DATA SCHOOL BY DATA WITH F&F

Course: Machine Learning Fundamentals - Linear Regression & Classification Metrics

Welcome to your first training course with DATA SCHOOL BY DATA WITH F&F! This module is designed to provide a hands-on introduction to two fundamental concepts in machine learning: Linear Regression for predictive modeling and key metrics for evaluating classification models. By the end of this course, you will be able to build, train, and evaluate your own machine learning models using Python.

Section 1: Linear Regression

Linear regression is a foundational algorithm in machine learning used to predict a continuous outcome variable (like price, sales, or salary) based on one or more predictor variables. In this section, we will walk through two practical examples.

Problem 1: Predicting a Target Value from Multiple Features

In this first problem, we'll generate a synthetic dataset to understand the core components of a linear regression model, from data creation to model evaluation.

Step 1: Import Libraries and Generate Data

First, we import the necessary Python libraries. We use `numpy` and `pandas` for data manipulation, `sklearn` for building and evaluating our model, and `matplotlib` and `seaborn` for visualization. We then generate a sample dataset with four input features and a target variable `y` that has a linear relationship with the features, plus some random noise to simulate a real-world scenario.

```
# Import necessary libraries
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import seaborn as sns

# Generate a sample dataset
np.random.seed(42)
data_size = 100
X = pd.DataFrame({
 'Feature1': np.random.rand(data_size),
```

```
 'Feature2': np.random.rand(data_size) * 2,
 'Feature3': np.random.rand(data_size) * 3,
 'Feature4': np.random.rand(data_size) * 4
})
y = 3.5*X['Feature1'] + 2.5*X['Feature2'] + 1.5*X['Feature3'] + X['Feature4'] +
np.random.normal(0, 0.2, data_size)
```

Step 2: Split Data, Scale Features, and Train the Model

We split our data into training (80%) and testing (20%) sets. Feature scaling (StandardScaler) is applied to normalize the features, which helps the model converge faster. We then initialize and train our LinearRegression model using the training data.

```
# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Feature scaling
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Initialize and train the linear regression model
model = LinearRegression()
model.fit(X_train, y_train)
```

Step 3: Make Predictions and Evaluate the Model

Using the trained model, we predict the target values for our test set. We then evaluate the model's performance using three key metrics:

- **Mean Squared Error (MSE):** The average of the squared differences between actual and predicted values.
- **Mean Absolute Error (MAE):** The average of the absolute differences between actual and predicted values.
- **R-squared () Score:** Indicates the proportion of the variance in the dependent variable that is predictable from the independent variable(s).

```
# Predict using the test set
y_pred = model.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

```
print("Model Performance:")
print(f"Mean Squared Error: {mse}")
print(f"Mean Absolute Error: {mae}")
print(f"R-squared Score: {r2}")
```

Output:

```
Model Performance:
Mean Squared Error: 0.03842349095532515
Mean Absolute Error: 0.1601611181153835
R-squared Score: 0.9950153359149303
```

Step 4: Visualize Results

Visualizations help us understand model performance. We plot the actual vs. predicted values to see how well our predictions align. We also plot the distribution of residuals (the errors between actual and predicted values); a normal distribution centered around zero indicates a good model fit.

```
# Plotting predictions vs actual values
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred, color="blue", edgecolor="w", s=100, alpha=0.6)
plt.plot([y.min(), y.max()], [y.min(), y.max()], "r--", lw=2)
plt.xlabel("Actual values")
plt.ylabel("Predicted values")
plt.title("Actual vs Predicted Values")
plt.show()

# Plotting residuals
residuals = y_test - y_pred
plt.figure(figsize=(10, 6))
sns.histplot(residuals, kde=True, color="orange", bins=20)
plt.title("Residual Distribution")
plt.xlabel("Residuals")
plt.ylabel("Frequency")
plt.show()
```

Problem 2: Salary Prediction using Multiple Regression

In this problem, we'll predict salary based on categorical and numerical data, introducing preprocessing pipelines to handle mixed data types.

Step 1: Create Dataset and Define Features

We create a dataset containing an employee's 'Domain', 'Experience', 'Education', and 'Salary'. Our goal is to predict Salary using the other three features.

```
# Create a sample dataset
data_size = 100
data = pd.DataFrame({
 'Domain': np.random.choice(['IT', 'Finance', 'Healthcare', 'Education'], data_size),
 'Experience': np.random.randint(1, 21, data_size),
 'Education': np.random.choice(['Bachelors', 'Masters', 'PhD'], data_size),
 'Salary': np.random.normal(50000, 10000, data_size) + np.random.randint(1, 21, data_size) *
3000
})

# Define features (X) and target (y)
X = data[['Domain', 'Experience', 'Education']]
y = data['Salary']
```

Step 2: Create a Preprocessing and Modeling Pipeline

Since our data contains both categorical ('Domain', 'Education') and numerical ('Experience') features, we need to preprocess them. We use OneHotEncoder to convert categorical features into a numerical format. A Pipeline is created to chain these steps: preprocessing, scaling, and regression.

```
# Preprocessing for categorical features
preprocessor = ColumnTransformer(
 transformers=[
 ('cat', OneHotEncoder(drop='first'), ['Domain', 'Education']),
 ('num', 'passthrough', ['Experience'])
 ]
)

# Create a pipeline with scaling and regression
model = Pipeline(steps=[
 ('preprocessor', preprocessor),
 ('scaler', StandardScaler()),
 ('regressor', LinearRegression())
])
```

Step 3: Train, Predict, and Evaluate

We split the data, train our pipeline, and make predictions on the test set. Finally, we evaluate the model using MSE, MAE, and R-squared.

```
# Split the dataset and train the model
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
model.fit(X_train, y_train)

# Predict and evaluate
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("Model Performance:")
print(f"Mean Squared Error: {mse}")
print(f"Mean Absolute Error: {mae}")
print(f"R-squared Score: {r2}")
```

Output:

```
Model Performance:
Mean Squared Error: 111394553.79973882
Mean Absolute Error: 8645.698501193387
R-squared Score: 0.1764653556108571
```

Note: The lower R-squared score here is expected, as our synthetic salary data has more randomness compared to the first problem.

Step 4: Predict on New Data

We can use our trained model to predict the salary for a new, unseen data point.

```
# Prediction on new input data
new_data = pd.DataFrame({
 'Domain': ['IT'],
 'Experience': [10],
 'Education': ['Masters']
})
predicted_salary = model.predict(new_data)
print(f"Predicted Salary for new data {new_data.iloc[0].to_dict()}: ${predicted_salary[0]:.2f}")
```

Output:

```
Predicted Salary for new data {'Domain': 'IT', 'Experience': 10, 'Education': 'Masters'}: $72763.53
```

Section 2: Classification Metrics

When the prediction task is to assign a category or class (e.g., "High Salary" vs. "Low Salary"), we use different metrics to evaluate the model. This section covers the most common classification metrics.

Problem: Predicting High vs. Low Salary

We will use the same salary dataset but transform it into a classification problem: predicting whether an employee has a "High Salary."

Step 1: Prepare Data for Classification

We define a salary threshold to create a binary target variable, `HighSalary`. If `Salary` is above the threshold, `HighSalary` is 1; otherwise, it is 0. We then build a pipeline using `LogisticRegression`, a common algorithm for classification.

```
# Create a binary target variable
salary_threshold = 70000
data['HighSalary'] = (data['Salary'] >= salary_threshold).astype(int)

# Define features (X) and target (y)
X = data[['Domain', 'Experience', 'Education']]
y = data['HighSalary']

# Create a pipeline with scaling and logistic regression
model = Pipeline(steps=[
 ('preprocessor', preprocessor), # Re-using the preprocessor from the previous problem
 ('scaler', StandardScaler()),
 ('classifier', LogisticRegression())
])
```

Step 2: Train the Model and Make Predictions

We split the data, train the logistic regression model, and predict the classes for the test set.

```
# Split data, train, and predict
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

Step 3: Evaluate Using Classification Metrics

We evaluate the model using the following metrics:

- **Accuracy:** The proportion of correct predictions out of all predictions.
- **Precision:** Of all the positive predictions made by the model, how many were actually correct.
- **Recall (Sensitivity):** Of all the actual positive cases, how many did the model correctly identify.
- **F1-Score:** The harmonic mean of Precision and Recall, providing a single score that balances both.

```
# Calculate evaluation metrics
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix

accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print("Model Performance Metrics:")
print(f"Accuracy: {accuracy:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1-Score: {f1:.2f}")
```

Output:

```
Model Performance Metrics:
Accuracy: 0.75
Precision: 0.50
Recall: 0.25
F1-Score: 0.33
```

Step 4: Visualize with a Confusion Matrix

A **Confusion Matrix** is a table that summarizes the performance of a classification model. It shows the number of True Positives, True Negatives, False Positives, and False Negatives.

```
# Confusion matrix for visualization
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix")
plt.show()
```

This concludes our introductory module. Congratulations on building and evaluating both regression and classification models!