Trabalho Final - Machine Learning

AUTHOR
Rennan Dalla Guimarães

1. Introdução e Configuração

Primeiro, importamos todas as bibliotecas necessárias e definimos algumas configurações globais.

```
# Imports e Configurações Globais
# ------------------------------------------------------------------------
import warnings, pathlib, sys, time, joblib
from copy import deepcopy

warnings.filterwarnings("ignore")

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import pyreadr
import ydata_profiling
from scipy.stats import ks_2samp

from sklearn.model_selection import (
    train_test_split,
    StratifiedKFold,
    RandomizedSearchCV,
)
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    roc_auc_score,
    RocCurveDisplay,
    PrecisionRecallDisplay,
)
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier

from imblearn.pipeline import Pipeline as ImbPipeline
from imblearn.over_sampling import SMOTE
from feature_engine.outliers import Winsorizer
from lightgbm import LGBMClassifier
from xgboost import XGBClassifier
```

Upgrade to ydata-sdk

Improve your data and profiling with ydata-sdk, featuring data quality scoring, redundancy detection, outlier identification, text validation, and synthetic data generation.

2. Funções Auxiliares

Definimos um dicionário de métricas e uma função para avaliar os modelos durante a validação cruzada. Isso nos ajuda a manter o código limpo e a calcular de forma consistente a performance de cada modelo.

```
METRICS = {
    "accuracy": accuracy_score,
    "precision": precision_score,
    "recall": recall_score,
    "f1": f1_score,
    "roc_auc": roc_auc_score,
    "ks": lambda y_true, y_prob: ks_2samp(y_prob[y_true == 1], y_prob[
}

def evaluate_cv(model, X, y, cv):
    """Return DataFrame with metric values for each CV split."""
    rows = []
    for train_idx, val_idx in cv.split(X, y):
        X_tr, X_val = X.iloc[train_idx], X.iloc[val_idx]
        y_tr, y_val = y.iloc[train_idx], y.iloc[val_idx]

        model.fit(X_tr, y_tr)
        y_pred = model.predict(X_val)
        y_prob = model.predict_proba(X_val)[:, 1]

        rows.append(
            {
                m: f(y_val, y_pred if m not in ["roc_auc", "ks"] else
                for m, f in METRICS.items()
            }
        )
    return pd.DataFrame(rows)
```

3. Carga e Limpeza dos Dados

Carregamos o dataset, que está em formato `.rds`. Em seguida, limpamos os nomes das colunas e os valores das variáveis categóricas para garantir consistência e compatibilidade com os modelos. A variável alvo `Good_loan` é convertida para um formato numérico (0 e 1).

```
DATA_PATH = pathlib.Path("german.rds")
if not DATA_PATH.exists():
    sys.exit("ERROR: german.rds not found.")

result = pyreadr.read_r(DATA_PATH)
german = result[None]
german.columns = german.columns.str.replace("[^0-9a-zA-Z]+", "_", rege
```

```
for col in german.select_dtypes(exclude=["number"]).columns:
    if col != "Good_loan":
        german[col] = german[col].astype(str).str.replace(r"[\[\]<]", '

german["Good_loan"] = german["Good_loan"].astype("category").cat.codes

X = german.drop(columns=["Good_loan"])
y = german["Good_loan"]
```

4. Análise Exploratória de Dados (EDA)

Geramos um relatório de profiling dos dados para entender suas características. O relatório é salvo como um arquivo HTML.

```
# Este bloco está marcado como não-executável para evitar gerar o rela
# Remova o "eval: false" acima se quiser executá-lo.
print("=== Gerando relatório de Análise Exploratória (EDA) ===")
profile = ydata_profiling.ProfileReport(
    german, title="German Credit Data Profiling", minimal=True
)
profile.to_file("german_credit_eda_report.html")
print("--> Relatório salvo em german_credit_eda_report.html\n")
```

5. Divisão em Treino e Teste

Dividimos os dados em conjuntos de treino e teste, mantendo a proporção da variável alvo em ambos os conjuntos (`stratify=y`).

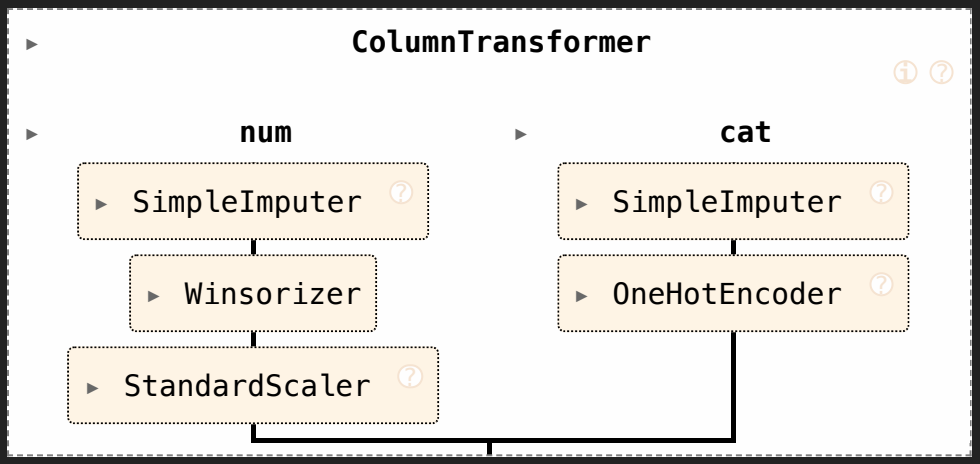
```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=17, stratify=y
)

num_cols = X_train.select_dtypes(include=["number"]).columns
cat_cols = X_train.select_dtypes(exclude=["number"]).columns
```

6. Pipeline de Pré-processamento

Construímos pipelines separados para variáveis numéricas e categóricas, que são então combinados em um único `ColumnTransformer`.

```
winsor = Winsorizer(capping_method="quantiles", tail="both", fold=0.01

num_pipeline = Pipeline(
    steps=[
        ("imputer", SimpleImputer(strategy="median")),
        ("winsor", winsor),
        ("scaler", StandardScaler()),
    ]
)

cat_pipeline = Pipeline(
    steps=[
```

```
        ("imputer", SimpleImputer(strategy="most_frequent")),
        ("onehot", OneHotEncoder(handle_unknown="ignore", sparse_outpu
    ]
)

preprocessor = ColumnTransformer(
    [("num", num_pipeline, num_cols), ("cat", cat_pipeline, cat_cols)]
)
preprocessor.set_output(transform="pandas")
```



7. Definição dos Modelos e Otimização

Definimos os 5 modelos que serão testados. Para cada um, criamos um pipeline que inclui o pré-processamento, o `SMOTE` para balanceamento, e o modelo em si. Também definimos o espaço de busca de hiperparâmetros para o `RandomizedSearchCV`.

```
cv = StratifiedKFold(n_splits=2, shuffle=True, random_state=17)
searches = {}

# -- KNN
knn_pipe = ImbPipeline(steps=[("prep", preprocessor), ("smote", SMOTE(
param_knn = {"model__n_neighbors": range(3, 31), "model__weights": ["u
searches["KNN"] = RandomizedSearchCV(knn_pipe, param_knn, n_iter=20, c

# -- Decision Tree
dt_pipe = ImbPipeline(steps=[("prep", preprocessor), ("smote", SMOTE(r
param_dt = {"model__max_depth": range(1, 31), "model__min_samples_spli
searches["Decision Tree"] = RandomizedSearchCV(dt_pipe, param_dt, n_it

# -- Random Forest
rf_pipe = ImbPipeline(steps=[("prep", preprocessor), ("smote", SMOTE(r
param_rf = {"model__n_estimators": range(500, 2001, 250), "model__max_
searches["Random Forest"] = RandomizedSearchCV(rf_pipe, param_rf, n_it

# -- LightGBM
lgbm_pipe = ImbPipeline(steps=[("prep", preprocessor), ("smote", SMOTE
param_lgbm = {"model__n_estimators": range(500, 2001, 250), "model__ma
searches["LightGBM"] = RandomizedSearchCV(lgbm_pipe, param_lgbm, n_ite

# -- XGBoost
xgb_pipe = ImbPipeline(steps=[("prep", preprocessor), ("smote", SMOTE(
```

```
param_xgb = {"model__n_estimators": range(500, 2001, 250), "model__max_
searches["XGBoost"] = RandomizedSearchCV(xgb_pipe, param_xgb, n_iter=2
```

8. Treinamento e Avaliação em Validação Cruzada

Iteramos sobre cada modelo, treinamos com o `RandomizedSearchCV`, e avaliamos a performance do melhor estimador na validação cruzada. Os resultados são salvos em um arquivo CSV.

```
import json

cv_results = {}
tuning_times = {}
best_params = {}

for name, search in searches.items():
    print(f"=== Treinando {name} ===")
    start_time = time.time()
    search.fit(X_train, y_train)
    end_time = time.time()
    duration = round(end_time - start_time, 2)
    tuning_times[name] = duration
    best_params[name] = search.best_params_

    cv_scores = evaluate_cv(search.best_estimator_, X_train, y_train,
    cv_results[name] = cv_scores

    print("\nMelhores parâmetros encontrados:")
    print(search.best_params_)
    print("\nMétricas médias na CV:")
    print(cv_scores.mean().round(3))
    print(f"--> Tempo de tuning: {duration}s\n")

# Salva métricas da CV
cv_summary = pd.concat(cv_results, names=["Model"]).groupby(level=0).a
cv_summary.to_csv("metrics_cv.csv")
print("Métricas da Validação Cruzada salvas em metrics_cv.csv")
print(cv_summary)

# Salva os melhores hiperparâmetros em um arquivo JSON
with open('best_hyperparameters.json', 'w') as f:
    json.dump(best_params, f, indent=4)
print("\nMelhores hiperparâmetros salvos em 'best_hyperparameters.json
```

```
=== Treinando KNN ===

Melhores parâmetros encontrados:
{'model__weights': 'distance', 'model__p': 2, 'model__n_neighbors': 8}

Métricas médias na CV:
accuracy     0.663
precision    0.882
recall       0.598
f1           0.713
roc_auc      0.753
```

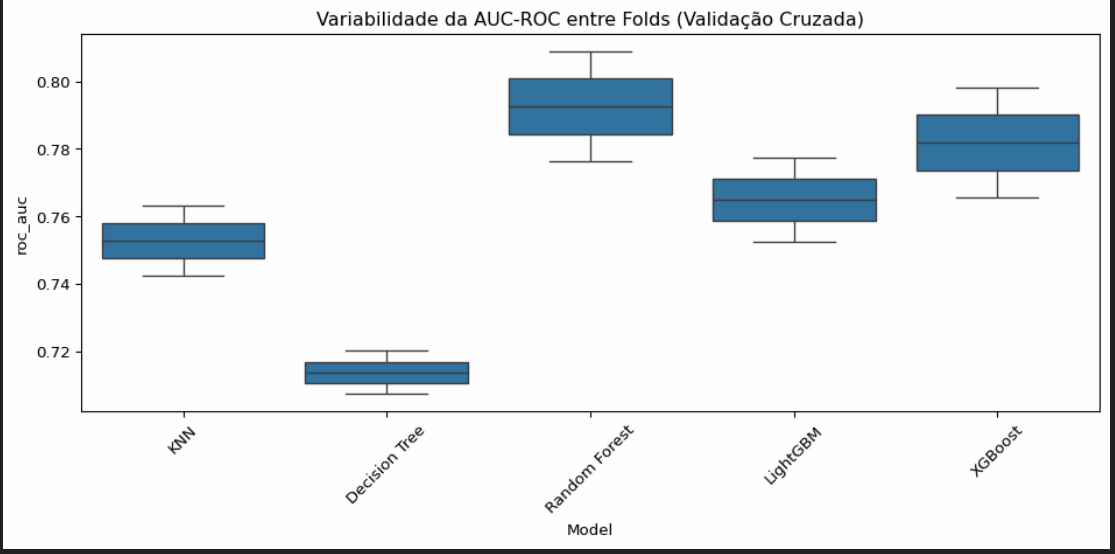
```
ks           0.433
dtype: float64
--> Tempo de tuning: 1.53s

=== Treinando Decision Tree ===

Melhores parâmetros encontrados:
{'model__min_samples_split': 12, 'model__max_depth': 22,
'model__ccp_alpha': np.float64(0.016842105263157894)}

Métricas médias na CV:
accuracy     0.706
precision    0.801
recall       0.779
f1           0.786
roc_auc      0.714
ks           0.360
dtype: float64
--> Tempo de tuning: 0.16s

=== Treinando Random Forest ===

Melhores parâmetros encontrados:
{'model__n_estimators': 500, 'model__min_samples_split': 12,
'model__max_features': 'log2', 'model__max_depth': 10}

Métricas médias na CV:
accuracy     0.769
precision    0.805
recall       0.884
f1           0.842
roc_auc      0.793
ks           0.443
dtype: float64
--> Tempo de tuning: 3.65s

=== Treinando LightGBM ===

Melhores parâmetros encontrados:
{'model__subsample': np.float64(0.868421052631579),
'model__n_estimators': 1500, 'model__max_depth': 14,
'model__learning_rate': np.float64(0.01), 'model__colsample_bytree':
np.float64(0.5)}

Métricas médias na CV:
accuracy     0.750
precision    0.795
recall       0.868
f1           0.829
roc_auc      0.765
ks           0.408
dtype: float64
--> Tempo de tuning: 111.21s

=== Treinando XGBoost ===

Melhores parâmetros encontrados:
{'model__subsample': np.float64(0.868421052631579),
```

```
'model__n_estimators': 1500, 'model__min_child_weight': 3,
'model__max_depth': 4, 'model__learning_rate': np.float64(0.01),
'model__colsample_bytree': np.float64(0.5)}

Métricas médias na CV:
accuracy     0.758
precision    0.804
recall       0.864
f1           0.833
roc_auc      0.782
ks           0.427
dtype: float64
--> Tempo de tuning: 3.19s

Métricas da Validação Cruzada salvas em metrics_cv.csv
              accuracy        precision        recall            f1
\
                  mean    std      mean    std   mean    std   mean
std
Model
Decision Tree    0.706  0.019     0.801  0.043  0.779  0.106  0.786
0.034
KNN              0.663  0.004     0.882  0.019  0.598  0.023  0.713
0.010
LightGBM         0.750  0.014     0.795  0.018  0.868  0.061  0.829
0.018
Random Forest    0.769  0.023     0.805  0.003  0.884  0.048  0.842
0.020
XGBoost          0.758  0.025     0.804  0.005  0.864  0.056  0.833
0.023

              roc_auc            ks
                 mean    std   mean    std
Model
Decision Tree   0.714  0.009  0.360  0.015
KNN             0.753  0.015  0.433  0.024
LightGBM        0.765  0.018  0.408  0.029
Random Forest   0.793  0.023  0.443  0.031
XGBoost         0.782  0.023  0.427  0.034

Melhores hiperparâmetros salvos em 'best_hyperparameters.json'
```

9. Visualização da Performance (CV)

Criamos um boxplot para comparar a distribuição da métrica AUC-ROC entre os folds da validação cruzada para cada modelo.

```
auc_long = pd.concat(cv_results, names=["Model"]).reset_index().loc[:,
plt.figure(figsize=(10, 5))
sns.boxplot(data=auc_long, x="Model", y="roc_auc")
plt.title("Variabilidade da AUC-ROC entre Folds (Validação Cruzada)")
plt.xticks(rotation=45)
plt.tight_layout()
plt.savefig("auc_boxplot.png", dpi=300)
plt.show()
```



10. Avaliação Final no Conjunto de Teste

Agora, avaliamos os melhores estimadores de cada modelo no conjunto de teste para obter uma estimativa imparcial de sua performance. Geramos curvas ROC e Precision-Recall e salvamos as métricas em `metrics_test.csv`.

```
best_models = {name: s.best_estimator_ for name, s in searches.items()}
test_rows = []
fig_roc, ax_roc = plt.subplots(figsize=(10, 8))
fig_pr, ax_pr = plt.subplots(figsize=(10, 8))

for name, mdl in best_models.items():
    model = deepcopy(mdl)
    model.fit(X_train, y_train)

    start_pred_time = time.time()
    y_pred = model.predict(X_test)
    y_prob = model.predict_proba(X_test)[:, 1]
    end_pred_time = time.time()

    test_rows.append(
        {
            "Model": name,
            "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),
            "roc_auc": roc_auc_score(y_test, y_prob),
            "ks": ks_2samp(y_prob[y_test == 1], y_prob[y_test == 0])[0
            "tuning_time_s": tuning_times[name],
            "predict_time_s": round(end_pred_time - start_pred_time, 4
        }
    )

    RocCurveDisplay.from_predictions(y_test, y_prob, name=name, ax=ax_
    PrecisionRecallDisplay.from_predictions(y_test, y_prob, name=name,

# Salva gráficos
ax_roc.set_title("ROC Curves – Test Set")
ax_roc.legend()
```

```
fig_roc.savefig("roc_curves.png", dpi=300)

ax_pr.set_title("Precision-Recall Curves — Test Set")
ax_pr.legend()
fig_pr.savefig("pr_curves.png", dpi=300)

plt.close('all')

# Salva métricas de teste
test_metrics = pd.DataFrame(test_rows).round(3)
test_metrics.to_csv("metrics_test.csv", index=False)
print("\nMétricas do Conjunto de Teste:")
print(test_metrics)
```

```
Métricas do Conjunto de Teste:
           Model  accuracy  precision  recall     f1  roc_auc     ks
\
0            KNN     0.620      0.814   0.593  0.686    0.698  0.324
1  Decision Tree     0.700      0.833   0.714  0.769    0.721  0.381
2  Random Forest     0.765      0.789   0.907  0.844    0.811  0.538
3       LightGBM     0.750      0.778   0.900  0.834    0.803  0.519
4        XGBoost     0.740      0.778   0.879  0.826    0.798  0.505

   tuning_time_s  predict_time_s
0           1.53           0.016
1           0.16           0.012
2           3.65           0.039
3         111.21           0.020
4           3.19           0.018
```

11. Seleção e Treinamento do Modelo Final

Finalmente, selecionamos o melhor modelo com base na maior AUC-ROC no conjunto de teste. Em seguida, retreinamos este modelo usando todos os dados disponíveis (treino + teste) e o salvamos em um arquivo `.pkl` para uso futuro.

```
best_name = test_metrics.loc[test_metrics["roc_auc"].idxmax(), "Model"
print(f"\n>>> Melhor modelo selecionado: {best_name}")

final_model = best_models[best_name]

# Treina o modelo final com todos os dados
final_model.fit(pd.concat([X_train, X_test]), pd.concat([y_train, y_te

# Salva o pipeline do modelo final
joblib.dump(final_model, "final_classification_model.pkl")
print(f"\nModelo final ('{best_name}') salvo em 'final_classification_
```

```
>>> Melhor modelo selecionado: Random Forest

Modelo final ('Random Forest') salvo em
'final_classification_model.pkl'
```