UNIVERSIDAD DEL VALLE DE GUATEMALA

Machine Learning

Sección 10

Ejercicio 4

Juan Ángel Carrera 20593
Juan Carlos Baján 20109
José Mariano Reyes 20074
Esteban Aldana Guerra 20591
Luis Pedro Gonzalez Aldana 20008

GUATEMALA, 17 de Agosto de 2024

Ejercicio 4: Implementación del Pipeline y Contenedorización con Docker

Pipeline

El pipeline implementado en este ejercicio automatiza las etapas de preprocesamiento de datos y el entrenamiento del modelo de red neuronal. El código realiza lo siguiente:

Imputación de Valores Nulos (SimpleImputer):

Utiliza la media de las columnas numéricas para rellenar valores faltantes en los datos, asegurando que no haya datos nulos que afecten al entrenamiento del modelo.

Estandarización (StandardScaler):

Normaliza las características seleccionadas para que tengan una media de 0 y una desviación estándar de 1, asegurando que las variables contribuyan equitativamente al modelo.

Modelado (KerasRegressor):

Define un modelo de red neuronal utilizando KerasRegressor, con dos capas ocultas de 128 y 64 neuronas, respectivamente, y capas de Dropout para regularización. Se entrena el modelo durante 200 épocas, con early stopping activado si la pérdida de validación no mejora durante 10 épocas consecutivas.

Código Implementado

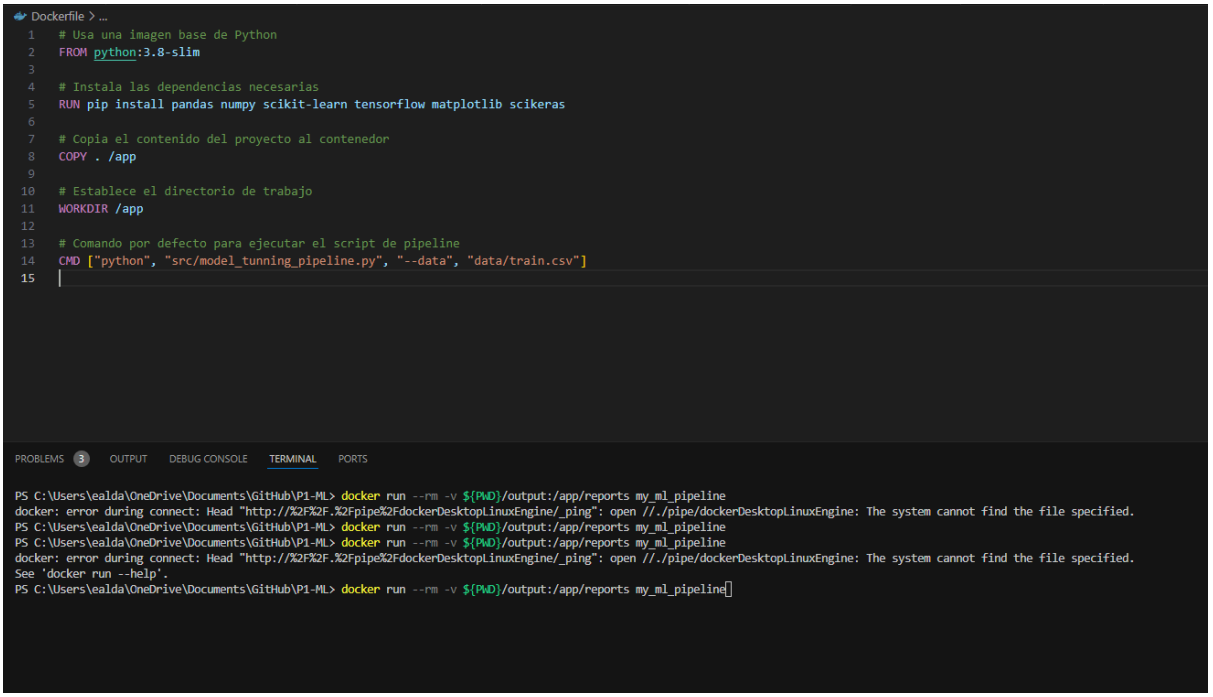
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Input
from scikeras.wrappers import KerasRegressor
from tensorflow.keras.callbacks import EarlyStopping
import matplotlib.pyplot as plt
import os

# Función para crear el modelo de Keras
```

```
def create_model():
    model = Sequential([
        Input(shape=(5,)),  # Ajusta según las características
        Dense(128, activation='relu'),
        Dropout(0.3),
        Dense(64, activation='relu'),
        Dropout(0.3),
        Dense(32, activation='relu'),
        Dense(1)
    ])
    model.compile(optimizer='adam', loss='mse')
    return model

# Crear el pipeline
pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')),  # Imputación de valores nulos
    ('scaler', StandardScaler()),  # Estandarización de características
    ('model', KerasRegressor(model=create_model, epochs=200, batch_size=32, verbose=0))
# Modelo
])

# Crear directorios para guardar modelos si no existen
os.makedirs('models', exist_ok=True)
os.makedirs('reports', exist_ok=True)

# Cargar los datos
train_data = pd.read_csv('data/train.csv')

# Selección de características y variable objetivo
features = ['OverallQual', 'GrLivArea', 'GarageCars', 'GarageArea', 'TotalBsmtSF']
target = 'SalePrice'

X = train_data[features]
y = train_data[target]

# División de datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Entrenar todo el pipeline y capturar el historial del entrenamiento
early_stopping = EarlyStopping(monitor='val_loss', patience=10,
restore_best_weights=True)
history = pipeline.fit(X_train, y_train, model__validation_data=(X_test, y_test),
model__callbacks=[early_stopping])

# Guardar el modelo entrenado con regularización
pipeline.named_steps['model'].model_.save('models/nn_model_tuned.keras')

# Predicciones y evaluación
```

```
y_pred = pipeline.predict(X_test)
rmse_tuned = np.sqrt(mean_squared_error(y_test, y_pred))
print(f'RMSE después de ajuste: {rmse_tuned}')

# Guardar los resultados en un archivo
with open('reports/tuning_report_nn.txt', 'w') as f:
    f.write(f'RMSE después de ajuste: {rmse_tuned}\n')

# Gráfica de la pérdida durante el entrenamiento
plt.figure(figsize=(10, 6))
plt.plot(history.named_steps['model'].history_['loss'], label='Pérdida de entrenamiento')
plt.plot(history.named_steps['model'].history_['val_loss'], label='Pérdida de validación')
plt.title('Pérdida durante el entrenamiento (con ajuste)')
plt.xlabel('Época')
plt.ylabel('Pérdida')
plt.legend()
plt.grid(True)
plt.savefig('reports/loss_curve_tuned.png')
plt.show()

# Gráfica de predicciones vs valores reales
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred)
plt.xlabel('Valores Reales')
plt.ylabel('Predicciones')
plt.title('Predicciones vs Valores Reales (con ajuste)')
plt.grid(True)
plt.savefig('reports/pred_vs_real_tuned.png')
plt.show()
```

Contenedorización con Docker

Para asegurar que el pipeline pueda ejecutarse de manera consistente en cualquier entorno, se creó un contenedor Docker que encapsula todas las dependencias necesarias. A continuación se presenta el Dockerfile utilizado para contenerizar el proyecto:



Este Dockerfile realiza los siguientes pasos:

1. Imagen Base: Usa la imagen python:3.8-slim como base, lo cual es ligero y adecuado para entornos de producción.
2. Instalación de Dependencias: Se instalan las bibliotecas necesarias para ejecutar el pipeline, incluyendo pandas, numpy, scikit-learn, tensorflow, matplotlib y scikeras.
3. Copia del Proyecto: Copia todos los archivos del proyecto al directorio /app dentro del contenedor.
4. Directorio de Trabajo: Establece /app como el directorio de trabajo dentro del contenedor.
5. Comando por Defecto: Define el comando que ejecuta el script principal del pipeline, permitiendo la ejecución directa cuando el contenedor se inicia.