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

Euribe Zambrano Sebastian Alexis 20231173B

Evangelista Aguedo, María Fernanda 20231286ª

Jaco Malpartida Jazmín Fiorella 20234507I

Silva Vasquez Angela Antonella 20234526C

Curso:

Programación Orietada 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.

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 em 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 Clase:

Este diagrama muestra la estructura de las clases en tu proyecto. Cada clase, como AnalizadorDatos, ActualizarDatos, GenerarGraficas, PropuestasSoluciones y Poblacion, tiene atributos y métodos que definen su comportamiento. Las flechas indican las relaciones entre las clases, como herencia y utilización. Este diagrama ayuda a entender cómo se organizan y relacionan las diferentes partes de tu sistema.

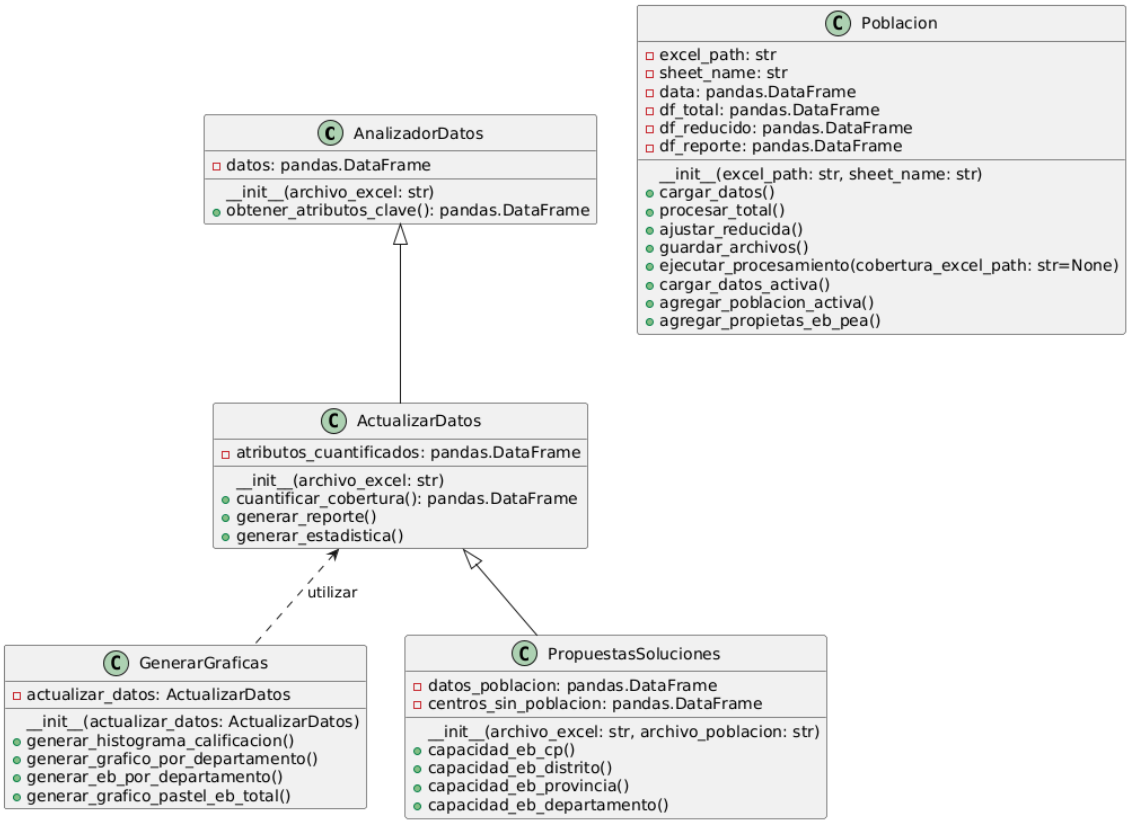


Diagrama de actividades:

El diagrama de actividad representa el flujo de trabajo del sistema. Muestra las acciones secuenciales que se realizan desde la provisión de archivos hasta el

procesamiento de datos y generación de gráficos. Este diagrama ayuda a visualizar el proceso completo y las interacciones entre diferentes etapas del sistema.

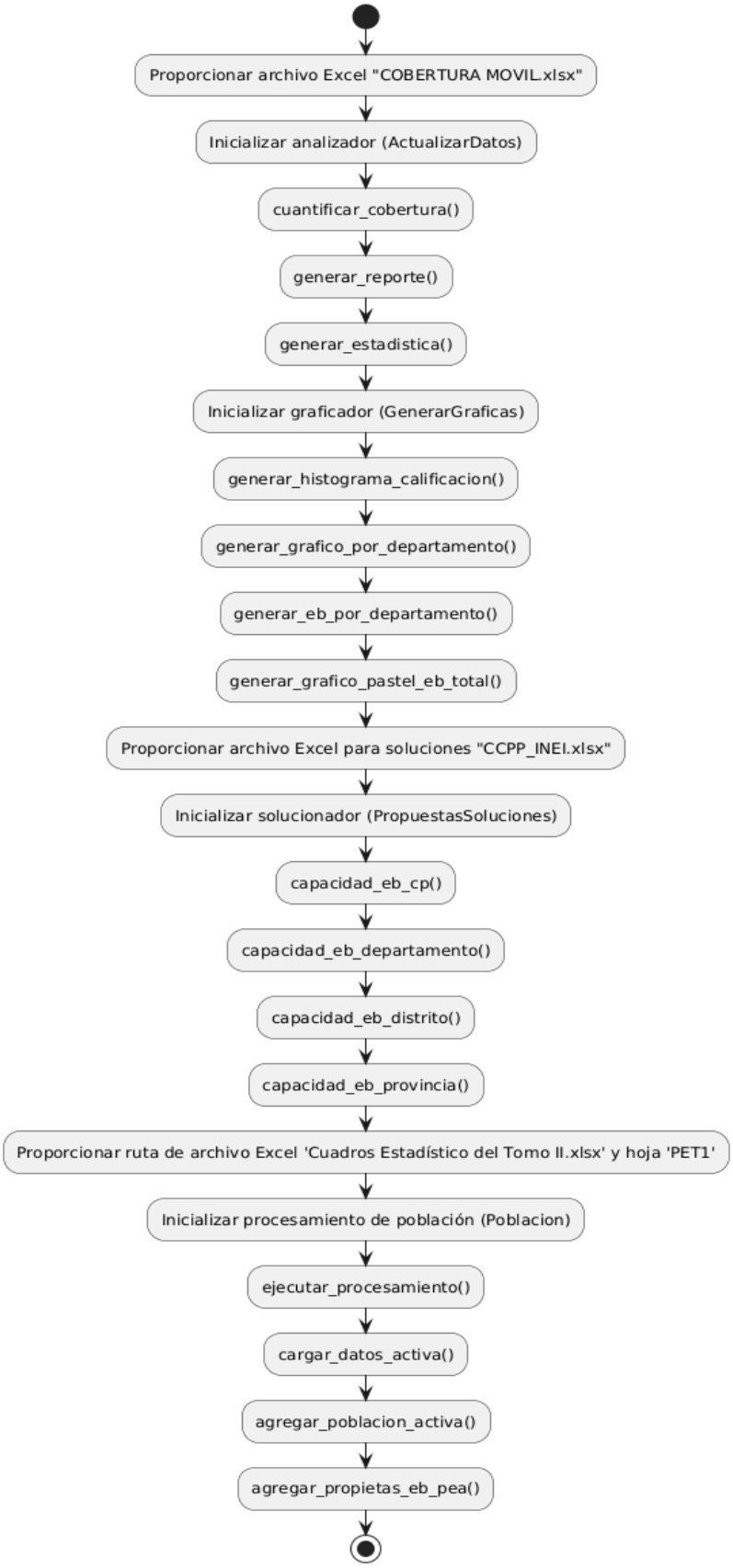


Diagrama de secuencia:

El diagrama de secuencia ilustra las interacciones temporales entre los objetos en tu sistema. Muestra cómo el usuario inicia acciones que involucran diferentes clases, como ActualizarDatos, GenerarGraficas, PropuestasSoluciones, y Poblacion. Las flechas horizontales representan los mensajes enviados entre objetos, y los bloques activados indican las llamadas de métodos. Este diagrama es útil para entender el orden de las operaciones y cómo se comunican las clases entre sí.

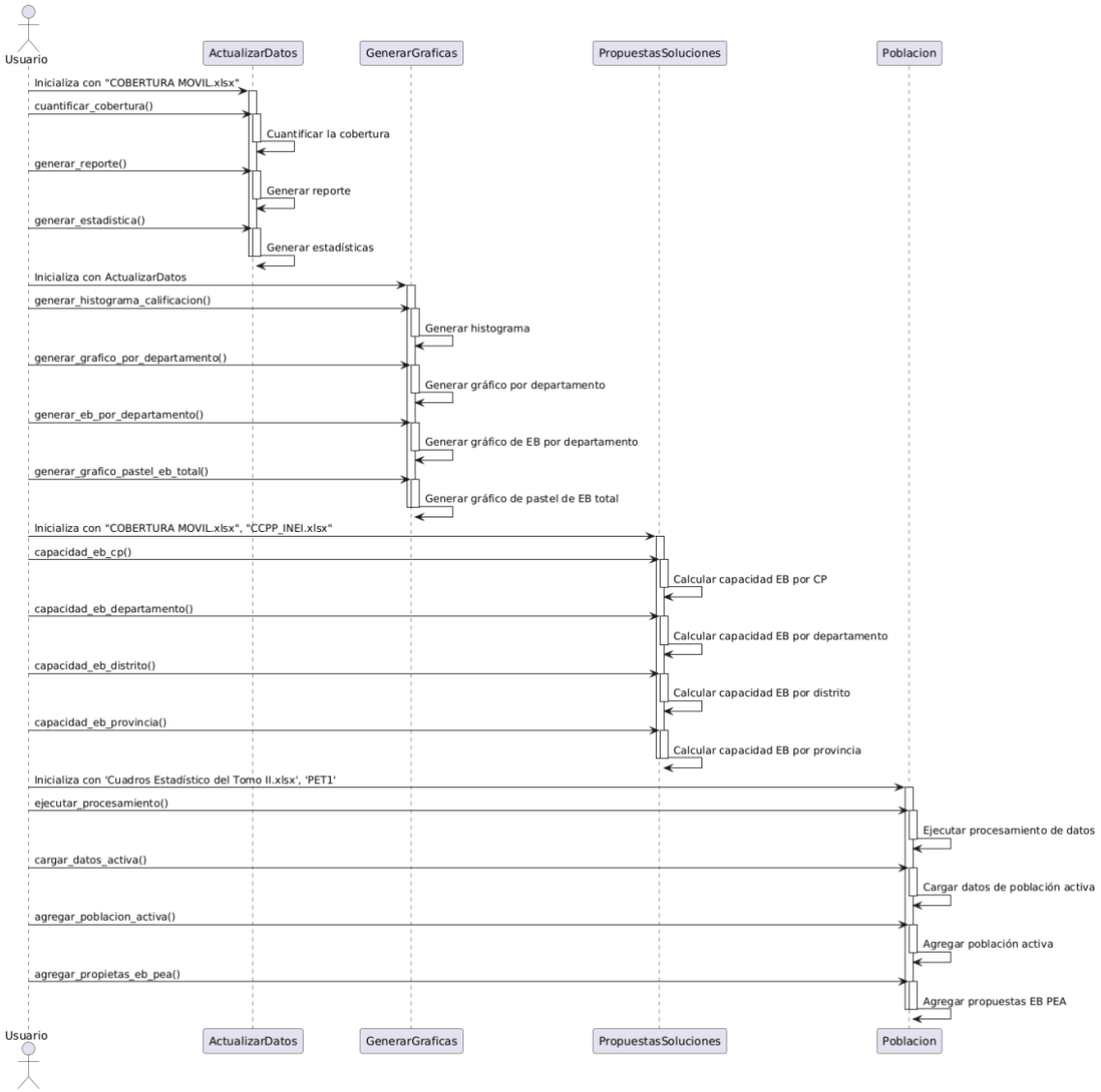


Diagrama de componentes:

El diagrama de componentes muestra la organización de los diferentes módulos del sistema. Cada componente, como ActualizarDatos y GenerarGraficas, está representado como un bloque y las flechas indican las dependencias entre ellos. Este diagrama destaca la estructura modular del sistema, facilitando la

comprensión de cómo se pueden añadir o modificar componentes sin afectar otras partes del sistema.

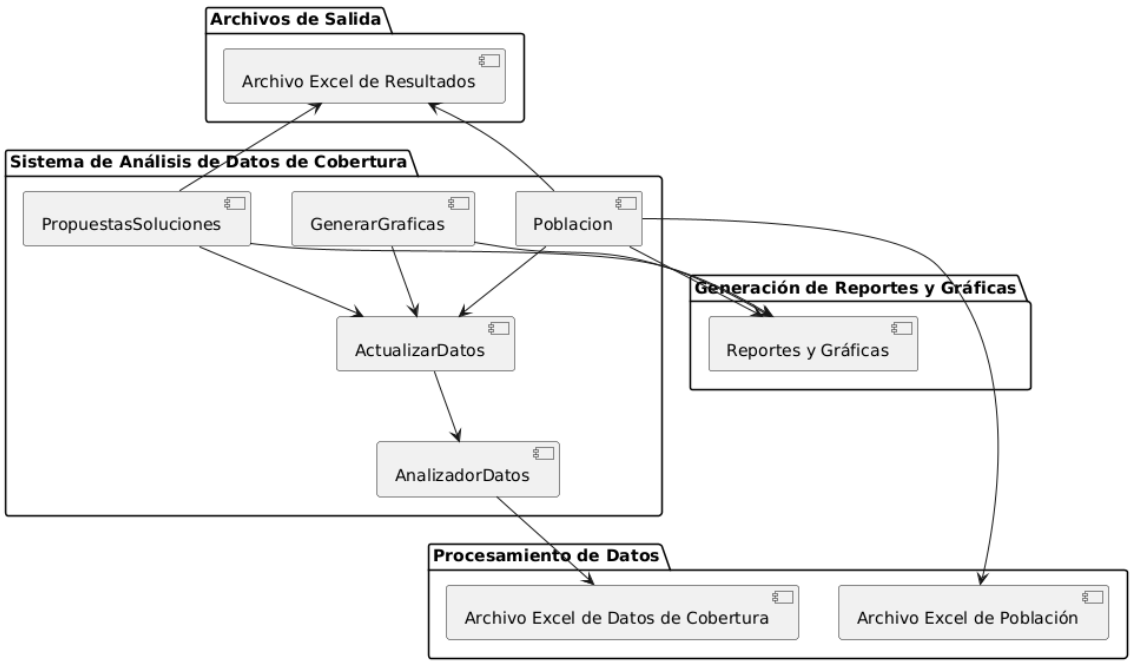


Diagrama de casos de uso:

El diagrama de casos de uso muestra las interacciones entre el usuario y las funcionalidades del sistema. Cada caso de uso, como "Cuantificar cobertura" o "Generar reporte", representa una funcionalidad específica. Las líneas conectan al usuario con los casos de uso que pueden ejecutar. Este diagrama es útil para entender qué funciones están disponibles para los usuarios y cómo interactúan con el sistema.

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.

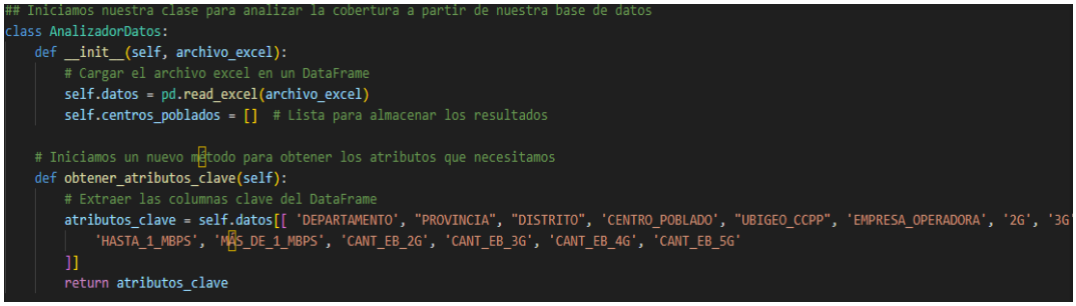
```
import pandas as pd # para hacer un DataFrame
import openpyxl # Para utilizar el excel deirectamente
import matplotlib.pyplot as plt # para graficar
import os #
import unicodedata
import numpy as np # para arreglos
import tensorflow as tf # Para el aprendizaje automatico  yredes neuronales
from tensorflow.keras.models 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
```

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 que necesitamos
    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', 'MÁS_DE_1_MBPS', 'CANT_EB_2G', 'CANT_EB_3G', 'CANT_EB_4G', 'CANT_EB_5G'
        ]]
        return atributos_clave
```

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)

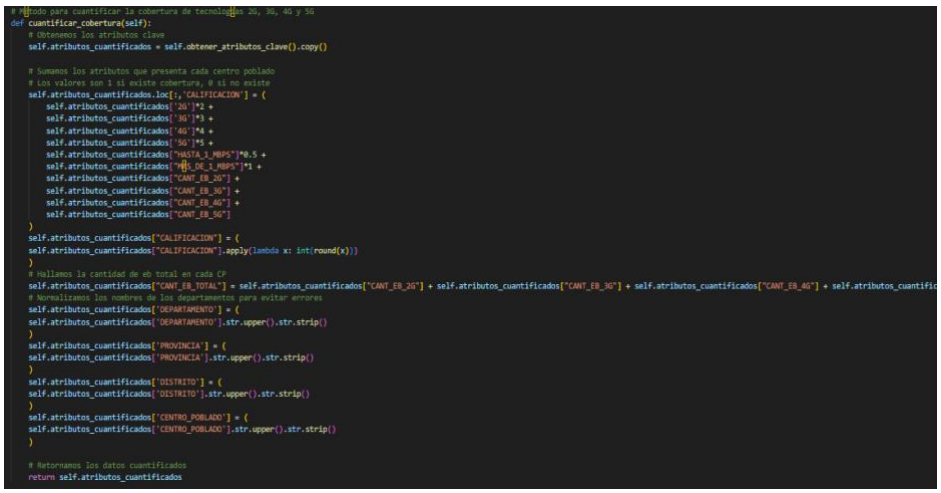
```python
# Creamos una clase que herede la anterior
class ActualizarDatos(AnalizadorDatos):
    def __init__(self, archivo_excel):
        # Inicializamos la clase base
        super().__init__(archivo_excel)
        self.atributos_cuantificados = None  # Atributo para almacenar los datos cuantificados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

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.



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["MÁS_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"]
)
```

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)))
```

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"]
```

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()
)
```

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}")
```

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}")
```

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
```

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)
        #Titulo 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}")
```

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 ycon 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_grafico_por_departamento(self)

```
def generar_grafico_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}")
```

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 DEPARTAMENTOy calcula la media de las calificaciones.

Creación del gráfico:

- plt.figure(figsize=(10, 6)): Defina 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 xpara 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}")
```

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:

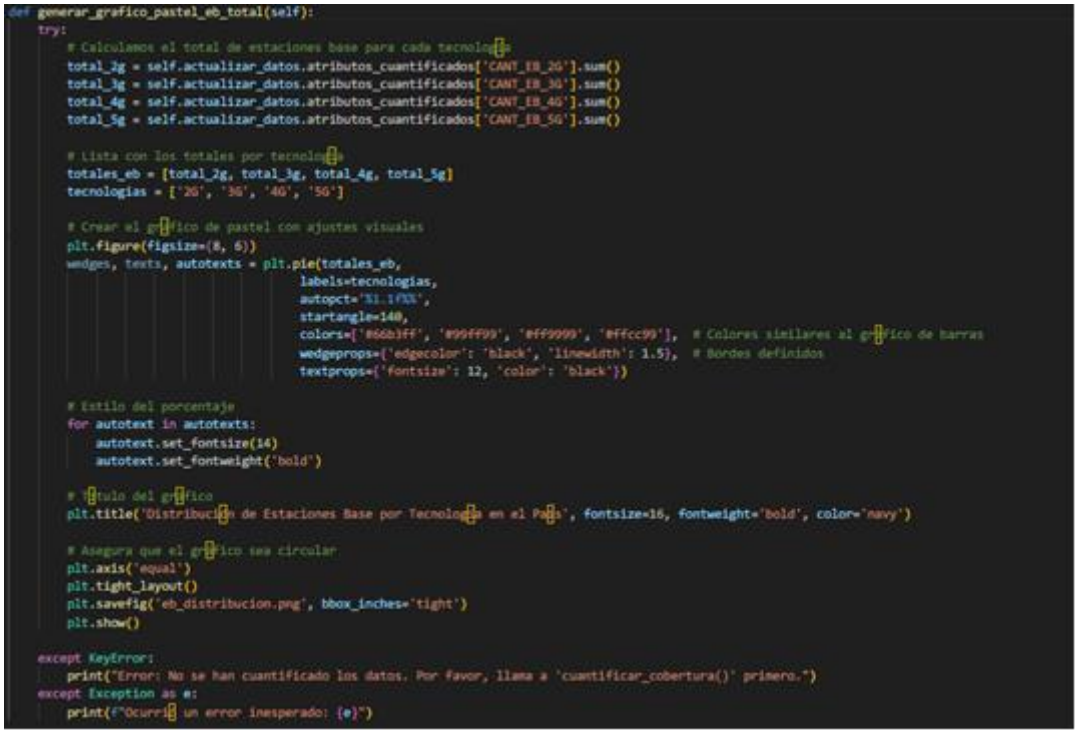
- plt.xticks(rotation=90): Rota las etiquetas del eje x90 grados para mejor legibilidad.
- plt.tight_layout(): Ajusta los márgenes para evitar que las etiquetas se corten.
- plt.savefig: Mira el gráfico como total_eb_departamento.png.
- plt.show: Muestra el gráfico en pantalla.

b) Manejo de errores:

KeyError: Si la columna CANT_EB_TOTALno 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='%1.1f%%',
                                           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}")
```

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, 5Ga cada segmento.
- autopct='%1.1f%%': 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 ActualizarDatos y agrega funcionalidad para analizar la cobertura móvil, considerando la población de cada centro poblado, y genera un informe con propuestas de mejora.

Constructor __init__(self, archivo_excel, archivo_poblacion):

```python
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 está vacío ser 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"]]

        # Renombramos las columnas en el DataFrame de población.
        # 'inplace=True' asegura que los cambios se apliquen directamente sobre el DataFrame original,
        # sin necesidad de crear una copia del mismo.
        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)
```

a) Carga y validación de datos:

```python
    # 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:
        # Realiza una combinación entre el DataFrame 'atributos_cuantificados' y 'datos_poblacion'
        # usando las columnas 'DEPARTAMENTO', 'PROVINCIA', 'DISTRITO' y 'CENTRO_POBLADO' como claves.
        # Solo se incluyen filas con coincidencias en ambas tablas (unión 'inner').
        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'
        # El resultado se actualiza en un archivo Excel sin incluir el índice.
        self.atributos_cuantificados.to_excel(reporte_cuantificado , index=False)
        print(f"Reporte con la población generado: {reporte_cuantificado}")
        if self.atributos_cuantificados.empty:
            print("Error: La fusión resultó en un DataFrame vacío.")
            return

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

        # Ahora filtramos los datos para trabajar con los que tienen poblacion
        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}")
```

- **Archivo Excel de población:**
    - Se carga con pd.read_excel(archivo_poblacion) y se seleccionan columnas específicas como Departamento, Provincia, etc.
    - Renombra estas columnas para uniformidad y rellena valores nulos en la columna POBLACION con 0 usando fillna(0).
- **Archivo Excel de cobertura móvil:**
    - Se carga en self.atributos_cuantificados.
- **Validación:**
    - Si los datos están vacíos (empty o None), se genera un error (ValueError).

b) Unión de datos:

Utiliza pd.merge para combinar datos de cobertura y población según las columnas comunes (DEPARTAMENTO, PROVINCIA, etc.). El método how="inner" asegura que solo las filas coincidentes en ambos conjuntos se incluyan en el resultado.

- **Exportación:**
    - El resultado se guarda en REPORTE_CUANTIFICADO.xlsx.

c) Verificación de datos faltantes:

- Identifica centros poblados sin información de población (POBLACION = 0) y exporta este subconjunto a CCPP_SIN_INFORMACION.xlsx.
- Filtra datos para que solo incluyan aquellos con población (POBLACION != 0).

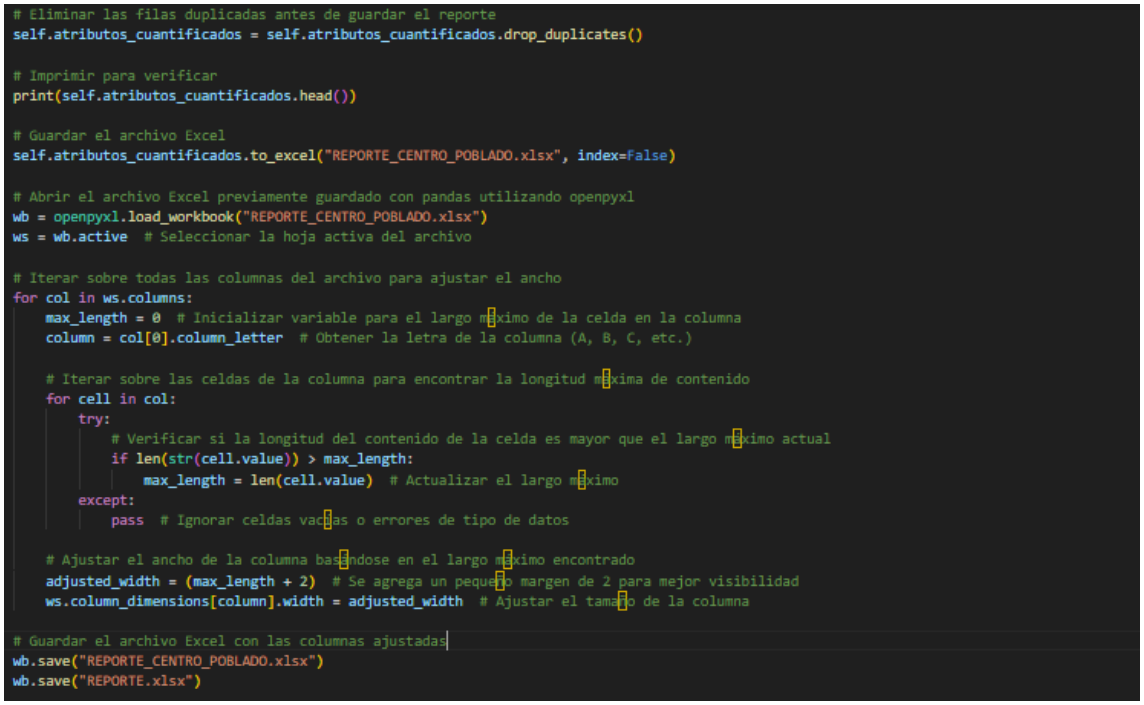
d) Manejo de errores:

- KeyError: Se captura si faltan columnas esperadas.
- Otras excepciones: Se capturan errores generales, mostrando un mensaje descriptivo.

Método capacidad_eb_cp(self):

```python
def capacidad_eb_cp(self):

    # Calcula la cantidad de habitantes por antena en cada centro poblado,
    # dividiendo la población entre el número total de estaciones base.
    # Los valores infinitos (por división entre cero) y los valores NaN se reemplazan por 0.
    self.atributos_cuantificados["ALCANCE_EB"] = (
        self.atributos_cuantificados['POBLACION'] / self.atributos_cuantificados['CANT_EB_TOTAL']
    ).replace([float('inf'), float('nan')], 0)

    # Redondea los valores de la columna 'ALCANCE_EB' y los convierte a enteros
    self.atributos_cuantificados["ALCANCE_EB"] = (
        self.atributos_cuantificados["ALCANCE_EB"].apply(lambda x: int(round(x)))
    )

    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)))
    )

    # Si la población cubierta es mayor o igual a la población, asignar 0 a POBLACION_NO_CUBIERTA
    # Usamos ".loc" para acceder a un df cuando cumple ciertas condiciones
    self.atributos_cuantificados.loc[
        self.atributos_cuantificados["POBLACION_CUBIERTA"] >= self.atributos_cuantificados["POBLACION"],
        "POBLACION_NO_CUBIERTA"
    ] = 0

    # Si la población cubierta es menor que la población, calcular la diferencia y reemplazar valores inf y NaN por 0
    self.atributos_cuantificados.loc[
        self.atributos_cuantificados["POBLACION_CUBIERTA"] < self.atributos_cuantificados["POBLACION"],
        "POBLACION_NO_CUBIERTA"
    ] = (
        self.atributos_cuantificados['POBLACION'] - self.atributos_cuantificados["POBLACION_CUBIERTA"]
    ).replace([float('inf'), float('nan')], 0)

    self.atributos_cuantificados["EB_NECESARIAS"] = self.atributos_cuantificados["POBLACION"] / 150
    # Asignar 1 a valores menores que 1 y mayores que 0
    self.atributos_cuantificados.loc[
        (self.atributos_cuantificados["EB_NECESARIAS"] < 1) & (self.atributos_cuantificados["EB_NECESARIAS"] > 0),
        "EB_NECESARIAS"
    ] = 1
```

```python
# Redondeo de los valores y convertir a entero
# "np.round" es una función de numpy que redondea al valor más cercano
# "astype(int)" esto convierte a entero el valor asegurando que no sea de tipo flotante
self.atributos_cuantificados["EB_NECESARIAS"] = np.round(self.atributos_cuantificados["EB_NECESARIAS"]).astype(int)

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

# Iterar sobre cada fila para evaluar si el centro poblado necesita más estaciones base
for index, row in self.atributos_cuantificados.iterrows():
    # Caso 1: Sin datos de población
    if row["POBLACION"] == 0:
        self.atributos_cuantificados.at[index, "PROPUESTA_EB"] = (
            f"No hay registros de población en el centro poblado de {row['CENTRO_POBLADO']}, se recomienda obtener datos sobre la población para poder realizar un mejor análisis"
        )

    # Caso 2: Población mayor a 150
    elif row["POBLACION"] > 150:
        if row["ALCANCE_EB"] <= 150 and row["ALCANCE_EB"] > 0:
            self.atributos_cuantificados.at[index, "PROPUESTA_EB"] = (
                f"La cantidad de EB ({row['CANT_EB_TOTAL']}) es suficiente y óptima para la población en el centro poblado de {row['DEPARTAMENTO']}."
            )
        else:
            self.atributos_cuantificados.at[index, "PROPUESTA_EB"] = (
                f"El alcance de las estaciones base en el centro poblado de {row['CENTRO_POBLADO']} es superior a 150, por lo que se recomienda aumentar la cantidad de EB a {row['EB_NECESARIAS']} para mejor
            )

    # Caso 3: Población menor o igual a 150
    elif row["POBLACION"] <= 150:
        if row["CANT_EB_TOTAL"] == 0:
            self.atributos_cuantificados.at[index, "PROPUESTA_EB"] = (
                f"La población en el centro poblado de {row['CENTRO_POBLADO']} es muy baja, por lo que se recomienda instalar un repetidor para compartir cobertura con otro lugar. No obstante, si se desea e
            )
        else:
            self.atributos_cuantificados.at[index, "PROPUESTA_EB"] = (
                f"La cantidad de EB ({row['CANT_EB_TOTAL']}) es suficiente y óptima para la población en el centro poblado de {row['CENTRO_POBLADO']}."
            )
```



```python
# Eliminar las filas duplicadas antes de guardar el reporte
self.atributos_cuantificados = self.atributos_cuantificados.drop_duplicates()

# Imprimir para verificar
print(self.atributos_cuantificados.head())

# Guardar el archivo Excel
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

# Iterar sobre todas las columnas del archivo para ajustar el ancho
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(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_CENTRO_POBLADO.xlsx")
wb.save("REPORTE.xlsx")
```

a) Cálculo del alcance de las estaciones base (ALCANCE_EB):

- **Fórmula:** Población total del centro poblado (POBLACION) dividida entre el número de estaciones base (CANT_EB_TOTAL).
- Si hay divisiones por 0 (infinitos) o valores nulos (nan), se reemplazan por 0 para evitar errores en el análisis.
- El valor se redondea al entero más cercano.

b) Cálculo de la población cubierta (POBLACION_CUBIERTA):

- **Suposición:** Cada estación base cubre un máximo de 150 personas.
- **Fórmula:** POBLACION_CUBIERTA = CANT_EB_TOTAL * 150.
- Redondea los resultados y reemplaza valores infinitos o nulos por 0.

c) Cálculo de la población no cubierta (POBLACION_NO_CUBIERTA):

- **Evalúa cada centro poblado:**
    - Si la población cubierta es mayor o igual a la total: POBLACION_NO_CUBIERTA = 0.
    - **Si es menor:** Se calcula la diferencia (POBLACION - POBLACION_CUBIERTA), asegurando que no haya valores infinitos o nulos.

d) Cálculo de EB necesarias (EB_NECESARIAS):

- **Fórmula:** Población total dividida entre 150.

- Asigna al menos 1 EB si el valor es menor que 1 pero mayor que 0.
- Redondea los valores al entero más cercano.

e) Cálculo de EB faltantes (EB_FALTANTES):

- **Fórmula:** EB_FALTANTES = EB_NECESARIAS - CANT_EB_TOTAL.
- Reemplaza valores infinitos o nulos por 0 para mantener consistencia.

f) Generación de propuestas por centro poblado:

- **Itera fila por fila para determinar recomendaciones:**
    - **Sin datos de población:** Sugiere recopilar información antes de analizar.
    - **Población mayor a 150:**
        - Si las EB actuales son suficientes y cubren eficientemente la población, indica que no se necesitan cambios.
        - Si no es suficiente, sugiere incrementar las EB al número calculado.
    - **Población menor o igual a 150:**
        - Si no hay EB, propone instalar un repetidor o, opcionalmente, una EB adicional.
        - Si ya hay EB, confirma que la cobertura es adecuada.

g) Eliminación de duplicados:

- Elimina filas repetidas para evitar redundancias en el reporte.

h) Generación del reporte Excel

- Guarda el lDataFrame final en un archivo llamado REPORTE_CENTRO_POBLADO.xlsx.

i) Ajuste del formato del archivo Excel

- Utiliza openpyxl para ajustar el ancho de cada columna basado en la longitud máxima de las celdas, asegurando legibilidad.
- Guarda el archivo con el formato ajustado.

Método capacidad_eb_distrito(self):

```python
def capacidad_eb_distrito(self):
    # Creamos un nuevo dataframe
    self.reporte_distrito = self.atributos_cuantificados.copy()

    # Calificación promedio por distrito
    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)))

    # Sumar la cantidad de estaciones base por tecnología
    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")

    # Sumar la población total por distrito
    self.reporte_distrito["POBLACION_TOTAL"] = self.atributos_cuantificados.groupby("DISTRITO")["POBLACION"].transform("sum")

    # Calcular habitantes por antena en cada centro poblado
    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)))

    # Calcular población cubierta (asumimos 150 personas por cada antena base)
    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)))

    # Si la población cubierta es mayor o igual que la población total, asignar 0 a población no cubierta
    self.reporte_distrito.loc[self.reporte_distrito["POBLACION_CUBIERTA"] >= self.reporte_distrito["POBLACION_TOTAL"], "POBLACION_NO_CUBIERTA"] = 0

    # Si la población cubierta es menor que la población total, calcular la diferencia
    self.reporte_distrito.loc[self.reporte_distrito["POBLACION_CUBIERTA"] < self.reporte_distrito["POBLACION_TOTAL"], "POBLACION_NO_CUBIERTA"] = (self.reporte_distrito['PO

    # Calcular el número de estaciones base necesarias para cubrir la población total
    self.reporte_distrito["EB_NECESARIAS"] = self.reporte_distrito['POBLACION_TOTAL'] / 150

    # Asignar 1 a valores menores que 1 y mayores que 0
    self.reporte_distrito.loc[(self.reporte_distrito["EB_NECESARIAS"] < 1) & (self.reporte_distrito["EB_NECESARIAS"] > 0), "EB_NECESARIAS"] = 1

    # Redondeo de los valores y convertir a entero
    self.reporte_distrito["EB_NECESARIAS"] = np.round(self.reporte_distrito["EB_NECESARIAS"]).astype(int)

    # Calcular las estaciones base faltantes
    self.reporte_distrito["EB_FALTANTES"] = (self.reporte_distrito["EB_NECESARIAS"] - self.reporte_distrito["CANT_EB_TOTAL"]).replace([float('inf'), float('nan')], 0)
```

```python
# Eliminar las columnas innecesarias
columnas_a_eliminar = ['CENTRO_POBLADO', 'UBIGEO_CCPP', 'EMPRESA_OPERADORA', '2G', '3G', '4G', '5G', 'HASTA_1_MBPS', 'MÁS_DE_1_MBPS', 'POBLACION', "PROPUESTA_EB"]
self.reporte_distrito.drop(columnas_a_eliminar, axis=1, inplace=True)

# Eliminar filas duplicadas
self.reporte_distrito = self.reporte_distrito.drop_duplicates()

# Inicializar la columna "PROPUESTA_EB"
self.reporte_distrito["PROPUESTA_EB"] = None

# Evaluar las propuestas de estaciones base
for index, row in self.reporte_distrito.iterrows():
    # Caso 1: Sin datos de población
    if row["POBLACION_TOTAL"] == 0:
        self.reporte_distrito.at[index, "PROPUESTA_EB"] = (
            f"No hay registros de población en el distrito de {row['DISTRITO']}, se recomienda obtener datos sobre la población para poder realizar un mejor análisis"
        )

    # Caso 2: Población mayor a 150
    elif row["POBLACION_TOTAL"] > 150:
        if row["ALCANCE_EB"] <= 150 and row["ALCANCE_EB"] > 0:
            self.reporte_distrito.at[index, "PROPUESTA_EB"] = (
                f"La cantidad de EB ({row['CANT_EB_TOTAL']}) es suficiente y óptima para la población en el distrito de {row['DISTRITO']}."
            )
        else:
            self.reporte_distrito.at[index, "PROPUESTA_EB"] = (
                f"El alcance de las estaciones base en el distrito de {row['DISTRITO']} es superior a 150, por lo que se recomienda aumentar la cantidad de EB a {row['EB_NECESARIAS']} para mejorar la c
            )

    # Caso 3: Población menor o igual a 150
    elif row["POBLACION_TOTAL"] <= 150:
        if row["CANT_EB_TOTAL"] == 0:
            self.reporte_distrito.at[index, "PROPUESTA_EB"] = (
                f"La población en el distrito de {row['DISTRITO']} es muy baja, por lo que se recomienda instalar un repetidor para compartir cobertura con otro lugar. No obstante, si se desea evitar l
            )
        else:
            self.reporte_distrito.at[index, "PROPUESTA_EB"] = (
                f"La cantidad de EB ({row['CANT_EB_TOTAL']}) es suficiente y óptima para la población en el distrito de {row['DISTRITO']}."
            )
```

```python
# Eliminar filas duplicadas
self.reporte_distrito = self.reporte_distrito.drop_duplicates()

# Imprimir para verificar
print(self.reporte_distrito.head())

# Guardar el reporte a Excel
self.reporte_distrito.to_excel("REPORTE_DISTRITO.xlsx", index=False)

# Ajustar el tamaño de las columnas del Excel
wb = openpyxl.load_workbook("REPORTE_DISTRITO.xlsx")
ws = wb.active
for col in ws.columns:
    max_length = 0
    column = col[0].column_letter
    for cell in col:
        try:
            if len(str(cell.value)) > max_length:
                max_length = len(cell.value)
        except:
            pass
    adjusted_width = (max_length + 2)
    ws.column_dimensions[column].width = adjusted_width

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

a) Preparación inicial:

- **copy():** Crea una copia del DataFrame original (atributos_cuantificados). Esto es importante para no modificar los datos base, permitiendo un análisis independiente.

b) Calcular calificación promedio por distrito:

- **groupby('DISTRITO'):** Agrupa los datos por el campo DISTRITO.
- **transform('mean'):** Calcula el promedio de la columna CALIFICACION para cada grupo (distrito).
- **apply(lambda x: int(round(x))):** Redondea el promedio al entero más cercano y lo convierte a un número entero.

c) Sumar estaciones base por tecnología:

- **groupby("DISTRITO"):** Agrupa los datos por distrito.
- **transform("sum"):** Suma la cantidad total de estaciones base 2G por distrito.
- Repite este proceso para 3G, 4G, 5G y el total de estaciones base (CANT_EB_TOTAL).

d) Calcular población total:

- Suma la población de todos los centros poblados de un distrito.

e) Habitantes por estación base:

- **/:** Divide la población total entre el total de estaciones base disponibles.
- **replace([float('inf'), float('nan')], 0):** Reemplaza valores no definidos o infinitos por 0.

- **apply(lambda x: int(round(x))):** Redondea el resultado y lo convierte a un número entero.

f) Población cubierta y no cubierta:

Población cubierta:

- Supone que cada estación base cubre 150 personas.
- Multiplica el total de estaciones base por 150.
- Aplica reemplazos y redondeo como en pasos anteriores.

Población no cubierta:

- **loc:** Condiciona los valores de las filas según criterios.
- Si la población cubierta es mayor o igual a la total, la población no cubierta es 0.
- En caso contrario, calcula la diferencia entre la población total y la cubierta.

g) Estaciones base necesarias y faltantes:

Estaciones base necesarias:

- Calcula el número de estaciones base requeridas para cubrir la población total.
- Ajusta el valor mínimo a 1 si el resultado es menor a 1 pero mayor a 0.
- **np.round:** Redondea el resultado y lo convierte en entero.

Estaciones base faltantes:

- Calcula la diferencia entre estaciones base necesarias y existentes.

h) Generar recomendaciones:

- **iterrows():** Itera fila por fila en el DataFrame.
- **at:** Asigna un valor a una celda específica en función de las condiciones evaluadas.

i) Generación del archivo Excel:

- **to_excel():** Exporta el DataFrame a un archivo Excel.

j) Ajuste de columnas:

- **load_workbook():** Carga el archivo Excel generado.
- **column_dimensions:** Ajusta el ancho de las columnas en función del contenido.

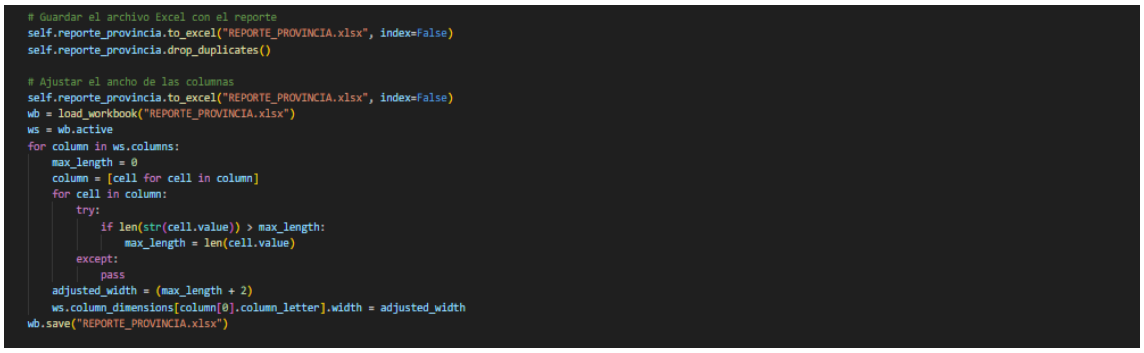
Método capacidad_eb_provincia(self):

```python
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")

    # 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)))
    )

    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)))
    )

    self.reporte_provincia.loc[
        self.reporte_provincia["POBLACION_CUBIERTA"] >= self.reporte_provincia["POBLACION_TOTAL"],
        "POBLACION_NO_CUBIERTA"
    ] = 0

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



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

    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)))
    )

    self.reporte_provincia.loc[
        self.reporte_provincia["POBLACION_CUBIERTA"] >= self.reporte_provincia["POBLACION_TOTAL"],
        "POBLACION_NO_CUBIERTA"
    ] = 0

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

    self.reporte_provincia["EB_NECESARIAS"] = self.reporte_provincia['POBLACION_TOTAL'] / 150

    # Asignar 1 a valores menores que 1 y mayores que 0
    self.reporte_provincia.loc[
        (self.reporte_provincia["EB_NECESARIAS"] < 1) & (self.reporte_provincia["EB_NECESARIAS"] > 0),
        "EB_NECESARIAS"
    ] = 1

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

```
# Eliminar columnas innecesarias
columnas_a_eliminar = [
    'DISTRITO', 'CENTRO_POBLADO', 'UBIGEO_CCPP',
    'EMPRESA_OPERADORA', '2G', '3G', '4G', '5G', 'HASTA_1_MBPS', 'MÁS_DE_1_MBPS', 'POBLACION', "PROPUESTA_EB"
]
self.reporte_provincia.drop(columnas_a_eliminar, axis=1, inplace=True)

# Eliminar filas duplicadas
self.reporte_provincia = self.reporte_provincia.drop_duplicates()
self.reporte_provincia["PROPUESTA_EB"] = None
for index, row in self.reporte_provincia.iterrows():
    # Caso 1: Sin datos de población
    if row["POBLACION_TOTAL"] == 0:
        self.reporte_provincia.at[index, "PROPUESTA_EB"] = (
            f"No hay registros de población en la provincia de {row['PROVINCIA']}, se recomienda obtener datos sobre la población para poder realizar un mejor análisis"
        )

    # Caso 2: Población mayor a 150
    elif row["POBLACION_TOTAL"] > 150:
        if row["ALCANCE_EB"] <= 150 and row["ALCANCE_EB"] > 0:
            self.reporte_provincia.at[index, "PROPUESTA_EB"] = (
                f"La cantidad de EB ({row['CANT_EB_TOTAL']}) es suficiente y óptima para la población en la provincia de {row['PROVINCIA']}."
            )
        else:
            self.reporte_provincia.at[index, "PROPUESTA_EB"] = (
                f"El alcance de las estaciones base en la provincia de {row['PROVINCIA']} es superior a 150, por lo que se recomienda aumentar la cantidad de EB a {row['EB_NECESARIAS']} para mejorar
            )

    # Caso 3: Población menor o igual a 150
    elif row["POBLACION_TOTAL"] <= 150:
        if row["CANT_EB_TOTAL"] == 0:
            self.reporte_provincia.at[index, "PROPUESTA_EB"] = (
                f"La población en la provincia de {row['PROVINCIA']} es muy baja, por lo que se recomienda instalar un repetidor para compartir cobertura con otro lugar. No obstante, si se desea evit
            )
        else:
            self.reporte_provincia.at[index, "PROPUESTA_EB"] = (
                f"La cantidad de EB ({row['CANT_EB_TOTAL']}) es suficiente y óptima para la población en la provincia de {row['PROVINCIA']}."
            )
```

```
# Guardar el archivo Excel con el reporte
self.reporte_provincia.to_excel("REPORTE_PROVINCIA.xlsx", index=False)
self.reporte_provincia.drop_duplicates()

# Ajustar el ancho de las columnas
self.reporte_provincia.to_excel("REPORTE_PROVINCIA.xlsx", index=False)
wb = load_workbook("REPORTE_PROVINCIA.xlsx")
ws = wb.active
for column in ws.columns:
    max_length = 0
    column = [cell for cell in column]
    for cell in column:
        try:
            if len(str(cell.value)) > max_length:
                max_length = len(cell.value)
        except:
            pass
    adjusted_width = (max_length + 2)
    ws.column_dimensions[column[0].column_letter].width = adjusted_width
wb.save("REPORTE_PROVINCIA.xlsx")
```

a) Creación de un DataFrame base:

- Crea una copia del DataFrame atributos_cuantificados, asegurando que los datos originales no se modifiquen durante el análisis.

b) Calcular calificación promedio por provincia:

- **Agrupación por provincia (groupby):** Calcula la calificación promedio de calidad de las conexiones móviles.
- **Redondeo (apply con lambda):** Convierte los promedios en números enteros para facilitar la interpretación.

c) Sumar estaciones base por tecnología:

- **Suma total por provincia:** Calcula la cantidad de estaciones base (2G, 3G, 4G, 5G y total) disponibles en cada provincia.

d) Calcular población total y métricas relacionadas:

Población total:

- Suma la población de todos los centros poblados dentro de cada provincia.

Habitantes por EB:

- Divide la población total entre las estaciones base disponibles para estimar el alcance promedio por EB.
- Los valores indefinidos (nan, infinito) se reemplazan por 0, y se redondean a enteros.

Población cubierta:

- Se estima que cada estación base cubre hasta 150 personas.
- Calcula la población que potencialmente tiene acceso a las EB disponibles.

Población no cubierta:

- Determina la población sin acceso adecuado a las EB.
- Si la población cubierta es mayor o igual a la total, no hay población no cubierta; de lo contrario, calcula la diferencia.

e) Necesidades de estaciones base:

EB necesarias:

- Calcula cuántas EB son necesarias para cubrir completamente la población de cada provincia.
- Asegura que provincias con poblaciones muy pequeñas (pero mayores a 0) tengan al menos 1 EB.

EB faltantes:

- Calcula cuántas EB adicionales hacen falta para cubrir las necesidades estimadas.

f) Generación de recomendaciones:

Iteración por provincia:

- Genera propuestas personalizadas basadas en:
    1. Provincias sin datos de población.
    2. Provincias con gran población pero cobertura insuficiente.
    3. Provincias pequeñas con recomendaciones específicas.

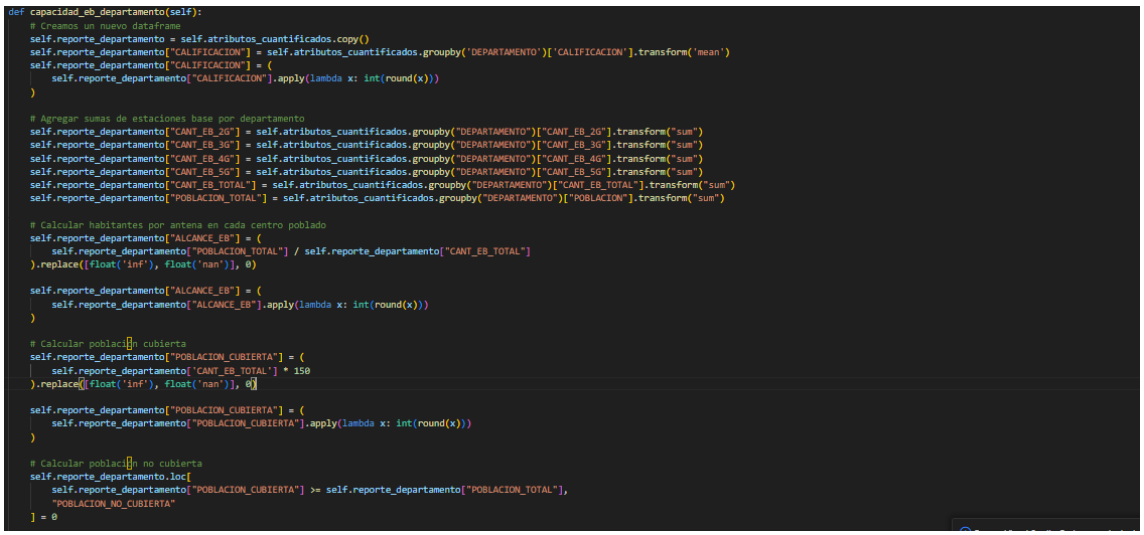
g) Exportación del reporte:

- Guarda el DataFrame en un archivo Excel para su análisis.

h) Ajuste de columnas:

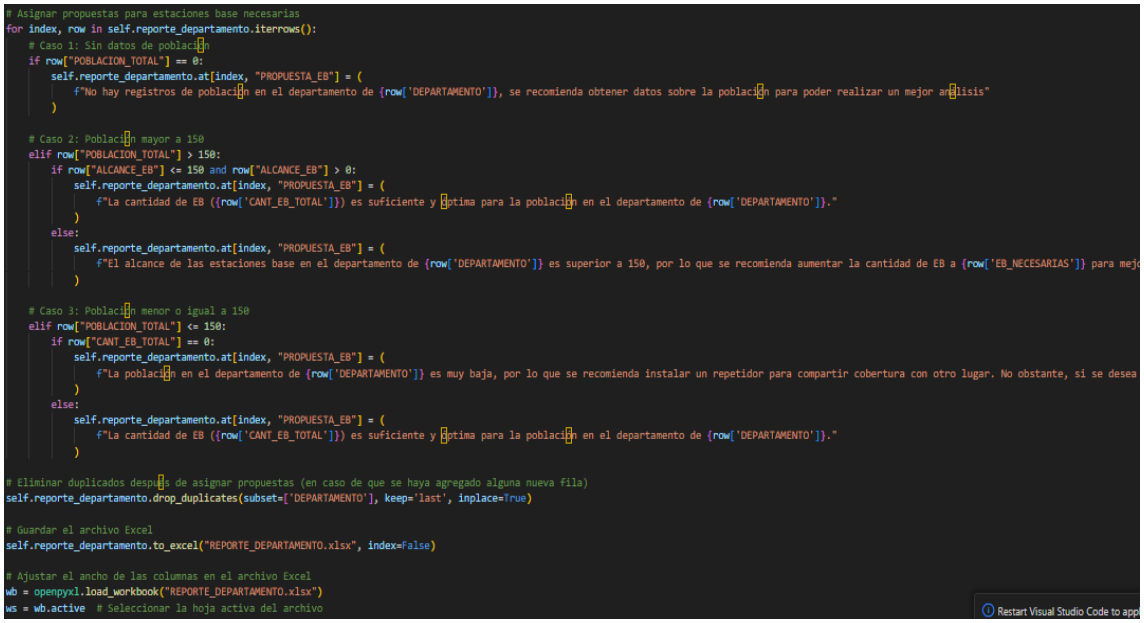
- Ajusta automáticamente el ancho de las columnas según el contenido para mejorar la legibilidad.

Método capacidad_eb_departamento(self):

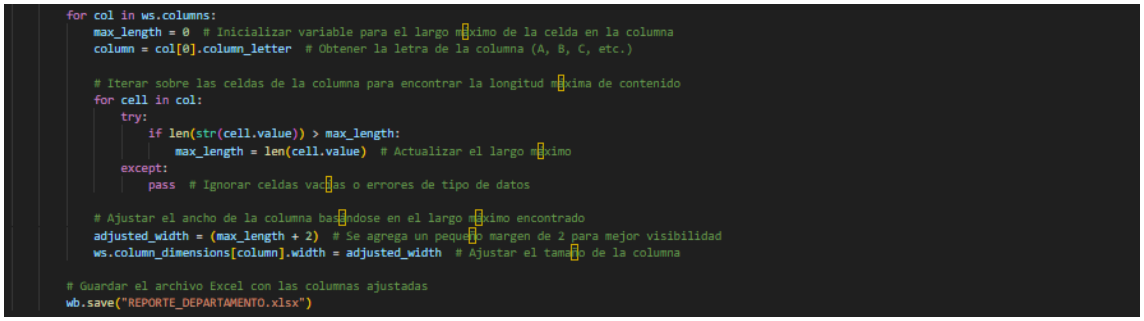


```
def capacidad_eb_departamento(self):
    # Creamos un nuevo dataframe
    self.reporte_departamento = self.atributos_cuantificados.copy()
    self.reporte_departamento["CALIFICACION"] = self.atributos_cuantificados.groupby('DEPARTAMENTO')['CALIFICACION'].transform('mean')
    self.reporte_departamento["CALIFICACION"] = (
        self.reporte_departamento["CALIFICACION"].apply(lambda x: int(round(x)))
    )

    # Agregar sumas de estaciones base por departamento
    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")

    # 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)))
    )

    # Calcular población cubierta
    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)))
    )

    # Calcular población no cubierta
    self.reporte_departamento.loc[
        self.reporte_departamento["POBLACION_CUBIERTA"] >= self.reporte_departamento["POBLACION_TOTAL"],
        "POBLACION_NO_CUBIERTA"
    ] = 0
```

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

self.reporte_departamento["EB_NECESARIAS"] = self.reporte_departamento['POBLACION_TOTAL'] / 150
self.reporte_departamento.loc[
    (self.reporte_departamento["EB_NECESARIAS"] < 1) & (self.reporte_departamento["EB_NECESARIAS"] > 0),
    "EB_NECESARIAS"
] = 1

# Redondeo de los valores y convertir a entero
self.reporte_departamento["EB_NECESARIAS"] = np.round(self.reporte_departamento["EB_NECESARIAS"]).astype(int)

# Calcular faltantes de EB
self.reporte_departamento["EB_FALTANTES"] = (
    self.reporte_departamento["EB_NECESARIAS"] - self.reporte_departamento["CANT_EB_TOTAL"]
).replace([float('inf'), float('nan')], 0)

# Eliminar columnas innecesarias
columnas_a_eliminar = [
    'PROVINCIA', 'DISTRITO', 'CENTRO_POBLADO', 'UBIGEO_CCPP',
    'EMPRESA_OPERADORA', '2G', '3G', '4G', '5G', 'HASTA_1_MBPS', 'MÁS_DE_1_MBPS', 'POBLACION', "PROPUESTA_EB"
]
self.reporte_departamento.drop(columnas_a_eliminar, axis=1, inplace=True)

# **Eliminar duplicados**: Eliminamos duplicados solo basados en la columna 'DEPARTAMENTO', conservando la última aparición
self.reporte_departamento.drop_duplicates(subset=['DEPARTAMENTO'], keep='last', inplace=True)
```



```
# Asignar propuestas para estaciones base necesarias
for index, row in self.reporte_departamento.iterrows():
    # Caso 1: Sin datos de población
    if row["POBLACION_TOTAL"] == 0:
        self.reporte_departamento.at[index, "PROPUESTA_EB"] = (
            f"No hay registros de población en el departamento de {row['DEPARTAMENTO']}, se recomienda obtener datos sobre la población para poder realizar un mejor análisis"
        )

    # Caso 2: Población mayor a 150
    elif row["POBLACION_TOTAL"] > 150:
        if row["ALCANCE_EB"] <= 150 and row["ALCANCE_EB"] > 0:
            self.reporte_departamento.at[index, "PROPUESTA_EB"] = (
                f"La cantidad de EB ({row['CANT_EB_TOTAL']}) es suficiente y óptima para la población en el departamento de {row['DEPARTAMENTO']}."
            )
        else:
            self.reporte_departamento.at[index, "PROPUESTA_EB"] = (
                f"El alcance de las estaciones base en el departamento de {row['DEPARTAMENTO']} es superior a 150, por lo que se recomienda aumentar la cantidad de EB a {row['EB_NECESARIAS']} para mej
            )

    # Caso 3: Población menor o igual a 150
    elif row["POBLACION_TOTAL"] <= 150:
        if row["CANT_EB_TOTAL"] == 0:
            self.reporte_departamento.at[index, "PROPUESTA_EB"] = (
                f"La población en el departamento de {row['DEPARTAMENTO']} es muy baja, por lo que se recomienda instalar un repetidor para compartir cobertura con otro lugar. No obstante, si se desea
            )
        else:
            self.reporte_departamento.at[index, "PROPUESTA_EB"] = (
                f"La cantidad de EB ({row['CANT_EB_TOTAL']}) es suficiente y óptima para la población en el departamento de {row['DEPARTAMENTO']}."
            )

# Eliminar duplicados después de asignar propuestas (en caso de que se haya agregado alguna nueva fila)
self.reporte_departamento.drop_duplicates(subset=['DEPARTAMENTO'], keep='last', inplace=True)

# Guardar el archivo Excel
self.reporte_departamento.to_excel("REPORTE_DEPARTAMENTO.xlsx", index=False)

# Ajustar el ancho de las columnas en el archivo Excel
wb = openpyxl.load_workbook("REPORTE_DEPARTAMENTO.xlsx")
ws = wb.active  # Seleccionar la hoja activa 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:
            if len(str(cell.value)) > max_length:
                max_length = len(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_DEPARTAMENTO.xlsx")
```

a) Creación de un DataFrame para el análisis:

- Se crea una copia del DataFrame original atributos_cuantificados para evitar modificaciones no deseadas en los datos originales.

b) Calificación promedio por departamento:

- **Agrupación por departamento (groupby):** Calcula la calificación promedio de la cobertura móvil para cada departamento.
- **Redondeo y conversión a entero:** Facilita la interpretación al simplificar los valores.

c) Sumar estaciones base por tecnología:

- Se suman las estaciones base de cada tecnología móvil para obtener totales departamentales.

d) Población y alcance por EB:

Población total:

- Calcula la población total en cada departamento.

Habitantes por EB:

- Divide la población total entre el número de EB disponibles para estimar el alcance promedio de cada una.

e) Cobertura de población:

Población cubierta:

- Se asume que cada EB puede cubrir 150 personas.

Población no cubierta:

- Si la población cubierta es mayor o igual a la total, no hay población no cubierta. De lo contrario, calcula la diferencia.

f) Necesidades de estaciones base:

EB necesarias:

- Calcula cuántas EB adicionales se necesitan para cubrir la población total.

EB faltantes:

- Determina cuántas EB faltan en cada departamento.

g) Propuestas de mejora:

- Genera propuestas adaptadas a cada departamento según la población y las estaciones base disponibles.

h) Generación del archivo Excel:

Guardar y ajustar columnas:

- Exporta los datos a Excel y ajusta automáticamente el ancho de las columnas para mejorar la legibilidad.

Método quitar_tildes(texto):

```
# Función para quitar tildes
def quitar_tildes(texto):
    return ''.join((c for c in unicodedata.normalize('NFD', texto) if unicodedata.category(c) != 'Mn'))
#Esta función descompone los caracteres acentuados en su forma base y los signos diacríticos
```

a) Descomposición de caracteres con unicodedata.normalize:

- El método unicodedata.normalize('NFD', texto) descompone caracteres con signos diacríticos en una combinación de su carácter base y los acentos por separado.

b) Filtrado de signos diacríticos:

- Se utiliza una expresión generadora para recorrer cada carácter descompuesto.
- La función unicodedata.category(c) clasifica cada carácter. Los signos diacríticos, como tildes, pertenecen a la categoría 'Mn' (Mark, Nonspacing).
- Sólo se conservan caracteres cuya categoría no sea 'Mn'.

c) Reensamblado del texto:

- Se aplica ''.join() para reconstruir el texto limpio sin los signos diacríticos.

Observación: En esta clase se ha tenido que hacer un último ajuste, puesto que en la interfaz salía negativo el valor de EB_FALTANTES, se le agrego a los métodos:

- Capacidad_cp en la línea 367 a 371 (en el código)
- Capacidad_distrito en la línea 492 a 496 (en el código)

- Capacidad_provincia en la línea 623 a 627 (en el código)
- Capacidad_departamento en la línea 751 a 755 (en el código)

Lo que hemos añadido es:

En caso de que las eb necesarias sean menores a las eb que ya se encuentran instaladas

self.atributos_cuantificados.loc[

(self.atributos_cuantificados["EB_NECESARIAS"]<self.atributos_cuantificados["CANT_EB_TOTAL"]),

"EB_NECESARIAS"

] = 0

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'))
#Esta función descompone los caracteres acentuados en su forma base y los signos diacríticos
```

función quitar_tildes(texto):

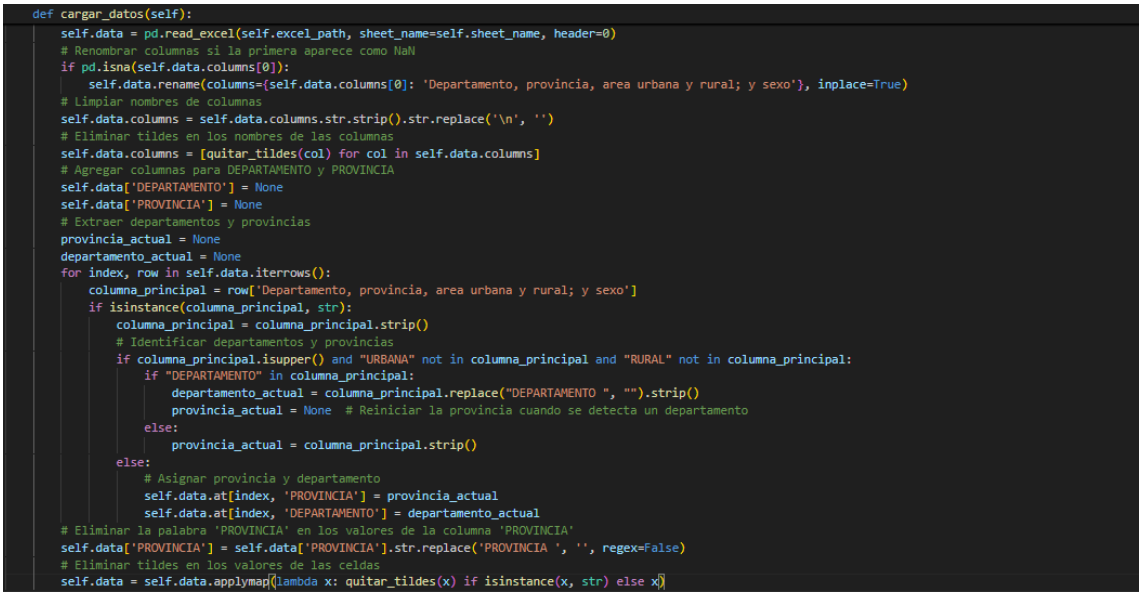
Elimina los acentos o tildes de un texto. Utiliza la normalización Unicode (NFD), que descompone los caracteres acentuados en dos partes (la letra base y el acento). Luego, filtra los caracteres de categoría 'Mn' (marcas de acento), eliminándolos. Así, solo quedan las letras sin tildes.

Constructor: _init_(self, archivo_excel)

```python
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):



```python
def cargar_datos(self):
    self.data = pd.read_excel(self.excel_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)]

Filtra 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):

```python
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:

Filtra 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) Recalcular el total: Suma las columnas de población por grupo de edad (excluyendo "Total") para recalcular 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):

```python
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):

```python
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)

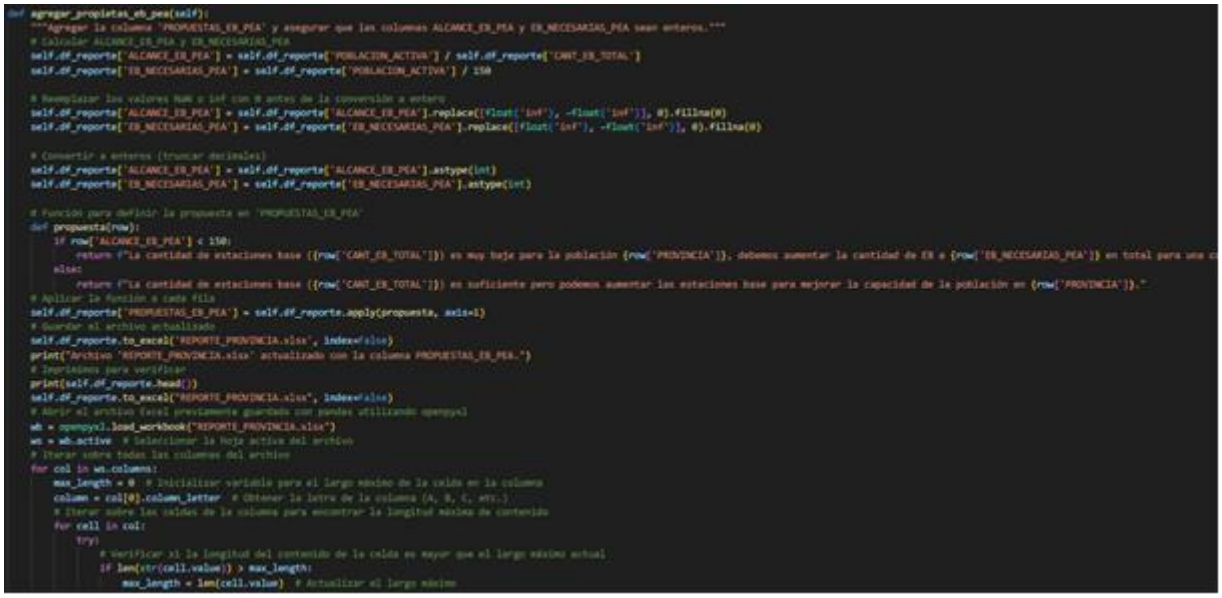
```
def agregar_poblacion_activa(self):
    """Agregar la columna 'Total' de archivo_poblacion_reducida a REPORTE_PROVINCIA."""
    # Realizar el merge entre ambos DataFrames usando las columnas 'DEPARTAMENTO' y 'PROVINCIA'
    df_merge = pd.merge(self.df_reporte, self.df_reducido[['DEPARTAMENTO', 'PROVINCIA',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



a) Calcular las columnas ALCANCE_EB_PEA y EB_NECESARIAS_PEA:

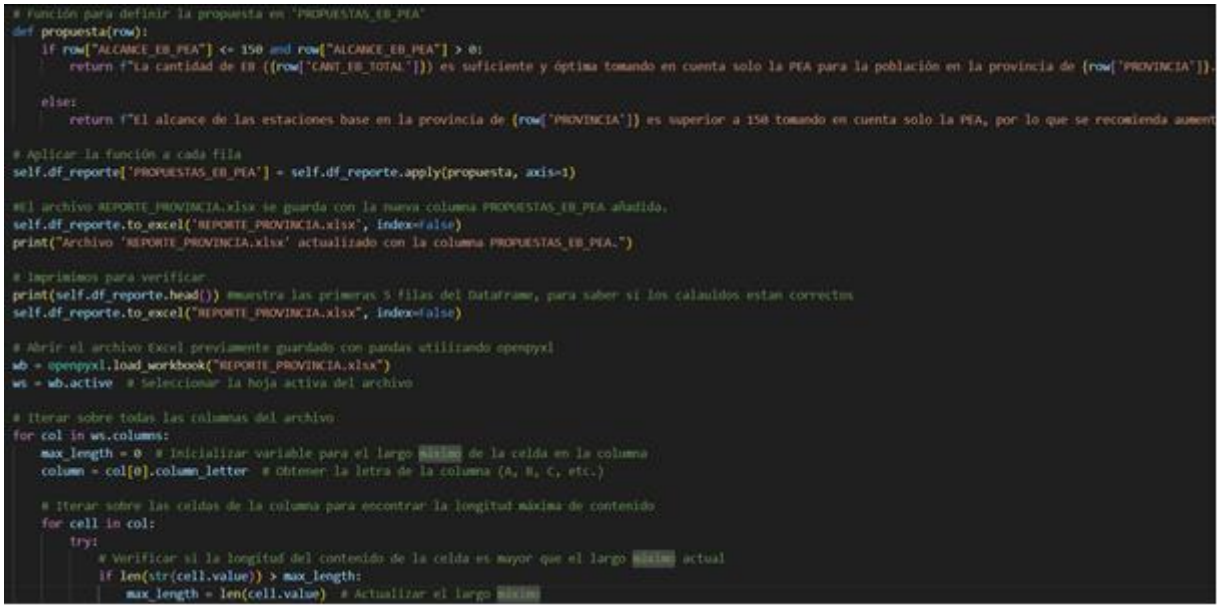
self.df_reporte['ALCANCE_EB_PEA'] = self.df_reporte['POBLACION_ACTIVA'] / self.df_reporte['CANT_EB_TOTAL']

self.df_reporte['EB_NECESARIAS_PEA'] = 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_propietas_eb_pea(self):



```
# Función para definir la propuesta en 'PROPUESTAS_EB_PEA'
def propuesta(row):
    if row["ALCANCE_EB_PEA"] <= 150 and row["ALCANCE_EB_PEA"] > 0:
        return f"La cantidad de EB ({row['CANT_EB_TOTAL']}) es suficiente y óptima tomando en cuenta solo la PEA para la población en la provincia de {row['PROVINCIA']}."

    else:
        return f"El alcance de las estaciones base en la provincia de {row['PROVINCIA']} es superior a 150 tomando en cuenta solo la PEA, por lo que se recomienda aument

# Aplicar la función a cada fila
self.df_reporte['PROPUESTAS_EB_PEA'] = self.df_reporte.apply(propuesta, axis=1)

#El archivo REPORTE_PROVINCIA.xlsx se guarda con la nueva columna PROPUESTAS_EB_PEA añadida.
self.df_reporte.to_excel('REPORTE_PROVINCIA.xlsx', index=False)
print("Archivo 'REPORTE_PROVINCIA.xlsx' actualizado con la columna PROPUESTAS_EB_PEA.")

# Imprimimos para verificar
print(self.df_reporte.head()) #muestra las primeras 5 filas del Dataframe, para saber si los calculdos estan correctos
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(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 o igual que 150 y a su vez mayor a cero, sugiere que la cantidad de estaciones base son suficientes y optima tomando en cuenta solo la PEA. aumentar el número de estaciones base.

- Si no cumple esas condiciones, indica que el número actual es superior a 150 tomando en cuenta solo la PEA y se recomienda aumentar la cantidad de estaciones base.

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

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