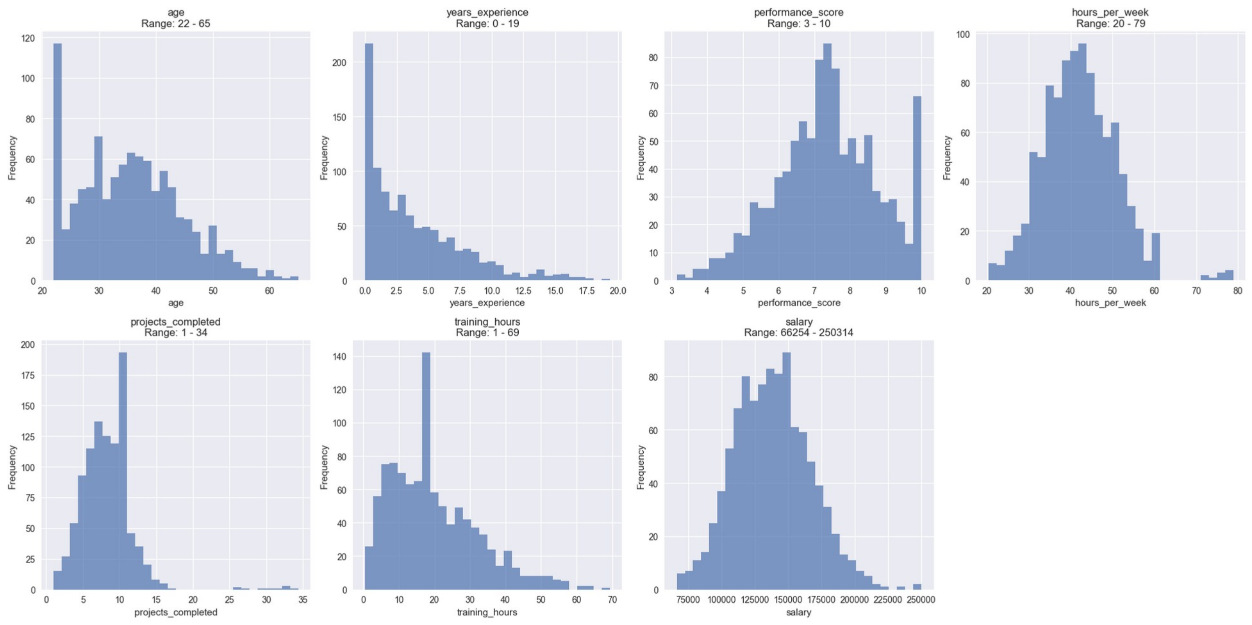
Step 4: Feature Scaling and Normalization

Different features have different scales, which can bias machine learning algorithms.

```
# Identify numerical columns for scaling
numerical_columns = ['age', 'years_experience', 'performance_score',
'hours_per_week',
                    'projects_completed', 'training_hours', 'salary']

print("Numerical columns statistics before scaling:")
print(df_processed[numerical_columns].describe().round(2))
```

```
Numerical columns statistics before scaling:
           age  years_experience  performance_score  hours_per_week  \
count  1003.00           1003.00            1003.00         1003.00
mean     35.52              3.90               7.37           42.09
std       9.11              3.79               1.42            8.73
min      22.00              0.00               3.15           20.30
25%      28.49              0.86               6.47           36.00
50%      35.25              2.93               7.42           41.74
75%      41.47              5.93               8.33           47.69
max      65.00             19.34              10.00           78.85

       projects_completed  training_hours     salary
count             1003.00         1003.00    1003.00
mean                 8.30           19.86  138370.24
std                  3.60           12.40   28602.46
min                  1.00            0.52   66253.73
25%                  6.00           10.23  117935.23
50%                  8.00           17.61  137586.00
75%                 10.00           27.27  157437.15
max                 34.36           69.34  250314.44
```

```
# Visualize the scale differences
fig, axes = plt.subplots(2, 4, figsize=(20, 10))
axes = axes.ravel()

for i, col in enumerate(numerical_columns):
    axes[i].hist(df_processed[col], bins=30, alpha=0.7)
    axes[i].set_title(f'{col}\nRange: {df_processed[col].min():.0f} -
{df_processed[col].max():.0f}')
    axes[i].set_xlabel(col)
    axes[i].set_ylabel('Frequency')

# Remove empty subplot
fig.delaxes(axes[7])

plt.tight_layout()
plt.show()
```



4.1 Standard Scaling (Z-score normalization)

```
# Apply StandardScaler (mean=0, std=1)
scaler_standard = StandardScaler()

# We'll exclude salary from scaling since it's our target variable
features_to_scale = [col for col in numerical_columns if col !=
'salary']

# Fit and transform
df_standard_scaled = df_processed.copy()
df_standard_scaled[features_to_scale] =
scaler_standard.fit_transform(df_processed[features_to_scale])

print("Standard Scaling Applied:")
print("\nFeatures scaled (mean=0, std=1):")
print(df_standard_scaled[features_to_scale].describe().round(3))
```

```
Standard Scaling Applied:

Features scaled (mean=0, std=1):
            age  years_experience  performance_score
hours_per_week  \
count  1003.000          1003.000           1003.000          1003.000

mean     -0.000            -0.000              0.000             0.000

std       1.000             1.000              1.000             1.000

min      -1.484            -1.031             -2.976            -2.497

25%      -0.772            -0.803             -0.636            -0.698
```

```
50%      -0.030              -0.257                 0.032              -0.040
75%       0.653               0.536                 0.673               0.641
max       3.237               4.080                 1.852               4.211

       projects_completed  training_hours
count            1003.000        1003.000
mean                0.000           0.000
std                 1.000           1.000
min                -2.032          -1.561
25%                -0.640          -0.777
50%                -0.084          -0.182
75%                 0.473           0.597
max                 7.252           3.994
```

4.2 Min-Max Scaling (0-1 normalization)

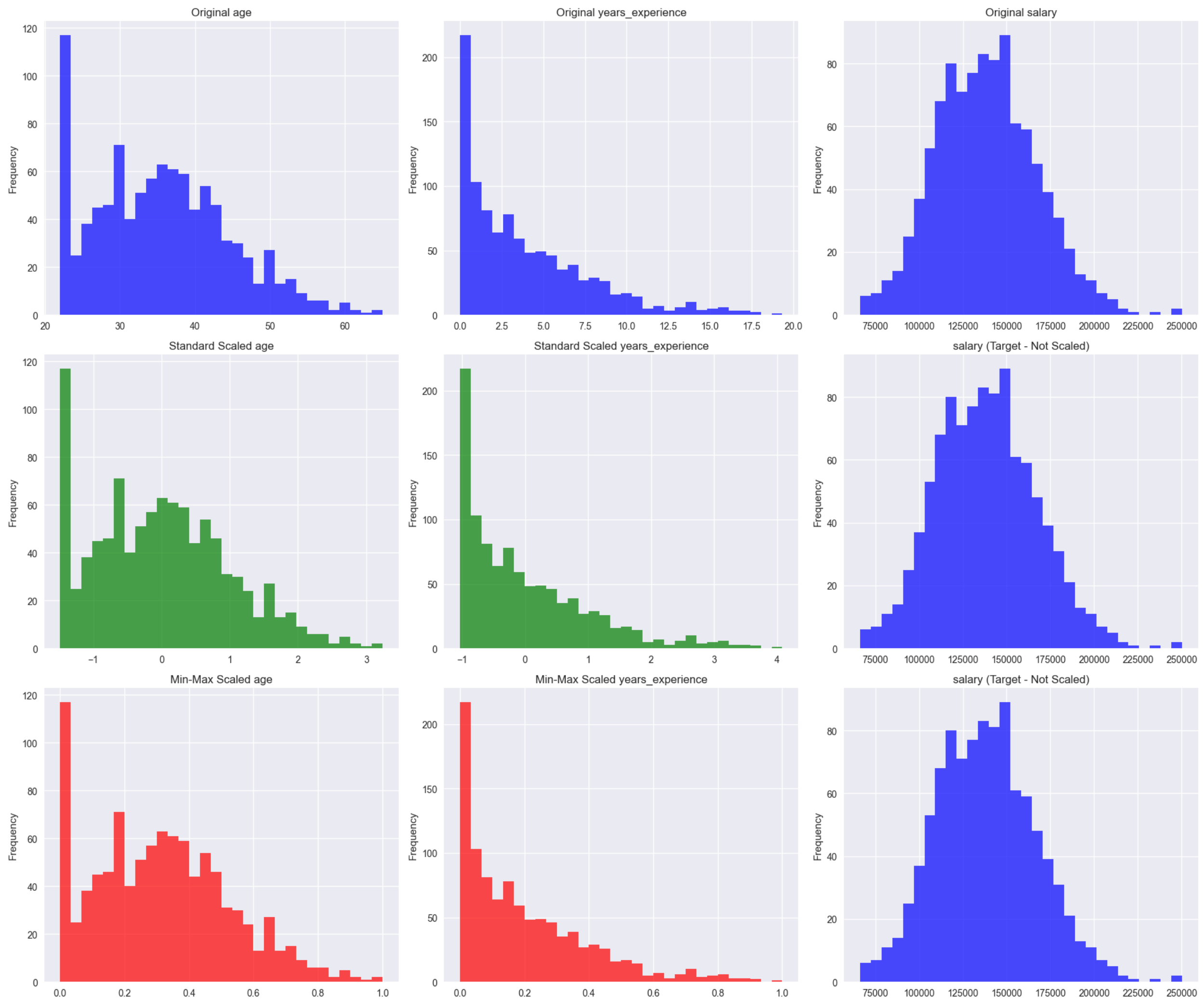
```
# Apply MinMaxScaler (range 0-1)
scaler_minmax = MinMaxScaler()

# Fit and transform
df_minmax_scaled = df_processed.copy()
df_minmax_scaled[features_to_scale] =
scaler_minmax.fit_transform(df_processed[features_to_scale])

print("Min-Max Scaling Applied:")
print("\nFeatures scaled (range 0-1):")
print(df_minmax_scaled[features_to_scale].describe().round(3))
```

```
Min-Max Scaling Applied:

Features scaled (range 0-1):
            age  years_experience  performance_score
hours_per_week  \
count  1003.000          1003.000           1003.000          1003.000
mean      0.314             0.202              0.616             0.372
std       0.212             0.196              0.207             0.149
min       0.000             0.000              0.000             0.000
25%       0.151             0.045              0.485             0.268
50%       0.308             0.151              0.623             0.366
75%       0.453             0.307              0.756             0.468
```

```
max       1.000              1.000                1.000            1.000


       projects_completed  training_hours
count            1003.000        1003.000
mean                0.219           0.281
std                 0.108           0.180
min                 0.000           0.000
25%                 0.150           0.141
50%                 0.210           0.248
75%                 0.270           0.389
max                 1.000           1.000
```

```
# Compare scaling methods
fig, axes = plt.subplots(3, 3, figsize=(18, 15))

# Select a few key features for comparison
comparison_features = ['age', 'years_experience', 'salary']

for i, feature in enumerate(comparison_features):
    # Original
    axes[0, i].hist(df_processed[feature], bins=30, alpha=0.7,
color='blue')
    axes[0, i].set_title(f'Original {feature}')
    axes[0, i].set_ylabel('Frequency')

    if feature != 'salary':  # Don't scale target variable
        # Standard scaled
        axes[1, i].hist(df_standard_scaled[feature], bins=30,
alpha=0.7, color='green')
        axes[1, i].set_title(f'Standard Scaled {feature}')
        axes[1, i].set_ylabel('Frequency')

        # Min-max scaled
        axes[2, i].hist(df_minmax_scaled[feature], bins=30, alpha=0.7,
color='red')
        axes[2, i].set_title(f'Min-Max Scaled {feature}')
        axes[2, i].set_ylabel('Frequency')
    else:
        # For salary, show the same distribution
        axes[1, i].hist(df_processed[feature], bins=30, alpha=0.7,
color='blue')
        axes[1, i].set_title(f'{feature} (Target - Not Scaled)')
        axes[1, i].set_ylabel('Frequency')

        axes[2, i].hist(df_processed[feature], bins=30, alpha=0.7,
color='blue')
        axes[2, i].set_title(f'{feature} (Target - Not Scaled)')
        axes[2, i].set_ylabel('Frequency')
```

```
plt.tight_layout()
plt.show()
```



Step 5: Outlier Detection and Treatment

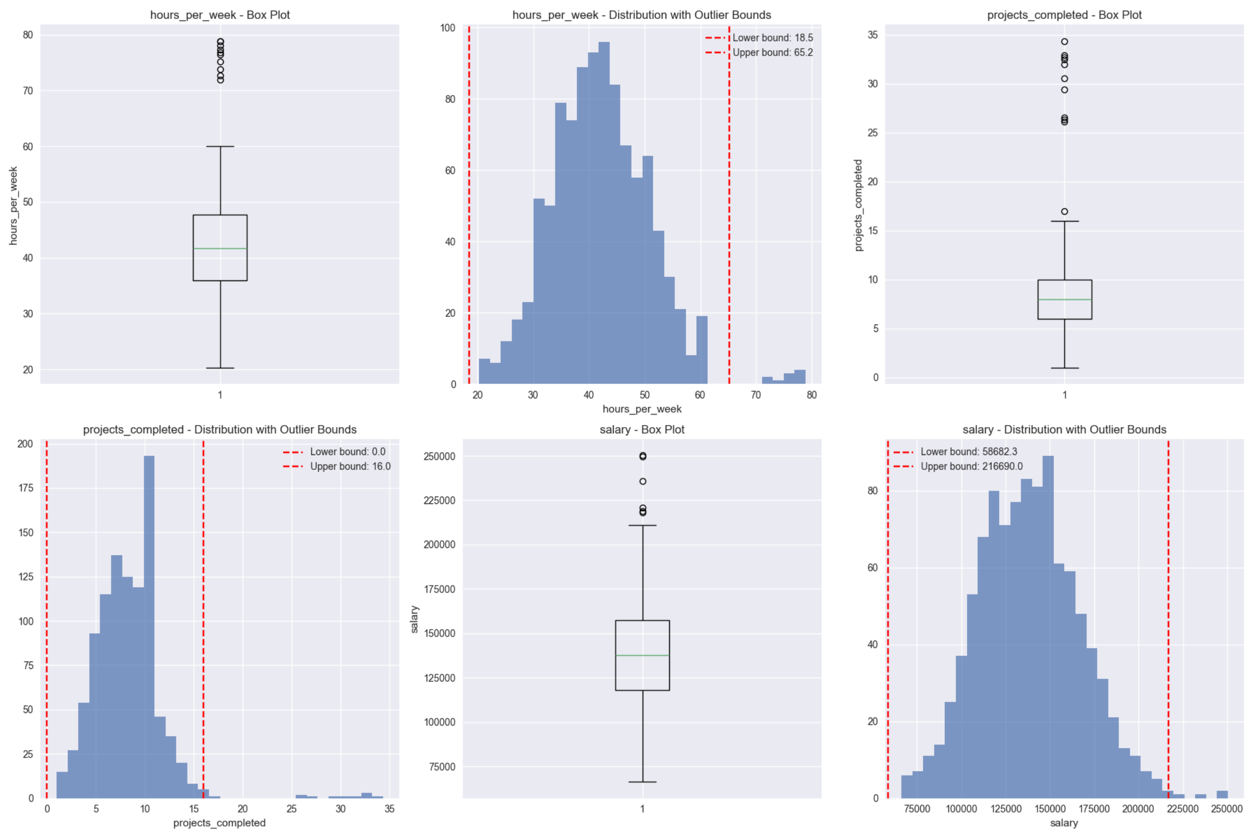
Outliers can significantly impact model performance.

```
# Detect outliers using IQR method
def detect_outliers_iqr(df, column, multiplier=1.5):
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - multiplier * IQR
    upper_bound = Q3 + multiplier * IQR

    outliers = df[(df[column] < lower_bound) | (df[column] >
upper_bound)]
```

```
    return outliers, lower_bound, upper_bound

# Detect outliers in key numerical columns
outlier_columns = ['hours_per_week', 'projects_completed', 'salary']

print("Outlier Detection Results:")
for col in outlier_columns:
    outliers, lower, upper = detect_outliers_iqr(df_processed, col)
    print(f"\n{col}:")
    print(f"  Normal range: {lower:.2f} to {upper:.2f}")
    print(f"  Outliers found: {len(outliers)}
({len(outliers)/len(df_processed)*100:.1f}%)")
    if len(outliers) > 0:
        print(f"  Outlier values: {sorted(outliers[col].values)
[:5]}...")  # Show first 5
```

```
Outlier Detection Results:

hours_per_week:
  Normal range: 18.46 to 65.22
  Outliers found: 10 (1.0%)
  Outlier values: [71.94835522592277, 72.7064801321802,
73.86425859846484, 75.22909034668608, 76.49865962409]...

projects_completed:
  Normal range: 0.00 to 16.00
  Outliers found: 11 (1.1%)
  Outlier values: [17.0, 26.124664359613647, 26.308214262651816,
26.589054348754964, 29.42997696754455]...

salary:
  Normal range: 58682.34 to 216690.04
  Outliers found: 6 (0.6%)
  Outlier values: [218159.66297778895, 218796.9428226131,
221022.70465305357, 235655.16665500513, 249670.1677638195]...
```

```
# Visualize outliers
fig, axes = plt.subplots(2, 3, figsize=(18, 12))
axes = axes.ravel()

for i, col in enumerate(outlier_columns):
    # Box plot
    axes[i*2].boxplot(df_processed[col])
    axes[i*2].set_title(f'{col} - Box Plot')
    axes[i*2].set_ylabel(col)

    # Histogram with outlier boundaries
    outliers, lower, upper = detect_outliers_iqr(df_processed, col)
    axes[i*2+1].hist(df_processed[col], bins=30, alpha=0.7)
    axes[i*2+1].axvline(lower, color='red', linestyle='--',
label=f'Lower bound: {lower:.1f}')
    axes[i*2+1].axvline(upper, color='red', linestyle='--',
label=f'Upper bound: {upper:.1f}')
    axes[i*2+1].set_title(f'{col} - Distribution with Outlier Bounds')
    axes[i*2+1].set_xlabel(col)
    axes[i*2+1].legend()

plt.tight_layout()
plt.show()
```



```
# Handle outliers - we'll use capping (Winsorization)
df_outlier_treated = df_processed.copy()

for col in outlier_columns:
    outliers, lower, upper = detect_outliers_iqr(df_processed, col)

    # Cap outliers at the bounds
    df_outlier_treated[col] =
df_outlier_treated[col].clip(lower=lower, upper=upper)

    print(f"\n{col} outlier treatment:")
    print(f"  Values capped below {lower:.2f}:
{len(df_processed[df_processed[col] < lower])}")
    print(f"  Values capped above {upper:.2f}:
```

```
{len(df_processed[df_processed[col] > upper])}")

print("\nOutlier treatment completed using Winsorization (capping).")
```

```
hours_per_week outlier treatment:
  Values capped below 18.46: 0
  Values capped above 65.22: 10

projects_completed outlier treatment:
  Values capped below 0.00: 0
  Values capped above 16.00: 11

salary outlier treatment:
  Values capped below 58682.34: 0
  Values capped above 216690.04: 6

Outlier treatment completed using Winsorization (capping).
```

Step 6: Feature Engineering

Creating new features that might be more predictive than the original ones.

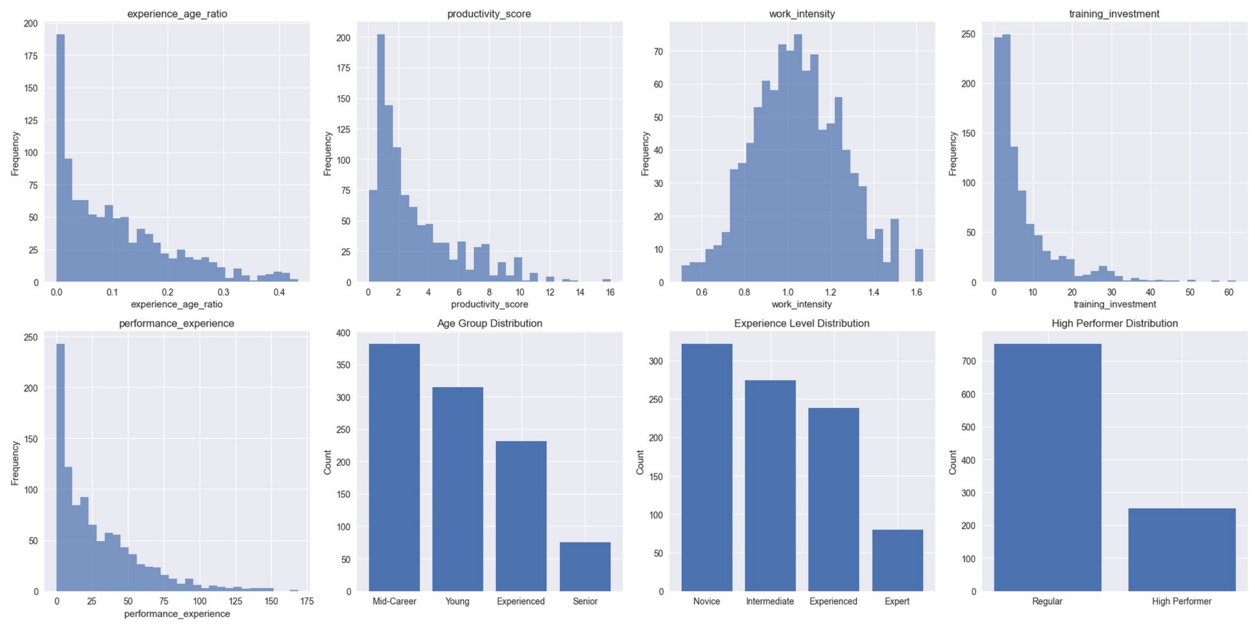
```
# Create new features based on domain knowledge
df_engineered = df_outlier_treated.copy()

print("Creating new features...")

# 1. Experience-to-age ratio (career focus indicator)
df_engineered['experience_age_ratio'] =
df_engineered['years_experience'] / df_engineered['age']

# 2. Productivity score (projects per year of experience)
df_engineered['productivity_score'] =
df_engineered['projects_completed'] /
(df_engineered['years_experience'] + 1)  # +1 to avoid division by
zero

# 3. Work intensity (hours per week relative to standard 40)
df_engineered['work_intensity'] = df_engineered['hours_per_week'] / 40

# 4. Training investment (training hours per year of experience)
df_engineered['training_investment'] = df_engineered['training_hours']
/ (df_engineered['years_experience'] + 1)

# 5. Performance-experience interaction
df_engineered['performance_experience'] =
df_engineered['performance_score'] * df_engineered['years_experience']

# 6. Age groups (categorical feature from numerical)
```

```
df_engineered['age_group'] = pd.cut(df_engineered['age'],
                                    bins=[0, 30, 40, 50, 100],
                                    labels=['Young', 'Mid-Career',
'Experienced', 'Senior'])

# 7. Experience level (categorical feature from numerical)
df_engineered['experience_level'] =
pd.cut(df_engineered['years_experience'],
                                           bins=[0, 2, 5, 10, 100],
                                           labels=['Novice',
'Intermediate', 'Experienced', 'Expert'])

# 8. High performer flag
df_engineered['high_performer'] = (df_engineered['performance_score']
> df_engineered['performance_score'].quantile(0.75)).astype(int)

print("New features created:")
new_features = ['experience_age_ratio', 'productivity_score',
'work_intensity',
                'training_investment', 'performance_experience',
'age_group',
                'experience_level', 'high_performer']
for feature in new_features:
    print(f"  • {feature}")

print(f"\nDataset shape after feature engineering:
{df_engineered.shape}")
```

```
Creating new features...
New features created:
  • experience_age_ratio
  • productivity_score
  • work_intensity
  • training_investment
  • performance_experience
  • age_group
  • experience_level
  • high_performer

Dataset shape after feature engineering: (1003, 32)
```

```
# Analyze the new features
fig, axes = plt.subplots(2, 4, figsize=(20, 10))
axes = axes.ravel()

numerical_new_features = ['experience_age_ratio',
'productivity_score', 'work_intensity',
                         'training_investment',
'performance_experience']
```

```python
for i, feature in enumerate(numerical_new_features):
    axes[i].hist(df_engineered[feature], bins=30, alpha=0.7)
    axes[i].set_title(f'{feature}')
    axes[i].set_xlabel(feature)
    axes[i].set_ylabel('Frequency')

# Age group distribution
age_group_counts = df_engineered['age_group'].value_counts()
axes[5].bar(age_group_counts.index, age_group_counts.values)
axes[5].set_title('Age Group Distribution')
axes[5].set_ylabel('Count')

# Experience level distribution
exp_level_counts = df_engineered['experience_level'].value_counts()
axes[6].bar(exp_level_counts.index, exp_level_counts.values)
axes[6].set_title('Experience Level Distribution')
axes[6].set_ylabel('Count')

# High performer distribution
high_perf_counts = df_engineered['high_performer'].value_counts()
axes[7].bar(['Regular', 'High Performer'], high_perf_counts.values)
axes[7].set_title('High Performer Distribution')
axes[7].set_ylabel('Count')

plt.tight_layout()
plt.show()
```

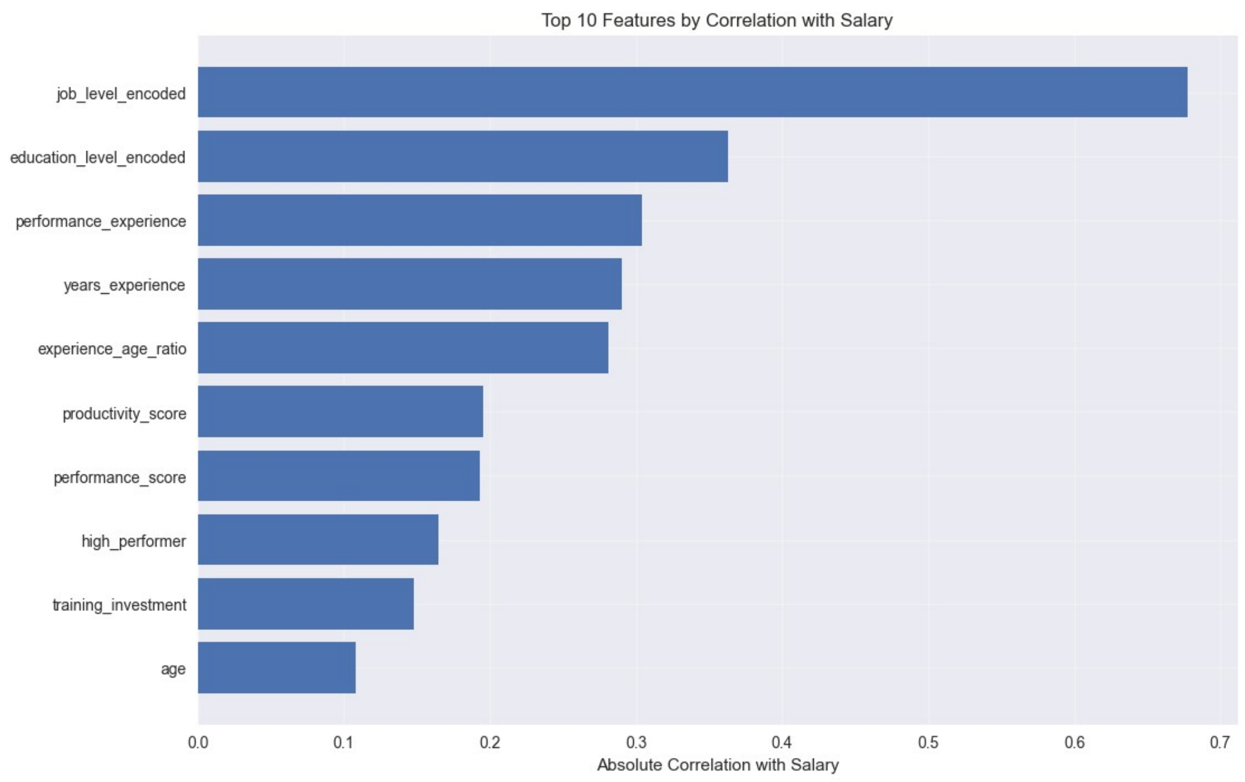


Step 7: Feature Selection

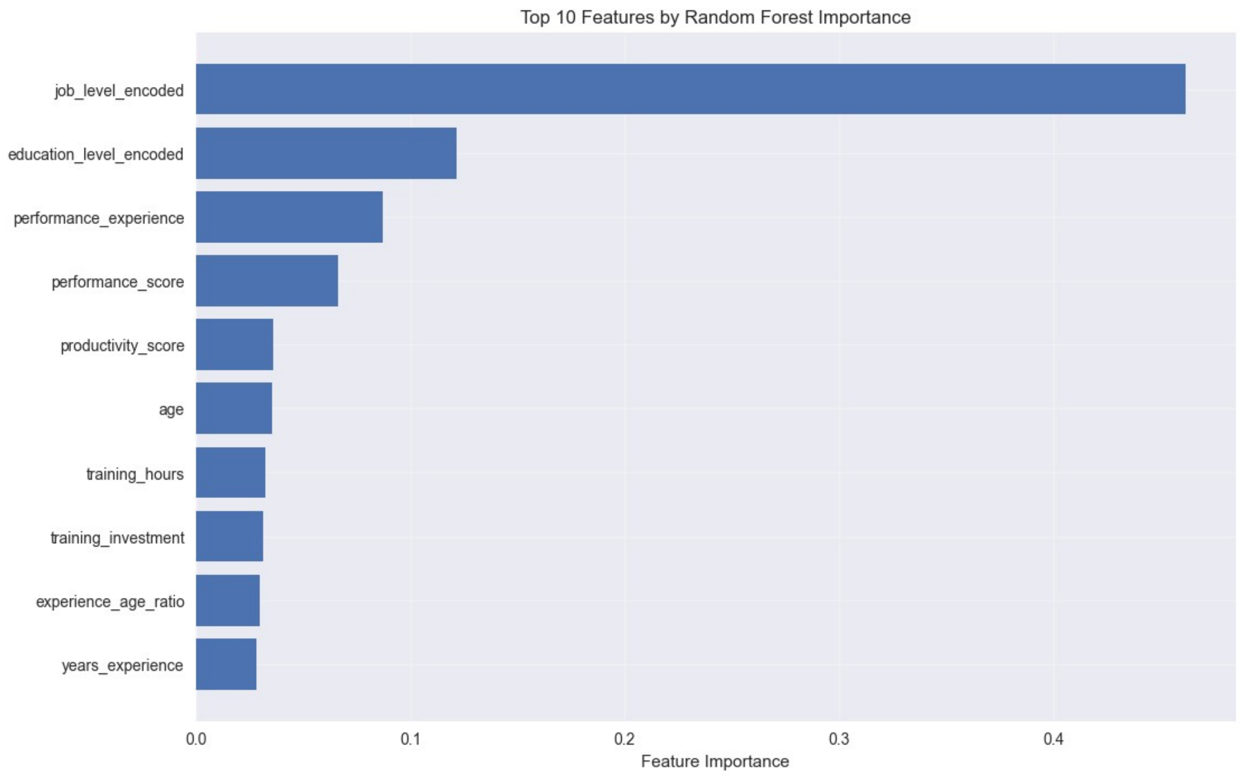
Not all features are equally important. Let's identify the most predictive ones.

```python
# Prepare data for feature importance analysis
# Select numerical features for correlation analysis
numerical_features_all =
df_engineered.select_dtypes(include=[np.number]).columns.tolist()
# Remove ID and target variable
numerical_features_all = [col for col in numerical_features_all if col
not in ['employee_id', 'salary']]

# Calculate correlation with target variable (salary)
correlations = df_engineered[numerical_features_all +
['salary']].corr()['salary'].abs().sort_values(ascending=False)
correlations = correlations.drop('salary')  # Remove self-correlation

print("Feature Correlation with Salary (absolute values):")
print(correlations.round(3))

# Visualize feature importance
plt.figure(figsize=(12, 8))
top_features = correlations.head(10)
plt.barh(range(len(top_features)), top_features.values)
plt.yticks(range(len(top_features)), top_features.index)
plt.xlabel('Absolute Correlation with Salary')
plt.title('Top 10 Features by Correlation with Salary')
plt.gca().invert_yaxis()
plt.grid(True, alpha=0.3)
plt.show()
```

```
Feature Correlation with Salary (absolute values):
job_level_encoded          0.678
education_level_encoded    0.363
performance_experience     0.304
years_experience           0.290
experience_age_ratio       0.282
productivity_score         0.196
performance_score          0.193
high_performer             0.165
training_investment        0.148
age                        0.108
training_hours             0.029
projects_completed         0.002
hours_per_week             0.002
work_intensity             0.002
Name: salary, dtype: float64
```



```python
# Use Random Forest for feature importance
# Prepare features (only numerical for this example)
X = df_engineered[numerical_features_all]
y = df_engineered['salary']

# Train a Random Forest to get feature importance
rf = RandomForestRegressor(n_estimators=100, random_state=42)
rf.fit(X, y)

# Get feature importance
feature_importance = pd.DataFrame({
    'feature': X.columns,
    'importance': rf.feature_importances_
}).sort_values('importance', ascending=False)

print("\nRandom Forest Feature Importance:")
print(feature_importance.round(4))

# Visualize feature importance
plt.figure(figsize=(12, 8))
top_rf_features = feature_importance.head(10)
plt.barh(range(len(top_rf_features)), top_rf_features['importance'])
plt.yticks(range(len(top_rf_features)), top_rf_features['feature'])
plt.xlabel('Feature Importance')
plt.title('Top 10 Features by Random Forest Importance')
plt.gca().invert_yaxis()
plt.grid(True, alpha=0.3)
plt.show()


Random Forest Feature Importance:
                    feature  importance
7          job_level_encoded      0.4617
6    education_level_encoded      0.1216
12    performance_experience      0.0871
2          performance_score      0.0664
9         productivity_score      0.0363
0                        age      0.0355
5             training_hours      0.0327
11       training_investment      0.0315
8       experience_age_ratio      0.0297
1           years_experience      0.0285
3             hours_per_week      0.0230
10            work_intensity      0.0222
4         projects_completed      0.0218
13            high_performer      0.0021
```



Step 8: Building a Preprocessing Pipeline

Let's create a reusable pipeline for all our preprocessing steps.

```python
# Create a comprehensive preprocessing pipeline
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer

# Define column groups
numerical_cols = ['age', 'years_experience', 'performance_score', 
'hours_per_week',
                 'projects_completed', 'training_hours']
categorical_cols = ['education_level', 'department', 'job_level', 
'location']

# Create preprocessing pipelines for different column types
numerical_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

categorical_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(drop='first', sparse_output=False))
])

# Combine pipelines
preprocessor = ColumnTransformer([
    ('num', numerical_pipeline, numerical_cols),
    ('cat', categorical_pipeline, categorical_cols)
])

print("Preprocessing Pipeline Created:")
print("\nNumerical Pipeline:")
print("  1. Impute missing values with median")
print("  2. Standard scaling (mean=0, std=1)")
print("\nCategorical Pipeline:")
print("  1. Impute missing values with mode")
print("  2. One-hot encoding (drop first category)")
```

```
Preprocessing Pipeline Created:

Numerical Pipeline:
  1. Impute missing values with median
  2. Standard scaling (mean=0, std=1)

Categorical Pipeline:
  1. Impute missing values with mode
  2. One-hot encoding (drop first category)
```

```python
# Test the pipeline on our data
# Start with original data (with missing values)
```

```
X_original = df[numerical_cols + categorical_cols]
y_original = df['salary']

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X_original,
y_original, test_size=0.2, random_state=42)

print(f"Training set shape: {X_train.shape}")
print(f"Test set shape: {X_test.shape}")
print(f"\nMissing values in training set:
{X_train.isnull().sum().sum()}")
print(f"Missing values in test set: {X_test.isnull().sum().sum()}")

# Apply preprocessing pipeline
X_train_processed = preprocessor.fit_transform(X_train)
X_test_processed = preprocessor.transform(X_test)

print(f"\nAfter preprocessing:")
print(f"Training set shape: {X_train_processed.shape}")
print(f"Test set shape: {X_test_processed.shape}")
print(f"Missing values in processed training set:
{np.isnan(X_train_processed).sum()}")
print(f"Missing values in processed test set:
{np.isnan(X_test_processed).sum()}")
```

```
Training set shape: (802, 10)
Test set shape: (201, 10)

Missing values in training set: 113
Missing values in test set: 37

After preprocessing:
Training set shape: (802, 20)
Test set shape: (201, 20)
Missing values in processed training set: 0
Missing values in processed test set: 0
```

```
# Get feature names after preprocessing
# Numerical features keep their names
num_feature_names = numerical_cols

# Categorical features get expanded
cat_feature_names = preprocessor.named_transformers_['cat']
['onehot'].get_feature_names_out(categorical_cols)

all_feature_names = num_feature_names + list(cat_feature_names)

print(f"Total features after preprocessing: {len(all_feature_names)}")
print("\nFeature names:")
for i, name in enumerate(all_feature_names):
    print(f"  {i+1:2d}. {name}")
```

```
Total features after preprocessing: 20

Feature names:
   1. age
   2. years_experience
   3. performance_score
   4. hours_per_week
   5. projects_completed
   6. training_hours
   7. education_level_High School
   8. education_level_Master
   9. education_level_PhD
  10. department_Finance
  11. department_HR
  12. department_Marketing
  13. department_Sales
  14. job_level_Lead
  15. job_level_Mid
  16. job_level_Senior
  17. location_Chicago
  18. location_New York
  19. location_Remote
  20. location_San Francisco
```

Challenge: Your Turn to Practice!

Now it's your turn to apply preprocessing techniques.

Challenge 1: Create a new feature

Create a feature that represents "career progression speed" (job level encoded divided by years of experience). Handle the case where years of experience is 0.

```python
# Challenge 1: Create career progression speed feature
# Hint: Use the job_level_encoded and years_experience columns

# Create career progression speed feature
# Handle the case where years_experience is 0
df_engineered['career_progression_speed'] = np.where(
    df_engineered['years_experience'] == 0,
    0,
    df_engineered['job_level_encoded'] /
df_engineered['years_experience']
)

print("Career progression speed feature created")
print(df_engineered['career_progression_speed'].describe())
```

```
Career progression speed feature created
count    1003.000000
mean        1.149563
std         5.501231
min         0.000000
25%         0.000000
50%         0.209361
75%         0.664623
max       124.090519
Name: career_progression_speed, dtype: float64
```

Challenge 2: Handle a new categorical variable

Imagine we have a new column 'work_style' with values ['Remote', 'Hybrid', 'Office']. Add appropriate preprocessing for this column to our pipeline.

```
# Challenge 2: Add a new categorical variable to the preprocessing
pipeline
# Hint: You'll need to modify the categorical_cols list and recreate
the preprocessor

# Create the new 'work_style' column
np.random.seed(42)
df_with_work_style = df.copy()
df_with_work_style['work_style'] = np.random.choice(['Remote',
'Hybrid', 'Office'],

size=len(df_with_work_style))

# Add work_style to categorical columns
numerical_cols_updated = ['age', 'years_experience',
'performance_score', 'hours_per_week',
                          'projects_completed', 'training_hours']
categorical_cols_updated = ['education_level', 'department',
'job_level', 'location', 'work_style']

# Recreate the preprocessing pipeline
numerical_pipeline_updated = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

categorical_pipeline_updated = Pipeline([
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(drop='first', sparse_output=False))
])

preprocessor_updated = ColumnTransformer([
    ('num', numerical_pipeline_updated, numerical_cols_updated),
    ('cat', categorical_pipeline_updated, categorical_cols_updated)
])

print("Updated preprocessing pipeline with work_style column")
print(f"Categorical columns: {categorical_cols_updated}")
```

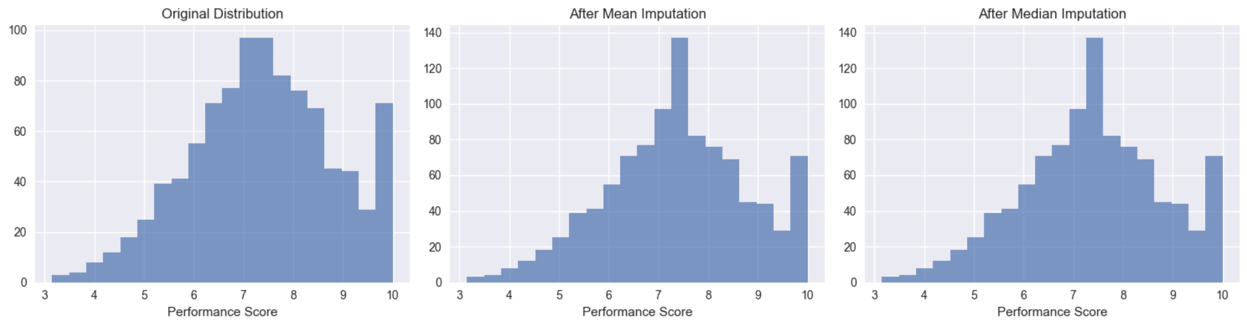
```
Updated preprocessing pipeline with work_style column
Categorical columns: ['education_level', 'department', 'job_level',
'location', 'work_style']
```

Challenge 3: Compare imputation strategies

Compare the effect of using mean vs median imputation for the 'performance_score' column. Which one preserves the distribution better?

```
# Challenge 3: Compare mean vs median imputation for performance_score
# Hint: Create two versions of the data with different imputation
strategies and compare histograms

# Create two versions with different imputation
df_mean_imputed = df.copy()
df_median_imputed = df.copy()

# Mean imputation
mean_value = df_mean_imputed['performance_score'].mean()
df_mean_imputed['performance_score'].fillna(mean_value, inplace=True)

# Median imputation
median_value = df_median_imputed['performance_score'].median()
df_median_imputed['performance_score'].fillna(median_value,
inplace=True)

# Compare distributions
fig, axes = plt.subplots(1, 3, figsize=(15, 4))

axes[0].hist(df['performance_score'].dropna(), bins=20, alpha=0.7)
axes[0].set_title('Original Distribution')
axes[0].set_xlabel('Performance Score')

axes[1].hist(df_mean_imputed['performance_score'], bins=20, alpha=0.7)
axes[1].set_title('After Mean Imputation')
axes[1].set_xlabel('Performance Score')

axes[2].hist(df_median_imputed['performance_score'], bins=20,
alpha=0.7)
axes[2].set_title('After Median Imputation')
axes[2].set_xlabel('Performance Score')

plt.tight_layout()
plt.show()
```

```
print(f"Mean imputation value: {mean_value:.3f}")
print(f"Median imputation value: {median_value:.3f}")
print("\nMedian imputation better preserves the distribution as it is
less affected by outliers")
```



```
Mean imputation value: 7.371
Median imputation value: 7.395

Median imputation better preserves the distribution as it is less
affected by outliers
```

Summary

Congratulations! You've mastered the essential data preprocessing and feature engineering techniques. Here's what you've learned:

Key Skills Mastered:

1. **Data Quality Assessment**: Identifying and understanding missing values, duplicates, and outliers
2. **Missing Value Handling**: Multiple imputation strategies (mean, median, mode, group-based)
3. **Categorical Encoding**: Ordinal encoding for ordered categories, one-hot encoding for nominal categories
4. **Feature Scaling**: StandardScaler and MinMaxScaler for different use cases
5. **Outlier Treatment**: Detection using IQR method and treatment using Winsorization
6. **Feature Engineering**: Creating new features from existing ones using domain knowledge
7. **Feature Selection**: Using correlation and Random Forest importance for feature ranking
8. **Pipeline Creation**: Building reusable preprocessing pipelines with scikit-learn

Best Practices Learned:

- Always understand your data before preprocessing
- Choose imputation strategies based on data distribution and missing patterns
- Use appropriate encoding for different types of categorical variables
- Scale features when algorithms are sensitive to feature magnitude

- Handle outliers based on domain knowledge and model requirements
- Create features that capture domain expertise
- Use pipelines to ensure reproducibility and prevent data leakage

Next Steps:

In the next lab, we'll use this preprocessed data to build and evaluate classification models, learning how to:

- Train logistic regression and decision tree classifiers
- Evaluate model performance using various metrics
- Use cross-validation for robust model assessment
- Interpret model results and make predictions

Additional Resources:

- Scikit-learn Preprocessing Guide
- Feature Engineering Techniques
- Handling Missing Data
- Pipeline Documentation