Selección sencilla de un modelo de clasificación

Entregable Individual

Nataly Rocha

Escuela de Ingeniería Informática, Universidad de Valladolid

Técnicas Escalables de Análisis de Datos en entornos Big Data: Clasificadores

2025 - 2026

Tabla de Contenidos

1. Introducción

El presente trabajo constituye el entregable individual que tiene como objetivo seleccionar el mejor modelo de clasificación basado en árboles de decisión, mediante la exploración de los parámetros *maxDepth* y *maxBins*. Se utiliza el conjunto de datos Census Income (KDD), proveniente de las encuestas de la Oficina del Censo de Estados Unidos (1994–1995), para predecir si el ingreso anual de una persona supera los 50.000 dólares.

El proceso implementado incluye: (1) la división del conjunto de entrenamiento original en subconjuntos de entrenamiento y validación, (2) la evaluación de múltiples combinaciones de parámetros mediante métricas apropiadas para datos desbalanceados, (3) la selección del mejor modelo con base en el error de validación, y (4) la construcción y evaluación del modelo final sobre el conjunto de prueba.

2. Resumen Ejecutivo

2.1. Resumen ejecutivo del conjunto de datos

El conjunto de datos Census Income (KDD) contiene información censal ponderada obtenida de las Encuestas de Población Actual (Current Population Survey) de los años 1994 y 1995, realizadas por la Oficina de Censo de los Estados Unidos. El dataset está compuesto por 299.285 registros y 41 atributos tanto categóricos como numéricos, que incluyen variables como la edad, el sexo, nivel educativo, ocupación, estado civil, tamaño del empleador, número de horas trabajadas por semana y nacionalidad, entre otras.

2.2. Origen de los datos

UCI Machine Learning Repository – Census Income (KDD) Data Set https://archive.ics.uci.edu/dataset/117/census+income+kdd

2.3. Propósito y uso de los datos.

El objetivo principal de este conjunto de datos es predecir si el ingreso anual de una persona supera los 50.000 dólares, a partir del análisis de variables demográficas, educativas, familiares y laborales. El problema se formula como una clasificación binaria, donde la variable objetivo refleja el nivel de ingresos de cada individuo.

2.4. Tamaño y estructura del conjunto de datos

El dataset contiene 299.285 registros divididos en entrenamiento (199.523 instancias) y prueba (99.762 instancias). Incluye 40 atributos relevantes para clasificación (6 numéricos y 34 categóricos), excluyendo el atributo "instance weight" que no se debe considerar para clasificadores. Las variables describen características demográficas, educativas y laborales como edad, nivel educativo, ocupación, estado civil, relación familiar, nacionalidad y horas trabajadas por semana.

2.5. Descripción de la Clase.

La clase, denominada "PTOTVAL" (total person income), representa el rango de ingresos totales de cada individuo. Es una variable binaria con las siguientes categorías: "-50000": ingresos menores o iguales a 50.000 USD (93,8 % de los casos) "+50000": ingresos superiores a 50.000 USD (6,2 % de los casos) Esta distribución altamente desbalanceada plantea un desafío adicional para los modelos de clasificación, que deberán ser evaluados teniendo en cuenta este desequilibrio de clases [Quy et al., 2022].

2.6. Estrategia de selección del mejor modelo

Para este entregable se emplearon exclusivamente árboles de decisión, evaluando diversas combinaciones de los parámetros *maxDepth* y *maxBins* con el objetivo de identificar la configuración que minimice la tasa de error en el conjunto de validación.

Dado el desbalance existente entre clases (93,8 % vs 6,2 %), aunque el criterio principal de selección es la tasa de error según los requisitos del entregable, también se calcularon y analizaron métricas complementarias como *AUC-PR*, matriz de confusión y *recall* por clase. Estas métricas adicionales permiten realizar un análisis crítico más profundo del rendimiento del modelo, especialmente en su capacidad para identificar la clase minoritaria, información valiosa para trabajos futuros.

3. Proceso de exploración

3.1. Creación de subconjuntos de entrenamiento y validación

Para evitar el sesgo del error de resustitución y cumplir con la restricción de no utilizar el conjunto de prueba para la selección de parámetros, se dividió el conjunto de entrenamiento original en dos subconjuntos: *subTrain* (75 %) para entrenar los modelos y *validation* (25 %) para estimar la tasa de error y seleccionar el mejor modelo.

```
/**
 * Dividir el conjunto de entrenamiento con particion estratificada:
 */
def createTrainingSubsets(originalDF: DataFrame, percentage: Double = 0.75,
seed: Long): Array[Dataset[Row]] = {
  val class0 = originalDF.filter(col("label") === 0.0)
  val class1 = originalDF.filter(col("label") === 1.0)

  // Creamos una particin estratificada
  // Separamos cada clase en entrenamiento y validacin con la misma proporcin
  val Array(train0, val0) = class0.randomSplit(Array(percentage, 1 - percentage
), seed)
  val Array(train1, val1) = class1.randomSplit(Array(percentage, 1 - percentage
), seed)

  // Creamos los conjuntos de entrenamiento y validacin uniendo las clases
  val subTrain = train0.union(train1)
  val validation = val0.union(val1)

  Array(subTrain, validation)
}

```

Se utilizó una semilla aleatoria (*seed*) para garantizar la reproducibilidad de la partición. Adicionalmente, se verificó que ambos subconjuntos mantuvieran una distribución de clases similar a la del conjunto original, evitando así sesgos que pudieran afectar negativamente el rendimiento del modelo (ver figura 1).

```
 *** Distribución de clases en sub-train:
+-----+------+----------+
|label| count|porcentaje|
+-----+------+----------+
|  0.0|  9249|      7.45|
|  1.0|114894|     92.55|
+-----+------+----------+


 *** Distribución de clases en validation:
+-----+-----+----------+
|label|count|porcentaje|
+-----+-----+----------+
|  0.0| 3110|      7.48|
|  1.0|38495|     92.52|
+-----+-----+----------+
```

Figura 1: Distribución de clases en los subconjuntos de entrenamiento y validación

3.2. Selección y justificación de combinaciones de parámetros

Se exploraron 12 combinaciones de los parámetros *maxDepth* y *maxBins*, cuya selección se fundamenta en los siguientes criterios:

maxDepth (profundidad máxima del árbol): Este parámetro controla la complejidad del modelo. A mayor profundidad, el árbol puede capturar patrones más complejos, mejorando potencialmente el rendimiento. Sin embargo, valores excesivos incrementan el riesgo de sobreajustar el modelo [Dua et al., 2017].

Dado el desbalance de clases (93,8 % vs 6,2 %), árboles demasiado superficiales tienden a ignorar la clase minoritaria. Se exploraron valores de 5, 10 y 15, comenzando por el valor por defecto (5) y aumentando progresivamente para encontrar el equilibrio entre la complejidad y evitar el sobreajuste.

maxBins (número máximo de bins): Determina cómo se discretizan los atributos continuos y el número de divisiones posibles para atributos categóricos. El valor mínimo debe ser al menos igual al número de categorías del atributo categórico con mayor cardinalidad; en este dataset, el atributo ADTIND requiere 52 bins. Se evaluaron valores de 52, 60, 80 y 100. Valores más altos permiten mayor granularidad en las divisiones, pero más allá de cierto umbral pueden provocar sobreajuste sin mejorar el rendimiento [Dua et al., 2017].

Las 12 combinaciones se generaron mediante el producto cartesiano de ambos conjuntos de valores:

```
    val maxDepthValues = Array(5, 10, 15)
    val maxBinsValues = Array(52, 60, 80, 100)
    val combinations = maxDepthValues.flatMap(maxDepth => maxBinsValues.map(maxBins
     => (maxDepth, maxBins)))

```

```
** Combinaciones a explorar:
maxDepth: 5 - maxBins: 52
maxDepth: 5 - maxBins: 60
maxDepth: 5 - maxBins: 80
maxDepth: 5 - maxBins: 100
maxDepth: 10 - maxBins: 52
maxDepth: 10 - maxBins: 60
maxDepth: 10 - maxBins: 80
maxDepth: 10 - maxBins: 100
maxDepth: 15 - maxBins: 52
maxDepth: 15 - maxBins: 60
maxDepth: 15 - maxBins: 80
maxDepth: 15 - maxBins: 100
```

Figura 2: Combinaciones de parámetros exploradas

3.3. Consideraciones sobre la indexación de clases

Es importante aclarar cómo se indexaron las clases, ya que esto afecta la interpretación de las métricas. Se utilizó `StringIndexer` con `stringOrderType = ''alphabetDesc''`, lo que resulta en:

- Clase **0.0**: corresponde a "50000+." (ingresos $>$ 50.000 USD) — clase minoritaria (6,2 %)
- Clase **1.0**: corresponde a "- 50000." (ingresos $\leq$ 50.000 USD) — clase mayoritaria (93,8 %)

Esta codificación es relevante para la interpretación de métricas binarias como *AUC-ROC* y *AUC-PR*, que consideran la clase **1.0** como la positiva. En este contexto, el modelo intenta identificar a las personas con ingresos superiores a 50.000 dólares.

3.4. Métricas de evaluación

Dada la naturaleza desbalanceada del problema, se utilizaron múltiples métricas para obtener una evaluación comprehensiva del rendimiento de cada modelo:

Tasa de error: Calculada como 1−accuracy utilizando `MulticlassMetrics`. Este entregable se basa en la evaluación de los modelos mediante esta métrica, pero debido a que puede ser engañosa en problemas desbalanceados, se utilizarán métricas adicionales para realizar un análisis adicional que permitirá evaluar mejor en trabajos futuros.

Matriz de confusión, recall y precisión por clase: Proporcionadas por `MulticlassMetrics`, permiten analizar el comportamiento del modelo para cada clase individualmente, revelando si el modelo realmente identifica la clase minoritaria o simplemente predice la mayoritaria [Project, 2025b].

AUC-ROC y AUC-PR: Calculadas mediante `BinaryClassificationMetrics`. El *AUC-PR* es especialmente relevante en este contexto, ya que es más sensible al desbalance de clases y refleja mejor la capacidad del modelo para identificar correctamente la clase minoritaria (+50000) [Project, 2025a].

```
    /**
    * Evalua el modelo usando BinaryClassificationMetrics
    * Para obtener AUC-ROC y AUC-PR
    * ModelAucEvaluationMetrics es una case class creada para los resultados
    */
    def evaluateModelAuc(predictionsAndLabels: DataFrame):
    ModelAucEvaluationMetrics = {
      val probabilitiesAndLabelsRDD = predictionsAndLabels.select("label", "
    probability").rdd
        .map{row => (row.getAs[Vector](1).toArray, row.getDouble(0))}
        .map{r => (r._1(1), r._2)}

      val binaryMetrics = new BinaryClassificationMetrics(probabilitiesAndLabelsRDD
    )

      // AUC-ROC y AUC-PR
      val aucROC = binaryMetrics.areaUnderROC()
      val aucPR = binaryMetrics.areaUnderPR()

      ModelAucEvaluationMetrics(aucROC, aucPR)
    }

```

```
    /**
    * Evalua el modelo con MulticlassMetrics
    * Usa MulticlassMetrics para obtener el error, matriz de confusin, precision,
    recall
    */
    def evaluateMulticlassMetrics(predictions: DataFrame): MulticlassMetrics = {
        val predictionAndLabels = predictions.select("prediction", "label").rdd.map
    (row =>
            (row.getDouble(0), row.getDouble(1))
        )
        new MulticlassMetrics(predictionAndLabels)
    }
```

```
    /**
     * Evaluar el modelo con las metricas binarias y multiclase
     * Imprime el resultado de la evaluacion
     * Devuelve una tupla con las respectivas metricas para posterior comparacion
     */
    def evaluateModelWithSubset(subsetName: String, model:
    DecisionTreeClassificationModel, subSetDF: DataFrame) = {
      println("\n  Procesando Evaluacin en :" + subsetName)
      val predictionsAndLabels = model.transform(subSetDF)
      val multiclassMetrics = evaluateMulticlassMetrics(predictionsAndLabels)
      val aucEvaluationMetrics = evaluateModelAuc(predictionsAndLabels)
      printModelMetrics(multiclassMetrics, aucEvaluationMetrics)
      (multiclassMetrics, aucEvaluationMetrics)
    }
```

3.5. Procesando la evaluación de combinaciones de parámetros

Para cada una de las combinaciones se realizó la creación del modelo y su respectiva evaluación tanto para el conjunto *subTrain* como para *validation*. Finalmente, se obtuvieron los valores de las características del modelo para tenerlas presentes en la comparación.

```
    val results = combinations.map { case (maxDepth, maxBins) =>
      printTitle(s"Modelo: maxDepth=$maxDepth, maxBins=$maxBins")

      // Crear y entrenar el arbol de decision
      val dt = new DecisionTreeClassifier()
        .setMaxDepth(maxDepth)
        .setMaxBins(maxBins)

      val model = dt.fit(subTrainDF)
      val numNodes = model.numNodes

      println(f"  Nmero de nodos del arbol: $numNodes")

      // Evaluacion en conjunto de entrenamiento
      val (subTrainMetrics, subTrainAuc) = evaluateModelWithSubset("ENTRENAMIENTO",
```

```
      model, subTrainDF)

      // Evaluacion en conjunto de validacin
      val (validationMetrics, validationAuc) = evaluateModelWithSubset("VALIDACIN",
    model, validationDF)

      val modelCharacteristics = ModelCharacteristics(maxDepth, maxBins, model,
    numNodes)

      // Retornar resultado completo en el case class de resultados
      ModelResult(
        modelCharacteristics,
        subTrainMetrics, subTrainAuc,
        validationMetrics, validationAuc
      )
    }
```

Al ejecutar el map sobre combinaciones se crea un array del case class ModelResult `Array[ModelResult]` que contiene todas las métricas obtenidas, y mediante el cual se pueden imprimir en pantalla o exportar, si se desea, los resultados de todas las métricas que se muestran a continuación:

3.5.1. Conjunto de sub entrenamiento

*** TABLA RESUMEN - Métricas de Training:

Depth	Bins	Nodos	Error	Recall 0.0	Recall 1.0	Precision 0.0	Precision 1.0	AUC-ROC	AUC-PR
5	52	35	0,0647	0,1905	0,9953	0,7654	0,9386	0,8341	0,9843
5	60	39	0,0646	0,1918	0,9953	0,7660	0,9386	0,8394	0,9849
5	80	45	0,0643	0,2407	0,9916	0,6987	0,9419	0,8459	0,9831
5	100	49	0,0629	0,2266	0,9943	0,7622	0,9411	0,8444	0,9853
10	52	649	0,0554	0,3867	0,9895	0,7480	0,9525	0,9128	0,9917
10	60	621	0,0551	0,4043	0,9884	0,7375	0,9537	0,9074	0,9908
10	80	729	0,0528	0,4317	0,9887	0,7552	0,9558	0,9124	0,9914
10	100	665	0,0526	0,4257	0,9894	0,7633	0,9554	0,9141	0,9918
15	52	3305	0,0403	0,5555	0,9922	0,8518	0,9652	0,9542	0,9958
15	60	3115	0,0394	0,5638	0,9926	0,8591	0,9658	0,9532	0,9957
15	80	3463	0,0372	0,5741	0,9941	0,8863	0,9667	0,9566	0,9960
15	100	3217	0,0370	0,5835	0,9935	0,8784	0,9674	0,9556	0,9959

Figura 3: Resultados de las combinaciones exploradas en el conjunto subTrain

De esta tabla se puede concluir que en subTrain, con valores mayores de maxDepth, hay una clara disminución de la tasa de error, siendo la mejor combinación maxDepth 15 con maxBins 52, como se observa en la figura 4

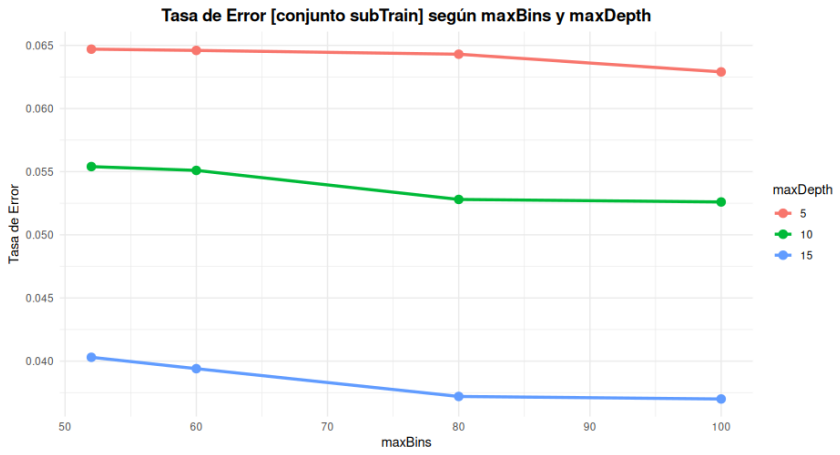

Figura 4: Comparación de las tasas de error en subTrain por maxBins y maxDepth

3.5.2. Conjunto de validación

*** TABLA RESUMEN - Métricas de Validación:

Depth	Bins	Nodos	Error	Recall 0.0	Recall 1.0	Precision 0.0	Precision 1.0	AUC-ROC	AUC-PR
5	52	35	0,0654	0,1833	0,9953	0,7600	0,9378	0,8303	0,8496
5	60	39	0,0654	0,1842	0,9952	0,7579	0,9379	0,8380	0,8505
5	80	45	0,0653	0,2264	0,9919	0,6936	0,9407	0,8423	0,9826
5	100	49	0,0637	0,2164	0,9945	0,7605	0,9402	0,8411	0,8699
10	52	649	0,0617	0,3399	0,9867	0,6737	0,9487	0,8875	0,9551
10	60	621	0,0613	0,3550	0,9858	0,6695	0,9498	0,8844	0,9554
10	80	729	0,0616	0,3662	0,9846	0,6580	0,9506	0,8856	0,9468
10	100	665	0,0596	0,3730	0,9862	0,6864	0,9511	0,8912	0,9640
15	52	3305	0,0656	0,3842	0,9788	0,5945	0,9516	0,8311	0,9630
15	60	3115	0,0650	0,3859	0,9794	0,6018	0,9518	0,8360	0,9634
15	80	3463	0,0640	0,3849	0,9805	0,6145	0,9518	0,8245	0,9572
15	100	3217	0,0627	0,4064	0,9802	0,6239	0,9534	0,8426	0,9636

Figura 5: Resultados de las combinaciones exploradas en el conjunto validation

Para validation, a diferencia de subTrain, se observa una tasa de error aparentemente más estable. Sin embargo, al analizar mediante la figura 6 se puede apreciar que existe una combinación con una tasa de error significativamente menor: maxDepth 10 y maxBins 100.

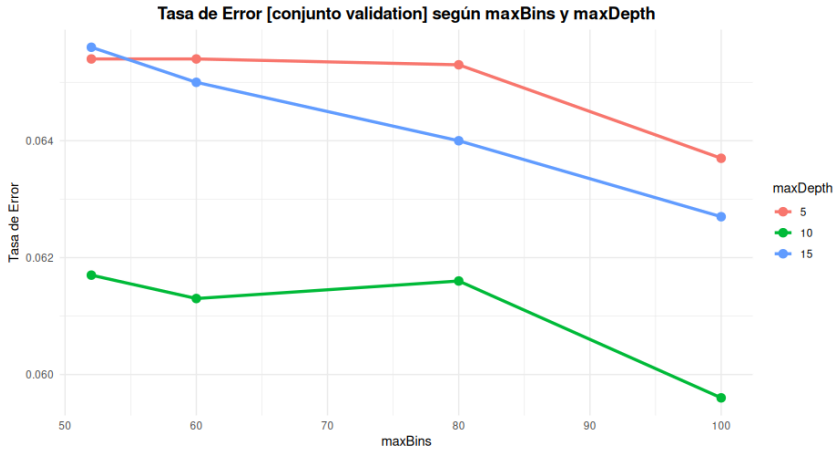

Figura 6: Comparación de las tasas de error en validation por maxBins y maxDepth

3.6. Análisis de resultados y selección del mejor modelo

3.6.1. Comparación de combinaciones de parámetros

Del análisis de las 12 combinaciones evaluadas se observan los siguientes patrones:

Efecto de maxBins: Los modelos con valores bajos de *maxBins* (52) presentan tasas de error superiores comparadas con valores más altos (80, 100), independientemente de la profundidad del árbol. Esto indica que una mayor granularidad en la discretización de atributos mejora la capacidad predictiva del modelo.

Detección de sobreajuste: Al comparar las métricas entre los conjuntos de subTrain y validation, se observa que a partir de $maxDepth = 15$, la diferencia en la tasa de error entre ambos conjuntos aumenta significativamente. Este comportamiento es indicativo de sobreajuste: el modelo se ajusta excesivamente a los datos de entrenamiento y pierde capacidad de generalización como se muestra en la figura 7.

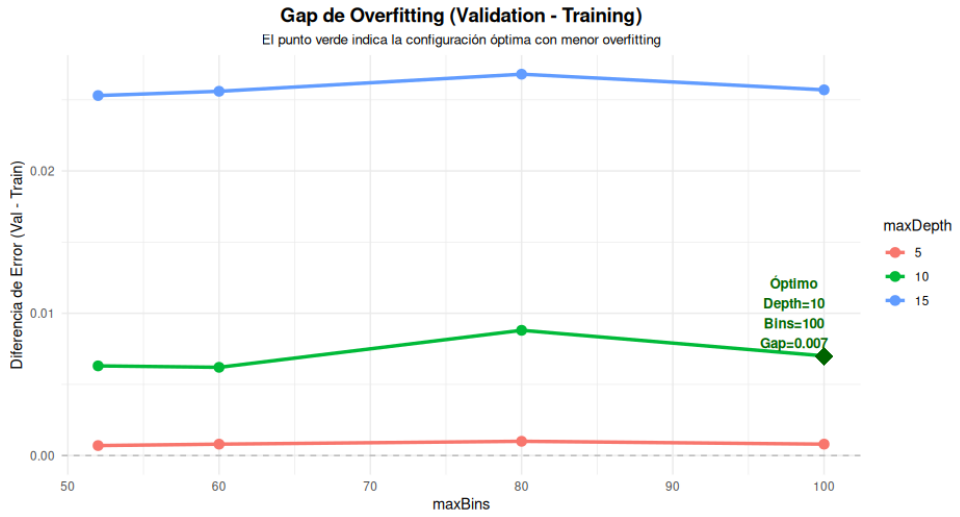

Figura 7: Visualización de la brecha de sobreajuste entre subTrain y validation

Selección del modelo óptimo: Con base en el criterio de minimizar la tasa de error en validación evitando el sobreajuste, se seleccionó el modelo con $maxDepth = 10$ y $maxBins = 100$. Esta configuración presenta:

- La menor tasa de error en el conjunto de validación
- Tasas de error similares entre subentrenamiento y validación, descartando sobreajuste
- Un balance adecuado entre complejidad del modelo y capacidad de generalización

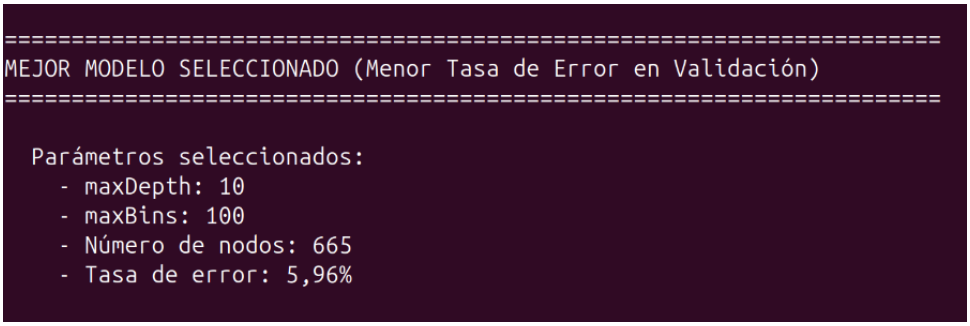

Figura 8: Configuración del modelo seleccionado: maxDepth=10, maxBins=100

3.6.2. Análisis crítico del rendimiento

Limitaciones del modelo seleccionado: Aunque la tasa de error general es baja (5,96 %), el análisis desagregado por clase revela limitaciones importantes. El *recall* para la clase minoritaria (ingresos mayores a 50.000 USD, label 0.0) es significativamente bajo (aproximadamente 36 %), lo que indica que el modelo tiene dificultades para identificar correctamente estos casos.

Como se observa en la matriz de confusión (ver figura 9), el modelo tiende a clasificar erróneamente muchas instancias de la clase minoritaria como pertenecientes a la clase mayoritaria. Este comportamiento es típico en problemas desbalanceados y subraya la importancia de no basarse únicamente en la tasa de error.

Implicaciones prácticas: Dependiendo del contexto de aplicación, la baja tasa de detección de la clase minoritaria podría tener consecuencias importantes. Para mejorar el rendimiento en esta clase, sería necesario explorar técnicas de balanceo de datos (SMOTE, undersampling, etc.), considerar otros hiperparámetros o considerar modelos más complejos como Random Forest, Gradient Boosting u otros.

Métricas complementarias: Los valores de *AUC-ROC* obtenidos (> 0.70) indican que el modelo tiene capacidad discriminativa superior a un clasificador aleatorio. Sin embargo, el *AUC-PR* por clase proporciona una evaluación más realista en este contexto desbalanceado.

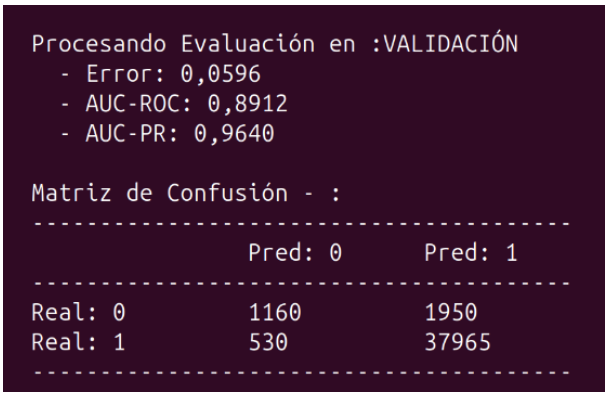

Figura 9: Matriz de confusión del modelo seleccionado en el conjunto de validación

3.7. Construcción del modelo final

Una vez identificada la configuración óptima ($maxDepth = 10$, $maxBins = 100$) con base en la tasa de error de validación, se construyó el modelo final utilizando todo el conjunto de entrenamiento original. El modelo se implementó como un pipeline completo que integra las

etapas de transformación y el árbol de decisión, facilitando su reutilización y garantizando que las transformaciones se apliquen consistentemente.

```
    // Crear clasificador con los mejores parametros
    val finalDT = new DecisionTreeClassifier()
    .setMaxDepth(bestModel.modelCharacteristics.maxDepth)
    .setMaxBins(bestModel.modelCharacteristics.maxBins)

    // Crear Pipeline completo: transformacin + modelo
    val completePipeline = new Pipeline().setStages(
      transformationPipeline.stages :+ finalDT
    )

    // Entrenar el pipeline completo con el conjunto de entrenamiento limpio y
    completo
    val finalCompleteModel = completePipeline.fit(selectedCensusDataDF)

```

3.8. Evaluación en el conjunto de prueba

El modelo final se evaluó sobre el conjunto de prueba, que no había sido utilizado en ninguna etapa previa del proceso de selección de parámetros. Para ello, se aplicó el mismo pipeline que contiene transformación para generar los vectores de características (*features*) y las etiquetas (*label*) y el modelo seleccionado.

```
    // Usar el pipeline completo: recibe datos limpios de Test y devuelve
    predicciones
    val testPredictions = finalCompleteModel.transform(selectedTestCensusDataDF)

    // Evaluar el modelo final en test
    val testMetrics = evaluateMulticlassMetrics(testPredictions)
    val testAuc = evaluateModelAuc(testPredictions)

```

3.9. Resultados finales en el conjunto de prueba

El modelo final presentó un comportamiento consistente con los resultados observados en el conjunto de validación (ver figura 10).

```
================================================================================
MÉTRICAS DEL MODELO FINAL EN TEST:
================================================================================
    - Error: 0,0593
    - AUC-ROC: 0,8957
    - AUC-PR: 0,9613

  Matriz de Confusión - :
  ---------------------------------------
                  Pred: 0      Pred: 1
  ---------------------------------------
  Real: 0         1960         4222
  Real: 1         720          76494
  ---------------------------------------
```

Figura 10: Resultados del modelo final utilizando el conjunto de pruebas

La consistencia entre las métricas de validación y prueba confirma que el modelo no presenta sobreajuste y que la metodología de selección de parámetros fue apropiada (ver figura 11) con base en la tasa de error.

```
 *** RESUMEN COMPARATIVO:
================================================================================
Métrica              Validación      Test            Diferencia
================================================================================
Tasa de Error        0,0596          0,0593          0,0003
AUC-ROC              0,8912          0,8957          0,0045
AUC-PR               0,9640          0,9613          0,0027
================================================================================
```

Figura 11: Comparación de resultados usando validation y test

4. Exportando el Pipeline del mejor modelo

Para exportar el pipeline que contiene la transformación más el modelo se utilizó un tryCatch que en caso de éxito crea o sobreescribe el modelo en la carpeta ./Model. Al ser un pipeline para ejecutarlo es necesario darle los datos limpios que pasarán a ser transformados y clasificados con el modelo de árbol de decision. Por ejemplo para usarlo directamente con test se puede ejecutar el siguiente código:

```
        // Cargar el modelo
        val loadedPipeline = PipelineModel.load("./Modelo")

        // Obtener las predicciones de test ya limpio
        val predictions = loadedPipeline.transform(selectedTestCensusDataDF)

        // Para realizar operaciones sobre el arbol de decision del pipeline
        val treeModel = loadedPipeline.stages(5).asInstanceOf[
    DecisionTreeClassificationModel]

```

5. Mejoras en la implementación

Se realizaron modificaciones al código entregado en la versión anterior, específicamente en la etapa de transformación de datos.

5.1. Pipeline de transformación reutilizable

Se creó un pipeline de transformación reutilizable que permite aplicar las mismas transformaciones (indexación de variables categóricas, creación del vector de características y codificación de la etiqueta) tanto al conjunto de entrenamiento como al conjunto de prueba, garantizando consistencia.

```
/**
 * Crear el pipeline para que sea reutilizable para conjuntos de entrenamiento
y tests
 */
val transformationPipeline = new Pipeline().setStages(Array(
  stringIndexerCategoricalCols,
  numericAssembler,
  normalizeNumericCols,
  featuresVector,
  labelIndexer
))

/**
 * Fit al pipeline SOLO con el conjunto de entrenamiento
 */
val transformationPipeline = transformationPipeline.fit(selectedCensusDataDF)

println("\n Transformando conjunto de ENTRENAMIENTO:")
val censusFeaturesLabelDF = transformationPipeline.transform(
selectedCensusDataDF).select("features", "label")
censusFeaturesLabelDF.show(5)

```

5.2. Normalización de datos numéricos

A partir del feedback recibido sobre normalizar los datos numéricos, se agregó la normalización utilizando *StandardScaler* [Project, 2025c].

```
/**
 * Normalizar los atributos numricos usando StandardScaler
 */
val normalizeNumericCols = new StandardScaler().
  setInputCol("numericFeatures").
  setOutputCol("scaledNumericFeatures").
  setWithStd(true).
  setWithMean(true)

```

6. Conclusiones

En este trabajo se llevó a cabo la selección de un modelo de clasificación basado en árboles de decisión para predecir el nivel de ingresos en el conjunto de datos Census Income (KDD), explorando 12 combinaciones de los parámetros *maxDepth* y *maxBins*.

Metodología aplicada: Se implementó una estrategia de validación apropiada, dividiendo el conjunto de entrenamiento en subconjuntos de entrenamiento (75 %) y validación (25 %), evitando el uso inadecuado del conjunto de prueba para la selección de parámetros. Esta metodología permitió identificar configuraciones con tendencia al sobreajuste y seleccionar el modelo que mejor equilibra capacidad predictiva y generalización.

Limitaciones identificadas: A pesar de una tasa de error general baja, el modelo presenta limitaciones significativas para identificar la clase minoritaria (ingresos mayores a 50.000 USD), con un *recall* de aproximadamente 36 %. Este comportamiento era esperado desde el análisis inicial del conjunto de datos, donde se evidenció el alto desbalance de clases (93,8 % vs 6,2 %), y destaca la importancia de no evaluar modelos únicamente mediante la tasa de error global.

Trabajo futuro: Para mejorar el rendimiento en la clase minoritaria, sería necesario explorar: (1) técnicas de balanceo de datos como SMOTE o undersampling, (2) otras combinaciones de hiperparámetros, (3) modelos de conjunto como Random Forest o Gradient Boosting Trees u otros, y (4) ajuste de umbrales de decisión considerando los costos asociados a falsos negativos y falsos positivos.

7. Tiempo empleado

Horas totales: 12 horas

Actividades realizadas:

- Desarrollo de scripts en Scala para la exploración de parámetros y evaluación de modelos
- Implementación de métricas de evaluación (MulticlassMetrics, BinaryClassificationMetrics)
- Análisis comparativo de las 12 combinaciones de parámetros
- Generación de visualizaciones y tablas de resultados
- Investigación sobre criterios de selección de parámetros en árboles de decisión
- Redacción y estructuración de la memoria técnica
- Análisis crítico de resultados y limitaciones del modelo

Referencias

[Dua et al., 2017] Dua, R., Ghotra, M. S., and Pentreath, N. (2017 - 2017). *Machine learning with spark : develop intelligent machine learning systems with spark 2.x*. Packt, Birmingham, England ;, second edition. edition.

[Project, 2025a] Project, A. S. (2025a). Binaryclassificationmetrics (pyspark mllib) — spark 4.0.1 documentation. https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.mllib.evaluation.BinaryClassificationMetrics.html#pyspark.mllib.evaluation.BinaryClassificationMetrics.areaUnderPR. Accessed: 2025-11-02.

[Project, 2025b] Project, A. S. (2025b). Multiclassmetrics (pyspark mllib) — spark 4.0.1 documentation. https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.mllib.evaluation.MulticlassMetrics.html. Accessed: 2025-11-02.

[Project, 2025c] Project, A. S. (2025c). Standardscaler (spark ml) — spark 4.0.1 scaladoc. https://spark.apache.org/docs/latest/api/scala/org/apache/spark/ml/feature/StandardScaler.html. Accessed: 2025-11-02.

[Quy et al., 2022] Quy, T. L., Roy, A., Iosifidis, V., Zhang, W., and Ntoutsi, E. (2022). A survey on datasets for fairness-aware machine learning. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 12(3):e1452.