Introduction (fairness.py)

Fairness is an essential consideration in modern machine learning, particularly in domains like credit scoring, hiring, and healthcare, where biased decisions can have significant societal impacts. This code provides a simple yet effective tool to analyze fairness across different sensitive features using the Fairlearn library.

The FairnessAnalyzer class is designed to evaluate models by computing key fairness metrics, such as selection rate, demographic parity difference, and equalized odds difference. These metrics measure whether outcomes are distributed fairly across groups defined by sensitive attributes (e.g., gender, race, or age). The results are presented in a structured table, allowing practitioners to quickly assess potential biases in their models.

Full Explanation of the Code

Importing Libraries

```
from fairlearn.metrics import (
    demographic_parity_difference,
    equalized_odds_difference,
    selection_rate
)
import pandas as pd
```

- **fairlearn.metrics** → provides fairness-related evaluation metrics.
    - selection_rate → measures how often individuals in a sensitive group are given a positive outcome (e.g., loan approved).
    - demographic_parity_difference → measures the difference in selection rates between groups (should be small for fairness).
    - equalized_odds_difference → measures whether groups have similar true positive and false positive rates (a stricter fairness condition).
- **pandas** → used to store and display results in a structured table.

Defining the Class

```
class FairnessAnalyzer:
    def __init__(self, sensitive_features):
        self.sensitive_features = sensitive_features
```

- Defines a class FairnessAnalyzer.
- The constructor __init__ takes a list of **sensitive features** (for example: ['gender', 'race']).

- These are the features across which fairness metrics will be computed.

Analysis Method

```
def analyze(self, y_true, y_pred):
    results = {}

    for feature in self.sensitive_features:
        sr = selection_rate(y_true, y_pred, sensitive_features=feature)
        dpd = demographic_parity_difference(y_true, y_pred, sensitive_features=feature)
        eod = equalized_odds_difference(y_true, y_pred, sensitive_features=feature)

        results[feature] = {
            'selection_rate': sr,
            'demographic_parity_difference': dpd,
            'equalized_odds_difference': eod
        }
    return pd.DataFrame(results).T
```

Step-by-step:

1. **results = {}** → creates an empty dictionary to store fairness metrics for each sensitive feature.
2. **Loop over features**: For each sensitive feature (e.g., gender), it calculates fairness metrics:
    - selection_rate → the proportion of positive outcomes (e.g., how many females get approved compared to males).
    - demographic_parity_difference → measures disparity in positive outcome rates between groups (closer to 0 is fairer).
    - equalized_odds_difference → measures disparities in error rates (false positives/negatives) between groups.
3. **Store results** in a dictionary where keys are feature names and values are metric results.
4. **Convert dictionary to DataFrame** (pd.DataFrame(results).T) for a clean, table-like result.

Introduction (optimize.py)

Hyperparameter optimization plays a crucial role in building robust machine learning models. Traditional approaches such as grid search and random search are computationally expensive and often fail to efficiently explore large search spaces. Optuna, a state-of-the-art hyperparameter optimization framework, leverages efficient search strategies like the Tree-structured Parzen Estimator (TPE) to find optimal hyperparameters faster.

This code integrates **Optuna** with **XGBoost**, one of the most powerful gradient boosting algorithms for tabular data. By defining an objective function and running multiple trials, the framework automatically tunes hyperparameters to maximize the model's performance, measured by the **ROC-AUC score**. Additionally, a custom preprocessing class (LoanDataPreprocessor) is applied to ensure data is properly prepared before training.

Code Explanation

1. Importing Libraries

```
import optuna

import xgboost as xgb

from sklearn.metrics import roc_auc_score

from ..data.preprocessing import LoanDataPreprocessor
```

- **optuna** → for automatic hyperparameter optimization.
- **xgboost** → provides the XGBClassifier for binary classification tasks.
- **roc_auc_score** → metric that measures the model's ability to distinguish between classes.
- **LoanDataPreprocessor** → a custom preprocessing pipeline (likely handles missing values, scaling, encoding, etc.).

---

2. Defining the Objective Function

```
def objective(trial, X_train, y_train, X_val, y_val):

  params = {

    'objective': 'binary:logistic',

    'eval_metric': 'auc',

    'booster': trial.suggest_categorical('booster', ['gbtree', 'gblinear', 'dart']),

    'lambda': trial.suggest_float('lambda', 1e-8, 1.0, log=True),

    'alpha': trial.suggest_float('alpha', 1e-8, 1.0, log=True),

    'max_depth': trial.suggest_int('max_depth', 3, 10),
```

```
    'eta': trial.suggest_float('eta', 0.01, 0.3),
    'gamma': trial.suggest_float('gamma', 0, 1),
    'subsample': trial.suggest_float('subsample', 0.5, 1.0),
    'colsample_bytree': trial.suggest_float('colsample_bytree', 0.5, 1.0),
    'min_child_weight': trial.suggest_int('min_child_weight', 1, 10),
    'n_estimators': trial.suggest_int('n_estimators', 100, 1000)
}
```

- The **trial object** suggests different hyperparameter values during each optimization round.
- Key hyperparameters tuned:
  - booster: type of boosting (tree-based, linear, or DART).
  - lambda & alpha: regularization terms to avoid overfitting.
  - max_depth: maximum tree depth.
  - eta: learning rate.
  - gamma: minimum loss reduction for further splits.
  - subsample & colsample_bytree: fraction of samples/features used for training.
  - min_child_weight: minimum data in a child node (controls overfitting).
  - n_estimators: number of boosting rounds.

---

3. Data Preprocessing

```
preprocessor = LoanDataPreprocessor()
X_train_processed = preprocessor.fit_transform(X_train)
X_val_processed = preprocessor.transform(X_val)
```

- Initializes the preprocessing pipeline.
- Fits transformations on training data, applies the same transformations to validation data.
- Ensures consistent feature engineering across datasets.

---

4. Model Training

```
model = xgb.XGBClassifier(**params)
model.fit(X_train_processed, y_train,
          eval_set=[(X_val_processed, y_val)],
```

early_stopping_rounds=50,

verbose=False)

- Creates an **XGBoost classifier** with the suggested hyperparameters.
- Uses **early stopping** (50 rounds without improvement) to prevent overfitting.
- Evaluation happens on the validation set during training.

---

5. Model Evaluation

y_pred = model.predict_proba(X_val_processed)[:, 1]

auc = roc_auc_score(y_val, y_pred)

return auc

- Predicts probabilities for the positive class.
- Computes **ROC-AUC score**, which is the metric being optimized.
- Returns the score to Optuna.

---

6. Running the Optimization

def run_optimization(X_train, y_train, X_val, y_val, n_trials=50):

study = optuna.create_study(direction='maximize',
sampler=optuna.samplers.TPESampler(seed=42))

study.optimize(lambda trial: objective(trial, X_train, y_train, X_val, y_val),

n_trials=n_trials)

return study.best_params

- Creates an Optuna study with **TPE sampler** (efficient Bayesian optimization strategy).
- Direction is set to **maximize** since we want to maximize ROC-AUC.
- Runs multiple trials (n_trials=50 by default).
- Returns the best hyperparameters found.

Conclusion

This implementation showcases how **Optuna** can be combined with **XGBoost** and a custom preprocessing pipeline to efficiently optimize hyperparameters. By leveraging automated trial suggestions and evaluation, the framework significantly reduces the time and effort required compared to manual tuning.

The use of **ROC-AUC score** ensures that the model is evaluated on its ability to separate classes, which is particularly important in imbalanced datasets such as loan default prediction. The inclusion of **early stopping** prevents overfitting and enhances generalization performance.

Overall, this pipeline provides a robust and scalable solution for hyperparameter tuning in financial risk modeling and other binary classification tasks.

Introduction (preprocessing.py)

Data preprocessing is a critical step in machine learning pipelines, particularly for financial datasets such as loan applications. Raw data often contains missing values, inconsistent formats, and a mix of numerical and categorical variables, which can negatively impact model performance.

The LoanDataPreprocessor class provides a clean and reusable preprocessing framework. It ensures that numerical and categorical features are properly imputed, scaled, and encoded before being passed into a machine learning model. This not only standardizes the data but also helps improve model accuracy, stability, and fairness.

Code Explanation

1. Importing Libraries

```
import pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
```

- **pandas** → handles structured tabular data.
- **ColumnTransformer** → applies different preprocessing steps to different column groups (numerical vs categorical).
- **Pipeline** → chains multiple transformations into a single step.
- **SimpleImputer** → fills missing values (with median for numbers, most frequent for categories).
- **StandardScaler** → standardizes numerical values (mean = 0, standard deviation = 1).
- **OneHotEncoder** → converts categorical features into binary indicator variables.

2. Defining the Preprocessor Class

```
class LoanDataPreprocessor:
    def __init__(self):
        self.numeric_features = ['age', 'income', 'employment_length', 'credit_score',
                                 'debt_to_income', 'loan_amount', 'loan_term', 'recent_inquiries']
        self.categorical_features = ['home_ownership', 'loan_purpose']
        self.preprocessor = self._create_preprocessor()
```

- Initializes the class.
- Defines **numeric features** (continuous variables like age, income, loan amount, etc.).
- Defines **categorical features** (qualitative variables such as home ownership status, loan purpose).
- Calls _create_preprocessor() to build the full preprocessing pipeline.

---

3. Building the Preprocessing Pipelines

```
def _create_preprocessor(self):
    numeric_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='median')),
        ('scaler', StandardScaler())])

    categorical_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='most_frequent')),
        ('onehot', OneHotEncoder(handle_unknown='ignore'))])

    return ColumnTransformer(
        transformers=[
            ('num', numeric_transformer, self.numeric_features),
            ('cat', categorical_transformer, self.categorical_features)])
```

- **Numeric pipeline**:
    - Missing values replaced with **median** (robust to outliers).
    - Features standardized with **StandardScaler**.

- **Categorical pipeline**:
    - Missing values filled with **most frequent** category.
    - Features encoded with **OneHotEncoder** (ignores unseen categories at test time).
- **ColumnTransformer** combines both pipelines, applying them only to the relevant features.

---

4. Applying the Preprocessor

```
def fit_transform(self, X):
    return self.preprocessor.fit_transform(X)


def transform(self, X):
    return self.preprocessor.transform(X)
```

- **fit_transform**: Fits the preprocessing pipeline on the training dataset and transforms it.
- **transform**: Applies the same transformations to new/unseen data (e.g., validation or test sets).
- Ensures consistent feature treatment across datasets.

Conclusion

The LoanDataPreprocessor class provides a robust and modular framework for preparing loan application data. By handling missing values, standardizing numeric variables, and encoding categorical variables, it ensures that the dataset is clean and suitable for machine learning models.

This preprocessing pipeline is especially useful when combined with automated hyperparameter tuning and fairness analysis, as it guarantees that the model receives consistent and unbiased input data. Ultimately, this approach improves model performance and reliability in financial risk prediction tasks.

Introduction (test_models.py)

Unit testing is essential to guarantee the reliability of machine learning pipelines. This code provides a **pytest-based test** for verifying that the model training function (train_model) works correctly when given structured loan application data.

By using a small synthetic dataset, the test ensures that the training process returns a valid model and meaningful evaluation metrics, such as ROC AUC. The use of fixtures allows test data to be reusable, keeping the test modular and clean.

Code Explanation

1. Importing Libraries

```
import pytest
import pandas as pd
import numpy as np
from src.models.train import train_model
from src.data.preprocessing import LoanDataPreprocessor
```

- **pytest** → testing framework that makes it easy to write small tests.
- **pandas, numpy** → used for creating test data.
- **train_model** → the training function (imported from your src/models/train.py).
- **LoanDataPreprocessor** → the preprocessing pipeline (though not directly used here, but could be part of train_model).

---

2. Creating Sample Test Data

```
@pytest.fixture
def sample_data():
    data = {
        'age': [25, 30, 35],
        'income': [50000, 60000, 70000],
        'credit_score': [650, 700, 750],
        'debt_to_income': [0.2, 0.3, 0.4],
        'home_ownership': ['Rent', 'Own', 'Mortgage'],
        'defaulted': [0, 1, 0]
    }
```

return pd.DataFrame(data)

- Defines a **pytest fixture** called sample_data.
- This creates a small synthetic dataset with:
  - Numeric features (age, income, credit_score, debt_to_income).
  - Categorical feature (home_ownership).
  - Target variable (defaulted) indicating loan default (0 = no, 1 = yes).
- The fixture ensures test data is reusable in different test functions.

---

3. Testing Model Training

```
def test_model_training(sample_data):
    X = sample_data.drop('defaulted', axis=1)
    y = sample_data['defaulted']

    model, metrics = train_model(X, y)

    assert model is not None
    assert 'roc_auc' in metrics
    assert metrics['roc_auc'] >= 0.5
```

- **Splitting features/labels**:
  - X = input features.
  - y = target variable (defaulted).
- **Calls train_model(X, y)**: trains the ML model and returns the trained model along with evaluation metrics.
- **Assertions (unit tests)**:
  - model is not None → checks if a model was successfully trained.
  - 'roc_auc' in metrics → ensures that the training function returns ROC AUC as a metric.
  - metrics['roc_auc'] >= 0.5 → ensures the model performs at least better than random guessing.

Conclusion

This test demonstrates how **automated testing can validate machine learning workflows**. By checking that a trained model is produced and achieves a minimum ROC AUC score, it safeguards against silent failures and ensures basic predictive performance.

In practice, extending such tests with additional metrics, cross-validation checks, and edge cases (e.g., missing values, imbalanced data) would further improve the robustness and reliability of the overall credit risk modeling pipeline.