Scoring

December 19, 2024

Importamos las librerias necesarias

[1]:
```
import joblib
import pymysql
import warnings
warnings.filterwarnings('ignore')
from sklearn.metrics import mean_squared_error, r2_score
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Cargamos el modelo entrenado en "Training.py" usando la función joblib.load() y el transformador PolynomialFeatures Usamos PolynomialFeatures para transformar un conjunto de características numéricas en un conjunto de características polinomiales.

[2]:
```
# Cargamos el modelo en el notebook
poly_reg_model = joblib.load('modelo_entrenado.pkl')
```

[3]:
```
# Cargamos el transformador PolynomialFeatures
poly = joblib.load('transformador_polynomial.pkl')
```

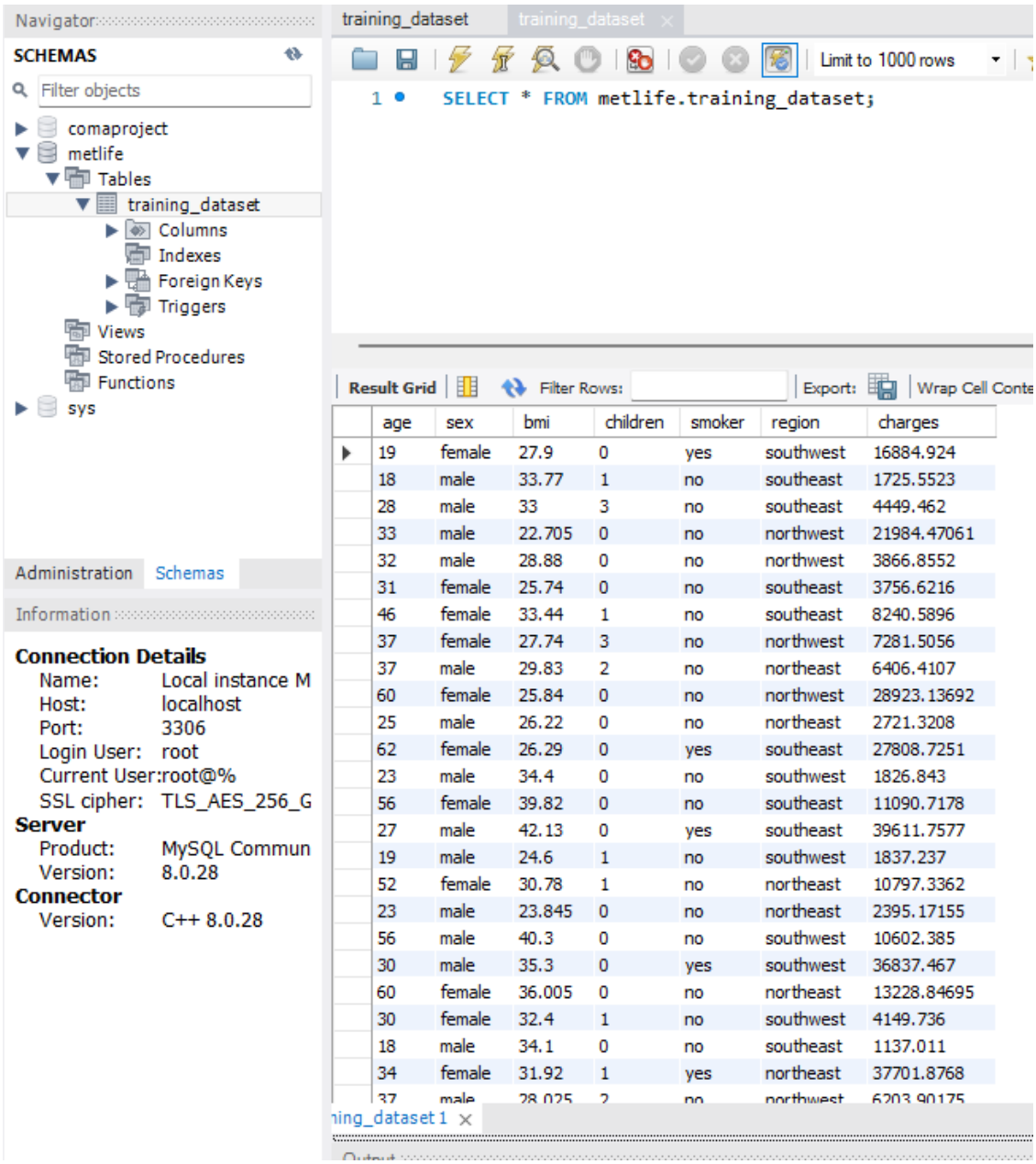
Realizamos la conexión a la base de datos de MySQL, previamente creada (usando el archivo csv compartido). Al mismo tiempo que, creamos una excepción en caso de que se llegué a generar un mensaje de error.

[4]:
```
db_name = "METLIFE"
db_host = "localhost"
db_username = "root"
db_password = "123456"

try:
    conexion = pymysql.connect(host = db_host,
                               port = int(3306),
                               user = "root",
                               password = db_password,
                               db = db_name)
except e:
    print (e)
if conexion:
```

```
    print ("Conexión exitosa")
else:
        print ("error")
```

```
Conexión exitosa
```



Una vez realizada la conexión, obtenemos todo el contenido de la tabla mediente una sentencia de SQL y almacenamos el contenido en la variable df.

[5]:
```
df = pd.read_sql_query("select * from training_dataset", conexion)
```

[6]:
```
df['log_charges'] = np.log(df['charges'] )
```

Realizamos la transformación de la variable “charges” aplicando la función de logartimo y almacenando los nuevos valores en una columna llamada “log_charges”, esto basado en la distribución de la variable y las diferencias de magnitudes con las demás variables.

[7]:
```
# Definimos el orden de las categorías para las columnas
encoding_orders = {
    'sex': ['male', 'female'],
    'smoker': ['no', 'yes'],
    'region': ['southeast','southwest','northwest','northeast']
}

# Aplicamos la codificación ordinal para cada columna
for column, order in encoding_orders.items():
    df[column] = df[column].map({cat: idx for idx, cat in enumerate(order)})
```

En el script Training.py ya identificamos las frecuencias para cada columna categorica por lo que, es facil definir los valores que puede tomar cada columna.

[8]:
```
# Seleccionamos 10 valores aleatorios de la tabla
muestra_aleatoria = df.sample(n=10, random_state=123)
```

[9]:
```
print(muestra_aleatoria)
```

```
      age  sex     bmi  children  smoker  region      charges  log_charges
650    49    1  42.680         2       0       0   9800.88820     9.190228
319    32    0  37.335         1       0       3   4667.60765     8.448402
314    27    1  31.400         0       1       1  34838.87300    10.458489
150    35    0  24.130         1       0       2   5125.21570     8.541928
336    60    0  25.740         0       0       0  12142.57860     9.404473
970    50    1  28.160         3       0       0  10702.64240     9.278246
169    27    0  18.905         3       0       3   4827.90495     8.482168
684    33    1  18.500         1       0       1   4766.02200     8.469267
1097   22    0  33.770         0       0       0   1674.63230     7.423349
512    51    0  22.420         0       0       3   9361.32680     9.144342
```

Una vez que, hemos realizado las transformaciones y codificaciones necesarias al dataframe (df) para que la data este lista para ser ingresada al modelo. Tomamos de manera aleatoria 10 registros de la tabla usando la funcion sample(), además de plantar la semilla para la reproductividad usando “random_state=123”.

[10]:
```
# Nos aseguramos de que la muestra aleatoria tiene las mismas columnas de entrada
X_unseen = muestra_aleatoria[["smoker","children","age",'region',"bmi","sex"]]
y_unseen = muestra_aleatoria["log_charges"]

# Transformamos las características con PolynomialFeatures
poly_features_unseen = poly.transform(X_unseen)
```

```
# Realizamos predicciones con el modelo entrenado
poly_y_unseen_predict = poly_reg_model.predict(poly_features_unseen)

# Calculamos el error RMSE para la muestra no vista
poly_rmse_unseen = np.sqrt(mean_squared_error(y_unseen, poly_y_unseen_predict))
print(f"poly RMSE en muestra no vista = {poly_rmse_unseen}")

# Mostramos las predicciones y los valores reales
muestra_aleatoria["predicted_log_charges"] = poly_y_unseen_predict
```

```
poly RMSE en muestra no vista = 0.15800897766227984
```

1.- Selección de las columnas de entrada y salida: X_unseen = muestra_aleatoria[[“smoker”,“children”,“age”,‘region’,“bmi”,“sex”]]
y_unseen = muestra_aleatoria[“log_charges”]
Se extraen las columnas de entrada (X_unseen) y la variable objetivo (y_unseen) de la muestra aleatoria. Esto asegura que los datos utilizados tengan las mismas características que el modelo espera.

2.- Transformación de características: poly_features_unseen = poly.transform(X_unseen)
Se aplica la misma transformación “PolynomialFeatures” que se utilizó en el conjunto de entrenamiento. “poly.transform” transforma X_unseen para que incluya las características polinómicas necesarias.

3.- Predicción con el modelo entrenado: poly_y_unseen_predict = poly_reg_model.predict(poly_features_unseen)
El modelo entrenado (poly_reg_model) realiza predicciones sobre las características polinómicas de la muestra no vista. Los resultados se almacenan en poly_y_unseen_predict.

4.- Cálculo del RMSE en la muestra no vista: poly_rmse_unseen = np.sqrt(mean_squared_error(y_unseen, poly_y_unseen_predict))
print(f“poly RMSE en muestra no vista = {poly_rmse_unseen}”)

Se calcula el RMSE (raíz del error cuadrático medio), que mide qué tan lejos están las predicciones (poly_y_unseen_predict) de los valores reales (y_unseen). El RMSE obtenido es 0.15800897766227984, lo cual sugiere que el error en la muestra no vista es bajo y el modelo generaliza bien.

5.-Agregar las predicciones a la muestra: muestra_aleatoria[“predicted_log_charges”] = poly_y_unseen_predict
Se añade una nueva columna predicted_log_charges a la muestra aleatoria, donde se almacenan las predicciones del modelo.

Conclusión:

poly RMSE en muestra no vista = 0.1580: El RMSE en la muestra no vista es bajo (0.15800897766227984), lo que indica que el modelo sigue siendo preciso al predecir datos nuevos no utilizados en el entrenamiento. Valores bajos de RMSE significan que las predicciones del modelo están muy cercanas a los valores reales.

[11]:
```
# Aplicamos la exponencial a la columna de predicciones
muestra_aleatoria["predicted_charges"] = np.
 ↪exp(muestra_aleatoria["predicted_log_charges"])

# Mostramos la tabla con los valores reales, predicciones logarítmicas y␣
 ↪predicciones exponenciales
print(muestra_aleatoria[["charges","log_charges", "predicted_log_charges",␣
 ↪"predicted_charges"]])
```

	charges	log_charges	predicted_log_charges	predicted_charges
650	9800.88820	9.190228	9.185786	9757.449725
319	4667.60765	8.448402	8.705045	6033.274826
314	34838.87300	10.458489	10.217141	27368.298933
150	5125.21570	8.541928	8.579529	5321.599099
336	12142.57860	9.404473	9.548819	14028.115540
970	10702.64240	9.278246	9.382985	11884.439320
169	4827.90495	8.482168	8.562177	5230.054575
684	4766.02200	8.469267	8.652289	5723.231883
1097	1674.63230	7.423349	7.649133	2098.825505
512	9361.32680	9.144342	9.106433	9013.089162

Aplicamos la función exponencial a las predicciones logarítmicas: muestra_aleatoria[“predicted_charges”] = np.exp(muestra_aleatoria[“predicted_log_charges”])

np.exp() aplica la función exponencial a los valores de la columna predicted_log_charges. Como el modelo hizo predicciones sobre la variable logarítmica (log_charges), se necesita aplicar la exponencial para regresar las predicciones a su escala original (charges).

Comparación entre charges y predicted_charges: Los valores de predicted_charges son generalmente cercanos a los valores reales (charges), pero hay diferencias en algunos casos.

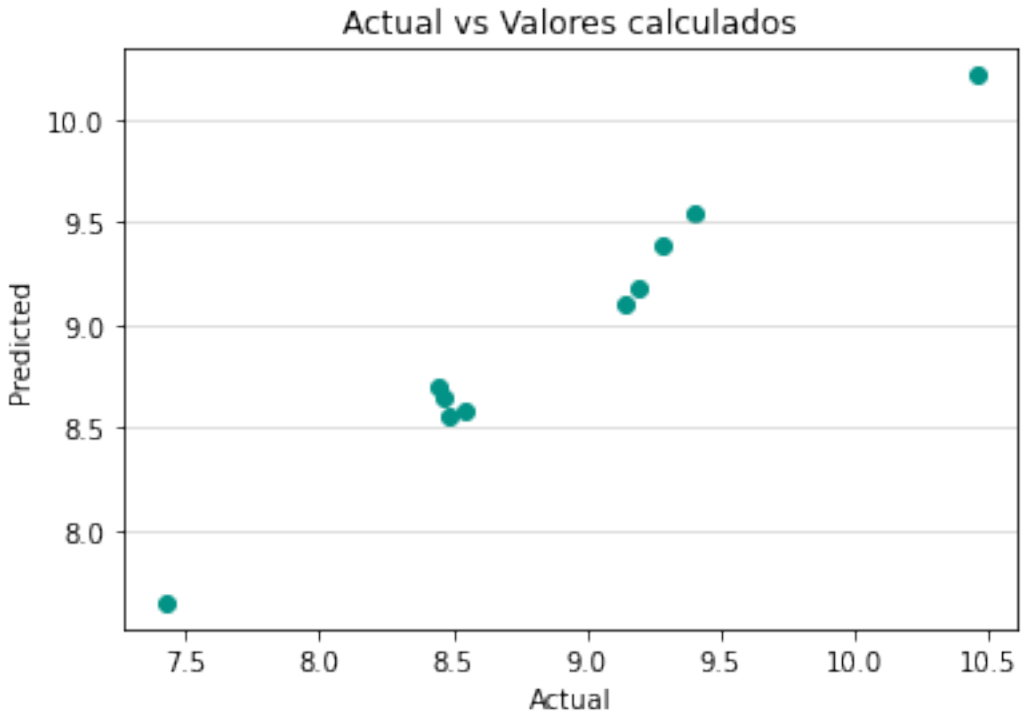
Comportamiento del modelo:

El modelo predice bien para valores pequeños y medianos, pero presenta diferencias en valores extremos.

Importancia de la transformación: La transformación logarítmica permitió capturar mejor la estructura de los datos y reducir el efecto de valores muy grandes. Al aplicar la exponencial, regresamos las predicciones a su escala original para que puedan compararse directamente con los valores reales.

[12]:
```
```

```
# Visualizamos la diferencia entre los precios actuales y los que el modelo␣
 ↪logró predecir
plt.
 ↪scatter(muestra_aleatoria["log_charges"],muestra_aleatoria["predicted_log_charges"],␣
 ↪color='#029386')
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.title('Actual vs Valores calculados')
plt.grid(axis='y', alpha=0.5)
plt.show()
```



[13]:
```
r2 =␣
 ↪r2_score(muestra_aleatoria["log_charges"],muestra_aleatoria["predicted_log_charges"])
print(f"R-squared: {r2}")
```

```
R-squared: 0.9572531420290018
```

R^2 alto (0.9572): El modelo captura muy bien la tendencia de los datos no vistos. Los resultados muestran que el modelo tiene un buen desempeño en general, pero puede subestimar valores extremos.