Índice

3. Processamento e Pré-processamento de Dados Geoespaciais

> Nesta seção, vamos explorar operações fundamentais de pré-processamento de dados geoespaciais, essenciais para preparar os dados para análises e modelagens mais avançadas. Cada tópico será acompanhado de uma aplicação prática.

3.1 Geocoding e Geocoding Reverso

> O geocoding é o processo de converter um endereço (como "Cristo Redentor, Rio de Janeiro") em coordenadas geográficas (latitude e longitude). Já o geocoding reverso realiza o caminho oposto: ele transforma coordenadas em um endereço textual.

Exemplo de Geocoding (endereço -> coordenadas)

> A partir da string `"Cristo Redentor, Rio de Janeiro"` é possível obter a latitude e longitude do Cristo Redentor no Rio de Janeiro

In [18]:
```
import numpy as np
from geopy.geocoders import Nominatim # Servico de Geocoding fornecido pelo Open

geolocator = Nominatim(user_agent="geoapi")
location = geolocator.geocode("Cristo Redentor, Rio de Janeiro")

print("Coordenadas do Cristo Redentor:")
print(f"Latitude: {location.latitude}, Longitude: {location.longitude}")
```

```
Coordenadas do Cristo Redentor:
Latitude: -22.9519173, Longitude: -43.2104585
```

Exemplo de Geocoding Reverso (coordenadas -> endereço)

> A partir das coordenadas do Cristo `Latitude: -22.9519173, Longitude: -43.2104585` é possível obter o endereço aproximado

In [19]:
```
reverse_location = geolocator.reverse((location.latitude, location.longitude), l
print("\nEndereço reverso aproximado:")
print(reverse_location.address)
```

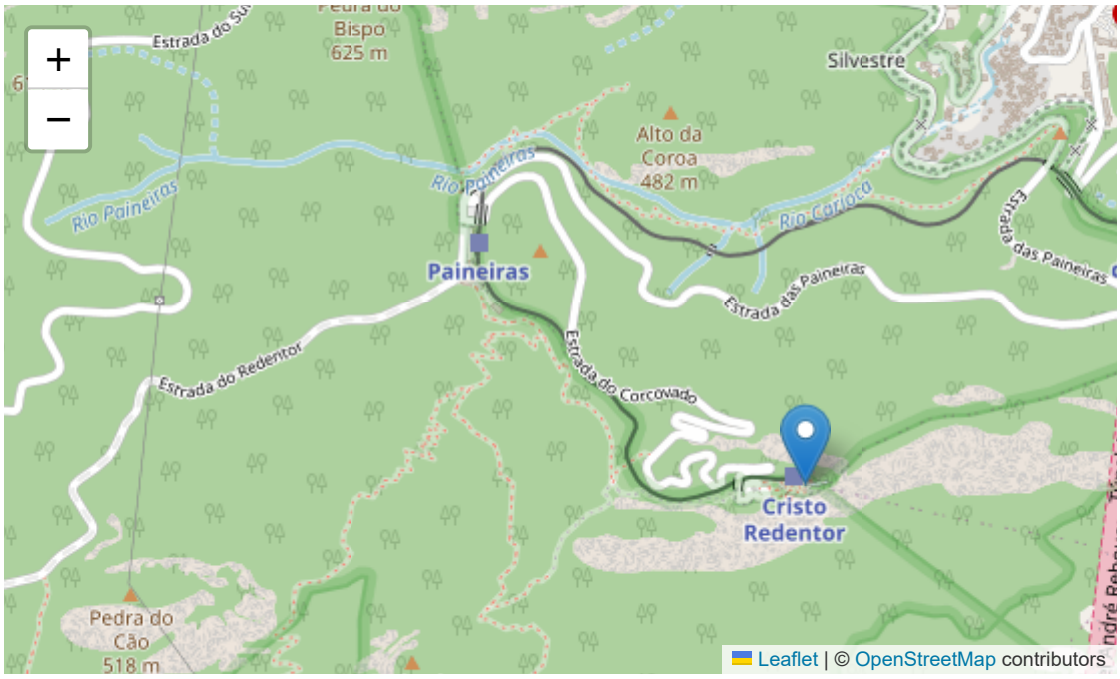
```
Endereço reverso aproximado:
Cristo Redentor, Cristo del Corcovado, Alto da Boa Vista, Rio de Janeiro, Região
Geográfica Imediata do Rio de Janeiro, Região Metropolitana do Rio de Janeiro, Re
gião Geográfica Intermediária do Rio de Janeiro, Rio de Janeiro, Região Sudeste,
22470-180, Brasil
```

Plotando Coordenadas em mapas

> Depois de converter um endereço em coordenadas geográficas (geocoding), podemos plotar o ponto em um mapa usando a biblioteca `folium`.

In [20]:
```
import folium
m = folium.Map(location=[location.latitude, location.longitude], zoom_start=15)
folium.Marker([location.latitude, location.longitude], popup='Cristo Redentor').
m
```

Out[20]:



`Aplicação: Transformando Endereços em Coordenadas`

Vamos aplicar o processo de geocodificação em um conjunto de dados real para converter endereços em coordenadas geográficas.

> Neste exemplo, vamos trabalhar com um dataset real de revenda de apartamentos públicos (HDB flats) em Singapura. A ideia é simples: **usar localização para encontrar boas oportunidades de compra**.
>
> Vamos aplicar técnicas geoespaciais simples para entender se um imóvel está caro ou barato comparado à sua vizinhança. Isso pode ser extremamente útil para investidores, compradores ou até para políticas públicas de habitação.

> **Passo a passo:**
>
> - **Geocodificar os endereços**: transformar o endereço dos imóveis em coordenadas geográficas (latitude e longitude) para poder colocá-los no mapa.
> - **Mapear os imóveis no espaço**: visualizar onde estão localizados e associá-los aos bairros ou regiões de interesse.
> - **Analisar a variação de preços por localização**: calcular o valor por metro quadrado e comparar com a média da região para descobrir quais imóveis estão "fora da curva".

Essa análise ajuda a enxergar padrões que números soltos não mostram — como duas casas com o mesmo preço, mas em localizações bem diferentes.

O script abaixo automatiza o geocoding de endereços no dataset de imóveis `sg-resale-flat-prices-2017-onwards.csv`, transformando cada endereço em coordenadas geográficas (latitude e longitude).

Ele é necessario pois como a API do OpenStreetMap limita 60 requisicões por minuto, é necessario pré-processar e criar o dataset em que faremos as analises.

> *Alem disso, ele que irá "traduzir" os enderecos para coordenadas.*

In [21]:
```
import pandas as pd
import time
from geopy.geocoders import Nominatim
from geopy.extra.rate_limiter import RateLimiter

rodar = False # Coloque True se deseja Rodar o Script

if rodar:
    df = pd.read_csv("datasets/Singapore/sg-resale-flat-prices-2017-onwards.csv"

    #Criando uma Coluna 'endereco' com o  bloco + rua, cidade, Singapore
    #Exemplo: "330 SEMBAWANG CL, SEMBAWANG, Singapore"
    df["endereco"] = df["block"] + " " + df["street_name"] + ", " + df["town"] +

    # Geocodificador
    geolocator = Nominatim(user_agent="geoapi_sg_full")
    geocode = RateLimiter(geolocator.geocode, min_delay_seconds=1)

    # Tentar carregar resultados anteriores
    try:
        df_geo = pd.read_csv("datasets/Singapore/geocodificados.csv")
        processados = set(df_geo["endereco"])
        print(f"Retomando, {len(processados)} endereços já processados.")
    except FileNotFoundError:
        df_geo = pd.DataFrame()
        processados = set()

    novos = []
    batch_size = 50
    pause_seconds = 60
```

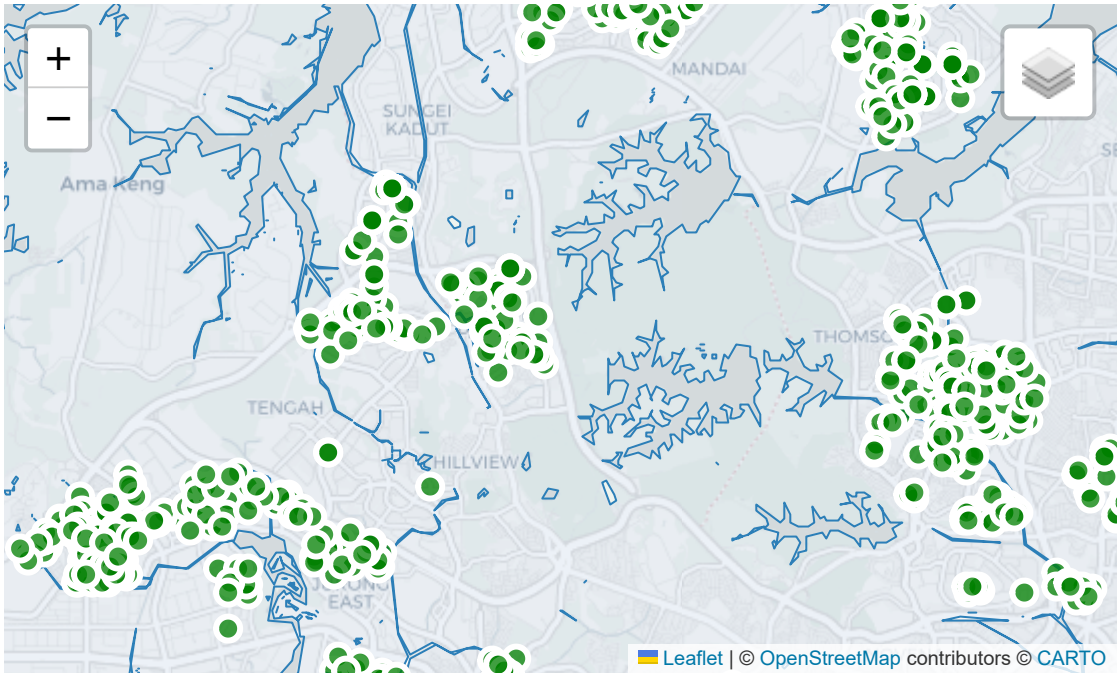
```
    # Iniciar processamento por blocos
    for i, row in df.iterrows():
        endereco = row["endereco"]
        if endereco in processados:
            continue

        try:
            location = geocode(endereco)
            lat = location.latitude if location else None
            lon = location.longitude if location else None
        except:
            lat, lon = None, None

        # Copiar a linha original e adicionar coordenadas
        nova_linha = row.copy()
        nova_linha["latitude"] = lat
        nova_linha["longitude"] = lon
        novos.append(nova_linha)

        print(f"{len(novos)} -> {endereco} => ({lat}, {lon})")

        if len(novos) % batch_size == 0:
            df_novos = pd.DataFrame(novos)
            df_geo = pd.concat([df_geo, df_novos], ignore_index=True).drop_dupli
            df_geo.to_csv("datasets/Singapore/geocodificados.csv", index=False)
            print(f" Salvo após {len(novos)} registros.")
            novos.clear()
            print(f" Pausando por {pause_seconds} segundos...")
            time.sleep(pause_seconds)

    # Salvar o restante
    if novos:
        df_novos = pd.DataFrame(novos)
        df_geo = pd.concat([df_geo, df_novos], ignore_index=True).drop_duplicate
        df_geo.to_csv("datasets/Singapore/geocodificados.csv", index=False)
        print(" Processamento finalizado.")
```

> Agora, Com o dataset `geocodificados.csv` já é possivel montar um simples Mapa Com os pontos desses imoveis

In [22]:
```
# Carregar os dados ja pre-processados
df = pd.read_csv("datasets/Singapore/geocodificados.csv")

# Amostrar para testes rápidos
df_amostra = df.sample(n=30, random_state=42).copy()
df = df.dropna()
m = folium.Map(location=[1.3521, 103.8198], zoom_start=12, tiles="CartoDB Positr

folium.GeoJson(
    "datasets/Singapore/singapore_boundary.geojson",
    name="Limite de Singapura",
    style_function=lambda feature: {
        "color": "#2c7fb8",          # azul mais suave
        "weight": 1,
        "fillColor": "#a6bddb",
        "fillOpacity": 0.2
```

```
    }
).add_to(m)

for _, row in df.iterrows():
    endereco = row["endereco"]
    popup_text = f"<b>Endereço:</b> {endereco}<br>"

    folium.CircleMarker(
        location=(row["latitude"], row["longitude"]),
        radius=6,
        color='white',
        fill_color='green',
        fill=True,
        fill_opacity=0.75,
        popup=folium.Popup(popup_text, max_width=250)
    ).add_to(m)



# Adicionar controle de camadas
folium.LayerControl().add_to(m)

m
```

Out[22]:



Mapa Interativo: Distribuição Espacial dos Preços por m²

> | Faixa | Critério | Cor | |-------|----------|-----| | **Barato** | Preço/m² abaixo do 1º quartil (Q1) | 🟢 Verde | | **Na média** | Entre Q1 e Q3 (mediana) | 🟠 Laranja | | **Caro** | Acima do 3º quartil (Q3) | 🔴 Vermelho |

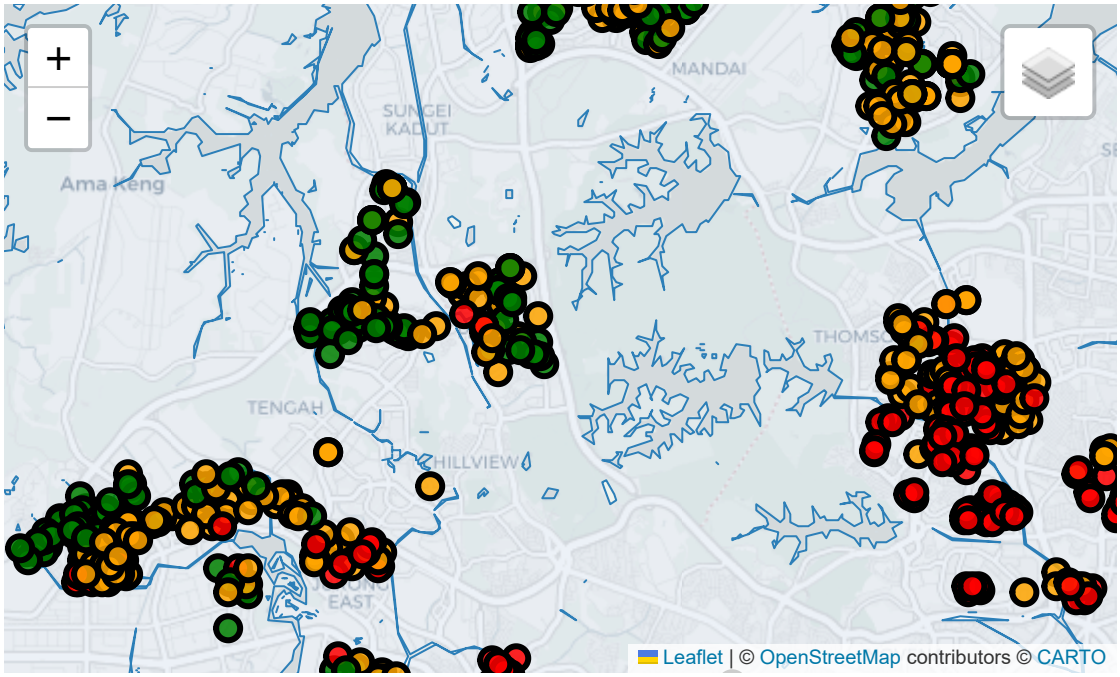
In [23]:

```
import folium
from folium.plugins import MarkerCluster

# Calcular o preço por m²
df["preco_m2"] = df["resale_price"] / df["floor_area_sqm"]

# Calcular quartis
q1 = df["preco_m2"].quantile(0.25)
```

```
q3 = df["preco_m2"].quantile(0.75)

# Função de cor baseada nos quartis
def cor_por_preco_m2(valor):
    if valor < q1:
        return "green"
    elif valor > q3:
        return "red"
    else:
        return "orange"

# Criar o mapa
m = folium.Map(location=[1.3521, 103.8198], zoom_start=12, tiles="CartoDB positr

# Adicionar camada com o limite de Singapura
folium.GeoJson(
    "datasets/Singapore/singapore_boundary.geojson",
    name="Limite de Singapura",
    style_function=lambda feature: {
        "color": "#2c7fb8",
        "weight": 1,
        "fillColor": "#a6bddb",
        "fillOpacity": 0.2
    }
).add_to(m)


# Adicionar marcadores ao cluster
for _, row in df.iterrows():
    preco_total = row["resale_price"]
    preco_m2 = row["preco_m2"]
    popup_text = (
        f"<b>Preço total:</b> ${preco_total:,.0f}<br>"
        f"<b>Área:</b> {row['floor_area_sqm']} m²<br>"
        f"<b>Preço/m²:</b> ${preco_m2:,.0f}<br>"
        f"<b>Tipo:</b> {row['flat_type']}"
    )
    folium.CircleMarker(
        location=(row["latitude"], row["longitude"]),
        radius=6,
        color="black",  # contorno
        fill=True,
        fill_color=cor_por_preco_m2(preco_m2),
        fill_opacity=0.85,
        popup=folium.Popup(popup_text, max_width=250)
    ).add_to(m)

# Controles
folium.LayerControl().add_to(m)

print(f"{len(df)} imóveis plotados com cores baseadas em preço/m².")
m
```

```
1404 imóveis plotados com cores baseadas em preço/m².
```

Out[23]:



Conclusão

Com base apenas nos endereços textuais, conseguimos:

> - Localizar imóveis no mapa
> - Visualizar a distribuição espacial dos preços
> - Preparar os dados para análises

Isso mostra como geocoding pode ser usado como uma ponte entre dados tradicionais e geoespaciais.

3.2 Projecoes Cartograficas

Projeções cartográficas são transformações matemáticas que convertem a superfície curva da Terra (esferoide) em uma superfície plana (mapa). Como não é possível representar perfeitamente uma esfera em um plano, cada projeção faz **compromissos entre preservar forma, área, distância ou direção**.

Por que isso importa?

Para muitas análises espaciais — como **cálculo de área, distância entre pontos, buffers e rotas** — é essencial que os dados estejam em uma **projeção apropriada**. A maioria dos > dados geográficos brutos vem no sistema **WGS84** (EPSG:4326), que usa latitude e longitude (graus), mas não é adequado para medições métricas.

> ### Exemplos de Projeções
>
> - **WGS84 (EPSG:4326)**: padrão global, usa graus. Bom para visualização, ruim para cálculos.
> - **UTM (Universal Transverse Mercator)**: divide o globo em zonas com projeção métrica, ideal para cálculos de distância e área em regiões menores.

> - **Albers Equal Area**: preserva área, útil para análise de distribuição espacial em grandes regiões.

Aplicação: Transformar um dataset de coordenadas geográficas (WGS84) para projeção métrica (UTM), facilitando o cálculo de distâncias reais.

In [24]:
```
import geopandas as gpd

# Carregando shapefile dos municípios do RJ
shapefile_path = "datasets/RJ_2023/RJ_Municipios_2023.shp"
gdf = gpd.read_file(shapefile_path)

# Selecionar apenas um município (por exemplo, Rio de Janeiro)
rio = gdf[gdf["NM_MUN"] == "Rio de Janeiro"]

# Calcular área no sistema WGS84 (em graus) - NÃO CONFIÁVEL!
area_wgs84 = rio.geometry.area.iloc[0]

# Agora convertemos para UTM zona 23S (projeção métrica adequada)
rio_utm = rio.to_crs(epsg=31983)

# Calcular área no sistema UTM (em metros quadrados) - CONFIÁVEL!
area_utm = rio_utm.geometry.area.iloc[0]

# Resultados:
print(f"Área no sistema WGS84 (graus²): {area_wgs84}")
print(f"Área no sistema UTM (m²): {area_utm:,.2f}")
print(f"Área no sistema UTM (km²): {area_utm/1e6:,.2f}")
```

```
Área no sistema WGS84 (graus²): 0.1056623137465809
Área no sistema UTM (m²): 1,200,125,190.48
Área no sistema UTM (km²): 1,200.13
```

```
C:\Users\PC\AppData\Local\Temp\ipykernel_13960\912035043.py:11: UserWarning: Geom
etry is in a geographic CRS. Results from 'area' are likely incorrect. Use 'GeoSe
ries.to_crs()' to re-project geometries to a projected CRS before this operation.

  area_wgs84 = rio.geometry.area.iloc[0]
```

Exemplo didático: comparação entre WGS84 e UTM com um quadrado de 1 grau x 1 grau

In [25]:
```
import geopandas as gpd
from shapely.geometry import box

# Criar um quadrado de 1 grau x 1 grau sobre o Brasil (latitude e longitude)
gdf_deg = gpd.GeoDataFrame(geometry=[box(-43, -23, -42, -22)], crs="EPSG:4326")

# Calcular a área diretamente (em graus²) — ERRADO
area_graus = gdf_deg.geometry.area.iloc[0]

# Projetar para UTM zona 23S (sistema métrico adequado ao RJ)
gdf_metros = gdf_deg.to_crs(epsg=31983)

# Calcular a área em m² — CORRETO
area_metros = gdf_metros.geometry.area.iloc[0]

# Mostrar a comparação
print(f"Área em WGS84 (graus²): {area_graus}")
```

```
print(f"Área em UTM (m²): {area_metros:,.2f}")
print(f"Área em UTM (km²): {area_metros / 1e6:,.2f}")
```

```
Área em WGS84 (graus²): 1.0
Área em UTM (m²): 11,403,924,070.61
Área em UTM (km²): 11,403.92
```

```
C:\Users\PC\AppData\Local\Temp\ipykernel_13960\3667702788.py:8: UserWarning: Geom
etry is in a geographic CRS. Results from 'area' are likely incorrect. Use 'GeoSe
ries.to_crs()' to re-project geometries to a projected CRS before this operation.

  area_graus = gdf_deg.geometry.area.iloc[0]
```

`Conclusão`

O valor em graus quadrados não tem interpretação prática direta — ele depende da curvatura da Terra e da latitude! Embora o polígono tenha "1 grau²", isso representa 12 km² reais — e esse valor muda dependendo da latitude (quanto mais longe do equador, menor o "tamanho real" de um grau).

> - Já o valor em metros quadrados, obtido com uma projeção como UTM, é o que você >precisa para:
> - Calcular densidade populacional
> - Estimar área útil de uma zona
> - Fazer modelagem espacial confiável

Se você errar a projeção, você compromete os resultados do modelo.

3.3 Intersecoes de Camadas Geoespaciais

> Uma das operações mais poderosas em ciência de dados geoespaciais é a **interseção entre camadas (layers)**. Essa técnica permite responder a perguntas espaciais complexas, como:
>
> - 🏥 Quais hospitais estão em áreas de risco ambiental?
> - 🌳 Quais bairros têm escolas dentro de zonas verdes?
> - 🚨 Quais ocorrências policiais aconteceram dentro de determinada região?

Essas respostas emergem ao **cruzar diferentes fontes de dados espaciais** — como a sobreposição de **pontos, linhas e polígonos** com camadas raster (como imagens de satélite ou dados > climáticos).

Aplicação: `Estações de metrô e Imóveis a venda em São Paulo`

> Neste exemplo, carregamos um arquivo com geometria do limite municipal de São Paulo e o salvamos no formato `geojson`. Esse tipo de

> conversão é comum quando precisamos reutilizar dados geoespaciais em ferramentas interativas ou compartilhamento entre sistemas.

In [26]:
```
import geopandas as gpd

# Carregar o arquivo .json (desde que seja um GeoJSON válido)
gdf_limite = gpd.read_file("datasets/Sao_Paulo/limite_sao_paulo.json")

# Verificar rapidamente
print(gdf_limite.head())

# Salvar como .geojson (formato padrão para mapas)
gdf_limite.to_file("datasets/Sao_Paulo/limite_sao_paulo.geojson", driver="GeoJSO
```

```
        id              name       description  \
0  3500105        Adamantina        Adamantina
1  3500204            Adolfo            Adolfo
2  3500303             Aguaí             Aguaí
3  3500402    Águas da Prata    Águas da Prata
4  3500501  Águas de Lindóia  Águas de Lindóia

                                            geometry
0  POLYGON ((-51.05787 -21.39888, -51.05365 -21.4...
1  POLYGON ((-49.65478 -21.20607, -49.63847 -21.2...
2  POLYGON ((-47.2089 -21.97129, -47.20297 -21.97...
3  POLYGON ((-46.70755 -21.82895, -46.7038 -21.84...
4  POLYGON ((-46.61147 -22.43496, -46.60302 -22.4...
```

Neste exemplo, construímos um mapa interativo de São Paulo integrando diferentes camadas de dados geoespaciais:

- **Imóveis residenciais**, com coordenadas em WGS84;
- **Estações de metrô**, carregadas de um CSV com localização geográfica;
- **Limite municipal de São Paulo**, carregado de um arquivo `.geojson`.

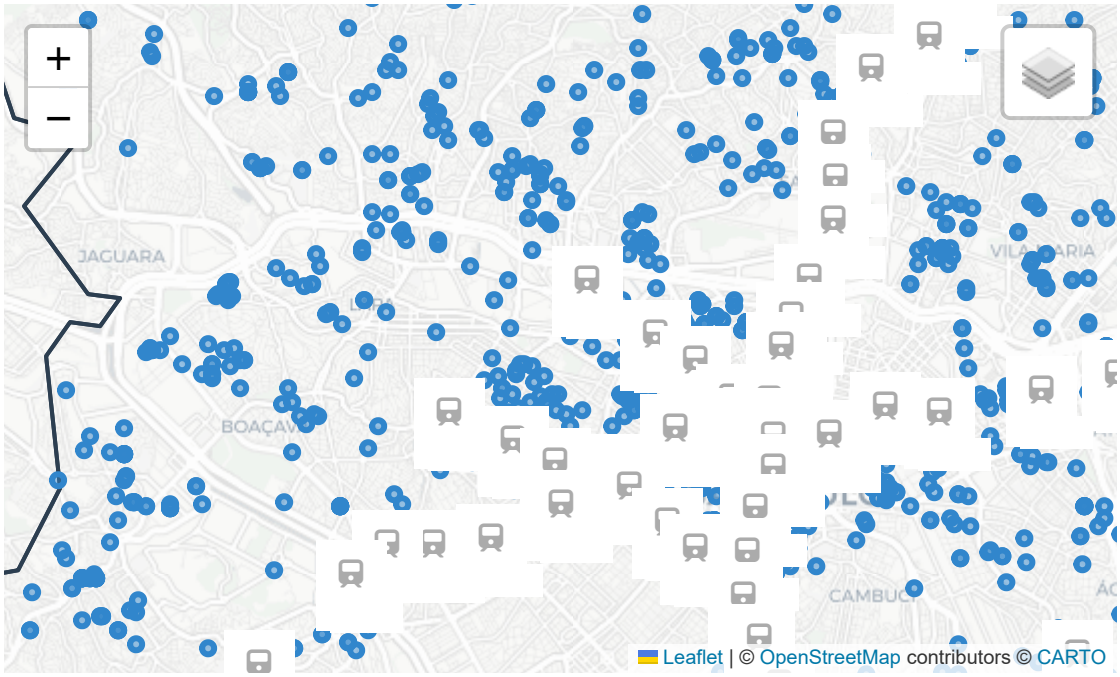
Utilizamos a biblioteca `Folium` para gerar a visualização e sobrepomos:

- Círculos azuis representando imóveis;
- Ícones vermelhos com o símbolo de trem para estações de metrô;
- O contorno do município, com estilo semitransparente.

In [27]:
```
import folium
import geopandas as gpd
import pandas as pd
import json
from shapely.geometry import Point

# Carregar imóveis
df_imoveis = pd.read_csv("datasets/Sao_Paulo/Real_Estate/dados_wgs.csv", sep=";"
df_imoveis["lon"] = df_imoveis["lon"].str.replace(",", ".").astype(float)
df_imoveis["lat"] = df_imoveis["lat"].str.replace(",", ".").astype(float)
gdf_imoveis = gpd.GeoDataFrame(df_imoveis, geometry=gpd.points_from_xy(df_imovei

# Carregar estações de metrô
df_metro = pd.read_csv("datasets/Sao_Paulo/Metro/metrosp_stations_v2.csv")
gdf_metro = gpd.GeoDataFrame(df_metro, geometry=gpd.points_from_xy(df_metro.lon,
```

```
# Criar o mapa
m = folium.Map(location=[-23.55, -46.63], zoom_start=11.5, tiles="CartoDB positr

# Carregar o limite da cidade com encoding correto
with open("datasets/Sao_Paulo/limite_sao_paulo.geojson", encoding="utf-8") as f:
    limite_geojson = json.load(f)

# Adicionar limite da cidade
folium.GeoJson(
    limite_geojson,
    name="Limite São Paulo",
    style_function=lambda feature: {
        "color": "#2c3e50",
        "weight": 2,
        "fillColor": "#f5f5f5",
        "fillOpacity": 0.1
    }
).add_to(m)

# Adicionar imóveis
for _, row in gdf_imoveis.iterrows():
    folium.CircleMarker(
        location=(row.geometry.y, row.geometry.x),
        radius=3,
        color="#3186cc",
        fill=True,
        fill_opacity=0.5
    ).add_to(m)

# Adicionar estações de metrô com ícone personalizado
for _, row in gdf_metro.iterrows():
    folium.Marker(
        location=(row.geometry.y, row.geometry.x),
        icon=folium.Icon(color='red', icon='train', prefix='fa'),
        popup=row.get("station_name", "Estação de Metrô")
    ).add_to(m)

# Adicionar controle de camadas
folium.LayerControl().add_to(m)

# Exibir o mapa
m
```

Out[27]:



Esse tipo de visualização é útil para responder perguntas como:

- "Onde há maior concentração de imóveis?"
- "Quais imóveis estão mais próximos de estações de metrô?"
- "Como os imóveis se distribuem dentro dos limites da cidade?"

3.3 Análise Exploratória: Distância ao Metrô e Preço dos Imóveis

Nesta etapa, buscamos entender se a **localização em relação ao transporte público** (mais especificamente, estações de metrô) influencia o **preço por metro quadrado (R$/m²)** dos imóveis na cidade de São Paulo.

Para isso, seguimos os seguintes passos:

1. **Cálculo do preço por m²** (`valor_total / area_util`) para cada imóvel.
2. **Geocodificação dos imóveis e estações de metrô** em um sistema de coordenadas compatível com medição em metros (UTM).
3. **Cálculo da distância até a estação de metrô mais próxima** para cada ponto de imóvel.
4. **Criação de uma nova coluna categórica** com três faixas de distância:
   - `≤ 500m`
   - `500m - 1km`
   - `> 1km`

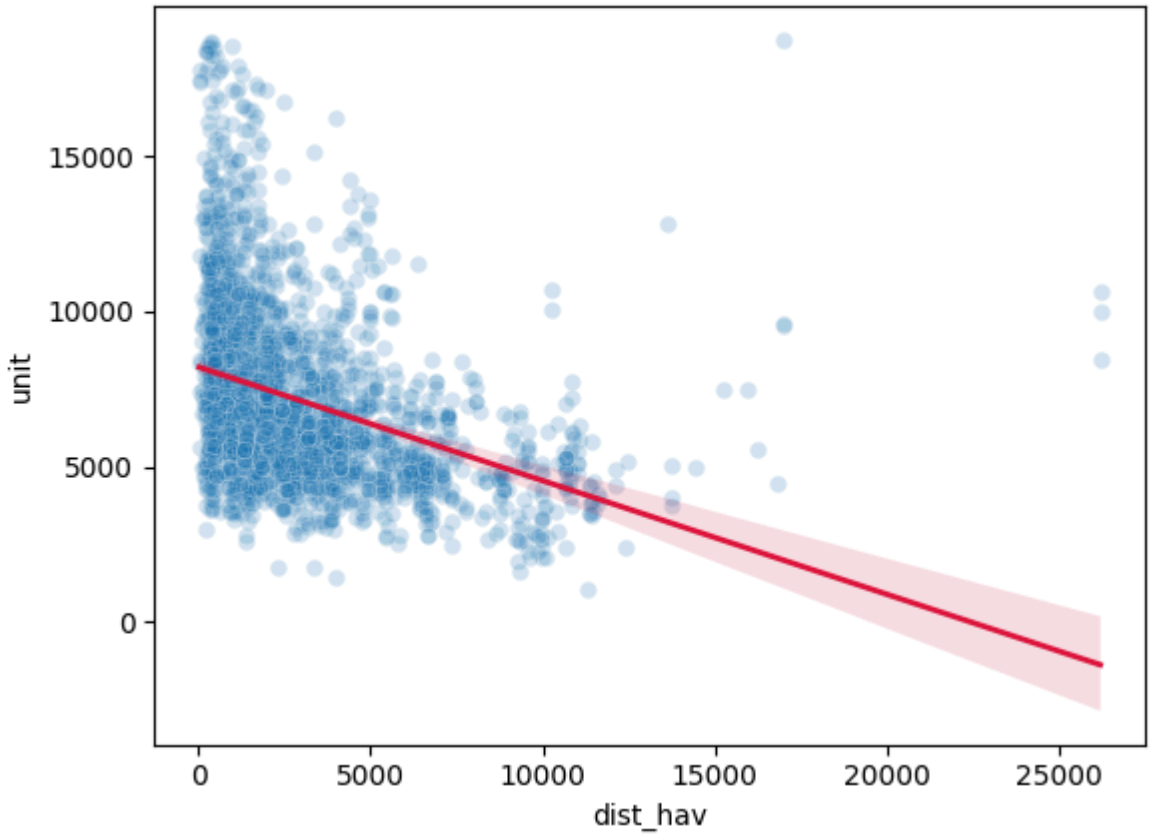
Em seguida, realizamos uma **análise descritiva** do preço por m² em cada grupo.

In [28]:
```
from haversine import haversine
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

# ----------------------------
```

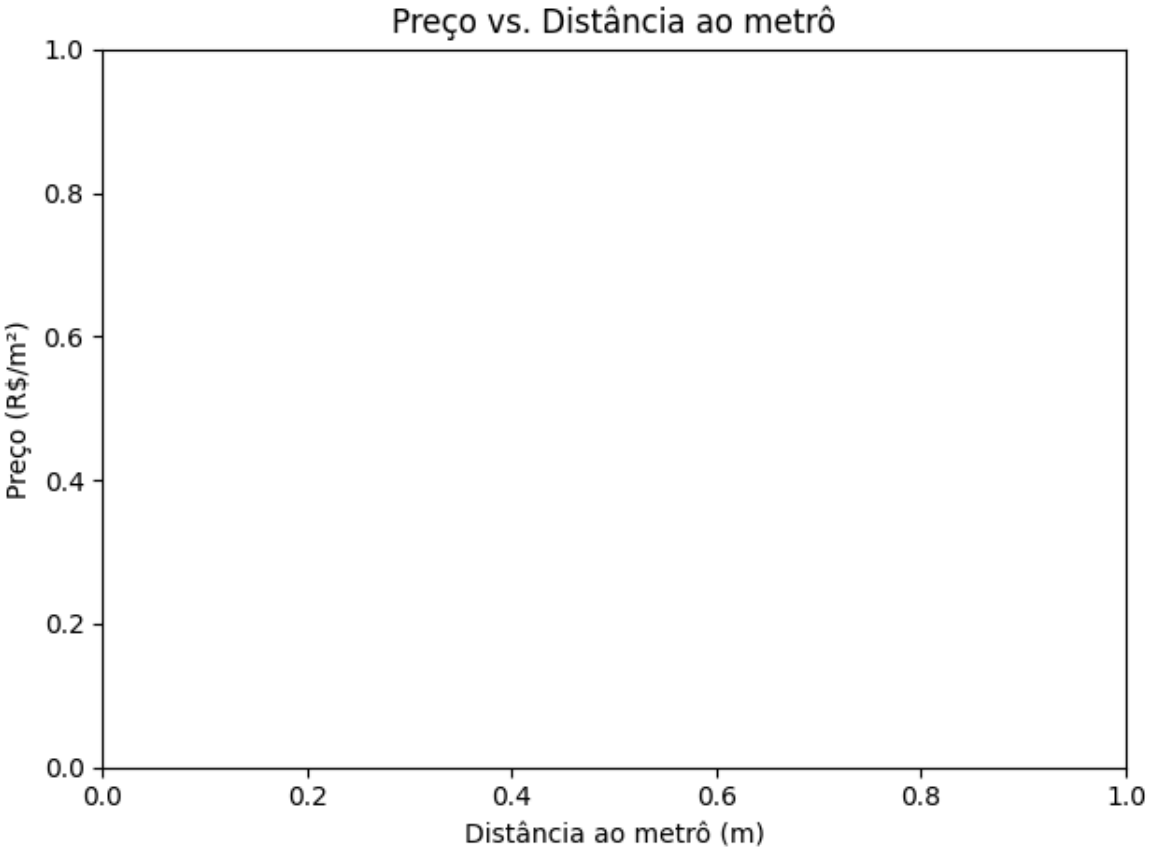
```
# 1.1 Distância exata ao metrô
# ----------------------------
# cria tuplas lat/lon p/ acelerar lookup
gdf_metro = gdf_metro.to_crs(epsg=4326)  # Garantir que está no sistema WGS84
gdf_imoveis = gdf_imoveis.to_crs(epsg=4326)  # Garantir que está no sistema WGS8

metro_coords = list(zip(gdf_metro.lat, gdf_metro.lon))

def min_dist(row):
    _, d = min(
        ((i, haversine((row.lat, row.lon), c)*1000) for i, c in enumerate(metro_
        key=lambda x: x[1]
    )
    return d

gdf_imoveis["dist_hav"] = gdf_imoveis.apply(min_dist, axis=1)

# Corrigir dist caso esteja como string com vírgula
if "dist" in gdf_imoveis.columns and gdf_imoveis["dist"].dtype == "object":
    gdf_imoveis["dist"] = gdf_imoveis["dist"].str.replace(",", ".").astype(float

# Calcular erro médio entre distância declarada e distância calculada via havers
print("Erro médio (m):", np.mean(np.abs(gdf_imoveis["dist"] - gdf_imoveis["dist_


# 1.2 Correlação preço × distância

gdf_imoveis["unit"] = (
    gdf_imoveis["unit"]
    .astype(str)
    .str.replace(".", "", regex=False)
    .str.replace(",", ".", regex=False)
    .astype(float)
)

corr = gdf_imoveis["unit"].corr(gdf_imoveis["dist_hav"])
print(f"Correlação Pearson (unit × dist_hav): {corr:.3f}")

sns.scatterplot(data=gdf_imoveis, x="dist_hav", y="unit", alpha=0.2)
sns.regplot(data=gdf_imoveis, x="dist_hav", y="unit",
            scatter=False, color="crimson", line_kws={"lw":2})
```

```
Erro médio (m): 91.70230090539975
Correlação Pearson (unit × dist_hav): -0.352
```

Out[28]: <Axes: xlabel='dist_hav', ylabel='unit'>



In [29]:
```
plt.xlabel("Distância ao metrô (m)"); plt.ylabel("Preço (R$/m²)")
plt.title("Preço vs. Distância ao metrô"); plt.tight_layout()
plt.show()
```



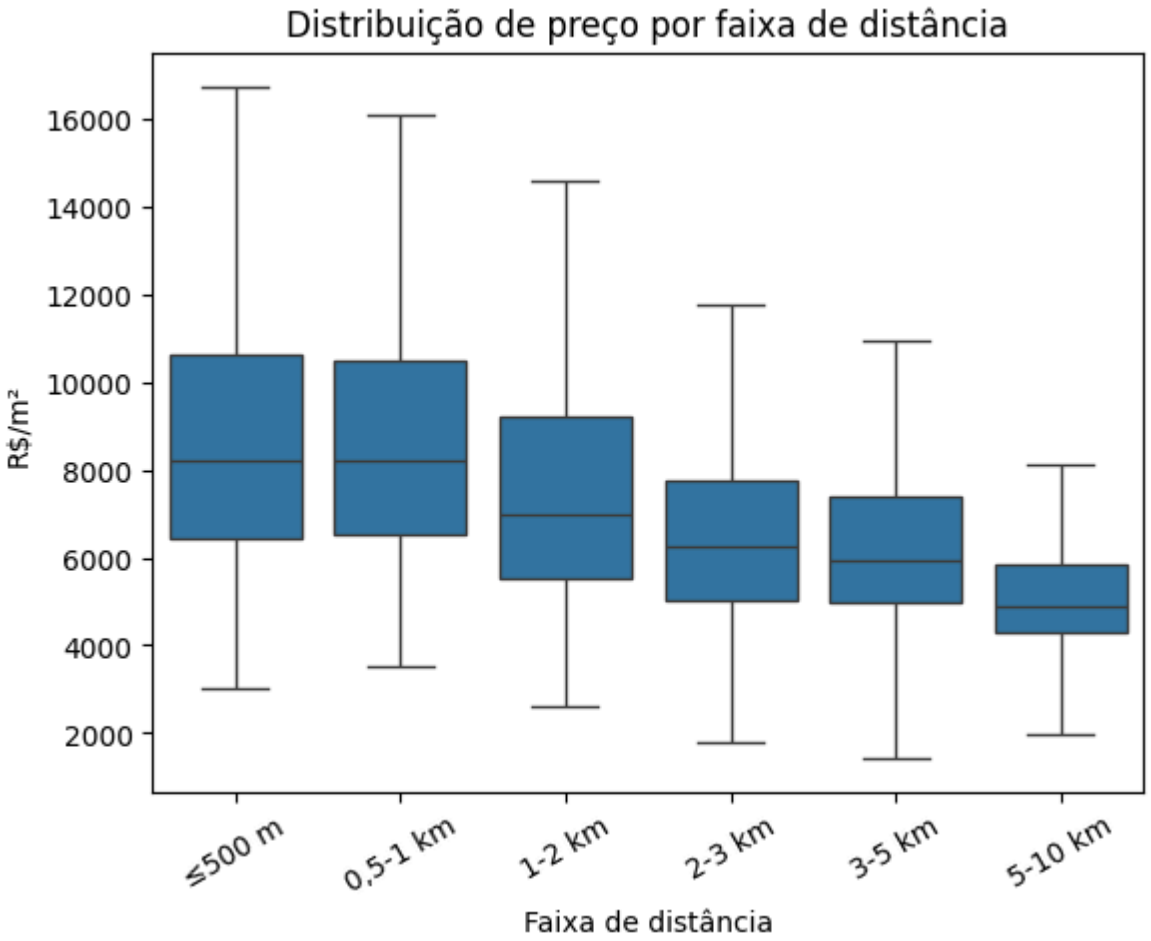
O gráfico acima apresenta a relação entre o preço dos imóveis por metro quadrado (R$/m²) e a distância até a estação de metrô mais próxima.

Cada ponto representa um imóvel, enquanto a linha vermelha indica a tendência geral obtida por regressão linear.

- É possível observar uma **correlação negativa significativa**: imóveis localizados **mais próximos do metrô tendem a ser mais caros**. À medida que a distância aumenta, o preço por metro quadrado tende a diminuir. Essa relação sugere que a **acessibilidade ao transporte público é um fator importante de valorização imobiliária** na cidade de São Paulo.

Além disso, a maior concentração de pontos está abaixo de 1000 metros de distância, indicando que muitos imóveis de alto valor estão próximos a estações. Essa análise pode ser útil para modelagem preditiva de preços e também para embasar decisões de planejamento urbano e investimento.

In [30]:
```
bins = [0, 500, 1000, 2000, 3000, 5000, 10000]
labels = ["≤500 m","0,5-1 km","1-2 km","2-3 km","3-5 km","5-10 km"]
gdf_imoveis["dist_bin"] = pd.cut(gdf_imoveis["dist_hav"], bins=bins, labels=labe

ax = sns.boxplot(x="dist_bin", y="unit", data=gdf_imoveis, showfliers=False)
ax.set_xlabel("Faixa de distância"); ax.set_ylabel("R$/m²")
ax.set_title("Distribuição de preço por faixa de distância"); plt.xticks(rotatio
plt.show()

print(gdf_imoveis.groupby("dist_bin")["unit"].agg(["count","mean","median"]).rou
```



```
          count    mean  median
dist_bin
≤500 m      335  8839.0  8219.0
0,5-1 km    318  8699.0  8213.0
1-2 km      554  7686.0  6962.0
2-3 km      400  6633.0  6244.0
3-5 km      474  6393.0  5938.0
5-10 km     342  5167.0  4885.0
```

```
C:\Users\PC\AppData\Local\Temp\ipykernel_13960\2610370243.py:10: FutureWarning: T
he default of observed=False is deprecated and will be changed to True in a futur
e version of pandas. Pass observed=False to retain current behavior or observed=T
rue to adopt the future default and silence this warning.
  print(gdf_imoveis.groupby("dist_bin")["unit"].agg(["count","mean","median"]).ro
und(0))
```

Resultados

Abaixo estão as principais estatísticas por categoria de distância:

- Imóveis **a até 500 metros** de estações de metrô possuem **preço por m² mais elevado** (R$8.827/m² em média).
- Imóveis **entre 500m e 1km** também mantêm um valor alto e estável (R$8.690/m²).
- Já imóveis **acima de 1km** apresentam um valor significativamente menor (R$6.570/m²).

Essa diferença **reforça a valorização de imóveis bem conectados ao transporte público**, informação crucial para investidores, urbanistas e planejadores de política habitacional.

O gráfico de boxplot abaixo reforça essas diferenças, mostrando também a variabilidade dentro de cada grupo.

3.4 Interpolacao Espacial Kriging

O problema

Imagine que você possui medições de temperatura do fundo do mar feitas por sensores em pontos esparsos. Você quer criar um **mapa contínuo** para visualizar a variação da temperatura ao longo de toda a região.

Para isso, usamos **interpolação espacial** — uma técnica que estima valores em locais onde não há dados, com base na localização e valor dos pontos conhecidos.

Comparando abordagens

Método	Como funciona	Limitações
IDW (Inverse Distance Weighting)	A média ponderada dos pontos vizinhos, com pesos baseados na distância (“pontos mais próximos são mais parecidos”)	Não considera padrões espaciais complexos, só a distância

Método	Como funciona	Limitações
Kriging	Ajusta um modelo estatístico (variograma) para estimar valores com base na **estrutura espacial do dado**	Mais complexo, exige ajuste de modelo e interpretação mais técnica

O que é o Kriging?

Kriging é uma técnica de interpolação **baseada em modelos estatísticos** que incorporam:

- A **distância** entre os pontos
- A **variabilidade dos dados**
- A **direção ou tendência** dos dados (em versões mais avançadas)

Ideal para fenômenos naturais como temperatura, poluição, teor de minério, etc.

In [31]:
```
from matplotlib import pyplot as plt
from pykrige.ok import OrdinaryKriging
from scipy.interpolate import griddata
# Recarregar o CSV pulando a linha com as unidades
df = pd.read_csv("datasets/temp_sea_bottom/northeast_atlatic_sea_bottom_temp.csv
# Garantir conversão correta para tipos numéricos
df["latitude"] = pd.to_numeric(df["latitude"], errors="coerce")
df["longitude"] = pd.to_numeric(df["longitude"], errors="coerce")
df["sea_bottom_temperature"] = pd.to_numeric(df["sea_bottom_temperature"], error

# Remover linhas com valores faltantes
df = df.dropna(subset=["latitude", "longitude", "sea_bottom_temperature"])

# Exibir preview
df.head()
```

Out[31]:

	time	latitude	longitude	sea_bottom_temperature
0	2017-11-30T00:00:00Z	48.00625	-17.99375	2.149733
1	2017-11-30T00:00:00Z	48.00625	-17.98125	2.149733
2	2017-11-30T00:00:00Z	48.00625	-17.96875	2.149733
3	2017-11-30T00:00:00Z	48.00625	-17.95625	2.149733
4	2017-11-30T00:00:00Z	48.00625	-17.94375	2.172670

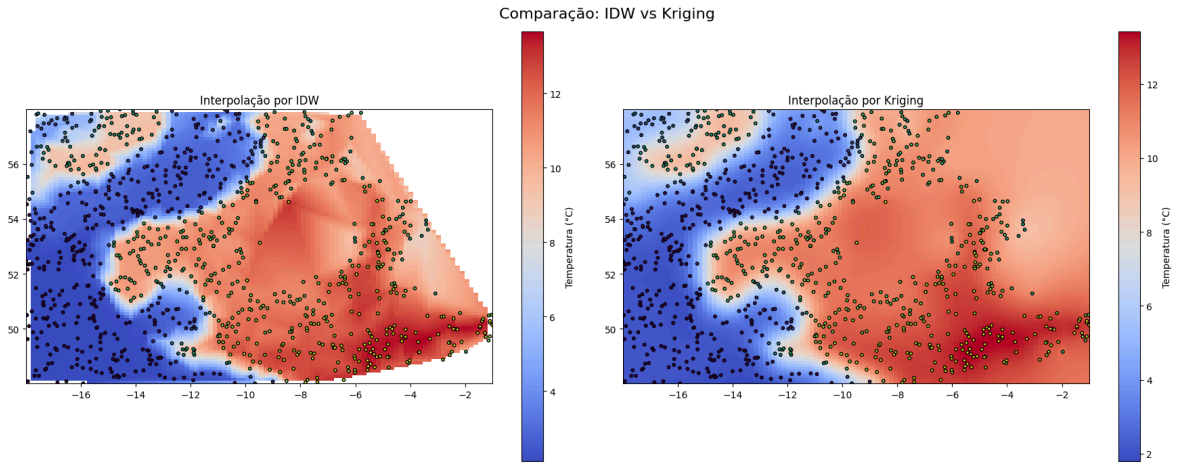
In [32]:
```
# Amostrar 500 pontos para agilizar a krigagem
df_sample = df.sample(n=1000, random_state=42)

# Extrair variáveis
x = df_sample["longitude"].values
y = df_sample["latitude"].values
z = df_sample["sea_bottom_temperature"].values
```

In [33]:
```
# Interpolação com IDW usando griddata
grid_x, grid_y = np.meshgrid(
    np.linspace(x.min(), x.max(), 100),
```

```
    np.linspace(y.min(), y.max(), 100)
)

zi_idw = griddata((x, y), z, (grid_x, grid_y), method='linear')
```

In [34]:
```
OK = OrdinaryKriging(x, y, z, variogram_model='linear', verbose=False, enable_pl
zi_krig, _ = OK.execute("grid", grid_x[0], grid_y[:, 0])
```

In [35]:
```
# Comparação visual
fig, axs = plt.subplots(1, 2, figsize=(18, 7))

# IDW
im1 = axs[0].imshow(zi_idw, extent=(x.min(), x.max(), y.min(), y.max()), origin=
axs[0].set_title("Interpolação por IDW")
axs[0].scatter(x, y, c=z, edgecolors='k', s=10)
plt.colorbar(im1, ax=axs[0], label="Temperatura (°C)")

# Kriging
im2 = axs[1].imshow(zi_krig, extent=(x.min(), x.max(), y.min(), y.max()), origin
axs[1].set_title("Interpolação por Kriging")
axs[1].scatter(x, y, c=z, edgecolors='k', s=10)
plt.colorbar(im2, ax=axs[1], label="Temperatura (°C)")

plt.suptitle("Comparação: IDW vs Kriging", fontsize=16)
plt.tight_layout()
plt.show()
```



A imagem acima compara dois métodos de interpolação espacial aplicados à estimativa de temperaturas em uma região:

- **IDW (Inverse Distance Weighting)** — à esquerda;
- **Kriging** — à direita.

O que os mapas mostram

Ambos os mapas exibem uma superfície contínua de temperatura (em °C) gerada a partir de uma nuvem de pontos amostrados (indicados pelos círculos). As cores representam diferentes valores de temperatura, com tons mais quentes indicando temperaturas mais altas.

Interpolação por IDW

O método IDW estima os valores em locais não amostrados com base em uma média ponderada dos pontos vizinhos, onde os mais próximos têm maior influência. É simples e rápido, mas pode gerar superfícies com **transições abruptas** e **zonas artificiais**, como pode ser visto na granularidade do mapa à esquerda.

Interpolação por Kriging

O Kriging é um método geoestatístico que considera tanto a distância quanto a **estrutura espacial de variância** entre os pontos (modelada via variograma). Isso permite gerar uma superfície mais **suave, contínua e estatisticamente confiável**, como visto no mapa à direita.

Conclusão

- O IDW é útil em aplicações rápidas ou quando não se tem variograma definido;
- O Kriging oferece resultados mais robustos, principalmente em aplicações que exigem precisão (geologia, agricultura de precisão, recursos naturais).

Essa comparação ilustra como a escolha do método de interpolação pode impactar diretamente a interpretação dos dados espaciais.