

Maestría en Inteligencia Analítica de Datos

Machine Learning y Procesamiento de Lenguaje Natural

Integrantes:

- Sergio Rojas
- Cristian Amaya
- Gloria Ramos
- Andrés Beltrán

Informe Final: Proyecto 1

El objetivo principal de este informe es presentar los resultados y análisis del modelo de prospección para los precios de los automóviles teniendo en cuenta distintas variables como: año, marca, modelo, entre otras. Asimismo, disponibilizar los resultados en la web a través del despliegue del modelo en la nube AWS.

Preprocesamiento de datos

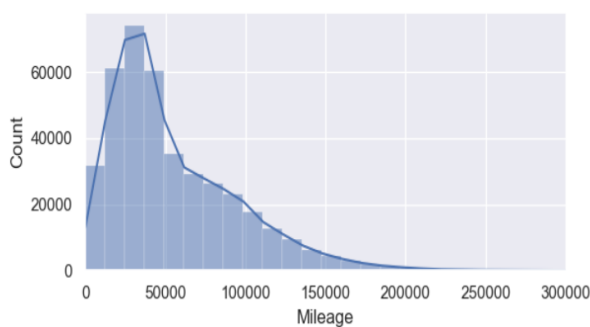
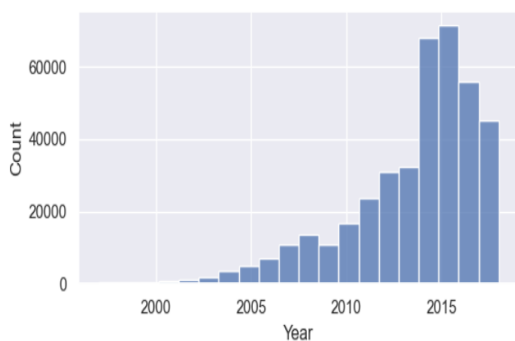
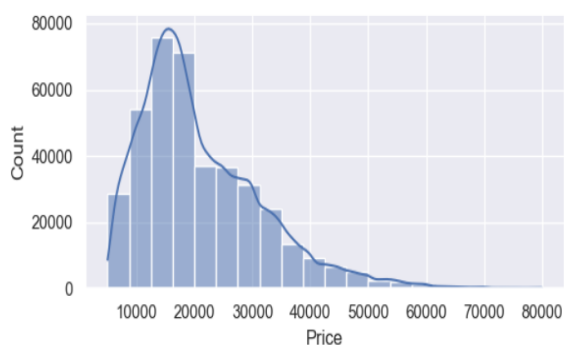
Al iniciar el preprocesamiento de los datos, se llaman las bases de datos dentro de las variables “data Training” y “data Testing” y se cargan las librerías necesarias para el ejercicio de prospección y determinar el precio de los automóviles. Al estudiar las bases de datos se encuentra que el dataframe de entrenamiento contiene tres variables numéricas (Price, Year, Mileage), de las cuales sólo Price y Mileage son continuas y tres variables categóricas (State, Make, Model), mientras que el dataframe de testeo contiene las mismas columnas sin la variable a predecir ‘Price’.

Al realizar una análisis preliminar de los datos se encuentra que, 1) Existen 400 mil registros en la base de datos de entrenamiento, lo que significa un reto en procesamiento de datos, esto implica que los modelos a usar no deberían utilizar estructuras complejas para poder procesar esta gran cantidad de datos, 2) la variable de respuesta (Price) parece tener una distribución ligeramente sesgada a la derecha, mostrando que en el mercado existe una mayor cantidad de oferta en el segmento de precios bajos. 3) Por otro lado, la variable ‘Year’ tiene una distribución sesgada a la izquierda, mientras que la variable ‘Mileage’ tiene una distribución sesgada a la derecha. Esta concentración de datos en ‘Year’ y ‘Mileage’, podría confirmar que existe una alta proporción de autos recientes que no tienen mucho kilometraje, es decir, puede que estas variables tengan una alta correlación negativa.

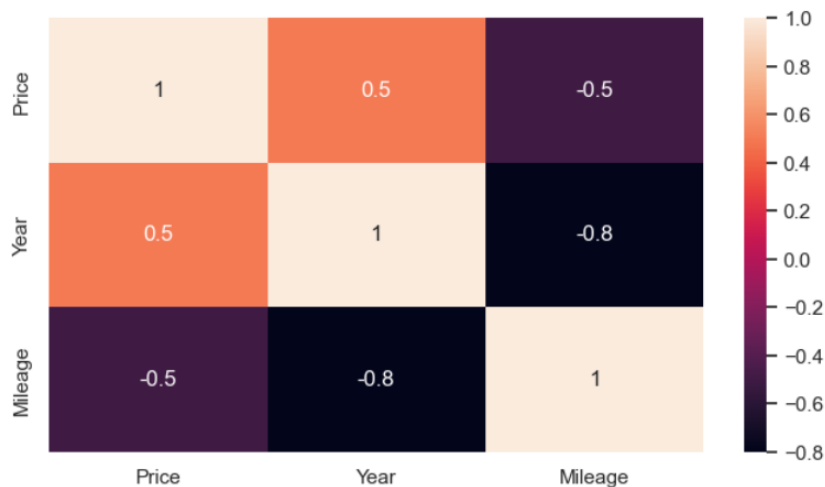
```
1 dataTraining.describe()
```

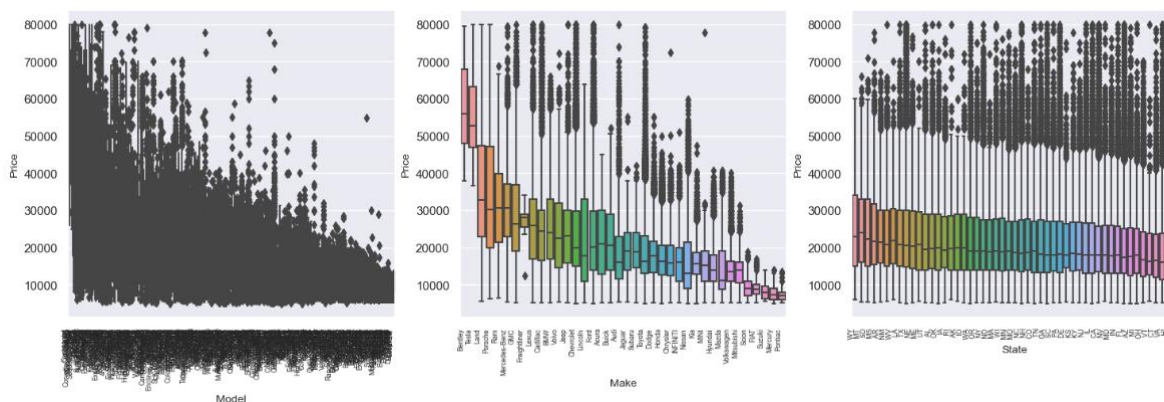
	Price	Year	Mileage
count	400000.000000	400000.000000	4.000000e+05
mean	21146.919312	2013.198125	5.507296e+04
std	10753.664940	3.292326	4.088102e+04
min	5001.000000	1997.000000	5.000000e+00
25%	13499.000000	2012.000000	2.584100e+04
50%	18450.000000	2014.000000	4.295500e+04
75%	26999.000000	2016.000000	7.743300e+04
max	79999.000000	2018.000000	2.457832e+06

```
<seaborn.axisgrid.FacetGrid at 0x15d2b2fd130>
```



Al observar la tabla de correlaciones se confirma que las variables 'Year' y 'Mileage' tienen una alta correlación negativa. La variable Price tiene una correlación moderada con las variables continuas.





Al analizar las distribuciones las variables categóricas con respecto a la variable de respuesta, es evidente que la variable 'State' parecería no tener una influencia significativa en el promedio de la variable precio, siendo el estado con mayor precio Wyoming, y el de menor precio DC. La marca más valorada es Bentley y la menos costosa Pontiac. Tanto las variables 'Model' y 'Make' parecen afectar de manera significativa la variable del precio.

Dada la descripción anterior, se toma la decisión de no realizar ninguna transformación de las variables para realizar el ejercicio, por lo que procederemos a procesar los datos para ingresarlos al modelo.

```
1 dTrain = dataTraining.copy()
2 dTest = dataTesting.copy()
```

```
1 dTrain = pd.get_dummies(dTrain)
2 dTest = pd.get_dummies(dTest)
```

Dividimos los conjuntos de entrenamiento y validación del modelo.

```
1 XTotal = dTrain.drop(columns = {"Price"})
2 yTotal = dTrain["Price"]
3 XTrain, XTest, yTrain, yTest = train_test_split(XTotal, yTotal, test_size=0.33, random_state=0)
```

Como se observa, se divide la variable de interés o variable respuesta, que para el caso de negocio es la variable 'Price' y se realiza la transformación de las variables categóricas a dummies. Al utilizar solo variables dummies, se evita la necesidad de preprocesar las variables categóricas y se puede entrenar el modelo directamente con estas variables. Sin embargo, en algunos casos, incluir variables numéricas junto con las variables dummies puede mejorar el rendimiento del modelo.

Selección y calibración del modelo

Luego de realizar algunas pruebas con modelos de regresión y de ensamblaje como Bagging, Random Forest y XGBoost, se toma la decisión de elegir y calibrar el modelo XGBoost debido a que es conocido por ser altamente eficiente y escalable, lo que lo hace adecuado para grandes conjuntos de datos y problemas complejos, y en los resultados preliminares obtuvo un mejor desempeño que otros modelos evaluados (MSE: Random Forest = 6.353, Bagging = 7.205, XGBoost = 5.626).

Una vez seleccionado este modelo, se procede a efectuar una macro calibración con el diccionario 'param_grid', donde guardamos posibles valores que pueden tomar los hiper parámetros del modelo. Una vez definido el diccionario de datos, es utilizar la función 'GridSearchCV' que procederá a encontrar el mejor modelo por criterio de MSE. Se realiza la calibración seleccionando parámetro de 3 pliegues para la validación cruzada del modelo (cv=3) de la función.

```
: regressor=xgb.XGBRegressor(eval_metric='rmse')

: param_grid = {"max_depth": [7, 8],
               "n_estimators" = [700,800],
               "learning_rate": [0.02, 0.1]}

: search = GridSearchCV(regressor, param_grid, cv=3).fit(XTrain, yTrain)
print("Mejores Parametros ",search.best_params_)

Mejores Parametros {'learning_rate': 0.1, 'max_depth': 8}

: regressor=xgb.XGBRegressor(learning_rate = 0.1,
                             n_estimators = 700,
                             max_depth = 8,
                             eval_metric='rmse')
```

Esta calibración permite determinar que los valores que minimizan el MSE para los hiper parámetros 'learning rate' y 'max_depth' son 0.1 y 8 respectivamente.

Si bien el 'learning rate' tiene un valor bajo, esto es positivo para este ejercicio, dado que se puede producir un modelo más generalizado, a pesar de tener un aprendizaje lento, caso contrario a tener un 'learning rate' alto, que termine generando sobre ajuste en los datos. El valor predeterminado del parámetro de 'learning rate' es 0.1, lo que suele ser un buen punto de partida para muchos conjuntos de datos y en especial para el tamaño del conjunto que ahora se está entrenando.

Por otro lado, recordando que el 'max depth' controla la cantidad de ramificaciones que se permiten en cada árbol de decisión y que por default este tiene un valor de 6 para este modelo, es interesante que la función sugiera un valor óptimo más alto (8). Esto podría significar que el modelo óptimo requiere crear ramificaciones más profundas en sus respectivos árboles de decisión, lo que a su vez concede que el modelo capture relaciones más complejas en los datos de entrenamiento.

```
regressor=xgb.XGBRegressor(learning_rate = 0.1,
                           n_estimators = 700,
                           max_depth = 8,
                           eval_metric='rmse')
```

Al identificar que el modelo óptimo fue obtenido por una mayor tasa de aprendizaje, se realiza una segunda calibración del modelo, pero se aumenta el valor máximo de 'Learning Rate' para probar con una tasa de 0.2, adicional probar el cambio del hiper parámetro 'n_estimators' entre 500 y 700, con el fin de que el modelo pueda tomar el número óptimo de árboles en un rango sin entrar en sobreajuste.

```
1 param_grid = {"max_depth": [7, 8],  
2               "n_estimators" = [500,700],  
3               "learning_rate": [0.1, 0.2]}
```

```
1 regressor=xgb.XGBRegressor(learning_rate = 0.2,  
2                             n_estimators = 700,  
3                             max_depth = 8,  
4                             eval_metric='rmsle')
```

Con esta segunda calibración se halla que los valores de los hiper parámetros que optimizan el desempeño del modelo XGBoost, son 'learning_rate' igual a 0.2, 'n_estimators' igual a 700 y 'max_depth' igual a 8.

Entrenamiento del modelo

Con los hiper parámetros calibrados se realiza el entrenamiento del modelo XgBoost, creando el modelo 'regressor', seleccionando estimators igual a 700, max_depth igual a 8 y una tasa de aprendizaje de 0.2.

Tomando como referencia el conjunto de entrenamiento Xtrain con el 67% de los datos y un conjunto de validación Xtest con el 33% restante

```
regressor=xgb.XGBRegressor(learning_rate = 0.2,  
                           n_estimators = 700,  
                           max_depth = 8,  
                           eval_metric='rmsle')  
  
regressor2.fit(XTrain, yTrain)  
  
y_pred = regressor.predict(XTest)  
  
y_pred = regressor.predict(XTest)  
  
mse = mean_squared_error(yTest, y_pred, squared=False)  
mse  
3517.54262871653
```

El desempeño del modelo óptimo muestra un MSE óptimo de 3.517, lo cual indica que es el mínimo error promedio entre el predictivo del modelo y el valor real de la prueba. Este desempeño permite inferir que el modelo realiza una proyección promedio óptima del precio de los vehículos teniendo en cuenta los parámetros anteriormente calibrados. Finalmente, se elige este modelo como el mejor modelo.

Disponibilización del modelo

Una vez se elige el modelo con el mejor desempeño en términos de MSE, se procede a disponibilizar en la nube:

Creación de la API predictora

Creamos el archivo .plk con el modelo ajustado de nuestra preferencia.

```
1 XTotal = dTrain.drop(columns ={"Price"})
2 yTotal = dTrain["Price"]
3 XTrain, XTest, yTrain, yTest = train_test_split(XTotal, yTotal, test_size=0.33, random_state=0)
```

```
1 regressor=XGBRegressor(learning_rate = 0.2,
2                         n_estimators = 700,
3                         max_depth = 8,
4                         eval_metric='rmsle')
```

```
1 regressor.fit(XTrain, yTrain)
```

```
XGBRegressor(base_score=None, booster=None, callbacks=None,
             colsample_bylevel=None, colsample_bynode=None,
             colsample_bytree=None, early_stopping_rounds=None,
             enable_categorical=False, eval_metric='rmsle', feature_types=None,
             gamma=None, gpu_id=None, grow_policy=None, importance_type=None,
             interaction_constraints=None, learning_rate=0.2, max_bin=None,
             max_cat_threshold=None, max_cat_to_onehot=None,
             max_delta_step=None, max_depth=8, max_leaves=None,
             min_child_weight=None, missing=nan, monotone_constraints=None,
             n_estimators=700, n_jobs=None, num_parallel_tree=None,
             predictor=None, random_state=None, ...)
```

```
1 joblib.dump(regressor, 'Proyecto_1/car_listing_XGB.pkl', compress=3)
```

```
['Proyecto_1/car_listing_XGB.pkl']
```

Creación función datos de entrada para predicción

Creamos la función que generará la muestra para predecir el precio del automóvil, con los valores que insertamos como entradas en nuestra interfaz https.

```
import pandas as pd
import joblib
import sys
import os

def predict_price(features):

    regressor = joblib.load(os.path.dirname(__file__) + '/car_listing_XGB.pkl')
    #regressor = joblib.load(os.path.dirname(__file__) + '\car_listing_XGB.pkl')
    print(regressor)

    dataTraining = pd.read_csv('https://raw.githubusercontent.com/albahnsen/MIAD_ML_and_NLP/main/datasets/dataTrain_carListings.zip')
    dataTraining['State'] = dataTraining['State'].str.replace(" ", "")
    dataTraining['Model'] = dataTraining['Model'].str.replace(" ", "")
    dataTraining['Make'] = dataTraining['Make'].str.replace(" ", "")
    dTrain = dataTraining.copy()
    dTrain = pd.get_dummies(dTrain)
    columns = dTrain.columns
    df1 = dTrain.iloc[:,0:]
    df1[df1 > 0] = 0

    State_ = 'State_'+features[2]
    Make_ = 'Make_'+features[3]
    Model_ = 'Model_'+features[4]
    #
    df1['Year'] = int(features[0])
    df1['Mileage'] = int(features[1])
    df1[State_] = int(1)
    df1[Make_] = int(1)
    df1[Model_] = int(1)

    # Make prediction
    p1 = regressor.predict(df1.drop('Price', axis=1))[0]

    return p1
```

Api de ejecución en la web

Creamos el código para la interfaz https para ejecutar la predicción.

```
from flask import Flask
from flask_restx import Api, Resource, fields
import joblib
#from Proyecto_1.model_XGB import predict_price
from flask_cors import CORS
from model_XGB import predict_price

app = Flask(__name__)
CORS(app) # Enable CORS for all routes and origins

api = Api(
    app,
    version='1.0',
    title='Car price Prediction API',
    description='Car price Prediction API')

ns = api.namespace('predict',
    description='Car price regressor')

parser = api.parser()

parser.add_argument(
    'Año',
    type=str,
    required=True,
    help='Año del modelo del carro',
    location='args')

parser.add_argument(
    'Millaje',
    type=str,
    required=True,
    help='Millas recorridas',
    location='args')

parser.add_argument(
    'Estado',
    type=str,
    required=True,
    help='Ubicación geográfica del carro',
    location='args')

parser.add_argument(
    'Marca',
    type=str,
    required=True,
    help='Marca del carro',
    location='args')

parser.add_argument(
    'Modelo',
    type=str,
    required=True,
    help='Modelo de la marca seleccionada',
    location='args')

resource_fields = api.model('Resource', {
    'result': fields.String,
})

@ns.route('/')
class CarApi(Resource):

    @api.doc(parser=parser)
    @api.marshal_with(resource_fields)
    def get(self):
        args = parser.parse_args()
        return {
            "result": predict_price([args['Año'],args['Millaje'],args['Estado'],args['Marca'],args['Modelo']])
        }, 200

if __name__ == '__main__':
    app.run(debug=True, use_reloader=False, host='0.0.0.0', port=5000)
```

Paquetes en la instancia

Subimos los paquetes a la instancia creada en AWS.

```
ubuntu@ip-172-31-8-179:~/Sergio/MIAD_ML_NLP_2023/Proyecto_1$ ls
__pycache__  api_cars.py  car_listing_XGB.pkl  model_XGB.py
ubuntu@ip-172-31-8-179:~/Sergio/MIAD_ML_NLP_2023/Proyecto_1$
```

i-04297c16d5fb91356 (ServerML)

PublicIPs: 18.222.20.218 PrivateIPs: 172.31.8.179

Ejecución

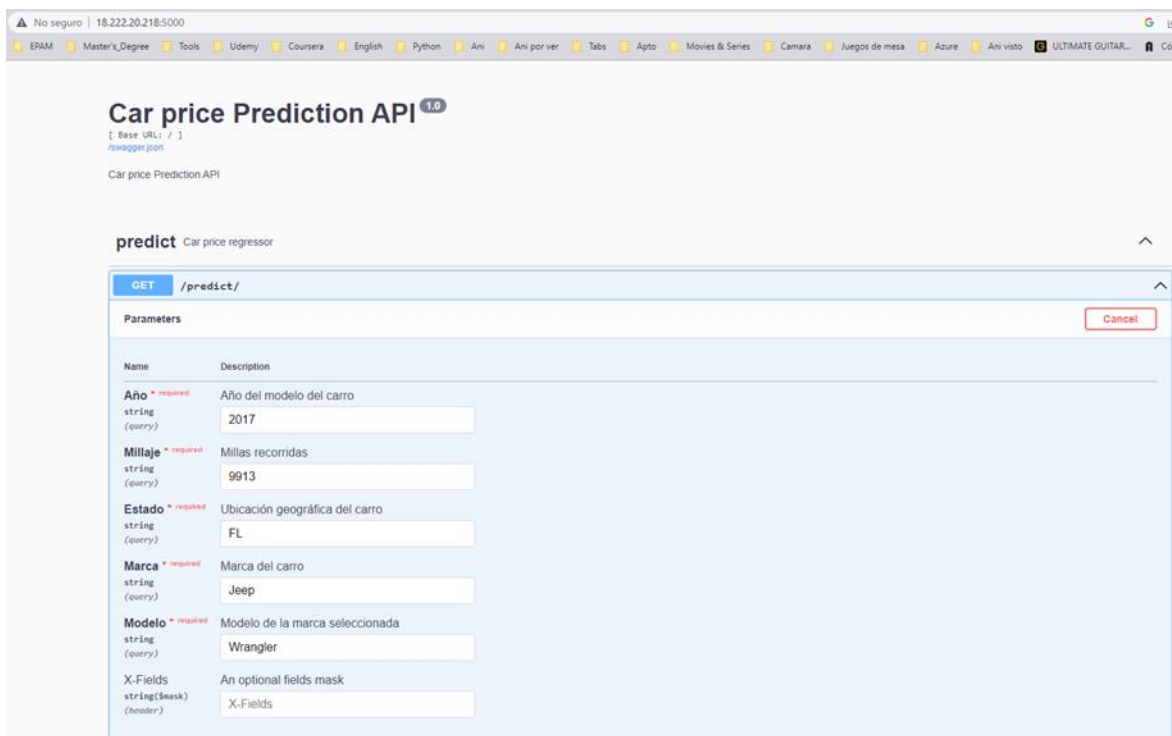
Ejecutamos nuestra api llamada api_cars.py

```
ubuntu@ip-172-31-8-179:~/Sergio/MIAD_ML_NLP_2023/Proyecto_1$ python3 api_cars.py
* Serving Flask app 'api_cars'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://172.31.8.179:5000
Press CTRL+C to quit
```

Respuesta

Nos dirigimos a nuestra versión web e insertamos los valores del carro para el cual deseamos predecir su precio

Predicción 1:



The screenshot shows a web browser window with the address bar displaying "18.222.20.218:5000". The page title is "Car price Prediction API 1.0". Below the title, there is a Swagger UI interface for the API. The "predict" endpoint is selected, and the "GET /predict/" method is shown. The "Parameters" section is expanded, showing a table of input fields:

Name	Description
Año * required	Año del modelo del carro
string (query)	2017
Millaje * required	Millas recorridas
string (query)	9913
Estado * required	Ubicación geográfica del carro
string (query)	FL
Marca * required	Marca del carro
string (query)	Jeep
Modelo * required	Modelo de la marca seleccionada
string (query)	Wrangler
X-Fields	An optional fields mask
string(mask) (header)	X-Fields

Execute
Clear

Responses Response content type: application/json

Curl

```
curl -X 'GET' \
  'http://18.222.20.218:5000/predict/?AÑO=2017&Millaje=9913&Estado=FL&Marca=Jeep&Modelo=Wrangler' \
  -H 'accept: application/json'
```

Request URL

```
http://18.222.20.218:5000/predict/?AÑO=2017&Millaje=9913&Estado=FL&Marca=Jeep&Modelo=Wrangler
```

Server response

Code	Details
200	<div> <p>Response body</p> <pre>{ "result": "36403.05" }</pre> <p>Response headers</p> <pre>access-control-allow-origin: * connection: close content-length: 29 content-type: application/json date: Sat, 29 Apr 2023 07:08:23 GMT server: Werkzeug/2.2.3 Python/3.10.6</pre> </div>

Responses

Code	Description
200	Success

Example Value | Model

```
{
  "result": "string"
}
```

El precio que nos muestra es \$36403.05

Predicción 2:

predict Car price regressor

GET /predict/
Cancel

Parameters

Name	Description
Año * required integer (query)	Año del modelo del carro <input type="text" value="2014"/>
Millaje * required integer (query)	Millas recorridas <input type="text" value="28729"/>
Estado * required string (query)	Ubicación geográfica del carro <input type="text" value="OH"/>
Marca * required string (query)	Marca del carro <input type="text" value="Cadillac"/>
Modelo * required string (query)	Modelo de la marca seleccionada <input type="text" value="SRXLuxury"/>
X-Fields string(\$mask) (header)	An optional fields mask <input type="text" value="X-Fields"/>

Execute
Clear

```
Curl
curl -X 'GET' \
  'http://3.145.44.15:5000/predict/?ANC38B1o=2014&MillaJe=28729&Estado=OH&Marca=Cadillac&Modelo=SRXLuxury' \
  -H 'accept: application/json'

Request URL
http://3.145.44.15:5000/predict/?ANC38B1o=2014&MillaJe=28729&Estado=OH&Marca=Cadillac&Modelo=SRXLuxury

Server response
Code    Details
200
Response body
{
  "result": "26220.184"
}
Response headers
access-control-allow-origin: *
connection: close
content-length: 30
content-type: application/json
date: Sun, 30 Apr 2023 21:02:08 GMT
server: Werkzeug/2.2.3 Python/3.10.6
```

El precio que nos muestra es \$26220.184

Conclusiones

- Los modelos XGBoost continúan mostrando su alta capacidad predictiva, escalabilidad y eficiencia para casos donde se tienen bases de datos con altos números de registros y de variables. Su modelo interno de ensamblaje secuencial basado en árboles de decisión y su gran capacidad para capturar relaciones no lineales entre variables, lo hicieron un modelo ideal para la predicción de precios de vehículos usados.
- La calibración de los hiper parámetros de un modelo XGBoost ha permitido obtener un buen resultado en cuanto a la predicción del conjunto de datos del precio de los vehículos, la optimización de los hiper parámetros mediante la validación cruzada ha mejorado la estimación y con ello la reducción del MSE y RMSE.
- El uso de variables categóricas que contienen una gran cantidad de clases produce un aumento en la dimensionalidad del conjunto de datos, lo que puede llevar a problemas de sobreajuste y velocidad en procesamiento. Sin embargo, en el presente caso el modelo XGBoost mostró ser un modelo altamente escalable, capaz de manejar grandes conjuntos de datos. Además, se puede ejecutar en paralelo en múltiples CPU y GPU, lo que lo hace ideal para problemas de gran escala.
- Cuando existen variables que tienen alta correlación como Year y Mileage, el modelo XGBoost demostró que cuenta con un conjunto de técnicas de regularización que ayudan a prevenir el sobreajuste y mejorar la generalización del modelo. Esto lo hace ideal para problemas de alta dimensionalidad.