Pre-processing and Pipelines

Created	@September 30, 2024 10:24 PM
Class	Supervised Learning with scikit-learn

- Preprocessing Categorical Features
  - Key Details
  - Dummy Variables
    - Example: Music Genre
  - Implementation in Python
    - Using Pandas get_dummies
  - Model Fitting with Dummy Variables
  - Mathematical Concept: One-Hot Encoding
  - Key Takeaways
- Handling Missing Data
  - Key Details
  - Approaches to Handling Missing Data
    - Removal
    - Imputation
  - Implementation in Python
    - Using SimpleImputer
    - Using Pipeline
  - Mathematical Concept: Mean Imputation
  - Key Takeaways
- Data Preprocessing for Machine Learning
  - Key Details
  - Centering and Scaling Data
    - Importance of Scaling
    - Scaling Techniques
    - Implementing Standardization with Scikit-learn
  - Using Pipelines for Preprocessing and Modeling
    - Basic Pipeline Example
    - Importance of Scaling

Cross-validation with Pipeline
Key Takeaways
Choosing and Evaluating Machine Learning Models
Key Details
Factors Influencing Model Selection
Dataset Characteristics
Model Interpretability
Model Flexibility
Model Evaluation Techniques
Regression Metrics
Classification Metrics
Comparing Models: A Practical Approach
Example: Binary Classification of Song Genre
Key Takeaways

Preprocessing Categorical Features

Key Details

- Real-world data often contains categorical features that need preprocessing
- Scikit-learn requires numeric data with no missing values
- Categorical features must be converted to numeric features using dummy variables

Dummy Variables

- Binary features created for each category in a categorical variable
- 0 means the observation was not that category, 1 means it was
- For n categories, only n-1 dummy variables are needed to avoid duplicating information

Example: Music Genre

- 10 genres (e.g., Electronic, Hip-Hop, Rock)
- Create 9 binary features, omitting one (e.g., Rock)

- Each song has 1 in one column and 0s in the rest

Implementation in Python

Using Pandas get_dummies

```
import pandas as pd

# Read the DataFrame
df = pd.read_csv('music_data.csv')

# Create dummy variables
genre_dummies = pd.get_dummies(df['genre'], drop_first=True)

# Combine with original DataFrame
df_with_dummies = pd.concat([df, genre_dummies], axis=1)

# Remove original categorical column
df_with_dummies = df_with_dummies.drop('genre', axis=1)

# Alternative: Create dummies for entire DataFrame
music_dummies = pd.get_dummies(df, columns=['genre'], drop_fi
rst=True)
```

Model Fitting with Dummy Variables

```
from sklearn.model_selection import train_test_split, cross_v
al_score, KFold
from sklearn.linear_model import LinearRegression
import numpy as np

# Split data
X = music_dummies.drop('popularity', axis=1)
y = music_dummies['popularity']
X_train, X_test, y_train, y_test = train_test_split(X, y, tes
```

```
t_size=0.2, random_state=42)

# Create KFold object
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Instantiate model
model = LinearRegression()

# Perform cross-validation
scores = cross_val_score(model, X_train, y_train, cv=kf, scor
ing='neg_mean_squared_error')

# Calculate RMSE
rmse = np.sqrt(-scores)
print(f"Average RMSE: {rmse.mean():.2f}")
```

Mathematical Concept: One-Hot Encoding

One-hot encoding can be represented mathematically as a transformation of a categorical variable into a set of binary variables:

Let $C = \{c_1, c_2, ..., c_n\}$ be a categorical variable with n categories.

The one-hot encoding transformation $f : C \rightarrow \{0, 1\}^{n-1}$ is defined as:

$f(c_i) = (x_1, x_2, ..., x_{n-1})$ where:

- $x_j = 1$ if $i = j$
- $x_j = 0$ if $i \neq j$
- The n^{th} category is represented by all zeros $(0, 0, ..., 0)$

This transformation allows categorical data to be used in algorithms that require numerical input features.

Key Takeaways

- Categorical features must be converted to numeric features for use in scikit-learn

- Dummy variables (one-hot encoding) is a common method for this conversion
- Only n-1 dummy variables are needed for n categories to avoid multicollinearity
- Pandas' get_dummies function provides an easy way to create dummy variables
- After creating dummy variables, model fitting and evaluation proceed as usual
- Cross-validation with negative MSE is used for regression problems in scikit-learn

Handling Missing Data

Key Details

- Missing data occurs when there is no value for a feature in a particular row
- Common approaches: removal or imputation
- Data leakage must be avoided when handling missing data
- Imputers are considered transformers in scikit-learn
- Pipelines can be used to streamline the process of handling missing data and building models

Approaches to Handling Missing Data

Removal

- Remove observations with missing values if they account for less than 5% of all data

```
import pandas as pd

# Identify columns with less than 5% missing values
columns_to_keep = df.columns[df.isnull().mean() < 0.05].tolis
t()
```

```
# Remove rows with missing values in these columns
df_cleaned = df.dropna(subset=columns_to_keep)
```

Imputation

- Make educated guesses for missing values
- Common methods:
  - Numeric data: mean or median
  - Categorical data: most frequent value

Implementation in Python

Using SimpleImputer

```
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
import numpy as np

# Split data into categorical and numeric features
X_cat = df[['genre']]
X_num = df[['danceability', 'energy', 'loudness', 'speechines
s', 'acousticness', 'instrumentalness', 'liveness', 'valenc
e', 'tempo']]
y = df['popularity']

# Split into train and test sets
X_cat_train, X_cat_test, X_num_train, X_num_test, y_train, y_
test = train_test_split(
    X_cat, X_num, y, test_size=0.2, random_state=42
)

# Impute categorical data
cat_imputer = SimpleImputer(strategy='most_frequent')
```

```
X_cat_train_imputed = cat_imputer.fit_transform(X_cat_train)
X_cat_test_imputed = cat_imputer.transform(X_cat_test)

# Impute numeric data
num_imputer = SimpleImputer(strategy='mean')
X_num_train_imputed = num_imputer.fit_transform(X_num_train)
X_num_test_imputed = num_imputer.transform(X_num_test)

# Combine imputed data
X_train_imputed = np.append(X_cat_train_imputed, X_num_train_
imputed, axis=1)
X_test_imputed = np.append(X_cat_test_imputed, X_num_test_imp
uted, axis=1)
```

Using Pipeline

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import numpy as np

# Prepare data
df_cleaned = df.dropna(subset=[col for col in df.columns if d
f[col].isnull().mean() < 0.05])
df_cleaned['is_rock'] = np.where(df_cleaned['genre'] == 'Roc
k', 1, 0)

X = df_cleaned.drop(['genre', 'is_rock'], axis=1)
y = df_cleaned['is_rock']

# Create pipeline
pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')),
```

```
    ('classifier', RandomForestClassifier(random_state=42))
])

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, tes
t_size=0.2, random_state=42)

# Fit pipeline
pipeline.fit(X_train, y_train)

# Predict and evaluate
y_pred = pipeline.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

Mathematical Concept: Mean Imputation

Mean imputation can be represented mathematically as follows:

For a feature X with n non-missing values $x_1, x_2, ..., x_n$ the imputed value $\hat{x}$ for a missing entry is:

$\hat{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$

This method preserves the mean of the observed data but reduces the variance, which can lead to underestimation of standard errors and distortion of relationships between variables.

Key Takeaways

- Missing data is common in real-world datasets and must be handled before model fitting
- Removal is suitable for a small percentage of missing data (< 5%)
- Imputation methods include mean, median, and most frequent value
- Data must be split before imputation to avoid data leakage

- SimpleImputer in scikit-learn provides easy implementation of imputation methods
- Pipelines can streamline the process of data imputation and model building
- Choice of imputation method can impact model performance and should be considered carefully

Data Preprocessing for Machine Learning

Key Details

- Data imputation and centering/scaling are important preprocessing steps
- Centering and scaling help normalize feature ranges for better model performance
- Many ML models use distance metrics, so feature scale can disproportionately influence results
- Standardization is a common scaling technique

Centering and Scaling Data

Importance of Scaling

- Feature ranges can vary widely (e.g., duration_ms: 0 to 1.62 million, speechiness: decimal places, loudness: negative values)
- Models using distance metrics (e.g., KNN) are sensitive to feature scales
- Goal: Put features on similar scales to avoid disproportionate influence

Scaling Techniques

1. Standardization:
    - Subtract mean, divide by variance
    - Results in features centered around 0 with variance of 1

- Formula: $z = \frac{x-\mu}{\sigma}$
  Where
  z is the standardized value, x is the original value, μ is the mean, and σ is the standard deviation

2. Min-Max Normalization:
   - Subtract minimum, divide by range
   - Results in features ranging from 0 to 1
   - Formula: $x_{normalized} = \frac{x-x_{min}}{x_{max}-x_{min}}$
3. Feature Scaling to [-1, 1] range

Implementing Standardization with Scikit-learn

```
from sklearn.preprocessing import StandardScaler

# Create feature and target arrays
X = df[features]
y = df[target]

# Split data before scaling
X_train, X_test, y_train, y_test = train_test_split(X, y, tes
t_size=0.2, random_state=42)

# Instantiate and apply StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Verify scaling
print(X_train.mean(), X_train.std())
print(X_train_scaled.mean(), X_train_scaled.std())
```

Using Pipelines for Preprocessing and Modeling

Basic Pipeline Example

```
from sklearn.pipeline import Pipeline
from sklearn.neighbors import KNeighborsClassifier

# Create pipeline
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('knn', KNeighborsClassifier(n_neighbors=6))
])

# Fit and predict
pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_test)

# Compute accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

Importance of Scaling

- Scaled data accuracy: 0.81
- Unscaled data accuracy: 0.53
- Scaling improved accuracy by over 50%

Cross-validation with Pipeline

```
from sklearn.model_selection import GridSearchCV

# Create pipeline
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('knn', KNeighborsClassifier())
])
```

```
# Define parameter grid
parameters = {
    'knn__n_neighbors': [1, 3, 5, 7, 9, 11, 13, 15]
}

# Perform grid search
grid_search = GridSearchCV(pipeline, param_grid=parameters)
grid_search.fit(X_train, y_train)

# Make predictions
y_pred = grid_search.predict(X_test)

# Print results
print(f"Best score: {grid_search.best_score_:.2f}")
print(f"Best parameters: {grid_search.best_params_}")
```

Key Takeaways

- Centering and scaling are crucial preprocessing steps for many ML models
- Standardization is a common scaling technique that centers data around 0 with variance of 1
- Scikit-learn's `StandardScaler` can be used for easy implementation of standardization
- Pipelines allow for combining preprocessing steps with model training
- Cross-validation can be performed on pipelines to tune hyperparameters
- Scaling can significantly improve model performance (e.g., 50% improvement in KNN accuracy)

Choosing and Evaluating Machine Learning Models

Key Details

- Model selection depends on various factors
- Scikit-learn provides consistent APIs across models, facilitating comparison
- Initial model comparison can be done without hyperparameter tuning
- Data scaling is important for fair model comparison

Factors Influencing Model Selection

Dataset Characteristics

- Size of the dataset
- Number of features
- Amount of data required for model performance (e.g., Artificial Neural Networks need large datasets)

Model Interpretability

- Some situations require explainable predictions
- Example: Linear Regression coefficients can be interpreted

Model Flexibility

- Flexible models make fewer assumptions about data
- Example: KNN doesn't assume linear relationships between features and target

Model Evaluation Techniques

Regression Metrics

- Root Mean Squared Error (RMSE)
- R-squared value

Classification Metrics

- Accuracy

- Confusion Matrix and associated metrics
- ROC AUC (Receiver Operating Characteristic Area Under Curve)

Comparing Models: A Practical Approach

1. Select multiple models
2. Choose an evaluation metric
3. Scale the data (important for fair comparison)
4. Evaluate models without hyperparameter tuning

Example: Binary Classification of Song Genre

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split, cross_v
al_score, KFold
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

# Prepare data
X = df[features]
y = df[target]
X_train, X_test, y_train, y_test = train_test_split(X, y, tes
t_size=0.2, random_state=42)

# Scale features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Define models
models = {
    'KNN': KNeighborsClassifier(),
```

```
    'Logistic Regression': LogisticRegression(),
    'Decision Tree': DecisionTreeClassifier()
}

# Perform cross-validation
results = []
for model in models.values():
    kfold = KFold(n_splits=5, shuffle=True, random_state=42)
    cv_results = cross_val_score(model, X_train_scaled, y_tra
in, cv=kfold)
    results.append(cv_results)

# Visualize results
plt.boxplot(results, labels=models.keys())
plt.title('Model Comparison: Cross-validation Accuracy')
plt.show()

# Evaluate on test set
for name, model in models.items():
    model.fit(X_train_scaled, y_train)
    accuracy = model.score(X_test_scaled, y_test)
    print(f"{name} Test Accuracy: {accuracy:.4f}")
```

Key Takeaways

- Consider dataset size, interpretability needs, and flexibility when choosing models
- Scikit-learn's consistent API allows easy model comparison
- Always scale data before comparing models (especially important for KNN, linear/logistic regression)
- Cross-validation provides robust performance estimates
- Visualizing cross-validation results (e.g., with box plots) helps compare model distributions

- Test set evaluation confirms model performance on unseen data

The mathematical concept of cross-validation ties into model evaluation by providing a more robust estimate of model performance. It helps assess how well a model generalizes to unseen data by repeatedly splitting the training data into training and validation sets. The formula for K-fold cross-validation score can be expressed as:

$$CV_{score} = \frac{1}{K} \sum_{i=1}^{K} Score_i$$

Where K is the number of folds and $Score_i$ is the performance metric (e.g., accuracy) for the i^{th} fold. This approach helps to reduce the impact of data splitting variability on model evaluation.