## data\_processing.py

import os

import numpy as np

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import MinMaxScaler

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from joblib import dump

# Ścieżki folderów

# Tworzenie ścieżek dla folderów przechowujących dane, modele oraz wyniki.

data\_dir = "./data"

outputs\_dir = "./outputs"

models\_dir = "./models"

# Upewnienie się, że foldery istnieją, w przeciwnym razie zostaną utworzone.

os.makedirs(data\_dir, exist\_ok=True)

os.makedirs(outputs\_dir, exist\_ok=True)

os.makedirs(models\_dir, exist\_ok=True)

# Ścieżki do plików z danymi wejściowymi i wyjściowymi.

train\_data\_path = os.path.join(data\_dir, "train.npy")

test\_data\_path = os.path.join(data\_dir, "test.npy")

train\_labels\_path = os.path.join(data\_dir, "train\_labels.npy")

test\_labels\_path = os.path.join(data\_dir, "test\_labels.npy")

# Wczytaj dane

# Wczytanie danych z pliku CSV zawierającego informacje o nieruchomościach.

data = pd.read\_csv("housing\_data.csv", encoding="latin1")

# Wybierz cechy i cel

# Definiowanie listy cech wejściowych (features) oraz zmiennej docelowej (target).

features = ['latitude', 'longitude', 'floor', 'rooms', 'sq', 'year']

target = 'price'

# Walidacja

# Sprawdzenie czy dane wejściowe nie zawierają brakujących lub błędnych wartości.

assert data[features].isnull().sum().sum() == 0, "Cechy mają brakujące wartości!"

assert data[target].isnull().sum() == 0, "Cel ma brakujące wartości!"

assert (data[features] >= 0).all().all(), "Niektóre cechy mają ujemne wartości!"

# Usunięcie wierszy z brakującymi wartościami (jeśli istnieją).

data = data.dropna()

# Wykres rozkładu celu przed usunięciem wartości odstających

# Wizualizacja rozkładu zmiennej docelowej przed usunięciem wartości odstających.

plt.figure(figsize=(10, 6))

sns.histplot(data[target], bins=50, kde=True, color='skyblue')

plt.title("Rozkład celu (price) przed usunięciem wartości odstających")

plt.xlabel("Cena")

plt.ylabel("Ilość")

plt.savefig(os.path.join(outputs\_dir, "price\_distribution\_before\_outliers.png"))

plt.show()

# Usuń wartości odstające # Usunięcie wierszy, w których wartość zmiennej docelowej przekracza 99. percentyl.

data = data[data[target] < data[target].quantile(0.99)]

# Wykres rozkładu celu po usunięciu wartości odstających # Wizualizacja rozkładu zmiennej docelowej po usunięciu wartości odstających.

plt.figure(figsize=(10, 6))

sns.histplot(data[target], bins=50, kde=True, color='orange')

plt.title("Rozkład celu (price) po usunięciu wartosci odstajacych")

plt.xlabel("Cena")

plt.ylabel("Ilość")

plt.savefig(os.path.join(outputs\_dir, "price\_distribution\_after\_outliers.png"))

plt.show()

# Cechy i cel # Przygotowanie macierzy cech (X) oraz wektora zmiennej docelowej (y).

X = data[features].values

y = data[target].values

# Skalowanie

# Normalizacja danych wejściowych za pomocą MinMaxScaler.

scaler\_X = MinMaxScaler()

X\_scaled = scaler\_X.fit\_transform(X)

scaler\_y = MinMaxScaler()

y\_scaled = scaler\_y.fit\_transform(y.reshape(-1, 1)).flatten()

# Zapisz skalery

# Zapisanie obiektów skalujących w celu późniejszego wykorzystania.

dump(scaler\_X, os.path.join(models\_dir, "scaler\_X.joblib"))

dump(scaler\_y, os.path.join(models\_dir, "scaler\_y.joblib"))

# Podziel dane

# Podział danych na zbiory treningowy i testowy w stosunku 80:20.

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y\_scaled, test\_size=0.2, random\_state=42)

# Zapisz dane

# Zapisanie podzielonych danych do plików w formacie NumPy.

np.save(train\_data\_path, X\_train)

np.save(test\_data\_path, X\_test)

np.save(train\_labels\_path, y\_train)

np.save(test\_labels\_path, y\_test)

# Zapisz statystyki opisowe

# Obliczenie i zapisanie statystyk opisowych dla danych.

descriptive\_stats = data[features + [target]].describe()

descriptive\_stats.to\_csv(os.path.join(data\_dir, "descriptive\_stats.csv"))

# Heatmap korelacji

# Wizualizacja macierzy korelacji między cechami a zmienną docelową.

plt.figure(figsize=(10, 8))

correlation\_matrix = data[features + [target]].corr()

sns.heatmap(correlation\_matrix, annot=True, cmap='coolwarm', fmt=".2f")

plt.title("Mapa korelacji")

plt.savefig(os.path.join(outputs\_dir, "correlation\_heatmap.png"))

plt.show()

# Augmentacja danych

# Dodanie szumu do cech w celu zwiększenia różnorodności danych treningowych.

augmented\_data = pd.DataFrame(X\_train, columns=features)

augmented\_data['latitude'] += np.random.normal(0, 0.01, size=augmented\_data.shape[0])

augmented\_data['longitude'] += np.random.normal(0, 0.01, size=augmented\_data.shape[0])

augmented\_data['floor'] = augmented\_data['floor'] \* np.random.uniform(0.9, 1.1, size=augmented\_data.shape[0])

# Połączenie oryginalnych i zaugmentowanych danych.

X\_train\_augmented = np.vstack((X\_train, augmented\_data.values))

y\_train\_augmented = np.hstack((y\_train, y\_train))

# Zapisz dane po augmentacji

# Zapisanie zaugmentowanych danych do plików w formacie NumPy.

np.save(os.path.join(data\_dir, "train\_augmented.npy"), X\_train\_augmented)

np.save(os.path.join(data\_dir, "train\_labels\_augmented.npy"), y\_train\_augmented)

# Zapisz losowe próbki

# Wybór losowych próbek z danych treningowych i zapisanie ich do pliku CSV.

random\_samples = X\_train[np.random.choice(X\_train.shape[0], 3, replace=False)]

random\_samples\_df = pd.DataFrame(random\_samples, columns=features)

random\_samples\_df['price'] = y\_train[np.random.choice(y\_train.shape[0], 3, replace=False)]

random\_samples\_df.to\_csv(os.path.join(data\_dir, "random\_samples.csv"), index=False)

# Histogramy dla cech

# Generowanie histogramów dla każdej cechy w celu analizy ich rozkładów.

for feature in features:

plt.figure(figsize=(8, 6))

sampled\_data = data[feature].sample(n=1000, random\_state=42) if feature in ['sq', 'year'] else data[feature]

sns.histplot(sampled\_data, kde=True, color='blue')

plt.title(f'Histogram i wykres gęstości dla {feature}')

plt.savefig(os.path.join(outputs\_dir, f"{feature}\_histogram.png"))

plt.show()

# Raport podsumowujący

# Tworzenie raportu tekstowego podsumowującego eksplorację i przetwarzanie danych.

report\_path = os.path.join(data\_dir, "data\_report.txt")

with open(report\_path, "w") as report:

report.write("Podsumowanie danych:\n")

report.write(str(descriptive\_stats))

report.write("\n\nMapa korelacji zapisana jako 'correlation\_heatmap.png'.\n")

report.write("Dane po augmentacji zapisane jako 'train\_augmented.npy' i 'train\_labels\_augmented.npy'.\n")

report.write("Losowe próbki zapisane w 'random\_samples.csv'.")

## 

## build\_model.py

import os

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Dropout, BatchNormalization, Conv1D, Flatten, MaxPooling1D

from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

# Ścieżki do folderów

models\_dir = "./models"

logs\_dir = "./logs"

visualizations\_dir = "./outputs"

# Upewnienie się, że foldery istnieją, w przeciwnym razie zostaną utworzone.

os.makedirs(models\_dir, exist\_ok=True)

os.makedirs(logs\_dir, exist\_ok=True)

os.makedirs(visualizations\_dir, exist\_ok=True)

# Ścieżki do plików modelu

dense\_architecture\_path = os.path.join(models\_dir, "dense\_model\_architecture.json")

dense\_model\_path = os.path.join(models\_dir, "dense\_model\_complete.keras")

conv1d\_architecture\_path = os.path.join(models\_dir, "conv1d\_model\_architecture.json")

conv1d\_model\_path = os.path.join(models\_dir, "conv1d\_model\_complete.keras")

# Funkcja do budowy modeli regresji

def build\_model(architecture="dense"):

if architecture == "dense":

model = Sequential([

Dense(256, activation='relu', input\_dim=6),

BatchNormalization(),

Dropout(0.3),

Dense(128, activation='relu'),

BatchNormalization(),

Dropout(0.3),

Dense(64, activation='relu'),

BatchNormalization(),

Dropout(0.2),

Dense(1, activation='linear') # Dla wyników regresji

])

elif architecture == "conv1d":

model = Sequential([

Conv1D(filters=32, kernel\_size=3, activation='relu', input\_shape=(6, 1)),

MaxPooling1D(pool\_size=2),

Dropout(0.2),

Flatten(),

Dense(128, activation='relu'),

BatchNormalization(),

Dropout(0.3),

Dense(64, activation='relu'),

BatchNormalization(),

Dropout(0.3),

Dense(1, activation='linear') # Dla wyników regresji

])

else:

raise ValueError("Nieobsługiwany typ architektury")

model.compile(

optimizer=tf.keras.optimizers.Adam(learning\_rate=0.001),

loss='mean\_squared\_error', # Alternatywy: 'mean\_absolute\_error', 'huber'

metrics=['mean\_absolute\_error', 'mean\_squared\_error']

)

return model

# Inicjalizacja modelu Dense

model\_dense = build\_model(architecture="dense")

# Zapisz architektury Dense do JSON

with open(dense\_architecture\_path, "w") as file:

file.write(model\_dense.to\_json())

# Zapisz nieprzeszkolony model Dense

model\_dense.save(dense\_model\_path)

# Wyświetl podsumowanie modelu Dense

print("Architektura Dense:")

model\_dense.summary()

# Inicjalizacja modelu Conv1D

model\_conv1d = build\_model(architecture="conv1d")

# Zapisz architekturę Conv1D do JSON

with open(conv1d\_architecture\_path, "w") as file:

file.write(model\_conv1d.to\_json())

# Zapisz nieprzeszkolony model Conv1D

model\_conv1d.save(conv1d\_model\_path)

# Wyświetl podsumowanie modelu Conv1D

print("\nArchitektura Conv1D:")

model\_conv1d.summary()

## training.py

import os

import numpy as np

import pandas as pd

from tensorflow.keras.models import load\_model

from tensorflow.keras.optimizers import Adam, SGD, RMSprop

from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

from tensorflow.keras.utils import plot\_model

import itertools

import matplotlib.pyplot as plt

# Ścieżki

model\_dir = "./models"

data\_dir = "./data"

output\_dir = "./outputs"

os.makedirs(output\_dir, exist\_ok=True) # Upewnij się, że katalog wyjściowy istnieje

# Ścieżki do wyników

tuning\_results\_path = os.path.join(output\_dir, "hyperparameter\_tuning\_results.csv")

tuning\_plot\_path = os.path.join(output\_dir, "hyperparameter\_tuning\_plot.png")

epoch\_analysis\_path = os.path.join(output\_dir, "epoch\_analysis.csv")

epoch\_plot\_path = os.path.join(output\_dir, "epoch\_analysis\_plot.png")

training\_history\_path = os.path.join(output\_dir, "training\_history.csv")

learning\_curve\_path = os.path.join(output\_dir, "learning\_curve.png")

model\_architecture\_path = os.path.join(output\_dir, "model\_architecture.png")

# Wczytaj dane treningowe

X\_train = np.load(os.path.join(data\_dir, "train.npy"))

y\_train = np.load(os.path.join(data\_dir, "train\_labels.npy"))

# Wczytaj model

model = load\_model(os.path.join(model\_dir, "dense\_model\_complete.keras"))

# Generuj i zapisz wizualizacje architektury modelu

plot\_model(model, to\_file=model\_architecture\_path, show\_shapes=True)

print(f"Wizualizacja architektury modelu zapisana do {model\_architecture\_path}.")

# Callbacks do lepszego treningu

callbacks = [

EarlyStopping(monitor='val\_loss', patience=10, restore\_best\_weights=True),

ReduceLROnPlateau(monitor='val\_loss', factor=0.5, patience=5, verbose=1)

]

# Strojenie hiperparametrów

batch\_sizes = [16, 32, 64] # Lista różnych rozmiarów batcha do przetestowania

learning\_rates = [0.001, 0.0005, 0.0001] # Lista różnych współczynników uczenia

optimizers = { # Słownik z różnymi optymalizatorami

'adam': Adam,

'sgd': SGD,

'rmsprop': RMSprop

}

results = [] # Lista do przechowywania wyników strojenia hiperparametrów

# Iteracja przez wszystkie kombinacje parametrów: batch\_size, learning\_rate i optymalizatora

for batch\_size, lr, opt\_name in itertools.product(batch\_sizes, learning\_rates, optimizers.keys()):

print(f"Trening z batch\_size={batch\_size}, learning\_rate={lr}, optimizer={opt\_name}")

optimizer = optimizers[opt\_name](learning\_rate=lr)

model.compile(optimizer=optimizer, loss='mse', metrics=['mae'])

# Trenowanie modelu z bieżącą konfiguracją hiperparametrów

history = model.fit(

X\_train,

y\_train,

validation\_split=0.2,

epochs=50,

batch\_size=batch\_size,

callbacks=callbacks,

verbose=0

)

final\_val\_loss = history.history['val\_loss'][-1]

# Zapisanie wyników dla aktualnej konfiguracji hiperparametrów

results.append({

'batch\_size': batch\_size,

'learning\_rate': lr,

'optimizer': opt\_name,

'val\_loss': final\_val\_loss

})

# Zapisanie wyników strojenia do pliku CSV

results\_df = pd.DataFrame(results)

results\_df.to\_csv(tuning\_results\_path, index=False)

print(f"Wyniki strojenia hiperparametrow zapisane do {tuning\_results\_path}.")

# Wykres wyników strojenia hiperparametrów

plt.figure(figsize=(10, 5))

for opt\_name in results\_df['optimizer'].unique():

subset = results\_df[results\_df['optimizer'] == opt\_name]

plt.plot(subset['learning\_rate'], subset['val\_loss'], marker='o', label=f"Optimizer: {opt\_name}")

plt.xlabel('Learning Rate')

plt.ylabel('Validation Loss')

plt.title('Strojenie Hiperparametrow: Validation Loss vs Learning Rate')

plt.legend()

plt.savefig(tuning\_plot\_path)

plt.show()

# Analiza epok

epochs\_to\_test = [50, 100, 200]

epoch\_results = []

# Iteracja przez różne liczby epok

for num\_epochs in epochs\_to\_test:

print(f"Trening z {num\_epochs} epokami.")

history = model.fit(

X\_train,

y\_train,

validation\_split=0.2,

epochs=num\_epochs,

batch\_size=32,

callbacks=callbacks,

verbose=0

)

best\_val\_loss = min(history.history['val\_loss'])

epoch\_results.append({'epochs': num\_epochs, 'best\_val\_loss': best\_val\_loss})

# Zapisanie wyników analizy epok do pliku CSV

epoch\_results\_df = pd.DataFrame(epoch\_results)

epoch\_results\_df.to\_csv(epoch\_analysis\_path, index=False)

print(f"Wyniki analizy epok zapisane do {epoch\_analysis\_path}.")

# Wykres analizy epok

plt.figure(figsize=(10, 5))

plt.plot(epoch\_results\_df['epochs'], epoch\_results\_df['best\_val\_loss'], marker='o')

plt.xlabel('Liczba Epok')

plt.ylabel('Najlepszy Validation Loss')

plt.title('Analiza Epok: Validation Loss vs Liczba Epok')

plt.savefig(epoch\_plot\_path)

plt.show()

# Finalny trening i zapis modelu

model.compile(optimizer=Adam(learning\_rate=0.0005), loss='mse', metrics=['mae'])

history = model.fit(

X\_train,

y\_train,

validation\_split=0.2,

epochs=100,

batch\_size=32,

callbacks=callbacks

)

model.save(os.path.join(model\_dir, "dense\_model\_complete.keras"))

print(f"Przeszkolony model zapisany do {os.path.join(model\_dir, 'dense\_model\_complete.keras')}.")

# Eksport finalnej historii treningu

history\_df = pd.DataFrame(history.history)

history\_df.to\_csv(training\_history\_path, index=False)

print(f"Historia treningu zapisana do {training\_history\_path}.")

# Wykres finalnej krzywej uczenia

plt.figure(figsize=(10, 5))

plt.plot(history.history['loss'], label='Training Loss')

plt.plot(history.history['val\_loss'], label='Validation Loss')

plt.xlabel('Epoki')

plt.ylabel('Loss')

plt.title('Krzywa Uczenia')

plt.legend()

plt.savefig(learning\_curve\_path)

plt.show()

print(f"Krzywa uczenia zapisana do {learning\_curve\_path}.")

## 

## evaluate\_model.py

import os

import numpy as np

import pandas as pd

from tensorflow.keras.models import load\_model

from sklearn.metrics import mean\_squared\_error, mean\_absolute\_error, r2\_score

import matplotlib.pyplot as plt

import seaborn as sns

import shap

from sklearn.inspection import permutation\_importance

# Klasa wrappera dla kompatybilności ze scikit-learn

# Definiowanie klasy 'KerasRegressorWrapper', która owija model Keras, aby był kompatybilny ze scikit-learn

class KerasRegressorWrapper:

def \_\_init\_\_(self, model):

self.model = model

# Funkcja 'fit', która nic nie robi (nie jest wymagana w tym wraperze)

def fit(self, X, y):

pass

# Funkcja 'score', która oblicza wynik dla modelu na podstawie średniego błędu kwadratowego (MSE)

def score(self, X, y):

predictions = self.model.predict(X).flatten()

return -mean\_squared\_error(y, predictions)

# Konfiguracja ścieżek

data\_dir = "./data"

outputs\_dir = "./outputs"

models\_dir = "./models"

os.makedirs(outputs\_dir, exist\_ok=True)

# Funkcja do zapisywania plików tekstowych

def save\_text(filepath, content):

with open(filepath, "w", encoding="utf-8") as file:

file.write(content)

# Funkcja do tworzenia i zapisywania wykresów

def save\_plot(fig, filepath):

fig.savefig(filepath)

plt.close(fig)

# Wczytaj dane i model

def load\_data\_and\_model():

model\_path = os.path.join(models\_dir, "dense\_model\_complete.keras")

X\_test = np.load(os.path.join(data\_dir, "test.npy"))

y\_test = np.load(os.path.join(data\_dir, "test\_labels.npy"))

model = load\_model(model\_path)

return model, X\_test, y\_test

# Ewaluacja modelu

def evaluate\_model(model, X\_test, y\_test):

y\_pred = model.predict(X\_test).flatten()

mse = mean\_squared\_error(y\_test, y\_pred)

mae = mean\_absolute\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

return y\_pred, mse, mae, r2

# Generowanie raportu z ewaluacji

def generate\_evaluation\_report(mse, mae, r2):

content = f"Mean Squared Error (MSE): {mse:.4f}\n"

content += f"Mean Absolute Error (MAE): {mae:.4f}\n"

content += f"R^2 Score: {r2:.4f}\n"

save\_text(os.path.join(outputs\_dir, "evaluation\_summary.txt"), content)

# Zapisz przewidywania

def save\_predictions(y\_test, y\_pred):

predictions\_df = pd.DataFrame({"True Price": y\_test, "Predicted Price": y\_pred})

predictions\_df.to\_csv(os.path.join(outputs\_dir, "predictions.csv"), index=False)

# Wykres prawdziwe vs przewidywane wartości

def plot\_true\_vs\_predicted(y\_test, y\_pred):

fig, ax = plt.subplots(figsize=(10, 6))

ax.scatter(y\_test, y\_pred, alpha=0.6, label="Predykcja")

ax.plot([y\_test.min(), y\_test.max()], [y\_test.min(), y\_test.max()], 'r--', label="Idealne Dopasowanie")

ax.set\_xlabel('Prawdziwe Ceny')

ax.set\_ylabel('Przewidywane Ceny')

ax.set\_title('Prawdziwe vs Przewidywane Ceny')

ax.legend()

plt.show()

save\_plot(fig, os.path.join(outputs\_dir, "true\_vs\_predicted.png"))

# Wykres rozkładu błędów

def plot\_error\_distribution(errors):

fig, ax = plt.subplots(figsize=(10, 5))

sns.histplot(errors, kde=True, color="blue", ax=ax)

ax.set\_title("Rozkład Błędów")

ax.set\_xlabel("Błąd")

ax.set\_ylabel("Ilość")

plt.show()

save\_plot(fig, os.path.join(outputs\_dir, "error\_distribution.png"))

# Zapisz statystyki błędów

def save\_error\_statistics(errors):

mean\_error = np.mean(errors)

std\_error = np.std(errors)

content = f"Średni Błąd: {mean\_error:.4f}\nOdchylenie Standardowe Błędu: {std\_error:.4f}\n"

save\_text(os.path.join(outputs\_dir, "error\_statistics.txt"), content)

# Zapisz największe błędy

def save\_largest\_errors(y\_test, y\_pred, errors):

errors\_df = pd.DataFrame({"True Price": y\_test, "Predicted Price": y\_pred, "Error": np.abs(errors)})

largest\_errors = errors\_df.sort\_values(by="Error", ascending=False).head(10)

largest\_errors.to\_csv(os.path.join(outputs\_dir, "largest\_errors.csv"), index=False)

# Generowanie analizy SHAP

def generate\_shap\_analysis(model, X\_test):

explainer = shap.Explainer(model, X\_test)

shap\_values = explainer(X\_test)

shap.summary\_plot(shap\_values, X\_test, show=False)

plt.show()

save\_plot(plt.gcf(), os.path.join(outputs\_dir, "feature\_importance.png"))

# Ważność cech za pomocą permutacji

def generate\_permutation\_importance(model, X\_test, y\_test):

wrapped\_model = KerasRegressorWrapper(model)

perm\_importance = permutation\_importance(wrapped\_model, X\_test, y\_test, n\_repeats=10, random\_state=42)

feature\_importances = pd.DataFrame({

"Feature": [f"Feature {i}" for i in range(X\_test.shape[1])],

"Importance": perm\_importance.importances\_mean

}).sort\_values(by="Importance", ascending=False)

feature\_importances.to\_csv(os.path.join(outputs\_dir, "feature\_importances.csv"), index=False)

# Główna funkcja pipeline

def main():

model, X\_test, y\_test = load\_data\_and\_model()

y\_pred, mse, mae, r2 = evaluate\_model(model, X\_test, y\_test)

# Zapisz wyniki ewaluacji

generate\_evaluation\_report(mse, mae, r2)

save\_predictions(y\_test, y\_pred)

# Wykresy wyników

plot\_true\_vs\_predicted(y\_test, y\_pred)

errors = y\_test - y\_pred

plot\_error\_distribution(errors)

# Zapisz analizę błędów

save\_error\_statistics(errors)

save\_largest\_errors(y\_test, y\_pred, errors)

# Generowanie ważności cech

generate\_shap\_analysis(model, X\_test)

generate\_permutation\_importance(model, X\_test, y\_test)

# Warunek wywołania main

if \_\_name\_\_ == "\_\_main\_\_":

main()