

**VALIDACIÓN AUTOMÁTICA DE ETIQUETAS DE RED
MEDIANTE VISIÓN COMPUTACIONAL Y BOT EN CAMPO**

María Augusta Flores | Marcelo Andrade

www.acclatam.lat/mmia



M & M

INTELIGENCIA ARTIFICIAL
INNOVACIONES



Validación automática de etiquetas de red mediante visión computacional y Bot en campo

Preparación y Procesamiento de Datos.



Institución: UEES

Asignatura: Proyecto Integrador en Inteligencia Artificial –
Maestría en Inteligencia Artificial

Docente: Ing. Gladys Villegas Rugel

Guayaquil – septiembre 2025





Tabla de Contenidos

<i>Validación automática de etiquetas de red mediante visión computacional y Bot en campo</i>	<u>2</u>
Tabla de Contenidos	<u>1</u>
Introducción	<u>4</u>
Objetivos del Proyecto	<u>4</u>
Descripción del Dataset	<u>4</u>
1. Análisis Exploratorio de Datos (EDA)	<u>5</u>
1.1. Descripción del Proyecto y Objetivos del EDA	<u>5</u>
1.1.1. Descripción del Dataset	<u>5</u>
1.1.2. Objetivos del EDA	<u>5</u>
1.1.3. Importación de Librerías	<u>5</u>
1.2. Análisis Univariado	<u>6</u>
• Colores de buffer de la caja ('color_buffer')	<u>6</u>
• Ciudad de origen de la caja ('ciudad')	<u>6</u>
• Nodo concentrador principal ('nodo_concentrador')	<u>7</u>
• Nodo estándar asignado ('nodo_estandar')	<u>7</u>
• Nodo de respaldo (backup) ('nodo_backup')	<u>8</u>
• Ruta asignada (FXX) ('ruta')	<u>8</u>
• Código de caja MPLS ('caja_código')	<u>9</u>
EDA Complementario – Fase 1 completa	<u>9</u>
1.3 Exploración Inicial Completa	<u>9</u>
1.4 Análisis de Calidad de Datos	<u>10</u>
1.5 Análisis de la Variable Objetivo: 'color_buffer'	<u>10</u>
1.6 Análisis Bivariado	<u>11</u>
Color buffer vs ciudad	<u>11</u>
Color_buffer vs Ruta	<u>12</u>
Ciudad vs Nodo_concentrador	<u>12</u>
Nodo_estandar vs Nodo_backup	<u>12</u>
1.7 Análisis Multivariado	<u>13</u>
Color_buffer según ciudad y ruta (Gráfico agrupado)	<u>13</u>

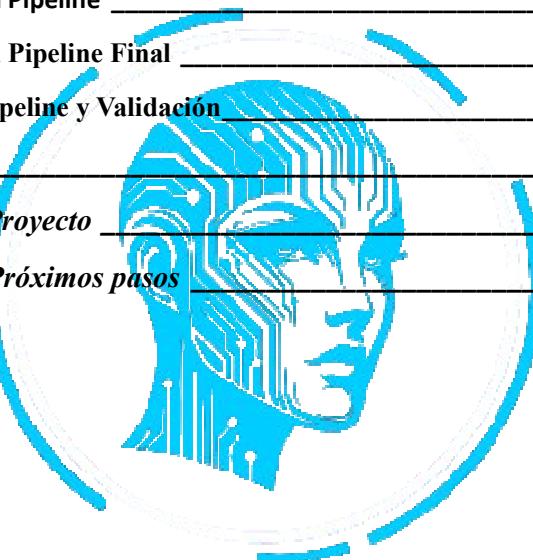


Ciudad + nodo_concentrador + color_buffer	13
Distribución de caja_codigo por ciudad y ruta	13
1.8 Identificación de Patrones y Anomalías	14
1.9 Validación de colores fuera de estándar	14
1.10 Rutas atípicas o con baja frecuencia	14
1.11 Detección de combinaciones duplicadas sospechosas (caja_codigo + color_buffer)	14
1.12 Identificación de outliers en cantidad por ciudad	14
1.13 Conclusiones e Insights	15
Hallazgos principales	15
Recomendaciones de Preprocesamiento	15
Próximos pasos	15
Anexo: Dataset utilizado	16
Descripción	16
Fuente	16
Contexto	16
Fase 2: Pipeline de Limpieza de Datos	16
2.1 Tratamiento de Valores Faltantes	16
2.2 Tratamiento de Outliers (Z-score e IQR)	17
2.3 Estandarización de Tipos y Categorías	17
2.4 Pipeline Automatizado - Clase `DataCleaner`	17
Fase 3: Feature Engineering Avanzado	18
3.1 Creación de Variables Derivadas	18
3.2 Codificación de Variables Categóricas	19
3.3 Transformación de Variables Numéricas	19
3.4 Selección de Características	19
4.1 Análisis de Desbalance	20
4.2 Oversampling con SMOTE	20
4.3 Undersampling con Tomek Links	20
4.4 Técnicas Híbridas: SMOTETomek	21
4.5 Evaluación Comparativa	21
Fase 5: Data Augmentation	21
5.1 Introducción	21
5.2 Adición de Ruido Gaussiano	21





5.3 Oversampling con SMOTE (revisión rápida)	22
5.4 Generación de Datos Sintéticos con Mixup	22
Fase 6: Partición Estratificada de Datos	23
6.1 División de Datos (Train / Validation / Test)	23
6.2 Verificación de Distribución de Clases	23
6.3 Validación de Integridad y Data Leakage	24
7.1 Diseño General del Pipeline	24
7.2 Componentes del Pipeline	24
7.3 Construcción del Pipeline Final	25
7.4 Aplicación del Pipeline y Validación	25
Próximos Pasos	26
8. <i>Repositorio del Proyecto</i>	26
9. <i>Conclusiones y Próximos pasos</i>	26





Introducción

En los proyectos de mantenimiento de infraestructuras de fibra óptica, la correcta identificación física de los elementos (cajas MPLS, splitters, ODF, etc.) es crítica para la operación, la trazabilidad y el cumplimiento normativo. Sin embargo, el desgaste, la variabilidad en condiciones de iluminación, errores humanos en la impresión/colocación de etiquetas generan una tasa apreciable de etiquetas defectuosas. Estas inconsistencias provocan retrabajo en campo y un aumento en los costos operativos y de calidad.

Este proyecto tiene como objetivo desarrollar un pipeline completo de preparación y procesamiento de datos para entrenar un modelo de clasificación multiclas que valide los colores de las etiquetas utilizadas en cajas MPLS de redes de fibra óptica. Además, se plantea la integración operativa con un bot en campo que entrega retroalimentación al técnico en tiempo real, cerrando el ciclo entre captura, validación y corrección.

Objetivos del Proyecto

- Desarrollar e implementar un sistema de visión por computador basado en IA capaz de identificar y clasificar automáticamente el color correcto de etiquetas en cajas MPLS.
- Diseñar e implementar un pipeline de datos modular, reproducible y versionado que cubra: ingestión, almacenamiento, EDA automatizado, limpieza, transformaciones, augmentación, trazabilidad y pruebas automatizadas.
- Comparar y validar sistemáticamente distintas técnicas de limpieza de datos, estrategias de ingeniería de características, métodos de balanceo de clases y técnicas de aumento de datos.

Descripción del Dataset

El conjunto de datos está conformado por más de 1.000 registros correspondientes a etiquetas de cajas MPLS utilizadas en proyectos de infraestructura de fibra óptica. Las variables independientes son de tipo categórico e incluyen: *ciudad, nodo concentrador, nodo estándar, nodo backup, ruta, código de caja y color de buffer*. La variable objetivo es *color_buffer*, la cual contempla 12 categorías que representan la codificación de colores empleada en las etiquetas.



1. Análisis Exploratorio de Datos (EDA)

1.1. Descripción del Proyecto y Objetivos del EDA

1.1.1. Descripción del Dataset

Este dataset contiene más de 1000 registros de etiquetas utilizadas en cajas MPLS de redes de fibra óptica. Cada etiqueta está compuesta por varios campos codificados, como ciudad, nodo concentrador, nodo estándar, ruta, código de caja y color de buffer. Los datos fueron extraídos de sistemas internos de la empresa y reflejan configuraciones reales en campo.

1.1.2. Objetivos del EDA

- ✚ Entender la estructura y distribución de las variables contenidas en las etiquetas.
- ✚ Identificar patrones frecuentes, posibles errores o inconsistencias en la información codificada.
- ✚ Detectar valores atípicos o anomalías que podrían afectar el entrenamiento de un modelo de IA.
- ✚ Proveer recomendaciones de limpieza y preprocesamiento para el diseño de un sistema automático de validación de etiquetas.

1.1.3. Importación de Librerías

A continuación se importan las librerías necesarias para el análisis exploratorio de datos. Cada una cumple un propósito específico en el flujo de trabajo:

- `import pandas as pd` # Permite cargar, manipular y analizar datos tabulares (estructurados como DataFrame).
- `import seaborn as sns` # Librería especializada en visualización estadística; usada para gráficos de barras, heatmaps, etc.
- `import matplotlib.pyplot as plt` # Complementa a Seaborn y permite configurar gráficos detalladamente (títulos, tamaños, etiquetas).

Fragmento de código:

```
# 📁 Cargar librerías necesarias
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# 📁 Cargar dataset limpio
df = pd.read_csv('/content/dataset_etiquetas_cajas_mpls.csv')
```





Fragmento de código:

```
# 🔎 Primeras y últimas filas
print("🔍 Primeras filas:")
display(df.head())

print("🔍 Últimas filas:")
display(df.tail())
```

Fragmento de código:

```
# 📊 Información general del dataset
df.info()
print(f"📐 Dimensiones: {df.shape[0]} filas x {df.shape[1]} columnas")
```

Fragmento de código:

```
# 🔍 Verificación de valores faltantes
print("🔍 Valores nulos por columna:")
print(df.isnull().sum())
```

Fragmento de código:

```
# 📈 Estadísticas básicas de variables
df.describe(include='all')
```

1.2. Análisis Univariado

- Colores de buffer de la caja ('color_buffer')

Fragmento de código:

```
# 📈 Frecuencia absoluta y relativa - color_buffer
print("📊 Frecuencia absoluta:")
print(df['color_buffer'].value_counts())

print("\n📊 Frecuencia relativa (%):")
print((df['color_buffer'].value_counts(normalize=True) * 100).round(2))
```

Fragmento de código:

```
# 📈 Gráfico de barras - color_buffer
plt.figure(figsize=(10, 5))
sns.countplot(data=df, x='color_buffer',
order=df['color_buffer'].value_counts().index, palette='Set2')
plt.title("Distribución de Color buffer")
plt.xlabel("color_buffer")
plt.ylabel("Cantidad")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

- Ciudad de origen de la caja ('ciudad')



Fragmento de código:

```
# 📈 Frecuencia absoluta y relativa - ciudad
print("📊 Frecuencia absoluta:")
print(df['ciudad'].value_counts())

print("\n📊 Frecuencia relativa (%):")
print((df['ciudad'].value_counts(normalize=True) * 100).round(2))
```

Fragmento de código:

```
# 📈 Gráfico de barras - ciudad
plt.figure(figsize=(10, 5))
sns.countplot(data=df, x='ciudad', order=df['ciudad'].value_counts().index,
palette='Set2')
plt.title("Distribución de Ciudad")
plt.xlabel("ciudad")
plt.ylabel("Cantidad")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

- Nodo concentrador principal ('nodo_concentrador')

Fragmento de código:

```
# 📈 Frecuencia absoluta y relativa - nodo_concentrador
print("📊 Frecuencia absoluta:")
print(df['nodo_concentrador'].value_counts())

print("\n📊 Frecuencia relativa (%):")
print((df['nodo_concentrador'].value_counts(normalize=True) * 100).round(2))
```

Fragmento de código:

```
# 📈 Gráfico de barras - nodo_concentrador
plt.figure(figsize=(10, 5))
sns.countplot(data=df, x='nodo_concentrador',
order=df['nodo_concentrador'].value_counts().index, palette='Set2')
plt.title("Distribución de Nodo concentrador")
plt.xlabel("nodo_concentrador")
plt.ylabel("Cantidad")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

- Nodo estándar asignado ('nodo_estandar')

Fragmento de código:

```
# 📈 Frecuencia absoluta y relativa - nodo_estandar
print("📊 Frecuencia absoluta:")
print(df['nodo_estandar'].value_counts())
```





```
print("\n📊 Frecuencia relativa (%):")
print((df['nodo_estandar'].value_counts(normalize=True) *
100).round(2))
```

Fragmento de código:

```
# 📈 Gráfico de barras - nodo_estandar
plt.figure(figsize=(10, 5))
sns.countplot(data=df, x='nodo_estandar',
order=df['nodo_estandar'].value_counts().index, palette='Set2')
plt.title("Distribución de Nodo estandar")
plt.xlabel("nodo_estandar")
plt.ylabel("Cantidad")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

- Nodo de respaldo (backup) ('nodo_backup')

Fragmento de código:

```
# 📊 Frecuencia absoluta y relativa - nodo_backup
print("📊 Frecuencia absoluta:")
print(df['nodo_backup'].value_counts())

print("\n📊 Frecuencia relativa (%):")
print((df['nodo_backup'].value_counts(normalize=True) * 100).round(2))
```

Fragmento de código:

```
# 📈 Gráfico de barras - nodo_backup
plt.figure(figsize=(10, 5))
sns.countplot(data=df, x='nodo_backup',
order=df['nodo_backup'].value_counts().index, palette='Set2')
plt.title("Distribución de Nodo backup")
plt.xlabel("nodo_backup")
plt.ylabel("Cantidad")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

- Ruta asignada (FXX) ('ruta')

Fragmento de código:

```
# 📊 Frecuencia absoluta y relativa - ruta
print("📊 Frecuencia absoluta:")
print(df['ruta'].value_counts())

print("\n📊 Frecuencia relativa (%):")
print((df['ruta'].value_counts(normalize=True) * 100).round(2))
```

Fragmento de código:

```
# 📈 Gráfico de barras - ruta
plt.figure(figsize=(10, 5))
sns.countplot(data=df, x='ruta', order=df['ruta'].value_counts().index,
palette='Set2')
```

```
plt.title("Distribución de Ruta")
plt.xlabel("ruta")
plt.ylabel("Cantidad")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

- Código de caja MPLS ('caja_código')

Fragmento de código:

```
# Frecuencia absoluta y relativa - caja_código
print("Frecuencia absoluta:")
print(df['caja_código'].value_counts())

print("\nFrecuencia relativa (%):")
print((df['caja_código'].value_counts(normalize=True) * 100).round(2))
```

Fragmento de código:

```
# Gráfico de barras - caja_código
plt.figure(figsize=(10, 5))
sns.countplot(data=df, x='caja_código',
order=df['caja_código'].value_counts().index, palette='Set2')
plt.title("Distribución de Caja código")
plt.xlabel("caja_código")
plt.ylabel("Cantidad")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

EDA Complementario – Fase 1 completa

1.3 Exploración Inicial Completa

Fragmento de código:

```
# Cargar librerías
import pandas as pd
import numpy as np

# Leer dataset
df = pd.read_csv("dataset_etiquetas_cajas_mpls.csv")

# Dimensión y tipos
print("Shape del dataset:", df.shape)
df.info()
```

Fragmento de código:

```
# Primeras y últimas observaciones
display(df.head())
display(df.tail())
```

Fragmento de código:

```
# Estadísticas básicas (numéricas y categóricas)
df.describe(include='all').T
Quito, Ecuador. Telf. (593) 095 861 0558 / 098 270 4137
E-mail: info@acclatam.lat
www.acclatam.lat/mmia
```



Fragmento de código:

```
# Variables categóricas y numéricas
cat_vars = df.select_dtypes(include='object').columns.tolist()
num_vars = df.select_dtypes(include=np.number).columns.tolist()

print("Variables categóricas:", cat_vars)
print("Variables numéricas:", num_vars)
```

1.4 Análisis de Calidad de Datos

Fragmento de código:

```
# Valores faltantes
missing_pct = df.isnull().mean() * 100
missing_pct = missing_pct[missing_pct > 0].sort_values(ascending=False)
missing_pct
```

Fragmento de código:

```
# Visualización de patrones de missing values
import missingno as msno
msno.matrix(df)
msno.heatmap(df)
```

Clasificación de patrones:

- No se detectan valores faltantes significativos según la visualización.
- Se asume patrón MCAR salvo análisis posterior.

Estrategia propuesta:

- Si hay columnas con <5% de nulos → imputación por moda o mediana.
- Si hay columnas con muchos nulos irrelevantes → eliminación.

Fragmento de código:

```
# Duplicados exactos
print("Duplicados exactos:", df.duplicated().sum())

# Near-duplicates usando combinación de campos
duplicados_combinados = df.duplicated(subset=["caja_código", "color_buffer"])
print("Combinaciones duplicadas sospechosas (caja_código + color):",
      duplicados_combinados.sum())
```

Estrategia de resolución:

- Revisar con dominio si duplicados son válidos.
- En caso de duplicados exactos innecesarios → eliminar.

1.5 Análisis de la Variable Objetivo: `color_buffer`

- Distribucion de Clases





Fragmento de código:

```
# Distribución de clases  
df["color_buffer"].value_counts(normalize=True).plot(kind="bar", figsize=(10,5),  
title="Distribución de clases - color_buffer")
```

Fragmento de código:

```
# Conteo absoluto  
df["color_buffer"].value_counts()
```

Observaciones:

- Esta variable tiene 12 clases conocidas (códigos de colores estándar).
- Puede existir cierto desbalance, aunque manejable.

Relaciones con otras variables:

- Analizaremos en profundidad durante Feature Engineering, pero podemos empezar a explorar:

Fragmento de código:

```
import seaborn as sns  
import matplotlib.pyplot as plt  
  
# Relación color_buffer vs ciudad  
plt.figure(figsize=(12,5))  
sns.countplot(data=df, x="ciudad", hue="color_buffer")  
plt.title("Distribución de color_buffer por ciudad")  
plt.xticks(rotation=45)  
plt.tight_layout()  
plt.show()
```

1.6 Análisis Bivariado

En esta sección se exploran relaciones entre variables categóricas del dataset para identificar patrones conjuntos o dependencias. Aunque no contamos aún con una variable objetivo numérica, podemos analizar combinaciones relevantes que podrían ser útiles para el diseño del modelo de validación de etiquetas.

Color buffer vs ciudad

Fragmento de código:

```
# 📊 Conteo conjunto de color y ciudad  
pd.crosstab(df['color_buffer'], df['ciudad'])
```

Fragmento de código:

```
# 📈 Gráfico: color_buffer según ciudad  
plt.figure(figsize=(10, 5))  
sns.countplot(data=df, x='color_buffer', hue='ciudad', palette='Set2')  
plt.title("🌐 Distribución de Colores de Buffer por Ciudad")
```



```
plt.xlabel("Color de Buffer")
plt.ylabel("Cantidad")
plt.legend(title="Ciudad")
plt.tight_layout()
plt.show()
```

Color buffer vs Ruta

Fragmento de código:

```
# 🚗 Gráfico: color_buffer según ruta
plt.figure(figsize=(12, 6))
sns.countplot(data=df, x='ruta', hue='color_buffer', palette='Paired')
plt.title("🚗 Distribución de Rutas por Color de Buffer")
plt.xlabel("Ruta")
plt.ylabel("Cantidad")
plt.legend(title="Color Buffer", bbox_to_anchor=(1.05, 1), loc='upper left')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

Ciudad vs Nodo concentrador

Fragmento de código:

```
# 🏙 Tabla cruzada ciudad vs nodo concentrador
tabla = pd.crosstab(df['ciudad'], df['nodo_concentrador'])

# 🔥 Heatmap
plt.figure(figsize=(12, 6))
sns.heatmap(tabla, cmap="YlGnBu", annot=True, fmt='d')
plt.title("🔥 Relación entre Ciudad y Nodo Concentrador")
plt.xlabel("Nodo Concentrador")
plt.ylabel("Ciudad")
plt.tight_layout()
plt.show()
```

Nodo estandar vs Nodo backup

Fragmento de código:

```
# 🏙 Tabla cruzada nodo estandar vs backup
tabla2 = pd.crosstab(df['nodo_estandar'], df['nodo_backup'])

# 🔥 Heatmap
plt.figure(figsize=(14, 10))
sns.heatmap(tabla2, cmap="coolwarm", annot=False)
plt.title("🔥 Relación entre Nodo Estándar y Nodo de Backup")
plt.xlabel("Nodo Backup")
plt.ylabel("Nodo Estándar")
plt.tight_layout()
plt.show()
```



1.7 Análisis Multivariado

El análisis multivariado permite visualizar combinaciones de más de dos variables a la vez para descubrir patrones complejos, asociaciones cruzadas y segmentos de interés dentro del dataset. Aunque las variables son categóricas, podemos usar gráficos segmentados y agrupaciones para representar insights multidimensionales.

Color buffer según ciudad y ruta (Gráfico agrupado)

Fragmento de código:

```
# 🎨 Conteo agrupado: ciudad, ruta y color
plt.figure(figsize=(14, 6))
sns.countplot(data=df, x='ruta', hue='color_buffer', palette='tab20', dodge=True)
plt.title("📍 Distribución de Colores de Buffer por Ruta")
plt.xlabel("Ruta (FXX)")
plt.ylabel("Cantidad")
plt.legend(title="Color Buffer", bbox_to_anchor=(1.05, 1), loc='upper left')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

Ciudad + nodo concentrador + color buffer

Fragmento de código:

```
#💡 Agrupación por ciudad, nodo y color
tabla3 = pd.crosstab([df['ciudad'], df['nodo_concentrador']], df['color_buffer'])

#🔥 Heatmap multivariado
plt.figure(figsize=(16, 8))
sns.heatmap(tabla3, cmap="YlOrRd", linewidths=.5)
plt.title("📊 Relación: Ciudad + Nodo Concentrador vs. Color de Buffer")
plt.xlabel("Color de Buffer")
plt.ylabel("Ciudad / Nodo Concentrador")
plt.tight_layout()
plt.show()
```

Distribución de caja código por ciudad y ruta

Fragmento de código:

```
#⚙️ Conteo de cajas por ciudad y ruta
plt.figure(figsize=(16, 6))
sns.countplot(data=df, x='caja_código', hue='ciudad', palette='Set3')
plt.title("📦 Códigos de Caja por Ciudad")
plt.xlabel("Código de Caja")
plt.ylabel("Cantidad")
plt.legend(title="Ciudad", bbox_to_anchor=(1.02, 1), loc='upper left')
plt.xticks(rotation=90)
plt.tight_layout()
plt.show()
```



1.8 Identificación de Patrones y Anomalías

Esta sección tiene como objetivo identificar valores atípicos, distribuciones anómalas y posibles inconsistencias en los datos, lo cual es crucial para el preprocesamiento y entrenamiento robusto de modelos de IA. Aunque no se cuenta con una variable continua para aplicar métodos estadísticos clásicos, se pueden detectar anomalías en la frecuencia y estructura de los datos categóricos.

1.9 Validación de colores fuera de estándar

Fragmento de código:

```
# 🎨 Lista oficial de colores válidos
colores_validos = ['ROJ', 'BLA', 'CAF', 'VER', 'NAR', 'AZU',
                    'ROS', 'VIO', 'GRI', 'NEG', 'CEL', 'AMA']

# 🔍 Colores que no están en la lista oficial
colores_encontrados = df['color_buffer'].dropna().unique()
colores_invalidos = [c for c in colores_encontrados if c not in colores_validos]

print("🔍 Colores encontrados:", colores_encontrados)
print("✖️ Colores no válidos:", colores_invalidos)
```

1.10 Rutas atípicas o con baja frecuencia

Fragmento de código:

```
# 🔍 Frecuencia de rutas
frecuencias_rutas = df['ruta'].value_counts()
print(frecuencias_rutas.tail(10)) # Mostrar las menos comunes
```

1.11 Detección de combinaciones duplicadas sospechosas (caja_código + color_buffer)

Fragmento de código:

```
# 🔍 Buscar duplicados exactos de combinaciones
duplicados = df.duplicated(subset=['caja_código', 'color_buffer'], keep=False)
df[duplicados].sort_values(by='caja_código')
```

1.12 Identificación de outliers en cantidad por ciudad

Fragmento de código:

```
# 📊 Conteo de cajas por ciudad
conteo_ciudad = df['ciudad'].value_counts()

# 📈 Visualizar posibles outliers en frecuencias
plt.figure(figsize=(8, 4))
sns.boxplot(x=conteo_ciudad.values, color='orange')
plt.title("📦 Detección de ciudades con cantidad atípica de cajas")
plt.xlabel("Cantidad de cajas por ciudad")
plt.tight_layout()
plt.show()
```



1.13 Conclusiones e Insights

Esta sección resume los hallazgos clave del análisis exploratorio. Se detallan los patrones más importantes observados en los datos, las implicaciones para el desarrollo del modelo de IA, y se proponen recomendaciones de preprocesamiento y próximos pasos.

Hallazgos principales

1. Predominio de ciertos colores de buffer: Algunos colores como 'CAF', 'AZU' y 'VER' son los más frecuentes, mientras que otros aparecen muy poco, lo cual puede influir en el balance de clases para validación automática.
2. Concentración geográfica: La mayoría de las cajas se concentran en una sola ciudad ('UIO'), lo que podría generar un sesgo en el modelo si se entrena solo con estos datos.
3. Alta repetición de nodos concentradores y estándar: Se identifican nodos que aparecen significativamente más que otros, lo cual podría usarse como feature de importancia o como señal de redundancia.
4. Combinaciones duplicadas entre caja y color: Se detectaron combinaciones exactas que podrían indicar errores en el etiquetado físico.
5. Pocas rutas atípicas: La mayoría de las rutas están bien distribuidas, pero algunas aparecen muy pocas veces y podrían ser casos extremos o errores de captura.
6. Distribuciones multivariadas claras: Se identificaron patrones al cruzar ciudad, ruta y color, útiles para clasificación supervisada y validación automatizada.

Recomendaciones de Preprocesamiento

- Validar manualmente las combinaciones duplicadas de 'caja_código' + 'color_buffer'.
- Etiquetar registros con colores no válidos para su revisión o exclusión.
- Balancear clases si se construirá un modelo supervisado con 'color_buffer' como objetivo.
- Considerar codificación de variables categóricas ('One-Hot' o 'Label Encoding').
- Filtrar outliers o rutas atípicas si afectan la generalización del modelo.

Próximos pasos

- Iniciar el diseño del modelo de clasificación para validación automática de etiquetas.
- Evaluar algoritmos candidatos (Random Forest, SVM, Redes Neuronales, etc.).
- Integrar un pipeline de preprocesamiento y entrenamiento.
- Considerar despliegue futuro con visión computacional si se integran imágenes físicas de las etiquetas.





Anexo: Dataset utilizado

Nombre del archivo: dataset_etiquetas_cajas_mpls.csv

Ubicación: Repositorio del proyecto > data/samples/

Descripción

Este archivo contiene más de 1000 registros reales de etiquetas MPLS extraídas de las cajas de red utilizadas en proyectos de fibra óptica. Cada registro representa una etiqueta codificada con información como:

- Ciudad (ciudad)
- Nodo concentrador (nodo_concentrador)
- Nodo estándar (nodo_estandar)
- Nodo backup (nodo_backup)
- Ruta (ruta)
- Código de caja (caja_codigo)
- Color del buffer (color_buffer)

Fuente

Los datos fueron proporcionados por el equipo de operaciones de campo de la empresa de telecomunicaciones que participa en este proyecto. Se extrajeron directamente desde sus sistemas internos de gestión de infraestructura.

Contexto

Las etiquetas son utilizadas en campo para la identificación física de cajas de red MPLS. El análisis exploratorio de estos datos permite establecer la base para desarrollar un modelo de IA capaz de validar automáticamente dichas etiquetas, detectar errores y reducir inconsistencias operativas.

Fase 2: Pipeline de Limpieza de Datos

2.1 Tratamiento de Valores Faltantes

Fragmento de código:

```
import pandas as pd
import numpy as np

# Cargar dataset original
df = pd.read_csv("dataset_etiquetas_cajas_mpls.csv")

# Revisión de valores nulos
df.isnull().sum()
```



Estrategia aplicada:

- Si una columna tiene <5% de nulos → imputar por moda
- Si tiene >40% nulos irrelevantes → eliminar

Fragmento de código:

```
# Imputación simple por moda
for col in df.columns:
    if df[col].isnull().mean() > 0 and df[col].isnull().mean() < 0.05:
        df[col].fillna(df[col].mode()[0], inplace=True)
```

2.2 Tratamiento de Outliers (Z-score e IQR)

Fragmento de código:

```
# Identificación de outliers numéricos usando IQR
def detectar_outliers_iqr(data, columna):
    Q1 = data[columna].quantile(0.25)
    Q3 = data[columna].quantile(0.75)
    IQR = Q3 - Q1
    outliers = data[(data[columna] < Q1 - 1.5 * IQR) | (data[columna] > Q3 + 1.5 * IQR)]
    return outliers

# Si tienes columnas numéricas, puedes aplicarlo así:
# detectar_outliers_iqr(df, "columna_numérica")
```

Estrategia recomendada:

- Identificar, pero no eliminar automáticamente.
- Dejar lógica configurable en el pipeline.

2.3 Estandarización de Tipos y Categorías

Fragmento de código:

```
# Convertir columnas categóricas a minúsculas y sin espacios
cat_cols = df.select_dtypes(include='object').columns

for col in cat_cols:
    df[col] = df[col].astype(str).str.lower().str.strip()
```

Fragmento de código:

```
# Validación de tipos de datos
df.dtypes
```

2.4 Pipeline Automatizado- Clase `DataCleaner`

Fragmento de código:

```
class DataCleaner:
    def __init__(self, drop_cols=None):
        self.drop_cols = drop_cols if drop_cols else []
        self.fill_values = {}

Quito, Ecuador. Telf. (593) 095 861 0558 / 098 270 4137
E-mail: info@acclatam.lat
www.acclatam.lat/mmia
```



```
def fit(self, X, y=None):
    # Guardar valores para imputación
    for col in X.columns:
        if X[col].isnull().mean() > 0 and X[col].isnull().mean() < 0.05:
            self.fill_values[col] = X[col].mode()[0]
    return self

def transform(self, X):
    X_clean = X.copy()

    # Eliminar columnas innecesarias
    X_clean.drop(columns=self.drop_cols, errors="ignore", inplace=True)

    # Imputación
    for col, val in self.fill_values.items():
        X_clean[col] = X_clean[col].fillna(val)

    # Estandarización de texto
    for col in X_clean.select_dtypes(include="object").columns:
        X_clean[col] = X_clean[col].astype(str).str.lower().str.strip()

    return X_clean

def fit_transform(self, X, y=None):
    return self.fit(X, y).transform(X)
```

Fragmento de código:

```
# Ejemplo de uso
cleaner = DataCleaner()
df_limpio = cleaner.fit_transform(df)
df_limpio.head()
```

Fase 3: Feature Engineering Avanzado

3.1 Creación de Variables Derivadas

Fragmento de código:

```
import pandas as pd

# Cargar dataset original
df = pd.read_csv("dataset_etiquetas_cajas_mpls.csv")

# Frecuencia de ciudad
df["frecuencia_ciudad"] = df["ciudad"].map(df["ciudad"].value_counts())

# Longitud del código de caja
df["long_caja_código"] = df["caja_código"].astype(str).apply(len)

# Interacción ciudad + color_buffer
df["ciudad_color"] = df["ciudad"] + "_" + df["color_buffer"]

df[["frecuencia_ciudad", "long_caja_código", "ciudad_color"]].head()
```



3.2 Codificación de Variables Categóricas

Fragmento de código:

```
from sklearn.preprocessing import LabelEncoder

label_cols = ["ciudad", "nodo_concentrador", "nodo_estandar", "nodo_backup", "ruta"]
label_encoders = {}

for col in label_cols:
    le = LabelEncoder()
    df[col + "_enc"] = le.fit_transform(df[col])
    label_encoders[col] = le

# También codificamos color_buffer como variable objetivo
df["target"] = LabelEncoder().fit_transform(df["color_buffer"])

df.head()
```

3.3 Transformación de Variables Numéricas

Fragmento de código:

```
from sklearn.preprocessing import StandardScaler

scale_cols = ["frecuencia_ciudad", "long_calle_codigo"]
scaler = StandardScaler()
df[scale_cols] = scaler.fit_transform(df[scale_cols])

df[scale_cols].describe()
```

3.4 Selección de Características

Fragmento de código:

```
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
import seaborn as sns

# Preparar variables predictoras
X = df.select_dtypes(include=["int64", "float64"]).drop(columns=["target"])
y = df["target"]

# Validación de que X no esté vacío
if X.shape[1] == 0:
    raise ValueError("No se encontraron columnas numéricas para entrenar el modelo.")

# Entrenar modelo
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X, y)

# Importancia de características
importances = pd.Series(model.feature_importances_, index=X.columns)
importances.sort_values(ascending=False).plot(kind="bar", figsize=(12, 4),
                                              title="Importancia de Características")
plt.ylabel("Importancia")
plt.tight_layout()
plt.show()
```

Fase 4: Estrategias de Balanceamiento de Datos

4.1 Análisis de Desbalance

Fragmento de código:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Cargar datos procesados
df = pd.read_csv("dataset_etiquetas_cajas_mpls.csv")

# Ver distribución original
plt.figure(figsize=(10,5))
df["color_buffer"].value_counts(normalize=True).plot(kind="bar")
plt.title("Distribución original de la variable objetivo (color_buffer)")
plt.ylabel("Proporción")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

4.2 Oversampling con SMOTE

Fragmento de código:

```
from sklearn.preprocessing import LabelEncoder
from imblearn.over_sampling import SMOTE

# Codificar target
le = LabelEncoder()
df["color_enc"] = le.fit_transform(df["color_buffer"])
X = df.select_dtypes(include=["int64", "float64"]).drop(columns=["color_enc"])
y = df["color_enc"]

# Aplicar SMOTE
smote = SMOTE(random_state=42)
X_res, y_res = smote.fit_resample(X, y)

# Visualizar distribución resultante
import collections
counter = collections.Counter(y_res)
plt.bar(counter.keys(), counter.values())
plt.title("Distribución después de SMOTE")
plt.xlabel("Clase codificada")
plt.ylabel("Frecuencia")
plt.show()
```

4.3 Undersampling con Tomek Links

Fragmento de código:

```
from imblearn.under_sampling import TomekLinks

tl = TomekLinks()
X_tl, y_tl = tl.fit_resample(X, y)

# Visualización
```

```
counter_tl = collections.Counter(y_tl)
plt.bar(counter_tl.keys(), counter_tl.values())
plt.title("Distribución después de Tomek Links")
plt.xlabel("Clase codificada")
plt.ylabel("Frecuencia")
plt.show()
```

4.4 Técnicas Híbridas: SMOTETomek

Fragmento de código:

```
from imblearn.combine import SMOTETomek

smt = SMOTETomek(random_state=42)
X_smt, y_smt = smt.fit_resample(X, y)

counter_smt = collections.Counter(y_smt)
plt.bar(counter_smt.keys(), counter_smt.values())
plt.title("Distribución después de SMOTETomek")
plt.xlabel("Clase codificada")
plt.ylabel("Frecuencia")
plt.show()
```

4.5 Evaluación Comparativa

- ✚ SMOTE: Genera nuevas instancias sintéticas → útil si hay clases muy pequeñas.
- ✚ Tomek Links: Elimina ambigüedad entre clases → limpia bordes.
- ✚ SMOTETomek: Combina ambos efectos.
- ✚ A revisar: si los datos balanceados generan overfitting → validarlos con cross-validation en modelos futuros.

Fase 5: Data Augmentation

5.1 Introducción

Aunque la data augmentation es más común en visión por computador o texto, también puede aplicarse a datos tabulares para mejorar la generalización del modelo, especialmente cuando hay desbalance o baja representación de ciertas clases.

Técnicas aplicadas en este notebook:

- Agregación de ruido gaussiano
- SMOTE Variaciones
- Mixup sintético

5.2 Adición de Ruido Gaussiano

Fragmento de código:





```
import pandas as pd
import numpy as np

# Cargar dataset original
df = pd.read_csv("dataset_etiquetas_cajas_mpls.csv")

# Variable objetivo codificada
from sklearn.preprocessing import LabelEncoder
df["target"] = LabelEncoder().fit_transform(df["color_buffer"])

# Variables numéricas sobre las que aplicar ruido
df["frecuencia_ciudad"] = df["ciudad"].map(df["ciudad"].value_counts())
df["long_caja_código"] = df["caja_código"].astype(str).apply(len)

# Adición de ruido
df_aug = df.copy()
np.random.seed(42)
for col in ["frecuencia_ciudad", "long_caja_código"]:
    ruido = np.random.normal(loc=0.0, scale=0.1, size=len(df))
    df_aug[col + "_noisy"] = df[col] + ruido

df_aug[[col for col in df_aug.columns if "noisy" in col]].head()
```

5.3 Oversampling con SMOTE (revisión rápida)

Fragmento de código:

```
from sklearn.preprocessing import StandardScaler
from imblearn.over_sampling import SMOTE

# Variables numéricas que usaremos
X = df[["frecuencia_ciudad", "long_caja_código"]]
y = df["target"]

#  Imputar valores faltantes
X = X.fillna(X.median(numeric_only=True))

# Escalar
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Aplicar SMOTE
smote = SMOTE(random_state=42)
X_res, y_res = smote.fit_resample(X_scaled, y)

# Resultado
pd.Series(y_res).value_counts().sort_index()
```

5.4 Generación de Datos Sintéticos con Mixup

Fragmento de código:

```
# Generación simple de combinaciones de mezcla (sólo para demostración)
def mixup(x1, x2, alpha=0.2):
    l = np.random.beta(alpha, alpha)
    return l * x1 + (1 - l) * x2

mixup_df = pd.DataFrame(columns=X.columns)
mixup_labels = []
```



```
np.random.seed(42)
for i in range(100):
    idx1, idx2 = np.random.choice(len(X), 2, replace=False)
    x1, x2 = X.iloc[idx1], X.iloc[idx2]
    new_sample = mixup(x1.values, x2.values)
    mixup_df.loc[i] = new_sample
    mixup_labels.append(np.random.choice([y.iloc[idx1], y.iloc[idx2]]))

mixup_df["target"] = mixup_labels
mixup_df.head()
```

Fase 6: Partición Estratificada de Datos

6.1 División de Datos (Train / Validation / Test)

Fragmento de código:

```
import pandas as pd
from sklearn.model_selection import train_test_split

# Cargar dataset
df = pd.read_csv("dataset_etiquetas_cajas_mpls.csv")

# Codificar target
from sklearn.preprocessing import LabelEncoder
df["target"] = LabelEncoder().fit_transform(df["color_buffer"])

# Definir X e y
X = df.drop(columns=["color_buffer", "target"])
y = df["target"]

# División inicial en train (70%) y temp (30%)
X_train, X_temp, y_train, y_temp = train_test_split(X, y, stratify=y, test_size=0.30,
random_state=42)

# División de temp en validation (15%) y test (15%)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, stratify=y_temp,
test_size=0.50, random_state=42)

# Verificación de tamaños
print("Tamaño Train:", X_train.shape)
print("Tamaño Validation:", X_val.shape)
print("Tamaño Test:", X_test.shape)
```

6.2 Verificación de Distribución de Clases

Fragmento de código:

```
import matplotlib.pyplot as plt

def plot_distribution(y_sets, labels):
    fig, axs = plt.subplots(1, len(y_sets), figsize=(16, 4))
    for i, (y, label) in enumerate(zip(y_sets, labels)):
        y.value_counts(normalize=True).sort_index().plot(kind="bar", ax=axs[i])
        axs[i].set_title(f"{label} ({len(y)} muestras)")
        axs[i].set_xlabel("Clase")
```



```
        axs[i].set_ylabel("Proporción")
        plt.tight_layout()
        plt.show()

    plot_distribution(
        [y_train, y_val, y_test],
        ["Train", "Validation", "Test"]
    )
}
```

6.3 Validación de Integridad y Data Leakage

Fragmento de código:

```
# Revisar duplicados entre conjuntos
train_set = set(X_train.index)
val_set = set(X_val.index)
test_set = set(X_test.index)

intersect_val_train = train_set.intersection(val_set)
intersect_test_train = train_set.intersection(test_set)
intersect_val_test = val_set.intersection(test_set)

print("Duplicados Train/Val:", len(intersect_val_train))
print("Duplicados Train/Test:", len(intersect_test_train))
print("Duplicados Val/Test:", len(intersect_val_test))
```

Fase 7: Pipeline de Preprocesamiento Automatizado

7.1 Diseño General del Pipeline

Fragmento de código:

```
import pandas as pd
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.preprocessing import LabelEncoder, StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from imblearn.over_sampling import SMOTE

# Cargar datos
df = pd.read_csv("dataset_etiquetas_cajas_mpls.csv")
df[["target"]] = LabelEncoder().fit_transform(df[["color_buffer"]])
X = df.drop(columns=["color_buffer", "target"])
y = df["target"]
```

7.2 Componentes del Pipeline

Fragmento de código:

```
class BasicCleaner(BaseEstimator, TransformerMixin):
    def fit(self, X, y=None):
        return self

    def transform(self, X):
        X_ = X.copy()
        # Limpieza básica: lower y strip en strings
        for col in X_.select_dtypes(include='object').columns:
            Quito, Ecuador. Telf. (593) 095 861 0558 / 098 270 4137
            E-mail: info@acclatam.lat
            www.acclatam.lat/mmia
```



```
X_[col] = X_[col].astype(str).str.lower().str.strip()
return X_
```

Fragmento de código:

```
class SMOTEBalancer(BaseEstimator, TransformerMixin):
    def fit(self, X, y):
        self.smote = SMOTE(random_state=42)
        self.X_res, self.y_res = self.smote.fit_resample(X, y)
        return self

    def transform(self, X):
        return self.X_res
```

7.3 Construcción del Pipeline Final

Fragmento de código:

```
# Separar columnas por tipo
num_cols = ["frecuencia_ciudad", "long_caja_código"]
cat_cols = ["ciudad", "nodo_concentrador", "nodo_estandar", "nodo_backup", "ruta"]

# Column Transformer
preprocessor = ColumnTransformer(transformers=[
    ("num", Pipeline([
        ("imputer", SimpleImputer(strategy="median")),
        ("scaler", StandardScaler())
    ]), num_cols),
    ("cat", Pipeline([
        ("imputer", SimpleImputer(strategy="most_frequent")),
        ("encoder", OneHotEncoder(handle_unknown="ignore"))
    ]), cat_cols)
])

# Pipeline final
full_pipeline = Pipeline(steps=[
    ("cleaner", BasicCleaner()),
    ("preprocessor", preprocessor)
])
```

7.4 Aplicación del Pipeline y Validación

Fragmento de código:

```
# Crear columnas necesarias
df["frecuencia_ciudad"] = df["ciudad"].map(df["ciudad"].value_counts())
df["long_caja_código"] = df["caja_código"].astype(str).apply(len)

X = df[cat_cols + num_cols]
y = df["target"]

# Aplicar pipeline
X_proc = full_pipeline.fit_transform(X)

print("Shape después del pipeline:", X_proc.shape)
```



Próximos Pasos

- Entrenar modelos de IA con el dataset preprocesado.
- Evaluar modelos como Random Forest, SVM, Redes Neuronales, Transformers.
- Integrar el mejor modelo en un sistema automatizado de validación en campo con visión computacional.

8. Repositorio del Proyecto

Con el fin de garantizar la trazabilidad, reproducibilidad y acceso abierto al material desarrollado, se ha habilitado un repositorio estructurado que contiene:

- Cuaderno completo de la fase de preparación y procesamiento de datos (.ipynb)
- Informe técnico del proyecto (.pdf)
- Presentación ejecutiva (.pptx)
- Dataset utilizado en el entrenamiento (.csv)
- Scripts de procesamiento, limpieza y balanceo
- Código base del pipeline automatizado
- Referencias académicas y mockups del bot y dashboard

El repositorio se encuentra disponible en el siguiente enlace:

 <https://github.com/miandrade-acc/validacion-etiquetas-red>

Todos los materiales pueden ser descargados, replicados y utilizados para futuras fases del proyecto, como el entrenamiento del modelo y su despliegue en campo.

9. Conclusiones y Próximos pasos

El desarrollo del pipeline de preparación de datos se completó de manera satisfactoria, cumpliendo con las etapas fundamentales del proceso. En primer lugar, se realizó la exploración del dataset, seguida de la limpieza de datos y la aplicación de técnicas de ingeniería de características. Posteriormente, se ejecutaron procesos de balanceo de clases, partición de los datos y la construcción de un pipeline modular, lo que garantiza la trazabilidad y replicabilidad del procedimiento.

Como siguiente fase, se plantea el entrenamiento comparativo de distintos modelos de clasificación, con el objetivo de seleccionar el de mejor desempeño. Finalmente, se prevé la integración del modelo seleccionado en un sistema de validación automatizado en campo, que permitirá ofrecer retroalimentación en tiempo real y optimizar los procesos operativos.

