Documentación del Proyecto: Sistema de Análisis y Visualización de Rutas Seguras en Bogotá

---

1. Resumen Ejecutivo

Este proyecto tiene como propósito desarrollar una aplicación analítica que contribuya a la toma de decisiones en materia de seguridad ciudadana en Bogotá. Utilizando datos históricos de incidentes reportados a la Policía, la aplicación permite identificar patrones espaciales y temporales de ocurrencia, y emplea esta información para ofrecer rutas seguras entre dos puntos de la ciudad.

El sistema combina técnicas de modelado estadístico con visualización geoespacial interactiva. En particular, se utilizan modelos ARIMA para analizar la evolución mensual de incidentes en cada Unidad de Planeamiento Zonal (UPZ), extrayendo coeficientes que permiten entender la dinámica de cada zona.

La visualización de rutas se realiza mediante un componente interactivo en el que el usuario puede seleccionar coordenadas de origen y destino, y elegir entre rutas más cortas o más seguras, según el nivel histórico de incidentes en las zonas por donde pasa la ruta.

Objetivos del Proyecto

- Desarrollar un sistema capaz de identificar tendencias temporales de incidentes en Bogotá mediante series de tiempo.
- Integrar modelos estadísticos con herramientas de visualización geoespacial para construir una interfaz accesible.
- Permitir a los usuarios comparar rutas en función de la distancia y la seguridad percibida históricamente.
- Generar información útil para ciudadanos, entidades gubernamentales y cuerpos de seguridad.

Público Objetivo

- Tomadores de decisión en instituciones públicas de seguridad y planificación urbana.
- Investigadores en análisis espacial y ciencia de datos.

- Ciudadanos que buscan rutas más seguras en sus trayectos cotidianos.

Descripción

El sistema está diseñado como una aplicación de análisis de datos que integra múltiples componentes técnicos para estudiar la distribución temporal y espacial de los incidentes reportados en Bogotá. A partir de esta información, el sistema genera visualizaciones y rutas seguras que facilitan la interpretación de los datos por parte de usuarios técnicos y no técnicos.

Está compuesto por tres capas principales:

Componentes Principales

1. Capa de Datos

- **Funcionalidad:** Carga, limpieza, transformación y preparación de los datos.
- **Tareas realizadas:**
    - Conversión de columnas de año y mes en una columna de fecha (`FECHA`).
    - Agrupación de los datos por UPZ y por mes.
    - Validación de la calidad de las series temporales (mínimo un año de datos por UPZ).
- **Herramientas utilizadas:** `pandas`.

2. Modelado Estadístico

- **Funcionalidad:** Ajuste de modelos ARIMA para cada UPZ con suficiente información histórica.
- **Tareas realizadas:**
    - Ajuste de modelos `(p,d,q)` para capturar la dinámica de incidentes en el tiempo.
    - Almacenamiento de los coeficientes AR y MA.
    - Pronóstico de los próximos 3 meses de incidentes por UPZ.
    - Generación de gráficos comparativos de coeficientes.

- **Herramientas utilizadas:** `statsmodels`, `matplotlib`.

3. Visualización y Rutas

- **Funcionalidad:** Interfaz para comparar rutas entre dos puntos según distancia y riesgo histórico.
- **Tareas realizadas:**
    - Visualización de rutas seguras usando mapas interactivos.
    - Cálculo de rutas utilizando lógica de penalización según el número histórico de incidentes por zona.
    - Visualización de resultados mediante componentes de `Dash` y `Leaflet`.
- **Herramientas utilizadas:** `Dash`, `Dash Bootstrap Components`, `Dash Leaflet`.

Flujo General del Sistema

1. **Ingreso de datos** → 2. **Transformación temporal y espacial** → 3. **Modelado ARIMA por UPZ** → 4. **Extracción de coeficientes y pronósticos** → 5. **Visualización de patrones y rutas seguras**

Descripción General

El sistema se basa en datos reales de incidentes reportados a la línea de emergencias de la Policía Nacional de Colombia, recopilados por la Secretaría Distrital de Seguridad, Convivencia y Justicia de Bogotá. Estos datos permiten analizar la frecuencia, distribución espacial y evolución temporal de eventos que afectan la seguridad ciudadana.

Dataset Principal: Incidentes Reportados

- **Nombre:** Registro histórico de incidentes por UPZ.
- **Variables clave:**
    - `ANIO`: Año del incidente.
    - `MES`: Mes del incidente.

    - `CANT_INCIDENTES`: Número total de incidentes reportados en esa UPZ y mes.
    - `COD_UPZ`: Código de la Unidad de Planeamiento Zonal donde ocurrió el incidente.
    - `NOM_UPZ`: Nombre de la UPZ (para visualización).
- **Frecuencia:** Mensual.
- **Cobertura temporal:** [indicar rango, ej. 2018-2024].
- **Cobertura espacial:** Todas las UPZ de Bogotá.

Dataset Geográfico: UPZ GeoJSON

- **Nombre:** Límite geográfico de las UPZ en Bogotá.
- **Formato:** GeoJSON.
- **Variables clave:**
    - `CODIGO_UPZ`: Código de la UPZ (clave de unión con `COD_UPZ`).
    - `geometry`: Polígonos geográficos que delimitan cada UPZ.
- **Uso:** Visualización geoespacial en mapas y asociación espacial de incidentes a zonas.

Relación entre datasets

Los datos tabulares de incidentes se cruzan con los datos geoespaciales utilizando la clave común `COD_UPZ` ↔ `CODIGO_UPZ`. Esta unión permite:

- Visualizar los incidentes sobre un mapa de Bogotá.
- Evaluar las rutas según el nivel de seguridad histórica de las zonas que atraviesan.

Procesamiento y Modelado de Series Temporales

Objetivo

Modelar la evolución mensual de los incidentes por cada UPZ mediante modelos ARIMA, identificando patrones temporales relevantes y generando pronósticos a corto plazo. Esta información permite evaluar dinámicas de riesgo por zona y mejorar la toma de decisiones para seguridad y prevención.

Proceso de Preparación

1. **Construcción de la fecha:**
   `df['FECHA'] = pd.to_datetime(df['ANIO'].astype(str) + '-' + df['MES'].astype(str) + '-01')`

Se genera una fecha estándar para cada registro, lo que facilita la conversión de los datos a una serie temporal por UPZ.

2. **Agrupación por UPZ:**
   Para cada UPZ se agrupa la cantidad de incidentes por mes y se ordena cronológicamente:

```
for upz, grupo in df.groupby('COD_UPZ'):
  serie = grupo.groupby('FECHA')['CANT_INCIDENTES'].sum().sort_index()
```

3. **Validación de datos:**
   Solo se modelan UPZ con al menos 12 meses de datos, asegurando robustez en el ajuste y pronóstico.

Selección automática del modelo ARIMA

Se utiliza `auto_arima` de la biblioteca `pmdarima` para encontrar automáticamente la mejor combinación de parámetros (p, d, q) por cada UPZ:

```
modelo_auto = auto_arima(serie, seasonal=False, stepwise=True, suppress_warnings=True, error_action='ignore')
orden = modelo_auto.order
```

Luego, se ajusta un modelo ARIMA con `statsmodels` (más conveniente para análisis e interpretación de parámetros):

```
model = ARIMA(serie, order=orden)
model_fit = model.fit()
```

Pronósticos y Coeficientes

- **Pronóstico:** Se realiza para los siguientes 3 meses.
- **Extracción de coeficientes:** Se extraen los parámetros AR y MA del modelo ajustado, cuando están disponibles.

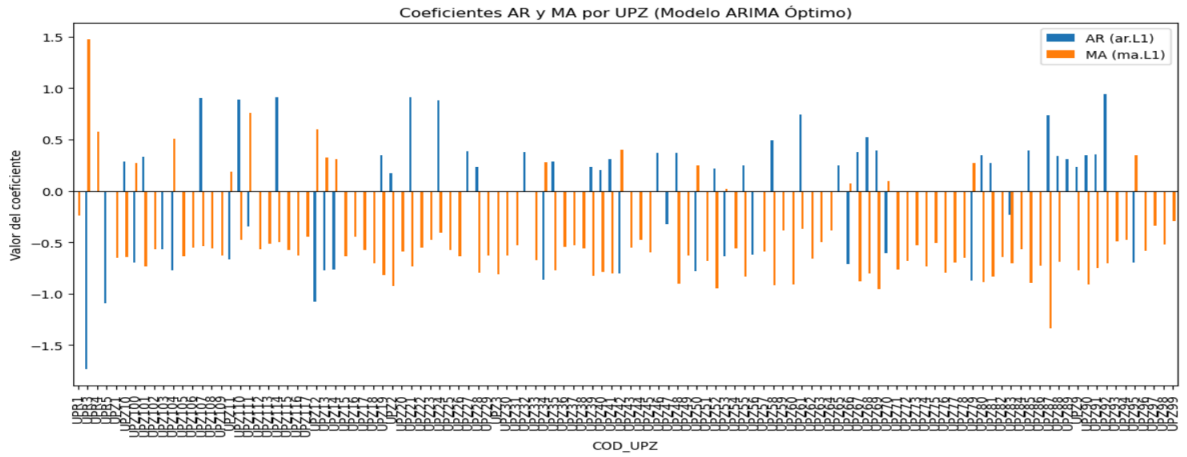
```
params = model_fit.params
coef_ar[upz] = params.get('ar.L1', 0)
coef_ma[upz] = params.get('ma.L1', 0)
```

Estos coeficientes nos permiten entender la dinámica interna de cada serie:

- Un coeficiente **AR (autorregresivo)** alto indica que el número de incidentes pasados tiene fuerte influencia en los valores actuales.
- Un coeficiente **MA (media móvil)** alto sugiere que los errores de predicción pasados tienen influencia significativa.

Visualización

Se construye un gráfico de barras para comparar los coeficientes `ar.L1` y `ma.L1` por cada UPZ modelada:



Asignación de Riesgo a la Red Vial

Este módulo es responsable de incorporar la noción de **riesgo espacial** a una red vial representada mediante un grafo. Para ello, utiliza predicciones por UPZ obtenidas con modelos ARIMA.

Objetivo

Vincular un valor de riesgo proyectado (por ejemplo, cantidad esperada de incidentes) a cada **arista** de la red vial, basándose en la ubicación geográfica de los **nodos extremos** de cada tramo y el riesgo predicho para la **UPZ** en la que se encuentran.

Flujo del proceso

```
nodes = []
for n, data in G.nodes(data=True):
    nodes.append({'node': n, 'x': data['x'], 'y': data['y']})
```

1. **Extracción de nodos**
   Se construye una lista con los nodos del grafo, extrayendo sus coordenadas para posteriormente convertirlos en objetos geométricos.

```
nodes_df['geometry'] = nodes_df.apply(lambda row: Point(row['x'],
row['y']), axis=1)
nodes_gdf = gpd.GeoDataFrame(nodes_df, geometry='geometry',
crs='EPSG:4326')
```

2. **Conversión a `GeoDataFrame`**
   Se convierte el DataFrame en un `GeoDataFrame` para poder realizar operaciones espaciales. Se usa el CRS WGS 84 (`EPSG:4326`) ya que es el sistema más común en datos geográficos con latitud y longitud.

```
nodes_upz = gpd.sjoin(nodes_gdf, gdf_upz[['CODIGO_UPZ',
'geometry']], how='left', predicate='within')
```

3. **Unión espacial con polígonos de UPZ**
   Se determina en qué polígono de UPZ se encuentra cada nodo usando `sjoin` con el predicado `within`. Esto permite etiquetar cada nodo con un `CODIGO_UPZ`.

```
nodes_upz_pred = nodes_upz.merge(df_pred[['CODIGO_UPZ', 'RIESGO']],
on='CODIGO_UPZ', how='left')
```

4. **Asociación de riesgo por predicción**
   Se realiza una fusión con la tabla `df_pred`, que contiene el riesgo estimado por UPZ. El resultado es un nuevo DataFrame donde cada nodo tiene su valor de riesgo (si pertenece a una UPZ).

```
risk_dict = dict(zip(nodes_upz_pred['node'],
nodes_upz_pred['RIESGO'].fillna(0.5)))
```

5. **Creación de diccionario de riesgos**
   Se crea un diccionario donde la clave es el ID del nodo y el valor es su riesgo asociado. Si un nodo no tiene riesgo asignado (por ejemplo, si está fuera de todas las UPZ), se le asigna un valor neutral (0.5).

```
for u, v, k, data in G.edges(keys=True, data=True):
    risk_u = risk_dict.get(u, 0.5)
    risk_v = risk_dict.get(v, 0.5)
    avg_risk = (risk_u + risk_v) / 2
    data['incident_count_normalized'] = avg_risk
```

6. **Asignación de riesgo a las aristas**
   Para cada arista, se calcula el **riesgo medio** entre sus dos nodos extremos, y se almacena en el atributo `incident_count_normalized`. Este valor será utilizado posteriormente como parte del peso en la construcción de rutas.

Cálculo de Rutas Seguras

Este módulo toma como entrada dos coordenadas geográficas (origen y destino) y calcula dos rutas: la más corta y la más segura.

Objetivo

Implementar un sistema de ruteo geoespacial que tenga en cuenta no solo la distancia sino también el nivel de **riesgo vial proyectado** para encontrar trayectos más seguros.

Detalle del algoritmo

```
for u, v, k, data in G.edges(keys=True, data=True):
    if 'geometry' not in data:
        point_u = (G.nodes[u]['x'], G.nodes[u]['y'])
        point_v = (G.nodes[v]['x'], G.nodes[v]['y'])
        data['geometry'] = LineString([point_u, point_v])
```

1. **Asignación de geometría a las aristas**
   Se asegura que cada arista tenga una representación geométrica (`LineString`), necesaria para la visualización y el análisis espacial. Si no la tiene, se construye a partir de las coordenadas de los nodos extremos.

```
length = data.get('length', 1)
risk = data.get('incident_count_normalized', 0.5)
data['custom_weight'] = alpha * length + beta * (risk * 1000)
```

2. **Cálculo del peso personalizado (riesgo + distancia)**
   Para cada arista, se calcula un **peso compuesto** que considera:

   - `length`: longitud física del tramo.
   - `risk`: riesgo proyectado.
   - `alpha`, `beta`: coeficientes que determinan la importancia relativa de distancia y seguridad.
   - El factor `1000` escala el riesgo para que esté en una magnitud comparable a la distancia.

```
orig_node = ox.distance.nearest_nodes(G, X=start_point[0],
Y=start_point[1])
dest_node = ox.distance.nearest_nodes(G, X=end_point[0],
Y=end_point[1])
```

3. **Búsqueda de nodos más cercanos**
   Se determina el nodo de inicio y el nodo de destino más cercanos a las coordenadas proporcionadas por el usuario.

```
route_shortest = nx.shortest_path(G, orig_node, dest_node,
weight='length')
route_safest = nx.shortest_path(G, orig_node, dest_node,
weight='custom_weight')
```

4. **Cálculo de rutas**

   - **Ruta más corta**: minimiza únicamente la distancia.
   - **Ruta más segura**: minimiza el peso compuesto que penaliza tramos de mayor riesgo.

```
def route_to_coords(route):
    segments, risks = [], []
    for u, v in zip(route[:-1], route[1:]):
        edge_data = G.get_edge_data(u, v)
        best = min(edge_data.values(), key=lambda d: d.get('length',
1))
        geom = best.get('geometry', LineString([
            (G.nodes[u]['x'], G.nodes[u]['y']),
            (G.nodes[v]['x'], G.nodes[v]['y']),
        ]))
        segments.append([(lat, lon) for lon, lat in geom.coords])
        risks.append(best.get('incident_count_normalized', 0.5))
    return segments, risks
```

5. **Extracción de coordenadas y riesgos por segmento**
   Se reconstruye la ruta como una lista de segmentos con sus coordenadas geográficas y se almacena el nivel de riesgo de cada uno, lo que permite representar visualmente tramos más o menos peligrosos.

Despliegue Web Interactivo con Dash

Este módulo permite la **visualización interactiva** de las rutas calculadas, proporcionando una interfaz sencilla donde el usuario puede:

- Ingresar coordenadas de origen y destino.
- Visualizar en un mapa tanto la **ruta más corta** como la **más segura**.
- Observar el nivel de riesgo por tramo.

Objetivo

Desplegar una aplicación web accesible que permita a los usuarios comparar rutas desde el punto de vista de seguridad, utilizando datos de riesgo proyectado por UPZ.

Dependencias clave

- `Dash`: Framework para crear interfaces web con Python.
- `dash-leaflet`: Componente para visualización geoespacial.

- `dash-bootstrap-components`: Mejora visual con componentes estilizados.
- `networkx`, `osmnx`, `geopandas`: Backend de cálculo de rutas y operaciones espaciales.

Estructura del layout

```
app = Dash(__name__, external_stylesheets=[dbc.themes.SLATE])
```

1. **Inicialización de la app**
    - Se define la aplicación con una apariencia oscura (`SLATE`) usando Bootstrap.

```
app.layout = html.Div([
    html.H2("Rutas más seguras y rutas más cortas", ...),
```

2. **Título y estilo**
    - Se presenta el título principal de la aplicación, centrado y estilizado.

```
html.Div([
    dbc.Row([...], className="mb-2"),
    dbc.Row([...], className="mb-4"),
]),
```

3. **Formulario de entrada**
    - Dos filas permiten al usuario ingresar coordenadas:
        - Latitud y longitud del **punto de origen**.
        - Latitud y longitud del **punto de destino**.
    - Cada entrada es un `dbc.Input` organizado dentro de columnas (`dbc.Col`) para una distribución limpia.

```
dbc.Button("Calcular rutas", id="submit-button", color="primary",
className="mb-4"),
```

4. **Botón de envío**

- Al presionar el botón, se desencadena el cálculo de rutas y la actualización del mapa.

```
dl.Map([...], center=[4.65, -74.1], zoom=13, ...)
```

5. **Componente `Map`**
    - Mapa centrado en Bogotá, con un nivel de zoom medio.
    - Capas añadidas:
        - `TileLayer`: base de mapa.
        - `Marker`: para el origen y destino.
        - `Polyline`: para las rutas.
        - `LayerGroup`: para gestionar dinámicamente los elementos visuales.

Callback de interacción

```
@app.callback(
    Output('map', 'children'),
    Input('submit-button', 'n_clicks'),
    State('start-lat', 'value'),
    State('start-lon', 'value'),
    State('end-lat', 'value'),
    State('end-lon', 'value')
)
```

1. **Estructura del callback**
    - Se activa cuando el usuario presiona el botón.
    - Recibe como `State` los valores ingresados por el usuario.

```
segments_short, _ = route_to_coords(route_shortest)
segments_safe, risks = route_to_coords(route_safest)
```

2. **Obtención de rutas**

- Se calcula tanto la ruta más corta como la más segura (ya explicadas en el backend).

```
marker_start = dl.Marker(position=[start_lat, start_lon],
children=dl.Tooltip("Origen"))
marker_end = dl.Marker(position=[end_lat, end_lon],
children=dl.Tooltip("Destino"))
```

3. **Creación de marcadores**

    - Se colocan marcadores en el mapa para indicar claramente los puntos de inicio y fin.

```
poly_short = dl.Polyline(positions=[pt for seg in segments_short for
pt in seg], color='blue', weight=4)
```

4. **Visualización de la ruta más corta**

    - Se representa con una línea azul.

```
poly_safe_segments = []
for seg, risk in zip(segments_safe, risks):
    color = get_color_for_risk(risk)
    poly_safe_segments.append(dl.Polyline(positions=seg,
color=color, weight=4))
```

5. **Visualización de la ruta más segura**

    - Cada tramo se pinta con un color acorde a su nivel de riesgo.

    - La función `get_color_for_risk()` convierte un valor numérico de riesgo en un color

```
def get_color_for_risk(risk):
    if risk < 0.3:
        return 'green'
    elif risk < 0.6:
        return 'orange'
    else:
        return 'red'
```

6. **Leyenda visual del riesgo**
    - Verde: bajo riesgo.
    - Naranja: riesgo medio.
    - Rojo: riesgo alto.

```
return [dl.TileLayer(), marker_start, marker_end, poly_short] +
poly_safe_segments
```

7. **Actualización del mapa**
    - Devuelve la lista completa de elementos visuales al mapa principal, que se renderiza en el navegador.

Ejecución del servidor

```
if __name__ == '__main__':
    app.run_server(debug=True)
```

- Inicia el servidor en `localhost:8050` con `debug=True` para facilitar el desarrollo y depuración.