Proyecto 1 ML & PLN: Predicción de precios de vehículos

Team 16: Diego González, Sandra Rodríguez, Hugo Ruíz

1. Análisis Exploratorio de Datos

Antes de iniciar el alistamiento de los datos para su posterior entrenamiento, se realizó el análisis de la información suministrada en el conjunto de datos de nombre data Training.

El análisis se dividió en tres pasos los cuales se muestran a continuación:

Paso 1: Visualización de los datos

	Price	Year	Mileage	State	Make	Model
0	34995	2017	9913	FL	Jeep	Wrangler
1	37895	2015	20578	OH	Chevrolet	Tahoe4WD
2	18430	2012	83716	TX	BMW	X5AWD
3	24681	2014	28729	OH	Cadillac	SRXLuxury
4	26998	2013	64032	CO	Jeep	Wrangler

En la tabla se observa que el conjunto de datos tiene 6 variables: Price, Year, Mileage, State, Make y Model. Fácilmente, podemos identificar que hay 3 variables numéricas y 3 categóricas:

- Variables numéricas: Price, Year y Mileage
- Variables categóricas: State, Make y Model

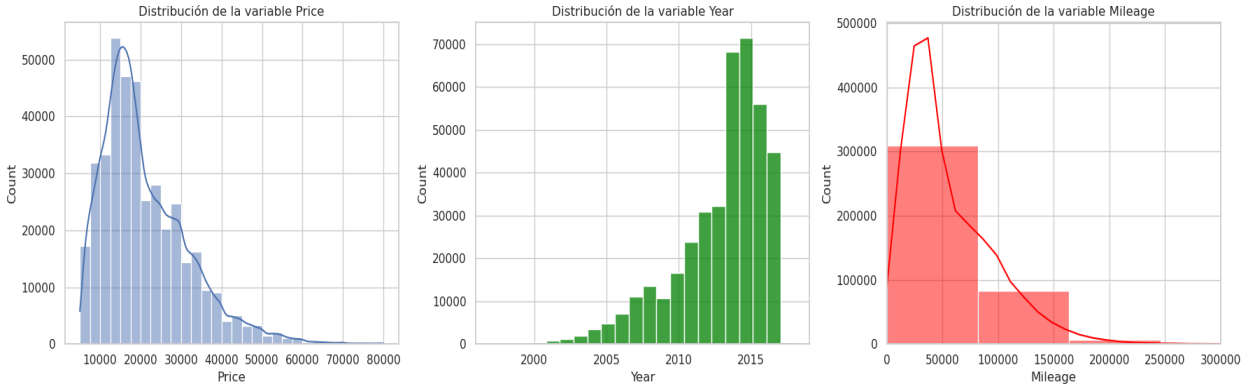
Paso 2: Estadísticas descriptivas

	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

Basados en estos resultados, se puede concluir:

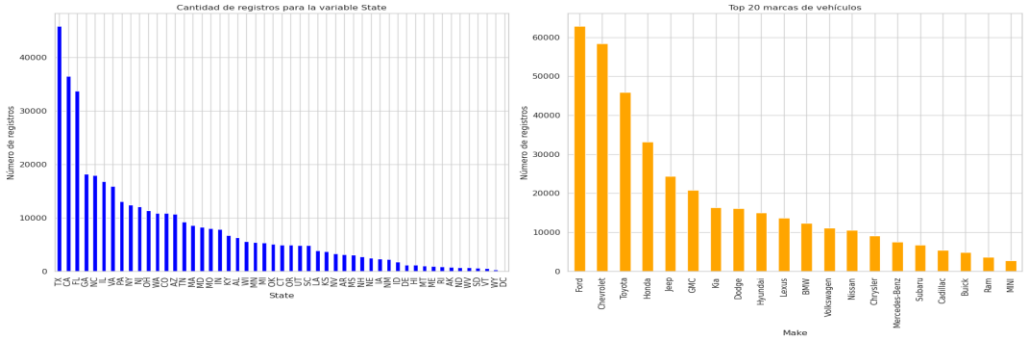
1. El conjunto de datos tiene 400.000 registros
2. No hay valores negativos en las variables
3. La variable año tiene un rango de valores entre 1997 y 2018

Paso 3. Gráficos univariados y bivariados



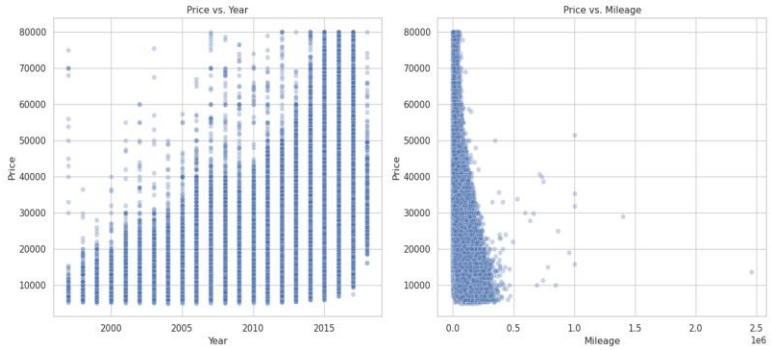
En esta gráfica se observan los histogramas correspondientes a las 3 variables numéricas. De esta se puede identificar que:

- La distribución de la variable Price es sesgada a la derecha
- Los datos contienen más información de los años 2014 en adelante
- La distribución de la variable Mileage también es sesgada a la derecha

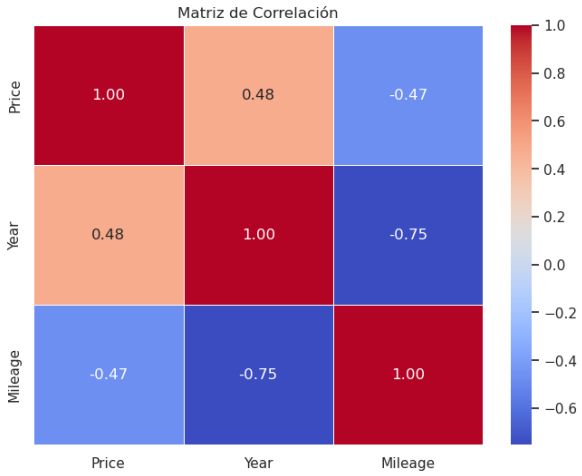


De igual manera a la gráfica anterior, de la gráfica de State y Make se tiene:

- Los 3 estados que más datos tienen dentro del dataset son: Texas (TX), California (CA) y Florida (FL)
- La marca que más datos reporta es: Ford

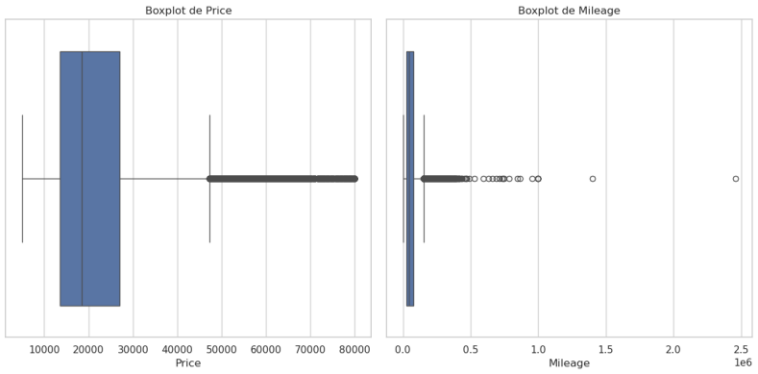


De la gráfica de Year vs Price, podemos concluir que mientras más reciente el año del vehículo, más alto es el precio. Con respecto de la gráfica de Mileage vs Price, se evidencia que mientras más millas tiene un vehículo menor es su precio.



De la matriz de correlación se precisa:

- Price y Year están correlacionadas positivamente como se vio en la gráfica anterior. Adicionalmente, se tiene que la correlación lineal es de 0.48.
- Price y Mileage están correlacionados negativamente. Adicionalmente, se tiene que la correlación lineal es igual a -0.47.
- Year y Mileage están correlacionadas negativamente. Lo cual si se analiza un poco más es intuitivo, ya que si un vehículo es más reciente (año más reciente al actual), la probabilidad de que no tenga tantas millas recorridas es menor.



Con el objetivo de buscar valores atípicos, se construyó para las variables Price y Mileage el gráfico de cajas. De donde se observa que en cada variable hay valores fuera del rango intercuartil, sin embargo, para este ejercicio se tomarán como datos correctos teniendo en cuenta la naturaleza de la información.

Finalmente, con el siguiente código se buscaron valores nulos y únicos:

```
# Calcular los valores faltantes de cada columna
missing_values = dataTraining.isnull().sum()
# Calcular el número de entradas únicas por cada columna
unique_entries = dataTraining.nunique()
```

Con los siguientes resultados:

```
Price      0
Year       0
Mileage    0
State      0
Make       0
Model      0
dtype: int64
Price       35867
Year           22
Mileage    130600
State          51
Make           38
Model         525
dtype: int64
```

De lo anterior, se concluye que el dataset no tiene valores nulos y que hay una gran cantidad de modelos, marcas y estados. Esto último es importante de cara al procesamiento de los datos a través de codificación usando variables dummy.

2. Procesamiento de datos

Primero, se hizo feature engineering sobre las variables de los conjuntos de entrenamiento. Con el siguiente código se crearon 2 variables:

- Age: Corresponde a la antigüedad del vehículo con base en el año actual
- CarModel: Concatenación de las variables Make y Model

```python
current_year = 2024

# Calcular la edad de cada vehículo
dataTraining['Age'] = current_year - dataTraining['Year']
dataTesting['Age'] = current_year - dataTesting['Year']
```
✓ 0.0s Python

```python
dataTraining['CarModel'] = dataTraining['Make'] + ' ' + dataTraining['Model']
dataTesting['CarModel'] = dataTesting['Make'] + ' ' + dataTraining['Model']
```
✓ 0.0s Python

```python
dataTraining.drop(['Year', 'Make', 'Model'], axis=1, inplace=True)
dataTesting.drop(['Year', 'Make', 'Model'], axis=1, inplace=True)
```
✓ 0.0s Python

Como paso siguiente, se identificaron las variables numéricas y categóricas para acto seguido y una vez hecha la partición en train y test, aplicar un escalamiento a las variables numéricas y el encoding one hot a las categóricas. El paso anterior se hizo a través de la función ColumnTransformer de Sckit-Learn, mientras que el paso de separación se hizo de la manera usual con la instrucción train_test_split del mismo paquete. Finalmente, se ajustaron los datos de entrenamiento y prueba con base en lo anteriormente descrito y se validó el tamaño de los conjuntos creados (el objeto que almacena los datos una vez creados es uno de tipo Sparse Matrix que de entrada no permite ver la estructura en forma "amigable). Todos los pasos anteriores se resumen en las siguientes líneas de código:

```python
# Separación de los datos de entrenamiento en X y y
X = df_Training.drop('Price', axis=1)
y = df_Training['Price']
```
✓ 0.0s Python

```python
# Identificación de las variables por tipo
categorical_features = ['State', 'CarModel']
numerical_features = ['Mileage', 'Age']
```
✓ 0.0s Python

```python
# Train y Test split
from sklearn.model_selection import train_test_split

# Partición de los datos
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=43)
```
✓ 0.0s Python

```python
# Procesador para aplicar escalamiento y encoding
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer

preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_features),
        ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_features)
    ],
    remainder='passthrough')
```
✓ 0.0s Python

```python
# Ajuste de los datos de entrenamiento
X_train_processed = preprocessor.fit_transform(X_train)
```
✓ 0.1s Python

```python
# Ajuste de los datos de prueba
X_test_processed = preprocessor.transform(X_test)
```
✓ 0.0s Python

```python
# Validación del tamaño de los datos de entrenamiento y prueba
print(X_train_processed.shape)
print(X_test_processed.shape)
```
✓ 0.0s Python

```
(300000, 589)
(100000, 589)
```

3. Entrenamiento de modelos

Para el entrenamiento, se probaron los siguientes modelos: regresión lineal múltiple, regresión lineal con penalización ridge, random forest y XGBoost. Como métrica de evaluación se usó el RMSE teniendo en cuenta que este es el que se usa en la competencia de Kaggle asociada al proyecto.

a. Regresión Lineal

Como primer modelo se construyó una regresión lineal con solo las variables numéricas Milage y Age.

Tanto el proceso como los resultados de este se muestran a continuación:

```python
# Regresión con dos variables Mileage y Antiguedad

# Importación de librerías
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Preparación de los datos
X_lr = df_Training[['Mileage', 'Age']]

# Partición de los datos
X_train, X_test, y_train, y_test = train_test_split(X_lr, y, test_size=0.25, random_state=43)

# Configuración del modelo
model = LinearRegression()

# Entrenamiento del modelo
model.fit(X_train, y_train)

# Predicción de precios
y_pred = model.predict(X_test)

# Cálculo del RMSE
rmse = np.sqrt(mean_squared_error(y_test, y_pred))

print(f'Root Mean Squared Error: {rmse}')
```

✓ 0.0s Python

```
Root Mean Squared Error: 9249.622272701556
```

Para este modelo se tuvo que volver a hacer una selección que incluyera solo las variables Milage y Age, y la partición en train y test. Y se obtuvo un RMSE de 9249.6.

b. Regresión Lineal con todas las variables

Como segundo modelo se entrenó una regresión lineal con todas las variables. En este de nuevo se tomó el preprocesamiento que se hizo inicialmente. A continuación, se muestra el código y el RMSE del modelo:

```python
# Regresión con todas las variables

# Configuración del modelo
model = LinearRegression()
model.fit(X_train_processed, y_train)

# Predicción
y_pred = model.predict(X_test_processed)

# Evaluación
rmse = np.sqrt(mean_squared_error(y_test, y_pred))

print(f'Root Mean Squared Error: {rmse}')
```

✓ 1.4s Python

```
Root Mean Squared Error: 4406.145102554704
```

Con la inclusión de todas las variables se nota una mejora significativa en RMSE, pasando de uno de 9249.6 a uno de 4406.1.

c. Regresión lineal con penalización Ridge

Como tercer modelo se entrenó una regresión con penalización de tipo Ridge. Para esta primera aproximación, se hizo uso un factor Alpha igual a 0.01:

```python
# Regresión Ridge con todas las variables

# Importación de librerías
from sklearn.linear_model import Ridge

# Configuración
ridge_model = Ridge(alpha=0.01)

# Entrenamiento
ridge_model.fit(X_train_processed, y_train)

# Predicción
y_pred = ridge_model.predict(X_test_processed)

# Evaluación
rmse = np.sqrt(mean_squared_error(y_test, y_pred))

print(f'Root Mean Squared Error: {rmse}')
```

✓ 0.4s Python

```
Root Mean Squared Error: 4406.538695758921
```

Este modelo se reentrenó usando una grilla de valores para Alpha. A continuación, se muestra el código utilizado:

```python
# Regresión Ridge con búsqueda de mejor alpha

# Importación de librerías
from sklearn.linear_model import RidgeCV

# Alphas a validar
alpha_values = [0.1, 0.5, 1.0, 5.0, 10.0, 20.0, 50.0, 100.0]

# Configuración de modelo
ridge_cv_model = RidgeCV(alphas=alpha_values, store_cv_values=True)

# Entrenamiento
ridge_cv_model.fit(X_train_processed, y_train)

# Mejor alpha
best_alpha = ridge_cv_model.alpha_

# predicción
y_pred = ridge_cv_model.predict(X_test_processed)

# Evaluación
rmse = np.sqrt(mean_squared_error(y_test, y_pred))

print('RMSE:', rmse)
print('Mejor alpha:', best_alpha)
```

Python

```
RMSE: 4406.111634395623
Mejor alpha: 0.1
```

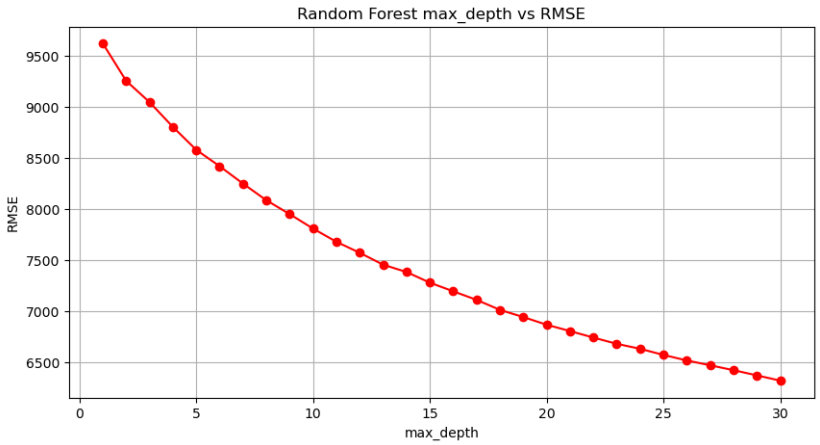
En este caso, se tiene que el mejor valor para Alpha de los distintos propuestos es 0.1. Sin embargo, el modelo no mejoró sustancialmente el valor del RMSE por lo que se procedió a entrenar otros modelos distintos.

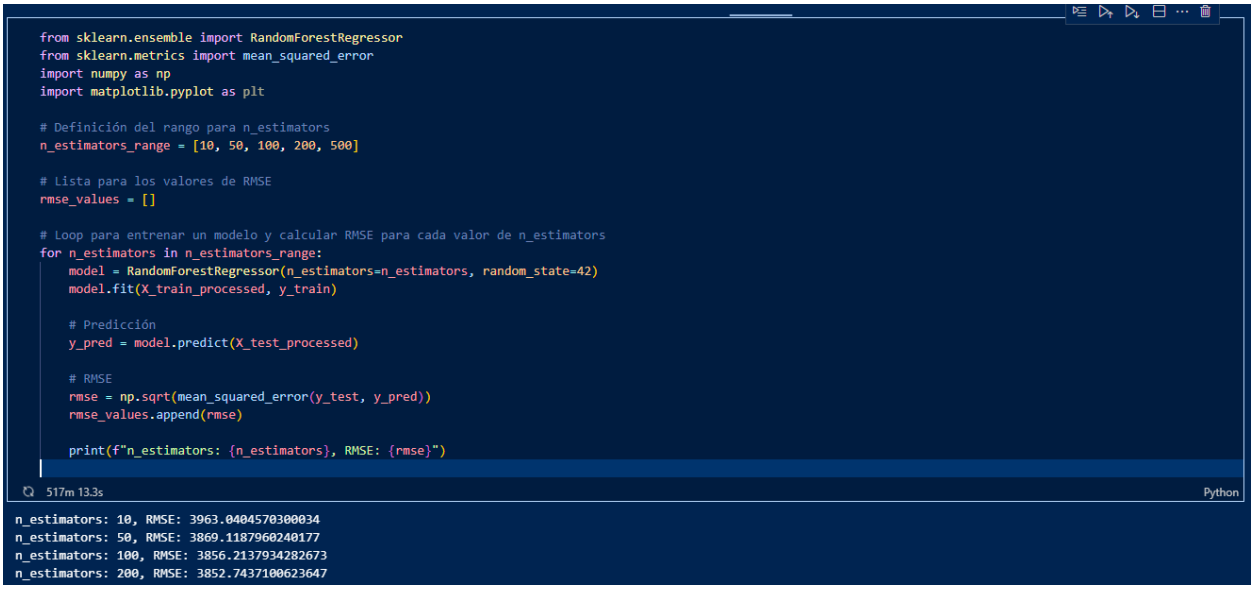
d. Random Forest

Para este modelo se trataron 3 metodologías distintas con respecto de la calibración de los parámetros: Ajuste manual, búsqueda unitaria de parámetros más relevantes y GridSearch.

Con respecto de la última metodología (GridSearch), no se obtuvieron buenos resultados ya que en ninguno de los equipos de los miembros del equipo se logró correr de manera satisfactoria el código y obtener un resultado.

Con respecto de la metodología en la que buscan de manera individual los mejores valores para cada parámetro, se obtuvieron los siguientes resultados para max_depth y n_estimators:





```python
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
import numpy as np
import matplotlib.pyplot as plt

# Definición del rango para n_estimators
n_estimators_range = [10, 50, 100, 200, 500]

# Lista para los valores de RMSE
rmse_values = []

# Loop para entrenar un modelo y calcular RMSE para cada valor de n_estimators
for n_estimators in n_estimators_range:
    model = RandomForestRegressor(n_estimators=n_estimators, random_state=42)
    model.fit(X_train_processed, y_train)

    # Predicción
    y_pred = model.predict(X_test_processed)

    # RMSE
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    rmse_values.append(rmse)

    print(f"n_estimators: {n_estimators}, RMSE: {rmse}")
```

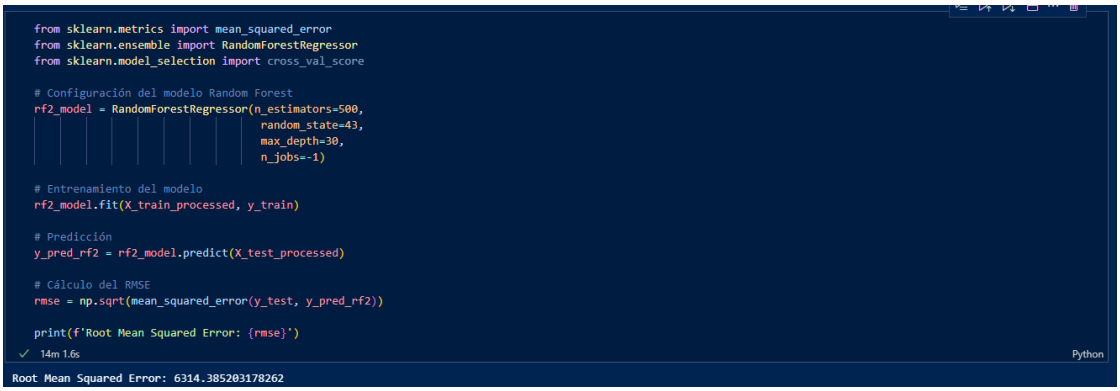
517m 13.3s Python

```
n_estimators: 10, RMSE: 3963.0404570300034
n_estimators: 50, RMSE: 3869.1187960240177
n_estimators: 100, RMSE: 3856.2137934282673
n_estimators: 200, RMSE: 3852.7437100623647
```

Así, con los resultados anteriores, se entrenó un modelo de Random Forest con los siguientes parámetros:

- n_estimators = 500 – Se tomó el siguiente parámetro como punto de partida teniendo en cuenta que a medida que este crece el RMSE disminuye.
- max_depth = 30

El código utilizado para el entrenamiento del modelo y los resultados del RMSE se muestran a continuación:



```python
from sklearn.metrics import mean_squared_error
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import cross_val_score

# Configuración del modelo Random Forest
rf2_model = RandomForestRegressor(n_estimators=500,
                                  random_state=43,
                                  max_depth=30,
                                  n_jobs=-1)

# Entrenamiento del modelo
rf2_model.fit(X_train_processed, y_train)

# Predicción
y_pred_rf2 = rf2_model.predict(X_test_processed)

# Cálculo del RMSE
rmse = np.sqrt(mean_squared_error(y_test, y_pred_rf2))

print(f'Root Mean Squared Error: {rmse}')
```

✓ 14m 1.6s Python

```
Root Mean Squared Error: 6314.385203178262
```

Finalmente, dado que los resultados del modelo no fueron mejores que los de los modelos anteriores, se optó por utilizar la metodología de ajuste manual en donde el número de árboles o n_estimators se igualó a 100.

```python
# Importación de librerías
from sklearn.ensemble import RandomForestRegressor

# Configuración del modelo
rf_model = RandomForestRegressor(n_estimators=100, random_state=43)

# Entrenamiento del modelo
rf_model.fit(X_train_processed, y_train)

# Predicción
y_pred_rf = rf_model.predict(X_test_processed)

# Evaluación
rmse_rf = np.sqrt(mean_squared_error(y_test, y_pred_rf))

print('RMSE:', rmse_rf)
```

✓ 80m 30.1s Python

```
RMSE: 3855.3120041979223
```

Finalmente, se entrenó un modelo de XGBoost con RandomSearch para encontrar los parámetros. A continuación, se muestran los resultados de los parámetros encontrados y el mejor RMSE:

```python
from sklearn.model_selection import RandomizedSearchCV
from xgboost import XGBRegressor
from sklearn.metrics import mean_squared_error



# Configurar el modelo XGBRegressor
xgb = XGBRegressor()

# Definir la distribución de parámetros para la búsqueda aleatoria
param_dist = {
    'n_estimators': np.arange(50, 400, 50),
    'learning_rate': np.linspace(0.01, 0.2, 10),
    'subsample': np.linspace(0.5, 1.0, 6),
    'max_depth': np.arange(3, 10, 1),
    'colsample_bytree': np.linspace(0.5, 1.0, 6),
    'min_child_weight': np.arange(1, 6, 1)
}

# Configurar RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=xgb,
    param_distributions=param_dist,
    n_iter=25,  # Puedes cambiar este número si necesitas más o menos iteraciones
    scoring='neg_mean_squared_error',
    cv=5,
    verbose=1,
    random_state=42,
    n_jobs=-1  # Usa todos los cores disponibles
)

# Ejecutar la búsqueda aleatoria
random_search.fit(X_train_processed, y_train)

# Mejores parámetros encontrados y el mejor score (convertido de MSE negativo a RMSE)
best_params = random_search.best_params_
best_score = np.sqrt(-random_search.best_score_)

print("Mejores parámetros:", best_params)
print("Mejor RMSE:", best_score)
```

Python

```
Fitting 5 folds for each of 25 candidates, totalling 125 fits
Mejores parámetros: {'subsample': 0.9, 'n_estimators': 250, 'min_child_weight': 3, 'max_depth': 9, 'learning_rate': 0.1788888888888889, 'colsample_bytree': 0.9}
Mejor RMSE: 3853.7949684498835
```

Con los parámetros encontrados anteriormente, se entrena el modelo y se valida su desempeño en los datos de prueba:

```python
from xgboost import XGBRegressor
from sklearn.metrics import mean_squared_error
import numpy as np

# Supongamos que random_search es tu objeto RandomizedSearchCV después de llamar a fit
best_params = random_search.best_params_

# Crear un nuevo modelo XGBRegressor con los mejores parámetros
best_xgb_model = XGBRegressor(**best_params)

# Entrenar el modelo con los mejores parámetros en el conjunto de entrenamiento completo
best_xgb_model.fit(X_train_processed, y_train)

# Opcional: Evaluar el modelo en el conjunto de prueba para ver cómo se desempeña
y_pred = best_xgb_model.predict(X_test_processed)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print(f'RMSE del modelo con los mejores parámetros: {rmse}')
```

Python

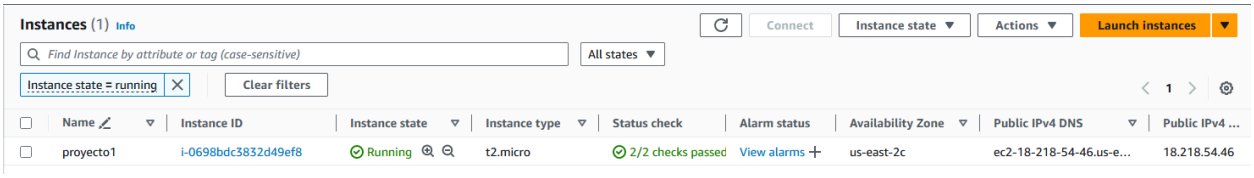
```
RMSE del modelo con los mejores parámetros: 3856.766538276313
```

Con base en todos los modelos entrenados anteriormente, se tiene eligió modelo de predicción para la competencia el Random Forest con ajuste manual, ya que este fue el que presentó menor RMSE de entre todos los modelos ajustados.

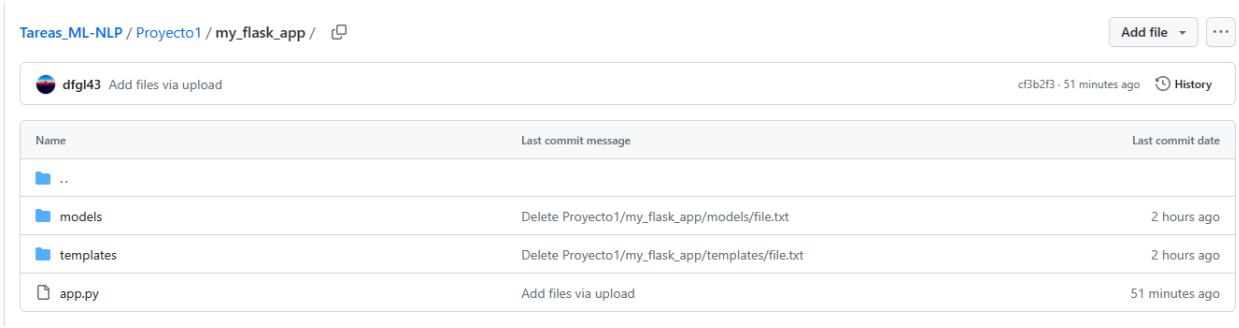
Nota: En los notebooks utilizados para los entrenamientos y cargados en el repositorio del proyecto, se evidencia el entrenamiento de otros modelos que no se incluyeron en este reporte ya que sus resultados no fueron mejores que los presentados en este documento.

4. Disponibilización del modelo

Para disponibilizar el modelo se creó una instancia EC2



Dentro del repositorio del proyecto se creó una estructura que facilita la publicación del servicio, como se muestra a continuación:



En la carpeta models, se almacena el modelo entrenado en formato joblib. Para la disponibilización del modelo, se usó una regresión lineal con las variables numéricas, esto con el objetivo de facilitar el procesamiento.

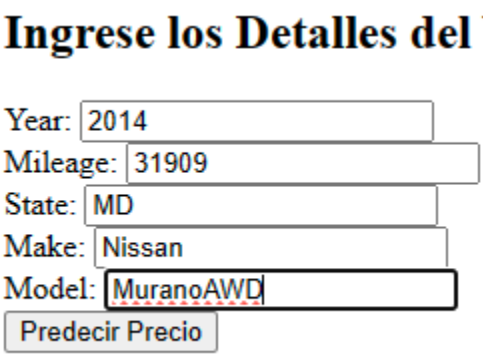
En la carpeta templates, se almacena un archivo de nombre index.html que contiene la estructura del servicio web que solicita las entradas y muestra las predicciones del modelo.

Finalmente, en el archivo app.py se guarda el código que llama la aplicación, procesa los datos, evalúa el modelo y muestra los resultados:

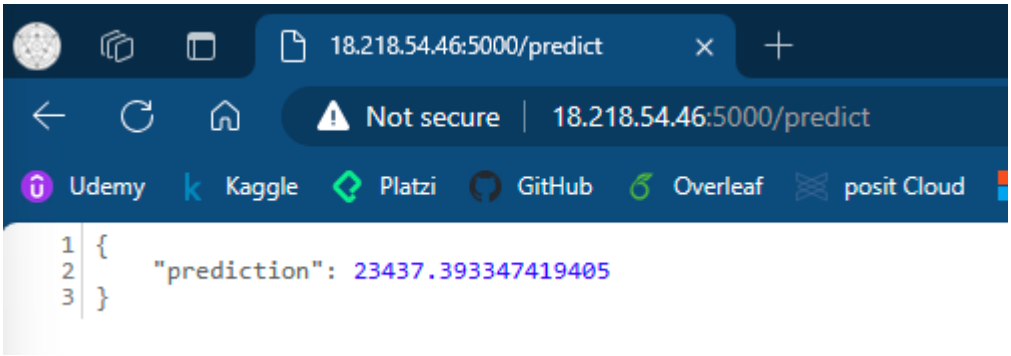
El servicio puede ser consultado en la siguiente dirección: http://18.218.54.46:5000. A continuación, se muestra la ejecución de dos pruebas hechas a los datos de set de validación:

Prueba 1

Datos:

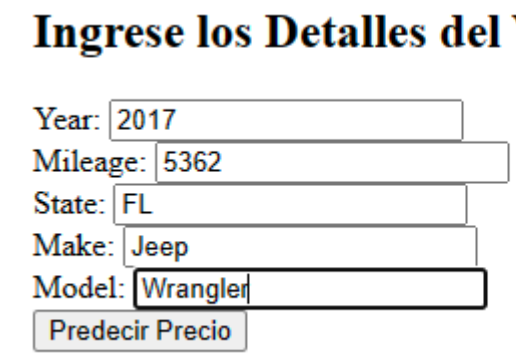


Resultados:

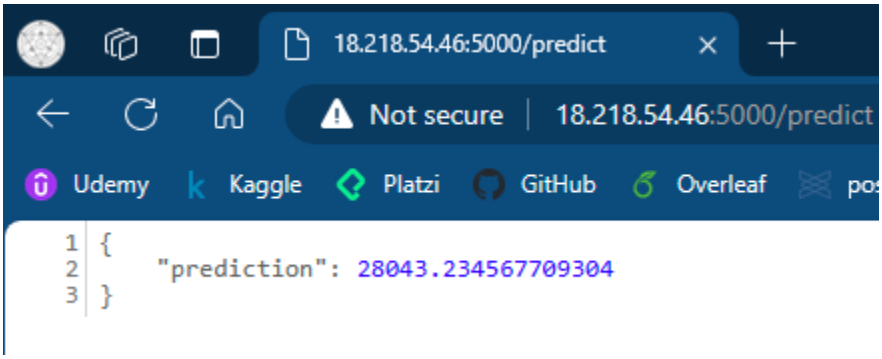


Prueba 2

Datos:



Resultados:



5. Conclusiones

1. En términos generales, la estructura de entrenamiento de este tipo de modelos sigue una receta que puede ser implementada sin problema siendo cuidadoso, sin embargo, se tiene como principal limitante los recursos de hardware disponibles.
2. El procesamiento de datos no vistos por el modelo es un paso en el que se debe tener cuidado, ya que cuando la estructura de los datos a predecir no sigue la misma que la de los datos de entrenamiento, se debe recurrir a diferentes estrategias para que los modelos funcionen correctamente.
3. La disponibilización de un modelo es un proceso que requiere de mucho cuidad y atención, sobretodo de cara al uso de herramientas distintas como lo son: los servicios de nube y su configuración, el control de versiones de los paquetes usados en el ambiente de entrenamiento y el que va a albergar el servicio a desplegar.

Nota: Los notebooks usados para la realización del proyecto y la carpeta creada para la aplicación se encuentran disponibles en el repositorio creado para el curso: https://github.com/dfgl43/Tareas_ML-NLP.git