Name : Kulboboev Shukhrat

Neptun code : JDN3HW

Task : Competition result report

Computer Science Competition Report

Introduction

This report shows the data processing, feature engineering, and model training pipeline used to develop a predictive model for electricity load . The workflow includes steps to preprocess the data, engineer features, handle missing values and imbalances, and optimize model performance through hyperparameter tuning and calibration.

Step 1: Data analysis and data importing

The dataset is loaded using pandas, followed by exploratory data analysis:

- **Dataset Overview**: Using info() and describe() to understand the structure and summary statistics.
- **Missing Values Analysis:** Using isna().sum() to identify columns with missing values.
- **Time-Series Trends:** Visualizing electricity price trends (hupx) across hours and seasons with a line plot.
- **Correlation Matrix:** Computing and visualizing the correlation matrix with a heatmap to identify relationships between features.

Step 2: Data Preprocessing

The preprocess_data function splits the data into training and prediction datasets based on the day_in_period column:

- **Training Data:** Rows where day_in_period is not equal to 4.
- **Prediction Data:** Rows where day_in_period equals 4.

Missing values in both datasets are filled with their respective column means to ensure continuity in analysis and model performance.

```
def preprocess_data(data):
    prediction_data = data[data['day_in_period'] == 4]
    training_data = data[data['day_in_period'] != 4]
    training_data.fillna(training_data.mean(), inplace=True)
    prediction_data.fillna(prediction_data.mean(), inplace=True)
    return training_data, prediction_data
```

Step 3: Feature Engineering

The feature_engineering function creates additional features to enhance model predictive power. Key operations include:

- **Rolling Statistics:** Rolling mean and standard deviation for key features (e.g., ke, hupx).
- **Lag Features:** Adding one-step lag values for selected features.
- **Interaction Features:** A normalized load metric (normalized_load) for load prediction.
- **Cyclical Encoding:** Sinusoidal transformations for hour and minute to capture periodicity.

```
# Step 3: Feature Engineering
def feature_engineering(data):
    # Columns for feature engineering
    rolling_features = ['ke', 'hupx', 'afrr_fel', 'afrr_le', 'mfrr_fel', 'mfrr_le', 'afrr']

    # Add rolling mean and standard deviation
    for feature in rolling_features:
        data[f'{feature}_rolling_mean'] = data.groupby('periodID')[feature].transform(
            lambda x: x.rolling(window=4, min_periods=1).mean())
        data[f'{feature}_rolling_std'] = data.groupby('periodID')[feature].transform(
            lambda x: x.rolling(window=4, min_periods=1).std())

    # Add lag features
    for feature in rolling_features:
        data[f'{feature}_lag_1'] = data.groupby('periodID')[feature].shift(1)

    # Add interaction feature: normalized load
    data['normalized_load'] = data['rendszerterheles_terv'] / (data['afrr_fel'] + 1)

    # Add cyclical features for time
    data['hour_sin'] = np.sin(2 * np.pi * data['hour'] / 24)
    data['hour_cos'] = np.cos(2 * np.pi * data['hour'] / 24)
    data['minute_sin'] = np.sin(2 * np.pi * data['minute'] / 60)
    data['minute_cos'] = np.cos(2 * np.pi * data['minute'] / 60)

    return data
```

Step 4: Define Features and Target

To prepare the data for model training:

- Features like rowID, season, and periodID are excluded.
- target_flag is set as the target variable.

Step 5: Feature Normalization

Features are normalized using StandardScaler to standardize the dataset for improved model performance.

Step 6: Handle Missing Values

Missing values in the training set are imputed using the mean strategy with SimpleImputer.

Step 7: Address Data Imbalance with SMOTE

The SMOTE algorithm is applied to address class imbalance in the target variable by generating synthetic samples.

```
# Step 7: Handle Imbalance using SMOTE
sm = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = sm.fit_resample(X_train_scaled_imputed, y_train)
```

Step 8: Model Optimization with GridSearchCV

Hyperparameter tuning is performed on a RandomForestClassifier using GridSearchCV with a parameter grid optimizing for AUC-ROC score.

```
# Step 8: Hyperparameter Tuning with GridSearchCV
param_grid = {
    'n_estimators': [100, 300, 500],
    'max_depth': [7, 9, 11],
    'min_samples_split': [2, 5, 10],
    'max_features': ['sqrt', 'log2']
}
```

Step 9: Model Evaluation

The best model from GridSearch is evaluated on a validation split. The AUC score is computed to assess performance.

```
# Step 9: Evaluate on Validation Set
val_predictions = best_model.predict_proba(X_val_split)[:, 1]
auc_score = roc_auc_score(y_val_split, val_predictions)
print(f"Validation AUC Score after Tuning: {auc_score:.4f}")
```

Step 10: Probability Calibration

To improve probability estimates, the model undergoes isotonic calibration using CalibratedClassifierCV.

Step 11: Test Predictions and Submission

The trained and calibrated model predicts probabilities on the test dataset. The predictions are saved in a CSV file for submission.

```
# Step 11: Prepare Test Predictions
X_test = prediction_data.drop(columns=features_to_drop)
X_test_scaled = scaler.transform(X_test)
test_predictions = calibrated_model.predict_proba(X_test_scaled)[:, 1]
```

Conclusion

I developed the model from scratch based on above steps and predcited NAN values on the given datasset.The result are uploaded to Kaggle platform.