Informe del Modelo XGboost Machine Learning

Descripción del Modelo

Se utilizó un modelo de **XGBoost** (eXtreme Gradient Boosting) para realizar una clasificación binaria basada en el conjunto de datos SECOM. XGBoost es un algoritmo eficiente y escalable que combina árboles de decisión con gradiente descendente, ideal para detectar patrones en conjuntos de datos con gran cantidad de características.

Este script implementa un modelo de clasificación binaria para analizar datos industriales dentro de un proceso (SECOM). Su objetivo es predecir si un proceso pasará ("Pass") o fallará ("Fail") basado en las características del proceso.

Conjunto de Datos

El conjunto de datos utilizado, "SECOM", proviene del conjunto de datos descargados de: https://www.kaggle.com/datasets/paresh2047/uci-semcom Estos datos fueron procesados y limpiados (ver script Modelo_Random_Forest) Estos datos contienen información de sensores para clasificar muestras en dos categorías: "Pass" (aprobado) y "Fail" (fallo). Después de la limpieza, se trabajó con datos balanceados mediante sub-muestreo manual para evitar sesgos hacia la clase mayoritaria.

- **Número total de características:** 590
- **Conjunto de entrenamiento:** 80% de los datos balanceados.
- **Conjunto de prueba:** 20% de los datos originales.

Parámetros del Modelo

El modelo se configuró con los siguientes Híper-parámetros:

- **`objective`**: `binary:logistic` (para clasificación binaria).
- **`eval_metric`**: `auc` (área bajo la curva ROC).
- **`eta`**: 0.1 (tasa de aprendizaje).
- **`max_depth`**: 4 (profundidad máxima de los árboles).
- **`min_child_weight`**: 10 (peso mínimo para dividir nodos).

Se entrenó el modelo durante un máximo de 100 iteraciones con detención anticipada en caso de que el AUC del conjunto de prueba no mejorara en 10 rondas consecutivas.

Informe paso a paso

1. Librerías utilizadas

```
library(xgboost)
library(data.table)
library(caret)
library(pROC)
library(ggplot2)
```

Propósito:
Importar librerías necesarias para manejar datos (`data.table`), dividir conjuntos (`caret`), graficar (`ggplot2`), calcular métricas de rendimiento (`pROC`) y entrenar el modelo (`xgboost`).

Por qué:
Cada librería aporta funciones clave:

- **`xgboost`**: para entrenar y evaluar el modelo.
- **`data.table`**: manipulación rápida de datos.
- **`caret`**: particionar datos y calcular métricas.
- **`pROC`**: curva ROC y AUC.
- **`ggplot2`**: gráficos personalizables.

---

2. Carga de datos

```
file_path <- "data/cleaned_secom.csv"
data <- fread(file_path)
```

Propósito:
Leer datos desde un archivo CSV y cargarlos en un objeto `data.table`.

---

3. Separación de variables

```
X <- data[, !"Pass.Fail"]
y <- ifelse(data$Pass.Fail == "Pass", 1, 0)
```

Propósito:
Separar las variables predictoras (`X`) de la variable objetivo (`y`).

Por qué:
La variable `Pass.Fail` es categórica y debe transformarse a formato binario (1 para "Pass" y 0 para "Fail") para el modelo.

---

4. Conversión de datos

```
X <- as.data.frame(X)
X <- X[, sapply(X, is.numeric)]
```

Propósito:
Convertir `X` a un formato de `data.frame` y mantener únicamente columnas numéricas.

Por qué:
XGBoost no admite datos no numéricos, por lo que eliminar columnas no válidas previene errores.

---

5. División del conjunto de datos

```
set.seed(42)
train_index <- createDataPartition(y, p = 0.8, list = FALSE)
X_train <- X[train_index, ]
X_test <- X[-train_index, ]
y_train <- y[train_index]
y_test <- y[-train_index]
```

Propósito:
Dividir el conjunto en datos de entrenamiento (80%) y prueba (20%) para validar el modelo.

Por qué:
Es necesario un conjunto separado para evaluar la capacidad del modelo de generalizar.

Ajustes posibles:

- Cambiar el tamaño de la partición (`p`) para equilibrar entrenamiento y validación.
- Fijar `set.seed` para resultados reproducibles.

---

6. Balanceo de clases

```
minority_class <- data.frame(X_train, Class = y_train) %>% filter(Class == 1)
majority_class <- data.frame(X_train, Class = y_train) %>% filter(Class == 0)
%>% sample_n(nrow(minority_class))
balanced_data <- rbind(minority_class, majority_class)
```

Propósito:
Crear un conjunto de entrenamiento balanceado entre clases minoritarias y mayoritarias.

Por qué:
Los datos desbalanceados pueden sesgar el modelo hacia la clase mayoritaria, reduciendo el rendimiento para la clase minoritaria.

Ajustes posibles:

- Experimentar con métodos como sobremuestreo (duplicar la clase minoritaria) o técnicas avanzadas (SMOTE).

---

7. Creación de matrices DMatrix

```
dtrain <- xgb.DMatrix(data = as.matrix(X_train), label = y_train)
dtest <- xgb.DMatrix(data = as.matrix(X_test), label = y_test)
```

Propósito:
Convertir los conjuntos de datos a `DMatrix`, formato optimizado para XGBoost.

Por qué:
`DMatrix` mejora la eficiencia en memoria y velocidad para grandes conjuntos.

---

8. Configuración de parámetros

```
params <- list(
  objective = "binary:logistic",
  eval_metric = "auc",
  eta = 0.1,
  max_depth = 4,
  min_child_weight = 10
)
```

Propósito:
Definir los hiperparámetros del modelo.

Por qué:

- **`objective`**: especifica un modelo de regresión logística para clasificación binaria.
- **`eval_metric`**: el AUC mide el desempeño.
- **`eta`**: controla la velocidad de aprendizaje.
- **`max_depth`**: limita la profundidad de los árboles, reduciendo sobreajuste.
- **`min_child_weight`**: evita la división de nodos con datos insuficientes.

---

9. Entrenamiento del modelo

```
xgb_model <- xgb.train(
  params = params,
  data = dtrain,
  nrounds = 100,
  watchlist = list(train = dtrain, test = dtest),
  early_stopping_rounds = 10,
  print_every_n = 1
)
```

Propósito:
Entrenar el modelo mientras se monitorea el rendimiento en entrenamiento y validación.

Por qué:
El seguimiento iterativo ayuda a prevenir sobreajuste (`early_stopping_rounds`).

Ajustes posibles:

- Incrementar `nrounds` para más iteraciones si los datos son complejos.

---

10. Predicciones y evaluación

Predicciones

```
y_pred <- predict(xgb_model, dtest)
y_pred_class <- ifelse(y_pred > 0.5, 1, 0)
```

Métricas

```
conf_matrix <- confusionMatrix(
  factor(y_pred_class, levels = c(0, 1)),
  factor(y_test, levels = c(0, 1))
)
```

Propósito:
Calcular sensibilidad, especificidad, precisión y matriz de confusión.

Por qué:
Estas métricas proporcionan una visión integral del rendimiento del modelo.

---

11. Gráficos

Curva ROC

```
roc_curve <- roc(y_test, y_pred)
ggplot(roc_data, aes(x = FPR, y = TPR)) +
  geom_line(color = "blue") +
```

```
  geom_abline(linetype = "dashed", color = "red") +
  ggtitle(sprintf("Curva ROC (AUC = %.3f)", auc_value)) +
  xlab("FPR") + ylab("TPR") + theme_minimal()
```

Importancia de características

```
importance <- xgb.importance(feature_names = colnames(X_train), model =
xgb_model)
ggplot(importance[1:10, ], aes(x = reorder(Feature, Gain), y = Gain)) +
  geom_bar(stat = "identity") +
  coord_flip() +
  labs(title = "Top 10 características más importantes") +
  theme_minimal()
```

Propósito:
Visualizar el AUC y las características más relevantes para el modelo.

Resultado del Modelo

El entrenamiento del modelo mostró un rendimiento consistente con una mejora del AUC durante las primeras iteraciones, alcanzando los siguientes valores finales:

- **AUC del conjunto de prueba**: 0.771
- Mejor iteración alcanzada: **11**.

Métricas Obtenidas

- **AUC**: 0.771 (muestra un buen equilibrio entre sensibilidad y especificidad).
- **Sensibilidad (Recall)**: 0.625 (capacidad para identificar correctamente los casos positivos).
- **Especificidad**: 0.765 (capacidad para identificar correctamente los casos negativos).
- **Precisión**: 0.979 (proporción de verdaderos positivos entre todas las predicciones positivas).

Ventajas

1. **Alta precisión (97.88%)**:
   El modelo predice correctamente la mayoría de las muestras positivas y negativas, lo que indica una capacidad de clasificación sólida para datos predominantemente negativos.
2. **Área bajo la curva (AUC) razonable (0.771)**:
   Un AUC superior a 0.75 demuestra un equilibrio adecuado entre sensibilidad y especificidad.
3. **Capacidad de manejo de datos con alta dimensionalidad**:
   El modelo trabajó eficientemente con el conjunto de datos que contenía una gran cantidad de características (590), aprovechando la capacidad de XGBoost para manejar datos de alta dimensión.
4. **Capacidad para manejar datos desequilibrados**: La estrategia de sub-muestreo permitió reducir el sesgo hacia la clase mayoritaria.

5. **Detención anticipada**:
   La configuración de `early_stopping_rounds` evitó un entrenamiento excesivo, mejorando la eficiencia y reduciendo el riesgo de sobreajuste.
6. **Interpretabilidad del modelo**:
   La generación de gráficos de importancia de características proporciona información valiosa sobre las variables más relevantes para las predicciones, lo que puede guiar decisiones posteriores.

Desventajas

1. **Baja sensibilidad (62.5%)**:
   Aunque el modelo detecta la mayoría de los casos negativos, tiene dificultades para identificar correctamente los casos positivos, lo que podría ser crítico en aplicaciones sensibles como detección de fallos o enfermedades.
2. **Dependencia del submuestreo manual**:
   La estrategia de balanceo utilizada podría haber reducido la cantidad de datos disponibles, limitando la capacidad del modelo para aprender patrones importantes de la clase minoritaria.
3. **Limitada especificidad en contextos estrictos**:
   Si bien la especificidad (76.47%) es razonable, podría no ser suficiente en aplicaciones donde es fundamental minimizar falsos positivos.
4. **Posible necesidad de validación cruzada**:
   Aunque el modelo se entrenó y evaluó con una partición única de los datos, la falta de validación cruzada podría limitar la confianza en su generalización.

Recomendaciones

1. **Ajustar los hiperparámetros**:
   Explorar combinaciones de `max_depth`, `eta` (tasa de aprendizaje) y `min_child_weight` podría mejorar el equilibrio entre sensibilidad y especificidad.
2. **Probar técnicas avanzadas de balanceo**:
   Métodos como SMOTE o ADASYN pueden generar datos sintéticos de la clase minoritaria, aumentando la diversidad del conjunto de entrenamiento y mejorando la sensibilidad.
3. **Implementar validación cruzada**:
   Realizar validación cruzada k-fold para evaluar mejor el desempeño del modelo y asegurar que sea robusto frente a diferentes particiones de los datos.
4. **Explorar arquitecturas complementarias**:
   Incorporar modelos alternativos como Random Forest o Redes Neuronales para comparar su desempeño frente a XGBoost y evaluar la combinación de modelos (ensemble).
5. **Analizar las características más importantes**:
   Revisar las 10 principales características destacadas por el modelo para identificar posibles mejoras en la recolección de datos o estrategias de preprocesamiento.

6. **Optimizar métricas según la aplicación**:
   Si la sensibilidad es crítica (por ejemplo, en aplicaciones médicas o de calidad), ajustar el umbral de clasificación (por defecto 0.5) para priorizar la detección de casos positivos.
7. **Expandir el conjunto de datos**:
   Obtener más datos, especialmente de la clase minoritaria, ayudaría a mejorar la representatividad y el desempeño general del modelo.

Conclusión

El script sigue un flujo lógico para entrenar y evaluar un modelo XGBoost, incluyendo balanceo de datos, optimización de hiperparámetros y visualización de resultados. Los gráficos ayudan a comunicar el rendimiento del modelo y las variables clave que pueden determinar puntos críticos que afectan la producción de Chip y generan colas de retraso o perdida en la producción. La atención de estos puntos revela importancia, según la historia que cuenta la data.

El modelo de XGBoost mostró un rendimiento sólido con un AUC de 0.771 y una precisión alta. Sin embargo, la baja sensibilidad sugiere que podrían perderse algunos casos positivos. Ajustes adicionales en la estrategia de balanceo y optimización de híper-parámetros podrían llevar a un mejor desempeño general. Este modelo es un buen punto de partida para análisis adicionales y ajustes iterativos en problemas similares.