assignment1

April 9, 2024

1 Information Visualization II

1.1 School of Information, University of Michigan

1.2 Week 1:

- Multivariate/Multidimensional + Temporal

1.3 Assignment Overview

1.3.1 This assignment's objectives include:

- Review, reflect on, and apply different strategies for multidimensional/multivariate/temporal datasets
- Recreate visualizations and propose new and alternative visualizations using Altair

1.3.2 The total score of this assignment will be 100 points consisting of:

- You will be producing four visualizations. Three of them will require you to follow the example closely, but the last will be fairly open-ended. For the last one, we'll also ask you to justify why you designed your visualization the way you did.

1.3.3 Resources:

- Article by FiveThirtyEight available online (Hickey, 2014)
- The associated dataset on Github
- A dataset of all the paintings from the show

1.3.4 Important notes:

1) Grading for this assignment is entirely done by manual inspection. For some of the visualizations, we'll expect you to get pretty close to our example (1-3). Problem 4 is more free-form.

2) Keep your notebooks clean and readable.

3) There are a few instances where our numbers do not align exactly with those from 538. We've pre-processed our data a little bit differently (had different exclusion criteria on guests and for some images we could not process the color data so we excluded those rows).

4) When turning in your PDF, please use the File -> Print -> Save as PDF option ***from your browser***. Do ***not*** use the File->Download as->PDF option. Complete instructions for this

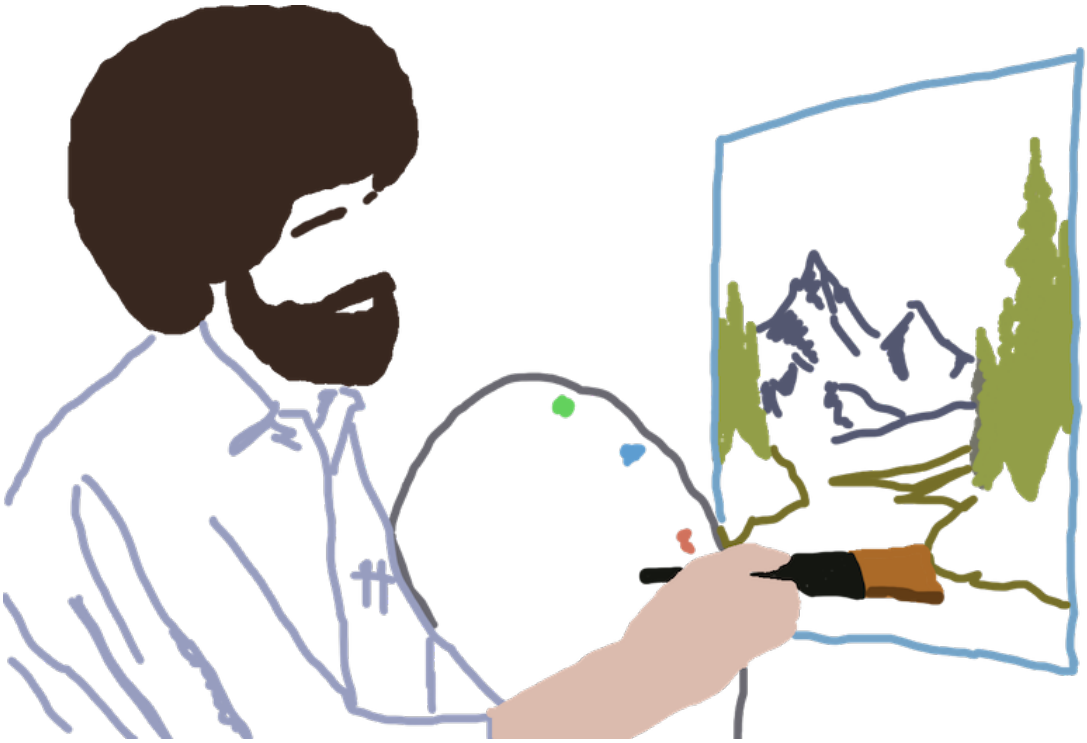
are under Resources in the Coursera page for this class. If you're having trouble with printing, take a look at this video.

[19]:
```
# load up the resources we need
import urllib.request
import os.path
from os import path
import pandas as pd
import altair as alt
import numpy as np
from sklearn import manifold
from sklearn.metrics import euclidean_distances
from sklearn.decomposition import PCA
import ipywidgets as widgets
from IPython.display import display
from PIL import Image
```

1.4 Bob Ross

Today's assignment will have you working with artwork created by Bob Ross. Bob was a very famous painter who had a televised painting show from 1983 to 1994. Over 13 seasons and approximately 400 paintings, Bob would walk the audience through a painting project. Often these were landscape images. Bob was famous for telling his audience to paint "happy trees" and sayings like, "We don't make mistakes, just happy little accidents." His soothing voice and bushy hair are well known to many generations of viewers.

If you've never seen an episode, I might suggest starting with this one.



Bob Ross left a long legacy of art which makes for an interesting dataset to analyze. It's both temporally rich and has a lot of variables we can code. We'll be starting with the dataset created by 538 for their article on a Statistical Analysis of Bob Ross. The authors of the article coded each painting to indicate what features the image contained (e.g., one tree, more than one tree, what kinds of clouds, etc.).

In addition, we've downloaded a second dataset that contains the actual images. We know what kind of paint colors Bob used in each episode, and we have used that to create a dataset for you containing the color distributions. For example, we approximate how much 'burnt umber' he used by measuring the distance (in color space) from each pixel in the image to the color. We then add the 'similarity' of each pixel to the burnt umber RGB value into the respective column. This is imperfect, of course (paints don't mix this way), but it'll be close enough for our analysis. Note that the sum of those rows will not add to 1 and the total value for any column can be more than 1. The only thing we can guarantee is that the metric is consistent across colors and between paintings.

[20]:
```
# the paints Bob used
rosspaints = ['alizarin crimson','bright red','burnt umber','cadmium␣
 ↪yellow','dark sienna',
              'indian yellow','indian red','liquid black','liquid clear','black␣
 ↪gesso',
              'midnight black','phthalo blue','phthalo green','prussian␣
 ↪blue','sap green',
              'titanium white','van dyke brown','yellow ochre']

# hex values for the paints above
rosspainthex =␣
 ↪['#94261f','#c06341','#614f4b','#f8ed57','#5c2f08','#e6ba25','#cd5c5c',
               ␣
 ↪'#000000','#ffffff','#000000','#36373c','#2a64ad','#215c2c','#325fa3',
                '#364e00','#f9f7eb','#2d1a0c','#b28426']

# boolean features about what an image includes
imgfeatures = ['Apple frame', 'Aurora borealis', 'Barn', 'Beach', 'Boat',
               'Bridge', 'Building', 'Bushes', 'Cabin', 'Cactus',
               'Circle frame', 'Cirrus clouds', 'Cliff', 'Clouds',
               'Coniferous tree', 'Cumulus clouds', 'Decidious tree',
               'Diane andre', 'Dock', 'Double oval frame', 'Farm',
               'Fence', 'Fire', 'Florida frame', 'Flowers', 'Fog',
               'Framed', 'Grass', 'Guest', 'Half circle frame',
               'Half oval frame', 'Hills', 'Lake', 'Lakes', 'Lighthouse',
               'Mill', 'Moon', 'At least one mountain', 'At least two␣
 ↪mountains',
               'Nighttime', 'Ocean', 'Oval frame', 'Palm trees', 'Path',
               'Person', 'Portrait', 'Rectangle 3d frame', 'Rectangular frame',
               'River or stream', 'Rocks', 'Seashell frame', 'Snow',
               'Snow-covered mountain', 'Split frame', 'Steve ross',
```

```
                'Man-made structure', 'Sun', 'Tomb frame', 'At least one tree',
                'At least two trees', 'Triple frame', 'Waterfall', 'Waves',
                'Windmill', 'Window frame', 'Winter setting', 'Wood framed']

# load the data frame
bobross = pd.read_csv("assets/bobross.csv")

# enable correct rendering (unnecessary in later versions of Altair)
alt.renderers.enable('default')

# uses intermediate json files to speed things up
alt.data_transformers.enable('json')
```

[20]:
```
DataTransformerRegistry.enable('json')
```

We have a few variables defined for you that you might find useful for the rest of this exercise. First is the `bobross` dataframe which, has a row for every painting created by Bob (we've removed those created by guest artists).

In the dataframe you will see an episode identifier (EPISODE, which contains the season and episode number), the image title (TITLE), the release date (RELEASE_DATE as well as another column for the year). There are also a number of boolean columns for the features coded by 538. A '1' means the feature is present, a '0' means it is not. A list of those columns is available in the `imgfeatures` variable.

[21]:
```
# run to see what's inside
print(imgfeatures)
```

```
['Apple frame', 'Aurora borealis', 'Barn', 'Beach', 'Boat', 'Bridge',
'Building', 'Bushes', 'Cabin', 'Cactus', 'Circle frame', 'Cirrus clouds',
'Cliff', 'Clouds', 'Coniferous tree', 'Cumulus clouds', 'Decidious tree', 'Diane
andre', 'Dock', 'Double oval frame', 'Farm', 'Fence', 'Fire', 'Florida frame',
'Flowers', 'Fog', 'Framed', 'Grass', 'Guest', 'Half circle frame', 'Half oval
frame', 'Hills', 'Lake', 'Lakes', 'Lighthouse', 'Mill', 'Moon', 'At least one
mountain', 'At least two mountains', 'Nighttime', 'Ocean', 'Oval frame', 'Palm
trees', 'Path', 'Person', 'Portrait', 'Rectangle 3d frame', 'Rectangular frame',
'River or stream', 'Rocks', 'Seashell frame', 'Snow', 'Snow-covered mountain',
'Split frame', 'Steve ross', 'Man-made structure', 'Sun', 'Tomb frame', 'At
least one tree', 'At least two trees', 'Triple frame', 'Waterfall', 'Waves',
'Windmill', 'Window frame', 'Winter setting', 'Wood framed']
```

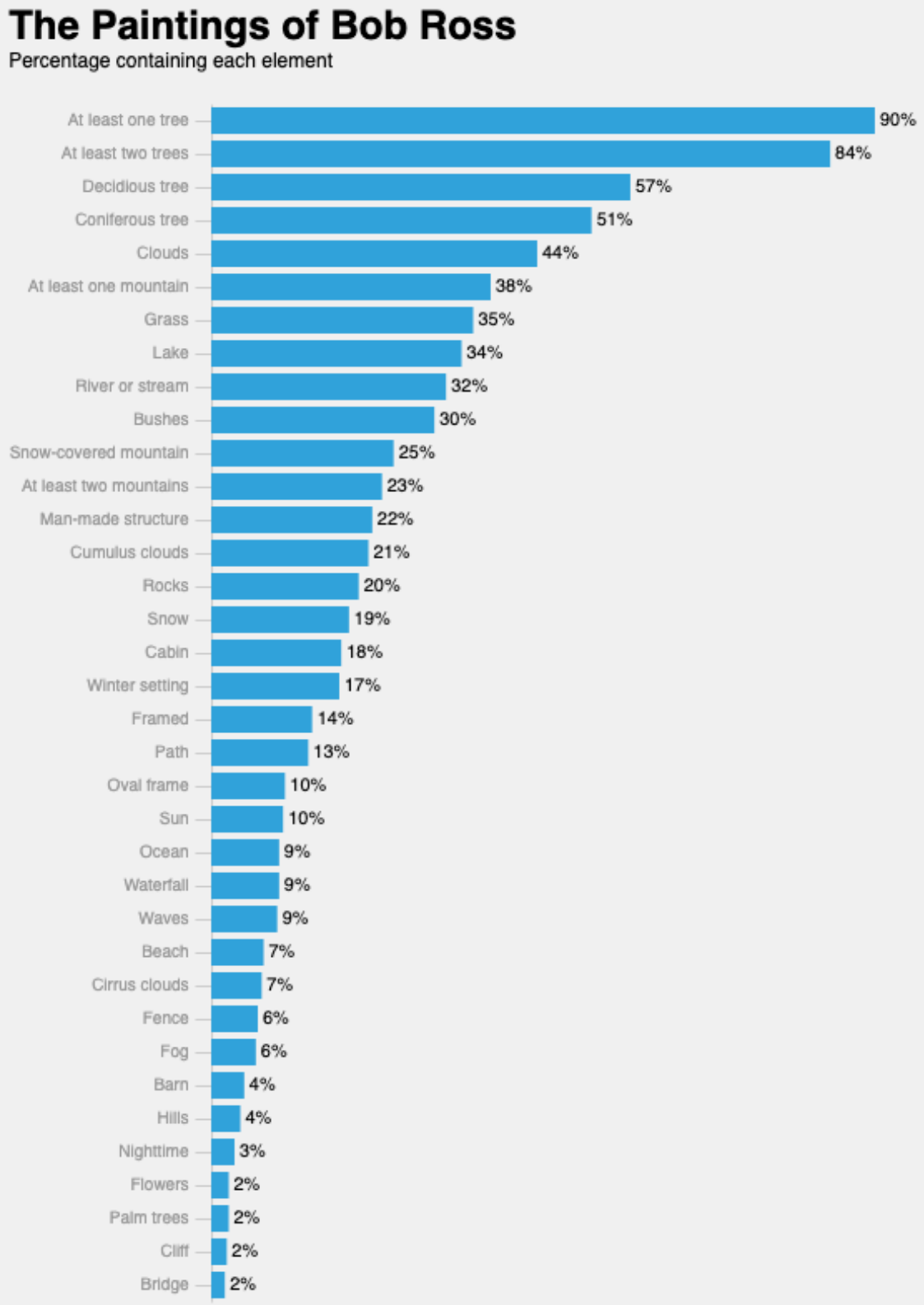
The columns that contain the amount of each color in the paintings are listed in `rosspaints`. There is also an analogous list variable called `rosspainthex` that has the hex values for the paints. These hex values are approximate.

[22]:
```
# run to see what's inside
print("paint names",rosspaints)
print("")
print("hex values", rosspainthex)
```

```
paint names ['alizarin crimson', 'bright red', 'burnt umber', 'cadmium yellow',
'dark sienna', 'indian yellow', 'indian red', 'liquid black', 'liquid clear',
'black gesso', 'midnight black', 'phthalo blue', 'phthalo green', 'prussian
blue', 'sap green', 'titanium white', 'van dyke brown', 'yellow ochre']

hex values ['#94261f', '#c06341', '#614f4b', '#f8ed57', '#5c2f08', '#e6ba25',
'#cd5c5c', '#000000', '#ffffff', '#000000', '#36373c', '#2a64ad', '#215c2c',
'#325fa3', '#364e00', '#f9f7eb', '#2d1a0c', '#b28426']
```

1.4.1 Problem 1 (20 points)

As a warmup, we're going to have you recreate the first chart from the Bob Ross article (source: Statistical Analysis of Bob Ross). This one simply shows a bar chart for the percent of images that have certain features. The Altair version is:



We'll be using the 538 theme for styling, so you don't have to do much beyond creating the chart (but do note that we want to see the percents, titles, and modifications to the axes).

You will replace the code for `makeBobRossBar()` and have it return an Altair chart. We suggest you first create a table that contains the names of the features and the percents. Something like this:

	index	value
0	Barn	0.044619
1	Beach	0.070866
2	Bridge	0.018373
3	Bushes	0.301837
4	Cabin	0.175853
5	Cirrus clouds	0.068241
6	Cliff	0.020997
7	Clouds	0.440945

Recall that this is the 'long form' representation of the data, which will make it easier to create a visualization with. Also, **note the order of the bars. It's not arbitrary, please re-create it.**

[23]:
```
import pandas as pd
import altair as alt

def makeBobRossBar(br, ifeatures):

    df = bobross[imgfeatures]

    percentages = (df.sum() / len(df))
    percentage_df = pd.DataFrame({'Percentage': percentages}).
 ↪sort_values(by='Percentage', ascending=False)[0:36]

    alt.themes.enable('fivethirtyeight')

    # Reset the index of the sorted DataFrame to make it accessible for Altair
    percentage_df.reset_index(inplace=True)

    # Format percentages
    percentage_df['Formatted Percentage'] = percentage_df['Percentage'].
 ↪map(lambda x: '{:.0%}'.format(x))

    # Create the bar chart
    chart = alt.Chart(percentage_df).mark_bar().encode(
```

```
        x=alt.X('Percentage:Q', axis=alt.Axis(format='%')),  # Format the␣
↪x-axis to show percentages
        y=alt.Y('index:O', axis=alt.Axis(title=None), sort='-x')).
↪properties(title='The Paintings of Bob Ross')

    # Display the chart with percentage values next to bars
    text = chart.mark_text(
        align='left',
        baseline='middle',
        dx=3  # Nudges text to right so it doesn't appear on top of the bar
    ).encode(text='Formatted Percentage:N')  # N for nominal since the values␣
↪are strings

    # Combine chart and text
    final_chart = chart + text

    # Show the chart
    return final_chart
```

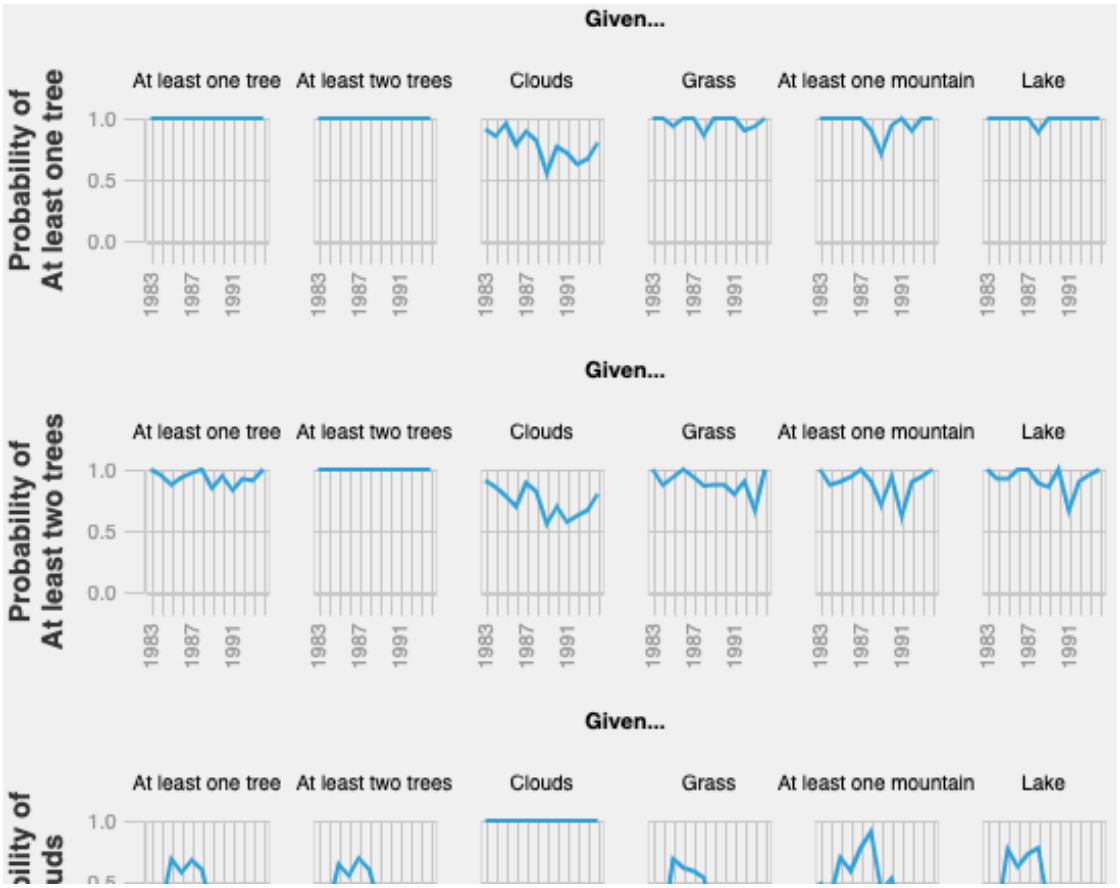
[24]:
```
# run this code to validate
alt.themes.enable('fivethirtyeight')
makeBobRossBar(bobross, imgfeatures)
```

[24]:
```
alt.LayerChart(…)
```

1.5 Problem 2 (25 points)

The 538 article (Statistical Analysis of Bob Ross) has a long analysis of conditional probabilities. Essentially, we want to know the probability of one feature given another (e.g., what is the probability of Snow given Trees?). The article calculates this over the entire history of the show, but we would like to visualize these probabilities over time. Have they been constant? or evolving? We will only be doing this for a few variables (otherwise, we'll have a matrix of over 3000 small charts). The example below is for: 'At least one tree','At least two trees','Clouds','Grass','At least one mountain','Lake.' Each small multiple plot will be a line chart corresponding to the conditional probability over time. The matrix "cell" indicates which pairs of variables are being considered (e.g., probability of at least two trees given the probability of at least one tree is the 2nd row, first column in our example).

Your task will be to generate small multiples plots. For example:



The full image is available here. While your small multiples visualization should contain all this data (the pairwise comparisons), you can ***feel free to style it as you think is appropriate***. We will be grading (minimally) on aesthetics. Implement the code for the function: `makeBobRossCondProb(...)` to return this chart.

Some notes on doing this exercise:

- Write test code for makeBobRossCondProb(…) to make sure it works with different inputs.
- If you don't remember how to calculate conditional probabilities, take a look at the article. Remember, we want the conditional probabilities given the images in a specific year. This is simply an implementation of Conditional Probability/Bayes' Theorem. We implemented a function called `condprobability(...)` as you can see below. You can do the same or pick your own strategy for this.
- We suggest creating a long-form representation of the table for this data. For example, here's a sample of ours (you can use this to double check your calculations):

	key1	key2	year	prob
392	Lake	Clouds	1991	0.142857
60	At least one tree	Lake	1983	1.000000
417	Lake	At least one mountain	1992	0.500000
264	Grass	At least one mountain	1983	0.214286
318	At least one mountain	Clouds	1989	0.333333
85	At least two trees	At least two trees	1984	1.000000
69	At least one tree	Lake	1992	1.000000
387	Lake	Clouds	1986	0.217391
278	Grass	Lake	1985	0.384615
68	At least one tree	Lake	1991	1.000000

- There are a number of strategies to build the small-multiple plots. Some are easier than others. You will find in this case that some combinations of repeated charts and faceting will not work. However, you should be able to use the standard concatenation approaches in combination with repeated charts or faceting.

[33]:
```
def condprobability(frame,column1,column2,year):
    # we suggest you implement this function to make your life easier.
    # input: frame -- the input dataframe in the style of the bobross dataframe␣
 ↪above
    # input: column1 -- the first column to test (e.g, the A in probability of␣
 ↪A given B)
    # input: column2 -- the second column to test (e.g., the B in the␣
 ↪probability of A given B)
    # input: year -- the year for which to calculate the probability
    # return: a conditional probability value

    # you can make variants of this function as you see fit, we will not be␣
 ↪calling it directly

    filtered_df = frame[frame['year'] == year]

    # Count total paintings with feature1
    total_feature2 = filtered_df[column2].sum()
```

```
    # Count paintings with both feature1 and feature2
    both = filtered_df[(filtered_df[column1] == 1) & (filtered_df[column2] ==␣
 ↪1)][column1].sum()

    # Calculate conditional probability
    if total_feature2 > 0:
        probability = both / total_feature2
    else:
        probability = 0

    return probability.round(6)
condprobability(bobross, 'Lake', 'Clouds', 1991)
```

[33]: 0.142857

```
from itertools import combinations
import altair as alt

totest = ['At least one tree','At least two trees','Clouds','Grass','At least␣
 ↪one mountain','Lake']

frame = pd.DataFrame(columns = ['key1','key2','year','prob'])

for year in bobross['year'].unique():
    for var1 in totest:
        for var2 in totest:
            # Calculate conditional probability
            prob = condprobability(bobross, var1, var2, year)

            # Append a row to the DataFrame
            frame = frame.append({'key1': var1, 'key2': var2, 'year': year,␣
 ↪'prob': prob}, ignore_index=True)

frame[frame['key1'] == 'At least one tree']
```

[48]:

	key1	key2	year	prob
0	At least one tree	At least one tree	1983	1.0
1	At least one tree	At least two trees	1983	1.0
2	At least one tree	Clouds	1983	0.909091
3	At least one tree	Grass	1983	1.0
4	At least one tree	At least one mountain	1983	1.0
..	…	…	…	…
397	At least one tree	At least two trees	1994	1.0
398	At least one tree	Clouds	1994	0.8
399	At least one tree	Grass	1994	1.0
400	At least one tree	At least one mountain	1994	1.0
401	At least one tree	Lake	1994	1.0

```
[72 rows x 4 columns]
```

[83]:
```
import altair as alt

# Sample data (assuming you have already created 'frame' DataFrame)
# frame = pd.DataFrame(...)

first_variable_data = frame[frame['key1'] == 'At least one tree']

# Define the order of variables for facetting
variables_order = ['At least one tree','At least two trees', 'Clouds', 'Grass',␣
 ↪'At least one mountain', 'Lake']

# Create Altair chart
chart1 = alt.Chart(first_variable_data).mark_line().encode(
    x=alt.X('year:O', axis=alt.Axis(title='Year')),
    y=alt.Y('prob:Q', axis=alt.Axis(title='Probability of At Least One Tree')),
).properties(
    width=200,
    height=200
).facet('key2:N', title='Given...')



chart1
```

[83]:
```
alt.FacetChart(…)
```

[75]:
```
def makeBobRossCondProb(br, totest):
    # implement this function to return an altair chart
    #
    # input: br the dataframe (e.g., the bobross frame as defined above)
    # input: totest is a variable that holds an array of properties we want␣
 ↪compared (see example below)

    # we have created a default 'totest' variable that has the columns for the␣
 ↪example above

    # return alt.Chart(...)
    frame = pd.DataFrame(columns = ['key1','key2','year','prob'])

    for year in br['year'].unique():
        for var1 in totest:
            for var2 in totest:
                # Calculate conditional probability
                prob = condprobability(bobross, var1, var2, year)
```

```
                # Append a row to the DataFrame
                frame = frame.append({'key1': var1, 'key2': var2, 'year': year,␣
↪'prob': prob}, ignore_index=True)

    variables_order = totest

    charts = []

    #iterate through each key
    for key in frame['key1'].unique():
        #filter data for the current key
        key_data = frame[frame['key1'] == key]

        #create a chart for each key
        chart = alt.Chart(key_data).mark_line().encode(
            x=alt.X('year:O', axis=alt.Axis(title='Year')),
            y=alt.Y('prob:Q', axis=alt.Axis(title=f'Probability of {key}')),
        ).properties(
            width=200,
            height=200
        ).facet(column=alt.Column('key2:N', title='Given...',␣
↪sort=variables_order))

        #append the chart to the list of charts
        charts.append(chart)

    return alt.vconcat(*charts)
```

[76]:
```
# If you did everything right, the following should produce the small multiples␣
 ↪grid for the example in
# the description.
makeBobRossCondProb(bobross, ['At least one tree','At least two␣
 ↪trees','Clouds','Grass','At least one mountain','Lake'])
```

[76]:
```
alt.VConcatChart(…)
```

1.5.1 Additional comments

If you deviated from our example, please use this cell to give us additional information about your design choices and why you think they are an improvement.

1.6 Problem 3 (25 points)

Recall that in some cases of multidimensional data a good strategy is to use dimensionality reduction to visualize the information. Here, we would like to understand how images are similar to each other in 'feature' space. Specifically, how similar are they based on the image features? Are images

that have beaches close to those with waves?

We are going to create a 2D MDS plot using the scikit learn package. We're going to do most of this for you in the next cell. Essentially we will use the euclidean distance between two images based on their image feature array to create the image. Your plot may look slightly different than ours based on the random seed (e.g., rotated or reflected), but in the end, it should be close. If you're interested in how this is calculated, we suggest taking a look at this documentation

Note that the next cell may take a minute or so to run, depending on the server.

[85]:
```
def augmentWithMDS(br=bobross, ifeatures=imgfeatures):
    # input: br -- the bobross shaped dataframe
    # input: ifeatures -- the features we want to use for calculate the MDS␣
 ↪layout
    # output: a modified bobross dataframe that has new columns for the x/y␣
 ↪coordinates

    # create the seed
    seed = np.random.RandomState(seed=3)

    # generate the MDS configuration, we want 2 components, etc. You can tweak␣
 ↪this if you want to see how
    # the settings change the layout
    mds = manifold.MDS(n_components=2, max_iter=3000, eps=1e-9,␣
 ↪random_state=seed, n_jobs=1)

    # fit the data. At the end, 'pos' will hold the x,y coordinates
    pos = mds.fit(br[ifeatures]).embedding_

    # we'll now load those values into the bobross data frame, giving us a new␣
 ↪x column and y column
    br['x'] = [x[0] for x in pos]
    br['y'] = [x[1] for x in pos]
    return(br)

bobross = augmentWithMDS()
bobross
```

[85]:

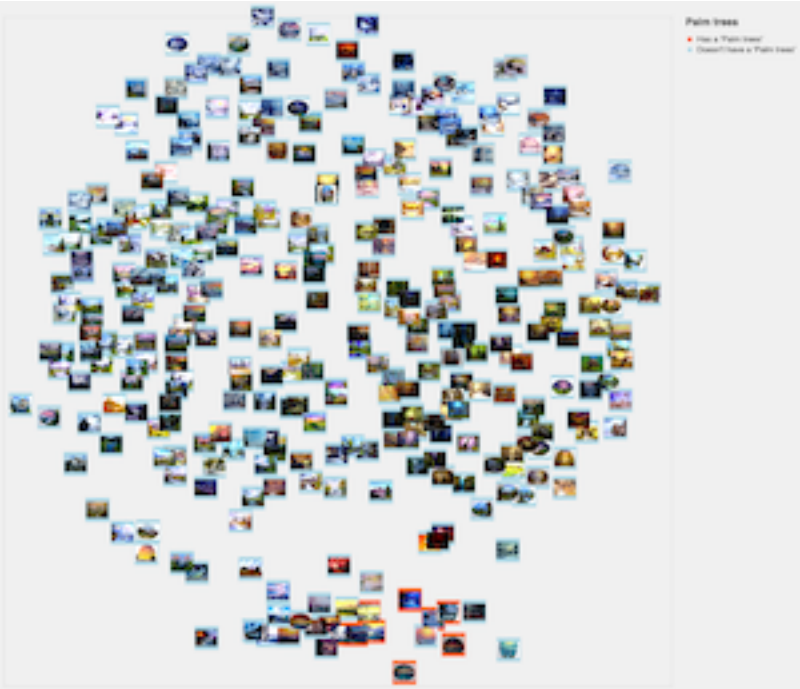
	EPISODE	TITLE	RELEASE_DATE	Apple frame	\
0	S01E01	"A WALK IN THE WOODS"	1/11/83	0	
1	S01E02	"MT. MCKINLEY"	1/11/83	0	
2	S01E03	"EBONY SUNSET"	1/18/83	0	
3	S01E04	"WINTER MIST"	1/25/83	0	
4	S01E05	"QUIET STREAM"	2/1/83	0	
..	…	…	…	…	
376	S31E08	"TRAIL'S END"	4/12/94	0	
377	S31E09	"EVERGREEN VALLEY"	4/19/94	0	
378	S31E10	"BALMY BEACH"	4/26/94	0	

```
379  S31E12  "IN THE MIDST OF WINTER"      5/10/94            0
380  S31E13           "WILDERNESS DAY"      5/17/94            0

     Aurora borealis  Barn  Beach  Boat  Bridge  Building  …  prussian blue  \
0                  0     0      0     0       0         0  …       0.335426
1                  0     0      0     0       0         0  …       0.333043
2                  0     0      0     0       0         0  …       0.226244
3                  0     0      0     0       0         0  …       0.489283
4                  0     0      0     0       0         0  …       0.364585
..               …   …    …   …     …       …  …            …
376                0     0      0     0       0         0  …       0.000000
377                0     0      0     0       0         0  …       0.467273
378                0     0      1     0       0         0  …       0.000000
379                0     1      0     0       0         0  …       0.578457
380                0     0      0     0       0         0  …       0.000000

     sap green  titanium white  van dyke brown  yellow ochre  \
0     0.291454        0.341764        0.301478      0.000000
1     0.205642        0.571301        0.223015      0.000000
2     0.744489        0.047932        0.718768      0.000000
3     0.000000        0.481240        0.269215      0.000000
4     0.267708        0.341428        0.286462      0.000000
..         …               …               …             …
376   0.499657        0.264233        0.525722      0.433769
377   0.548042        0.151461        0.586429      0.411393
378   0.000000        0.197300        0.591724      0.428434
379   0.000000        0.258982        0.435050      0.000000
380   0.690162        0.087771        0.723869      0.472715

                                                 img_url  week_number  year  \
0    https://raw.githubusercontent.com/jwilber/Bob_…            2  1983
1    https://raw.githubusercontent.com/jwilber/Bob_…            2  1983
2    https://raw.githubusercontent.com/jwilber/Bob_…            3  1983
3    https://raw.githubusercontent.com/jwilber/Bob_…            4  1983
4    https://raw.githubusercontent.com/jwilber/Bob_…            5  1983
..                                                 …          …   …
376  https://raw.githubusercontent.com/jwilber/Bob_…           15  1994
377  https://raw.githubusercontent.com/jwilber/Bob_…           16  1994
378  https://raw.githubusercontent.com/jwilber/Bob_…           17  1994
379  https://raw.githubusercontent.com/jwilber/Bob_…           19  1994
380  https://raw.githubusercontent.com/jwilber/Bob_…           20  1994

           x         y
0   0.468509 -0.655269
1  -0.787913  2.087431
2   0.088857  2.670687
3  -1.435869  0.895231
```

```
4    0.668885 -0.865571
..        …        …
376  1.024702 -0.307968
377 -1.837915  0.107299
378  1.041584 -3.094676
379  1.936991  1.863692
380  0.188533 -0.076906

[381 rows x 116 columns]
```

Your task is to implement the visualization for the MDS layout. We will be using a new mark, `mark_image`, for this. You can read all about this mark on the Altair site here. Note that we all already saved the images for you. They are accessible in the img_url column in the bobross table. You will use the `url` encode argument to mark_image to make this work.

In this case, we would also like to emphasize all the images that *have* a specific feature. So when you define your `genMDSPlot()` function below, it should take a key string as an argument (e.g., 'Beach') and visually highlight those images. A simple way to do this is to use a second mark underneath the image (e.g., a rectangle) that is a different color based on the absence or presence of the image. Here's an example output for `genMDSPlot("Palm trees")`:



Click here for a large version of this image. Notice the orange boxes indicating where the Palm tree images are. Note also that we have styled the MDS plot to not have axes. Recall that these are meaningless in MDS 'space' (this is not a scatterplot, it's a projection).

Important: *You can make some of your own choices on how to make the matched items salient but you need to make this this visualization usable (expressive & effective). We do expect that your solution not be less effective/expressive than our example.*

Hint: you may want to think about how to get "details" if you make images very small. We'd like to be able to figure out which image is what. A really simply strategy is to use something like tooltips.

[125]:
```
br = bobross

key_data = br[br[key] == 1]
key_data
```

[125]:

	EPISODE	TITLE	RELEASE_DATE	Apple frame	Aurora borealis	\
3	S01E04	"WINTER MIST"	1/25/83	0	0	
5	S01E06	"WINTER MOON"	2/8/83	0	0	
6	S01E07	"AUTUMN MOUNTAINS"	2/15/83	0	0	
7	S01E08	"PEACEFUL VALLEY"	2/22/83	0	0	
9	S01E10	"MOUNTAIN LAKE"	3/8/83	0	0	
..	…	…	…	…	…	
364	S30E08	"HOME IN THE VALLEY"	1/11/94	0	0	
366	S30E11	"COLD SPRING DAY"	2/1/94	0	0	
369	S31E01	"REFLECTIONS OF CALM"	2/22/94	0	0	
372	S31E04	"TRANQUILITY COVE"	3/15/94	0	0	
375	S31E07	"BRIDGE TO AUTUMN"	4/5/94	0	0	

	Barn	Beach	Boat	Bridge	Building	…	prussian blue	sap green	\
3	0	0	0	0	0	…	0.489283	0.000000	
5	0	0	0	0	0	…	0.454961	0.000000	
6	0	0	0	0	0	…	0.360541	0.205003	
7	0	0	0	0	0	…	0.419738	0.262598	
9	0	0	0	0	0	…	0.391097	0.216671	
..	…	…	…	…	…	…	…	…	
364	0	0	0	0	0	…	0.524274	0.412713	
366	0	0	0	0	0	…	0.506989	0.457878	
369	0	0	0	0	0	…	0.552920	0.451598	
372	0	0	0	0	0	…	0.000000	0.399030	
375	0	0	0	1	0	…	0.384495	0.481444	

```
     titanium white  van dyke brown  yellow ochre  \
3          0.481240        0.269215      0.000000
5          0.382906        0.351053      0.000000
6          0.555759        0.219132      0.000000
7          0.459936        0.288607      0.000000
9          0.542974        0.237525      0.000000
..              …               …             …
364        0.211663        0.451308      0.324456
366        0.229397        0.503986      0.395546
369        0.190052        0.497700      0.386849
372        0.274458        0.428438      0.510577
375        0.187265        0.503290      0.632580

                                                 img_url  week_number  year  \
3    https://raw.githubusercontent.com/jwilber/Bob_…            4  1983
5    https://raw.githubusercontent.com/jwilber/Bob_…            6  1983
6    https://raw.githubusercontent.com/jwilber/Bob_…            7  1983
7    https://raw.githubusercontent.com/jwilber/Bob_…            8  1983
9    https://raw.githubusercontent.com/jwilber/Bob_…           10  1983
..                                                 …          …   …
364  https://raw.githubusercontent.com/jwilber/Bob_…            2  1994
366  https://raw.githubusercontent.com/jwilber/Bob_…            5  1994
369  https://raw.githubusercontent.com/jwilber/Bob_…            8  1994
372  https://raw.githubusercontent.com/jwilber/Bob_…           11  1994
375  https://raw.githubusercontent.com/jwilber/Bob_…           14  1994

             x         y
3   -1.435869  0.895231
5   -0.672693  2.997108
6   -0.617327  1.026443
7   -0.984850  1.132381
9   -0.346189  0.794717
..        …       …
364  0.139546  1.747951
366 -1.436254  1.237957
369 -0.826806  0.974326
372  0.710793 -0.173504
375  1.918623  0.463837

[129 rows x 116 columns]
```

[177]:
```
import altair as alt

def genMDSPlot(br, key):
    # Filter the dataframe to include only images containing the specified␣
 ↪feature
```

```
    key_data = br[br[key] == 1]
    unkey_data = br[br[key] == 0]

    # Create the MDS plot
    mds_plot = alt.Chart(br).mark_image(
        width=15,
        height=15,
        clip=True
    ).encode(
        x='x:Q',
        y='y:Q',
        url='img_url',
        color=alt.Color('presence:N',
                        scale=alt.Scale(domain=['Has a(n) ' + key, 'Doesnt have␣
↪a(n) ' + key], range=['orange', 'blue']),
                        title=key)
    ).properties(
        width=500,
        height=500
    )

    # Highlight images containing the specified feature
    highlight = alt.Chart(key_data).mark_rect(
        width=15,
        height=15,
        color='orange',
        opacity=0.2
    ).encode(
        x='x:Q',
        y='y:Q'
    )

    highlight2 = alt.Chart(unkey_data).mark_rect(
        width=15,
        height=15,
        color='blue',
        opacity=0.05
    ).encode(
        x='x:Q',
        y='y:Q'
    )

    # Combine the MDS plot with the highlights
    combined_plot = mds_plot + highlight + highlight2

    # Remove axes and grid
    combined_plot = combined_plot.configure_axis(
```

```
        grid=False)

    return combined_plot



# Example usage:
genMDSPlot(bobross, 'Beach')
```

[177]:
```
alt.LayerChart(…)
```

We are going to create an interactive widget that allows you to select the feature you want to be highlighted. If you implemented your `genMDSPlot` code correctly, the plot should change when you select new items from the list. We would ordinarily do this directly in Altair, but because we don't have control over the way you created your visualization, it's easiest for us to use the widgets built into Jupyter.

It should look something like this:

It may take a few seconds the first time you run this to download all the images.

[172]:
```
# note that it might take a few seconds for the images to download
# depending on your internet connection

output = widgets.Output()

def clicked(b):
    output.clear_output()
    with output:
        # when the selection is changed, we pull the value and call the altair␣
 ↪plot generator
        highlight = filterdrop.value
        if (highlight == ""):
            print("please enter a query")
        else:
            genMDSPlot(bobross,highlight).display()
```

```
featurecount = bobross[imgfeatures].sum()

filterdrop = widgets.Dropdown(
    options=list(featurecount[featurecount > 2].keys()),
    description='Highlight:',
    disabled=False,
)

filterdrop.observe(clicked, names=['value'])

display(filterdrop,output)

with output:
    genMDSPlot(bobross,'Barn').display()
```

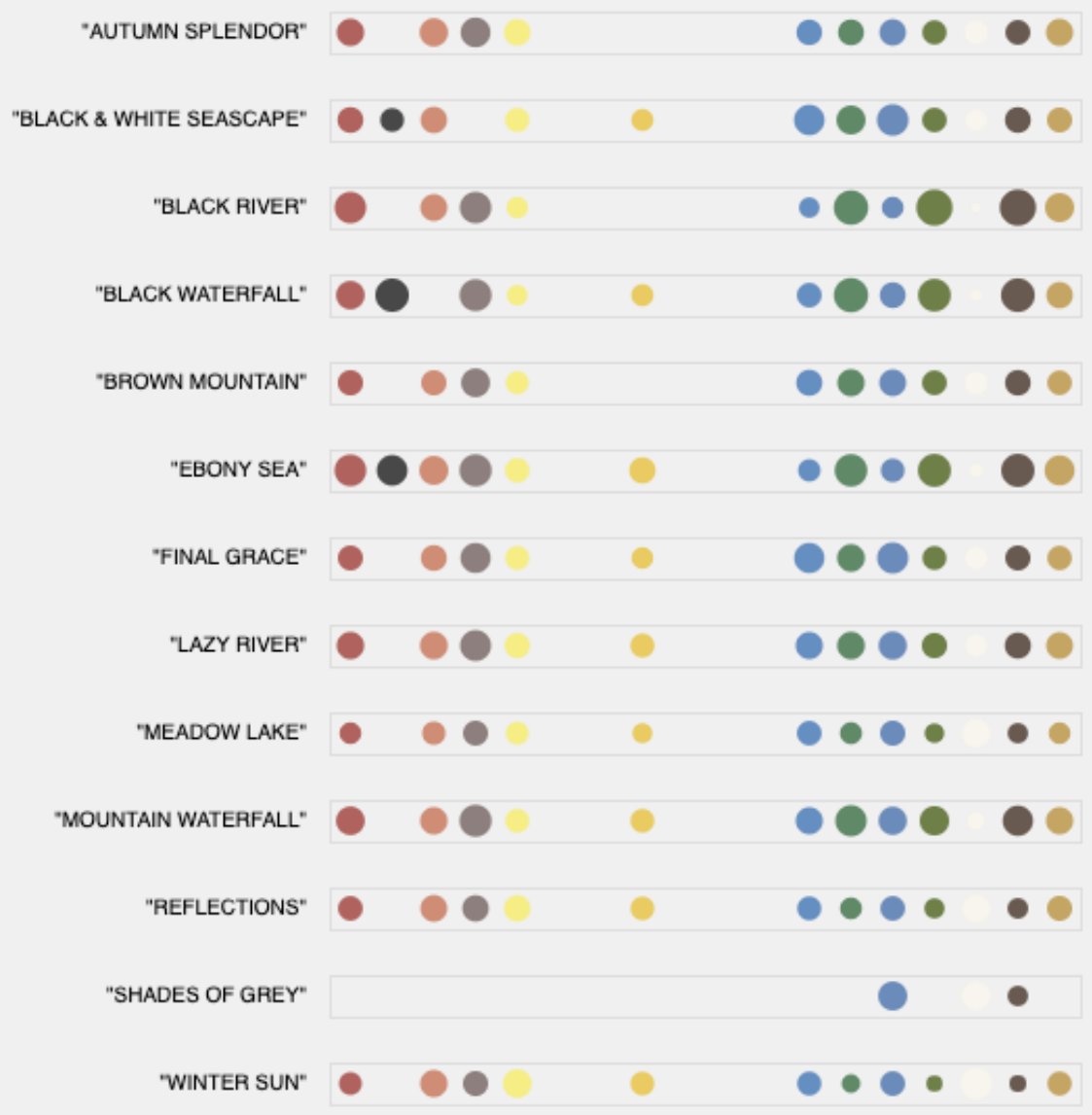
```
Dropdown(description='Highlight:', options=('Barn', 'Beach', 'Bridge', 'Bushes',␣
↪'Cabin', 'Cactus', 'Cirrus cl…
```

```
Output()
```

1.7 Problem 4 (30 points: 25 for solution, 5 for explanation)

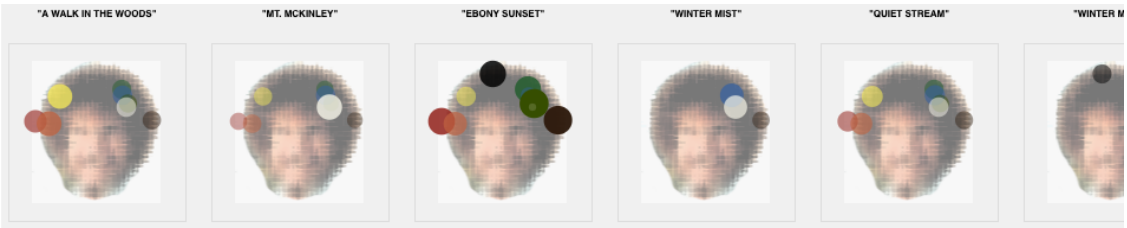
Your last problem is fairly open-ended in terms of visualization. We would like to analyze the colors used in different images for a given season as a small multiples plot. You can pick how you represent your small multiples, but we will ask you to defend your choices below. You must implement the function `colorSmallMultiples(season)` that takes a season number as input (e.g., 2) and returns an Altair chart. The "multiples" should be at the painting level–so, one multiple per painting (and each TV season shown at once).

Here's a really simple example (that isn't great):



This visualization has a row (a "mini plot") for every painting and a colored circle (in the color of the paint). The circle is sized based on the amount of the corresponding paint that is used in the image.

You can also go to something as crazy as this:



Here, we've overlaid circles as curls in Bob's massive hair. We're not claiming this is an effective

solution, but you're welcome to do this (or anything else) as long as you describe the pros and cons of your choices. And, yes, we generated both examples using Altair.

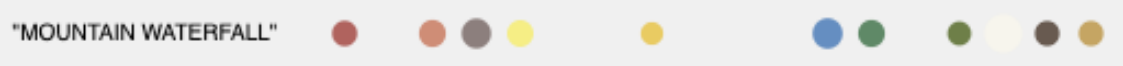
Again, the relevant columns are available are listed in `rosspaints` (there are 18 of them). The values range from 0 to 1 based on the fraction of pixel color allocated to that specific paint. The `rosspainthex` has the corresponding hex values for the paint color.

Some notes

1) We would like for you to be creative. You will not get full credit for this assignment if you simply copy our examples (or change them slightly).

2) Make sure your visualization is actually a small multiple approach. There should be "mini" visualizations for each painting. This is a "rough" check, but if you're not using repeating, faceting, concatenation, etc. you're probably just making one chart (e.g., a heatmap). Another check is if there are axis labels/information on each so that it's readable on its own (a shared legend is fine). All these are inexact tests but may be helpful as a starting point.

3) You *may* find it useful to implement "colorSmallMultiple" as below to generate your single small multiple. This may not be ideal if you're using faceting or repetition. For example, in our implementation calling `colorSmallMultiple(5,1)` will create a small multiple for season 5, episode 1:



4) As with all assignments in this course, test everything!

[220]:
```
import altair as alt
import pandas as pd


paint_names = ['alizarin crimson', 'bright red', 'burnt umber', 'cadmium␣
 ↪yellow', 'dark sienna',
               'indian yellow', 'indian red', 'liquid black', 'liquid clear',␣
 ↪'black gesso',
               'midnight black', 'phthalo blue', 'phthalo green', 'prussian␣
 ↪blue', 'sap green',
               'titanium white', 'van dyke brown', 'yellow ochre']
hex_values = ['#94261f', '#c06341', '#614f4b', '#f8ed57', '#5c2f08', '#e6ba25',␣
 ↪'#cd5c5c', '#000000', '#ffffff',
              '#000000', '#36373c', '#2a64ad', '#215c2c', '#325fa3', '#364e00',␣
 ↪'#f9f7eb', '#2d1a0c', '#b28426']

def colorSmallMultiple(season, episodenumber, br, rp, rph):
    # Filter the data for the specified season and episode
    data = br[(br['season'] == season) & (br['episode'] == episodenumber)]

    # Melt the episode data to make it suitable for plotting
```

```
    melted_data = data.melt(id_vars=['season', 'episode'],␣
↪value_vars=paint_names,
                            var_name='Paint', value_name='Value')

    # Merge with paint names and hex values
    merged_data = melted_data.merge(pd.DataFrame({'Paint': paint_names, 'Hex':␣
↪hex_values}), on='Paint')

    # Create scatter plots for each color
    charts = []
    for i, color in enumerate(hex_values):
        # Filter data for the current color
        color_data = merged_data[merged_data['Hex'] == color]

        # Create scatter plot
        scatter_plot = alt.Chart(color_data).mark_circle().encode(
            x='season:N',
            y='episode:N',
            size=alt.Size('Value:Q', scale=alt.Scale(range=[100, 1000])),  #␣
↪Adjust the range as needed
            color=alt.Color('Paint:N', scale=alt.Scale(domain=paint_names,␣
↪range=hex_values), legend=None),
            tooltip=['Paint', 'Value', 'Hex']
        ).properties(
            width=200,
            height=200,
            title=f'Color: {paint_names[i]}'
        )

        charts.append(scatter_plot)

    # Combine all scatter plots into a single chart
    combined_chart = alt.hconcat(*charts)

    return combined_chart


colorSmallMultiple(12, 10, bobross, rosspaints, rosspainthex)
```

[220]: alt.HConcatChart(…)

```
[232]: import altair as alt
       import pandas as pd

       # Define paint names and hex values
       paint_names = ['alizarin crimson', 'bright red', 'burnt umber', 'cadmium␣
        ↪yellow', 'dark sienna',
```

```
                'indian yellow', 'indian red', 'liquid black', 'liquid clear',␣
 ↪'black gesso',
                'midnight black', 'phthalo blue', 'phthalo green', 'prussian␣
 ↪blue', 'sap green',
                'titanium white', 'van dyke brown', 'yellow ochre']
hex_values = ['#94261f', '#c06341', '#614f4b', '#f8ed57', '#5c2f08', '#e6ba25',␣
 ↪'#cd5c5c', '#000000', '#ffffff',
              '#000000', '#36373c', '#2a64ad', '#215c2c', '#325fa3', '#364e00',␣
 ↪'#f9f7eb', '#2d1a0c', '#b28426']

def colorSmallMultiples(season):
    # Filter the data for the specified season
    data = br[br['season'] == season]

    # Create a list to hold the charts for each episode
    charts = []

    # Iterate through each episode in the season
    for episode in sorted(data['episode'].unique()):
        episode_data = data[data['episode'] == episode]

        # Melt the episode data to make it suitable for plotting
        melted_data = episode_data.melt(id_vars=['season', 'episode'],␣
 ↪value_vars=paint_names,
                                        var_name='Paint', value_name='Value')

        # Merge with paint names and hex values
        merged_data = melted_data.merge(pd.DataFrame({'Paint': paint_names,␣
 ↪'Hex': hex_values}), on='Paint')

        # Create scatter plots for each color
        episode_charts = []
        for i, color in enumerate(hex_values):
            # Filter data for the current color
            color_data = merged_data[merged_data['Hex'] == color]

            # Create scatter plot
            scatter_plot = alt.Chart(color_data).mark_circle().encode(
                x=alt.X('episode:N', title='Episode'),
                y=alt.Y('Value:Q', title='Value'),
                size=alt.Size('Value:Q', scale=alt.Scale(range=[100, 1000]),␣
 ↪legend=None),  # Adjust the range as needed
                color=alt.Color('Paint:N', scale=alt.Scale(domain=paint_names,␣
 ↪range=hex_values), legend=None),
                tooltip=['Paint', 'Value', 'Hex']
            ).properties(
```

```
                width=100,
                height=100,
                title=f'Color: {paint_names[i]}')

            episode_charts.append(scatter_plot)

        # Combine all scatter plots for the episode into a single chart
        episode_combined_chart = alt.hconcat(*episode_charts).
↪resolve_scale(y='independent')

        # Add the chart for the episode to the list of charts
        charts.append(episode_combined_chart)

    # Combine all charts for the episodes into a single row
    row_chart = alt.vconcat(*charts)

    return row_chart


colorSmallMultiples(12)
```

[232]:
```
alt.VConcatChart(…)
```

[233]:
```
# run this to test your code for season 1
colorSmallMultiples(1)
```

[233]:
```
alt.VConcatChart(…)
```

[234]:
```
# run this to test your code for season 2
colorSmallMultiples(2)
```

[234]:
```
alt.VConcatChart(…)
```

1.7.1 Explain your choices

Explain your design here. Describe the pros and cons in terms of visualization principles.

Some advice: Make sure to discuss the pros/cons of your solution in detail. We're looking for explanation of expressiveness and effectiveness. Your design choices will impact what is easy/hard to do (remember, wicked design!) and we want you to self-critique.

originally i thought the idea the idea to make to make the "subpar" example visualization more detailed with color names and size markers for amount of color used a good idea. it looked good when i used a single episode small multiple graph but when a whole season was inputted i feel like the data was overwhelming. The points that were also originally nicely compared by size with one row became asynchornous when multiple rows were added as the the blobs moved up their singular y axis as they got bigger. I think the best approach would have been to made a detailed bar graph of every episode which is what i started to do but figured that was not a true small

multiple approach.

[ ]: