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

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 tributos 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 ïnstance weight"que solo se utiliza como peso de muestra. 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"+50000": ingresos superiores a 50.000 USD (6,2Esta distribución altamente desbalanceada plantea un desafío adicional para los modelos de clasificación, que deberán ser evaluados teniendo en cuenta este desequilibrio de clases.

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:
         * NOTA IMPORTANTE: El conjunto de test NO se utiliza para selecci n
    de par metros
         */
        def createTrainingSubsets(originalDF: DataFrame, trainingPercentage:
    Double = 0.75, seed: Long): Array[Dataset[Row]] =
          originalDF.randomSplit(Array(trainingPercentage, 1 -
    trainingPercentage), seed)
```

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|114941|     92.57|
|  1.0|  9232|      7.43|
+-----+------+----------+


 *** Distribución de clases en validation:
+-----+-----+----------+
|label|count|porcentaje|
+-----+-----+----------+
|  0.0|38448|     92.48|
|  1.0| 3127|      7.52|
+-----+-----+----------+
```

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 aumentan el riesgo de sobreajuste [Dua et al., 201

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, 15, comenzando por el valor por defecto (5) y aumentando progresivamente para encontrar el equilibrio entre la complejidad y evitar overfit.

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, 20)
        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 codificaron 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 ≤ 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 (93,8 % vs 6,2 %), 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 precision 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.

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 positiva (+50000).

```
        // 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)
        }

```

```
        def evaluateMulticlassMetrics(predictions: DataFrame): MulticlassMetrics = {
            val predictionAndLabels = predictions.select("prediction", "label").rdd.map(row =>
                (row.getDouble(0), row.getDouble(1))
            )
            new MulticlassMetrics(predictionAndLabels)
        }

```

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 y validation. Finalmente, se obtuvo 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  rbol  de decision
          val dt = new DecisionTreeClassifier()
            .setMaxDepth(maxDepth)
            .setMaxBins(maxBins)

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

          println(f"  N mero de nodos del  rbol : $numNodes")

          // Evaluaci n en conjunto de entrenamiento
```

```
        val (subTrainMetrics, subTrainAuc) = evaluateModelWithSubset("ENTRENAMIENTO", model, subTrainDF)

        // Evaluaci n en conjunto de validaci n
        val (validationMetrics, validationAuc) = evaluateModelWithSubset("VALIDACI N", model, validationDF)

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

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

```

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	37	0,0630	0,9844	0,3464	0,9494	0,6410	0,8657	0,4830
5	60	41	0,0619	0,9867	0,3326	0,9485	0,6678	0,8656	0,4875
5	80	41	0,0624	0,9865	0,3281	0,9481	0,6616	0,8670	0,4908
5	100	35	0,0613	0,9874	0,3331	0,9485	0,6790	0,8662	0,4982
10	52	713	0,0535	0,9903	0,4006	0,9536	0,7687	0,9178	0,6207
10	60	727	0,0531	0,9906	0,4035	0,9539	0,7743	0,9221	0,6270
10	80	657	0,0532	0,9893	0,4182	0,9549	0,7577	0,9193	0,6235
10	100	611	0,0524	0,9912	0,4051	0,9540	0,7862	0,9221	0,6346
15	52	4083	0,0366	0,9898	0,6353	0,9713	0,8329	0,9700	0,8140
15	60	3791	0,0371	0,9901	0,6231	0,9703	0,8354	0,9673	0,8059
15	80	3799	0,0362	0,9907	0,6292	0,9708	0,8440	0,9682	0,8122
15	100	3445	0,0372	0,9905	0,6185	0,9700	0,8391	0,9672	0,8065

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

3.5.2. Conjunto de validación

*** 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	37	0,0630	0,9844	0,3464	0,9494	0,6410	0,8657	0,4830
5	60	41	0,0619	0,9867	0,3326	0,9485	0,6678	0,8656	0,4875
5	80	41	0,0624	0,9865	0,3281	0,9481	0,6616	0,8670	0,4908
5	100	35	0,0613	0,9874	0,3331	0,9485	0,6790	0,8662	0,4982
10	52	713	0,0535	0,9903	0,4006	0,9536	0,7687	0,9178	0,6207
10	60	727	0,0531	0,9906	0,4035	0,9539	0,7743	0,9221	0,6270
10	80	657	0,0532	0,9893	0,4182	0,9549	0,7577	0,9193	0,6235
10	100	611	0,0524	0,9912	0,4051	0,9540	0,7862	0,9221	0,6346
15	52	4083	0,0366	0,9898	0,6353	0,9713	0,8329	0,9700	0,8140
15	60	3791	0,0371	0,9901	0,6231	0,9703	0,8354	0,9673	0,8059
15	80	3799	0,0362	0,9907	0,6292	0,9708	0,8440	0,9682	0,8122
15	100	3445	0,0372	0,9905	0,6185	0,9700	0,8391	0,9672	0,8065

Figura 4: Resultados de las combinaciones exploradas en el conjunto validation

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 subentrenamiento y validación, 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.

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 para profundidades ≤ 10
- Tasas de error similares entre subentrenamiento y validación, descartando sobreajuste
- Un balance adecuado entre complejidad del modelo y capacidad de generalización

```
======================================================================
MEJOR MODELO SELECCIONADO (Menor Tasa de Error en Validación)
======================================================================

  Parámetros seleccionados:
    - maxDepth: 10
    - maxBins: 100
    - Número de nodos: 619
    - Tasa de error: 5,99%
```

Figura 5: 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 (5,9 %), el análisis desagregado por clase revela limitaciones importantes. El *recall* para la clase minoritaria (ingresos > 50.000, label 0.0) es significativamente bajo (aproximadamente 36 %), indicando que el modelo tiene dificultades para identificar correctamente estos casos.

Como se observa en la matriz de confusión (ver figura 6), 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.)

o considerar modelos más complejos como Random Forest o Gradient Boosting.

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* proporciona una evaluación más realista en este contexto desbalanceado.

```
Procesando Evaluación en :VALIDACIÓN
  - Error: 0,0599
  - AUC-ROC: 0,8992
  - AUC-PR: 0,9760

Matriz de Confusión - :
---------------------------------------
               Pred: 0      Pred: 1
---------------------------------------
Real: 0        1125         2004
Real: 1        488          37958
---------------------------------------
```

Figura 6: Matriz de confusión del modelo seleccionado en el conjunto de validación

3.7. Construcción del modelo final

Una vez identificada la configuración óptima ($maxDepth = 10$, $maxBins = 100$), se procedió a entrenar el modelo final utilizando todo el conjunto de entrenamiento original (sin dividir en subconjuntos). Este modelo final fue entrenado con los parámetros seleccionados y todos los demás parámetros en sus valores por defecto, según lo especificado en el entregable.

```
        // Entrenar el modelo final con todo el conjunto de entrenamiento
        val finalDT = new DecisionTreeClassifier()
          .setMaxDepth(10)
          .setMaxBins(100)

        val finalModel = finalDT.fit(trainingFeaturesLabelDF)

```

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 de transformación utilizado en el entrenamiento para generar los vectores de características (*features*) y las etiquetas (*label*).

```
        // Transformar el conjunto de prueba
        val testCensusFeaturesLabelDF = transformationPipeline.transform(
    selectedTestCensusDataDF)
          .select("features", "label")

        // Realizar predicciones y calcular m tricas
        val testPredictions = finalModel.transform(testCensusFeaturesLabelDF)
        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 7

```
================================================================================
MÉTRICAS DEL MODELO FINAL EN TEST:
================================================================================
   - Error: 0,0586
   - AUC-ROC: 0,8963
   - AUC-PR: 0,9614

  Matriz de Confusión - :
  ---------------------------------------
                 Pred: 0      Pred: 1
  ---------------------------------------
  Real: 0         2203         3979
  Real: 1         907          76307
  ---------------------------------------
```

Figura 7: 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 8. Sin embargo, persisten las limitaciones inherentes al desbalance de clases, con bajo *recall* para la clase minoritaria.

```
 *** RESUMEN COMPARATIVO:
================================================================================
Métrica              Validación      Test            Diferencia
================================================================================
Tasa de Error        0,0599          0,0586          0,0014
AUC-ROC              0,8992          0,8963          0,0030
AUC-PR               0,9760          0,9614          0,0146
================================================================================
```

Figura 8: Comparacion de resultados usando validation y test.

4. 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 $>$ 50.000 USD), con un *recall* de aproximadamente 36 %. Este comportamiento se esperaba desde el análisis del conjunto de datos en el cual 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) modelos de conjunto como Random Forest o Gradient Boosting Trees, y (3) ajuste de umbrales de decisión considerando los costos asociados a falsos negativos y falsos positivos.

5. Tiempo empleado

Horas totales: 10 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.