# Information Visualization II

## School of Information, University of Michigan

## Week 1:

- Multivariate/Multidimensional + Temporal

## Assignment Overview

### 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 (https://altair-viz.github.io/)

### 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.

### Resources:

- Article by FiveThirtyEight (https://fivethirtyeight.com) available online (https://fivethirtyeight.com/features/a-statistical-analysis-of-the-work-of-bob-ross/) (Hickey, 2014)
- The associated dataset on Github (https://github.com/fivethirtyeight/data/tree/master/bob-ross)
- A dataset of all the paintings from the show (https://github.com/jwilber/Bob_Ross_Paintings)

### 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).

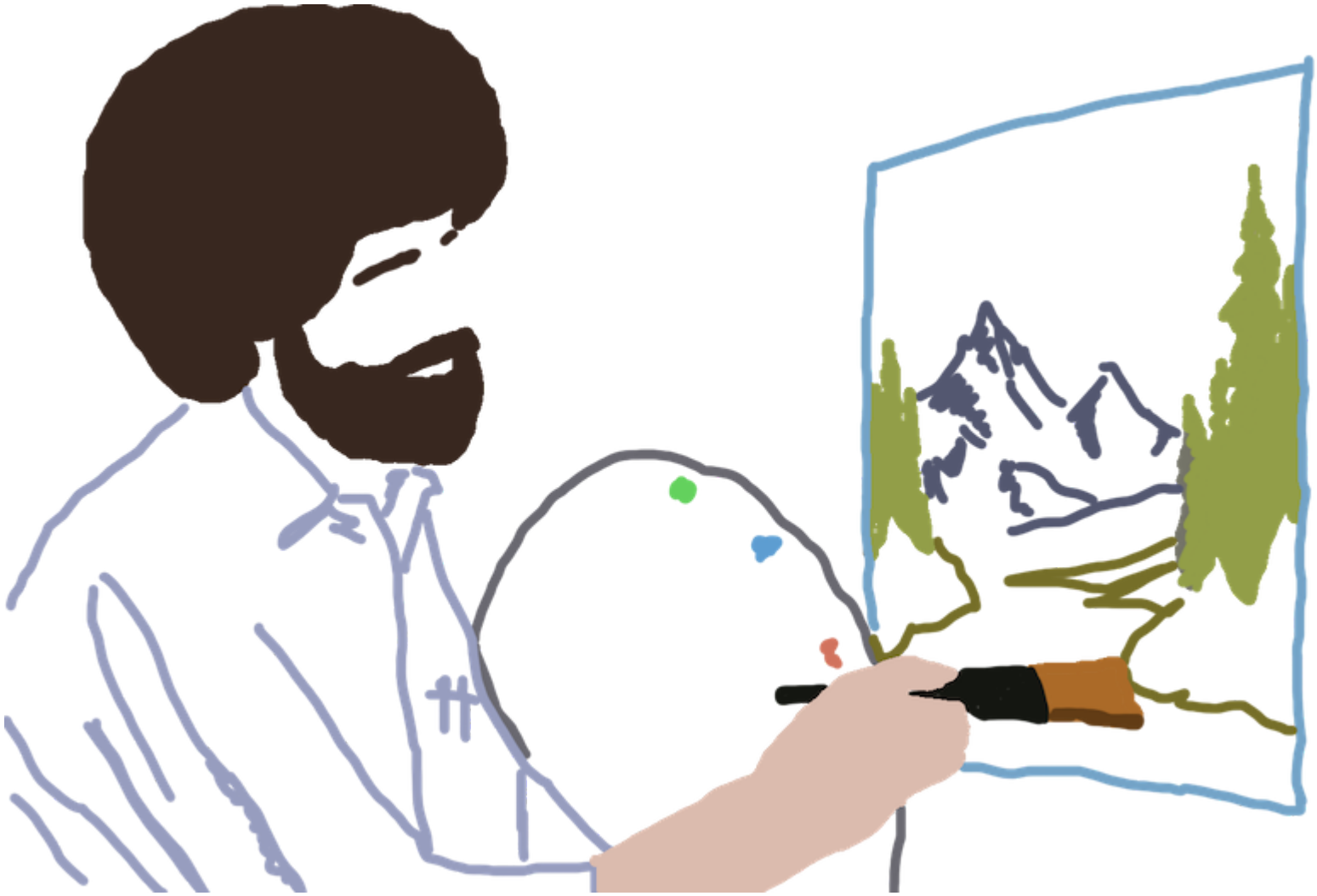
4) When turning in your PDF, please use the File -> Print -> Save as PDF option ***from your browser***. Do ***not*** use the File->Download as->PDF option. Complete instructions for this are under Resources in the Coursera page for this class. If you're having trouble with printing, take a look at this video (https://youtu.be/PiO-K7AoWjk).

In [1]:
```
# 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
```

# Bob Ross

Today's assignment will have you working with artwork created by Bob Ross (https://en.wikipedia.org/wiki/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 (https://www.youtube.com/watch?v=Fw6odlNp7_8).



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 (https://fivethirtyeight.com/features/a-statistical-analysis-of-the-work-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.

In [2]:
```
# 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')
```

Out[2]:
```
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 [3]:
```
# run to see what's inside
bobross.sample(5)
```

Out[3]:

| | EPISODE | TITLE | RELEASE_DATE | Apple frame | Aurora borealis | Barn | Beach | Boat | Bridge | Building | ... | phthalo blue | phthalo green | prussian blue | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **379** | S31E12 | "IN THE MIDST OF WINTER" | 5/10/94 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | ... | 0.533761 | 0.000000 | 0.578457 | ( |
| **27** | S03E03 | "BUBBLING STREAM" | 1/18/84 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0.360357 | 0.417753 | 0.390170 | ( |
| **142** | S12E11 | "SOFT MOUNTAIN GLOW" | 7/8/87 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0.378284 | 0.000000 | 0.415660 | ( |
| **288** | S24E06 | "MIRRORED IMAGES" | 2/11/92 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0.528702 | 0.000000 | 0.563582 | ( |
| **39** | S04E03 | "MAJESTIC MOUNTAINS" | 9/19/84 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0.427229 | 0.552663 | 0.464526 | ( |

5 rows × 114 columns

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.

In [4]:
```
# 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', 'Di
ane 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', 'Moo
n', 'At least one mountain', 'At least two mountains', 'Nighttime', 'Ocean', 'Oval frame', 'Palm trees', 'Pat
h', 'Person', 'Portrait', 'Rectangle 3d frame', 'Rectangular frame', 'River or stream', 'Rocks', 'Seashell fra
me', '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.

In [5]:
```
# run to see what's inside
print("paint names",rosspaints)
print("")
print("hex values", rosspainthex)
```

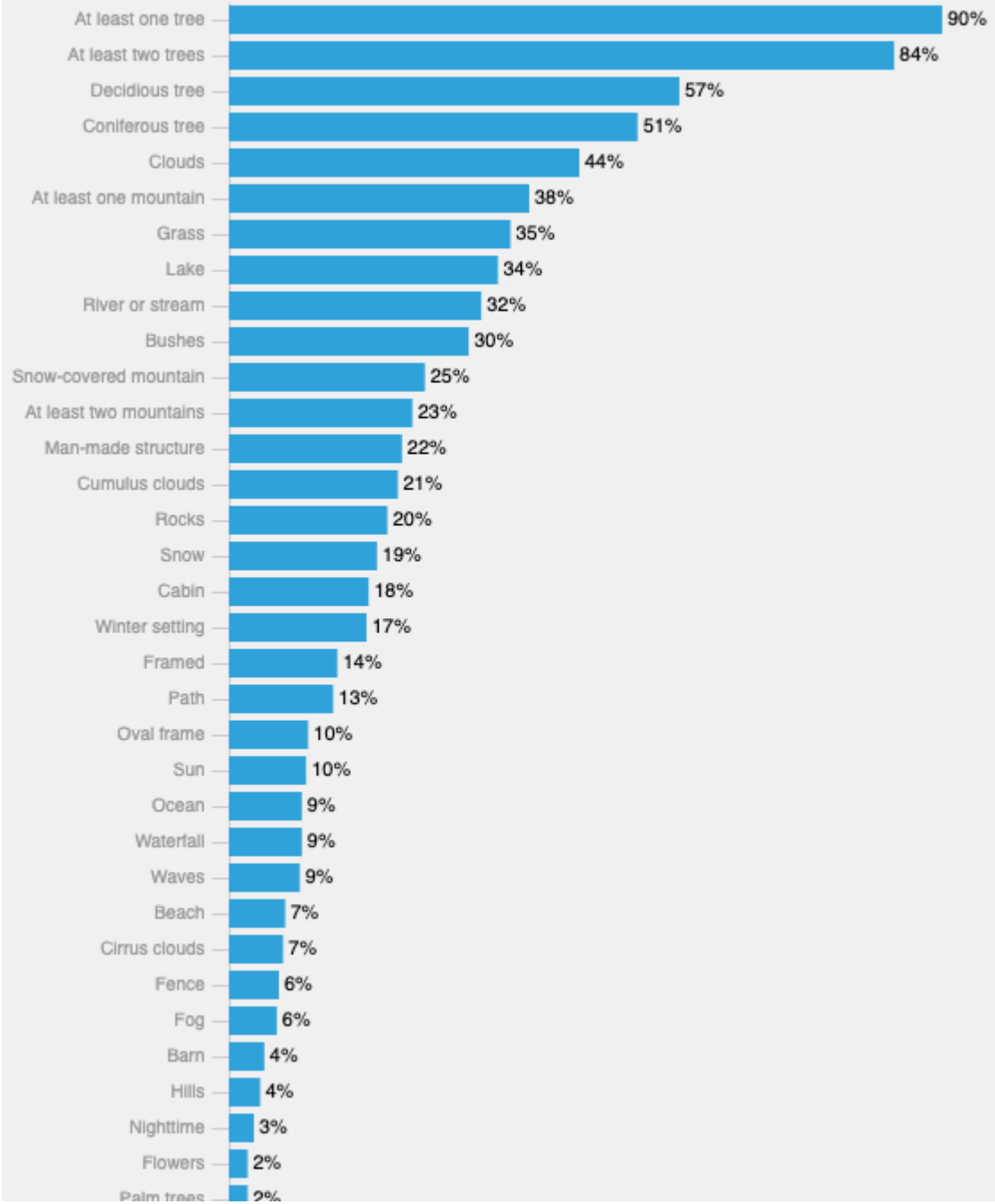
```
paint names ['alizarin crimson', 'bright red', 'burnt umber', 'cadmium yellow', 'dark sienna', 'indian yello
w', 'indian red', 'liquid black', 'liquid clear', 'black gesso', 'midnight black', 'phthalo blue', 'phthalo gr
een', 'prussian blue', 'sap green', 'titanium white', 'van dyke brown', 'yellow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

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

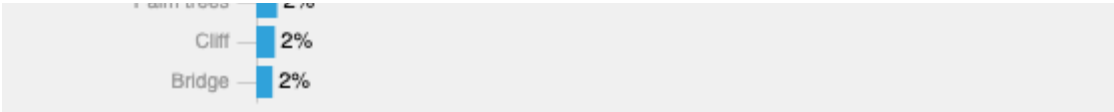
## Problem 1 (20 points)

As a warmup, we're going to have you recreate the first chart from the Bob Ross article (assets/bob_ross_538.png) (source: Statistical Analysis of Bob Ross (https://fivethirtyeight.com/features/a-statistical-analysis-of-the-work-of-bob-ross/)). This one simply shows a bar chart for the percent of images that have certain features. The Altair version is:

# The Paintings of Bob Ross

Percentage containing each element






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.**

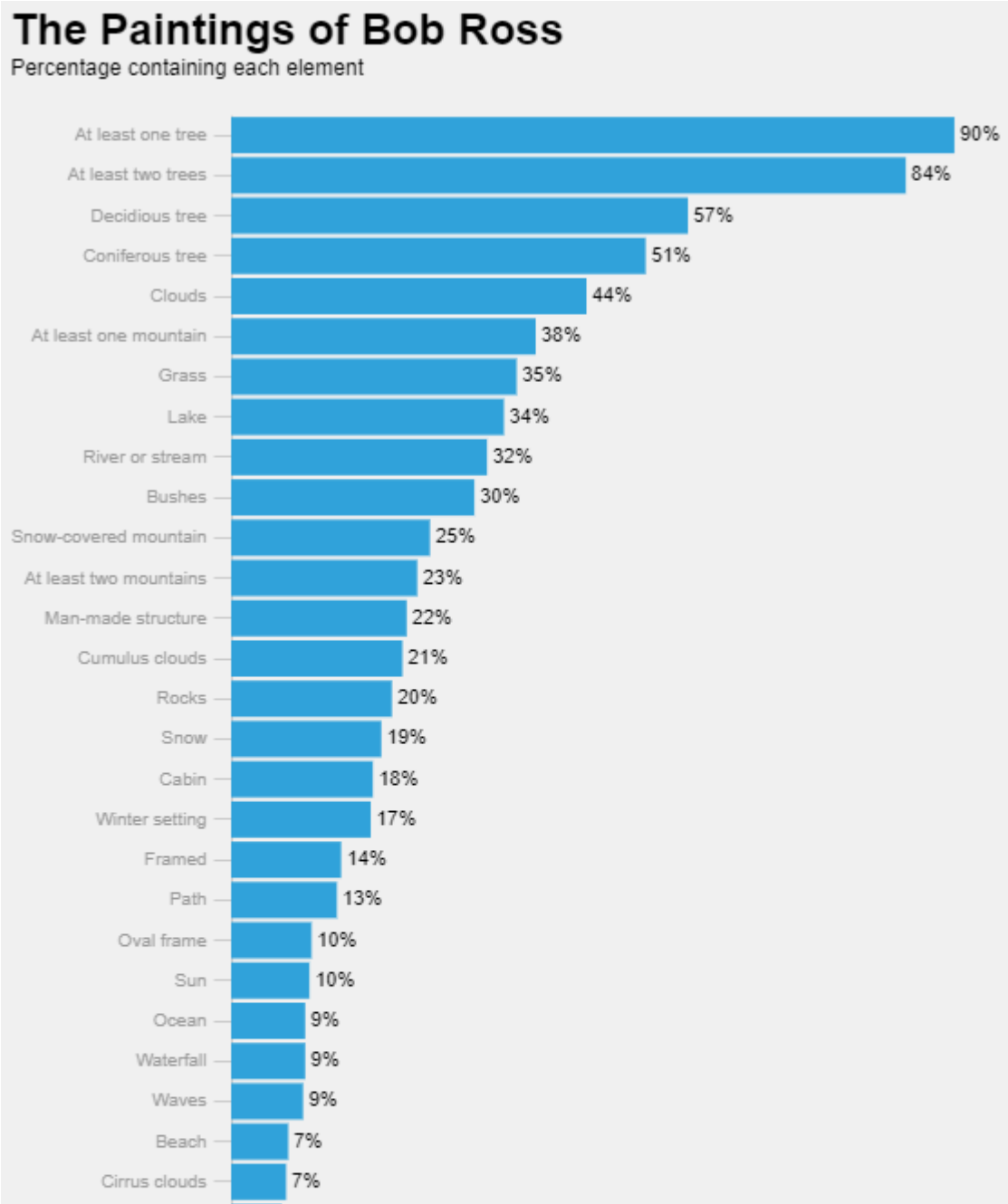
In [6]:

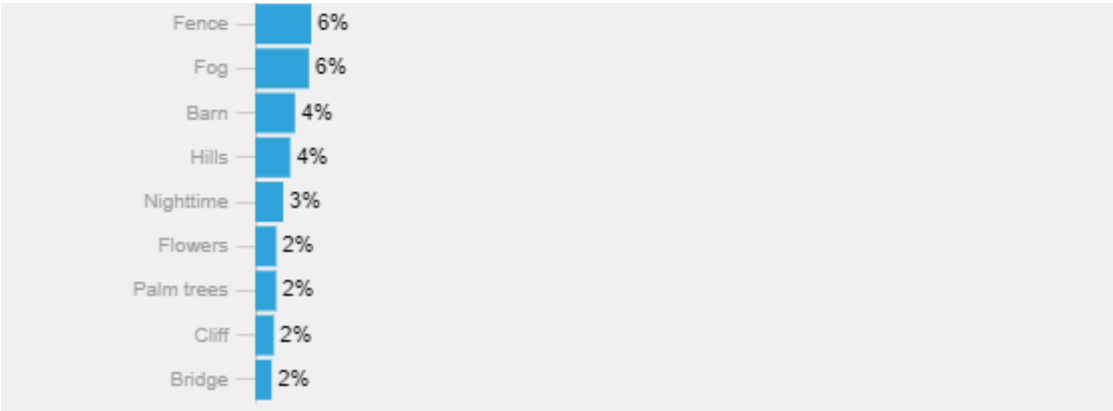
```
def makeBobRossBar(br, ifeatures):
    # input: br -- a dataframe in the shape of the bobross frame defined above
    # input: ifeatures -- a list of the features we want to test (see imgfeatures above)
    # return: implement this function to return an altair chart as defined above
    #         e.g., return alt.Chart(...)

    total_rows = len(br)

    # for each image, find the percentage of times it's found in a painting
    percents = []

    for image in ifeatures:
        percent = len(br[br[image] == 1]) / total_rows
        percents.append(percent)

    tuples = list(zip(ifeatures, percents))
    bar_df = pd.DataFrame(tuples, columns = ['images', 'percentages'])
    bar_df = bar_df[bar_df['percentages'] >= .015]


    bars = alt.Chart(bar_df).mark_bar(size=20) \
            .encode(
    # encode x as the percent, and hide the axis
                    x=alt.X('percentages', axis=None),
                    y=alt.Y('images:N', axis=alt.Axis(tickCount=5, title=''), sort= '-x')
                   )


    text = bars.mark_text(
                    align='left',
                    baseline='middle',
                    dx=3  # Nudges text to right so it doesn't appear on top of the bar
                         ) \
                .encode(
                    text=alt.Text('percentages:Q',format='.0%')
                       )

    final_bar = (text + bars).configure_mark(color='#008fd5') \
                             .configure_view(strokeWidth=0) \
                             .configure_scale(bandPaddingInner=0.2) \
                             .properties(width=400, height=800) \
                             .properties(
```

```
                                title={
                                        "text": ["The Paintings of Bob Ross"],
                                        "subtitle": ["Percentage containing each element"]
                                      }
                                        )
    return final_bar

    #raise NotImplementedError()
```

In [7]:
```
# run this code to validate
alt.themes.enable('fivethirtyeight')
makeBobRossBar(bobross, imgfeatures)
```

Out[7]:

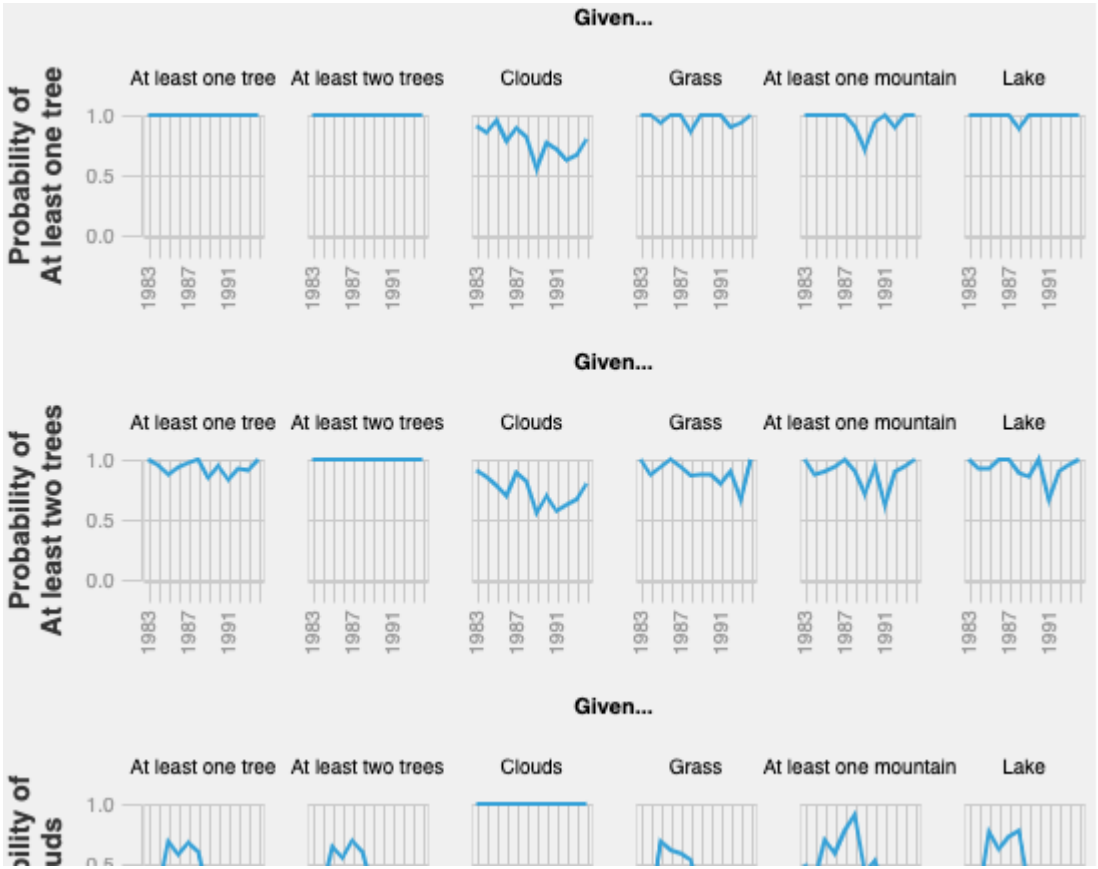





## Problem 2 (25 points)

The 538 article (Statistical Analysis of Bob Ross (https://fivethirtyeight.com/features/a-statistical-analysis-of-the-work-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 (assets/matrix_full.png)](). 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.

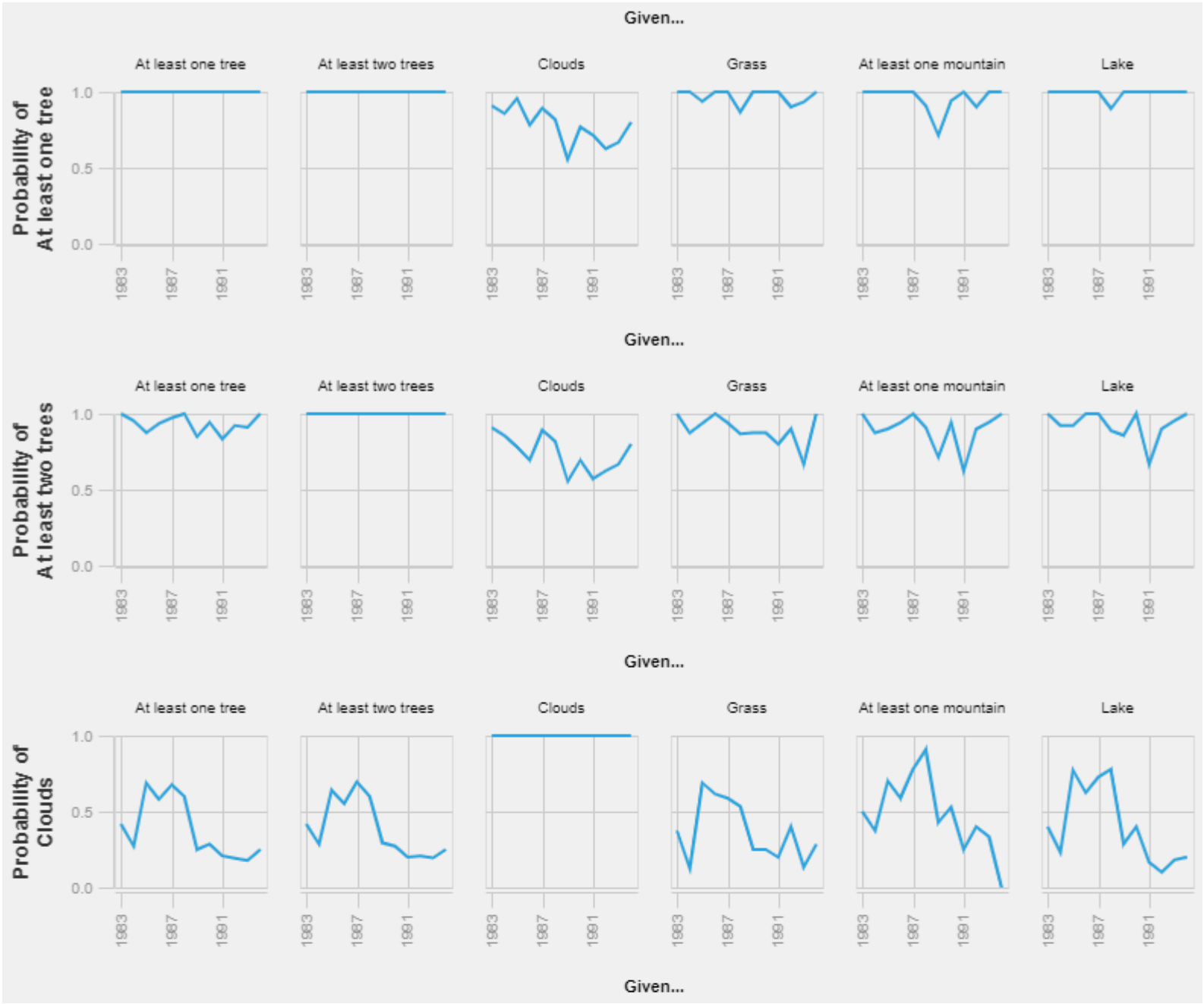
In [8]:
```python
def condprobability():
    # we suggest you implement this function to make your life easier.
    # input: frame -- the input dataframe in the style of the bobross dataframe above
    # input: column1 -- the first column to test (e.g, the A in probability of A given B)
    # input: 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

    keys1 = ['Lake', 'At least one tree', 'Grass', 'Clouds', 'At least one mountain', 'At least two trees']
    keys2 = ['Lake', 'At least one tree', 'Grass', 'Clouds', 'At least one mountain', 'At least two trees']
    years = [1983, 1984, 1985, 1986, 1987, 1988, 1989, 1990, 1991, 1992, 1993, 1994]

    tuple_list = []

    for key1 in keys1:
        for key2 in keys2:
            for year in years:
                bobross_year = bobross[bobross['year'] == year]
                prob_b = len(bobross_year[bobross_year[key2] == 1]) / len(bobross_year)

                prob_a_and_b = len(bobross_year[(bobross_year[key1] == 1) & (bobross_year[key2] == 1)]) / len(bo
                conditional_prob = prob_a_and_b / prob_b

                create_tuple = (key1, key2, year, conditional_prob)
                tuple_list.append(create_tuple)

    prob_df = pd.DataFrame(tuple_list, columns = ['key1', 'key2', 'year', 'prob'])
    return prob_df

    #raise NotImplementedError()
```

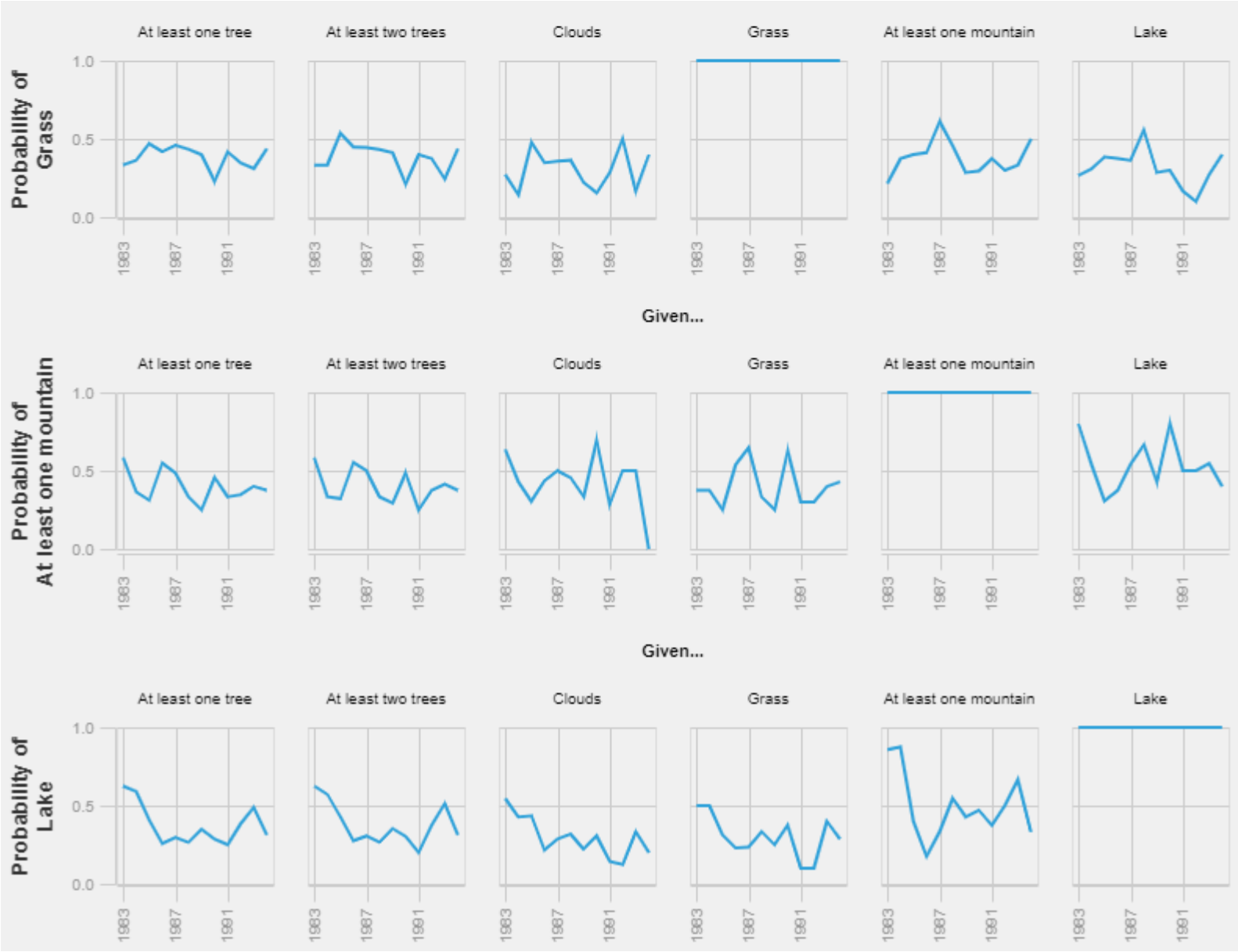
In [9]:
```
def makeBobRossCondProb(br, totest):
    # implement this function to return an altair chart
    #
    # input: br the dataframe (e.g., the bobross frame as defined above)
    # input: totest is a variable that holds an array of properties we want compared (see example below)

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

    # return alt.Chart(...)

    line_df = condprobability()

    prob_1_tree = line_df[line_df['key1'] == 'At least one tree']
    first_row = alt.Chart(prob_1_tree) \
                   .mark_line() \
                   .encode(
    # encode x as the percent, and hide the axis
                            x=alt.X('year:O', axis=alt.Axis(tickCount=10, title='', values = [1983, 1987, 1991])
                            y=alt.Y('prob:Q', axis=alt.Axis(tickCount=3, title=['Probability of', 'At least one
                            column = alt.Column('key2:N', sort = totest, title = 'Given...')
                           ) \
                    .properties(width = 100, height = 100)

    prob_2_tree = line_df[line_df['key1'] == 'At least two trees']
    second_row = alt.Chart(prob_2_tree) \
                   .mark_line() \
                   .encode(
    # encode x as the percent, and hide the axis
                            x=alt.X('year:O', axis=alt.Axis(tickCount=10, title='', values = [1983, 1987, 1991])
                            y=alt.Y('prob:Q', axis=alt.Axis(tickCount=3, title=['Probability of', 'At least two
                            column = alt.Column('key2:N', sort = totest, title = 'Given...')
                           ) \
                    .properties(width = 100, height = 100)

    prob_clouds = line_df[line_df['key1'] == 'Clouds']
    third_row = alt.Chart(prob_clouds) \
                   .mark_line() \
                   .encode(
    # encode x as the percent, and hide the axis
                            x=alt.X('year:O', axis=alt.Axis(tickCount=10, title='', values = [1983, 1987, 1991])
                            y=alt.Y('prob:Q', axis=alt.Axis(tickCount=3, title=['Probability of', 'Clouds'])),
                            column = alt.Column('key2:N', sort = totest, title = 'Given...')
```

```
                        ) \
                .properties(width = 100, height = 100)

    prob_grass = line_df[line_df['key1'] == 'Grass']
    fourth_row = alt.Chart(prob_grass) \
                .mark_line() \
                .encode(
    # encode x as the percent, and hide the axis
                        x=alt.X('year:O', axis=alt.Axis(tickCount=10, title='', values = [1983, 1987, 1991])
                        y=alt.Y('prob:Q', axis=alt.Axis(tickCount=3, title=['Probability of', 'Grass'])),
                        column = alt.Column('key2:N', sort = totest, title = 'Given...')
                        ) \
                .properties(width = 100, height = 100)

    prob_mountain = line_df[line_df['key1'] == 'At least one mountain']
    fifth_row = alt.Chart(prob_mountain) \
                .mark_line() \
                .encode(
    # encode x as the percent, and hide the axis
                        x=alt.X('year:O', axis=alt.Axis(tickCount=10, title='', values = [1983, 1987, 1991])
                        y=alt.Y('prob:Q', axis=alt.Axis(tickCount=3, title=['Probability of', 'At least one
                        column = alt.Column('key2:N', sort = totest, title = 'Given...')
                        ) \
                .properties(width = 100, height = 100)

    prob_lake = line_df[line_df['key1'] == 'Lake']
    sixth_row = alt.Chart(prob_lake) \
                .mark_line() \
                .encode(
    # encode x as the percent, and hide the axis
                        x=alt.X('year:O', axis=alt.Axis(tickCount=10, title='', values = [1983, 1987, 1991])
                        y=alt.Y('prob:Q', axis=alt.Axis(tickCount=3, title=['Probability of', 'Lake'])),
                        column = alt.Column('key2:N', sort = totest, title = 'Given...')
                        ) \
                .properties(width = 100, height = 100)

    return first_row & second_row & third_row & fourth_row & fifth_row & sixth_row



    #raise NotImplementedError()
```

In [10]:
```
# 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',
```

Out[10]:






## 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.

In terms of a deviation from the sample design, the only change that was made was to the x-axis such that, instead of 12 tick marks, only 3 (corresponding to the labelled years) are displayed. Despite this change, the output above has the same expressiveness as the sample plot and the decision to make this change was driven by readability. On such a small plot, the greater number of tick marks made it more difficult to view the lines themselves. Expressiveness is not lost and the viewer should still be able to assess the trend between years without the additional marks.

## 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?

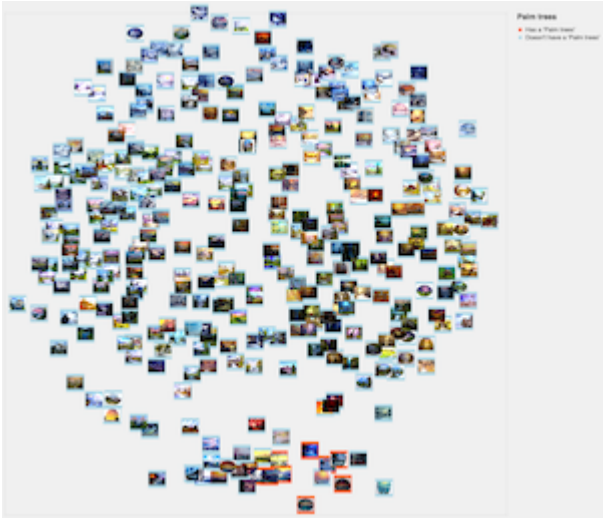
We are going to create a 2D MDS plot using the scikit learn package. We're going to do most of this for you in the next cell. Essentially we will use the euclidean distance between two images based on their image feature array to create the image. Your plot may look slightly different than ours based 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 (https://scikit-learn.org/stable/modules/generated/sklearn.manifold.MDS.html)](https://scikit-learn.org/stable/modules/generated/sklearn.manifold.MDS.html)

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

In [11]:
```
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()
```

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 (https://altair-viz.github.io/user_guide/marks.html#user-guide-image-mark). 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 (assets/mds_large.png) for a large version of this image. Notice the orange boxes indicating where the Palm tree images are. Note 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).*

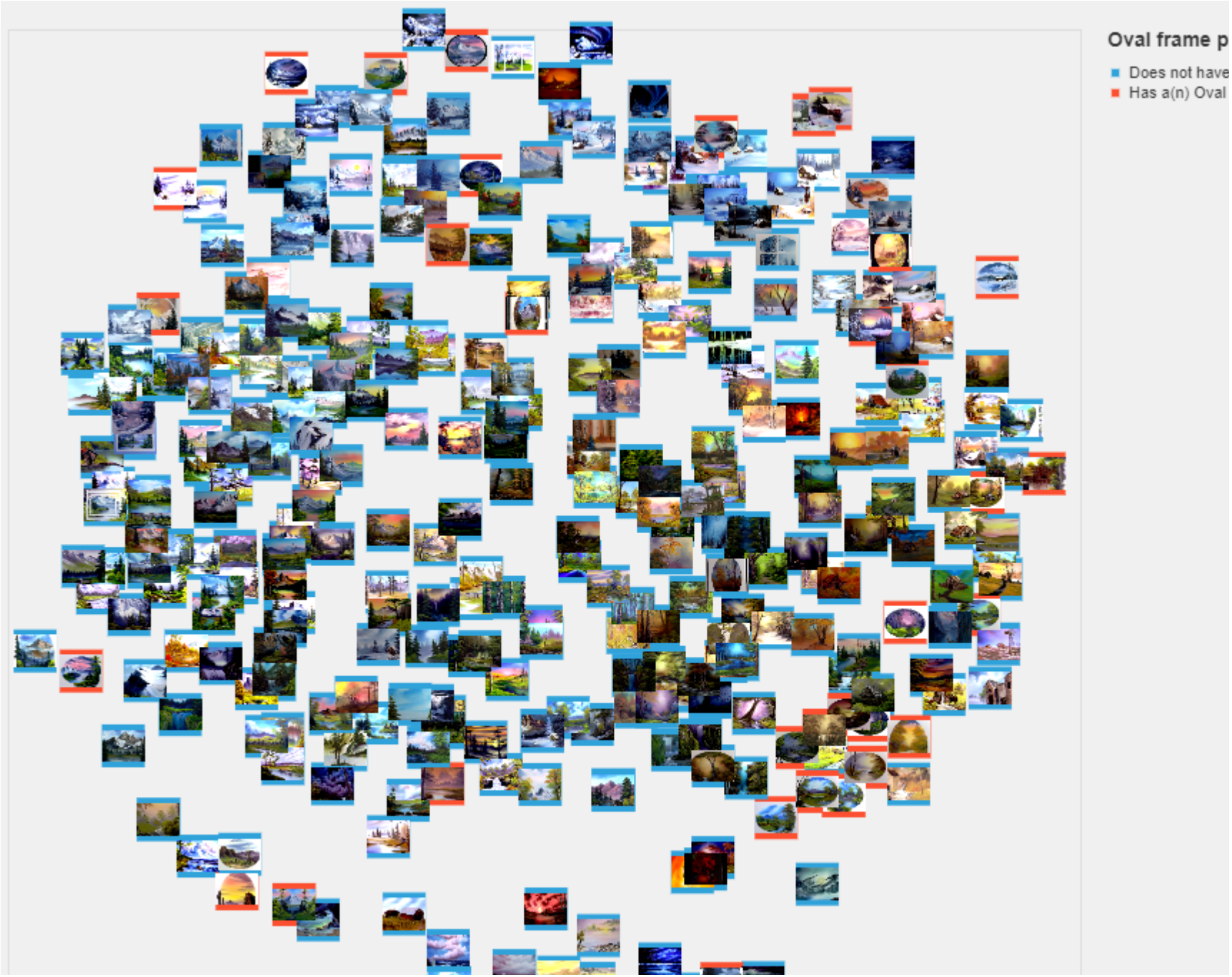
Hint: you may want to think about how to get "details" if you make images very small. We'd like to be able to figure out which image is what. A really simply strategy is to use something like tooltips.

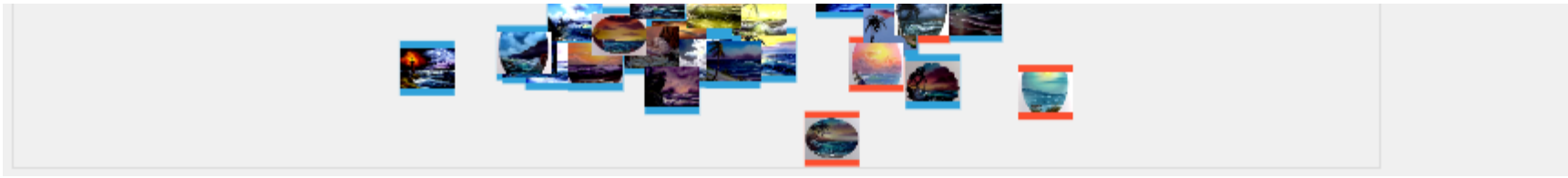
In [12]:

```
def genMDSPlot(br,key):
    # input: br -- a bobross dataframe (augmented with the x/y columns as describe above)
    # input: key -- is a string indicating which images should be visually highlighted (i.e., images containing
    #        should be made salient). For example: 'Barn'
    # return: an altair chart (e.g., return alt.Chart(...))

    br[key + ' present'] = np.where(br[key] == 1, 'Has a(n) ' + key, 'Does not have a(n) ' + key)

    img_chart = alt.Chart(br) \
                   .mark_image(width = 30, height = 30) \
                   .encode(
                            x=alt.X('x:Q', axis=None),
                            y=alt.Y('y:Q', axis=None),
                            url = 'img_url',
                            tooltip=['TITLE', key + ' present']
                           ) \
                    .properties(width = 750, height = 750)

    color_chart = alt.Chart(br) \
                     .mark_rect(width = 30, height = 30) \
                     .encode(
                              x=alt.X('x:Q', axis=None),
                              y=alt.Y('y:Q', axis=None),
                              color = key + ' present'
                             )

    return color_chart + img_chart

    #raise NotImplementedError()
```

In [13]:
```
# you should be able to test your code without interactivity, for example:
genMDSPlot(bobross,'Oval frame')
```

Out[13]:





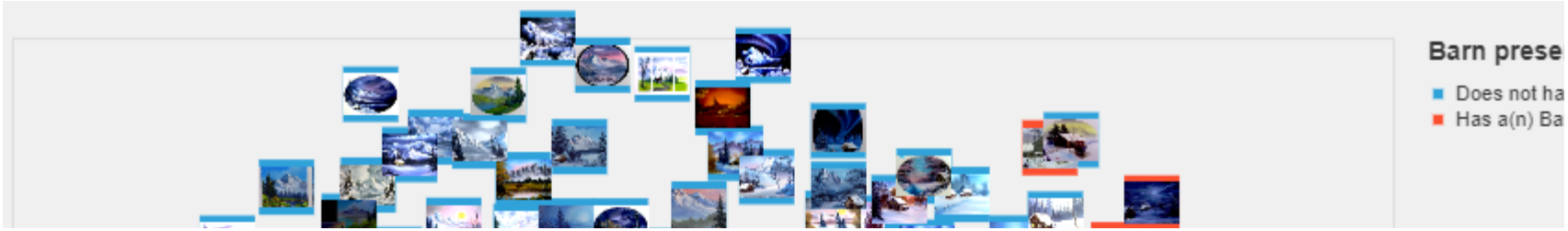
We are going to create an interactive widget that allows you to select the feature you want to be highlighted. If you implemented your `genMDSPlot` code correctly, the plot should change when you select new items from the list. We would ordinarily do this directly in Altair, but because we don't have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

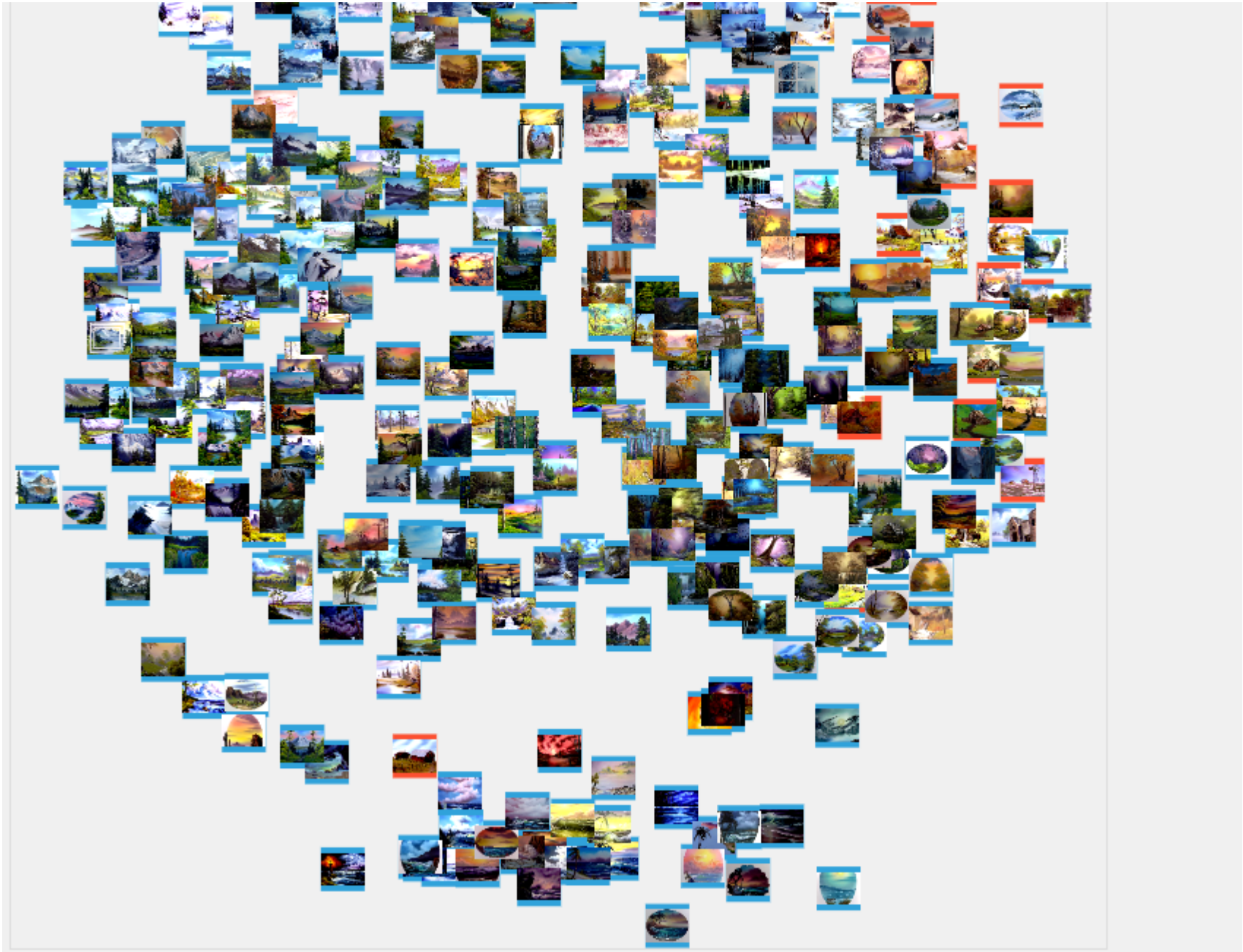
It should look something like this:


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

In [14]:
```
# 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()
```

Highlight: Barn

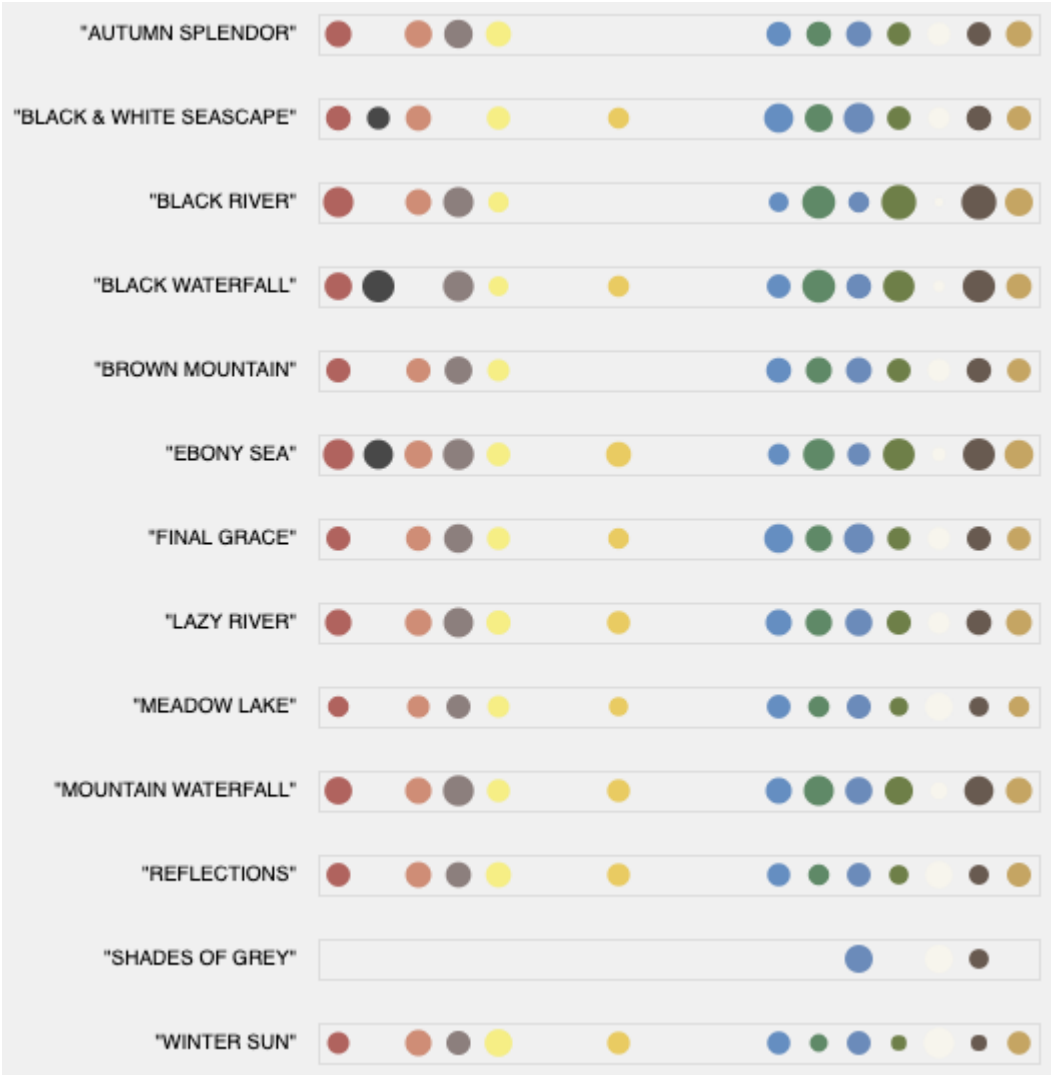




# Problem 4 (30 points: 25 for solution, 5 for explanation)

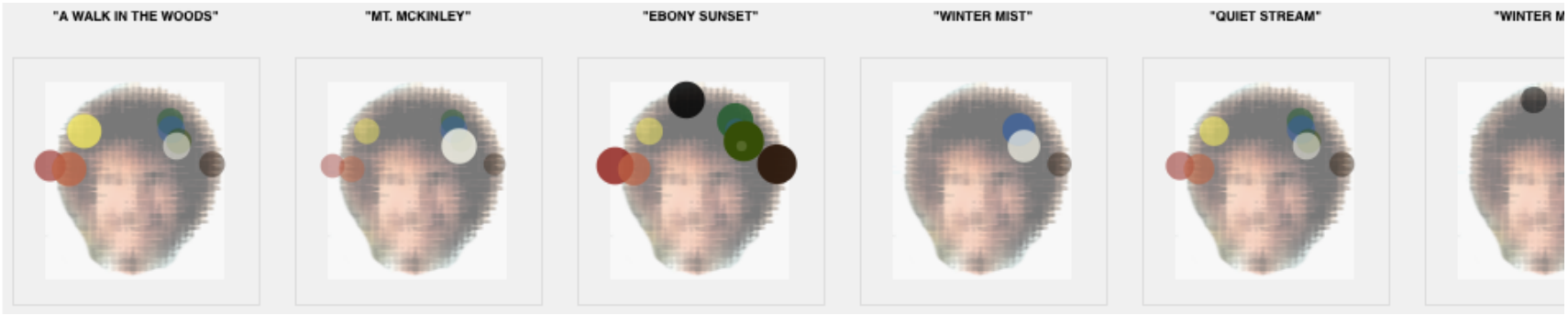
Your last problem is fairly open-ended in terms of visualization. We would like to analyze the colors used in different images for a given season as a small multiples plot. You can pick how you represent your small multiples, but we will ask you to defend your choices below. You must implement the function `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).

You can go something as simple as this:



This visualization has a row 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.

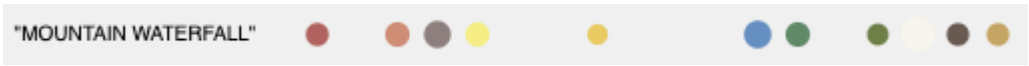
Again, the relevant columns are available are listed in `rosspaints` (there are 18 of them). The values range from 0 to 1 based on the fraction of pixel color allocated to that specific paint. The `rosspainthex` has the corresponding hex values for the paint color.

*Some notes*

1) We'd advise against trying to replicate our examples but if you do make sure you discuss the cons in detail

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.

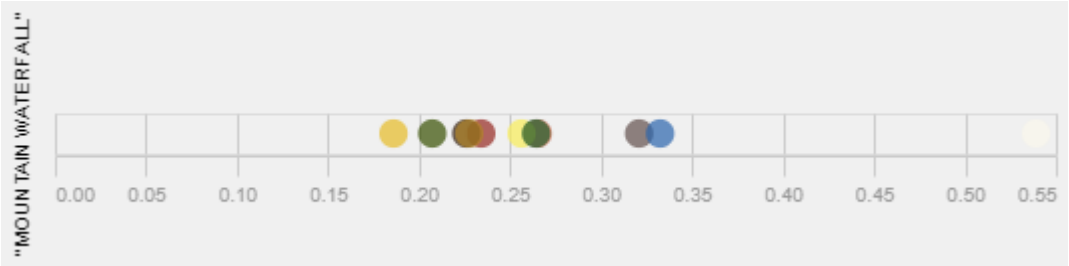
3) You *may* find it useful to implement "colorSmallMultiple" as below to generate your single small multiple. This may not be ideal if you're using faceting or repetition. For example, in our implementation calling `colorSmallMultiple(5,1)` will create a small multiple for season 5, episode 1:



In [15]:

```
# this is optional, you can use this to produce a single multiple
# you may not find this helpful for your solution
def colorSmallMultiple(season, episodenumber, br=bobross, rp=rosspaints, rph=rosspainthex):
    # input: season -- a season number (integer), assumed to exist in the dataset
    # input: episodenumber -- an episode number (integer), assumed to exist in the dataset
    # input: br -- a dataset structed as the bobross data above (default is "bobross")
    # input: rp -- the names of paints (default rosspaints as defined above)
    # input: rph -- the hex values of the paints (default rosspaintshex as defined above)
    # return: a single multiple visualization for the season/episode

    #Use lookaheads and lookbehinds
    br['season'] = br['EPISODE'].str.extract('((?<=S).*(?=E))')
    br['season'] = br['season'].str.lstrip('0')

    br['ep'] = br['EPISODE'].str.extract('((?<=E).*)')
    br['ep'] = br['ep'].str.lstrip('0')

    ep_df = br[(br['season'] == season) & (br['ep'] == episodenumber)]
    melt_df = pd.melt(ep_df, id_vars=['TITLE', 'season', 'ep'], value_vars=rp)
    melt_df = melt_df[melt_df['value'] > 0]

    palette = alt.Scale(domain=rp,
                        range=rph
                       )

    chart = alt.Chart(melt_df).mark_circle(size = 200) \
              .encode(
                   x=alt.X('value', title = ''),
                   color = alt.Color('variable', scale = palette, legend = None),
                   tooltip=['TITLE', 'variable', 'value'],
                   row = alt.Row('TITLE', title = '')
                     ) \
              .properties(width = 500, height = 20)

    return chart

    #raise NotImplementedError()


# test
colorSmallMultiple('12','10')  # season 12, episode 10
colorSmallMultiple('5','1')    # season 5, episode 1
```

Out[15]:

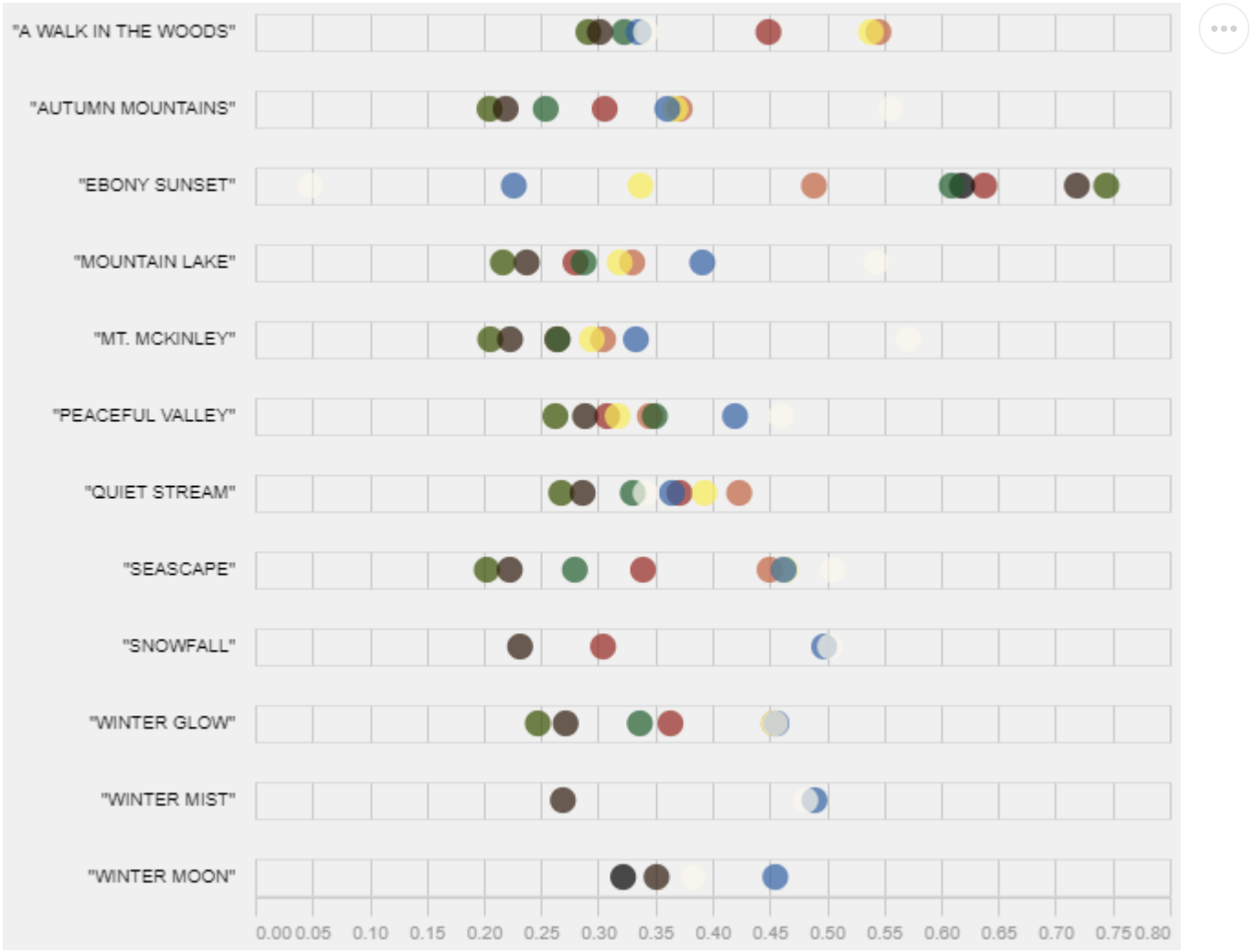

"MOUN TAIN WATERFALL"
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0.45
0.50
0.55

In [16]:

```
def colorSmallMultiples(season, br=bobross, rp=rosspaints, rph=rosspainthex):
    # input: season -- a season number (integer), assumed to exist in the dataset. This is the
    #               integer representing the season of the show are interested in. Limit your images
    #               to that season in the small multiples display.
    # input: br -- a dataset structed as the bobross data above (default is "bobross")
    # input: rp -- the names of paints (default rosspaints as defined above)
    # input: rph -- the hex values of the paints (default rosspainthex as defined above)
    # return: an Altair chart providing small multiples for that season

    #Use lookaheads and lookbehinds
    br['season'] = br['EPISODE'].str.extract('((?<=S).*(?=E))')
    br['season'] = br['season'].str.lstrip('0')

    br['ep'] = br['EPISODE'].str.extract('((?<=E).*)')
    br['ep'] = br['ep'].str.lstrip('0')

    season_df = br[br['season'] == season]
    melt_df = pd.melt(season_df, id_vars=['TITLE', 'season', 'ep'], value_vars=rp)
    melt_df = melt_df[melt_df['value'] > 0]

    palette = alt.Scale(domain=rp,
                        range=rph
                       )

    chart = alt.Chart(melt_df).mark_circle(size = 200) \
                .encode(
                     x=alt.X('value', title = ''),
                     color = alt.Color('variable', scale = palette, legend = None),
                     tooltip=['TITLE', 'variable', 'value'],
                     row = alt.Row('TITLE', title = '', header=alt.Header(labelAngle=0, labelAlign = 'left'))
                       ) \
                .properties(width = 500, height = 20)

    return chart

    #raise NotImplementedError()
```

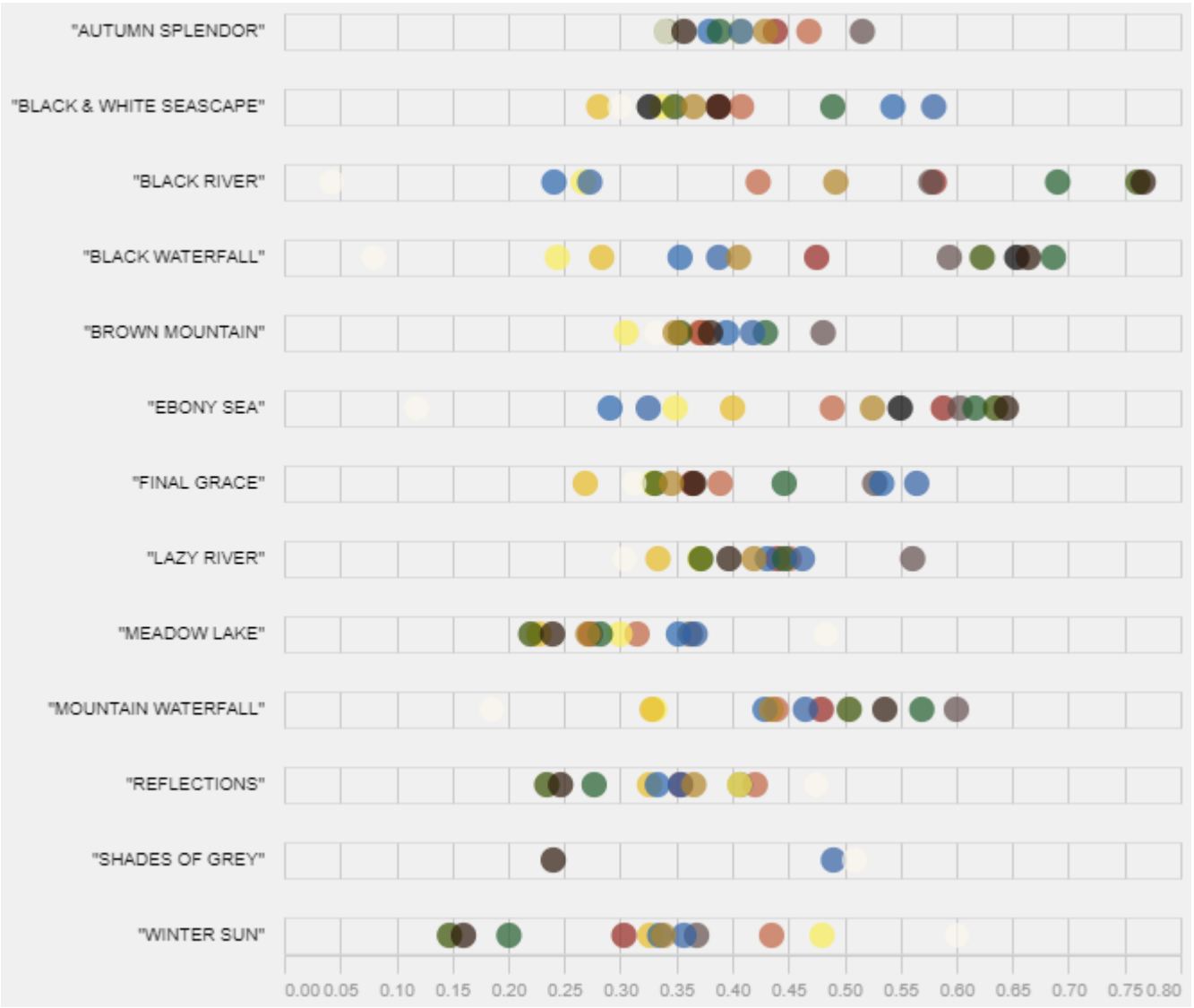
In [17]:
```
# run this to test your code for season 1
colorSmallMultiples('1')
```

Out[17]:



In [18]:
```
# run this to test your code for season 2
colorSmallMultiples('2')
```

Out[18]:




## Explain your choices

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

My design uses circles as the mark that have been colored based on the color they correspond to in the painting. A color's "prevalence" in a painting is encoded as x-position, and a tooltip encoding allows for a reader to scan over a mark and see the painting's name, the color the circle corresponds to, and its "prevalence" within the painting. Lastly, a row encoding allowed for this small multiple visualization, as each painting from a season is given its own row, with the title of each painting available as a label. The function to generate the visualization will just need to take a season input and will generate the above design.

The design used for the visualization would be perhaps the same level of expressiveness as the two sample examples, but would be a much more effective presentation of the data. The two sample visualizations rely on a size encoding to display the prevalence of colors in each of the paintings from a particular season. Viewers have more difficulty comparing the size of one object to another, and, therefore, a significant pro of my visualization is to encode a color's "prevalence" by x-position instead. A reader can now see much easier which colors were used in a painting and to the level in which they were used.

In addition to encoding prevalence in a more user-friendly way, my visualization I believe is also far more effective due to providing x-axis ticks and a tooltip encoding. A viewer is able to clearly see and compare prevalence based on the x-axis ticks, but can also get more exact information by hovering over a mark and seeing the color name and its exact prevalence. The sample encodings did encode prevalence based on size but no legend was available, which would have made comparisons and magnitude difficult to assess.

In terms of a con, there is a trade-off between mark size and visualization height and width. There are cases where multiple marks do overlap, making hovering with the tooltip encoding difficult as well as the viewer's ability to assess which colors are included and their prevalence. The mark size and visualization height/width were chosen to be as clear as possible for the viewer while not resulting in too large of a display, but color overlaps would still be a clear con for this design approach.

In [ ]: