1 Lecture 6: Maps, graphs, time series

Data Visualization · 1-DAV-105

Lecture by Broňa Brejová

1.1 Package installation

We need to install several packages which are not pre-installed in Colab.

[5]:
```
# uncomment and run the following commands:
#! apt install libgraphviz-dev
#! pip install geoplot pyvis pygraphviz
```

[6]:
```
# importing our usual libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
from IPython.display import Markdown


# new libraries for today:
#  geopandas is a library for working with geographical data
import geopandas as gpd
#  geoplot is a library for visualizing geographical data
import geoplot as gplt
#  we also will use a submodule of plotly
import plotly.graph_objects as go
#  networkx is a library for working with graphs and networks
import networkx as nx
# Pyvis is a library for drawing networks
from pyvis.network import Network
# another library, to be used for drawing graphs as a part of networkx
import pygraphviz
```

1.2 Maps

1.2.1 Introduction

- Each map is a visualization of data about location of objects.
- Maps have a rich set of conventions about colors and symbols, orientation etc. This allows us to quickly understand a map.

Examples:

- A topographic map from US looks similar to maps used in Slovakia, but striped lines are typically used for railroads in Slovakia (example).
- A map of European countries from 1721 can still be easily read by today's audience.

Data visualization in maps

- Maps visualizing data other than typical geographical features are usually called thematic maps (tematické mapy).
- We will see several examples, for others see e.g. Wikipedia, GeoPlot library gallery.
- Also recall Snow's map of cholera cases from the first lecture.

1.2.2 Map projection (kartografické zobrazenie)

- A map projection is a transformation to project the surface of a globe onto a plane.
- Each projection introduces **some distortion**.

Conformal projections preserve local angles, but distort other aspects, such as lengths, areas etc.

- For example, Mercator projection (1569) was developed for navigation, but shows Greenland bigger than Africa, while in fact it is 14x smaller. (See a nice visual comparison.)

Equal-area projections preserve areas (cannot be conformal at the same time).

- Equal-area projections are typically good for data visualization, as they make areas comparable.

Orthographic projection is similar to a photograph of the Earth from a very distant point.

- It is not an equal-area projection, but our sense of perspective may compensate.
- It displays one hemisphere.

Recommended projections (Cairo, The Truthful Art):

- Whole world: e.g. Mollweide equal-area projection (1805)
- Continents / large countries: e.g. Lambert azimuthal equal-area projection (1772)
- Countries in mid-latitudes: e.g. Albers equal-area conic projection (1805)
- Polar regions: e.g. Lambert azimuthal equal-area projection (1772)

Examples of projections in Plotly

- Plotly allows us to set projections for our plot. Here we use it to illustrate the projections on the map of continent outlines.
- The maps are interactive.

[7]:
```python
def show_world(projection, scope=None):
  """A function to display the whole Earth or a desired
  area (scope) using a selected projection. Both arguments
  are strings that name projections or scopes supported by Plotly."""
  # create a map figure with an empty scatterplot
  fig = go.Figure(go.Scattergeo())
  # set the desired projection
  fig.update_geos(projection_type=projection)
  # we can also limit the scope of the map
  if scope is not None:
    fig.update_geos(scope=scope)
  # finally, make the image smaller and with 0 margins
```

```
fig.update_layout(height=200, margin={"r":0,"t":0,"l":0,"b":0})
# show the figure
fig.show()
```

[8]:
```
display(Markdown("**Orthographic projection**"))
show_world("orthographic")
```

Orthographic projection

[9]:
```
display(Markdown("**Mollweide equal-area projection**"))
show_world("mollweide")
```

Mollweide equal-area projection

[10]:
```
display(Markdown("**Mercator conformal projection** (not recommended for data␣
 ↪visualization)"))
show_world("mercator")
```

Mercator conformal projection (not recommended for data visualization)

[11]:
```
display(Markdown("**Lambert azimuthal equal-area projection**"))
show_world("azimuthal equal area", "europe")
```

Lambert azimuthal equal-area projection

1.2.3 Adding data as points and lines to a map

- Geographic coordinates of places can be projected as x and y. Additional values can be shown using marker color and size or line color and width.
- We illustrate this using datasets of airport locations and airline connections.

Importing datasets

- The dataset of international airports of the world was downloaded from the World Bank under the CC-BY 4.0 license, and preprocessed. It includes the number of seats within a year, which is from unknown years, possibly not comparable between countries.
- Our preprocessed file is in GeoJSON format used for describing simple geographical features. It contains both location data and other attributes.
- We parse the file using GeoPandas, which is a library for working with geographical data.
- It is an extension of Pandas DataFrame, with location information.
- Each row of the table contains one airport, with its 3-letter code, name, country, 3-letter code of the country, the number of airplane seats per year and the location.

[12]:
```
display(Markdown("**Importing the list of airports**"))
# parse the file
airports = gpd.read_file("https://fmfi-compbio.github.io/viz/data/airports.
 ↪geojson")
# show the first 5 rows
display(Markdown("**The first five rows:**"), airports.head())
# show the total number of rows
```

```
display(Markdown(f"**The number of rows:** {airports.shape[0]}"))
display(Markdown("**International airports in Slovakia**"))
display(airports.query('Country == "Slovakia"'))
```

Importing the list of airports

The first five rows:

```
  Orig                   Name   TotalSeats      Country ISO3  \
0  HEA                  Herat    22041.971  Afghanistan  AFG
1  JAA              Jalalabad     6343.512  Afghanistan  AFG
2  KBL    Kabul International  1016196.825  Afghanistan  AFG
3  KDH  Kandahar International    39924.262  Afghanistan  AFG
4  MZR         Mazar-e-Sharif    58326.513  Afghanistan  AFG

                  geometry
0  POINT (62.2267 34.2069)
1        POINT (70.5 34.4)
2  POINT (69.2139 34.5639)
3  POINT (65.8475 31.5069)
4  POINT (67.2083 36.7042)
```

The number of rows: 2173

International airports in Slovakia

```
     Orig              Name   TotalSeats   Country ISO3  \
1489  BTS     M.R. Stefanik  1211732.116  Slovakia  SVK
1490  ILZ            Zilina     3986.360  Slovakia  SVK
1491  KSC             Barca   323259.132  Slovakia  SVK
1492  PZY  Piestany Airport     1403.892  Slovakia  SVK
1493  SLD             Sliac    11876.753  Slovakia  SVK
1494  TAT      Tatry/Poprad    39612.286  Slovakia  SVK

                     geometry
1489  POINT (17.2167 48.1667)
1490  POINT (18.7667 49.2333)
1491    POINT (21.25 48.6667)
1492  POINT (17.8333 48.6333)
1493  POINT (19.1333 48.6333)
1494  POINT (20.2403 49.0719)
```

- We will later need a datasets of coutry boundaries proved by Natural Earth.
- Below we estimate country area from its low resolution borders. For example, area of Slovakia estimated as 47070km2, while World Bank lists as 49030km2.

[13]:
```
# read world countries as a dataset provided by Natural Earth
countries = gpd.read_file("https://fmfi-compbio.github.io/viz/data/
↪country_boundaries.geojson")
# set 3-letter code as the index
```

```
countries = countries.set_index('ISO3')
# estimate country area in square km from geometry
# first the geometry is projected by equal-area projection
# the result is in square meters, converted to squared km (divide by 1e6)
# beware that areas are approximate due to low resolution borders
country_areas = countries['geometry'].to_crs({'proj':'cea'}).area  / 1e6
# add areas to countries
countries['Area'] = country_areas

display(Markdown("**Table of countries of the world**"))
display(countries.head())
display(Markdown("**Data for Slovakia**"))
display(countries.loc['SVK', :])
```

Table of countries of the world

```
                   Type                      Name   Population  \
ISO3
FJI   Sovereign country                      Fiji     889953.0
TZA   Sovereign country                  Tanzania   58005463.0
B28       Indeterminate                 W. Sahara     603253.0
CAN   Sovereign country                    Canada   37589262.0
USA             Country  United States of America  328239523.0

                          Region  \
ISO3
FJI          East Asia & Pacific
TZA           Sub-Saharan Africa
B28   Middle East & North Africa
CAN                North America
USA                North America


                                               geometry          Area
ISO3
FJI   MULTIPOLYGON (((180 -16.06713, 180 -16.55522, …  1.928760e+04
TZA   POLYGON ((33.90371 -0.95, 34.07262 -1.05982, 3…  9.327793e+05
B28   POLYGON ((-8.66559 27.65643, -8.66512 27.58948…  9.666925e+04
CAN   MULTIPOLYGON (((-122.84 49, -122.97421 49.0025…  1.003773e+07
USA   MULTIPOLYGON (((-122.84 49, -120 49, -117.0312…  9.509851e+06
```

Data for Slovakia

```
Type                                           Sovereign country
Name                                                    Slovakia
Population                                             5454073.0
Region                                     Europe & Central Asia
geometry      POLYGON ((22.558137648211755 49.08573802346714…
Area                                                47069.779734
Name: SVK, dtype: object
```

All airports as points using Plotly

- We use `scatter_geo` function from Plotly Express.
- We set parts of `geometry` column as latitude and longitude. Column `Name` is used as a tooltip.

[14]:
```
fig = px.scatter_geo(
    airports,
    lat=airports.geometry.y,
    lon=airports.geometry.x,
    hover_name="Name",
    projection="mollweide"
    )
fig.update_layout(height=300, margin={"r":0,"t":0,"l":0,"b":0})
fig.show()
```

Adding the size of the airport

- We use size of the circle to represent the number of seats.
- Scatterplots with point sizes are often called bubble graphs.
- We focus on Europe, add country borders and change the projection.

[15]:
```
fig = px.scatter_geo(
    airports,
    lat=airports.geometry.y,
    lon=airports.geometry.x,
    size="TotalSeats",
    hover_name="Name"
    )
fig.update_geos(
    projection_type="azimuthal equal area",
    lonaxis_range= [-20, 40],
    lataxis_range= [20, 70],
    showcountries = True
)
fig.update_layout(height=300, margin={"r":0,"t":0,"l":0,"b":0})
fig.show()
```

Airline connections from Slovakia as lines

- We import another table (originating from World bank as above), which shows international airline connections from Slovak airports (in an unknown year).
- Each connection is given by two airport codes, the number of airplane seats within a year, and geometry with a segment connecting the two airport locations.
- Line color will correspond to the airport of origin in Slovakia.
- The code is adapted from examples in the Plotly documentation.

[16]:
```
display(Markdown("**Importing airline connections**"))
connections = gpd.read_file("https://fmfi-compbio.github.io/viz/data/
↪airport_pairs_svk.geojson")
display(connections.head())
```

Importing airline connections

```
  OrigCode DestCode  TotalSeats                                        geometry
0      BTS      ADB    7370.433  LINESTRING (17.2167 48.1667, 27.1562 38.2943)
1      BTS      AGP   15152.501  LINESTRING (17.2167 48.1667, -4.4981 36.6717)
2      BTS      AHO   14740.866   LINESTRING (17.2167 48.1667, 8.2889 40.6306)
3      BTS      AQJ    3275.748  LINESTRING (17.2167 48.1667, 35.0194 29.6125)
4      BTS      ATH   19654.488  LINESTRING (17.2167 48.1667, 23.9444 37.9364)
```

[17]:
```
def draw_lines(connections):
  # create two lists with x and y coordinates of polylines
  # separated by None
  lats = []
  lons = []
  # also create lists of origin and destination codes parallel to lists above
  origCodes = []
  destCodes = []

  # iterate through table rows
  for index, row in connections.iterrows():
    # get lists of x and y coordinates (of length 2 in this case)
    x, y = row['geometry'].xy
    # add coordinates and None separator to lists
    lats.extend(list(y) + [None])
    lons.extend(list(x) + [None])
    # add airport codes for each coordinate and None separator
    origCodes.extend([row['OrigCode']] * len(x) + [None])
    destCodes.extend([row['DestCode']] * len(x) + [None])

  # create figure with these lists
  fig = px.line_geo(lat=lats, lon=lons, hover_name=destCodes, color=origCodes)
  # setup projection
  fig.update_geos(
    projection_type="azimuthal equal area",
    lonaxis_range= [-25, 55],
    lataxis_range= [10, 60],
    showcountries = True
  )
  fig.update_layout(height=300, margin={"r":0,"t":0,"l":0,"b":0})
  fig.show()

# call the function to draw the map
display(Markdown("**Airline connections from Slovak airports**"))
```

```
draw_lines(connections)
```

Airline connections from Slovak airports

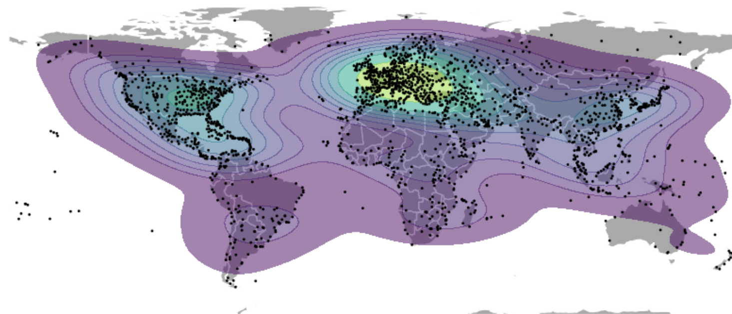
1.2.4 Isarithmic maps / isoline maps / heatmaps

- These maps display a continuous variable over the map area (elevation, temperature and other weather phenomena etc.).
- Value in each point can be shown by a color scale.
- Some contour lines can be displayed as well.
- A contour line (isoline, isopleth, isarithm, izočiara) connects points of the same value.
- Example: short-term forecasts from the Slovak Hydrometeorological Institute.

Density of airports

- Here we show world airports as both points and their local density as a isarithmic map.
- This is achieved using `kdeplot` function from the Geoplot library.
- KDE stands for kernel density estimation, and we will explain it in the next lecture.

[18]:
```
# plot countries as a background
ax = gplt.polyplot(
    countries.explode(index_parts=True),
    edgecolor='white',
    facecolor='darkgray',
    figsize=(10, 5),
)
# plot semi-transparent isarithmic map
gplt.kdeplot(
    airports, cmap='viridis',
    fill=True, alpha=0.5, ax=ax
)
# plot points on top
gplt.pointplot(airports, s=1, color='black', ax=ax)
pass
```



1.2.5 Choropleth maps (kartogramy)

- Often we have numerical / categorical values for administrative regions (countries, districts, etc.).
- Choropleth maps show such variables via colors applied to the whole region.

Variables over regions:

- **Spatially extensive (extenzitné)** variables apply to the unit as a whole (e.g. total population, area, the number of airports in the country). If we subdivide the region, spatially extensive variable will be often the sum of its parts (but not always, e.g. perimeter)
- **Spatially intensive (intenzitné)** variables may stay the same if you divide the unit, provided the unit is homogeneous without regional differences. Examples include population density, life expectancy, GDP per person.
- Spatially extensive variables are not appropriate for choropleths, because large value for a large country is visually attributed to each small subregion of the country. If counts are of interest, better use a bubble graph with marker of appropriate size in the region center.

Beware:

- A choropleth map is called kartogram in Slovak.
- English word **cartogram** means a map with regions rescaled according to some variable (such as the Levasseur's cartogram of country budgets and a modern example).

Choropleth maps of airports per country We will show three choropleth maps:

- the number of airports per 10000 km2 (spatially intensive variable),
- the number of airports per million inhabitants (also spatially intensive),
- the number of airports (spatially extensive, not recommended for choropleth).

We will also show the number of airports as a bubble graph (more appropriate than extensive variable).

All choropleth maps are created by Plotly. The bubble graph is also created by Plotly, and the bubble is placed to the representative point of each country.

[19]:
```
# compute the number of airports per country by groupby
airports_per_country = airports.groupby('ISO3').size()
# add the new column to a copy of the old table
countries2 = countries.copy(deep=True)
# add the number of airports as a new column
countries2['Airports'] = airports_per_country
# remove countries where airports or location are missing
countries2.dropna(subset=['geometry', 'Airports'], inplace=True)
# add columns with airport density and airports per million people
countries2['Airport_density'] = (countries2['Airports']
                                 / countries2['Area'] * 10000)
countries2['Airports_per_mil'] = (countries2['Airports']
                                  / countries2['Population'] * 1e6)
# show the new table
display(Markdown("**The first five rows of `countries2` table:**"))
```

```
display(countries2.head())
display(Markdown("**The values for Slovakia:**"))
display(countries2.loc['SVK'])
```

The first five rows of countries2 table:

```
                   Type                      Name   Population  \
ISO3
FJI   Sovereign country                      Fiji     889953.0
TZA   Sovereign country                  Tanzania   58005463.0
CAN   Sovereign country                    Canada   37589262.0
USA             Country  United States of America  328239523.0
KAZ         Sovereignty                Kazakhstan   18513930.0

                     Region  \
ISO3
FJI     East Asia & Pacific
TZA      Sub-Saharan Africa
CAN           North America
USA           North America
KAZ   Europe & Central Asia

                                               geometry          Area  \
ISO3
FJI   MULTIPOLYGON (((180 -16.06713, 180 -16.55522, …  1.928760e+04
TZA   POLYGON ((33.90371 -0.95, 34.07262 -1.05982, 3…  9.327793e+05
CAN   MULTIPOLYGON (((-122.84 49, -122.97421 49.0025…  1.003773e+07
USA   MULTIPOLYGON (((-122.84 49, -120 49, -117.0312…  9.509851e+06
KAZ   POLYGON ((87.35997 49.21498, 86.59878 48.54918…  2.728701e+06

      Airports  Airport_density  Airports_per_mil
ISO3
FJI        2.0         1.036935          2.247310
TZA        7.0         0.075045          0.120678
CAN       82.0         0.081692          2.181474
USA      291.0         0.305998          0.886548
KAZ       17.0         0.062301          0.918228
```

The values for Slovakia:

```
Type                                                  Sovereign country
Name                                                           Slovakia
Population                                                    5454073.0
Region                                            Europe & Central Asia
geometry            POLYGON ((22.558137648211755 49.08573802346714…
Area                                                       47069.779734
Airports                                                            6.0
Airport_density                                                1.274703
Airports_per_mil                                               1.100095
```

```
Name: SVK, dtype: object
```

[20]:
```
def draw_choropleth(data, column, range_color=None, label=None):
  fig = px.choropleth(
    data, locations=data.index, color=column,
    range_color=range_color,
    labels={column:label},
    hover_name="Name",
    projection = "mollweide"
  )
  fig.update_layout(height=300, margin={"r":0,"t":0,"l":0,"b":0})
  fig.show()

display(Markdown("**The number of airports per 10000 squared km**"))
draw_choropleth(countries2, 'Airport_density', (0, 2), 'airports / 10000 km2')
```

The number of airports per 10000 squared km

[21]:
```
display(Markdown("**The number of airports per million inhabitans**"))
draw_choropleth(countries2, 'Airports_per_mil', (0, 5), 'airports / million␣
 ↪people')
```

The number of airports per million inhabitans

[22]:
```
display(Markdown("**The number of airports in a country**"))
draw_choropleth(countries2, 'Airports', (0, 100), 'airports')
```

The number of airports in a country

[23]:
```
# make a new table of countries in which geometry is replaced
# with a single representative point
countries3 = countries2.copy(deep=True)
countries3['geometry'] = countries2['geometry'].representative_point()

# plot as a bubble plot
display(Markdown("**The number of airports in a country**"))
fig = px.scatter_geo(
    countries3,
    lat=countries3.geometry.y,
    lon=countries3.geometry.x,
    size="Airports",
    hover_name="Name",
    projection = "mollweide"
    )
fig.update_geos(showcountries = True)
fig.update_layout(height=300, margin={"r":0,"t":0,"l":0,"b":0})
fig.show()
```

The number of airports in a country

1.2.6 Summary of maps

- Many data sets contain geographic entities (countries, cities, coordinates).
- Displaying such data on maps highlights spatial relationships.
- In your maps, use appropriate equal-area projections.

Types of thematic maps:

- Bubble plots contain points of various sizes and colors.
- Isarithmic maps / isoline maps / heatmaps display continuously varying variables, such as elevation or temperature using color scales or isolines.
- Choropleth maps display variables characterizing whole region using colors. The variable used in choropleth should be intensive, e.g. normalized by area or population.

Several useful libraries:

- Geopandas for working with geographical data, extension of DataFrame
- Geoplot and Plotly for visualization

1.3 Graphs and hierarchies

1.3.1 Graphs

- A **graph / network** consists of **vertices** (vrcholy; also nodes, uzly) and **edges** (hrany; also links, arcs).
- Vertices often represent real-world **entities**.
- Edges often represent **relationships** and connections between pairs of vertices.

Many real-world examples of graphs: places connected by roads, computers connected by network cables, people connected by family or work relationships, companies connected by financial transactions, texts connected by references, tasks or courses connected by dependencies, any objects connected based on similarity / shared features.

- Edges can be directed (orientované) or undirected depending on whether the relationship is symmetrical.
- Recall: how did you define directed / undirected edges in discrete mathematics?

Graphs are very important in both computer science and data science. They are covered in several courses: discrete mathematics, programming, design of efficient algorithms, network science.

1.3.2 Trees and hierarchies

- An undirected graph is called a **tree** if it is connected and without cycles.
- In practice we usually encounter rooted (directed) trees, which have a single **root**, all other vertices can be reached from the root via a unique path.
- This gives rise to parent / child relationships between nodes (parent is the node closer to the root).
- Trees can express hierarchies in which each entity has a single direct superior, for example:
  - company structure in which each employee (except for the head of the company) has a single supervisor (similarly army command),
  - administrative divisions (country, region, district),
  - species taxonomy (animals, mammals, primates, …).
- However some hierarchies allow multiple direct superiors, for example:

  - family tree where each person has two parents (and they may be distantly related),
  - geometrical shapes, where a square is both a special case of a regular polygon and a special case of a rectangle and both of these are a special case of a polygon.
- These hierarchies can be represented as directed acyclic graphs.
  - Acyclic means that by following edges we never get back to the starting node (nobody is their own ancestor).

1.3.3 What do we study / visualize in real-life graphs?

- Details of connections for a particular node (requires zooming in large networks).
- Overall structure of the graph: connected components, density of edges, presence of cycles, weak places (bridges and articulations), clusters of densely connected nodes.
- Do nodes with some property cluster together? (Are they connected by many edges?)

See for example character co-occurrence in Shakespeare's tragedies.

1.3.4 Basics of graph drawing

- Vertices are typically displayed as markers (circles, rectangles etc.), possibly with labels, size, color, …
- Edges are displayed as lines connecting them, possibly of different color or width. They can be straight lines, arcs, polygonal lines or arbitrary curves.
- Edge direction displayed as arrows or in a hierarchy edges may be drawn to point in one direction, e.g. downwards.

Desirable properties:

- Nodes do not overlap.
- Edges are not too long and have a simple shape without many bends.
- The number of edge crossings is small.
- The graph uses the space of the figure well without large empty regions.

Node positioning:

- Sometimes the position of nodes is given by their properties, e.g. on a map (see airline connections), level of a hierarchy, timeline.
- Otherwise we try to place nodes to optimize desirable properties, e.g. using force-directed layout, which assigns attractive forces (springs) between nodes connected by edges and repulsive forces between other pairs of nodes.

Examples:

- https://en.wikipedia.org/wiki/Graphviz#/media/File:UnitedStatesGraphViz.svg
- https://upload.wikimedia.org/wikipedia/commons/9/90/Visualization_of_wiki_structure_using_prefuse_

1.3.5 Displaying a simple hierarchy in NetworkX

- We start by creating a simple tree representing taxonomy of selected even-toed ungulates (párnokopytníky) as a Pandas `DataFrame`.
- Each row of the data frame describes one node, giving its name, parent, and category, which is `land` for land animals, `sea` for sea animals and `group` for taxonomy groups.
- Group `Artiodactyla` is the root without a parent.

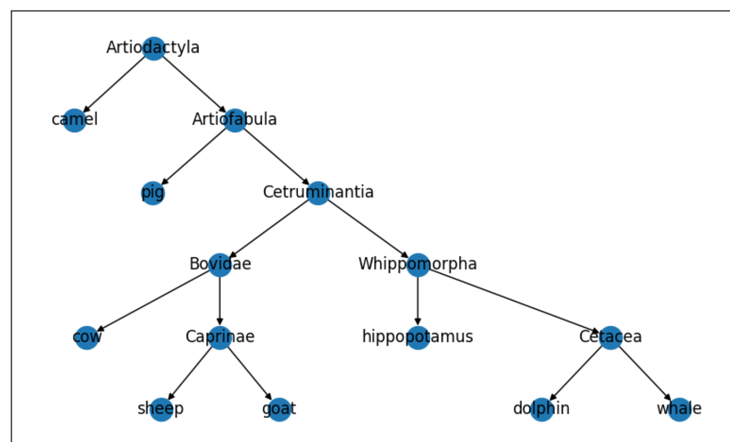
[24]:
```python
from io import StringIO

animal_csv = StringIO("""name,parent,category
camel,Artiodactyla,land
pig,Artiofabula,land
sheep,Caprinae,land
goat,Caprinae,land
cow,Bovidae,land
dolphin,Cetacea,sea
whale,Cetacea,sea
hippopotamus,Whippomorpha,land
Caprinae,Bovidae,group
Cetacea,Whippomorpha,group
Bovidae,Cetruminantia,group
Whippomorpha,Cetruminantia,group
Cetruminantia,Artiofabula,group
Artiofabula,Artiodactyla,group
Artiodactyla,,group""")

animals = pd.read_csv(animal_csv)
animals['category'] = animals['category'].astype('category')
display(animals)
```

	name	parent	category
0	camel	Artiodactyla	land
1	pig	Artiofabula	land
2	sheep	Caprinae	land
3	goat	Caprinae	land
4	cow	Bovidae	land
5	dolphin	Cetacea	sea
6	whale	Cetacea	sea
7	hippopotamus	Whippomorpha	land
8	Caprinae	Bovidae	group
9	Cetacea	Whippomorpha	group
10	Bovidae	Cetruminantia	group
11	Whippomorpha	Cetruminantia	group
12	Cetruminantia	Artiofabula	group
13	Artiofabula	Artiodactyla	group
14	Artiodactyla	NaN	group

- NetworkX is a large library for working with graphs, it implements many graph algorithms.
- Below we convert our DataFrame to `DiGraph` class (directed graph) from NetworkX by adding nodes and edges.
- Nodes and edges can have arbitrary attributes attached, here `category`.
- Then we plot a basic representation of the graph.
- The plotting works in two steps: first we compute coordinates of all nodes using `graphviz_layout`.
- Then we plot the network using `draw_networkx` into Matplotlib axes.

- The plot is not very nice, we will improve it below.

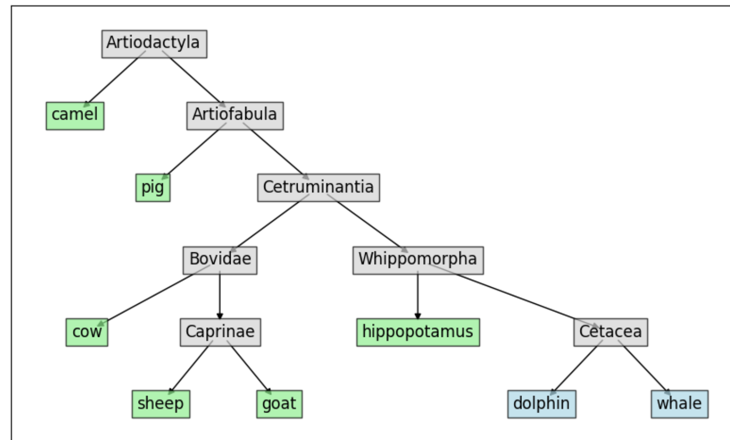
[25]:
```python
# create empty directed graph in NetworkX
G = nx.DiGraph()

# adding each table row as a node
for index, row in animals.iterrows():
  G.add_node(row['name'], category=row['category'])

# adding an edge to each node from its parent
for index, row in animals.iterrows():
  if row['parent'] is not np.nan:
    G.add_edge(row['parent'], row['name'])

# computing coordinates of nodes
coordinates = nx.nx_agraph.graphviz_layout(G, prog="dot")

# drawing the graph
(figure, axes) = plt.subplots(figsize=(10, 6))
nx.draw_networkx(G, coordinates, ax=axes)

pass
```



- Below we show an improved version of the plot.
- Edges are plotted first.
- Then we draw node lables as boxes with text.
- Each category of nodes is draw separately with a different background color.

[26]:
```python
# plot edges only, omit nodes for now
(figure, axes) = plt.subplots(figsize=(10, 6))
nx.draw_networkx_edges(G, coordinates, ax=axes)
```

```
# plot each category of nodes by a different color
color_dict = {'group':'lightgray', 'land':'lightgreen', 'sea':'lightblue'}
for category in color_dict:
  # create a list of nodes in the category
  category_nodes = [v for v in G.nodes if G.nodes[v]['category']==category]
  # select subgraph H of G
  H = G.subgraph(category_nodes)
  # create a dictionary of node label attributes
  label_options = {"ec": "black", "fc": color_dict[category], "alpha": 0.7}
  # draw the node labels as boxes
  nx.draw_networkx_labels(H, coordinates, font_size=12, bbox=label_options,␣
 ↪ax=axes)

pass
```



1.3.6 Hierarchy as a treemap in Plotly Express

- PlotlyExpress can be used to easily create treemaps.
- Leaves of the tree are empty labeled rectangles, upper categories are enclosing boxes.
- To select box colors, we use `color_dict` created above.

[27]:
```
import plotly.express as px
fig = px.treemap(
    names=animals['name'],
    parents=animals['parent'],
    color=animals['category'],
    color_discrete_map=color_dict
)
fig.show()
```

1.3.7 Book character connections in Pyvis

- Here we use an example network from the NetworkX library.
- It represents character co-occurence in the novel Les Misérables by Victor Hugo.
- Edges are weighted by how often characters co-occur.
- To make the plot interactive, we use Pyvis library.
- We convert NetworkX graph to `Network` class from Pyvis.
- We add two new features for each node: `title` (used as a tooltip, name of the character) and `value` (used as a size of the node, representing its number of neighbors, i.e. degree).
- Visualization is saved as a HTML file which is then displayed using HTML class from IPython library.

[28]:
```
# initializing an empty network, setup plot properties
pyvis_net = Network("500px", "500px", notebook=True, cdn_resources='in_line')
# loading network from NetworkX
pyvis_net.from_nx(nx.les_miserables_graph())

# get a dictionary of neighbors for each node
neighbors = pyvis_net.get_adj_list()
# add additional node properties
# used as tooltip and size
for node in pyvis_net.nodes:
    node["title"] = node["id"]
    node["value"] = len(neighbors[node["id"]])

# saving the visualization in an html file
pyvis_net.show("net.html")
# displaying the html file in the notebook
from IPython.display import display, HTML
display(HTML('net.html'))
pass
```

```
net.html
```

```
<IPython.core.display.HTML object>
```

1.3.8 Summary of graphs

- Graphs are important in many applications.
- NetworkX library has many functions for working with graphs, including several layout algorithms for visualization.
- Pyvis allows interactive visualization of graphs.
- Plotly can visualize trees as treemaps.

1.4 Time series (časové rady)

- Time series are sequences of measurements or values over time (in regular or irregular time intervals).
- Typically displayed as a line graph, with time as x-axis, time flowing from left to right (a cultural convention in western countries).

- Other options for drawing time series exist (bar graphs, heat maps, box plots, …).

Typical features of a time series:

- overall trend (increasing / decreasing / flat; rate of change),
- seasonality (daily / weekly / yearly cycles),
- noise (general variability / outliers)

We have seen some examples in the first lecture:

- Playfair's atlas, foreign trade
- Hockey stick graph of global temperature (Fig.3a)

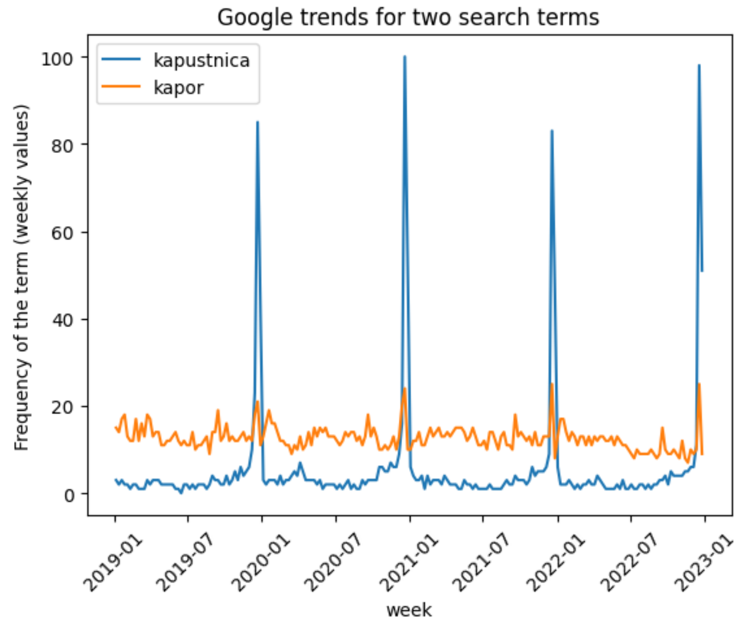
1.4.1 Two Google trend time series

- Google trends allow users to compare frequency of search terms over time and to each other.
- Here we use Christmas-related terms *kapustnica* and *kapor*.

[29]:
```
url = "https://fmfi-compbio.github.io/viz/data/kapustnica-kapor.csv"
trends1 = pd.read_csv(url, parse_dates=['week']).set_index('week')
display(trends1.head())
```

```
            kapustnica  kapor
week
2019-01-06           3     15
2019-01-13           2     14
2019-01-20           3     17
2019-01-27           2     18
2019-02-03           2     13
```

[30]:
```
axes = sns.lineplot(trends1, dashes=False)
axes.set_ylabel("Frequency of the term (weekly values)")
axes.set_title("Google trends for two search terms")
# rotate tick labels
axes.tick_params(axis='x', labelrotation = 45)
pass
```



- With kapustnica we see a clear seasonal trend.
- But kapor behaves differently. As related search terms Google reports zbgis kataster, zbgis mapa, zgbis, katasterportal list vlastnictva, dážďovka. Can you explain this?

1.4.2 Smoothing data (vyrovnanie, vyhladenie)

- Time series above is measured weekly and is quite noisy.
- We can smooth the data e.g. by **aggregating** them in longer time intervals. Here we compute mean value in each month (4 or 5 weeks).
- This is done using `resample` method from Pandas.
- An alternative is to use a **sliding window** (kĺzavé okno), where we choose a window size. e.g. 4 weeks and compute a new series, each value being mean or other summary of 4 consecutive windows in the input.
- For example with values 2,6,4,2,8,2 and window size 4, we get window means 3.5, 5, 4.

[31]:
```
# aggregate google trends from weekly to monthly mean (ME means month-end, in␣
↪older versions use M)
trends1monthly = trends1.resample('ME').mean()
display(Markdown("**New table of monthly means** (the first 5 rows)"))
display(trends1monthly.head())
display(Markdown("**The number of values aggregated in each month** (the first␣
↪5 values)"))
display(trends1.resample('ME').size().head())
```

New table of monthly means (the first 5 rows)

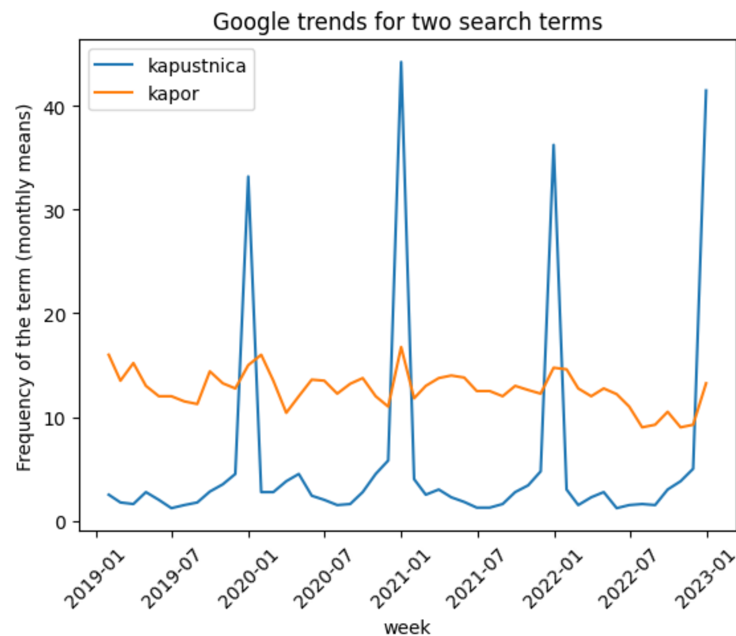
```
               kapustnica  kapor
week
```

```
2019-01-31         2.50    16.0
2019-02-28         1.75    13.5
2019-03-31         1.60    15.2
2019-04-30         2.75    13.0
2019-05-31         2.00    12.0
```

The number of values aggregated in each month (the first 5 values)

```
week
2019-01-31    4
2019-02-28    4
2019-03-31    5
2019-04-30    4
2019-05-31    4
Freq: ME, dtype: int64
```

[32]:
```
axes = sns.lineplot(trends1monthly, dashes=False)
axes.set_ylabel("Frequency of the term (monthly means)")
axes.set_title("Google trends for two search terms")
axes.tick_params(axis='x', labelrotation = 45)
pass
```



1.4.3 Trend: temperatures are growing in spring

- We will also look at a dataset displaying a trend: series of temperature values from Piešťany from January to June 2010, downloaded from US National Oceanic and Atmospheric Administration.
- We show both the original data and values smoothed with rolling average.

[33]:
```
# read a dataset of temperatures in Piestany
url="https://fmfi-compbio.github.io/viz/data/piestany-weather.csv"
weather = pd.read_csv(url, parse_dates=['DATE']).set_index('DATE')
# select only columns with daily maximum temperatures
temperature = weather["TMAX"]
# select only period from January to June 2010
spring2010 = temperature[pd.Timestamp('2010-01-01'):pd.Timestamp('2010-06-30')]
display(spring2010)
```

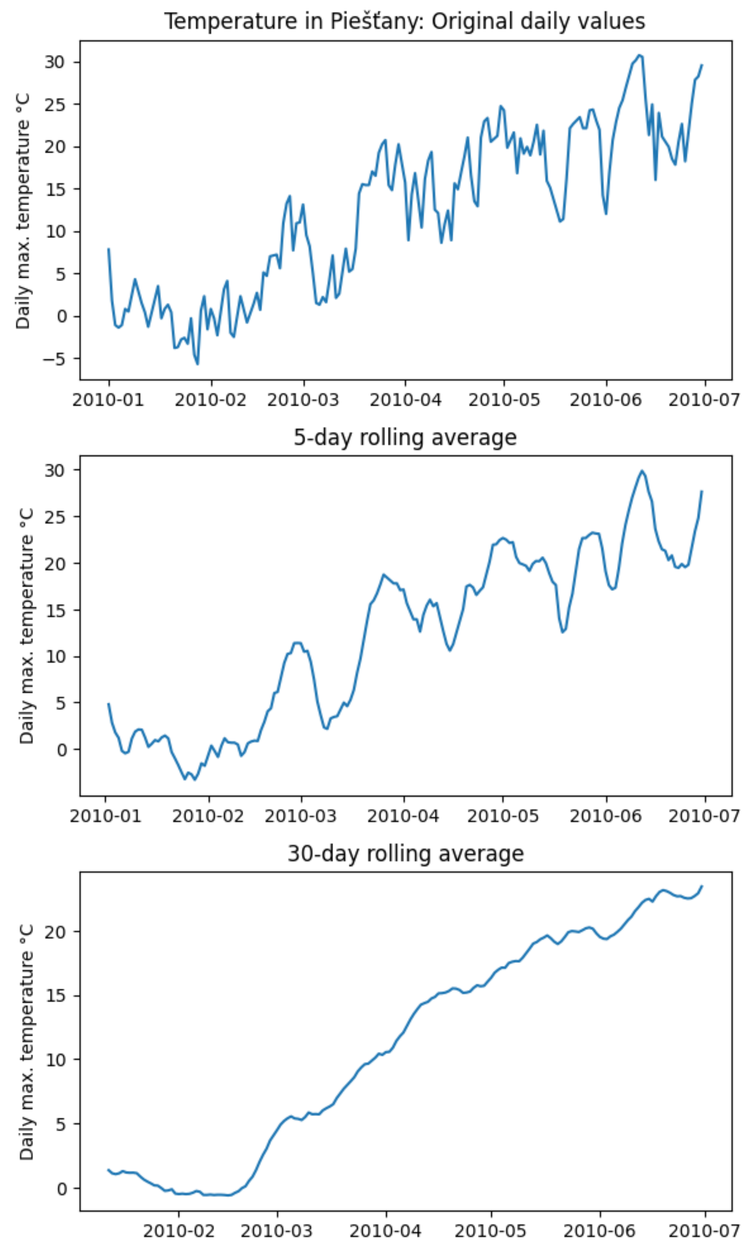
```
DATE
2010-01-01     7.8
2010-01-02     1.8
2010-01-03    -1.1
2010-01-04    -1.4
2010-01-05    -1.1
              …
2010-06-26     NaN
2010-06-27    25.0
2010-06-28    27.8
2010-06-29    28.2
2010-06-30    29.5
Name: TMAX, Length: 181, dtype: float64
```

[34]:
```
# compute rolling averages in a window of 5 and 30 days
spring2010rolling5 = spring2010.rolling(5, min_periods=2).mean()
spring2010rolling30 = spring2010.rolling(30, min_periods=10).mean()
spring2010rolling5.head(10)
```

[34]:
```
DATE
2010-01-01         NaN
2010-01-02    4.800000
2010-01-03    2.833333
2010-01-04    1.775000
2010-01-05    1.200000
2010-01-06   -0.200000
2010-01-07   -0.460000
2010-01-08   -0.300000
2010-01-09    1.125000
2010-01-10    1.866667
Name: TMAX, dtype: float64
```

[35]:
```
(figure, axes) = plt.subplots(3, 1, figsize=(6, 10))
sns.lineplot(spring2010, ax=axes[0])
sns.lineplot(spring2010rolling5, ax=axes[1])
sns.lineplot(spring2010rolling30, ax=axes[2])
axes[0].set_title("Temperature in Piešťany: Original daily values")
axes[1].set_title("5-day rolling average")
axes[2].set_title("30-day rolling average")
```

```python
for i in range(3):
  axes[i].set_ylabel("Daily max. temperature °C")
  axes[i].set_xlabel(None)
figure.tight_layout(pad=1.0)
pass
```



1.4.4 Overlapping timescales to display seasonality

- We can better see cyclical trends if we plot each cycle on the same x-axis scale.

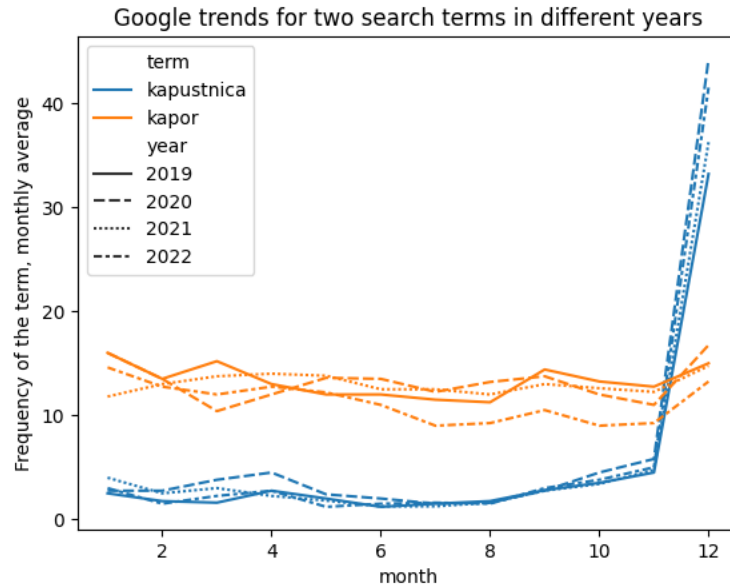
- In our Google example, we will use the month as the x axis and plot individual years as lines.

[36]:
```
# convert monthly table to long format with separate rows for kapustnica and␣
↪kapor
trends1monthlyLong = trends1monthly.reset_index().melt(id_vars=['week'])
trends1monthlyLong.rename(columns={'variable':'term', 'value':'frequency'},␣
↪inplace=True)
# create separate columns with year and month
trends1monthlyLong['month'] = trends1monthlyLong['week'].dt.month
trends1monthlyLong['year'] = trends1monthlyLong['week'].dt.year
display(Markdown("**Monthly table in the long format**"))
display(trends1monthlyLong)
```

Monthly table in the long format

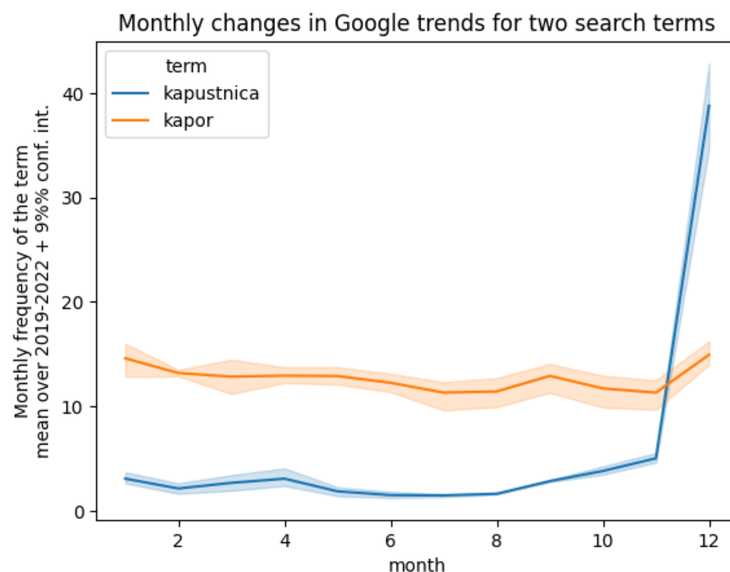
```
          week        term  frequency  month  year
0   2019-01-31  kapustnica       2.50      1  2019
1   2019-02-28  kapustnica       1.75      2  2019
2   2019-03-31  kapustnica       1.60      3  2019
3   2019-04-30  kapustnica       2.75      4  2019
4   2019-05-31  kapustnica       2.00      5  2019
..         …           …         …      …     …
91  2022-08-31       kapor       9.25      8  2022
92  2022-09-30       kapor      10.50      9  2022
93  2022-10-31       kapor       9.00     10  2022
94  2022-11-30       kapor       9.25     11  2022
95  2022-12-31       kapor      13.25     12  2022

[96 rows x 5 columns]
```

[37]:
```
# use month as x, separate years by line style and search terms by color
axes = sns.lineplot(trends1monthlyLong, x='month', y='frequency', hue='term',␣
↪style='year')
axes.set_title("Google trends for two search terms in different years")
axes.set_ylabel("Frequency of the term, monthly average")
pass
```



- We can see that across years the trend is quite stable.
- Below we see another version of the figure where multiple lines for years are replaced with mean and its 95% confidence interval expressing our uncertainty in the true value of the mean due to noise in data.

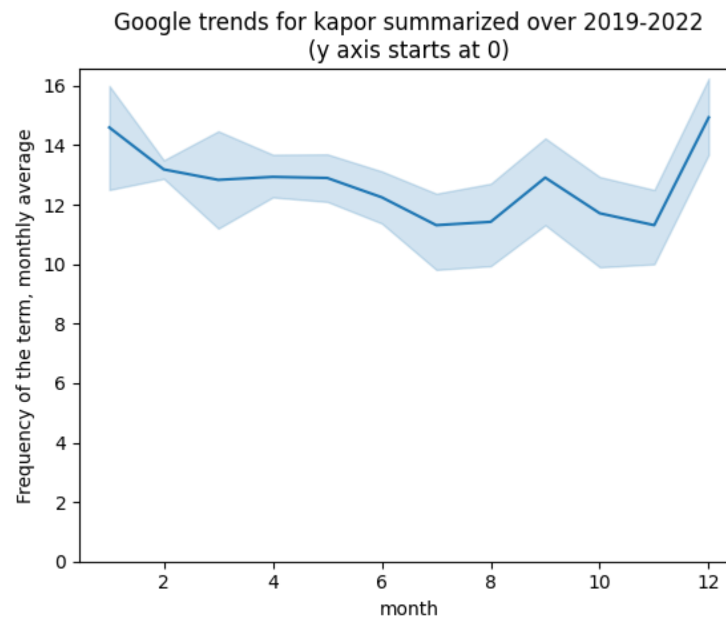
[38]:
```
axes = sns.lineplot(trends1monthlyLong, x='month', y='frequency', hue='term')
axes.set_title("Monthly changes in Google trends for two search terms")
axes.set_ylabel("Monthly frequency of the term\nmean over 2019-2022 + 9%% conf.␣
↪int.")
pass
```



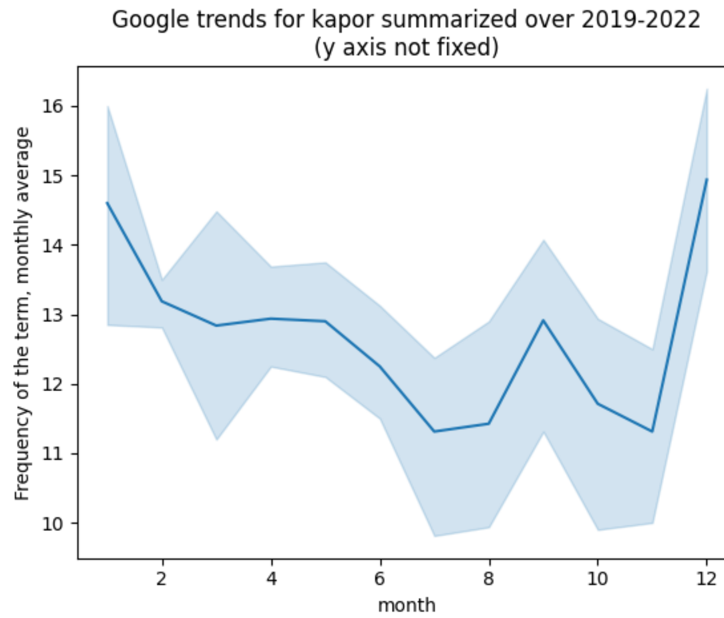
1.4.5 Importance of scales

- The plot below shows that if we do not start y axis at 0, differences in kapor searches may appear exaggerated.
- The next two plots show that even with y axis starting at 0, the time series may appear more variable with narrower aspect ratio of the figure.

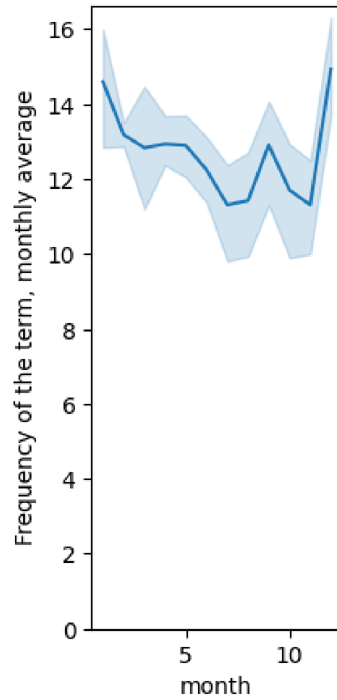
[39]:
```
kapor = trends1monthlyLong.query("term=='kapor'")
axes = sns.lineplot(kapor, x='month', y='frequency')
axes.set_title("Google trends for kapor summarized over 2019-2022\n(y axis␣
↪starts at 0)")
axes.set_ylabel("Frequency of the term, monthly average")
axes.set_ylim(ymin=0)
pass
```



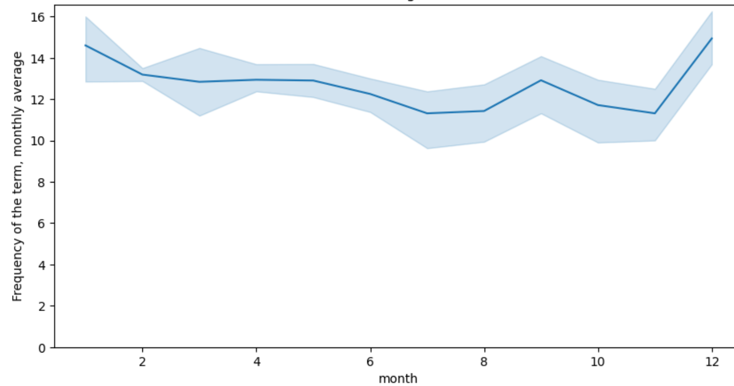
[40]:
```
axes = sns.lineplot(kapor, x='month', y='frequency')
axes.set_title("Google trends for kapor summarized over 2019-2022\n(y axis not␣
↪fixed)")
axes.set_ylabel("Frequency of the term, monthly average")
pass
```



[41]:
```
(figure, axes) = plt.subplots(figsize=(2,5))
sns.lineplot(kapor, x='month', y='frequency', ax=axes)
axes.set_title("Google trends for kapor summarized over 2019-2022\n(narrow␣
↪figure)")
axes.set_ylabel("Frequency of the term, monthly average")
axes.set_ylim(ymin=0)
pass
```



[42]:
```
(figure, axes) = plt.subplots(figsize=(10,5))
sns.lineplot(kapor, x='month', y='frequency', ax=axes)
axes.set_title("Google trends for kapor summarized over 2019-2022\n(wide␣
↪figure)")
axes.set_ylabel("Frequency of the term, monthly average")
axes.set_ylim(ymin=0)
pass
```



1.4.6 Relative scales

- When we care about rate of increase or decrease, it might be better to express values as a percentage compared to initial value.
- Here we compare values in each month with values in January of the same year.
- In this way even two time series with quite different values can be plotted in the same plot (e.g. revenue of a small and a large company and their relative changes within a year).

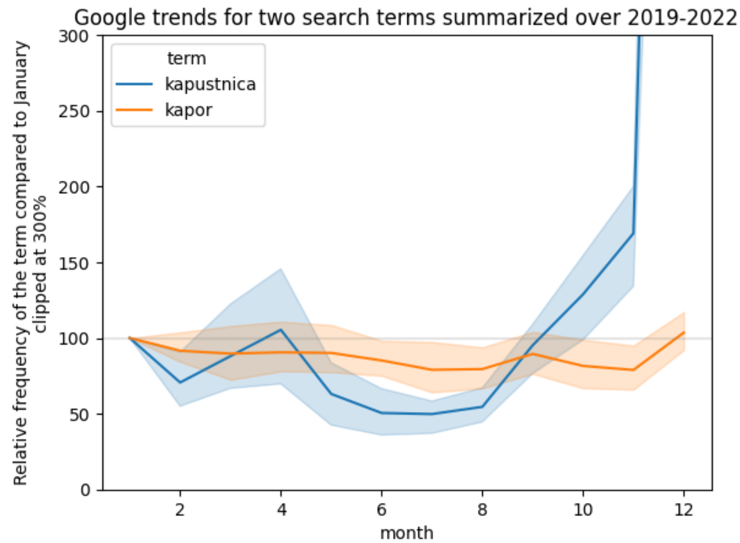
[43]:
```
# compute relative values by transforming each group of monthly values
# by dividing them by the first value (January)
relValue = (trends1monthlyLong.groupby(['year','term'])['frequency']
            .transform(lambda x : x * 100 / x.iloc[0]))
# add relative values as a column to the long table
relTable = trends1monthlyLong.assign(relValue=relValue)
display(Markdown("**Relative values added**"))
relTable.head()
```

Relative values added

[43]:

	week	term	frequency	month	year	relValue
0	2019-01-31	kapustnica	2.50	1	2019	100.0
1	2019-02-28	kapustnica	1.75	2	2019	70.0
2	2019-03-31	kapustnica	1.60	3	2019	64.0
3	2019-04-30	kapustnica	2.75	4	2019	110.0
4	2019-05-31	kapustnica	2.00	5	2019	80.0

[44]:
```
axes = sns.lineplot(relTable, x='month', y='relValue', hue='term')
axes.set_ylim(ymin=0, ymax=300)
axes.axhline(100, color="gray", alpha=0.2)
axes.set_title("Google trends for two search terms summarized over 2019-2022")
axes.set_ylabel("Relative frequency of the term compared to January\nclipped at␣
 ↪300%")

pass
```



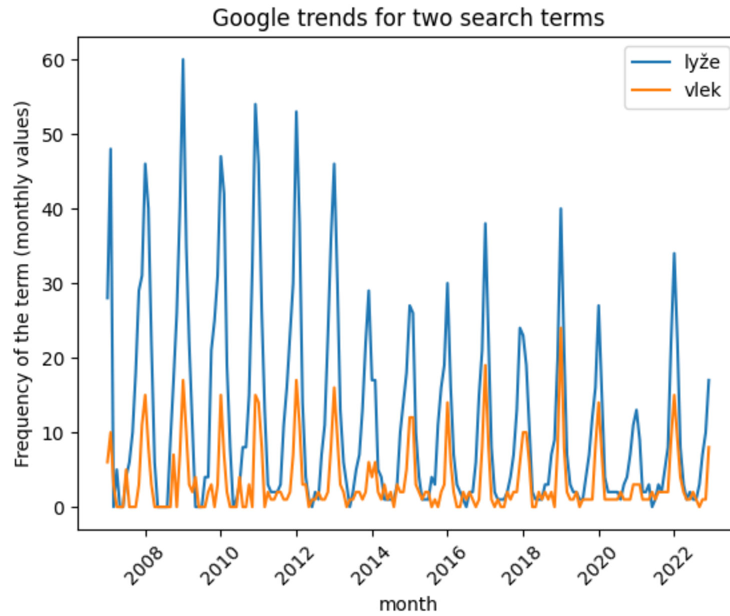
1.4.7 One more pair of Google trend lines

- Again very seasonal: lyže and vlek.
- This time we have monthly data over a longer period of time.
- We display the original data as well as yearly seasonal trend.
- The peak month is January for both queries, so we also display January values changing over the years.

[45]:
```
url = "https://fmfi-compbio.github.io/viz/data/lyze-vlek.csv"
trends2 = pd.read_csv(url, parse_dates=['month']).set_index('month')
display(trends2.head())
```

	lyže	vlek
month		
2007-01-01	28	6
2007-02-01	48	10
2007-03-01	0	3
2007-04-01	5	0
2007-05-01	0	0

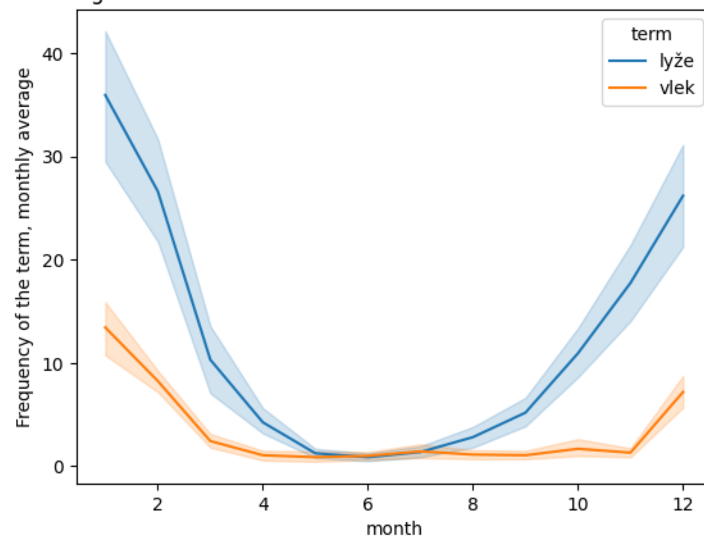
[46]:
```
axes = sns.lineplot(trends2, dashes=False)
axes.set_title("Google trends for two search terms")
axes.set_ylabel("Frequency of the term (monthly values)")
# rotate tick labels
axes.tick_params(axis='x', labelrotation = 45)
pass
```



[47]:
```
trends2long = trends2.reset_index().melt(id_vars=['month'])
trends2long.rename(columns={'month':'date','variable':'term', 'value':
↪'frequency'}, inplace=True)
trends2long['month'] = trends2long['date'].dt.month
trends2long['year'] = trends2long['date'].dt.year
display(trends2long)
```

```
          date  term  frequency  month  year
0   2007-01-01  lyže         28      1  2007
1   2007-02-01  lyže         48      2  2007
2   2007-03-01  lyže          0      3  2007
3   2007-04-01  lyže          5      4  2007
4   2007-05-01  lyže          0      5  2007
..         …     …          …      …     …
379 2022-08-01  vlek          1      8  2022
380 2022-09-01  vlek          0      9  2022
381 2022-10-01  vlek          1     10  2022
382 2022-11-01  vlek          1     11  2022
383 2022-12-01  vlek          8     12  2022

[384 rows x 5 columns]
```

[48]:
```
axes = sns.lineplot(trends2long, x='month', y='frequency', hue='term')
axes.set_title("Google trends for two search terms summarized over 2007-2022")
axes.set_ylabel("Frequency of the term, monthly average")
pass
```



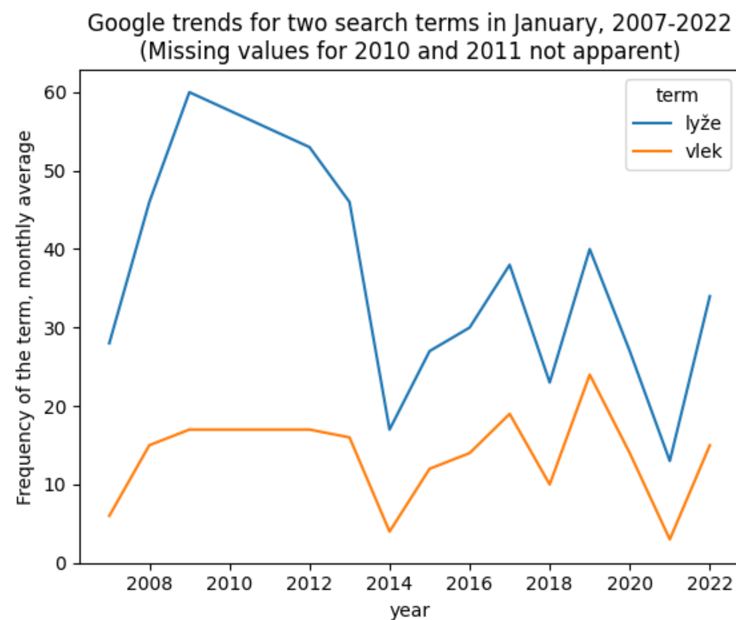
[49]:
```
lyzeJan = trends2long.query("month==1")
axes = sns.lineplot(lyzeJan, x='year', y='frequency', hue='term')
axes.set_ylim(ymin=0)
axes.set_title("Google trends for two search terms in January, 2007-2022")
axes.set_ylabel("Frequency of the term, monthly average")

pass
```

The drop in 2021 was due to pandemics, but what about 2014 and 2018?

1.4.8 Acknowledging missing values

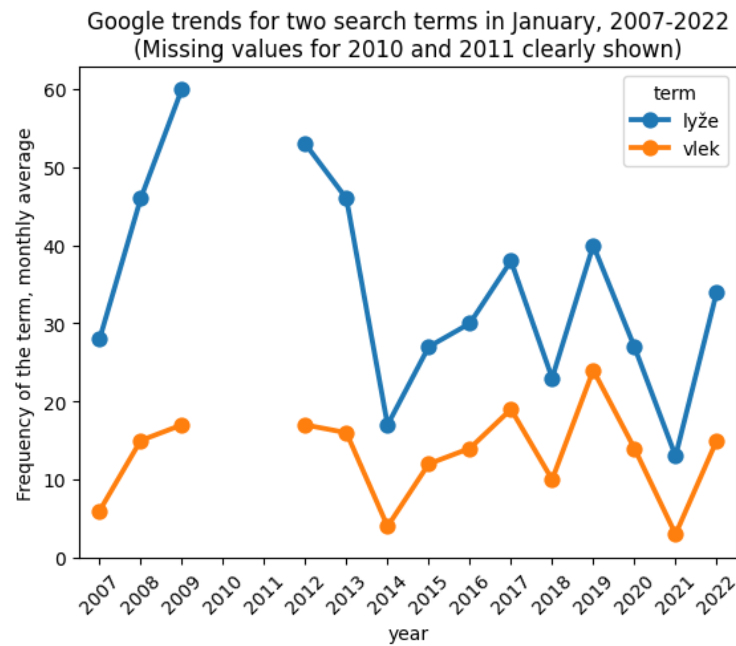
- Let us imagine that January values for 2010 and 2011 are missing.
- If we draw a lineplot in Seaborn, years 2009 and 2012 are connected by a straight line and viewer does not know that something is missing.
- This is not a good idea.
- Below we use pointplot which nicely shows the missing data and also locations of measured values.

[50]:
```
# remove values for two years
to_remove = lyzeJan["year"].isin([2010,2011])
lyzeJanMissing = lyzeJan.copy(deep=True)
lyzeJanMissing.loc[to_remove, 'frequency'] = np.nan
axes = sns.lineplot(lyzeJanMissing, x='year', y='frequency', hue='term')
axes.set_ylim(ymin=0)
axes.set_title("Google trends for two search terms in January,␣
 ↪2007-2022\n(Missing values for 2010 and 2011 not apparent)")
axes.set_ylabel("Frequency of the term, monthly average")
pass
```



[51]:
```
axes = sns.pointplot(lyzeJanMissing, x='year', y='frequency', hue='term')
axes.set_ylim(ymin=0)
axes.set_title("Google trends for two search terms in January,␣
 ↪2007-2022\n(Missing values for 2010 and 2011 clearly shown)")
axes.set_ylabel("Frequency of the term, monthly average")
```

```
axes.tick_params(axis='x', labelrotation = 45)
pass
```



1.4.9 Summary of time series

Typical goals are to observe and study:

- overall trend (increasing / decreasing / flat; rate of change),
- seasonality (daily / weekly / yearly cycles),
- noise (general variability / outliers)

Useful techniques:

- smoothing by aggregation and sliding window
- overlapping timescales
- relative scales
- showing uncertainty and missing values