Car Quality Predictor by Lakitus

Technical Documentation

App Link: https://lakitus-quality-predictor.streamlit.app/
GitHub repository: https://github.com/ZocoMacc/car-quirks-ml_InnovaLab25.git

Resumen

Este proyecto fue presentado en *InnovaLab 2025*, una hackathon de *Machine Learning* cuyo reto consistía en predecir la calidad de un vehículo en función de ciertas características. Los organizadores proporcionaron un archivo CSV con los datos necesarios, el cual utilizamos para construir y entrenar nuestro modelo. Utilizamos python y herramientas de machine learning enfocadas a la generación y entrenamiento de modelos. El resultado fue un modelo robusto que es capaz de predecir la calidad de un vehículo en base a “Alta”, “Media”, o “Baja”.

Introducción

En este hackathon, nuestro objetivo fue construir un modelo de aprendizaje automático capaz de predecir la categoría de calidad de un auto —“Baja”, “Media” o “Alta”— según sus especificaciones. Recibimos un conjunto de datos de 10,000 vehículos, cada uno descrito por características como año de fabricación, kilómetros recorridos, tipo de combustible, potencia del motor, clasificación de seguridad y eficiencia de combustible. La variable objetivo, calidad_auto, presentaba un desequilibrio (aproximadamente 84% “Media”, 10.5% “Alta” y 5.5% “Baja”), lo que requirió un manejo cuidadoso durante el entrenamiento y la evaluación. Nuestra motivación es doble: primero, demostrar cómo el análisis predictivo puede ayudar a compradores y concesionarios a evaluar rápidamente la calidad del vehículo; y segundo, ofrecer una aplicación web interactiva y explicable donde los usuarios puedan ingresar las especificaciones del auto, obtener una predicción de calidad inmediata y explorar el razonamiento del modelo mediante valores SHAP y controles deslizantes hipotéticos.

Carga de datos y exploración

Comenzamos inspeccionando el archivo CSV proporcionado, que contenía 10 000 filas y los siguientes campos: name, year, selling_price, km_driven, fuel, combustible_estimado_l, seller_type, transmission, owner, tipo_carroceria, potencia_motor_hp, nivel_seguridad, calidad_auto, eficiencia_km_l y score_calidad. Dado que no había valores faltantes, nos centramos inicialmente en comprender la distribución y las relaciones entre estas variables. En particular:

- **Distribución objetivo**: "Media" comprendía aproximadamente el 84 % de las entradas, "Alta" aproximadamente el 10,5 % y "Baja" aproximadamente el 5,5 %, lo que indica un desequilibrio de clase significativo.
- **Resúmenes numéricos**: Calculamos las medias, medianas y rangos para year, selling_price, km_driven, potencia_motor_hp, nivel_seguridad y eficiencia_km_l para identificar valores atípicos (por ejemplo, kilometraje extremadamente alto o calificaciones de seguridad muy bajas).
- **Correlaciones**: Una rápida correlación de Pearson mostró que la score_calidad tenía una correlación moderada con calidad_auto (≈0,74), pero optamos por descartarla posteriormente para evitar la fuga de

etiquetas. Otras correlaciones moderadas incluyeron eficiencia_km_l (≈0,53) y nivel_seguridad (≈0,50) con el objetivo codificado.
- **Desgloses categóricos**: Verificamos la distribución de las características codificadas con números enteros (p. ej., tipos de combustible 0-4, tipos de vendedor 0-2, tipos de carrocería 1-5) para asegurar que ninguna categoría predominara por completo.

Este paso exploratorio confirmó la limpieza de los datos, destacó el desequilibrio de clases y nos orientó a la decisión de diseñar características adicionales (como extraer una marca del nombre) y descartar la score_calidad para evitar la fuga de etiquetas.

Preprocesamiento de datos

Antes de entrenar cualquier modelo, limpiamos y transformamos los campos sin procesar en características que el pipeline pudiera consumir. Los pasos clave incluyen:

- Eliminación de columnas con fugas o irrelevantes: Aunque la variable "score_calidad" mostró una fuerte correlación (≈0,74) con "calidad_auto", se derivó parcialmente del propio objetivo, por lo que la eliminamos para evitar fugas. De igual forma, descartamos el campo de nombre sin procesar después de extraer únicamente la marca.
- Extracción de marca: Compilamos una lista de 29 marcas conocidas (p. ej., "maruti", "hyundai", "tata", etc.) y creamos una función auxiliar que convertía cada cadena de nombre en minúsculas y devolvía el prefijo de marca correspondiente (u "other" si no había ninguna coincidencia). Esta nueva columna de marca se convirtió a category.
- Codificación categórica y pipelines: Las características con codificación entera (fuel, seller_type, transmisión, owner, tipo_carroceria y brand) se convirtieron al tipo de datos category de Pandas. Construimos un ColumnTransformer con dos pipelines paralelos:
    - Un pipeline numérico que aplica StandardScaler a características continuas (año, precio de venta, km recorridos, combustible estimado, potencia motor hp, nivel seguridad, eficiencia km l).
    - Un pipeline categórico que utiliza OneHotEncoder(handle_unknown="ignore") en las seis columnas categóricas. Estas se combinaron para que, al momento del ajuste, todas las columnas numéricas y categóricas se preprocesaran de una sola vez.
- División de entrenamiento/prueba: Finalmente, realizamos una división estratificada 80/20 en calidad_auto para preservar las proporciones de clase, asegurando que nuestro conjunto de prueba retenido mantuviera la representatividad (≈84% "Media", ≈10.5% "Alta", ≈5.5% "Baja").

Selección de Modelo

Evaluamos varios clasificadores para encontrar el mejor equilibrio entre precisión e interpretabilidad:

- **Baseline: DummyClassifier**
  Utilizando la estrategia="most_frequent", el modelo predijo "Media" para cada muestra, con una precisión de aproximadamente el 84 % (la clase mayoritaria), pero una F1 macro cercana a 0,30. Esta línea base confirmó que cualquier modelo predictivo real debe superar la estimación por mayoría simple.

- **Regresión logística**

Creamos una secuencia de comandos que combina nuestro ColumnTransformer con LogisticRegression(multi_class="multinomial", solver="lbfgs", max_iter=5000). En una validación cruzada de 5 pliegues (scoring="f1_macro"), las puntuaciones de los pliegues oscilaron entre aproximadamente 0,91 y 0,96, con un promedio de aproximadamente 0,93. En el conjunto de prueba del 20% retenido, la precisión alcanzó 0,97 y la macro-F1 ≈ 0,93. Aunque lineal, este modelo captó suficiente señal como para constituir una línea base sólida.

- **Bosque aleatorio**
  Al reemplazar el clasificador con RandomForestClassifier(n_estimators=300, class_weight="balanced", random_state=42) se obtuvo un CV de 5 veces para la macro-F1 ≈ 0,86 y una precisión de la prueba ≈ 0,955. Las altas puntuaciones de entrenamiento (cercanas a 1,0) indicaron sobreajuste. Si bien RF gestionó interacciones no lineales, su generalización fue ligeramente inferior a la de la regresión logística y los métodos potenciados por gradiente.

- **LightGBM**
  A continuación, probamos LGBMClassifier(n_estimators=200, learning_rate=0,1, max_depth=6, subsample=0,8, colsample_bytree=0,8, random_state=42). La validación cruzada arrojó un macro-F1 de ≈ 0,93 y la precisión de la prueba fue de ≈ 0,975. La velocidad de LightGBM y su manejo nativo de grandes conjuntos de datos lo hicieron muy competitivo tanto en tiempo de entrenamiento como en rendimiento predictivo.

- **XGBoost (Modelo final)**
  El pipeline final utilizó XGBClassifier(objective="multi:softprob", num_class=3, learning_rate=0.1, n_estimators=200, max_depth=6, subsample=0.8, colsample_bytree=0.8, random_state=42, n_jobs=-1). En un CV de 5 pliegues, el macro-F1 promedió ≈ 0,936. En el conjunto de prueba retenido, la precisión alcanzó ≈ 0,977 con un macro-F1 de ≈ 0,94. Las matrices de confusión mostraron una alta recuperación para "Media" y una alta precisión/recuperación para "Alta" y "Baja", lo que convierte a XGBoost en nuestra mejor opción para la implementación final.

- **Ajuste de hiperparámetros**
  Ajustamos los parámetros clave de XGBoost mediante RandomizedSearchCV en los rangos de max_depth, learning_rate, n_estimators, subsample y colsample_bytree, optimizando para macro-F1. La mejor configuración mejoró ligeramente la precisión de la prueba a ≈ 0,98 y la macro-F1 a ≈ 0,95, lo que demuestra que un ajuste cuidadoso puede generar mejoras incrementales.

En general, XGBoost, tras el ajuste, ofreció la mejor relación entre velocidad, precisión y robustez, por lo que fue seleccionado como el modelo final para nuestra aplicación Streamlit.

Evaluación e interpretación del modelo

Utilizamos dos enfoques para comprender cómo nuestro modelo XGBoost toma decisiones:

- **Feature Importance (XGBoost)**
  Se extrajeron las 10 características principales mediante model.feature_importances_, incluyendo variables de alto impacto como

ciertas columnas one-hot de tipo_carroceria, año, nivel_seguridad y eficiencia_km_l.
Estas importancias se alinearon con el conocimiento del dominio: los autos más nuevos, las calificaciones de seguridad más altas y los motores eficientes tienden a tener una calificación "Alta".

- **Valores SHAP**
  - Se aprovechó shap.TreeExplainer del clasificador XGBoost entrenado para calcular las atribuciones por instancia.
  - Para cada entrada de auto, se trazaron las 10 características SHAP principales como un gráfico de barras horizontales (modo oscuro), con las barras positivas acercándose a la clase predicha y las negativas alejándose.
  - Esta explicación a nivel de instancia ayuda a los usuarios a ver exactamente qué atributos (por ejemplo, alto nivel_seguridad o bajo km_driven) llevaron al modelo hacia “Media” o “Alta”, mejorando la transparencia y la confianza.

Implementación web con Streamlit

Creamos una aplicación Streamlit de un solo archivo (app.py) que integra nuestra pipeline de XGBoost en una interfaz interactiva. Componentes clave:

- Estructura de carpetas y dependencias
  - app.py se encuentra en el root del repositorio.
  - Una subcarpeta models/ contiene xgb_final_pipeline.pkl y label_encoder.pkl.
  - train_data.csv (para la búsqueda del vecino más cercano) también se encuentra en la raíz de bajo de data/.
  - requirements.txt enumera todas las bibliotecas necesarias: streamlit, pandas, scikit-learn, xgboost, shap, matplotlib, joblib, etc.

- Carga y almacenamiento en caché del modelo
  - Usamos @st.cache_data para cargar tanto la pipeline serializada como LabelEncoder una vez al inicio, lo que evita la repetición de operaciones de E/S en disco.
  - Los paths se construyen en relación con app.py (p. ej., os.path.join(os.path.dirname(__file__), "models", "xgb_final_pipeline.pkl")) para garantizar la coherencia durante la implementación.

- Sidebar input
  - La barra lateral recopila las especificaciones estáticas del vehículo: año, precio de venta, combustible, combustible estimado, tipo de vendedor, transmisión, propietario y marca (extraída del nombre).

- Controles deslizantes y predicción de hipótesis
  - Debajo del gráfico SHAP, seis controles deslizantes permiten ajustar el año, los kilómetros recorridos, el tipo de carroceria, la potencia del motor, el nivel de seguridad y la eficiencia de km.
  - Con cada cambio en el control deslizante, Streamlit vuelve a ejecutar el script, recalculando una "predicción de hipótesis" (etiqueta de color + barra de progreso + probabilidades de clase) inmediatamente encima de los controles deslizantes.

- La fila de entrada actual se reconstruye a partir de los valores de la barra lateral y el control deslizante, se asigna una categoría donde sea necesario y se pasa a través de model.predict() y model.predict_proba().

En resumen, la aplicación Streamlit se ejecuta localmente o en Community Cloud como un panel totalmente interactivo que predice la calidad del automóvil, explica la predicción con SHAP y permite a los usuarios explorar escenarios hipotéticos.