

**INFORME EJECUTIVO 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

# Exploración preliminar de los datos

El set de datos contiene seis columnas, cinco de ellas corresponden a predictores.

# 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’.

## 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. Lo anterior utilizando el siguiente código:

1. def prepareData (df, isTesting=False):

2. # Concatenar las variables Make y Model

3. df['Make\_Model']=df['Make']+'\_'+df['Model']

4.

5. # Crear la lista de variables según el conjunto de datos que se está transformando

6. if isTesting == False:

7. varList1 = ['Price', 'Year', 'Mileage', 'State', 'Make\_Model']

8. varList2 = ['Price', 'Year', 'Mileage']

9. else:

10. varList1 = ['Year', 'Mileage', 'State', 'Make\_Model']

11. varList2 = ['Year', 'Mileage']

12.

13. # Filtrar el dataframe por las columnas de interes

14. df = df[varList1]

15.

16. # Crear la instancia de OneHotEncoder y las columnas dummy para State y Make\_Model

17. encoder = OneHotEncoder(drop='first', sparse=False) # Usamos drop=’first’ para eliminar la primera categoría en cada característica

18. colsToEncoded=['State', 'Make\_Model']

19. dfCoded = pd.DataFrame(encoder.fit\_transform(df[colsToEncoded]))

20. # nombrar las columnas dummy

21. dfCoded.columns = encoder.get\_feature\_names\_out(colsToEncoded)

22. # agregar las columnas 'Year', 'Mileage' o 'Price', 'Year', 'Mileage' según corresponda al conjunto de datos

23. dfCoded[varList2]=df[varList2]

24. return dfCoded

## 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)

## 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.

# Calibración del modelo

## Por su eficiencia computacional y poder predictivo decidimos usar como modelo base XGBoostRegressor.

## Consideramos pertinente calibrar los siguientes hiper parámetros para calibrar: '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.

## 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.

## 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 amplió, lo que facilita la exploración y selección óptima de resultados.

## 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.

* 1. 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 tienda a aumentar, de igual forma ocurre con valores de 0.3, y 0.2.
  2. 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.
  3. 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.
  4. 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.
  5. 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.
  6. 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.

# Entrenamiento del modelo

## 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}")

## 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).

* 1. Mejores parámetros encontrados: {'n\_estimators': 600}; Desempeño en los datos de test rmse: 3,530.7848 mae: 2,237.9993
  2. 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
  3. 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
  4. 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
  5. 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
  6. 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
  7. 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
  8. 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
  9. 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

## 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.

## 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 varible 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)

## 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)

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

* 1. Model g RMSE = 3454.53616
  2. Model h RMSE = 3450.95997
  3. Model i RMSE = 3465.96427

## Adicionalmente, ensamblamos las predicciones de los modelos g, h, i, promediando sus resultados y los subimos a Kaggle, siendo esta nuestra mejor predicción, 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')

# Publicación del modelo

• Se disponibiliza el modelo en una API alojada en un servicio en la nube. (20 puntos)

• Se hacen las predicciones sobre el valor del automóvil en al menos dos observaciones del set de validación. (10 puntos)

# Conclusiones

## XGBoost es un método robusto y computacionalmente eficiente, el cual permite gestionar grandes volúmenes de datos y de predictores de forma eficaz.

## 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.

## 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.

## 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.

Requisitos y rúbrica de calificación

Preprocesamiento de datos (10 puntos)

* Los datos de entrenamiento se dividen en datos de entrenamiento y validación. Si decidieron preprocesar los datos (estandarizar, normalizar, imputar valores, etc), estos son correctamente preprocesados al ajustar sobre los datos de entrenamiento (.fit\_transform()) y al transformar los datos del set de validación (.transform()). (10 puntos)

Calibración del modelo (15 puntos)

* Se calibran los parámetros que se consideren pertinentes del modelo de clasificación seleccionado. (5 puntos)
* Se justifica el método seleccionado de calibración. (5 puntos)
* Se analizan los valores calibrados de cada parámetro y se explica cómo afectan el modelo. (5 puntos)

Entrenamiento del modelo (15 puntos)

* Se entrena el modelo de clasificación escogido con los datos del set de entrenamiento preprocesados y los parámetros óptimos. (5 puntos)
* Se presenta el desempeño del modelo en los datos de validación con al menos una métrica de desempeño. (5 puntos)
* Se justifica la selección del modelo correctamente. (5 puntos)

Disponibilización del modelo (30 puntos)

* Se disponibiliza el modelo en una API alojada en un servicio en la nube. (20 puntos)
* Se hacen las predicciones sobre el valor del automóvil en al menos dos observaciones del set de validación. (10 puntos)

Conclusiones (10 puntos)

* Se presentan conclusiones claras y concisas sobre el desarrollo y los resultados del proyecto. (10 puntos)