



UNIVERSIDAD NACIONAL DE INGENIERÍA
FACULTAD DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA



Proyecto de software:

” IntiNet: Análisis y Optimización de la Cobertura en el Perú”

Versión: 1.0

Fecha: 29 de noviembre del 2024

Por:

Corpus Ramos Lia Esther 20232681A

Euríbe Zambrano Sebastian Alexis 20231173B

Evangelista Aguedo, María Fernanda 20231286^a

Jaco Malpartida Jazmín Fiorella 20234507I

Silva Vasquez Angela Antonella 20234526C

Curso:

Programación Orientada a Objetos (BMA15)

Ciclo relativo:

Tercer ciclo

Profesor:

Tello Canchapoma, Yuri Oscar

ÍNDICE

Tabla de contenido

Introducción	i
Objetivos	ii
Antecedentes	1
Diagrama UML	2
Diseño del código y clases (POO)	11
Conclusiones	58
Referencias	59

INTRODUCCIÓN

En el mundo actual, la conectividad móvil se ha convertido en una necesidad esencial, tanto para actividades personales como para el desarrollo de sectores productivos. Sin embargo, persisten desafíos significativos relacionados con la calidad y cobertura de las redes móviles, especialmente en áreas con alta densidad poblacional o ubicaciones geográficas remotas. Este proyecto se enfoca en analizar y evaluar la cobertura de las redes móviles en diferentes niveles geográficos, con el objetivo de identificar áreas de mejora y proponer soluciones efectivas.

El sistema desarrollado combina herramientas avanzadas de análisis de datos y aprendizaje automático con una interfaz web interactiva. Utilizando datos reales sobre cobertura, estaciones base y población, el sistema cuantifica la cobertura móvil, visualiza la información mediante gráficos intuitivos y genera reportes detallados. Además, se ofrece a los usuarios la posibilidad de explorar datos en diferentes niveles (departamento, provincia, distrito y centro poblado), permitiendo una comprensión clara de la situación actual y futura de la conectividad en cada área.

El sistema está basado en programación orientada a objetos (POO) y está implementado en Python, lo que permite una estructura modular y escalable. Esto facilita la extensión y mantenimiento del sistema, al mismo tiempo que mejora la capacidad de manejar grandes volúmenes de datos y realizar análisis complejos de manera eficiente.

La solución incluye una interfaz basada en Flask que proporciona una experiencia amigable para los usuarios. Los gráficos interactivos, generados con Plotly, y las tablas dinámicas, permiten visualizar de manera clara la información clave, como la población cubierta, estaciones base necesarias y propuestas específicas para mejorar la cobertura móvil. Con esta herramienta, se busca contribuir al desarrollo de estrategias más eficientes para optimizar la calidad de las redes móviles y garantizar una conectividad más inclusiva.

OBJETIVOS

OBJETIVO PRINCIPAL:

- Determinar la calidad de la red móvil en los diferentes niveles territoriales del Perú (distritos, provincias y departamentos) utilizando bases de datos relacionadas con la cantidad de estaciones base y la velocidad de conexión, con el fin de identificar áreas críticas y proponer soluciones.

OBJETIVOS ESPECÍFICOS:

- Identificar la cantidad de estaciones base (2G, 3G, 4G y 5G) en cada región, así como su distribución geográfica, para determinar si son adecuadas para atender la población de cada lugar.
- Obtener la proporción de población no cubierta o con acceso limitado a servicios de red móvil, conociendo la población de aquella área.
- Calcular la cantidad de estaciones base necesarias para cubrir la demanda actual y proyectar las necesidades futuras en las áreas críticas
- Generar informes detallados con tablas, gráficos y mapas que representen la calidad de la red en las diferentes regiones del Perú, para facilitar la toma de decisiones de las autoridades o empresas responsables.
- Aplicar algoritmos de machine learning para predecir cambios en la calidad del servicio, considerando diversos factores.

ANTECEDENTES

Algunas capitales provinciales, distritales y pueblos de las zonas rurales en el Perú permanecen aislados y estancados en términos socioeconómicos y han estado tradicionalmente desamparados en cuanto a una presencia activa de los organismos del Estado y a la provisión de servicios de información. Sin embargo, muchas de estas localidades sobre todo las capitales de provincia están adquiriendo importancia debido a que articulan actividades económicas de un mercado interno crecientemente más activo en la producción para el intercambio de mercancías, ya que se convierten en lugares centrales donde se asientan de manera efectiva el poder local (municipal) y los representantes de los sectores públicos involucrados en el desarrollo rural. (Luis Andrés Montes Bazalar, 2013)

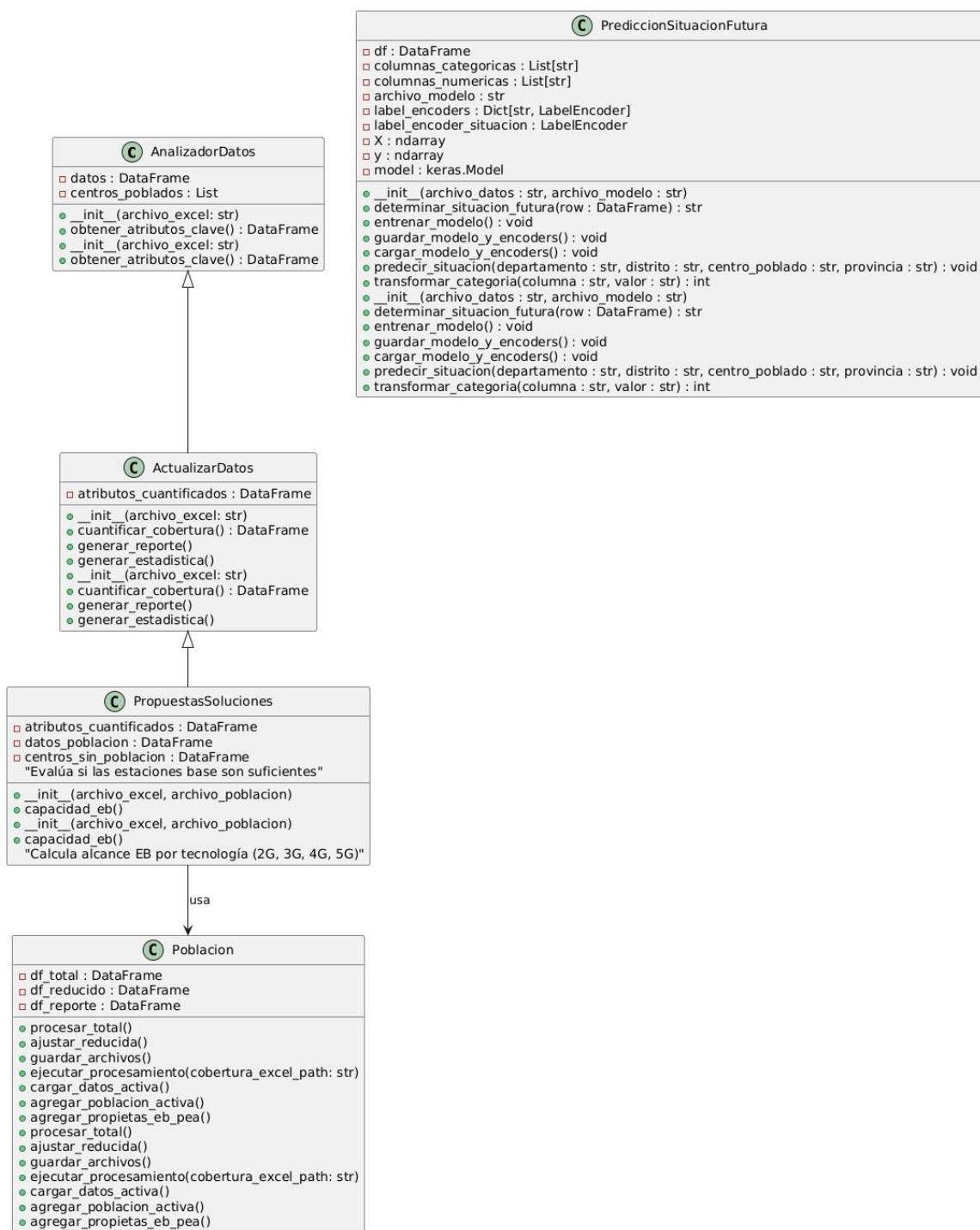
La creciente demanda de conectividad quedó evidenciada por el aumento masivo de dispositivos conectados a internet, que Cisco proyectó alcanzarían un tráfico de datos global de 278,1 Exabytes por mes en 2021, como resultado de la proliferación de dispositivos móviles de alto rendimiento y bajo costo (Bueno y Mejía, 2021, p. 11). En Perú, este fenómeno se intensificó durante la pandemia, cuando el Instituto Nacional de Estadística e Informática (INEI) reportó que el 87,7% de la población utilizó internet a través de dispositivos móviles, cifra que fue corroborada por OSIPTEL, que registró más de 27 millones de dispositivos móviles accediendo a internet entre enero y marzo de 2021 (Bueno y Mejía, 2021, p. 11).

En el Perú, el acceso a las tecnologías de la información y las telecomunicaciones (TIC) en áreas rurales enfrenta grandes desafíos debido al aislamiento geográfico y la falta de infraestructura adecuada. Montes (2013) destaca que las localidades rurales están tradicionalmente desamparadas en términos de acceso a servicios de información, lo que limita el desarrollo socioeconómico y el acceso a información crítica para la toma de decisiones, como precios, mercados e innovaciones técnicas (p. 1). Este aislamiento tecnológico se traduce en una “pobreza digital”, concepto que se refiere a la carencia de acceso a las TIC y a las habilidades necesarias para utilizarlas, lo cual afecta la capacidad de las comunidades rurales para participar en la economía digital (Montes, 2013, p. 4)

DIAGRAMA UML

Los diagramas UML (Unified Modeling Language) son herramientas visuales clave en la ingeniería de software, especialmente en proyectos complejos. En este proyecto, el uso de diagramas UML permite representar gráficamente los diferentes componentes, relaciones y flujos de datos que componen el sistema, facilitando su comprensión, desarrollo y mantenimiento. A continuación, se detallan algunas razones por las cuales es importante realizar diagramas UML para este proyecto:

Diagrama de Clases:



1. Clase Analizador Datos:

Esta clase se encarga de analizar los datos, representados por un DataFrame y una lista de centros poblados.

Sus métodos incluyen la inicialización con un archivo Excel y la obtención de atributos clave desde los datos.

2. Clase ActualizarDatos (hereda de AnalizadorDatos):

Es una clase base que extiende AnalizadorDatos y tiene métodos específicos para cuantificar la cobertura, generar reportes y estadísticas.

Cuantifica la cobertura de telecomunicaciones, generando estadísticas relevantes y reportes para el análisis.

3. Clase PropuestasSoluciones (hereda de ActualizarDatos):

Esta clase se especializa en generar soluciones para mejorar la cobertura móvil.

Usa datos poblacionales y centros sin población para determinar la capacidad de las estaciones base, calculando si son suficientes o si se necesita mejorar la infraestructura.

4. Clase Poblacion:

Esta clase está encargada del procesamiento de los datos poblacionales, tanto a nivel total como reducido.

Permite ejecutar el procesamiento de datos, cargar información activa sobre la población y agregar la población activa a las predicciones de cobertura.

5. Clase PrediccionSituacionFutura:

Esta clase se encarga de predecir la situación futura de la cobertura móvil utilizando un modelo de machine learning.

Tiene métodos para entrenar el modelo, predecir la situación en diferentes localidades, y transformar valores categóricos en valores numéricos para su posterior análisis.

Relaciones:

Herencia

- **AnalizadorDatos** es la clase base, y **ActualizarDatos** hereda de ella. Esto indica que **ActualizarDatos** extiende o especializa a **AnalizadorDatos**.
- **ActualizarDatos** es una clase base para **PropuestasSoluciones**, lo que significa que **PropuestasSoluciones** hereda métodos y atributos de **ActualizarDatos**.

Uso

- **PropuestasSoluciones** usa a **Poblacion**, lo que significa que una instancia de **PropuestasSoluciones** interactúa con objetos de la clase **Poblacion**.
- **PropuestasSoluciones** también usa a **PrediccionSituacionFutura**, ya que puede hacer uso de los resultados de predicción para evaluar las necesidades de cobertura.

Dependencia

- **PrediccionSituacionFutura** depende de **ActualizarDatos**, lo que indica que para predecir la situación futura, **PrediccionSituacionFutura** necesita la información proporcionada por **ActualizarDatos**.

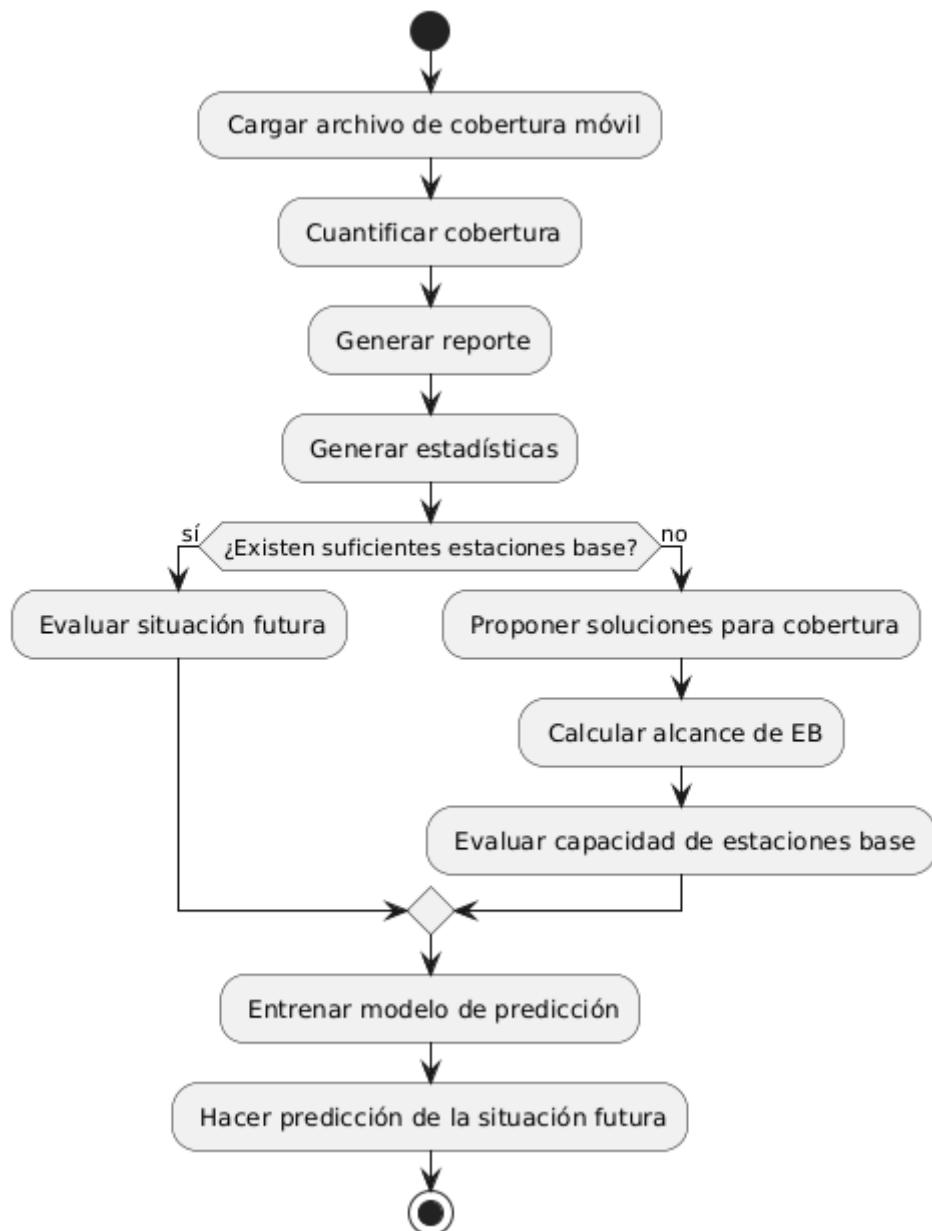
Composición

- **Poblacion** tiene una relación de **composición** con **DataFrame**, ya que los objetos de **DataFrame** son parte integral de la clase **Poblacion**. Si **Poblacion** se destruye, los **DataFrame** también lo harían.

Agregación

- **PropuestasSoluciones** tiene una relación de **agregación** con **Poblacion**, lo que significa que **PropuestasSoluciones** puede contener una instancia de **Poblacion** pero **Poblacion** puede existir independientemente.

Diagrama de Actividad: El diagrama de actividad te ayuda a visualizar cómo fluye el trabajo a través de las distintas actividades del proceso de análisis de cobertura y predicción. Muestra las tareas que se ejecutan, las decisiones que se toman en el camino, y cómo se manejan las bifurcaciones y las actividades concurrentes.



Flujo de actividades:

Inicio:

- Cargar archivo de cobertura móvil (archivo Excel).

Cuantificación de la cobertura:

- Cuantificar cobertura con los datos cargados.

Generación de reporte:

- Generar el reporte de cobertura basado en los datos cuantificados.
- Generar las estadísticas para análisis.

Decisión: ¿Existen suficientes estaciones base?

- Si **sí**, continuar con la evaluación de la situación futura.
- Si **no**, pasar a la propuesta de soluciones.

Propuestas de Soluciones (si las estaciones base son insuficientes):

- Calcular el alcance de las estaciones base para cada tecnología (2G, 3G, 4G, 5G).
- Evaluar si las estaciones base deben ser ajustadas o si es necesario agregar más estaciones.

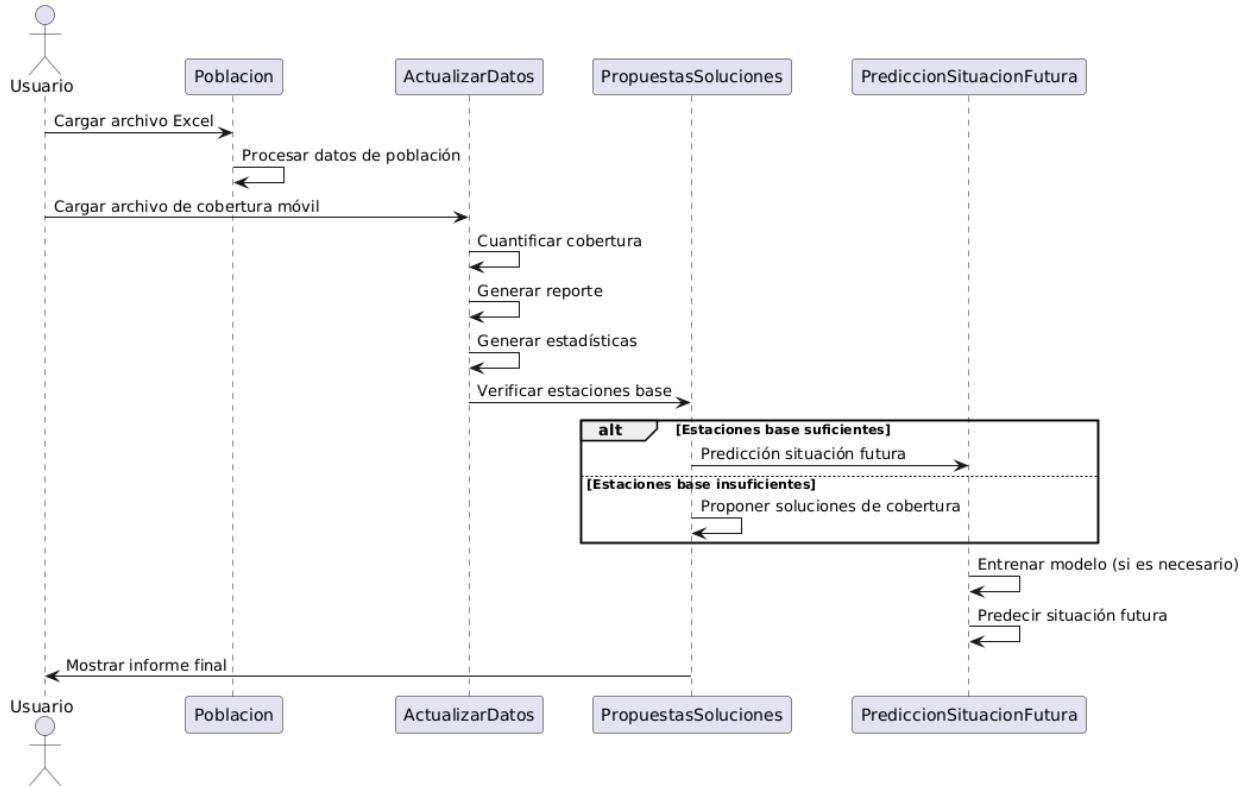
Predicción de la situación futura:

- Entrenar el modelo de predicción si es necesario (si no se ha entrenado previamente).
- Realizar la predicción de la situación para áreas específicas.

Fin:

- El proceso finaliza con la visualización de las predicciones y la generación del reporte final.

Diagrama de Secuencia: El diagrama de secuencia es útil porque muestra claramente cómo los distintos objetos del sistema interactúan entre sí en un proceso dinámico, permitiendo ver la secuencia temporal de los eventos y cómo las diferentes clases, como **ActualizarDatos**, **PropuestasSoluciones**, **Poblacion** y **PrediccionSituacionFutura**, trabajan juntas para completar el flujo de trabajo del sistema.



Explicación:

1. Inicio:

- El **Usuario** solicita que se cargue el archivo de datos de población a través de la clase **Poblacion**.
- **Poblacion** procesa los datos y luego el flujo pasa al siguiente paso.

2. Carga de cobertura móvil:

- El **Usuario** le solicita a **ActualizarDatos** que cargue el archivo de cobertura móvil.

- **ActualizarDatos** realiza la cuantificación de la cobertura, genera el reporte y las estadísticas correspondientes.

3. Evaluación de estaciones base:

- **PropuestasSoluciones** verifica si las estaciones base son suficientes. Si es así, pasa a la predicción futura. Si no, propone soluciones para mejorar la cobertura.

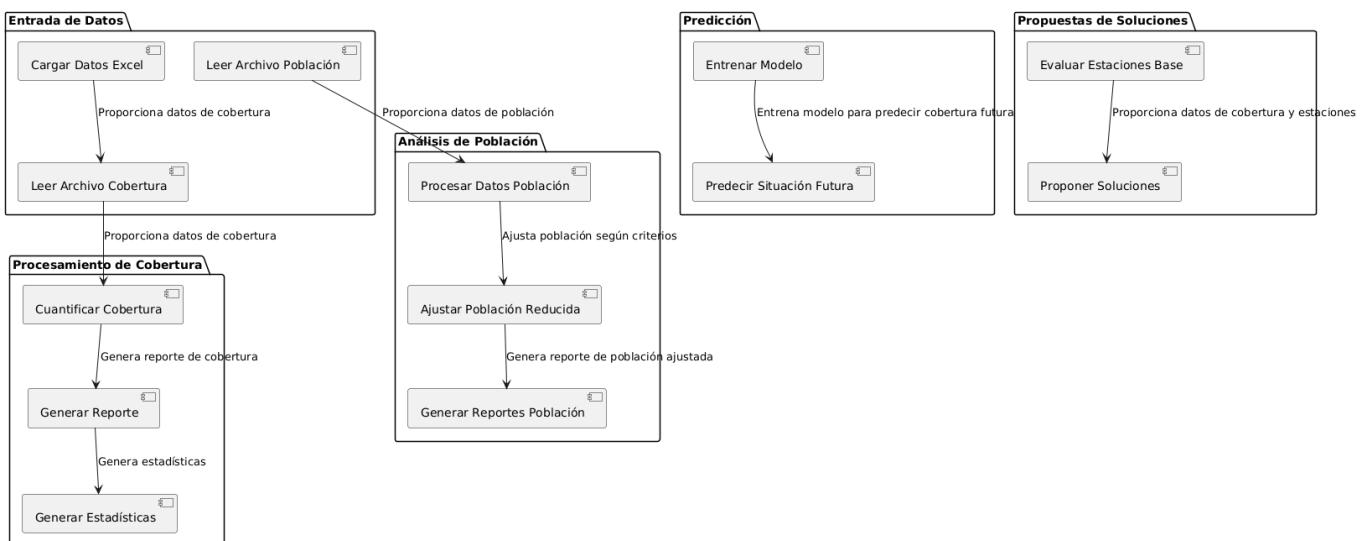
4. Predicción de situación futura:

- **PrediccionSituacionFutura** entrena el modelo si es necesario, y luego hace la predicción de la situación futura para las zonas seleccionadas.

5. Informe final:

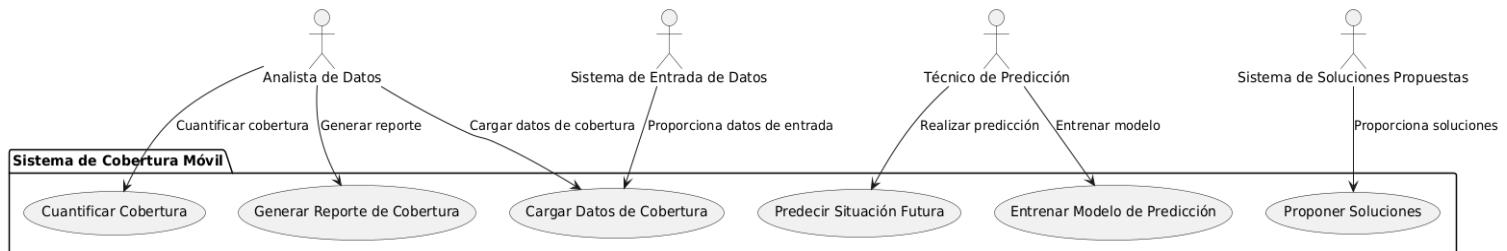
- **PropuestasSoluciones** y **PrediccionSituacionFutura** generan un informe final que se le muestra al **Usuario**.

Diagrama de Componentes: Este diagrama refleja cómo se organizan los distintos módulos, como los de entrada de datos, procesamiento de cobertura, análisis de población, predicción y soluciones propuestas, y cómo se interconectan para lograr el análisis de cobertura móvil y la predicción de la situación futura.



- **Entrada de Datos:** Este paquete agrupa los componentes relacionados con la carga de datos, tales como los archivos de cobertura móvil y población. Los componentes dentro de este paquete se encargan de leer y cargar los datos desde diferentes archivos Excel.
- **Procesamiento de Cobertura:** Aquí se encuentran los componentes encargados de realizar el análisis de los datos de cobertura, como la cuantificación de la cobertura, la generación de reportes y estadísticas.
- **Análisis de Población:** Este conjunto de componentes se encarga de procesar los datos de población, ajustarlos según criterios específicos y generar los reportes correspondientes.
- **Predicción:** En este componente se encuentran los elementos responsables de entrenar el modelo de Machine Learning y de predecir la situación futura de la cobertura móvil en las diferentes regiones.
- **Propuestas de Soluciones:** Este componente evalúa las estaciones base y propone soluciones para mejorar la cobertura si es necesario.

Diagrama de Casos de Uso: El diagrama de casos de uso permite una visión clara de las interacciones entre los actores y el sistema, facilitando la comprensión de las funcionalidades principales del sistema y cómo los usuarios interactúan con él para alcanzar sus objetivos. Este diagrama ayuda a visualizar cómo los diferentes actores (como los analistas y los técnicos) utilizan el sistema para realizar análisis de cobertura, entrenar modelos de predicción y proponer soluciones.



Explicación del Diagrama de Casos de Uso:

1. **Actores:**

- **Analista de Datos (AD):** Se encarga de cargar los datos de cobertura, cuantificar la cobertura y generar los reportes.
- **Técnico de Predicción (TP):** Interactúa con el sistema para entrenar el modelo de predicción y realizar las predicciones sobre la cobertura futura.
- **Sistema de Entrada de Datos (SED):** Proporciona los datos de cobertura y población que serán utilizados por el sistema para el análisis.
- **Sistema de Soluciones Propuestas (SSP):** Proporciona soluciones para mejorar la cobertura móvil.

2. Casos de Uso:

- **Cargar Datos de Cobertura (CDC):** Este caso de uso permite que el **Analista de Datos** cargue los datos de cobertura al sistema.
- **Cuantificar Cobertura (CC):** El sistema procesa los datos de cobertura y cuantifica la cobertura móvil, tarea realizada por el **Analista de Datos**.
- **Generar Reporte de Cobertura (GRC):** A partir de los datos procesados, el **Analista de Datos** genera un reporte detallado.
- **Entrenar Modelo de Predicción (EMP):** El **Técnico de Predicción** entrena el modelo de predicción utilizando los datos históricos.
- **Predecir Situación Futura (PSF):** El **Técnico de Predicción** hace una predicción sobre la cobertura futura usando el modelo entrenado.
- **Proponer Soluciones (PS):** El sistema propone soluciones a las deficiencias de cobertura con base en el análisis de las estaciones base y su capacidad.

Relaciones:

- "Cargar Datos de Cobertura" está asociado con "**Analista de Datos**", ya que el analista es quien carga los datos.

- "Cuantificar Cobertura" y "Generar Reporte de Cobertura" están asociados con "Analista de Datos" porque es su tarea realizar estos análisis.
- "Entrenar Modelo de Predicción" y "Predecir Situación Futura" están asociados con el "Técnico de Predicción" ya que este actor es quien realiza estas acciones.
- "Proponer Soluciones" está asociado con el "Sistema de Soluciones Propuestas", que proporciona las soluciones a los problemas de cobertura.

DISEÑO DEL CÓDIGO Y CLASES (POO)

El proyecto de software "**IntiNet: Análisis y Optimización de la Cobertura en el Perú**" emplea diversas librerías y herramientas de Python para el análisis y procesamiento de datos relacionados con la cobertura móvil. Cada una de estas librerías desempeña un papel clave en la gestión de datos, la creación de visualizaciones y la implementación de modelos de aprendizaje automático. A continuación, se detallan las principales bibliotecas utilizadas en el desarrollo del proyecto y su contribución al cumplimiento de los objetivos planteados.

1. Pandas (`import pandas as pd`):

Pandas es una de las bibliotecas más utilizadas en análisis de datos debido a su capacidad para gestionar grandes volúmenes de información estructurada. En este proyecto, se ha empleado para cargar, procesar y transformar los datos provenientes de archivos Excel, permitiendo realizar operaciones como filtrado y agrupación de información relevante sobre cobertura móvil.

2. OpenPyXL (`import openpyxl`):

OpenPyXL facilita la interacción directa con archivos Excel en formato .xlsx. Su uso ha sido crucial para leer y extraer datos específicos relacionados con la cobertura de tecnologías móviles, garantizando una integración fluida entre el análisis de datos y los documentos originales.

3. Matplotlib (`import matplotlib.pyplot as plt`):

La visualización de resultados es fundamental para interpretar los datos de manera efectiva. Matplotlib ha permitido la creación de gráficos que ilustran la cobertura móvil en diferentes regiones, facilitando así la identificación de patrones y áreas con necesidades de mejora.

4. OS (`import os`):

La biblioteca OS permite interactuar con el sistema operativo, lo que ha sido útil para gestionar rutas de archivos y automatizar la creación de directorios destinados al almacenamiento de resultados y gráficos generados por el sistema.

5. NumPy (`import numpy as np`):

NumPy es una herramienta eficiente para realizar cálculos matemáticos avanzados y manipulación de arreglos multidimensionales. En este proyecto, ha facilitado operaciones numéricas complejas necesarias para el análisis de grandes conjuntos de datos de cobertura móvil.

6. TensorFlow (`import tensorflow as tf`):

TensorFlow es una plataforma ampliamente utilizada en el campo del aprendizaje automático. Se ha empleado para desarrollar y entrenar modelos de redes neuronales que predicen patrones de cobertura y optimizan el despliegue de tecnologías móviles.

7. Keras Sequential (`from tensorflow.keras.models import Sequential`):

Keras, integrada en TensorFlow, permite construir modelos de redes neuronales de manera sencilla y modular mediante su API Sequential. Esta funcionalidad ha sido clave para definir la arquitectura de las redes neuronales utilizadas en el proyecto.

8. Keras Dense (`from tensorflow.keras.layers import Dense`):

La capa Dense de Keras es fundamental en las redes neuronales totalmente conectadas. Ha sido utilizada para añadir capas intermedias y de salida, permitiendo que el modelo aprenda relaciones complejas entre los datos de entrada y la cobertura móvil.

9. Train-Test Split (`from sklearn.model_selection import train_test_split`):

Una parte esencial del aprendizaje automático es dividir los datos en conjuntos de entrenamiento y prueba. Esta función garantiza que los modelos desarrollados sean evaluados de manera justa, evitando el sobreajuste y asegurando su capacidad para generalizar a nuevos datos.

10. StandardScaler (`from sklearn.preprocessing import StandardScaler`):

El proceso de normalización de datos es crucial cuando se trabaja con modelos de aprendizaje automático. StandardScaler se ha utilizado para estandarizar las características, asegurando que todas las variables tengan la misma escala y mejorando la eficiencia del modelo.

11. LabelEncoder (`from sklearn.preprocessing import LabelEncoder`):

LabelEncoder convierte datos categóricos en valores numéricos, un paso necesario para que los modelos de aprendizaje automático puedan procesar información no numérica. En este proyecto, ha sido utilizado para codificar nombres de departamentos y operadores móviles.

12. Unicodedata (`import unicodedata`)

La biblioteca estándar `unicodedata` en Python se utiliza para trabajar con caracteres Unicode y proporciona diversas funciones para acceder y manipular la base de datos de caracteres Unicode. Esta biblioteca es útil en tareas de procesamiento de texto, especialmente cuando necesitas trabajar con normalización, clasificación o eliminación de caracteres específicos en cadenas.

```
import pandas as pd # para hacer un DataFrame
import openpyxl # Para utilizar el excel deirectamente
import matplotlib.pyplot as plt # para graficar
import os #
import unicodedata
import numpy as np # para arreglos
import tensorflow as tf # Para el aprendizaje automatico yredes neuronales
from tensorflow.keras.models import Sequential #importa al modelo Secuencial
from tensorflow.keras.layers import Dense # Es para agregar capas de manera secuencial
from sklearn.model_selection import train_test_split # divide los datos en entrenamietno y prueba .-*
from sklearn.preprocessing import StandardScaler, LabelEncoder # las diferentes herramientas utilizadas
```

Ilustración 1: Librerías usadas

En el desarrollo de este proyecto, también se emplearán diversas clases para organizar y estructurar el análisis de los datos relacionados con la cobertura móvil en diferentes centros poblados. Estas clases están diseñadas para facilitar la carga, procesamiento y análisis de los datos de manera modular, permitiendo extender y modificar el comportamiento del sistema de forma eficiente.

Las clases que utilizaremos se basan en un enfoque orientado a objetos, lo que permite una mejor reutilización del código, mayor claridad en la organización y una fácil implementación de nuevas funcionalidades. A continuación, se describen las clases principales que componen el sistema:

1. Clase Analizador de Datos:

En el siguiente fragmento de código se describe la implementación de la clase `AnalizadorDatos`, diseñada para manejar y analizar datos de cobertura móvil contenidos en un archivo Excel.

```
## Iniciamos nuestra clase para analizar la cobertura a partir de nuestra base de datos
class AnalizadorDatos:
    def __init__(self, archivo_excel):
        # Cargar el archivo excel en un DataFrame
        self.datos = pd.read_excel(archivo_excel)
        self.centros_poblados = [] # Lista para almacenar los resultados

    # Iniciamos un nuevo método para obtener los atributos clave
    def obtener_atributos_clave(self):
        # Extraer las columnas clave del DataFrame
        atributos_clave = self.datos[['DEPARTAMENTO', 'PROVINCIA', 'DISTRITO', 'CENTRO_POBLADO', 'UBIGEO_CCPP', 'EMPRESA_OPERADORA', '2G', '3G',
        'HASTA_1_MBPS', 'MAS_DE_1_MBPS', 'CANT_EB_2G', 'CANT_EB_3G', 'CANT_EB_4G', 'CANT_EB_5G']]
        return atributos_clave
```

Ilustración 2: Clase Analizador de Datos

A continuación, se explica detalladamente cada parte de esta clase:

Constructor: __init__(self, archivo_excel)

El constructor se ejecuta cuando se crea una nueva instancia de la clase. Dentro de este constructor, se realiza lo siguiente:

a) Carga del archivo Excel: Se utiliza la función pd.read_excel (archivo_excel) para leer el archivo Excel proporcionado y cargar su contenido en un DataFrame . Un DataFrame es una estructura que organiza los datos en filas y columnas, permitiendo un manejo más fácil y eficiente. Esta estructura se guarda en el atributo self.datos.

b) Inicialización de la lista centros poblados: Se crea una lista vacía de llamadas self.centros_poblados. Esta lista estará destinada a almacenar los resultados que se generen más adelante durante el análisis de los centros poblados.

Método: obtener_atributos_clave(self)

Este método tiene la función de extraer las columnas más relevantes del DataFrame cargado. Se seleccionan únicamente las columnas que contienen la información clave para analizar la cobertura móvil:

a) Selección de las columnas relevantes: El método selecciona un conjunto de columnas del DataFrame, que incluye información sobre la ubicación (como DEPARTAMENTO, PROVINCIA, DISTRITO y CENTRO_POBLADO), datos sobre la empresa operadora de la red (EMPRESA_OPERADORA), así como la cobertura de las diferentes tecnologías móviles (2G, 3G, 4G, 5G) y las velocidades de conexión (por ejemplo, HASTA_1_MBPS y MÁS_DE_1_MBPS). También se incluyen columnas que indican la cantidad de estaciones bases por tipo de red (como CANT_EB_2G, CANT_EB_3G, etc.).

b) Retorno de los datos seleccionados: Una vez que se han extraído estas columnas, el método devuelve el DataFrame resultante, que contiene solo los datos relevantes para el análisis. Esto permite trabajar directamente con la información necesaria sin tener que filtrar constantemente.

2. Clase Actualizar Datos (Analizador de Datos):

Tenemos una segunda clase la cual hereda de la clase Analizador de Datos, la cual amplía sus capacidades para cuantificar la cobertura móvil y generar informes sobre los datos analizados.

Constructor: __init__(self, archivo_excel)

```
# Creamos una clase que herede la anterior
class ActualizarDatos(AnalizadorDatos):
    def __init__(self, archivo_excel):
        # Inicializamos la clase base
        super().__init__(archivo_excel)
        self.atributos_cuantificados = None # Atributo para almacenar los datos cuantificados
```

Ilustración 3: clase Actualizar Datos y Constructor

a) Inicialización de la Clase Base: Se invoca super().__init__(archivo_excel) para reutilizar el constructor de la clase AnalizadorDatos, lo que permite cargar el archivo Excel con los datos de cobertura móvil.

b) Definición de Atributo: Se crea self.atributos_cuantificados, una variable que almacenará los datos cuantificados tras aplicar los cálculos de cobertura.

Método: cuantificar_cobertura(self)

Este método calcula una calificación de cobertura basada en la disponibilidad de tecnologías móviles y el número de estaciones base (EB) en cada centro poblado.

```
# Método para cuantificar la cobertura de tecnologías 2G, 3G, 4G y 5G
def cuantificar_cobertura(self):
    # Obtenemos los atributos clave
    self.atributos_cuantificados = self.obtener_atributos_clave().copy()

    # Sumamos los atributos que presenta cada centro poblado
    # Los pesos que se asignan a cada conexión: 0 si no existe
    self.atributos_cuantificados[['CALIFICACION']] = (
        self.atributos_cuantificados['2G']*2 +
        self.atributos_cuantificados['3G']*3 +
        self.atributos_cuantificados['4G']*4 +
        self.atributos_cuantificados['5G']*5 +
        self.atributos_cuantificados['HASTA_1_MBPS']*0.5 +
        self.atributos_cuantificados['MAS_DE_1_MBPS']*1 +
        self.atributos_cuantificados['CANT_EB_2G'] +
        self.atributos_cuantificados['CANT_EB_3G'] +
        self.atributos_cuantificados['CANT_EB_4G'] +
        self.atributos_cuantificados['CANT_EB_5G']
    )
    self.atributos_cuantificados['CALIFICACION'] = (
        self.atributos_cuantificados['CALIFICACION'].apply(lambda x: int(round(x)))
    )

    # Hallamos la cantidad de eb total en cada CP
    self.atributos_cuantificados['CANT_EB_TOTAL'] = self.atributos_cuantificados['CANT_EB_2G'] + self.atributos_cuantificados['CANT_EB_3G'] + self.atributos_cuantificados['CANT_EB_4G'] + self.atributos_cuantificados['CANT_EB_5G']

    # Normalizamos los nombres de los departamentos para evitar errores
    self.atributos_cuantificados[['DEPARTAMENTO']] = (
        self.atributos_cuantificados[['DEPARTAMENTO']].str.upper().str.strip()
    )
    self.atributos_cuantificados[['PROVINCIA']] = (
        self.atributos_cuantificados[['PROVINCIA']].str.upper().str.strip()
    )
    self.atributos_cuantificados[['DISTRITO']] = (
        self.atributos_cuantificados[['DISTRITO']].str.upper().str.strip()
    )
    self.atributos_cuantificados[['CENTRO_Poblado']] = (
        self.atributos_cuantificados[['CENTRO_Poblado']].str.upper().str.strip()
    )

    # Retornamos los datos cuantificados
    return self.atributos_cuantificados
```

Ilustración 4: Método Cuantificar cobertura

a) Copia de Datos Clave: Se extraen los atributos necesarios (como DEPARTAMENTO, 2G, 3G, etc.) a partir del método obtener_atributos_clave() y se realiza una copia para no modificar los datos originales.

b) Calificación de Cobertura: Se suma la cobertura de cada tecnología y la cantidad de estaciones base, asignando distintos pesos:

- 2G, 3G, 4G, 5G (pesos: 2, 3, 4, 5).
- Velocidades de conexión: HASTA_1_MBPS (peso 0,5) y MÁS_DE_1_MBPS (peso 1).
- Se suman las estaciones base de cada tecnología.

```
self.atributos_cuantificados.loc[:, 'CALIFICACION'] = (
    self.atributos_cuantificados['2G']*2 +
    self.atributos_cuantificados['3G']*3 +
    self.atributos_cuantificados['4G']*4 +
    self.atributos_cuantificados['5G']*5 +
    self.atributos_cuantificados['HASTA_1_MBPS']*0.5 +
    self.atributos_cuantificados['MAS_DE_1_MBPS']*1 +
    self.atributos_cuantificados['CANT_EB_2G'] +
    self.atributos_cuantificados['CANT_EB_3G'] +
    self.atributos_cuantificados['CANT_EB_4G'] +
    self.atributos_cuantificados['CANT_EB_5G']
```

Ilustración 5: Calificación de Cobertura

c) Redondeo de Calificación: La calificación se redondea a un entero para facilitar su análisis.

```
self.atributos_cuantificados["CALIFICACION"] = (
    self.atributos_cuantificados["CALIFICACION"].apply(lambda x: int(round(x)))
```

Ilustración 6: Redondeo de Calificación

d)

Cálculo del Total de Estaciones Base: Se obtiene el número total de estaciones base sumando las de todas las tecnologías disponibles.

```
self.atributos_cuantificados["CANT_EB_TOTAL"] = self.atributos_cuantificados["CANT_EB_2G"] + self.atributos_cuantificados["CANT_EB_3G"] +
    self.atributos_cuantificados["CANT_EB_4G"] + self.atributos_cuantificados["CANT_EB_5G"]
```

Ilustración 7: Cálculo del Total de Estaciones Base

e) Normalización de Nombres: Se convierten los nombres de departamentos, provincias, distritos y centros poblados a mayúsculas y se eliminan espacios en blanco innecesarios para evitar inconsistencias.

```
self.atributos_cuantificados['DEPARTAMENTO'] = (
    self.atributos_cuantificados['DEPARTAMENTO'].str.upper().str.strip()
)
self.atributos_cuantificados['PROVINCIA'] = (
    self.atributos_cuantificados['PROVINCIA'].str.upper().str.strip()
)
self.atributos_cuantificados['DISTRITO'] = (
    self.atributos_cuantificados['DISTRITO'].str.upper().str.strip()
)
self.atributos_cuantificados['CENTRO_POBLADO'] = (
    self.atributos_cuantificados['CENTRO_POBLADO'].str.upper().str.strip()
)
```

Ilustración 8: Normalización de Nombres

f) Retorno de Datos Cuantificados: Se devuelve el DataFrame con las calificaciones y los datos procesados.

Método: generar_reporte(self)

Este método crea un archivo Excel con los datos cuantificados.

```
# Generar reporte de atributos_cuantificados
def generar_reporte(self):
    reporte_cobertura = "COBERTURA_MOVIL_CUANTIFICADA.xlsx"
    # Método para convertir el DataFrame a un archivo Excel
    self.atributos_cuantificados.to_excel(reporte_cobertura, index=False)
    print(f"Reporte generado: {reporte_cobertura}")
```

Ilustración 9: Método generar_reporte

- a) **Exportación a Excel:** Se utiliza to_excel() para guardar los datos cuantificados en un archivo llamado COBERTURA_MOVIL_CUANTIFICADA.xlsx.
- b) **Mensaje de Confirmación:** Se imprime un mensaje indicando que el reporte ha sido generado.

Método: generar_estadistica(self)

```
def generar_estadistica(self):  
    # Obtener estadísticas descriptivas  
    estadisticas = self.atributos_cuantificados['CALIFICACION'].describe()  
  
    # Convertir las estadísticas en un DataFrame  
    df_estadisticas = estadisticas.to_frame(name='Valor').reset_index()  
    df_estadisticas.columns = ['Estadística', 'Valor'] # Renombrar columnas para mayor claridad  
  
    # Guardar en un archivo Excel  
    reporte_estadistica = "ESTADISTICA_DESCRIPTIVA.xlsx"  
    df_estadisticas.to_excel(reporte_estadistica, index=False)  
    print(f"Estadística generada: {reporte_estadistica}")
```

Ilustración 10: Método generar_estadística

- a) **Obtención de Estadísticas:** Se utiliza describe() para calcular estadísticas como el promedio, mediana, mínimo y máximo de las calificaciones de cobertura.
- b) **Exportación a Excel:** Las estadísticas se exportan a un archivo llamado ESTADISTICA_DESCRIPTIVA.xlsx.
- c) **Mensaje de Confirmación:** Se muestra un mensaje indicando que el archivo ha sido generado.

3. Clase Generar Gráficas

La clase GenerarGraficas está diseñada para generar diversos tipos de gráficos a partir de los datos procesados por la clase ActualizarDatos. Estos gráficos ofrecen una representación visual de la cobertura móvil y el número de estaciones base por departamento y tecnología. A continuación, se describe cada método en detalle.

Constructor: __init__(self, actualizar_datos)

```
class GenerarGraficas:  
    def __init__(self, actualizar_datos):  
        self.actualizar_datos = actualizar_datos # Instancia de ActualizarDatos
```

Ilustración 10: Contructor de la Clase Generar gráficas

El constructor inicializa la clase GenerarGraficas recibiendo como parámetro una instancia de ActualizarDatos. Esto permite acceder a los datos cuantificados previamente.

a) Detalle:

self.actualizar_datos: Almacena la instancia de ActualizarDatos para utilizar los atributos y métodos relacionados con los datos procesados.

El constructor no realiza cálculos, solo prepara la instancia para utilizar los métodos gráficos.

Método: generar_histograma_calificacion(self)

```
def generar_histograma_calificacion(self):
    try:
        calificaciones = self.actualizar_datos.atributos_cuantificados['CALIFICACION']
        #Establece el tamaño de la figura
        plt.figure(figsize=(10, 6))
        #Genera histograma con 16 intervalos, color de fondo, borde y transparencia
        plt.hist(calificaciones, bins=16, color='skyblue', edgecolor='black', alpha=0.7)
        #Título y etiquetas para el eje x e y
        plt.title('Distribución de Calificaciones de Cobertura')
        plt.xlabel('Calificación')
        plt.ylabel('Frecuencia')
        #Cuadricula en el eje y con transparencia de 0.75
        plt.grid(axis='y', alpha=0.75)
        plt.savefig('calificaciones.png', bbox_inches='tight')
        plt.show()
    # En caso de que no exista la columna mencionada
    except KeyError:
        print("Error: No se han cuantificado los datos. Por favor, llama a 'cuantificar_cobertura()' primero.")
    # Cualquier otro tipo de error
    except Exception as e:
        print(f"Ocurrió un error inesperado: {e}")
```

Ilustración 11: Método generar_histograma_calificación

Este método genera un histograma que muestra la frecuencia de las calificaciones de cobertura móvil asignadas a los centros poblados.

a) Lógica interna:

Extracción de datos:

- **calificaciones = self.actualizar_datos.atributos_cuantificados['CALIFICACION']:**

Extrae la columna CALIFICACION del DataFrame.

Configuración del histograma:

- **plt.hist:** Crea el histograma con 16 intervalos (bins), color azul (color='skyblue') y bordes negros (edgecolor='black').
- **alpha=0.7:** Establece la transparencia de las barras.

Añadir títulos y etiquetas:

- **plt.title, plt.xlabel, plt.ylabel:** Configuran el título del gráfico y los nombres de los ejes.

Cuadrícula:

- **plt.grid:** Agregue una cuadrícula al eje y con transparencia (alpha=0.75).

Guardado y visualización:

- **plt.savefig:** Guarde el gráfico en un archivo llamado calificaciones.png.
- **plt.show:** Muestra el gráfico.

b) Manejo de errores:

KeyError: Ocurre si no se encuentra la columna CALIFICACION. Se muestra un mensaje indicando que primero se debe ejecutar el método cuantificar_cobertura().

Exception: Captura cualquier otro error inesperado y lo muestra.

Método: generar_gráfico_por_departamento(self)

```
def generar_gráfico_por_departamento(self):  
    # Agrupamos por departamento y calculamos la media de las calificaciones  
    promedio_departamentos = self.actualizar_datos.atributos_cuantificados.groupby('DEPARTAMENTO')[['CALIFICACION']].mean()  
  
    # Crear un gráfico de barras con los promedios de cada departamento  
    plt.figure(figsize=(10, 6)) # Tamaño de la imagen  
    barras = promedio_departamentos.plot(kind='bar', color='skyblue') # Color de las barras  
  
    # Añadir títulos y etiquetas  
    plt.title('Calificación Promedio por Departamento', fontsize=14)  
    plt.xlabel('Departamento', fontsize=12)  
    plt.ylabel('Calificación Promedio', fontsize=12)  
  
    # Rotar etiquetas de los departamentos  
    plt.xticks(rotation=90) # Rota la imagen 90 grados por estética  
  
    # Añadir las calificaciones sobre cada barra  
    for i, valor in enumerate(promedio_departamentos):  
        plt.text(i, valor + 0.05, f'{valor:.2f}', ha='center', va='bottom', fontsize=10, color='black')  
  
    # Ajustamos y mostramos el gráfico  
    plt.tight_layout()  
    plt.savefig('calificacion_distrito.png', bbox_inches='tight')  
    plt.show()  
    print("Gráfico de calificación promedio por departamento generado y mostrado.")  
except KeyError:  
    print("Error: No se han cuantificado los datos. Por favor, llama a 'cuantificar_cobertura()' primero.")  
except Exception as e:  
    print(f'Ocurrió un error inesperado: {e}')
```

Ilustración 12: Método generar_gráfico_por_departamento

Este método genera un gráfico de barras que muestra la calificación promedio de cobertura móvil para cada departamento.

a) Lógica interna:

Agrupación y cálculo de los medios:

- **groupby('DEPARTAMENTO')['CALIFICACION'].mean():** Agrupa los datos por DEPARTAMENTO y calcula la media de las calificaciones.

Creación del gráfico:

- **plt.figure(figsize=(10, 6)):** Define el tamaño del gráfico.

- **promedio_departamentos.plot(kind='bar')**: Genera el gráfico de barras con las calificaciones promedio.

Etiquetas y rotación:

- **plt.xlabel, plt.ylabel**: Establecen los nombres de los ejes.
- **plt.xticks(rotation=90)**: Rota las etiquetas del eje x para mejorar la legibilidad.

Añadir valores sobre las barras:

- **plt.text**: Coloca los valores sobre cada barra.

Guardado y visualización:

- **plt.savefig**: Mira el gráfico como calificacion_distrito.png.
- **plt.show**: Muestra el gráfico en pantalla.

b) Manejo de errores:

Similar al método anterior, manejo KeyError y otros errores generales.

Método: generar_eb_por_departamento(self)

```
def generar_eb_por_departamento(self):
    try:
        # Calculamos el total acumulado de estaciones base por departamento
        total_eb_dpt = self.actualizar_datos.atributos_cuantificados.groupby('DEPARTAMENTO')['CANT_EB_TOTAL'].sum()

        # Crear un gráfico de barras con los totales de estaciones base por departamento
        plt.figure(figsize=(10, 6)) # Tamaño de la imagen
        total_eb_dpt.plot(kind='bar', color='skyblue') # Color de las barras

        # Añadir títulos y etiquetas
        plt.title('Total de Estaciones Base por Departamento', fontsize=14)
        plt.xlabel('Departamento', fontsize=12)
        plt.ylabel('Total de Estaciones Base', fontsize=12)

        # Añadir las cantidades totales de EB sobre las barras
        for i, v in enumerate(total_eb_dpt):
            plt.text(i, v + 50, str(v), ha='center', fontsize=10) # Ajusta el valor 50 para posicionar el texto

        # Rotar etiquetas de los departamentos
        plt.xticks(rotation=90) # Rota las etiquetas de los departamentos

        # Ajustamos y mostramos el gráfico
        plt.tight_layout()
        plt.savefig('total_eb_departamento.png', bbox_inches='tight') # Guardar la imagen
        plt.show() # Mostrar el gráfico
        print("Gráfico de total de estaciones base por departamento generado y mostrado.")
    except KeyError:
        print("Error: No se han cuantificado los datos. Por favor, llama a 'cuantificar_cobertura()' primero.")
    except Exception as e:
        print(f"Ocurrió un error inesperado: {e}")
```

Ilustración 13: Método generar_eb_por_departamento

Este método crea un gráfico de barras que muestra el total acumulado de estaciones base (EB) en cada departamento del país. Sirve para identificar qué departamentos tienen mayor o menor cantidad de estaciones base, lo que es útil para evaluar la infraestructura de red.

a) Lógica interna:

Agrupación y suma de estaciones base: Agrupa los datos por DEPARTAMENTO y calcula la suma de estaciones base (CANT_EB_TOTAL) para cada uno.

- **groupby**: Agrupa los registros por el nombre del departamento.
- **sum()**: Calcula el total de estaciones base para cada grupo.

Configuración del gráfico:

- **plt.figure(figsize=(10, 6))**: Establece el tamaño de la figura (10 unidades de ancho por 6 de alto).
- **plot(kind='bar')**: Genera un gráfico de barras. El color de las barras se define como azul claro (skyblue).

Títulos y etiquetas:

- **plt.title**: Agrega un título descriptivo al gráfico.
- **plt.xlabel** y **plt.ylabel**: Establecen etiquetas para los ejes x(nombres de departamentos) e y(total de estaciones base).

Añadir etiquetas sobre las barras: Recorre cada barra y coloca el valor total (v) encima de ella.

- **plt.text**: Posiciona el texto en el centro horizontal (ha='center') y ligeramente por encima del valor (v + 50).

Rotación de etiquetas y guardados:

- **plt.xticks(rotation=90)**: Rota las etiquetas del eje x 90 grados para mejor legibilidad.
- **plt.tight_layout()**: Ajusta los márgenes para evitar que las etiquetas se corten.
- **plt.savefig**: Mira el gráfico como total_eb_departamento.png.
- **plt.show**: Muestra el gráfico en pantalla.

b) Manejo de errores:

KeyError: Si la columna CANT_EB_TOTAL no existe, se lanza un mensaje de error indicando que los datos no han sido cuantificados.

Exception: Captura otros errores y muestra el mensaje correspondiente.

Método: generar_grafico_pastel_eb_total(self)

```

def generar_grafico_pastel_eb_total(self):
    try:
        # Calculamos el total de estaciones base para cada tecnología
        total_2g = self.actualizar_datos.atributos_cuantificados['CANT_EB_2G'].sum()
        total_3g = self.actualizar_datos.atributos_cuantificados['CANT_EB_3G'].sum()
        total_4g = self.actualizar_datos.atributos_cuantificados['CANT_EB_4G'].sum()
        total_5g = self.actualizar_datos.atributos_cuantificados['CANT_EB_5G'].sum()

        # Lista con los totales por tecnología
        totales_eb = [total_2g, total_3g, total_4g, total_5g]
        tecnologias = ['2G', '3G', '4G', '5G']

        # Crear el gráfico de pastel con ajustes visuales
        plt.figure(figsize=(8, 6))
        wedges, texts, autotexts = plt.pie(totales_eb,
                                            labels=tecnologias,
                                            autopct='%.1f%%',
                                            startangle=140,
                                            colors=['#66b3ff', '#99ff99', '#ff9999', '#ffcc99'], # Colores similares al gráfico de barras
                                            wedgeprops={'edgecolor': 'black', 'linewidth': 1.5}, # Bordes definidos
                                            textprops={'fontsize': 12, 'color': 'black'})

        # Estilo del porcentaje
        for autotext in autotexts:
            autotext.set_fontsize(14)
            autotext.set_fontweight('bold')

        # Título del gráfico
        plt.title('Distribución de Estaciones Base por Tecnología en el País', fontsize=16, fontweight='bold', color='navy')

        # Asegura que el gráfico sea circular
        plt.axis('equal')
        plt.tight_layout()
        plt.savefig('eb_distribucion.png', bbox_inches='tight')
        plt.show()

    except KeyError:
        print("Error: No se han cuantificado los datos. Por favor, llama a 'cuantificar_cobertura()' primero.")
    except Exception as e:
        print(f"Ocurrió un error inesperado: {e}")

```

Ilustración 14: Método generar_grafico_pastel_eb_total

Este método crea un gráfico de pastel que muestra la distribución porcentual del total de estaciones base (EB) por tecnología (2G, 3G, 4G, 5G) a nivel nacional.

a) Lógica interna:

Suma de estaciones base por tecnología:

- Calcule el total de estaciones base para cada tecnología que utiliza sum().
- **CANT_EB_2G, CANT_EB_3G, CANT_EB_4G, CANT_EB_5G:** Columnas que almacenan el número de estaciones base para cada generación tecnológica.

Lista de datos y etiquetas:

- **totales_eb:** Lista que contiene los totales de estaciones base por tecnología.
- **tecnologias:** Etiquetas que describen cada segmento del gráfico de pastel.

Creación del gráfico de pastel:

- **plt.pie:** Crea el gráfico de pastel.
- **labels=tecnologias:** Asigna las etiquetas 2G, 3G, 4G, 5G a cada segmento.
- **autopct='%.1f%%':** Muestra el porcentaje con un decimal.
- **startangle=140:** Defina el ángulo de inicio para la disposición de los segmentos.

Configuración adicional y guardada:

- **plt.axis('equal')**: Asegúrese de que el gráfico tenga forma circular.

- **plt.savefig:** Mira el gráfico como eb_distribucion.png.
- **plt.show:** Muestra el gráfico en pantalla.

b) Manejo de errores:

KeyError: Si alguna de las columnas de estaciones base no está disponible, muestra un mensaje pidiendo cuantificar los datos.

Exception: Captura errores generales y muestra un mensaje con el detalle del problema.

4. Clase Propuestas Soluciones

Esta clase PropuestasSoluciones hereda de ActualizarDatosy amplía su funcionalidad integrando datos de población censada para cada centro poblado. Su propósito es fusionar la información de cobertura móvil con los datos poblacionales y generar informes útiles para identificar centros poblados sin datos de población o con problemas de cobertura. A continuación, se desglosa paso a paso el funcionamiento de la clase:

Constructor __init__(self, archivo_excel, archivo_poblacion)

```
Class PropuestasSoluciones(ActualizarDatos):
    def __init__(self, archivo_excel, archivo_poblacion):

        # Inicializamos la clase base
        super().__init__(archivo_excel)
        # Cargamos los datos de población
        self.datos_poblacion = pd.read_excel(archivo_poblacion)

        # Verificamos si los datos de población se han cargado correctamente
        if self.datos_poblacion is None or self.datos_poblacion.empty: # El empty nos dice que si el DataFrame esta vacío sera True
            raise ValueError("Error al cargar los datos de población: el DataFrame está vacío o es None.")

        # Seleccionamos las columnas que necesitamos
        self.datos_poblacion = self.datos_poblacion[["Departamento", "Provincia", "Distrito", "Centro Poblado", "Id Centro Poblado", "Población censada"]]

        # Renombrar columnas en el DataFrame de población
        self.datos_poblacion.rename(columns={
            "Departamento": "DEPARTAMENTO",
            "Provincia": "PROVINCIA",
            "Distrito": "DISTRITO",
            "Centro Poblado": "CENTRO_POBLADO",
            "Id Centro Poblado": "UBIGEO_CCPP",
            "Población censada": "POBLACION"
        }, inplace=True)
```

Ilustración 15: Clase Propuestas Soluciones (constructor 1)

```
# Asignar valor 0 si hay datos nulos en la columna "POBLACION"
self.datos_poblacion["POBLACION"] = self.datos_poblacion["POBLACION"].fillna(0)
self.atributos_cuantificados = pd.read_excel("COBERTURA_MOVIL CUANTIFICADA.xlsx") # lo que hace read_excel es leer el excel y transformarlo a un DataFrame
try:
    # Usamos la función merge para agregar la columna deseada
    self.atributos_cuantificados = pd.merge(
        self.atributos_cuantificados,
        self.datos_poblacion[["DEPARTAMENTO", "PROVINCIA", "DISTRITO", "CENTRO_POBLADO", "POBLACION"]],
        on=["DEPARTAMENTO", "PROVINCIA", "DISTRITO", "CENTRO_POBLADO"], # Especificar varias columnas como lista
        how="inner" # Esto indica que solo queremos filas donde haya coincidencias
    )
    reporte_cuantificado = 'REPORTE CUANTIFICADO.xlsx'
    self.atributos_cuantificados.to_excel(reporte_cuantificado, index=False)
    print(f"Estadística generada: {reporte_cuantificado}")
    if self.atributos_cuantificados.empty:
        print("Error: La fusión resultó en un DataFrame vacío.")
        return

    # Verificar que CCPN no tiene información de población
    self.centros_sin_poblacion = self.atributos_cuantificados[self.atributos_cuantificados["POBLACION"]==0]
    self.centros_sin_poblacion.to_excel("CCPN SIN INFORMACION.xlsx", index=False)

    # Ahora filtramos los datos para trabajar con los que tienen población
    self.atributos_cuantificados = self.atributos_cuantificados[self.atributos_cuantificados["POBLACION"] != 0]
except KeyError as e:
    # Esto indica si hay columnas faltantes
    print(f"Error: {e}")
except Exception as e:
    # Esto es para otro tipo de error
    print(f"Ocurrió un error inesperado: {e}")
```

Ilustración 16: Clase de Propuestas Soluciones (constructor 2)

El constructor inicializa la clase base, carga los datos de población y fusiona estos datos con los datos de cobertura móvil cuantificados.

archivo_excel: Ruta al archivo que contiene los datos de cobertura móvil.

archivo_poblacion: Ruta al archivo con los datos de población censada.

super().__init__(archivo_excel): Llama al constructor de la clase ActualizarDatos para cargar los datos de cobertura.

a) Verificación de la Carga de Datos

Validación de los datos de población: Si los datos no se cargan correctamente o el archivo está vacío, se lanza un error.

b) Selección y Renombrado de Columnas

El DataFrame datos_poblacion se filtra para seleccionar solo las columnas necesarias, y estas se renombran para que coincidan con los nombres utilizados en los datos de cobertura móvil.

Columnas seleccionadas: Departamento, Provincia, Distrito, Centro Poblado, Id Centro Poblado, Población censada.

Renombrado de columnas:

- Departamento a DEPARTAMENTO
- Provincia a PROVINCIA
- Distrito a DISTRITO
- Centro Poblado a CENTRO_POBLADO
- Id Centro Poblado a UBIGEO_CCPP
- Población censada a POBLACION

c) Manejo de Datos Nulos: Se reemplazan los valores nulos en la columna POBLACION con 0, para asegurar que todos los registros tengan un valor numérico válido.

d) Fusión de Datos Cuantificados: Se leen los datos de cobertura móvil desde un archivo Excel llamado COBERTURA_MOVIL CUANTIFICADA.xlsx y se fusionan con los datos de población.

- **pd.merge:** Fusiona los dos DataFrames utilizando las columnas DEPARTAMENTO, PROVINCIA, DISTRITO, y CENTRO_POBLADO.
- **how="inner":** Solo se mantienen las filas donde haya coincidencias en ambos DataFrames.

e) Generación del Informe Cuantificado: Una vez fusionados los datos, se guarda el DataFrame resultante en un archivo Excel llamado REPORTE_CUANTIFICADO.xlsx.

f) Identificación de Centros Poblados sin Población: Se filtran los centros poblados con población igual a 0 y se guardan en un archivo Excel llamado CCPP_SIN_INFORMACION.xlsx.

g) Filtrado de Centros con Población: Finalmente, se eliminan los registros con población igual a 0 para trabajar solo con los centros poblados que tienen información válida de población.

h) Manejo de errores: El código incluye mecanismos para manejar posibles errores:

KeyError: Si alguna columna necesaria falta durante la fusión se captura y se informa el error.

Exception: Cualquier otro error inesperado se captura y se imprime un mensaje con los detalles.

Método capacidad_eb_cp(self)

El método capacidad_eb_cp tiene como objetivo principal calcular indicadores clave sobre la capacidad de las Estaciones Base (EB) en cada centro poblado, generar propuestas de mejora y guardar los resultados en un archivo Excel. A continuación, se explica en detalle cada bloque de código:

Cálculo del Alcance de Estaciones Base (ALCANCE_EB)

```
def capacidad_eb_cp(self):
    # Calcular habitantes por antena en cada centro poblado
    self.atributos_cuantificados["ALCANCE_EB"] = (
        self.atributos_cuantificados['POBLACION'] / self.atributos_cuantificados['CANT_EB_TOTAL']
    ).replace([float('inf'), float('nan')], 0)

    self.atributos_cuantificados["ALCANCE_EB"] = (
        self.atributos_cuantificados["ALCANCE_EB"].apply(lambda x: int(round(x)))
    )
```

Ilustración 17: Cálculo del Alcance de Estaciones Base

En esta parte se va a calcular cuántos habitantes son atendidos por cada Estación Base (EB) en un centro poblado.

a) Detalles:

División poblacional: Se divide la población total (POBLACION) entre el número total de estaciones base (CANT_EB_TOTAL) en cada centro poblado.

Reemplazo de valores problemáticos: Si CANT_EB_TOTAL es así 0, la división resulta en inf(infinito). Este valor, junto con nan(datos faltantes), se reemplaza por 0.

b) Descripción:

Redondeo y conversión: El valor calculado de ALCANCE_EB se redondea al entero más cercano y se convierte en int.

Cálculo de la Población Cubierta (POBLACION_CUBIERTA)

```
self.atributos_cuantificados["POBLACION_CUBIERTA"] = (
    self.atributos_cuantificados['CANT_EB_TOTAL'] * 150
).replace([float('inf'), float('nan')], 0)

self.atributos_cuantificados["POBLACION_CUBIERTA"] = (
    self.atributos_cuantificados["POBLACION_CUBIERTA"].apply(lambda x: int(round(x)))
)
```

Ilustración 18: Cálculo de la Población Cubierta

Cada EB puede cubrir aproximadamente 150 habitantes, según la hipótesis establecida.

a) Detalles:

Multiplicación: Multiplica el número total de EB (CANT_EB_TOTAL) por 150 para calcular cuántas personas están cubiertas.

Manejo de errores: Si no hay EB, se asigna 0 para evitar valores indefinidos (inf o nan).

b) Descripción:

Redondeo y conversión: El valor calculado de POBLACION_CUBIERTA se redondea al entero más cercano y se convierte en int.

Cálculo de la Población No Cubierta (POBLACION_NO_CUBIERTA)

```
self.atributos_cuantificados["POBLACION_NO_CUBIERTA"] = (
    self.atributos_cuantificados['POBLACION'] - self.atributos_cuantificados["POBLACION_CUBIERTA"]
).replace([float('inf'), float('nan')], 0)
```

Ilustración 19: Cálculo de la Población No Cubierta

Resta de la población total la población ya cubierta por las EB.

a) Detalles:

Fórmula:

$$\text{POBLACION NO CUBIERTA} = \text{POBLACION} - \text{POBLACION_CUBIERTA}$$

Sustitución de errores: Si el resultado es indefinido, se reemplaza por 0.

Cálculo del Número de EB Necesarias (EB_NECESARIAS)

```

    self.atributos_cuantificados["EB_NECESARIAS"] = (
        self.atributos_cuantificados['POBLACION'] / 150
    ).replace([float('inf'), float('nan')], 0)

    self.atributos_cuantificados["EB_NECESARIAS"] = (
        self.atributos_cuantificados["EB_NECESARIAS"].apply(lambda x: int(round(x)))
    )

```

Ilustración 20: Cálculo del Número de EB Necesarias

5. Clase población:

Calcula cuántas EB serían necesarias para cubrir toda la población si cada EB atiende a 150 habitantes.

a) Detalles:

División:

$$\text{EB_NECESARIOS} = \text{POBLACION}/150$$

b) Descripción:

Redondeo y conversión: El valor calculado de EB_NECESARIAS se redondea al entero más cercano y se convierte en int.

Cálculo de Estaciones Base Faltantes (EB_FALTANTES)

```

    self.atributos_cuantificados["EB_FALTANTES"] = (
        self.atributos_cuantificados["EB_NECESARIAS"] - self.atributos_cuantificados["CANT_EB_TOTAL"]
    ).replace([float('inf'), float('nan')], 0)

```

Ilustración 21: Cálculo de Estaciones Base Faltantes

Calcula cuántas EB adicionales se necesitan instaladas en el centro poblado.

a) Detalles:

Fórmula:

$$\text{EB_FALTANTES} = \text{EB_NECESARIOS} - \text{CANT_EB_TOTAL}$$

Generación de Propuestas para Mejorar la Cobertura

```

for index, row in self.atributos_cuantificados.iterrows():
    if row["ALCANCE_EB"] <= 150:
        self.atributos_cuantificados.at[index, "PROPUESTA_EB"] = f"La cantidad de EB ({row['CANT_EB_TOTAL']}) es suficiente pero podemos aumentar las estaciones base para mejorar la capacidad en el centro poblado de {row['CENTRO_POBLADO']}, se necesitan por lo menos {row['EB_NECESARIAS']} en total para una cobertura óptima."
    else:
        self.atributos_cuantificados.at[index, "PROPUESTA_EB"] = f"La cantidad de estaciones base ({row['CANT_EB_TOTAL']}) es muy baja para la población en {row['CENTRO_POBLADO']}, debemos aumentar la cantidad de EB a {row['EB_NECESARIAS']} en total para una cobertura óptima."
    if row["CANT_EB_TOTAL"] == 0 and row["POBLACION"] != 0:
        self.atributos_cuantificados.at[index, "PROPUESTA_EB"] = f"La cantidad de EB ({row['CANT_EB_TOTAL']}) es nula, debemos aumentar las estaciones base a {row['EB_NECESARIAS']} para mejorar la capacidad en el centro poblado de {row['CENTRO_POBLADO']}."
    if row["CANT_EB_TOTAL"] == 0 and row["POBLACION"] == 0:
        self.atributos_cuantificados.at[index, "PROPUESTA_EB"] = f"Actualmente no se encuentran registros de población en {row['CENTRO_POBLADO']}."

# Imprimimos para verificar
print(self.atributos_cuantificados.head())
self.atributos_cuantificados.to_excel("REPORTE_CENTRO_POBLADO.xlsx", index=False)

# Abrir el archivo Excel previamente guardado con pandas utilizando openpyxl
wb = openpyxl.load_workbook("REPORTE_CENTRO_POBLADO.xlsx")
ws = wb.active # Seleccionar la hoja activa del archivo

```

Ilustración 22: Generación de Propuestas para Mejorar la Cobertura

a) Iteración sobre los datos cuantificados: Se utiliza un bucle for para recorrer cada fila del DataFrame self.atributos_cuantificados.

Index representa el índice de la fila actual, y row contiene los datos de esa fila como una serie de valores.

b) Propuesta basada en el alcance de las estaciones base

Condición: Si el número de habitantes atendidos por cada EB (ALCANCE_EB) es menor o igual a 150, se considera que la cobertura es suficiente, pero aún puede mejorarse.

Acción: Se añade una propuesta indicando que la cantidad actual de EB es adecuada, aunque se recomienda instalar más EB para mejorar la capacidad y alcanzar una cobertura óptima según la población total.

c) Propuesta para centros con cobertura insuficiente

Condición: Si ALCANCE_EB es mayor a 150, significa que la capacidad de las EB existentes es insuficiente para cubrir a la población.

Acción: Se genera una propuesta indicando que se debe incrementar el número de EB hasta la cantidad necesaria para cubrir toda la población de manera eficiente.

d) Propuesta para centros sin estaciones base pero con población

Condición: Si no hay estaciones base instaladas (CANT_EB_TOTAL es 0), pero el centro poblado tiene población, se detecta una carencia crítica de infraestructura.

Acción: La propuesta sugiere instalar el número necesario de EB para cubrir la totalidad de la población del centro.

e) Propuesta para centros sin datos de población ni EB

Condición: Si no hay estaciones base ni población registrada, se concluye que el centro poblado carece de datos relevantes.

Acción: Se documenta que no se dispone de información poblacional para dicho centro.

f) Guardado de resultados en un archivo Excel: El DataFrame actualizado se guarda en un archivo Excel llamado "REPORTE_CENTRO_POBLADO.xlsx".

Se imprime una vista previa de las primeras filas para verificar los resultados.

g) Ajuste del formato en el archivo Excel: Se carga el archivo Excel usando la biblioteca openpyxl.

ws selecciona la hoja activa del archivo para realizar ajustes de formato.

Ajuste del ancho de columnas en Excel

```
# Iterar sobre todas las columnas del archivo
for col in ws.columns:
    max_length = 0 # Inicializar variable para el largo m ximo de la celda en la columna
    column = col[0].column_letter # Obtener la letra de la columna (A, B, C, etc.)

    # Iterar sobre las celdas de la columna para encontrar la longitud m xima de contenido
    for cell in col:
        try:
            # Verificar si la longitud del contenido de la celda es mayor que el largo m ximo actual
            if len(str(cell.value)) > max_length:
                max_length = len(str(cell.value)) # Actualizar el largo m ximo
        except:
            pass # Ignorar celdas vac as o errores de tipo de datos

    # Ajustar el ancho de la columna bas ndose en el largo m ximo encontrado
    adjusted_width = (max_length + 2) # Se agrega un peque o margen de 2 para mejor visibilidad
    ws.column_dimensions[column].width = adjusted_width # Ajustar el tama o de la columna

# Guardar el archivo Excel con las columnas ajustadas
wb.save("REPORTE_CENTRO_POBLADO.xlsx")
wb.save("REPORTE.xlsx")
```

Ilustraci n 23: Ajuste del ancho de columnas en Excel

a) Recorrido por todas las columnas del archive: Se utiliza un bucle for para recorrer cada columna del archivo Excel.

- **ws.columns** contiene todas las columnas de la hoja activa del archivo.
- **max_length** es una variable que almacena la longitud m xima de los datos en la columna actual.
- **column** obtiene la letra de la columna para que posteriormente se ajuste su ancho.

b) C lculo de la longitud m xima en cada columna

- Se recorre cada celda dentro de la columna actual.

- Se convierte el valor de la celda en una cadena de texto (str(cell.value)) para calcular su longitud.
- Si la longitud del valor es mayor que max_length, se actualiza esta variable.

Control de errores: Si una celda está vacía o su contenido no se puede medir, se ignora con pass.

c) Ajuste del ancho de la columna: Una vez encontrada la longitud máxima, se le suma un pequeño margen de 2 caracteres para evitar que el contenido quede ajustado de manera exacta y permitir mayor visibilidad.

- `ws.column_dimensions[column].width` ajusta el ancho de la columna al valor calculado.

d) Guardado del archivo Excel: El archivo se guarda con el nombre "REPORTE_CENTRO_POBLADO.xlsx" y se crea una copia adicional como "REPORTE.xlsx". Este paso garantiza que los cambios en el formato se conserven y que el archivo esté listo para su uso.

Método capacidad_eb_distrito(self)

En esta parte del proceso, se genera un informe detallado sobre la capacidad de las estaciones base (EB) en cada distrito, basado en los datos proporcionados. El objetivo es analizar la cobertura de cada distrito, calcular las estaciones base necesarias para una cobertura óptima y generar un archivo Excel con la información procesada. A continuación, se detallan las etapas de este análisis.

Creación del DataFrame reporte_distrito

```
def capacidad_eb_distrito(self):
    # Creamos un nuevo dataframe
    self.reporte_distrito = self.atributos_cuantificados.copy()
    self.reporte_distrito["CALIFICACION"] = self.atributos_cuantificados.groupby('DISTRITO')[['CALIFICACION']].transform('mean')
    self.reporte_distrito["CALIFICACION"] = (
        self.reporte_distrito["CALIFICACION"].apply(lambda x: int(round(x)))
    )
    self.reporte_distrito["CANT_EB_2G"] = self.atributos_cuantificados.groupby("DISTRITO")["CANT_EB_2G"].transform("sum")
    self.reporte_distrito["CANT_EB_3G"] = self.atributos_cuantificados.groupby("DISTRITO")["CANT_EB_3G"].transform("sum")
    self.reporte_distrito["CANT_EB_4G"] = self.atributos_cuantificados.groupby("DISTRITO")["CANT_EB_4G"].transform("sum")
    self.reporte_distrito["CANT_EB_5G"] = self.atributos_cuantificados.groupby("DISTRITO")["CANT_EB_5G"].transform("sum")
    self.reporte_distrito["CANT_EB_TOTAL"] = self.atributos_cuantificados.groupby("DISTRITO")["CANT_EB_TOTAL"].transform("sum")
    self.reporte_distrito["POBLACION_TOTAL"] = self.atributos_cuantificados.groupby("DISTRITO")["POBLACION"].transform("sum")
    # Calculan habitantes por centro poblado
```

Ilustración 24: Creación del DataFrame reporte_distrito

Se inicia con la creación de un nuevo DataFramea partir de atributos_cuantificados, que contiene los datos necesarios para realizar los cálculos. Este DataFramese actualiza con varias columnas nuevas que reflejan las métricas clave, como la calificación media por distrito y el número total de estaciones base por tecnología (2G, 3G, 4G, 5G).

La calificación de cada distrito se calcula como el promedio de las calificaciones de todos los centros poblados dentro de dicho distrito. Además, se suman las estaciones base de cada tecnología por distrito.

Cálculo de Cobertura y Necesidades de Estaciones Base

```
self.reporte_distrito["ALCANCE_EB"] = (
    self.reporte_distrito['POBLACION_TOTAL'] /
    self.reporte_distrito['CANT_EB_TOTAL']
).replace([float('inf'), float('nan')], 0)

self.reporte_distrito["ALCANCE_EB"] = (
    self.reporte_distrito["ALCANCE_EB"].apply(lambda x: int(round(x)))
)

self.reporte_distrito["POBLACION_CUBIERTA"] = (
    self.reporte_distrito['CANT_EB_TOTAL'] * 150
).replace([float('inf'), float('nan')], 0)

self.reporte_distrito["POBLACION_CUBIERTA"] = (
    self.reporte_distrito["POBLACION_CUBIERTA"].apply(lambda x: int(round(x)))
)

self.reporte_distrito["POBLACION_NO_CUBIERTA"] = (
    self.reporte_distrito['POBLACION_TOTAL'] - self.reporte_distrito["POBLACION_CUBIERTA"]
).replace([float('inf'), float('nan')], 0)

self.reporte_distrito["EB_NECESARIAS"] = (
    self.reporte_distrito['POBLACION_TOTAL'] / 150
).replace([float('inf'), float('nan')], 0)

self.reporte_distrito["EB_NECESARIAS"] = (
    self.reporte_distrito["EB_NECESARIAS"].apply(lambda x: int(round(x)))
)

self.reporte_distrito["EB_FALTANTES"] = (
    self.reporte_distrito["EB_NECESARIAS"] - self.reporte_distrito["CANT_EB_TOTAL"]
).replace([float('inf'), float('nan')], 0)
```

Ilustración 25: Cálculo de Cobertura y Necesidades de Estaciones Bases

Una vez procesados los datos básicos, se calculan diversos indicadores cruciales:

- a) **Alcance de Estaciones Base (ALCANCE_EB):** Se calcula el número de habitantes por cada estación base. Si el valor es infinito o nulo, se reemplaza por 0 para evitar errores en los cálculos.
- b) **Población cubierta:** Se estima cuánta población está potencialmente cubierta por las estaciones base, asumiendo que cada estación cubre hasta 150 personas.
- c) **Población No Cubierta:** Se calcula como la diferencia entre la población total y la población cubierta.
- d) **Estaciones Base Necesarias (EB_NECESARIAS):** Se calcula cuántas estaciones base serían necesarias para cubrir adecuadamente a toda la población del distrito.

Eliminación de Columnas Irrelevantes

```
# Eliminar las columnas directamente
columnas_a_eliminar = [
    'CENTRO_POBLADO', 'UBIGEO_CCPP',
    'EMPRESA_OPERADORA', '2G', '3G', '4G', '5G', 'HASTA_1_MBPS', 'MAS_DE_1_MBPS', 'POBLACION', "PROPUESTA_EB"
]

# Eliminar las columnas del DataFrame
self.reporte_distrito.drop(columnas_a_eliminar, axis=1, inplace=True)
```

Ilustración 26: Eliminación de Columnas Irrelevantes

Para simplificar el informe y centrarse en los datos más relevantes, se eliminan las columnas que no contribuyen al análisis. Esto incluye información como el centro poblado, el código UBIGEO y los detalles sobre las tecnologías individuales (2G, 3G, 4G, 5G).

Generación de Propuestas de Estaciones Base

```
# Eliminar filas duplicadas basandose en todas las columnas excepto
self.reporte_distrito = self.reporte_distrito.drop_duplicates()
self.reporte_distrito["PROPIUESTA_EB"] = None
# Iterar sobre cada fila para evaluar si el centro poblado necesita mas estaciones base 4G
for index, row in self.reporte_distrito.iterrows():
    if row["POBLACION_TOTAL"] != 0:
        if row["ALCANCE_EB"] <= 150 :
            self.reporte_distrito.at[index, "PROPIUESTA_EB"] = f"La cantidad de EB ({row['CANT_EB_TOTAL']}) es suficiente pero podemos aumentar las estaciones base para mejorar la capacidad en el distrito de {row['DISTRITO']}, se necesitan por lo menos {row['EB_NECESARIAS']} en total para una cobertura óptima."
        elif row["ALCANCE_EB"] >= 150 :
            self.reporte_distrito.at[index, "PROPIUESTA_EB"] = f"La cantidad de estaciones base ({row['CANT_EB_TOTAL']}) es muy baja para la población en {row['DISTRITO']}, debemos aumentar la cantidad de EB a {row['EB_NECESARIAS']} en total para una cobertura óptima."
    if row["CANT_EB_TOTAL"] == 0:
        if row["POBLACION_TOTAL"] != 0:
            self.reporte_distrito.at[index, "PROPIUESTA_EB"] = f"La cantidad de EB ({row['CANT_EB_TOTAL']}) es nula, debemos aumentar las estaciones base a {row['EB_NECESARIAS']} para mejorar la capacidad en el distrito de {row['DISTRITO']}."
        else:
            self.reporte_distrito.at[index, "PROPIUESTA_EB"] = f"Actualmente no se encuentran registros de población en {row['DISTRITO']}."
```

Ilustración 27: Generación de Propuestas de Estaciones Bases

Se evalúa cada distrito para determinar si las estaciones base actuales son suficientes o si se requiere más infraestructura. Si el alcance de las estaciones base es bajo (es decir, más de 150 personas por antena), se propone aumentar la cantidad de estaciones base. En caso de que no haya estaciones base, se sugiere la implementación de nuevas.

Exportación del Reporte a Excel

```
# Imprimimos para verificar
print(self.reporte_distrito.head())
self.reporte_distrito.to_excel("REPORTE_DISTRITO.xlsx", index=False)

# Abrir el archivo Excel previamente guardado con pandas utilizando openpyxl
wb = openpyxl.load_workbook("REPORTE_DISTRITO.xlsx")
ws = wb.active # Seleccionar la hoja activa del archivo
```

Ilustración 28: Exportación del Reporte a Excel

Una vez procesados los datos, se guarda el informe en un archivo Excel, REPORTE_DISTRITO.xlsx para facilitar su distribución y análisis posterior.

Ajuste Automático del Ancho de las Columnas

```
# Iterar sobre todas las columnas del archivo
for col in ws.columns:
    max_length = 0 # Inicializar variable para el largo máximo de la celda en la columna
    column = col[0].column_letter # Obtener la letra de la columna (A, B, C, etc.)

    # Iterar sobre las celdas de la columna para encontrar la longitud máxima de contenido
    for cell in col:
        try:
            # Verificar si la longitud del contenido de la celda es mayor que el largo máximo actual
            if len(str(cell.value)) > max_length:
                max_length = len(str(cell.value)) # Actualizar el largo máximo
        except:
            pass # Ignorar celdas vacías o errores de tipo de datos

    # Ajustar el ancho de la columna basandose en el largo maximo encontrado
    adjusted_width = (max_length + 2) # Se agrega un pequeño margen de 2 para mejor visibilidad
    ws.column_dimensions[column].width = adjusted_width # Ajustar el tamaño de la columna
```

Para mejorar la presentación del archivo Excel, se ajustan automáticamente los anchos de las columnas para que se ajusten al contenido más largo. Esto garantiza que toda la información se muestre adecuadamente sin recortes o texto oculto.

Guardar los Cambios en el Archivo Excel

```
# Guardar el archivo Excel con las columnas ajustadas  
wb.save("REPORTE_DISTRITO.xlsx")
```

Ilustración 30: Guardar los Cambios en el Archivo Excel

Finalmente, se guarda el archivo Excel con las columnas ajustadas.

Método capacidad_eb_provincia(self)

El método capacidad_eb_provincia se utiliza para generar un informe sobre la cobertura de estaciones base (EB) a nivel provincial. A partir de los datos cuantificados previamente, calcula la capacidad actual de las antenas en cada provincia y propone mejoras basadas en la población total y la cantidad de estaciones base existentes.

```
def capacidad_eb_provincia(self):  
    # Creamos un nuevo dataframe  
    self.reporte_provincia = self.atributos_cuantificados.copy()  
    self.reporte_provincia["CALIFICACION"] = self.atributos_cuantificados.groupby("PROVINCIA")["CALIFICACION"].transform('mean')  
    self.reporte_provincia["CALIFICACION"] = (self.reporte_provincia["CALIFICACION"].apply(lambda x: int(round(x)))  
    )  
    self.reporte_provincia["CANT_EB_2G"] = self.atributos_cuantificados.groupby("PROVINCIA")["CANT_EB_2G"].transform("sum")  
    self.reporte_provincia["CANT_EB_3G"] = self.atributos_cuantificados.groupby("PROVINCIA")["CANT_EB_3G"].transform("sum")  
    self.reporte_provincia["CANT_EB_4G"] = self.atributos_cuantificados.groupby("PROVINCIA")["CANT_EB_4G"].transform("sum")  
    self.reporte_provincia["CANT_EB_5G"] = self.atributos_cuantificados.groupby("PROVINCIA")["CANT_EB_5G"].transform("sum")  
    self.reporte_provincia["CANT_EB_TOTAL"] = self.atributos_cuantificados.groupby("PROVINCIA")["CANT_EB_TOTAL"].transform("sum")  
    self.reporte_provincia["POBLACION_TOTAL"] = self.atributos_cuantificados.groupby("PROVINCIA")["POBLACION"].transform("sum")
```

Ilustración 31: Método capacidad_eb_provincia

a) Copia del DataFrame original:

Se crea una copia de atributos_cuantificados para trabajar sin alterar los datos originales.

b) Cálculo de la calificación media por provincia:

- Agrupa por PROVINCIA y calcula el promedio de la calificación.
- Redondea cada calificación al entero más cercano.

c) Suma de estaciones base por tecnología:

Agrupa por PROVINCIA y suma el número de estaciones base (EB) para cada tecnología (2G, 3G, 4G, 5G) y el total de EB.

d) Suma de población total por provincia:

Calcula la población total en cada provincia sumando las poblaciones de los centros poblados.

Cálculo de habitantes por estación base (ALCANCE_EB):

```
# Calcular habitantes por antena en cada centro poblado
self.reporte_provincia["ALCANCE_EB"] = (
    self.reporte_provincia['POBLACION_TOTAL'] /
    self.reporte_provincia['CANT_EB_TOTAL']
).replace([float('inf'), float('nan')], 0)

self.reporte_provincia["ALCANCE_EB"] = (
    self.reporte_provincia["ALCANCE_EB"].apply(lambda x: int(round(x)))
)
```

Ilustración 32: Cálculo de habitantes por estación base

a) Descripción:

Cálculo del alcance: Se divide la población total de cada provincia entre el número total de estaciones base (CANT_EB_TOTAL), dando como resultado la cantidad de habitantes que atiende cada EB.

Manejo de valores infinitos o nulos: Si no hay EB en una provincia (división por cero), los resultados infinitos (inf) o valores nan se reemplazan por 0.

Redondeo: Se redondea el resultado a un número entero para simplificar la interpretación.

Cálculo de la población cubierta por las estaciones base

```
self.reporte_provincia["POBLACION_CUBIERTA"] = (
    self.reporte_provincia['CANT_EB_TOTAL'] * 150
).replace([float('inf'), float('nan')], 0)

self.reporte_provincia["POBLACION_CUBIERTA"] = (
    self.reporte_provincia["POBLACION_CUBIERTA"].apply(lambda x: int(round(x)))
)
```

Ilustración 33: Cálculo de la población cubierta por las estaciones base

a) Descripción:

Estimación de la población cubierta: Cada estación base se considera capaz de cubrir 150 habitantes, por lo que se multiplica CANT_EB_TOTAL por 150 para obtener la población cubierta.

Manejo de valores infinitos o nulos: Al igual que en pasos anteriores, los valores infinitos y nulos se reemplazan por 0.

Redondeo: Se redondea el resultado a un número entero.

Cálculo de la población no cubierta

```

self.reporte_provincia["POBLACION_NO_CUBIERTA"] = (
    self.reporte_provincia["POBLACION_TOTAL"] - self.reporte_provincia["POBLACION_CUBIERTA"]
).replace([float('inf'), float('nan')], 0)

```

a) Descripción:

Se resta la población cubierta de la población total para calcular cuántos habitantes no están atendidos por las estaciones base existentes.

Cálculo de las estaciones base necesarias (EB_NECESARIAS)

```

self.reporte_provincia["EB_NECESARIAS"] = (
    self.reporte_provincia['POBLACION_TOTAL'] / 150
).replace([float('inf'), float('nan')], 0)

self.reporte_provincia["EB_NECESARIAS"] = (
    self.reporte_provincia["EB_NECESARIAS"].apply(lambda x: int(round(x)))
)

```

Ilustración 35: Cálculo de las estaciones base necesarias

a) Descripción:

Estimación de EB necesarias: La población total se divide entre 150 (habitantes por estación base) para calcular cuántas estaciones base se necesitan en total para una cobertura óptima.

Manejo de valores especiales: Se reemplazan valores infinitos y nulos 0 para evitar errores.

Redondeo: Se redondea el valor al entero más cercano.

Cálculo de las estaciones base faltantes (EB_FALTANTES)

```

self.reporte_provincia["EB_FALTANTES"] = (
    self.reporte_provincia["EB_NECESARIAS"] - self.reporte_provincia["CANT_EB_TOTAL"]
).replace([float('inf'), float('nan')], 0)

# Eliminar las columnas directamente
columnas_a_eliminar = [
    'DISTRITO', 'CENTRO_POBLADO', 'UBIGEO_CCPP',
    'EMPRESA_OPERADORA', '2G', '3G', '4G', '5G', 'HASTA_1_MBPS', 'MÁS_DE_1_MBPS', 'POBLACION', 'PROPIUESTA_EB'
]

```

Ilustración 36: Cálculo de las estaciones base faltantes

a) Descripción:

Se resta el número actual de EB (CANT_EB_TOTAL) del número total de EB necesario (EB_NECESARIAS).

Eliminación de columnas irrelevantes

```

# Eliminar las columnas directamente
columnas_a_eliminar = [
    'DISTRITO', 'CENTRO_POBLADO', 'UBIGEO_CCPP',
    'EMPRESA_OPERADORA', '2G', '3G', '4G', '5G', 'HASTA_1_MBPS', 'MAS_DE_1_MBPS', 'POBLACION', "PROPUESTA_EB"
]

# Eliminar las columnas del DataFrame
self.reporte_provincia.drop(columnas_a_eliminar, axis=1, inplace=True)

```

Ilustración 37: Eliminación de columnas irrelevantes

a) Descripción:

Se eliminan columnas que no son relevantes para el análisis a nivel de provincia, como datos específicos de los centros poblados (DISTRITO, CENTRO_POBLADO), detalles de las tecnologías por centro poblado, y propuestas específicas.

Eliminación de filas duplicadas y creación de columna de propuesta

```

# Eliminar filas duplicadas basandose en todas las columnas excepto
self.reporte_provincia = self.reporte_provincia.drop_duplicates()
self.reporte_provincia["PROPUESTA_EB"] = None

```

Ilustración 38: Eliminación de filas duplicadas y creación de columna de propuesta

Se eliminan filas duplicadas para evitar información repetida.

Se crea una columna PROPUESTA_EB para almacenar recomendaciones sobre estaciones base (EB).

Evaluación fila por fila

```

# Eliminar filas duplicadas basandose en todas las columnas excepto
self.reporte_provincia = self.reporte_provincia.drop_duplicates()
self.reporte_provincia["PROPUESTA_EB"] = None
# Iterar sobre cada fila para evaluar si el centro poblado necesita mas estaciones base
for index, row in self.reporte_provincia.iterrows():
    if row["POBLACION_TOTAL"] != 0:
        if row["ALCANCE_EB"] <= 150 :
            self.reporte_provincia.at[index, "PROPUESTA_EB"] = f"La cantidad de EB ({row['CANT_EB_TOTAL']}) es suficiente pero podemos aumentar las estaciones base para mejorar la capacidad en la provincia de {row['PROVINCIA']}, se necesitan por lo menos {row['EB_NECESARIAS']} en total para una cobertura óptima."
        elif row["ALCANCE_EB"] >= 150 :
            self.reporte_provincia.at[index, "PROPUESTA_EB"] = f"La cantidad de estaciones base ({row['CANT_EB_TOTAL']}) es muy baja para la población en {row['PROVINCIA']}, debemos aumentar la cantidad de EB a {row['EB_NECESARIAS']} en total para una cobertura óptima."

```

Ilustración 39: Evaluación fila por fila

Se analiza cada provincia usando un bucle for para determinar si se necesitan más estaciones base.

Población existente y cobertura suficiente (≤ 150): Sugiere aumentar EB para mejorar.

Población existente y cobertura insuficiente (> 150): Recomienda aumentar EB.

Provincias sin estaciones base

```

if row["CANT_EB_TOTAL"] == 0:
    if row["POBLACION_TOTAL"] != 0:
        self.reporte_provincia.at[index, "PROPUESTA_EB"] = f"La cantidad de EB ({row['CANT_EB_TOTAL']}) es nula, debemos aumentar las estaciones base a {row['EB_NECESARIAS']} para mejorar la capacidad en la provincia de {row['PROVINCIA']}."

```

Ilustración 40: Provincias sin estaciones base

Si no hay EB, pero sí población, se recomienda instalar EB necesario.

Provincias sin población registrada

```
else:  
    self.reporte_provincia.at[index, "PROPUESTA_EB"] = f"Actualmente no se encuentran registros de población en  
{row['PROVINCIA']}."
```

Ilustración 41: provincias sin población registrada

Si no hay población registrada, se informa que no hay datos.

Verificación del DataFrame

```
# Imprimimos para verificar  
print(self.reporte_provincia.head())  
self.reporte_provincia.to_excel("REPORTE_PROVINCIA.xlsx", index=False)  
  
# Abrir el archivo Excel previamente guardado con pandas utilizando openpyxl  
wb = openpyxl.load_workbook("REPORTE_PROVINCIA.xlsx")  
ws = wb.active # Seleccionar la hoja activa del archivo  
  
# Iterar sobre todas las columnas del archivo  
for col in ws.columns:  
    max_length = 0 # Inicializar variable para el largo máximo de la celda en la columna  
    column = col[0].column_letter # Obtener la letra de la columna (A, B, C, etc.)  
  
    # Iterar sobre las celdas de la columna para encontrar la longitud máxima de contenido  
    for cell in col:  
        try:  
            # Verificar si la longitud del contenido de la celda es mayor que el largo máximo actual  
            if len(str(cell.value)) > max_length:  
                max_length = len(str(cell.value)) # Actualizar el largo máximo  
        except:  
            pass # Ignorar celdas vacías o errores de tipo de datos  
  
    # Ajustar el ancho de la columna basándose en el largo máximo encontrado  
    adjusted_width = (max_length + 2) # Se agrega un pequeño margen de 2 para mejor visibilidad  
    ws.column_dimensions[column].width = adjusted_width # Ajustar el tamaño de la columna  
  
# Guardar el archivo Excel con las columnas ajustadas  
wb.save("REPORTE_PROVINCIA.xlsx")
```

Ilustración 42: Variación del Data Frame

Antes de exportar los datos, se realiza una inspección rápida para asegurarse de que la estructura y el contenido sean correctos. Esto se logra utilizando el método head() de pandas, que muestra las primeras filas del DataFrame reporte_provincia:

- print(self.reporte_provincia.head())

Esta verificación permite identificar posibles errores o inconsistencias antes de continuar con la exportación.

Exportación a Excel

El DataFrame `reporte_provincia` se guarda en un archivo llamado `REPORTE_PROVINCIA.xlsx` mediante el método `to_excel()` de pandas. La opción `index=False` se utiliza para omitir la columna de índices en el archivo final.

Carga del Archivo Excel

Para mejorar la legibilidad del informe, se utiliza la biblioteca `openpyxl` para abrir el archivo Excel generado y realizar ajustes en el ancho de las columnas.

Ajuste Automático del Ancho de Columnas

El siguiente bloque de código recorre todas las columnas de la hoja activa y ajusta automáticamente el ancho de cada columna en función del contenido más largo encontrado. Se agrega un pequeño margen para una mejor visibilidad:

- **Iteración sobre las columnas:** Se recorren todas las celdas de cada columna.
- **Cálculo del ancho máximo:** Se determina la longitud máxima del contenido de las celdas en cada columna.
- **Ajuste del ancho:** Se aplica un ajuste adicional de 2 caracteres para mejorar la presentación.

Guardado del Archivo Ajustado

Finalmente, los cambios realizados en el ancho de las columnas se guardan en el archivo original:
`wb.save("REPORTE_PROVINCIA.xlsx")`

Método `capacidad_eb_departamento(self)`

El método `capacidad_eb_departamento` tiene como objetivo generar un informe detallado sobre la cobertura móvil por departamento, calculando en la cantidad de estaciones base (EB) disponibles y su relación con la población. A continuación, se explica paso a paso cómo funciona este proceso.

```
def capacidad_eb_departamento(self):
    # Creamos un nuevo dataframe
    self.reporte_departamento = self.atributos_cuantificados.copy()
    self.reporte_departamento["CALIFICACION"] = self.atributos_cuantificados.groupby('DEPARTAMENTO')[['CALIFICACION']].transform('mean')
    self.reporte_departamento["CALIFICACION"] = (
        self.reporte_departamento["CALIFICACION"].apply(lambda x: int(round(x)))
    )
    self.reporte_departamento["CANT_EB_2G"] = self.atributos_cuantificados.groupby("DEPARTAMENTO")["CANT_EB_2G"].transform("sum")
    self.reporte_departamento["CANT_EB_3G"] = self.atributos_cuantificados.groupby("DEPARTAMENTO")["CANT_EB_3G"].transform("sum")
    self.reporte_departamento["CANT_EB_4G"] = self.atributos_cuantificados.groupby("DEPARTAMENTO")["CANT_EB_4G"].transform("sum")
    self.reporte_departamento["CANT_EB_5G"] = self.atributos_cuantificados.groupby("DEPARTAMENTO")["CANT_EB_5G"].transform("sum")
    self.reporte_departamento["CANT_EB_TOTAL"] = self.atributos_cuantificados.groupby("DEPARTAMENTO")["CANT_EB_TOTAL"].transform("sum")
    self.reporte_departamento["POBLACION_TOTAL"] = self.atributos_cuantificados.groupby("DEPARTAMENTO")["POBLACION"].transform("sum")
```

Ilustración 43: Método `capacidad_eb_departamento`

Creación del DataFrame: Se comienza con la creación de un nuevo DataFrame a partir de los datos previamente procesados (atributos cuantificados). Este DataFrame se utiliza para realizar los cálculos y generar el informe.

Cálculo de la calificación por departamento: Para cada departamento, se calcula la calificación promedio, que es redondeada a un número entero. Esta calificación refleja la calidad general de la cobertura en cada departamento.

Suma de estaciones base por tipo (2G, 3G, 4G, 5G): Para cada departamento, se calcula la cantidad total de estaciones base disponibles para cada tipo de tecnología (2G, 3G, 4G, 5G) utilizando la función transform("sum").

Cálculo de Cobertura y Necesidades de Estaciones Base

```
# Calcular habitantes por antena en cada centro poblado
self.reporte_departamento["ALCANCE_EB"] = (
    self.reporte_departamento["POBLACION_TOTAL"] /
    self.reporte_departamento["CANT_EB_TOTAL"]
).replace([float('inf'), float('nan')], 0)

self.reporte_departamento["ALCANCE_EB"] = (
    self.reporte_departamento["ALCANCE_EB"].apply(lambda x: int(round(x)))
)

self.reporte_departamento["POBLACION_CUBIERTA"] = (
    self.reporte_departamento["CANT_EB_TOTAL"] * 150
).replace([float('inf'), float('nan')], 0)

self.reporte_departamento["POBLACION_CUBIERTA"] = (
    self.reporte_departamento["POBLACION_CUBIERTA"].apply(lambda x: int(round(x)))
)

self.reporte_departamento["POBLACION_NO_CUBIERTA"] = (
    self.reporte_departamento["POBLACION_TOTAL"] - self.reporte_departamento["POBLACION_CUBIERTA"]
).replace([float('inf'), float('nan')], 0)

self.reporte_departamento["EB_NECESSARIAS"] = (
    self.reporte_departamento["POBLACION_TOTAL"] / 150
).replace([float('inf'), float('nan')], 0)

self.reporte_departamento["EB_NECESSARIAS"] = (
    self.reporte_departamento["EB_NECESSARIAS"].apply(lambda x: int(round(x)))
)

self.reporte_departamento["EB_FALTANTES"] = (
    self.reporte_departamento["EB_NECESSARIAS"] - self.reporte_departamento["CANT_EB_TOTAL"]
).replace([float('inf'), float('nan')], 0)
```

Ilustración 44: Cálculo de Cobertura y Necesidades de Estaciones Base (departamentos)

a) Cálculo del alcance de las estaciones base: Se calcula el número de habitantes cubiertos por cada estación base en cada departamento. El alcance de cada estación base se obtiene dividiendo la población total entre la cantidad total de estaciones base disponibles.

b) Cálculo de la población cubierta y no cubierta: Se estima la cantidad de población cubierta por las estaciones base, asumiendo que cada estación base cubre a 150 personas. Luego, se calcula la población no cubierta restando la población cubierta de la población total.

c) Cálculo de estaciones base necesarias: Para cada departamento, se calcula el número de estaciones base necesarias para cubrir adecuadamente a la población, considerando nuevamente que cada estación base cubre 150 personas.

d) Determinación de estaciones base faltantes: Se calcula la diferencia entre las estaciones base necesarias y las disponibles, identificando así cuántas más se requieren para mejorar la cobertura.

Eliminación de columnas innecesarias:

```

# Eliminar las columnas directamente
columnas_a_eliminar = [
    'PROVINCIA', 'DISTRITO', 'CENTRO_POBLADO', 'UBIGEO_CCPP',
    'EMPRESA_OPERADORA', '2G', '3G', '4G', '5G', 'HASTA_1_MBPS', 'MAS_DE_1_MBPS', 'POBLACION', "PROPUESTA_EB"
]

# Eliminar las columnas del DataFrame
self.reporte_departamento.drop(columnas_a_eliminar, axis=1, inplace=True)

```

Ilustración 45: Eliminación de columnas innecesarias

Para centrarse únicamente en los datos relevantes, se eliminan las columnas que no son necesarias para el análisis del informe, como información específica de la provincia, distrito o centro poblado.

Generación de propuesta de mejora:

```

# Eliminar filas duplicadas basandose en todas las columnas excepto 'POBLACION' y 'PROPUESTA_EB'
self.reporte_departamento = self.reporte_departamento.drop_duplicates()
self.reporte_departamento["PROPUESTA_EB"] = None
# Iterar sobre cada fila para evaluar si el centro poblado necesita mas estaciones base 4G
for index, row in self.reporte_departamento.iterrows():
    if row["POBLACION_TOTAL"] != 0:
        if row["ALCANCE_EB"] <= 150 :
            self.reporte_departamento.at[index, "PROPUESTA_EB"] = f"La cantidad de EB ({row['CANT_EB_TOTAL']}) es suficiente pero podemos aumentar las estaciones base para mejorar la capacidad en el departamento de {row['DEPARTAMENTO']}, se necesitan por lo menos {row['EB_NECESSARIAS']} en total para una cobertura óptima."
        elif row["ALCANCE_EB"] >= 150 :
            self.reporte_departamento.at[index, "PROPUESTA_EB"] = f"La cantidad de estaciones base ({row['CANT_EB_TOTAL']}) es muy baja para la población en {row['DEPARTAMENTO']}, debemos aumentar la cantidad de EB a {row['EB_NECESSARIAS']} en total para una cobertura óptima."
    if row["CANT_EB_TOTAL"] == 0:
        if row["POBLACION_TOTAL"] != 0:
            self.reporte_departamento.at[index, "PROPUESTA_EB"] = f"La cantidad de EB ({row['CANT_EB_TOTAL']}) es nula, debemos aumentar las estaciones base a {row['EB_NECESSARIAS']} para mejorar la capacidad en el departamento de {row['DEPARTAMENTO']}."
        else:
            self.reporte_departamento.at[index, "PROPUESTA_EB"] = f"Actualmente no se encuentran registros de población en {row['DEPARTAMENTO']}."

```

Se evalúa, para cada departamento, si la cantidad de estaciones base disponibles es suficiente o si es necesario aumentar su cantidad para asegurar una cobertura adecuada. Si las estaciones base son insuficientes, se sugiere la cantidad de estaciones base necesarias para alcanzar una cobertura óptima.

Iterar las columnas:

```

# Iterar sobre todas las columnas del archivo
for col in ws.columns:
    max_length = 0 # Inicializar variable para el largo máximo de la celda en la columna
    column = col[0].column_letter # Obtener la letra de la columna (A, B, C, etc.)

    # Iterar sobre las celdas de la columna para encontrar la longitud máxima de contenido
    for cell in col:
        try:
            # Verificar si la longitud del contenido de la celda es mayor que el largo máximo actual
            if len(str(cell.value)) > max_length:
                max_length = len(str(cell.value)) # Actualizar el largo máximo
        except:
            pass # Ignorar celdas vacías o errores de tipo de datos

    # Ajustar el ancho de la columna basandose en el largo máximo encontrado
    adjusted_width = (max_length + 2) # Se agrega un pequeño margen de 2 para mejor visibilidad
    ws.column_dimensions[column].width = adjusted_width # Ajustar el tamaño de la columna

```

Ilustración 46: Iterar las columnas

Para optimizar la presentación del archivo Excel, se ajustan automáticamente los anchos de las columnas según el contenido más extenso. De esta manera, se asegura que toda la información sea visible de forma clara, evitando recortes o texto oculto.

Creación del archivo Excel con los resultados:

```

# Guardar el archivo Excel con las columnas ajustadas
wb.save("REPORTE_DEPARTAMENTO.xlsx")

```

Ilustración 47: Creación del archivo Excel con los resultados

Finalmente, el informe se guarda en un archivo Excel llamado REPORTE_DEPARTAMENTO.xlsx. Además, se ajustan automáticamente los anchos de las columnas para mejorar la presentación visual del archivo, de forma que todo el contenido sea fácilmente legible.

5. Clase población:

En clase generaremos un Excel con datos por provincia, enfocándonos en la población activa, es decir, personas que utilizan activamente internet y servicios de red. Para calcularla, excluiremos a la población de la tercera edad, ya que menos del 40% de este grupo usa telefonía en áreas urbanas y rurales. Nos basaremos en reportes previos y añadiremos datos como la población activa y las soluciones correspondientes, considerando esta población ajustada.

```
# Función para quitar tildes
def quitar_tildes(texto):
    return ''.join((c for c in unicodedata.normalize('NFD', texto) if unicodedata.category(c) != 'Mn'))
```

función quitar_tildes(texto):

Elimina los acentos o tildes de un texto. Utiliza la normalización Unicode (NFD), que descompone los caracteres acentuados en dos partes (la letra base y el acento). Luego, filtra los caracteres de categoría 'Mn' (marcas de acento), eliminándolos. Así, solo quedan las letras sin tildes.

Constructor: __init__(self, archivo_excel)

```
class Poblacion:
    def __init__(self, excel_path, sheet_name):
        self.excel_path = excel_path
        self.sheet_name = sheet_name
        self.data = None
        self.df_total = None
        self.df_reducido = None
        self.df_estaciones = None
        self.df_reporte = None # Agregar un atributo para el reporte
```

En el método __init__, se inicializan los siguientes atributos: excel_path (la ruta al archivo Excel), sheet_name (el nombre de la hoja dentro del Excel), data (un espacio para almacenar los datos cargados del Excel), df_total, df_reducido, df_estaciones, df_reporte: atributos para almacenar diferentes versiones de los datos procesados, como el total de población, población reducida, estaciones base y reporte.

Método cargar_datos(self):

```

def cargar_datos(self):
    self.data = pd.read_excel(self.xlsx_path, sheet_name=self.sheet_name, header=0)
    # Renombrar columnas si la primera aparece como NaN
    if pd.isna(self.data.columns[0]):
        self.data.rename(columns={self.data.columns[0]: 'Departamento, provincia, area urbana y rural; y sexo'}, inplace=True)
    # Limpiar nombres de columnas
    self.data.columns = self.data.columns.str.strip().str.replace('\n', '')
    # Eliminar tildes en los nombres de las columnas
    self.data.columns = [quitar_tildes(col) for col in self.data.columns]
    # Agregar columnas para DEPARTAMENTO y PROVINCIA
    self.data['DEPARTAMENTO'] = None
    self.data['PROVINCIA'] = None
    # Extraer departamentos y provincias
    provincia_actual = None
    departamento_actual = None
    for index, row in self.data.iterrows():
        columna_principal = row['Departamento, provincia, area urbana y rural; y sexo']
        if isinstance(columna_principal, str):
            columna_principal = columna_principal.strip()
            # Identificar departamentos y provincias
            if columna_principal.isupper() and "URBANA" not in columna_principal and "RURAL" not in columna_principal:
                if "DEPARTAMENTO" in columna_principal:
                    departamento_actual = columna_principal.replace("DEPARTAMENTO ", "").strip()
                    provincia_actual = None # Reiniciar la provincia cuando se detecta un departamento
                else:
                    provincia_actual = columna_principal.strip()
            else:
                # Asignar provincia y departamento
                self.data.at[index, 'PROVINCIA'] = provincia_actual
                self.data.at[index, 'DEPARTAMENTO'] = departamento_actual
    # Eliminar la palabra 'PROVINCIA' en los valores de la columna 'PROVINCIA'
    self.data['PROVINCIA'] = self.data['PROVINCIA'].str.replace('PROVINCIA ', '', regex=False)
    # Eliminar tildes en los valores de las celdas
    self.data = self.data.applymap(lambda x: quitar_tildes(x) if isinstance(x, str) else x)

```

Este método `cargar_datos` se encarga de cargar y procesar datos desde un archivo Excel, del cual contiene información de las edades por provincia

a) Cargar el archivo: Utiliza `pd.read_excel` para leer los datos de un archivo Excel

b) Renombrar columnas (condicional): Si el primer nombre de columna del excel que estamos leyendo es NaN (vacío), lo renombra a 'Departamento, provincia, area urbana y rural; y sexo' para asegurar que no haya valores nulos.

c) Limpiar los nombres de las columnas y eliminamos tildes de las columnas:

`self.data.columns = self.data.columns.str.strip().str.replace('\n', '')`: Elimina espacios extras y saltos de línea (`\n`) de los nombres de las columnas.
`self.data.columns = [quitar_tildes(col) for col in self.data.columns]`: Usa la función `quitar_tildes` para eliminar las tildes en los nombres de las columnas.

d) Agregar columnas para 'DEPARTAMENTO' y 'PROVINCIA':

```

self.data['DEPARTAMENTO'] = None
self.data['PROVINCIA'] = None

```

Crea dos nuevas columnas vacías en el DataFrame (DEPARTAMENTO y PROVINCIA). Ambas inicializada con valores None (vacíos). Esto permitirá que posteriormente se pueda asignar valores a esas columnas.

e) Extraer departamentos y provincias:

Itera sobre cada fila del DataFrame. Si la columna principal ('Departamento, provincia, area urbana y rural; y sexo') contiene un texto que está en mayúsculas y no menciona "URBANA" ni "RURAL", considera que es un departamento o una provincia.

Si encuentra "DEPARTAMENTO", lo guarda como departamento_actual y reinicia la provincia.

Si no es un departamento, lo considera una provincia y lo asigna a provincia_actual.

Si no es un nombre de departamento o provincia, asigna el valor correspondiente a las columnas DEPARTAMENTO y PROVINCIA de esa fila.

f) Eliminar la palabra 'PROVINCIA' en los valores de la columna 'PROVINCIA':

Elimina la palabra 'PROVINCIA' de los valores en la columna PROVINCIA usando str.replace().

Esto nos servirá para que luego comparemos valores entre Excel y en ambos únicamente esté el nombre de la provincia

Método procesar_datos(self):

```
def procesar_total(self):
    """Procesar los datos de población total."""
    df_filtered = self.data[self.data['Departamento, provincia, area urbana y rural; y sexo']
                           .str.contains('URBANA|RURAL', na=False)]
    self.df_total = df_filtered.groupby(['DEPARTAMENTO', 'PROVINCIA'], as_index=False).sum()
    self.df_total = self.df_total[['DEPARTAMENTO', 'PROVINCIA', 'Total', '14 a 29', '30 a 44', '45 a 64', '65 y mas']]
```

a) Filtrar los datos:

df_filtered = self.data[self.data['Departamento, provincia, area urbana y rural; y sexo'].str.contains('URBANA|RURAL', na=False)]

Filtrar las filas del DataFrame self.data para seleccionar aquellas donde la columna principal contiene las palabras "URBANA" o "RURAL". Esto asegura que solo se trabajen con las áreas urbanas y rurales, ignorando otras categorías.

b) Agrupa los datos: Agrupa los datos por DEPARTAMENTO y PROVINCIA, sumando las demás columnas numéricas. Esto crea un nuevo DataFrame self.df_total, que contiene la suma de las poblaciones por cada combinación de departamento y provincia

c) Seleccionar columnas relevantes: Filtra las columnas para que el DataFrame final (self.df_total) contenga solo las columnas relevantes: DEPARTAMENTO, PROVINCIA, y los diferentes rangos de edad (Total, 14 a 29, 30 a 44, 45 a 64, 65 y más).

Método: ajustar_reducida(self):

```

def ajustar_reducida(self):
    """Ajustar la columna '65 y mas' y recalcular los totales."""
    df_filtered = self.data[self.data['Departamento, provincia, area urbana y rural; y sexo']
                           .str.contains('URBANA|RURAL', na=False)].copy()

def ajustar_poblacion(row):
    if 'URBANA' in row['Departamento, provincia, area urbana y rural; y sexo']:
        return round(row['65 y mas'] * 0.35) # 35% para zonas urbanas
    elif 'RURAL' in row['Departamento, provincia, area urbana y rural; y sexo']:
        return round(row['65 y mas'] * 0.20) # 20% para zonas rurales
    return row['65 y mas']

df_filtered['65 y mas'] = df_filtered.apply(ajustar_poblacion, axis=1)
df_filtered['Total'] = df_filtered[['14 a 29', '30 a 44', '45 a 64', '65 y mas']].sum(axis=1)

self.df_reducido = df_filtered.groupby(['DEPARTAMENTO', 'PROVINCIA'], as_index=False).sum()
self.df_reducido = self.df_reducido[['DEPARTAMENTO', 'PROVINCIA', 'Total', '14 a 29', '30 a 44', '45 a 64', '65 y mas']]

```

a) Filtrar los datos de áreas urbanas y rurales:

Filtrar los datos para seleccionar solo las filas correspondientes a áreas urbanas y rurales.

b) Definir la función de ajuste de población:

def ajustar_poblacion(row):

```

if 'URBANA' in row['Departamento, provincia, area urbana y rural; y sexo']:
    return round(row['65 y mas'] * 0.35) # 35% para zonas urbanas
elif 'RURAL' in row['Departamento, provincia, area urbana y rural; y sexo']:
    return round(row['65 y mas'] * 0.20) # 20% para zonas rurales
return row['65 y mas']

```

Se ajusta la población de la columna "65 y más", la cual es la menos afectada o influenciada por los avances relacionados con telecomunicaciones, dependiendo de si la fila corresponde a una zona urbana o rural: 35% se asigna a las zonas urbanas, 20% se asigna a las zonas rurales. Si no se encuentra en ninguna de estas categorías, mantiene el valor original.

c) Aplicar la función de ajuste: Aplica la función `ajustar_poblacion` a cada fila del DataFrame `df_filtered`.

d) Recacular el total: Suma las columnas de población por grupo de edad (excluyendo "Total") para recalculiar el total de la población en cada fila.

e) Agrupar y ajustar los datos: Agrupa los datos ajustados por DEPARTAMENTO y PROVINCIA, y luego suma las columnas correspondientes para obtener los totales ajustados por cada provincia.

Finalmente, selecciona las columnas relevantes: DEPARTAMENTO, PROVINCIA, y las categorías de población (Total, 14 a 29, 30 a 44, 45 a 64, 65 y más).

Método guardar_archivos(self):

```

def guardar_archivos(self):
    """Guardar los datos procesados en archivos Excel."""
    if self.df_total is not None:
        self.df_total.to_excel('archivo_poblacion_total.xlsx', index=False)
        print("Archivo 'archivo_poblacion_total.xlsx' guardado.")
    if self.df_reducido is not None:
        self.df_reducido.to_excel('archivo_poblacion_reducida.xlsx', index=False)
        print("Archivo 'archivo_poblacion_reducida.xlsx' guardado.")

```

- a) **Guardar df_total:** Si df_total no es None, se guarda en 'archivo_poblacion_total.xlsx'. Se imprime un mensaje confirmando el guardado.
- b) **Guardar df_reducido:** Si df_reducido no es None, se guarda en 'archivo_poblacion_reducida.xlsx'. Se imprime un mensaje confirmando el guardado.

Método ejecutar_procesamiento(self, cobertura_excel_path=None):

```

def ejecutar_procesamiento(self, cobertura_excel_path=None):
    """Ejecutar todo el proceso: cargar, procesar y guardar."""
    print("Cargando datos...")
    self.cargar_datos()

    print("Procesando población total...")
    self.procesar_total()

    print("Ajustando población reducida...")
    self.ajustar_reducida()

    print("Guardando archivos...")
    self.guardar_archivos()

```

- a) **Cargar los datos:** Llama al método cargar_datos para cargar los datos desde el archivo Excel.
- b) **Procesar población total:** Llama al método procesar_total para procesar los datos de población total.
- c) **Ajustar población reducida:** Llama al método ajustar_reducida para ajustar los datos de población según el área (urbana o rural).
- d) **Guardar los archivos:** Llama al método guardar_archivos para guardar los resultados en archivos Excel.

Método cargar_datos_activa(self):

```

def cargar_datos_activa(self):
    """Cargar los archivos necesarios para agregar la población activa."""
    # Cargar el archivo de población reducida
    self.df_reducido = pd.read_excel('archivo_poblacion_reducida.xlsx')

    # Cargar el archivo de reporte por provincia
    self.df_reporte = pd.read_excel('REPORTE_PROVINCIA.xlsx')

```

- a) Cargar el archivo de población reducida
- b) Cargar el archivo de reporte por provincia

Método agregar_poblacion_activa(self)

```

def agregar_poblacion_activa(self):
    """Aregar la columna 'Total' de archivo_poblacion_reducida a REPORTE_PROVINCIA."""
    # Realizar el merge entre ambos DataFrames usando las columnas 'DEPARTAMENTO' y 'PROVINCIA'
    df_merge = pd.merge(self.df_reporte, self.df_reducido[['DEPARTAMENTO', 'PROVINCIA', 'Total']],
                         on=['DEPARTAMENTO', 'PROVINCIA'], how='left')

    # Renombrar la columna 'Total' como 'POBLACION_ACTIVA'
    df_merge['POBLACION_ACTIVA'] = df_merge['Total']
    df_merge.drop(columns=['Total'], inplace=True) # Eliminar la columna 'Total' original

    # Asignar el DataFrame con la nueva columna al atributo df_reporte
    self.df_reporte = df_merge

    # Sobrescribir el archivo 'REPORTE_PROVINCIA.xlsx' con la nueva columna
    self.df_reporte.to_excel('REPORTE_PROVINCIA.xlsx', index=False)
    print("Archivo 'REPORTE_PROVINCIA.xlsx' actualizado con la columna POBLACION_ACTIVA.")

```

- a) **Fusionar los DataFrames:** Realiza una combinación (merge) entre df_reporte y una versión reducida de df_reducido, usando las columnas 'DEPARTAMENTO' y 'PROVINCIA'. Esto agrega la columna "Total" de df_reducido a df_reporte.
- b) **Renombrar la columna "Total":** Renombra la columna "Total" como "POBLACION_ACTIVA" y elimina la columna original "Total".
- c) **Actualizar el DataFrame y guardar el archivo:** Asigna el DataFrame con la nueva columna "POBLACION_ACTIVA" a df_reporte. Sobrescribe el archivo 'REPORTE_PROVINCIA.xlsx' con el DataFrame actualizado.

Método agregar_propiedades_eb_pea(self):

```

def agregar_propuestas_eb_pea(self):
    """Añadir la columna 'PROPIETAS_EB_PEA' y asegurar que las columnas ALCANCE_EB_PEA y EB_NECESSARIAS_PEA sean enteros."""
    # Calcular ALCANCE_EB_PEA y EB_NECESSARIAS_PEA
    self.df_reporte['ALCANCE_EB_PEA'] = self.df_reporte['POBLACION_ACTIVA'] / self.df_reporte['CANT_EB_TOTAL']
    self.df_reporte['EB_NECESSARIAS_PEA'] = self.df_reporte['POBLACION_ACTIVA'] / 150

    # Reemplazar los valores NaN o inf con 0 antes de la conversión a entero
    self.df_reporte['ALCANCE_EB_PEA'] = self.df_reporte['ALCANCE_EB_PEA'].replace([float('NaN'), -float('inf')], 0).fillna(0)
    self.df_reporte['EB_NECESSARIAS_PEA'] = self.df_reporte['EB_NECESSARIAS_PEA'].replace([float('NaN'), -float('inf')], 0).fillna(0)

    # Convertir a enteros (truncar decimales)
    self.df_reporte['ALCANCE_EB_PEA'] = self.df_reporte['ALCANCE_EB_PEA'].astype(int)
    self.df_reporte['EB_NECESSARIAS_PEA'] = self.df_reporte['EB_NECESSARIAS_PEA'].astype(int)

    # Función para definir la propuesta en 'PROPIETAS_EB_PEA'
    def propuesta(row):
        if row['ALCANCE_EB_PEA'] < 150:
            return f"La cantidad de estaciones base ({row['CANT_EB_TOTAL']}) es muy baja para la población {row['PROVINCIA']}, debemos aumentar la cantidad de EB a {row['EB_NECESSARIAS_PEA']} en total para una cobertura más completa."
        else:
            return f"La cantidad de estaciones base ({row['CANT_EB_TOTAL']}) es suficiente pero podemos aumentar las estaciones base para mejorar la capacidad de la población en {row['PROVINCIA']}."

    # Aplicar la función a cada fila
    self.df_reporte['PROPIETAS_EB_PEA'] = self.df_reporte.apply(propuesta, axis=1)
    # Guardar el archivo actualizado
    self.df_reporte.to_excel('REPORTE_PROVINCIA.xlsx', index=False)
    print("Archivo 'REPORTE_PROVINCIA.xlsx' actualizado con la columna PROPIETAS_EB_PEA.")

    # Imprimimos para verificar
    print(self.df_reporte.head())
    self.df_reporte.to_excel('REPORTE_PROVINCIA.xlsx', index=False)
    # Abrir el archivo Excel previamente guardado con pandas utilizando openpyxl
    wb = openpyxl.load_workbook("REPORTE_PROVINCIA.xlsx")
    ws = wb.active # Seleccionar la hoja activa del archivo
    # Iterar sobre todas las columnas del archivo
    for col in ws.columns:
        max_length = 0 # Inicializar variable para el largo máximo de la celda en la columna
        column = col[0].column_letter # Obtener la letra de la columna (A, B, C, etc.)
        # Iterar sobre las celdas de la columna para encontrar la longitud máxima de contenido
        for cell in col:
            try:
                # Verificar si la longitud del contenido de la celda es mayor que el largo máximo actual
                if len(str(cell.value)) > max_length:
                    max_length = len(str(cell.value)) # Actualizar el largo máximo
            except:
                pass # Ignorar celdas vacías o errores de tipo de datos
        # Ajustar el ancho de la columna basándose en el largo máximo encontrado
        adjusted_width = (max_length + 2) # Se agrega un pequeño margen de 2 para mejor visibilidad
        ws.column_dimensions[column].width = adjusted_width # Ajustar el tamaño de la columna
    # Guardar el archivo Excel con las columnas ajustadas
    wb.save("REPORTE_PROVINCIA.xlsx")

```

a) Calcular las columnas ALCANCE_EB_PEA y EB_NECESSARIAS_PEA:

$$\text{self.df_reporte['ALCANCE_EB_PEA']} = \text{self.df_reporte['POBLACION_ACTIVA']} / \text{self.df_reporte['CANT_EB_TOTAL']}$$

$$\text{self.df_reporte['EB_NECESARIAS_PEA']} = \text{self.df_reporte['POBLACION_ACTIVA']} / 150$$

b) Reemplazar valores NaN o inf con 0: Reemplaza los valores infinitos o NaN por 0 en ambas columnas para evitar errores en la conversión.

c) Convertir las columnas a enteros: Convierte ambas columnas a enteros, truncando los decimales.

Método propuestas(row) dentro de Método agregar_propuestas_eb_pea(self):

```

# Función para definir la propuesta en 'PROPIETAS_EB_PEA'
def propuesta(row):
    if row['ALCANCE_EB_PEA'] < 150:
        return f"La cantidad de estaciones base ({row['CANT_EB_TOTAL']}) es muy baja para la población {row['PROVINCIA']}, debemos aumentar la cantidad de EB a {row['EB_NECESSARIAS_PEA']} en total para una cobertura más completa."
    else:
        return f"La cantidad de estaciones base ({row['CANT_EB_TOTAL']}) es suficiente pero podemos aumentar las estaciones base para mejorar la capacidad de la población en {row['PROVINCIA']}."

    # Aplicar la función a cada fila
    self.df_reporte['PROPIETAS_EB_PEA'] = self.df_reporte.apply(propuesta, axis=1)
    # Guardar el archivo actualizado
    self.df_reporte.to_excel('REPORTE_PROVINCIA.xlsx', index=False)
    print("Archivo 'REPORTE_PROVINCIA.xlsx' actualizado con la columna PROPIETAS_EB_PEA.")

    # Imprimimos para verificar
    print(self.df_reporte.head())
    self.df_reporte.to_excel('REPORTE_PROVINCIA.xlsx', index=False)
    # Abrir el archivo Excel previamente guardado con pandas utilizando openpyxl
    wb = openpyxl.load_workbook("REPORTE_PROVINCIA.xlsx")
    ws = wb.active # Seleccionar la hoja activa del archivo
    # Iterar sobre todas las columnas del archivo
    for col in ws.columns:
        max_length = 0 # Inicializar variable para el largo máximo de la celda en la columna
        column = col[0].column_letter # Obtener la letra de la columna (A, B, C, etc.)
        # Iterar sobre las celdas de la columna para encontrar la longitud máxima de contenido
        for cell in col:
            try:
                # Verificar si la longitud del contenido de la celda es mayor que el largo máximo actual
                if len(str(cell.value)) > max_length:
                    max_length = len(str(cell.value)) # Actualizar el largo máximo
            except:
                pass # Ignorar celdas vacías o errores de tipo de datos
        # Ajustar el ancho de la columna basándose en el largo máximo encontrado
        adjusted_width = (max_length + 2) # Se agrega un pequeño margen de 2 para mejor visibilidad
        ws.column_dimensions[column].width = adjusted_width # Ajustar el tamaño de la columna
    # Guardar el archivo Excel con las columnas ajustadas
    wb.save("REPORTE_PROVINCIA.xlsx")

```

a) Definir la función propuesta: Esta función genera un mensaje de propuesta dependiendo del valor de ALCANCE_EB_PEA:

- Si es menor que 150, sugiere aumentar el número de estaciones base.
- Si es igual o mayor, indica que el número actual es suficiente, pero se podría mejorar.

b) Aplicar la función a cada fila: Aplica la función propuesta a cada fila del DataFrame df_reporte, creando una nueva columna PROPUESTAS_EB_PEA con los mensajes generados.

c) Guardar el archivo Excel actualizado

d) Ajustar el ancho de las columnas

6. Clase PrediccionSituacionFutura:

```
class PrediccionSituacionFutura:
    def entrenar_modelo(self):
        # Dividir los datos en entrenamiento y prueba (80% - 20%)
        X_train, X_test, y_train, y_test = train_test_split(self.X, self.y, test_size=0.2, random_state=42)

        # Crear el modelo de Deep Learning (Ejemplo sencillo de red neuronal)
        self.model = tf.keras.Sequential([
            tf.keras.layers.Input(shape=(self.X.shape[1],)),
            tf.keras.layers.Dense(64, activation='relu'),
            tf.keras.layers.Dense(64, activation='relu'),
            tf.keras.layers.Dense(3, activation='softmax') # 3 salidas para 'mejorará', 'igual', 'empeorará'
        ]) # Crear capas mas avanzadas

        # Compilar y entrenar el modelo
        self.model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
        # va ir tomando de 32 en 32
        # entrenara el modelo
        self.model.fit(X_train, y_train, epochs=15, batch_size=32)

        # Evaluar el modelo con los datos de prueba
        test_loss, test_acc = self.model.evaluate(X_test, y_test)
        print(f'Precisión en los datos de prueba: {test_acc * 100:.2f}%')
        self.guardar_modelo_y_encoders()

    def guardar_modelo_y_encoders(self):
        # Guardar el modelo
        self.model.save(self.archivo_modelo) # El modelo guarda su estado entrenado en ese URL
        print(f'Modelo guardado en {self.archivo_modelo}.')

        # Guardar los LabelEncoders para las columnas categóricas
        for columna, encoder in self.label_encoders.items():
            np.save(f'{columna}_encoder.npy', encoder.classes_)
            np.save('situacion_encoder.npy', self.label_encoder_situacion.classes_)
        print("Codificadores guardados correctamente.")

    def cargar_modelo_y_encoders(self):
        # Cargar el modelo
        self.model = tf.keras.models.load_model(self.archivo_modelo)
        print(f'Modelo cargado desde {self.archivo_modelo}.')

        # Cargar los LabelEncoders
        for columna in self.columnas_categoricas:
            clases = np.load(f'{columna}_encoder.npy', allow_pickle=True)
            self.label_encoders[columna].classes_ = clases
            self.label_encoder_situacion.classes_ = np.load('situacion_encoder.npy', allow_pickle=True)
            print(f'Clases cargadas para {columna}: {clases}')



```

Método Constructor def __init__(self, archivo_datos,
archivo_modelo='modelo_entrenado.h5'):

a) Cargar los datos

b) Definir columnas categóricas y numéricas: `self.columnas_categoricas` y `self.columnas_numericas`: Se definen dos listas, una con las columnas categóricas (como departamento, provincia, empresa operadora) y otra con las columnas numéricas (como 2G, 3G, 4G, etc.).

c) Codificación de las columnas categóricas:

- self.label_encoders = {col: LabelEncoder() for col in self.columnas_categoricas}:

Se crea un diccionario de codificadores para las columnas categóricas. LabelEncoder se utiliza para convertir las categorías en números.

- self.df[col] = self.label_encoders[col].fit_transform(self.df[col]): Cada columna categórica es transformada en números utilizando LabelEncoder

d) Lógica para insertar la columna SITUACION_FUTURA:

self.df['SITUACION_FUTURA'] = self.df.apply(self.determinar_situacion_futura, axis=1): Se agrega una nueva columna SITUACION_FUTURA utilizando una función (determinar_situacion_futura) aplicada fila por fila.

e) Codificación de la columna SITUACION_FUTURA:

self.label_encoder_situacion = LabelEncoder(): Se crea otro codificador para la columna SITUACION_FUTURA y se aplica a esta columna.

f) Rellenar valores vacíos:

self.df[self.columnas_numericas] = self.df[self.columnas_numericas].fillna(0):

Se rellenan los valores faltantes en las columnas numéricas con 0.

g) Preparar las características (X) y la etiqueta (y) para entrenamiento:

self.X = self.df.drop('SITUACION_FUTURA', axis=1).values.astype(np.float32): Se eliminan las columnas innecesarias (en este caso SITUACION_FUTURA) para crear las características (X).

self.y = self.df['SITUACION_FUTURA'].values.astype(np.float32): La columna SITUACION_FUTURA se convierte en la variable objetivo (y).

h) Carga del modelo entrenado (si existe) y si el modelo no se encuentra:

if os.path.exists(self.archivo_modelo): Verifica si el archivo del modelo ya existe. Si es así, carga el modelo entrenado y los codificadores. Si no existe, indica que debe entrenarse el modelo.

self.model = None: Si el modelo no se encuentra, se establece self.model como None.

Método determinar_situacion_futura(self, row):

a) Inicialización del puntaje:

La variable puntaje se inicializa en 0, y a medida que se revisan las diferentes condiciones, se incrementa o no.

b) Evaluación de las tecnologías disponibles:

Para cada tecnología de red (2G, 3G, 4G, 5G, HASTA_1_MBPS, MÁS_DE_1_MBPS), se asigna un puntaje dependiendo de si está disponible o no:

```

def entrenar_modelo(self):
    # Dividir los datos en entrenamiento y prueba (80% - 20%)
    X_train, X_test, y_train, y_test = train_test_split(self.X, self.y, test_size=0.2, random_state=42)

    # Crear el modelo de Deep Learning (Ejemplo sencillo de red neuronal)
    self.model = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(self.X.shape[1],)),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(3, activation='softmax') # 3 salidas para 'mejorará', 'igual', 'empeorará'
    ]) # Crear capas mas avanzadas

    # Compilar y entrenar el modelo
    self.model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    # va ir tomando de 32 en 32
    # entrenara el modelo
    self.model.fit(X_train, y_train, epochs=15, batch_size=32)

    # Evaluar el modelo con los datos de prueba
    test_loss, test_acc = self.model.evaluate(X_test, y_test)
    print(f"Precisión en los datos de prueba: {test_acc * 100:.2f}%")
    self.guardar_modelo_y_encoders()

```

Si la red está disponible (valor 1), se suma un puntaje específico. Si no está disponible (valor 0), no se suma puntaje. Los puntajes asignados a cada tecnología son:

2G → +1	4G → +3
3G → +2	5G → +4
HASTA_1_MBPS → +1	MÁS_DE_1_MBPS → +1

c) Evaluación de las estaciones base:

Para cada tipo de estación base (CANT_EB_2G, CANT_EB_3G, CANT_EB_4G, CANT_EB_5G), se evalúa si hay más de 2 estaciones base. Si hay más de 2, se suma 1 al puntaje.

Si CANT_EB_2G > 2 → +1	Si CANT_EB_4G > 2 → +1
Si CANT_EB_3G > 2 → +1	Si CANT_EB_5G > 2 → +1

4. Evaluación de la población:

Dependiendo del tamaño de la población (POBLACION), se ajusta el puntaje:

- Si la población es mayor a 1000, se resta 2 puntos, ya que se considera que la conectividad podría empeorar si la población es alta y las estaciones base son insuficientes.
- Si la población es baja (≤ 1000), se suma 1 punto, ya que se considera que la conectividad mejorará si hay suficientes estaciones base para cubrir una población pequeña.

5. Determinación de la situación futura:

Una vez calculado el puntaje total, la función determina el estado futuro:

- Si el puntaje es mayor o igual a 10, la situación mejorará.
- Si el puntaje está entre 5 y 9, la situación permanecerá igual.
- Si el puntaje es menor a 5, la situación empeorará.

La función devuelve una de las siguientes tres opciones:

- "Mejorará"
- "IGUAL"
- "Empeorará"

Método entrenar_modelo(self)

```
def entrenar_modelo(self):
    # Dividir los datos en entrenamiento y prueba (80% - 20%)
    X_train, X_test, y_train, y_test = train_test_split(self.X, self.y, test_size=0.2, random_state=42)

    # Crear el modelo de Deep Learning (Ejemplo sencillo de red neuronal)
    self.model = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(self.X.shape[1],)),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(3, activation='softmax') # 3 salidas para 'mejorará', 'igual', 'empeorará'
    ]) # Crear capas mas avanzadas

    # Compilar y entrenar el modelo
    self.model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    # va ir tomando de 32 en 32
    # entrenara el modelo
    self.model.fit(X_train, y_train, epochs=15, batch_size=32)

    # Evaluar el modelo con los datos de prueba
    test_loss, test_acc = self.model.evaluate(X_test, y_test)
    print(f"Precisión en los datos de prueba: {test_acc * 100:.2f}%")
    self.guardar_modelo_y_encoders()
```

a) División de datos en entrenamiento y prueba:

Utilizas la función `train_test_split` de scikit-learn para dividir tus datos en conjuntos de entrenamiento y prueba:

`X_train, X_test, y_train, y_test = train_test_split(self.X, self.y, test_size=0.2, random_state=42)`

El 80% de los datos se usa para el entrenamiento.

El 20% restante se usa para las pruebas.

b) Creación del modelo de Deep Learning::

Usas un modelo secuencial de Keras con las siguientes capas:

- Capa de entrada: La entrada tiene la forma (`self.X.shape[1]`), que corresponde al número de características (columnas) en tus datos.
- Capas ocultas: Dos capas `Dense` con 64 neuronas cada una y función de activación ReLU.
- Capa de salida: Tres neuronas de salida con activación softmax

c) Compilación y entrenamiento del modelo:

- Compilación: Usas el optimizador Adam y la función de pérdida `sparse_categorical_crossentropy` porque estás haciendo una clasificación multiclas. La métrica que monitorean es la precisión (accuracy).
- Entrenamiento: El modelo se entrena por 15 épocas con un tamaño de lote de 32.

```
self.model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
self.model.fit(X_train, y_train, epochs=15, batch_size=32)
```

d) Evaluación del modelo:

Después de entrenar el modelo, lo evalúas en el conjunto de prueba para ver cómo generaliza a nuevos datos:

La pérdida (test_loss) y la precisión (test_acc) se calculan y luego se imprime la precisión.

```
test_loss, test_acc = self.model.evaluate(X_test, y_test)
print(f"Precisión en los datos de prueba: {test_acc * 100:.2f}%")
```

e) Guardar el modelo y los codificadores:

Después de entrenar y evaluar el modelo, llamas a la función guardar_modelo_y_encoders, que guarda el modelo entrenado y los codificadores utilizados en el procesamiento de los datos (como el LabelEncoder).

Método guardar_modelo_y_encoders(self):

```
def guardar_modelo_y_encoders(self):
    # Guardar el modelo
    self.model.save(self.archivo_modelo) # El modelo guarda su estado entrenado en ese URL
    print(f"Modelo guardado en {self.archivo_modelo}.")

    # Guardar los LabelEncoders para las columnas categóricas
    for columna, encoder in self.label_encoders.items():
        np.save(f'{columna}_encoder.npy', encoder.classes_)
    np.save('situacion_encoder.npy', self.label_encoder_situacion.classes_)
    print("Codificadores guardados correctamente.")
```

a) Guardar el modelo:

El modelo entrenado se guarda en un archivo utilizando el método save de Keras, lo que guarda tanto la arquitectura como los pesos del modelo.

b) Guardar los codificadores (LabelEncoders):

- Guardas los codificadores de las columnas categóricas con np.save. Cada codificador se guarda en un archivo.npy cuyo nombre es el nombre de la columna.
- También guardas el codificador para la columna SITUACION_FUTURA en el archivo situacion_encoder.npy.

Método cargar_modelo_y_encoders(self):

a) Cargar el modelo:

- Se carga el modelo de Keras desde el archivo especificado (self.archivo_modelo) usando el método tf.keras.models.load_model. Este método carga tanto la arquitectura como los pesos del modelo entrenado.
- Se imprime un mensaje confirmando que el modelo ha sido cargado correctamente.

```

def cargar_modelo_y_encoders(self):
    # Cargar el modelo
    self.modelo = tf.keras.models.load_model(self.archivo_modelo)
    print(f"Modelo cargado desde {self.archivo_modelo}.")

    # Cargar los LabelEncoders
    for columna in self.columnas_categoricas:
        clases = np.load(f'{columna}_encoder.npy', allow_pickle=True)
        self.label_encoders[columna].classes_ = clases
    self.label_encoder_situacion.classes_ = np.load('situacion_encoder.npy', allow_pickle=True)
    print("Codificadores cargados correctamente.")

```

b) Cargar los LabelEncoders:

- Se itera sobre las columnas categóricas (self.columnas_categoricas).
- Para cada columna, se cargan las clases del LabelEncoder desde el archivo .npy correspondiente usando np.load. Luego, se asignan esas clases al atributo classes_ del LabelEncoder.

c) Cargar el codificador de la columna SITUACION_FUTURA:

Se carga el codificador para la columna SITUACION_FUTURA desde el archivo situacion_encoder.npy y se asigna al atributo classes_ del LabelEncoder correspondiente.

d) Mensaje de éxito:

Se imprime un mensaje confirmando que los codificadores han sido cargados correctamente.

Método predecir_situaciones (self,departamento,distrito, centro_poblado, provincia):

```

class PrediccionSituacionFutura:
    def predecir_situacion(self,departamento,distrito, centro_poblado, provincia):
        # Convertir los valores categóricos a su codificación numérica
        distrito_codificado = self.transformar_categoria('DISTRITO', distrito)
        centro_poblado_codificado = self.transformar_categoria('CENTRO_POBLADO', centro_poblado)
        provincia_codificada = self.transformar_categoria('PROVINCIA', provincia)
        departamento_codificado = self.transformar_categoria('DEPARTAMENTO', departamento)
        # Verificar si se pudo codificar correctamente
        # -1 = no encontrado o error
        if any(val == -1 for val in [ distrito_codificado, centro_poblado_codificado, provincia_codificada, departamento_codificado]):
            print("No se encontraron datos para la localidad ingresada.")
            return
        # Filtrar el DataFrame con los valores proporcionados por el usuario
        datos_filtrados = self.df[(self.df['DISTRITO'] == distrito_codificado) &
                                    (self.df['CENTRO_POBLADO'] == centro_poblado_codificado) &
                                    (self.df['PROVINCIA'] == provincia_codificada) &
                                    (self.df['DEPARTAMENTO'] == departamento_codificado)]
        # Verificar si se encontró algún registro
        # el empty verifica que si esta vacio
        if not datos_filtrados.empty:
            # Obtener las demás columnas que el modelo espera
            cant_2g = datos_filtrados['CANT_EB_2G'].values[0]
            cant_3g = datos_filtrados['CANT_EB_3G'].values[0]
            cant_4g = datos_filtrados['CANT_EB_4G'].values[0]
            cant_5g = datos_filtrados['CANT_EB_5G'].values[0]
            hasta_1mbps = datos_filtrados['HASTA_1_MBPS'].values[0]
            mas_de_1mbps = datos_filtrados['MÁS_DE_1_MBPS'].values[0]
            poblacion = datos_filtrados['POBLACION'].values[0]
            dosG = datos_filtrados['2G'].values[0]
            tresG = datos_filtrados['3G'].values[0]
            cuatroG = datos_filtrados['4G'].values[0]
            cincoG = datos_filtrados['5G'].values[0]
            Situacion = datos_filtrados['SITUACION_FUTURA'].values[0]
            ubigeo = datos_filtrados['UBIGEO_CCPP'].values[0]
            empresa_operadora = datos_filtrados['EMPRESA_OPERADORA'].values[0]
            # Crear el array con los datos para predecir y darle la forma correcta
            datos_para_predecir = np.array([[departamento_codificado, distrito_codificado, centro_poblado_codificado,
                                              provincia_codificada, ubigeo, empresa_operadora, dosG, tresG, cuatroG, cincoG,
                                              cant_2g, cant_3g, cant_4g, cant_5g, hasta_1mbps, mas_de_1mbps, poblacion, Situacion]]))
            # Hacer la predicción con el modelo
            prediccion = self.modelo.predict(datos_para_predecir)
            # Obtener la clase predicha. el axis= 1 es para las filas
            # Probabilidad max

```

a) Codificación de valores categóricos:

```
distrito_codificado = self.transformar_categoria('DISTRITO', distrito)
centro_poblado_codificado = self.transformar_categoria('CENTRO_POBLADO', centro_poblado)
provincia_codificada = self.transformar_categoria('PROVINCIA', provincia)
departamento_codificado = self.transformar_categoria('DEPARTAMENTO', departamento)
```

- Las entradas del usuario (distrito, centro_poblado, provincia, departamento) son datos categóricos. Los modelos de machine learning generalmente no entienden directamente los datos categóricos, por lo que deben ser transformados en valores numéricos.
- `transformar_categoria()`: usa un LabelEncoder previamente entrenado para convertir estas categorías en números, de modo que puedan ser usadas por el modelo.

b) Verificación de codificación

```
if any(val == -1 for val in [distrito_codificado, centro_poblado_codificado, provincia_codificada,
departamento_codificado]):  
    print("No se encontraron datos para la localidad ingresada.")  
    return
```

Después de transformar las categorías, se verifica si alguna de las variables codificadas resultó en un valor -1, lo que indica que la categoría ingresada no se encuentra en el conjunto de datos entrenados. Si esto sucede, se imprime un mensaje de error y se sale de la función.

c) Filtrado de datos:

```
datos_filtrados = self.df[(self.df['DISTRITO'] == distrito_codificado) &
                           (self.df['CENTRO_POBLADO'] == centro_poblado_codificado) &
                           (self.df['PROVINCIA'] == provincia_codificada) &
                           (self.df['DEPARTAMENTO'] == departamento_codificado)]
```

Se filtra el DataFrame (`self.df`) para encontrar las filas que coinciden con las categorías codificadas ingresadas. Aquí se busca el conjunto de datos específico que corresponde a la localidad indicada por el usuario.

d) Comprobación de existencia de datos:

Si el DataFrame resultante después del filtro está vacío, significa que no se encontró ninguna entrada para la localidad proporcionada. En ese caso, se imprime un mensaje indicando que no se encontraron datos.

```
if not datos_filtrados.empty:
```

e) Extracción de características para la predicción:

```
cant_2g = datos_filtrados['CANT_EB_2G'].values[0]
```

```

cant_3g = datos_filtrados['CANT_EB_3G'].values[0]
cant_4g = datos_filtrados['CANT_EB_4G'].values[0]
cant_5g = datos_filtrados['CANT_EB_5G'].values[0]
hasta_1mbps = datos_filtrados['HASTA_1_MBPS'].values[0]
mas_de_1mbps = datos_filtrados['MÁS_DE_1_MBPS'].values[0]
poblacion = datos_filtrados['POBLACION'].values[0]
dosG = datos_filtrados['2G'].values[0]
tresG = datos_filtrados['3G'].values[0]
cuatroG = datos_filtrados['4G'].values[0]
cincoG = datos_filtrados['5G'].values[0]
Situacion = datos_filtrados['SITUACION_FUTURA'].values[0]
ubigeo = datos_filtrados['UBIGEO_CCPP'].values[0]
empresa_operadora = datos_filtrados['EMPRESA_OPERADORA'].values[0]

```

Una vez que se encuentran los datos correspondientes al centro poblado, se extraen las características necesarias para la predicción. Estas características incluyen la cantidad de estaciones base para diferentes tecnologías (2G, 3G, 4G, 5G), la calidad del internet (hasta 1 Mbps, más de 1 Mbps), la población y la situación futura (el valor que estamos prediciendo).

f) Preparación de los datos para la predicción

```

datos_para_predecir=np.array([[departamento_codificado,distrito_codificado,
                               centro_poblado_codificado,provincia_codificada,ubigeo,empresa_o
                               peradora, dosG, tresG, cuatroG, cincoG, cant_2g, cant_3g,
                               cant_4g, cant_5g, hasta_1mbps, mas_de_1mbps, poblacion,
                               Situacion]])

```

Los datos extraídos se organizan en un array de Numpy, que es la estructura que el modelo espera para hacer la predicción. Cada entrada corresponde a una característica que el modelo ha aprendido a interpretar.

g) Predicción con el modelo

```
prediccion = self.model.predict(datos_para_predecir)
```

El modelo realiza la predicción sobre los datos preparados. La salida de `predict()` es una matriz de probabilidades para cada clase (mejorará, igual, empeorará).

h) Obtener la clase predicha

```
prediccion_clase = np.argmax(prediccion, axis=1)[0]
```

`np.argmax()` selecciona la clase con la probabilidad más alta. Dado que el modelo tiene tres posibles salidas (mejorará, igual, empeorará), se toma la clase con la mayor probabilidad.

i) Decodificación del resultado:

```
resultado = self.label_encoder_situacion.inverse_transform([prediccion_clase])[0]
```

El índice de la clase predicha se convierte de nuevo a su valor original (es decir, "mejorará", "igual", "empeorará") usando el `LabelEncoder`.

j) Mostrar el resultado

La información relevante del centro poblado, como las tecnologías disponibles, la cobertura y la infraestructura de estaciones base, se muestra para proporcionar contexto al resultado de la predicción.

k) Generación de recomendaciones basadas en el resultado:

Dependiendo de la clase predicha, se genera un mensaje con recomendaciones específicas sobre la infraestructura y las mejoras necesarias para cada situación.

- "empeorará": Recomienda priorizar mejoras en infraestructura.
- "Mejorará": Recomienda fortalecer la cobertura existente.
- "IGUAL": Sugiere mejoras en la infraestructura para no dejar que la situación se deteriore.

l) Manejo de la ausencia de datos:

else:

```
print("No se encontraron datos para la localidad ingresada.")
```

Si no se encuentra información en el DataFrame para la localidad indicada, se muestra un mensaje de error.

Método transformar_categoria(self, columna, valor):

```
def transformar_categoria(self, columna, valor):  
    """Función auxiliar para manejar la transformación de categorías con LabelEncoder"""\n    # Verificar si el valor existe en la columna original antes de codificar  
    if valor in self.label_encoders[columna].classes_:  
        return self.label_encoders[columna].transform([valor])[0]  
    else:  
        print(f"El valor '{valor}' no se encuentra en la columna '{columna}'")  
        return -1 # Valor por defecto si no se encuentra
```

a) Definición de la función:

```
def transformar_categoria(self, columna, valor):
```

La función toma dos parámetros:

- columna: El nombre de la columna en la que se encuentra el valor categórico (por ejemplo, "DISTRITO", "PROVINCIA", etc.).
- valor: El valor categórico que se desea transformar (por ejemplo, un nombre de un distrito).

b) Verificación de existencia en las clases:

```
if valor in self.label_encoders[columna].classes_:
```

self.label_encoders[columna]: Es el LabelEncoder que ha sido entrenado previamente para la columna específica (columna).

self.label_encoders[columna].classes_: Contiene todas las clases o categorías que el LabelEncoder ha aprendido durante su entrenamiento. Es decir, todos los valores posibles que han sido transformados en valores numéricos.

Condición if valor in ...: Verifica si el valor proporcionado (valor) está presente en las clases que el LabelEncoder ha aprendido para esa columna.

c) Transformación del valor categórico:

```
return self.label_encoders[columna].transform([valor])[0]
```

Si el valor está presente en las clases de la columna, se realiza la transformación:

- self.label_encoders[columna].transform([valor]): Usa el método transform() del LabelEncoder para convertir el valor categórico en su equivalente numérico.
- [valor]: El valor se pasa como una lista, ya que el método transform() espera una lista o array de valores.
- [0]: transform() devuelve un array, y como solo se está transformando un valor, se toma el primer (y único) valor del array.

d) Manejo de valores no encontrados:

```
else:
```

```
    print(f"El valor '{valor}' no se encuentra en la columna '{columna}'")
```

```
    return -1 # Valor por defecto si no se encuentra
```

Si el valor no se encuentra entre las clases del LabelEncoder para esa columna:

- Se imprime un mensaje de advertencia indicando que el valor no se encuentra en las clases de esa columna.
- La función devuelve -1 como un valor indicativo de error, lo que señala que no se pudo realizar la transformación.

CONCLUSIONES

Baja calidad de internet en zonas específicas: La calidad de internet en diversas localidades del Perú, especialmente en centros poblados alejados, es insuficiente para cubrir las necesidades de la población. Esto impacta negativamente en el acceso a servicios básicos como educación, salud y oportunidades económicas.

Desbalance entre población y estaciones base: Existe una disparidad notable entre el tamaño de la población y la cantidad de estaciones base disponibles. En muchas áreas, la infraestructura actual no es capaz de satisfacer la creciente demanda de conectividad.

Cobertura limitada en tecnologías avanzadas: La implementación de tecnologías como 4G y 5G es desigual, con una concentración en zonas urbanas y un acceso limitado o inexistente en áreas rurales. Esto genera una brecha digital que perpetúa las desigualdades socioeconómicas.

Falta de inversión estratégica: Los datos sugieren una necesidad urgente de inversión en infraestructura de telecomunicaciones, especialmente en regiones con alta densidad de población pero baja disponibilidad de estaciones base.

Estrategias de mejora necesarias: Es imprescindible priorizar la instalación de estaciones base adicionales en zonas críticas y fomentar políticas públicas que incentiven la inversión privada y la cooperación entre empresas operadoras y el gobierno.

Potencial de predicción para futuras decisiones: La implementación de modelos de machine learning para predecir situaciones futuras demuestra ser una herramienta valiosa. Este enfoque permite anticipar fallas y planificar soluciones efectivas con base en datos.

Importancia de los datos y la planificación: La recopilación, análisis y visualización de datos de cobertura son esenciales para comprender las necesidades locales y diseñar propuestas de solución alineadas con la realidad de cada comunidad.

REFERENCIAS

Luis Andrés Montes Bazalar-2013

Modelo de red de acceso para poblados rurales sin servicios de telecomunicaciones en el Perú –
ProQuest

<https://www.proquest.com/openview/cc89f1f963af212cf3b043cf7628c88f/1?pq-origsite=gscholar&cbl=51922&diss=y>

Xavier Alexis Pesantes Avilés

Richard Manuel Vivanco Granda 2018

Sistema de modelamiento dinámico para el análisis del tráfico de datos de la red de un ISP

[D-CD106653.pdf](#)

