



# INFORME PROYECTO 1 Y COMPETENCIA MIAD2024-12

## Predicción de precios de vehículos usados

Desarrollado por:

- KAREN YORLADY ROJAS GIRALDO
- JHOCEL DUVAN SUESCUN TORRES
- JUAN SEBASTIAN HERNANDEZ RAMIREZ
- JUAN PABLO MOGOLLÓN AVAUNZA

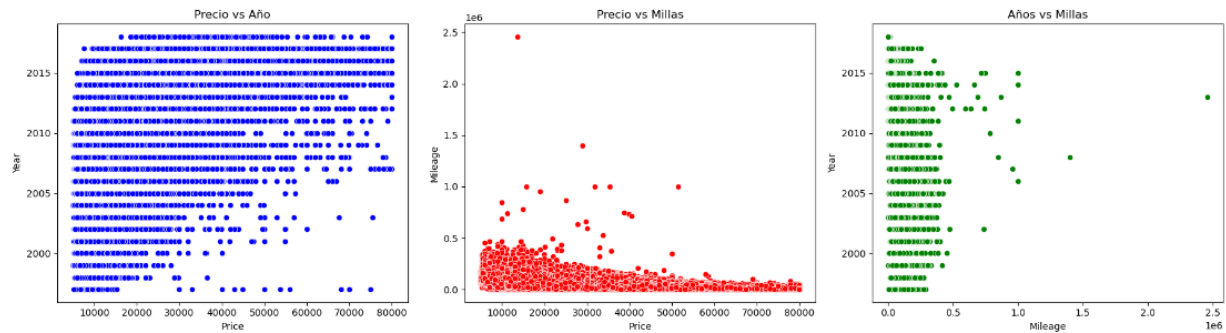
## 1 Exploración preliminar de los datos

- Se dispone de dos sets de datos: 'dataTrain\_carListings.zip' y 'dataTest\_carListings.zip', correspondientes a los datos de entrenamiento y de pruebas, el set de entrenamiento contiene 400 mil observaciones, y el de pruebas 100 mil.
- El set de datos de entrenamiento contiene seis columnas, cinco de ellas corresponden a predictores: 'Year', 'Mileage', 'State', 'Make', 'Model'; mientras que la columna 'Price' es la variable objetivo. El set de datos de pruebas contiene las mismas columnas, excepto la correspondiente a la variable objetivo 'Price'.
- No faltan datos ni en el set de entrenamiento ni en el de pruebas.
- Los predictores 'State', 'Make', 'Model' son categóricos. 'State' contiene 51 categorías, 'Make' contiene 38 categorías y 'Model' 525 categorías únicas. La combinación de 'Make' y 'Model' contiene 536 categorías únicas.
- El precio promedio es de 21149.9 dólares con un rango entre 5000 y 79999. El set de datos contiene modelos de los años 1997 a 2018. La distancia promedio que han recorrido son 55072 millas, con un rango entre 5 y 2.45 millones de millas.
- Todas las categorías de los predictores 'State' y 'Model' están contenidas tanto en el set de entrenamiento como de pruebas. La categoría 'Freightliner' del predictor 'Make' está contenida en los datos de entrenamiento, pero no en los datos de prueba.
- El precio tiene una correlación positiva con el año de fabricación del automóvil ( $R^2$ : 0.479765), y negativa con relación a la distancia recorrida ( $R^2$ : -0.471106). En otras palabras, entre más nuevo es el vehículo y menos distancia haya recorrido el precio es mayor. La

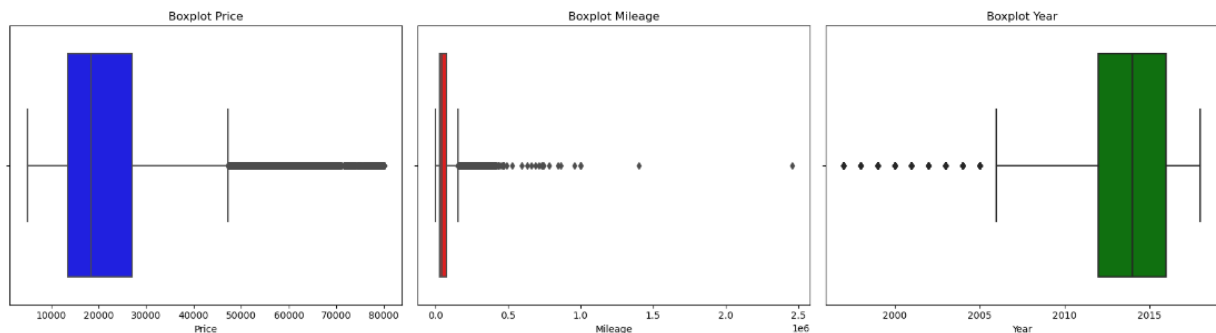
	Price	Year	Mileage
count	400000.000000	400000.000000	4.000000e+05
mean	21146.919312	2013.198125	5.507296e+04
std	10753.664940	3.292326	4.088102e+04
min	5001.000000	1997.000000	5.000000e+00
25%	13499.000000	2012.000000	2.584100e+04
50%	18450.000000	2014.000000	4.295500e+04
75%	26999.000000	2016.000000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

	Price	Year	Mileage
Price	1.000000	0.479765	-0.471106
Year	0.479765	1.000000	-0.751258
Mileage	-0.471106	-0.751258	1.000000

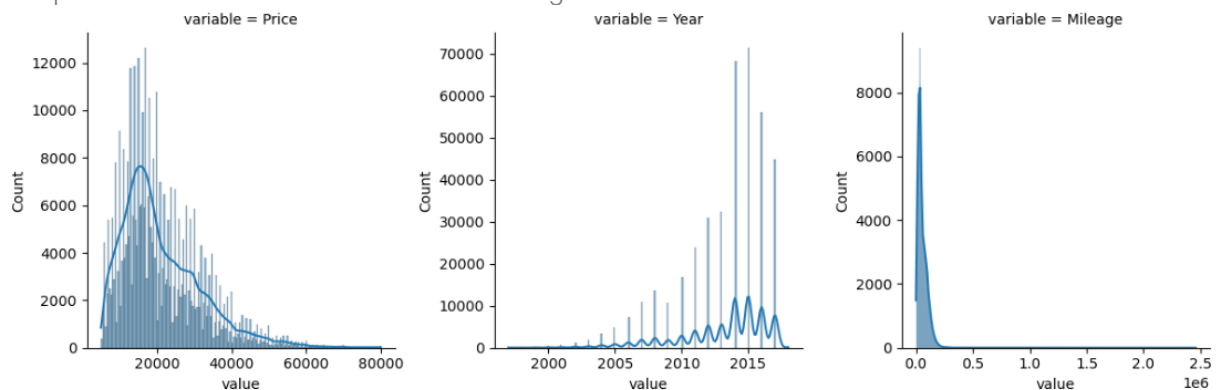
correlación entre la distancia recorrida y el año del vehículo es negativa ( $R^2: 0.751258$ ), es decir que entre más nuevo sea el vehículo menos distancia ha recorrido.



- h) Las gráficas de boxplot muestran algunos datos atípicos en la distribución de los datos de los tres predictores numéricos. Y la igual que se aprecia en los histogramas, se nota una concentración de los valores de precio y distancia recorrida a la izquierda de la distribución. Mientras que en año los valores se concentran a la derecha de la distribución, es decir a vehículos de fabricación más reciente.



- i) Los predictores de variables continuas no siguen una distribución normal.



## 2 Preprocesamiento de datos

- Teniendo en cuenta que para el modelo de predicción se usaran modelos basados en árboles, los cuales son pocos sensibles a la escala de las variables se decidió no escalar las variables.
- No hay datos faltantes en ninguna de las variables, por lo cual no fue necesario realizar imputación de valores.
- Las bases originales 'dataTrain\_carListings.zip' y 'dataTest\_carListings.zip' se almacenaron en los dataframe 'dataTraining' y 'dataTesting'.
- La combinación de los predictores 'Make' y 'Model' en un solo predictor 'Make\_Model' genera menos variables dummy que el codificar por separado. Teniendo en cuenta esto, en ambos dataframe se

concatenaron las columnas 'Make' y 'Model' creando la columna 'Make\_Model', se eliminaron las columnas 'Make' y 'Model', y utilizando el método OneHotEncoder de sklearn se crearon las variables dummy de las columnas 'State' y 'Make\_Model'. Al crear la instancia de 'OneHotEncoder' se utilizó el hiper parámetro drop = 'first', para eliminar la primera categoría de cada característica. A continuación, una fracción del código utilizado:

```
1. # Concatenar las variables Make y Model
2. df = dataTraining.copy()
3. df['Make_Model']=df['Make']+'_'+df['Model']
4.
5. # Filtrar el dataframe por las columnas de interes
6. df = df[['Price', 'Year', 'Mileage', 'State', 'Make_Model']]
7.
8. # Crear la instancia de OneHotEncoder y las columnas dummy para State y Make_Model
9. encoder = OneHotEncoder(drop='first', sparse=False) # Usamos drop='first' para eliminar la primera categoría en cada característica
10. colsToEncoded=['State', 'Make_Model']
11. dfCoded = pd.DataFrame(encoder.fit_transform(df[colsToEncoded]))
12.
13. df2 = df2Coded
14. df2Coded = pd.DataFrame(encoder.transform(df2[colsToEncoded]))
15.
16. # agregar las columnas de los predictores no transformados
17. dfCoded[['Price', 'Year', 'Mileage']]=df[['Price', 'Year', 'Mileage']]
18. dfCoded[['Year', 'Mileage']]=df[['Year', 'Mileage']]
19.
```

- e) Con el objeto de realizar pruebas de desempeño de los modelos antes de subirlos a la competencia el dataframe 'dataTraining' fue dividido en conjuntos de entrenamiento y prueba así:

```
1. # Para pruebas de desempeño separar dataTraing en bases de entrenamiento y pruebas
2. XTotal = dataTrainingCoded.drop(columns=['Price'])
3. yTotal = dataTrainingCoded[['Price']]
4. XTrain, XTest, yTrain, yTest = train_test_split(XTotal, yTotal, test_size=0.33, random_state=0)
```

- f) Nota: adicionalmente a lo descrito se probaron otras estrategias en el preprocesamiento de datos.
- No crear variables dummies y transformar el datatype de las variables categóricas a 'category' con el objeto de probar la funcionalidad del XGboost con variables categóricas.
  - No crear variables dummies y codificar las categóricas usando 'LabelEncoder', esto requirió asegurar que la codificación fuera igual para los predictores en los dataset de entrenamiento y pruebas.
  - Eliminar los valores atípicos en 'Price', 'Mileage' y 'Year' usando la técnica de rango intercuartílico.
  - En todos los casos los resultados de la predicción, comparados a través del 'RMSE' fueron inferiores a los obtenidos con el preprocesamiento descrito en el punto 2.4.

### 3 Calibración del modelo

- a) Teniendo en cuenta: que al crear las variables dummy de los predictores 'State' y 'Make\_Model' se crean más de 500 predictores; que el set de datos tiene 400 mil observaciones y que se trata de un problema de regresión, decidimos utilizar como modelo base para la predicción XGBoostRegressor. XGBoost tiene la capacidad de gestionar de forma eficiente problemas con conjuntos de datos grandes y con alta dimensionalidad de características, y es eficiente en términos de velocidad y uso de recursos computacionales. Además, es un algoritmo de boosting que combina múltiples modelos para mejorar su predicción, permitiéndole identificar relaciones no lineales entre las variables predictoras y de respuesta. Por otro lado, ofrece muchas opciones de ajuste de hiper parámetros

lo que permite optimizar el rendimiento del modelo según las necesidades de un problema específico.

- b) Consideramos pertinente calibrar los siguientes hiper parámetros: 'learning\_rate', 'gamma', 'colsample\_bytree', 'max\_depth' y 'n\_estimators'. La selección de estos hiper parámetros estuvo influenciada por la experiencia adquirida en el desarrollo del ejercicio anterior, en el que se identificó como cada uno de ellos afecta el poder predictivo y la complejidad del modelo.
- c) Adicionalmente probamos el parámetro 'enable\_categorical' que cuando es igual a 'True' permite que XGBoost reciba y procese predictores categóricos, para ello la columna del dataframe debe ser del tipo 'category'. No obstante, los resultados de las predicciones al usar esta opción sin transformar las columnas 'State' y 'Make' y 'Model' a sus dummies el poder predictivo del modelo disminuyo ostensiblemente.
- d) Para calibrar el modelo probamos tres métodos:
  - Calibración manual, a través de un procedimiento anidado con múltiples 'for' a través del cual probamos diferentes valores de los hiper parámetros, valores seleccionados a conveniencia nuestra.
  - GridSearchCV. Este método resulta eficiente ya que simplifica el código, permite encontrar la mejor combinación de parámetros entre todas las combinaciones de parámetros posibles de acuerdo con los lista de valores a probar en cada uno de los parámetros, y además permite seleccionar un parámetro de 'cross validation', el cual fijamos en dos (para reducir el tiempo de entrenamiento en la ejecución del taller). En el código que dejamos de ejemplo se generan 720 combinaciones posibles, las cuales son probadas con este método. Con este método probamos múltiples combinaciones de hiper parámetros, una vez obtenido la mejor combinación, dejamos fijos algunos hiper parámetros e iteramos sobre diferentes valores en uno de ellos hasta encontrar el mejor resultado predictivo. El código siguiente muestra un ejemplo de cómo utilizamos este método, para ahorrar tiempo de entrenamiento fijamos el hiper parámetro CV de GridSearchCV en 2. Los valores incluidos en 'param\_grid' generan 720 combinaciones posibles.

```
1. param_grid = {
2.     'learning_rate': [0.05, 0.1, 0.3, 0.5], # Valores a probar para learning rate
3.     'gamma': [0.5e6, 0.9e6, 1e6, 1.2e6], # Valores a probar para gamma
4.     'colsample_bytree': [0.2, 0.3, 0.4, 0.5, 0.8], # Valores a probar para colsample_bytree
5.     'max_depth': [8,9,10,11], # Valores para probar max_depth
6.     'n_estimators':[300, 500, 700]
7. }
8.
9. xgb_model = xgb.XGBRegressor(objective='reg:squarederror', random_state=0)
10. grid_search = GridSearchCV(estimator=xgb_model, param_grid=param_grid, cv=2, scoring='neg_mean_squared_error',
n_jobs=-1, verbose=1)
11. grid_search.fit(XTrain, yTrain)
```

- Dado que probamos todas las combinaciones posibles con " descartamos el uso de 'RandomSearchCV', debido a que este método en lugar de evaluar todas las combinaciones posibles de hiper parámetros (como en GridSearchCV), RandomSearchCV selecciona configuraciones de hiper parámetros de manera aleatoria dentro del espacio de búsqueda especificado.
- BayesSearchCV. Este método incluido en la biblioteca 'scikit-optimize', es un algoritmo de optimización bayesiana para la búsqueda de hiper parámetros en modelos de aprendizaje automático. Esta técnica utiliza información probabilística para dirigir la búsqueda hacia regiones prometedoras del espacio de hiper parámetros, siendo más eficiente que la búsqueda exhaustiva a través de GridSearchCV. A medida que se realizan más evaluaciones, el modelo probabilístico se ajusta y refina, adaptándose a las características del espacio de hiperparámetros y las observaciones realizadas. De otro lado, equilibra la exploración (buscar en nuevas áreas del espacio de hiper parámetros) con la explotación (utilizar información acumulada para seleccionar combinaciones más prometedoras), lo que puede conducir a la

identificación de mejores hiper parámetros de manera más efectiva. Este método tiene un parámetro propio muy importante que ayuda a regularizar el método y evitar sobreajuste, 'n\_iter'. De manera empírica encontramos que el mejor resultado lo obtuvimos con n\_iter=50.

- Tras probar las técnicas descritas concluimos que 'BayesSearchCV' resulta más eficiente en la búsqueda y selección de los hiper parámetros, evitando además la selección arbitraria de los mismos, permitiendo tener un espacio de búsqueda más amplio, lo que facilita la exploración y selección óptima de resultados.
- e) En relación con el poder predictivo del modelo, en general todos los hiper parámetros probados, tienen un comportamiento similar describiendo una curva cóncava en relación con el MSE del modelo y el valor del hiper parámetro. Es decir que a medida que se incrementa el valor del hiper parámetro el MSE disminuye hasta llegar a un valor óptimo en el que se obtiene el menor valor de MSE, a partir de este valor óptimo el incremento en el valor del hiper parámetro incrementa el valor del MSE; en otras palabras, disminuye el poder predictivo del modelo.
- a. colsample\_bytree = 0.4, al incrementar la proporción de características usadas, manteniendo el valor de los otros hiper parámetros el valor del MSE y el MAE tiende a aumentar, de igual forma ocurre con valores de 0.3, y 0.2.
  - b. gamma = 1000000.0, manteniendo los valores óptimos en los otros hiper parámetros probados, con valores inferiores a 500 mil en gamma la capacidad predictiva del modelo no se ve afectada, mientras que valores superiores a un millón deterioran su capacidad predictiva.
  - c. learning\_rate = 0.3, manteniendo los valores óptimos en los otros hiper parámetros probados, valores de learning\_rate inferiores a 0.25 afectan el desempeño del modelo requiriendo más tiempo de entrenamiento y disminuyendo su poder predictivo; mientras que valores superiores a 0.3, pueden disminuir el tiempo de entrenamiento, pero no aportan poder predictivo al modelo e incluso pueden deteriorarlo.
  - d. max\_depth = 9, probamos profundidades de 5 a 13 siendo la profundidad de 9 la que nos dio un mejor desempeño en combinación con los otros hiper parámetros probados. El desempeño predictivo del modelo con una profundidad de 8 es solo ligeramente inferior, no obstante profundidades menores afectaron negativamente el desempeño del modelo, al igual que profundidades mayores a 10, este posiblemente por un problema de sobreajuste.
  - e. n\_estimators = 600, este hiper parámetro tiene un gran efecto en el desempeño predictivo del modelo medido con RMSE y MAE, siendo 600 el valor con el que encontramos el mejor desempeño en combinación con los otros hiper parámetros calibrados. Con un valores menores o superiores a 600 obtuvimos un mayor RMSE y MAE.
  - f. Es importante anotar que los valores óptimos de cada hiper parámetro se ven afectados por el valor que se le da a los demás hiper parámetros probados.

## 4 Entrenamiento del modelo

- 4.1 En las diferentes corridas de GridSearchCV identificamos los mejores parámetros encontrados, creamos una instancia del modelo de XGBoost con estos parámetros y lo entrenamos con los datos de entrenamiento obtenidos utilizando el método 'train\_test\_split' de sklearn tal como se indicó antes. Evaluamos el desempeño del modelo utilizando como métricas RMSE y MAE. El código utilizado para implementar este punto es el siguiente:

```
1. # Obtener los mejores parámetros
2. best_params = grid_search.best_params_
3. print("Mejores parámetros encontrados:", best_params)
4.
5. # Entrenar un nuevo modelo con los mejores parámetros encontrados
6. best_xg_model = xgb.XGBRegressor(objective='reg:squarederror', **best_params, random_state=42)
```

```

7. best_xg_model.fit(XTrain, yTrain)
8. modelE1 = best_xg_model
9.
10. # Realizar predicciones utilizando el nuevo modelo y el conjunto de prueba
11. y_predXGB2 = best_xg_model.predict(XTest)
12.
13. # Evaluar el desempeño del modelo Bagging
14. score1 = mean_squared_error(yTest, y_predXGB2)
15. score2 = mean_absolute_error(yTest, y_predXGB2)
16.
17. print("\nDesempeño en los datos de test")
18. print(f"rmse: {score1**0.5:,.4f}")
19. print(f"mae: {score2:,.4f}")

```

4.2 Utilizando GridSearchCV se corrieron más de 500 combinaciones de hiper parámetros. Presentamos un resumen de los resultados de algunas de las corridas de GridSearchCV que incluye los mejores parámetros encontrados, así como las métricas de desempeño obtenidas (RMSE y MAE).

- a. Mejores parámetros encontrados: {'n\_estimators': 600}; Desempeño en los datos de test rmse: 3,530.7848 mae: 2,237.9993
- b. Mejores parámetros encontrados: {'learning\_rate': 0.1, 'max\_depth': 9, 'n\_estimators': 600}; Desempeño en los datos de test rmse: 3,752.5344 mae: 2,492.4308
- c. Mejores parámetros encontrados: {'colsample\_bytree': 0.4, 'max\_depth': 9, 'n\_estimators': 500}; Desempeño en los datos de test rmse: 3,526.71107mae: 2,228.3541
- d. Mejores parámetros encontrados: {'colsample\_bytree': 0.4, 'gamma': 0, 'max\_depth': 9, 'n\_estimators': 500}; Desempeño en los datos de test rmse: 3,526.711071 mae: 2,228.3541
- e. Mejores parámetros encontrados: {'colsample\_bytree': 0.4, 'gamma': 0, 'learning\_rate': 0.3, 'max\_depth': 9, 'n\_estimators': 500}; Desempeño en los datos de test rmse: 3,526.7110 mae: 2,228.3541
- f. Mejores parámetros encontrados: {'colsample\_bytree': 0.4, 'gamma': 1000000.0, 'max\_depth': 9, 'n\_estimators': 500}; Desempeño en los datos de test rmse: 3,524.8112 mae: 2,229.0188
- g. Mejores parámetros encontrados: {'colsample\_bytree': 0.4, 'gamma': 1000000.0, 'learning\_rate': 0.3, 'max\_depth': 9, 'n\_estimators': 600}; Desempeño en los datos de test rmse: 3,510.3821 mae: 2,208.9933
- h. Mejores parámetros encontrados: {'colsample\_bytree': 0.4, 'gamma': 900000.0, 'learning\_rate': 0.3, 'max\_depth': 9, 'n\_estimators': 600}; Desempeño en los datos de test rmse: 3,508.0440 mae: 2,208.1086
- i. Mejores parámetros encontrados: {'colsample\_bytree': 0.4, 'gamma': 900000.0, 'learning\_rate': 0.3, 'max\_depth': 8, 'n\_estimators': 600}; Desempeño en los datos de test rmse: 3,511.5030 mae: 2,221.0927

4.3 De acuerdo, con estos resultados el mejor desempeño se logró con el modelo de XGBoost con los parámetros del literal h: *'colsample\_bytree': 0.4, 'gamma': 900000.0, 'learning\_rate': 0.3, 'max\_depth': 9, 'n\_estimators': 600*; cuyas métricas de desempeño predictivo obtienen el menor valor de rmse: 3,508.0440 y mae: 2,208.1086. Los resultados son muy similares a los obtenidos con los hiper parámetros de los literales i y g en el punto anterior.

4.4 Teniendo en cuenta que las predicciones del modelo XGBoost utilizando los hiper parámetros g, h, i son muy similares, decidimos obtener las predicciones para la competencia de Kaggle usando cada una de ellas. En una primera corrida de las predicciones obtuvimos un error, debido a que el dataset de pruebas 'dataTest\_carListings.zip' no contenía todas las combinaciones de los predictores 'Make' y 'Model' contenidas en el dataset 'dataTrain\_carListings.zip', siendo imposible obtener predicciones. Para solucionar este inconveniente se utilizó el siguiente código para agregar las columnas faltantes al dataset de pruebas con valor cero en todas las filas, adicionalmente se ordenan las columnas para evitar errores en la coincidencia de los datos.

```

1. # preprocesar los datos de entrenamiento
2. dataTrainingCoded = prepareData(dataTraining.copy())
3. X_Test = prepareData(dataTesting, True)
4.
5. # Separar predictores y variable de resultado
6. XTotal = dataTrainingCoded.drop(columns=['Price'])
7. yTotal = dataTrainingCoded[['Price']]
8.
9. # Encontrar las columnas de XTotal que no están en X_Test
10. columnas_faltantes = set(XTotal.columns) - set(X_Test.columns)
11.
12. # Agregar las columnas faltantes a X_Test con valores igual a cero
13. for columna in columnas_faltantes:
14.     X_Test[columna] = 0

```



```

15.
16. # Ordenar las columnas de XTotal y X_Test en orden alfabetico
17. XTotal=XTotal.sort_index(axis=1)
18. X_Test = X_Test.sort_index(axis=1)

```

- 4.5 Para cada conjunto de hiper parámetros declaramos el modelo de XGBoost, lo entrenamos y obtuvimos las predicciones correspondientes, las cuales exportamos a un archivo CSV y cargamos en Kaggle.

```

1. model1 = xgb.XGBRegressor(objective='reg:squarederror', colsample_bytree = 0.4, gamma = 1000000.0, learning_rate = 0.3,
    max_depth = 9, n_estimators = 600, random_state=0)
2. model1.fit(XTotal, yTotal)
3. y_pred1 = model1.predict(X_Test)
4.
5. model2 = xgb.XGBRegressor(objective='reg:squarederror', colsample_bytree = 0.4, gamma = 900000.0, learning_rate = 0.3,
    max_depth = 9, n_estimators = 600, random_state=0)
6. model2.fit(XTotal, yTotal)
7. y_pred2 = model2.predict(X_Test)
8.
9. model3 = xgb.XGBRegressor(objective='reg:squarederror', colsample_bytree = 0.4, gamma = 900000.0, learning_rate = 0.3,
    max_depth = 8, n_estimators = 600, random_state=0)
10. model3.fit(XTotal, yTotal)
11. y_pred3 = model3.predict(X_Test)

```

- 4.6 El resultado obtenido en Kaggle para cada una de estas combinaciones de hiper parámetros fue el siguiente:

- i. Model g RMSE = 3454.53616
- ii. Model h RMSE = 3450.95997
- iii. Model i RMSE = 3465.96427

- 4.7 Adicionalmente, ensamblamos las predicciones de los modelos g, h, i, promediando sus resultados y los subimos a Kaggle, siendo esta nuestra mejor predicción con esta técnica de calibración, la cual logro en la competencia un RMSE igual a 3437.19643. Este fue el código utilizado para obtener las predicciones promedio de los tres modelos:

```

1. dfTotal = pd.concat([pd.DataFrame(y_pred1, columns=['Price1']), pd.DataFrame(y_pred2, columns=['Price2']),
    pd.DataFrame(y_pred3, columns=['Price3'])], axis=1)
2. dfTotal['Price'] = dfTotal.mean(axis=1)
3. dfTotal[['Price']].to_csv('test_submission5.csv', index_label='ID')

```

- 4.8 La calibración del XGBoost usando BayesSearchCV nos permitió obtener un mejor resultado. Con esta técnica calibramos los hiper parámetros seleccionados, indicando un rango para cada uno de ellos. Usamos dos aproximaciones, la primera de ellas permitiendo seleccionar cualquier valor de 'learning\_rate' y 'colsample\_bytree' en el rango de búsqueda especificado.

```

1. param_grid = {
2.     'learning_rate': (0.05, 1.0, prior='uniform'), # Valores a probar para learning rate
3.     'gamma': [x * 100000 for x in range(16)], # Valores a probar para gamma
4.     'colsample_bytree': (0.1, 1.0, prior='uniform'), # Valores a probar para colsample_bytree
5.     'max_depth': Integer(5, 11), # Valores para probar max_depth
6.     'n_estimators': [x * 100 for x in range(1, 11)]
7. }

```

La segunda limitando los valores de 'learning\_rate' a múltiplos de 0.05 y de 'colsample\_bytree' a múltiplos de 0.1.

```

1. def learning_rate_generator():
2.     return [x * 0.05 for x in range(1, 21)] # Genera valores en pasos de 0.05 entre 0.05 y 1.0
3.
4. def colsample_bytree_generator():
5.     return [x * 0.1 for x in range(1, 11)] # Genera valores en pasos de 0.05 entre 0.05 y 1.0
6.
7. param_grid = {
8.     'learning_rate': Categorical(learning_rate_generator()), # Valores a probar para learning rate

```

```

9. 'gamma': [x * 100000 for x in range(16)], # Valores a probar para gamma
10. 'colsample_bytree': Categorical(colsample_bytree_generator()), # Valores a probar para colsample_bytree
11. 'max_depth': list(range(4,13)), # Valores para probar max_depth
12. 'n_estimators': [x * 100 for x in range(1, 11)]
13. }

```

Corrimos la búsqueda de hiper parámetros con la técnica de BayesSearchCV en varias ocasiones, estos son los mejores resultados que obtuvimos, siendo el modelo de XGBoostRegressor con los parámetros listados en 'c' el mejor modelo predictivo al obtener los menores resultados en las métricas RMSE y MAE.

- a. Mejores parámetros encontrados: ('colsample\_bytree', 0.8186647939080193), ('gamma', 1200000), ('learning\_rate', 0.42239119586880713), ('max\_depth', 5), ('n\_estimators', 800); Desempeño en los datos de test rmse: 3,498.1512 y mae: 2,195.
- b. Mejores parámetros encontrados: ('colsample\_bytree', 1.0), ('gamma', 1100000), ('learning\_rate', 0.3848410248252016), ('max\_depth', 5), ('n\_estimators', 700); Desempeño en los datos de test rmse: 3,517.7968 y mae: 2,215.
- c. Mejores parámetros encontrados: ('colsample\_bytree', 0.9), ('gamma', 700000), ('learning\_rate', 0.45), ('max\_depth', 4), ('n\_estimators', 1000); Desempeño en los datos de test rmse: 3,487.1057 y mae: 2,192.7304.

- 4.9 Con los grupos de parámetros óptimos obtenidos, entrenamos el modelo con los datos de entrenamiento y realizamos la predicción en el conjunto de prueba de la competencia, estos son los resultados obtenidos en Kaggle:
  - i. Model a. RMSE 3446.86920
  - ii. Model b. RMSE 3440.13512
  - iii. Model c. RMSE 3443.31166
- 4.10 Obtuvimos los promedios de las predicciones a y b; y a, b y c y las subimos a la competencia obteniendo estos resultados:
  - iv. Model A y B. RMSE 3420.53043
  - v. Model A, B y C. RMSE 3414.64672
- 4.11 Nuestro mejor resultado se obtuvo obteniendo los hiper parámetros a través del uso de la técnica de BayesSearchCV y promediando las predicciones de los tres mejores modelos que logramos calibrar.

## 5 Publicación del modelo

- 5.1 Para la publicación del modelo creamos un servicio en AWS, y utilizamos una de las mejores combinaciones de hiper parámetros que encontramos durante el proceso de calibración:

```

1. model = xgb.XGBRegressor(objective='reg:squarederror', colsample_bytree = 0.4, gamma = 900000.0, learning_rate = 0.3,
max_depth = 9, n_estimators = 600, random_state=0)

```

- 5.2 Tras realizar el preprocesamiento de los datos como se indico en el literal 2d de este documento, se entreno el modelo y se guardó como 'usedCarPrices.pkl'.

```

1. # Transformar datos y crear dummies en train
2. dataTrainingCoded = prepareData(dataTraining.copy())
3. # Separar predictores y resultado
4. XTotalTrain = dataTrainingCoded.drop(columns=['Price'])
5. yTotalTrain = dataTrainingCoded[['Price']]
6. # Transformar datos y crear dummies en test
7. dataTestCoded = prepareData(dataTraining.copy(), True, XTotalTrain.columns)
8.
9. # Entrenar el modelo
10. model.fit(XTotalTrain, yTotalTrain)
11. # y_pred = model.predict(X_Test)
12.
13. # Exportar modelo a archivo binario .pkl
14. joblib.dump(model, 'usedCarPrices.pkl', compress=3)

```



5.3 El API se creó utilizando la librería Flask y se definieron sus argumentos así:

```
1. # Definición argumentos o parámetros de la API
2. parser = api.parser()
3. parser.add_argument(
4.     'Year',
5.     type=int,
6.     required=True,
7.     help='year of manufacture',
8.     location='args')
9.
10. parser.add_argument(
11.     'Mileage',
12.     type=float,
13.     required=True,
14.     help='vehicle mileage',
15.     location='args')
16.
17. parser.add_argument(
18.     'State',
19.     type=str,
20.     required=True,
21.     help='Abbreviation of US state names',
22.     location='args')
23.
24. parser.add_argument(
25.     'Make',
26.     type=str,
27.     required=True,
28.     help='vehicle manufacturer',
29.     location='args')
30.
31. parser.add_argument(
32.     'Model',
33.     type=str,
34.     required=True,
35.     help='vehicle model',
36.     location='args')
37.
```

5.4 Se creo la función 'usedCarPricesPredict' que se encarga de cargar el modelo, y transformar los datos sobre que recibe para realizar la predicción, y devuelve la predicción de precio con base en la información contenida en los predictores recibidos.

```
1. def usedCarPricesPredict(Year, Mileage, State, Make, Model):
2.     clf = joblib.load('usedCarPrices.pkl')
3.     data_ = pd.DataFrame({'Year': [Year], 'Mileage': [Mileage], 'State': [State], 'Make': [Make], 'Model': [Model]})
4.     X_Test = prepareData(data_, True, predictors)
5.     # Make prediction
6.     p1 = clf.predict(X_Test)
7.     return p1
8.
```

5.5 Para la publicación del modelo se realizaron los siguientes pasos:

- a) Se creó en el servicio EC2 de AWS Cloud la instancia t2.small donde se habilitó el puerto 5000 para consumir la api
- b) Se implementó Docker para incluir dentro de un container todo lo necesario para ejecutar la aplicación. Para esto se desarrolló el respectivo Dockerfile, donde dentro del folder proyecto tenemos los archivos:
  - usedCarPrices.pkl: librería del modelo desarrollado
  - app.py : api desarrollada
  - requirements.txt: contiene las diferentes librerías de Python utilizadas en la api (pandas, scikit-learn, xgboost, Flask, flask\_restx)

```
# Usa la imagen base de Python
FROM python:3.8

# Establece el directorio de trabajo en /api_precio
WORKDIR /api_precio

# Copia el archivo de requisitos (requirements.txt) al contenedor
COPY proyecto/ .

# Instala las dependencias
RUN pip install --no-cache-dir -r requirements.txt

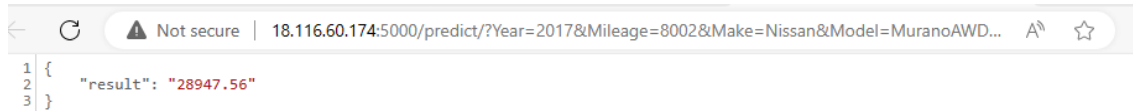
# Exponer el puerto 5000, que es donde se ejecutará la API
EXPOSE 5000

# Comando para ejecutar la API
CMD ["python", "app.py"]
```

5.6 El API está disponible en este [enlace](#). Teniendo en cuenta que debimos usar recursos de pago, no se encuentra habilitada de forma permanente. Para probarla les solicitamos contactar a Jhocel Duvan Suescun Torres, miembro de nuestro equipo, a través del correo [j.suescunt@uniandes.edu.co](mailto:j.suescunt@uniandes.edu.co) o de slack, o de informarnos un rango de tiempo en el que realizaran las pruebas a fin de que habilitemos la API.

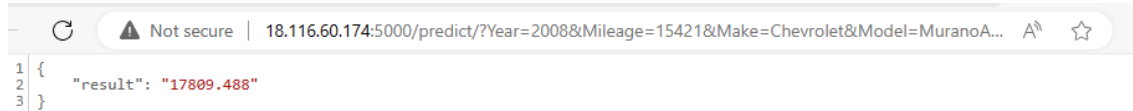
5.7 Para probar la API realizamos predicciones con dos observaciones del dataset de pruebas y obteniendo los siguientes resultados.

- Prueba 1:
  - <http://18.116.60.174:5000/predict/?Year=2017&Mileage=8002&Make=Nissan&Model=MuranoAWD&State=MD>
  - Resultado:



```
1 {
2   "result": "28947.56"
3 }
```

- Prueba 2:
  - <http://18.116.60.174:5000/predict/?Year=2008&Mileage=15421&Make=Chevrolet&Model=SonicSedan&State=MD>
  - Resultado:



```
1 {
2   "result": "17809.488"
3 }
```

5.8 [Aquí](#) incluimos un video como evidencia del funcionamiento del API.

## 6 Conclusiones

- 6.1 XGBoost es un método robusto y computacionalmente eficiente, el cual permite gestionar grandes volúmenes de datos y de predictores de forma eficaz.
- 6.2 La selección adecuada de los hiper parámetros de XGBoost permite gestionar el sesgo y varianza del modelo, permitiendo obtener un modelo con un gran poder predictivo.
- 6.3 La forma en que el valor de un hiper parámetro incide en el desempeño predictivo de XGBoost depende no solo del valor que toma este, sino de los valores de los otros hiper parámetros.
- 6.4 En general, los hiper parámetros tienen un efecto en la capacidad predictiva del modelo que describe una curva cóncava, donde valores bajos se asocian a un valor de MSE que desciende

progresivamente hasta alcanzar un valor óptimo a partir del cual no puede mejorarse el resultado del MSE, y al contrario este puede deteriorarse.

- 6.5 En ejercicio anterior usamos BayesSearchCV de forma incipiente. En el desarrollo de este taller comprendimos mejor su funcionamiento identificándola como una técnica muy eficiente para identificar los mejores hiper parámetros con los que puede calibrarse un modelo en un rango de espacio determinado. Al ser una técnica probabilística no requiere que se indiquen parámetros específicos sino un rango de búsqueda, y al usar las observaciones realizadas para refinar modelo probabilístico logra identificar de forma más eficaz la combinación de hiper parámetros que mejora el poder predictivo del modelo.
- 6.6 El ensamble de modelos permitió mejorar el poder predictivo de modelo final, obteniendo el mejor resultado al promediar las predicciones de los mejores modelos encontrados. De esta manera la mejor predicción con un RMSE=3414.64672 fue la obtenida al promediar los resultados de las predicciones realizadas con XGBoost y los parámetros indicados en el numeral 4.8 literales a, b y c.