In [1]:
```
# initializing otter-grader
import otter
grader = otter.Notebook()
```

HW 4: Principal Component Analysis

In lecture we discussed how PCA can be used for dimensionality reduction. Specifically, given a high-dimensional dataset, PCA allows us to:

1. Understand the rank of the data. If k principal components capture almost all of the variance, then the data is effectively rank k.
2. Create 2D scatterplots of the data. Such plots are a rank 2 representation of our data, and allow us to visually identify clusters of similar observations.

A solid geometric understanding of PCA will help you understand why PCA is able to do these things. In this homework, we'll build that geometric intuition, and will also look at PCA on different datasets.

Due Date

This assignment is due **Sunday 5/17 at 11:59pm PST**.

Collaboration Policy

Data science is a collaborative activity. While you may talk with others about the homework, we ask that you **write your solutions individually**. If you do discuss the assignments with others please **include their names** in the cell below.

Collaborators: ...

In [2]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
import altair as alt
```

The first time you run this notebook, you'll need to install a visualization package called plotly. To do so, you need to run the following command **only once**:

```
!pip install plotly
```

Paste and run the following command in the cell below and once the package is installed, you can delete the `pip` command and import `plotly` as usual.

In [3]:
```
!pip install plotly
```

```
Requirement already satisfied: plotly in /opt/conda/lib/python3.7/site-
packages (4.7.1)
Requirement already satisfied: retrying>=1.3.3 in /opt/conda/lib/python
3.7/site-packages (from plotly) (1.3.3)
Requirement already satisfied: six in /opt/conda/lib/python3.7/site-pac
kages (from plotly) (1.14.0)
```

In [4]:
```
import plotly.express as px

# Note: If you're having problems with the 3d scatter plots,
# uncomment the two lines below, and you should see a version that
#      number that is at least 4.1.1.
# import plotly
# plotly.__version__
```

Question 1: PCA on 3D Data

In question 1, our goal is to see visually how PCA is simply the process of rotating the coordinate axes of our data.

The code below reads in a 3D dataset. We have named the variable `surfboard` because the data resembles a surfboard when plotted in 3D space.

In [5]:
```
surfboard = pd.read_csv("data3d.csv")
surfboard.head(5)
```

Out[5]:

	x	y	z
0	0.005605	2.298191	1.746604
1	-1.093255	2.457522	0.170309
2	0.060946	0.473669	-0.003543
3	-1.761945	2.151108	3.132426
4	1.950637	-0.194469	-2.101949

The cell below will allow you to view the data as a 3d scatterplot. Rotate the data around and zoom in and out using your trackpad or the controls at the top right of the figure.

You should see that the data is an ellipsoid that looks roughly like a surfboard or a hashbrown patty (https://www.google.com/search?q=hashbrown+patty&source=lnms&tbm=isch). That is, it is pretty long in one direction, pretty wide in another direction, and relatively thin along its third dimension. We can think of these as the "length", "width", and "thickness" of the surfboard data.

Observe that the surfboard is not aligned with the x/y/z axes.

If you get an error that your browser does not support webgl, you may need to restart your kernel and/or browser.

In [6]:
```
fig = px.scatter_3d(surfboard, x='x', y='y', z='z', range_x = [-10, 10
], range_y = [-10, 10], range_z = [-10, 10])
fig.show()
```

To give the figure a little more visual pop, we will create a separate dataframe where we'll add pre-determined color values (that we've arbitrarily chosen) to each point. These colors do not mean anything important, they're simply there as a visual aid.

We will use `colorized_surfboard` only for visualization.

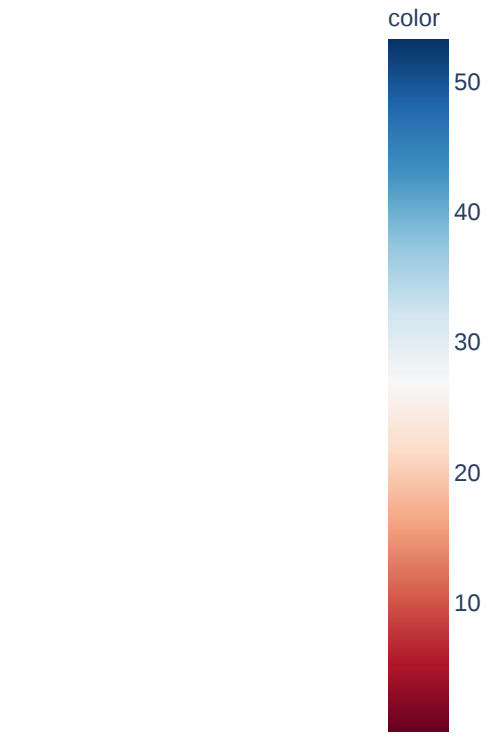
In [7]:
```
s_colors = pd.read_csv("surfboard_colors.csv", header = None).values
colorized_surfboard = surfboard.copy()
colorized_surfboard.insert(loc = 3, column = "color", value = s_colors)
colorized_surfboard.head(5)
```

Out[7]:

	x	y	z	color
0	0.005605	2.298191	1.746604	8.252443
1	-1.093255	2.457522	0.170309	7.170102
2	0.060946	0.473669	-0.003543	0.195313
3	-1.761945	2.151108	3.132426	17.628558
4	1.950637	-0.194469	-2.101949	8.082033

To give the figure a little more visual pop, the following cell does the same plot, but now uses `colorized_surfboard` to display the values.

In [8]:
```
fig = px.scatter_3d(colorized_surfboard, x='x', y='y', z='z', range_x =
 [-10, 10], range_y = [-10, 10], range_z = [-10, 10], color = "color",
color_continuous_scale = 'RdBu')
fig.show()
```



Question 1a: center the data

Now that we've understood the data, let's work on understanding what PCA will do when applied to this data.

If you consult Wikipedia (https://en.wikipedia.org/wiki/Principal_component_analysis), you'll see that "the original data is normalized before performing the PCA. The normalization of each attribute consists of *mean centering* – subtracting each data value from its variable's measured mean so that its empirical mean (average) is zero. Some fields, in addition to normalizing the mean, do so for each variable's variance (to make it equal to 1)..." (*Retireved May 2020*).

For this exercise, we will not be scaling the variables, so that we can see how to compute the total variance from the singular values.

To properly perform PCA, we will first need to "center" the data so that the mean of each feature is 0.

Compute the columnwise mean of `surfboard` in the cell below, and store the result in `surfboard_mean`. You can choose to make `surfboard_mean` a numpy array or a series, whichever is more convenient for you. Regardless of what data type you use, `surfboard_mean` should have **3 means**, 1 for each attribute, with the `x` coordinate first, then `y`, then `z`.

Then, subtract `surfboard_mean` from `surfboard`, and save the result in `surfboard_centered`. The order of the columns in `surfboard_centered` should be `x` first, then `y`, then `z`.

In [9]:
```
surfboard_mean = surfboard.mean()
surfboard_centered = np.add(surfboard, -surfboard_mean)
```

Question 1b: SVD

As you may recall from lecture, PCA is a specific application of the singular value decomposition (SVD) for matrices. If we have a data matrix X, we can decompose it into U, Σ and V^T such that $X = U\Sigma V^T$. Here, U is the left singular vectors, Σ is a diagonal matrix containing the singular values, and V^T are the right singular vectors.

In the following cell, use the `np.linalg.svd` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.svd.html) function to compute the SVD of `surfboard_centered`. Store the left singular vectors, singular values, and right singular vectors in `u`, `s`, and `vt` respectively. This is one line of simple code, exactly like what we saw in lecture.

Hint: Set the `full_matrices` argument of `np.linalg.svd` to `False`.

In [10]:
```
u, s, vt = np.linalg.svd(surfboard_centered, full_matrices=False)
```

Question 1c: Total Variance

Let's now consider the relationship between the singular values `s` and the variance of our data. Recall that the total variance is the sum of the variances of each column of our data. Below, we provide code that computes the variance for each column of the data.

Note: The variances are the same for both `surfboard_centered` and `surfboard`, so we show only one to avoid redundancy.

In [11]:
```
np.var(surfboard, axis=0)
```

Out[11]:
```
x    2.330704
y    5.727527
z    4.783513
dtype: float64
```

The total variance of our dataset is given by the sum of these numbers.

In [12]:
```
total_variance_computed_from_data = sum(np.var(surfboard, axis=0))
total_variance_computed_from_data
```

Out[12]:
```
12.841743509780109
```

As discussed in lecture, the total variance of the data is also equal to the sum of the squares of the singular values divided by the number of data points, that is:

$$Var(X) = \frac{\sum_{i=1}^{d} s_i^2}{N}$$

In the cell below, compute the total variance using the the formula above and store the result in the variable `total_variance_computed_from_singular_values`. Your result should be very close to `total_variance_computed_from_data`.

In [13]:
```
total_variance_computed_from_singular_values = sum(s**2)/len(surfboard_
centered)
```

Question 1d: Explained Variance and Scree Plots

In the cell below, set `variance_explained_by_1st_pc` to the proportion of the total variance explained by the 1st principal component. Your answer should be a number between 0 and 1.

Note: This topic was discussed in lecture.

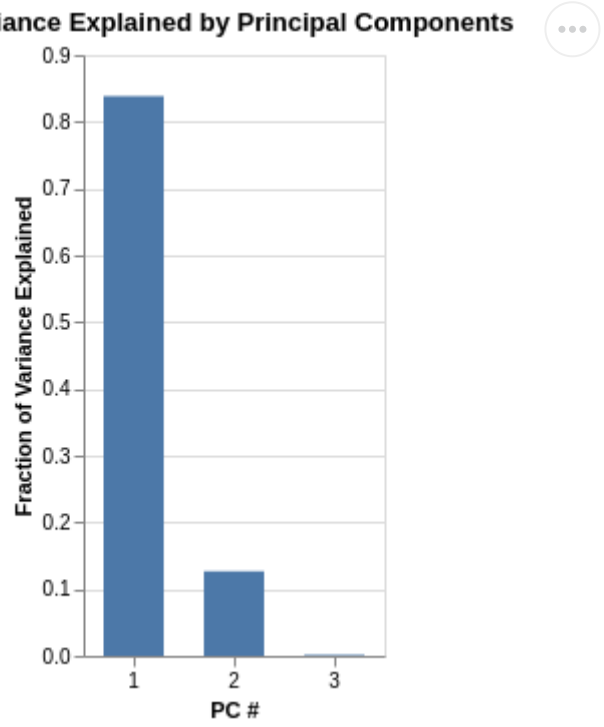
In [14]:
```
variance_explained_by_1st_pc = s[0]**2/(sum(s**2))
variance_explained_by_1st_pc
```

Out[14]:
```
0.8385084140449134
```

We can also create a scree plot that shows the variance explained by all of our principal components, ordered from most to least. Here the y-axis is the amount of variance explained by the ith principal component.

In [15]:
```
explained_var = pd.DataFrame({
    'PC #': [1, 2, 3],
    'Fraction of Variance Explained' : [(s[0]**2/(sum(s**2))), (s[1]**2
/(sum(s**2))), (s[2]/(sum(s**2)))]
})

# Draw your Altair visualization
alt.Chart(explained_var,
          title="Variance Explained by Principal Components"
).mark_bar(size=30).encode(
    alt.X('PC #:O'),
    alt.Y('Fraction of Variance Explained:Q')
).configure_axisX(labelAngle=0).properties(width=150)
```

Out[15]:



If we divide by the total variance, we can get the fraction explained by each component. Note that the first y value is the same as what you computed earlier in this problem.

Note: If you're wondering where `len(surfboard_centered)` went, it got canceled out when we divided the variance of a given PC by the total variance.

For this small toy problem, the scree plot is not particularly useful. We'll see why they are useful in practice later in this homework.

Question 1e: V as a Rotation Matrix

In lecture, we saw that the first column of $U\Sigma$ contained the first principal component values for each observation, the second column of $U\Sigma$ contained the second principal component values for each observation, and so forth.

Let's give this matrix a name: $P = U\Sigma$ is sometimes known as the "principal component matrix".

The code below computes P using U and Σ, then prints out the principal components for the 5th observation in the dataset.

(Note the use of `@` to multiply two matrices in Python.)

In [16]:
```
P = u @ np.diag(s)
print(f"The 5th observation in x/y/z is: {surfboard.iloc[4, 0]}, {surfboard.iloc[4, 1]}, {surfboard.iloc[4, 2]}")
print(f"The 1st, 2nd, and 3rd pcs of our 5th observation are: {P[4, 0]}, {P[4, 1]}, {P[4, 2]}")
```

```
The 5th observation in x/y/z is: 1.950637304600804, -0.1944692091869376, -2.1019492971930087
The 1st, 2nd, and 3rd pcs of our 5th observation are: 2.209110214853996, -1.784578358906228, -0.13093802139261515
```

We saw in lecture how to interpret V^T, which is sometimes known as the "mixing matrix" or "loadings matrix". In the cell below, we show V^T.

In [17]:
```
vt
```

Out[17]:
```
array([[ 0.38544534, -0.67267377, -0.63161847],
       [-0.5457216 , -0.7181477 ,  0.43180066],
       [-0.74405633,  0.17825229, -0.64389929]])
```

We know from discussion that V^T gives us a way to convert our data into principal component values. For example, since the first row of V^T is $[0.38, -0.67, -0.63]$, we have that:

$$\text{pc1} = 0.38x - 0.67y - 0.63z$$

Thus for the 5th observation we get $2.21 = 0.39 \cdot 1.95 - 0.67 \cdot -0.19 - 0.63 \cdot -2.1$.

Another way to think about V is that it is a rotation matrix that transforms our original data matrix X (through rotation) into P. This is given by the simple relationship that $P = XV$, which we prove below:

Proposition: $P = XV$

Proof:

1. Because $X = U\Sigma V^T$, we have that $X = PV^T$.
2. As mentioned in lecture 10 (https://drive.google.com/open?id=1_zgMHZ3IpJtBd3fe1j-TH9pkleAj8Okz), one special property of V^T is that its inverse is also its transpose.
3. Thus if we multiply both sides of $X = PV^T$, we get $XV = PV^TV$. Since V^TV is the identity matrix, we have $XV = P$.

In other words, another way to compute our principal component matrix P is $P = XV$. In the cell below, compute P using the original data and V. Assign the result to a variable called `surfboard_pcs`.

Hint: In python, you can use `.T` to form the transpose of a numpy array. For example `u.T` is equivalent to saying U^T.

In [18]:
```
surfboard_pcs = np.dot(surfboard, vt.T)
surfboard_pcs = pd.DataFrame(surfboard_pcs)
```

Visualizing the Principal Component Matrix

In some sense, we can think of P as an output of the PCA procedure.

It is simply a rotation of the data such that the data will now appear "axis aligned". Specifically, for a 3d dataset, if we plot pc1, pc2, and pc3 along the x, y, and z axes of our plot, then the greatest amount of variation happens along the x axis, the second greatest amount along the y axis, and the smallest amount along the z axis.

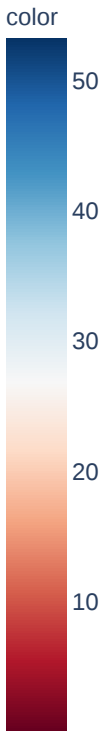
To visualize this, run the cell below, which will show our data now projected onto the principal component space. Compare with your original figure, and observe that the data is exactly the same, only it is now rotated.

In [19]:
```
surfboard_pcs = surfboard_pcs.rename(columns = {0: "pc1", 1: "pc2", 2: "pc3"})

colorized_surfboard_pcs = surfboard_pcs.copy()
colorized_surfboard_pcs.insert(loc = 3, column = "color", value = s_colors)
colorized_surfboard_pcs.head(5)
```

Out[19]:

	pc1	pc2	pc3	color
0	-2.646960	-0.899315	-0.719150	8.252443
1	-2.182070	-1.094711	1.141841	7.170102
2	-0.292896	-0.374954	0.041367	0.195313
3	-4.104626	0.769302	-0.322541	17.628558
4	2.210308	-1.832470	-0.132605	8.082033

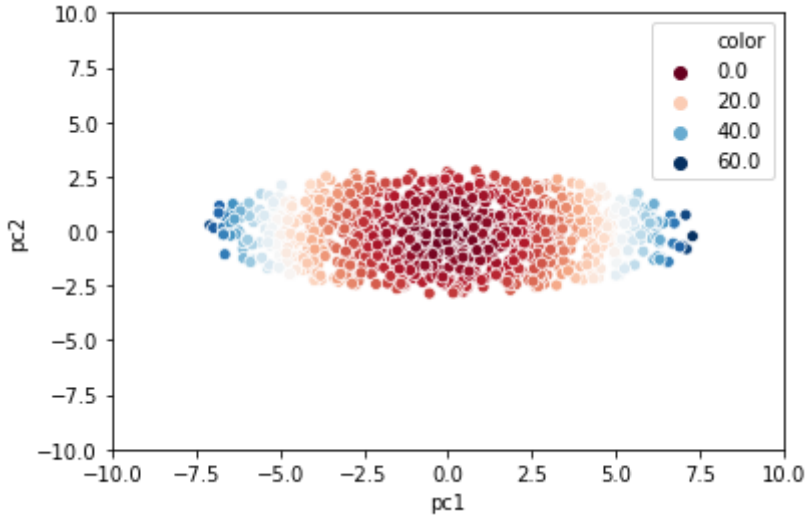
In [20]:
```
fig = px.scatter_3d(colorized_surfboard_pcs,
                    x='pc1', y='pc2', z='pc3',
                    range_x = [-10, 10], range_y = [-10, 10], range_z =
 [-10, 10],
                    color = 'color', color_continuous_scale = 'RdBu');
fig.show();
```



Recall in lecture that we created a scatter plot of only the first two principal components. We can do that here with our surfboard data as well.

Note that the result is just the 3D plot as viewed from directly "overhead".

In [21]:
```
sns.scatterplot(data = colorized_surfboard_pcs,
                x = 'pc1', y = 'pc2', hue = "color", palette = "RdBu")
plt.gca().set_xlim(-10, 10);
plt.gca().set_ylim(-10, 10);
```



Question 1 Summary

Above, we saw that the principal component matrix P is simply the original data rotated in space so that it appears axis aligned.

We also saw that P can be computed as $P = U\Sigma$ or as $P = XV$.

Whenever we do a 2D scatter plot of only the first 2 columns of P, we are simply looking at the data from "above", i.e. so that the 3rd (or higher) PC is invisible to us.

Question 2

PCA really shines on data where you have reason to believe that the data is relatively low in rank. For example, in lecture, we looked at congressional votes -- the current high degree of political polarization means that congresspeople will mostly vote in line with their party. And indeed, we saw that the first principal component very strongly separated republicans from democrats.

In this final question of the homework, we'll look at how states voted in presidential elections between 1972 and 2016. **Our ultimate goal in Question 2 is to show how 2D PCA scatterplots can allow us to identify clusters in a high dimensional dataset.** For this example, that means finding groups of states that vote similarly by plotting their 1st and 2nd principal components.

Question 2a: Get the source data

Unlike prior assignments, we're going to make you go **get the data yourself**. Specifically, we'd like you to use this table from wikipedia:
https://en.wikipedia.org/wiki/List_of_United_States_presidential_election_results_by_state (https://en.wikipedia.org/wiki/List_of_United_States_presidential_election_results_by_state).

You can convert the table into csv format this website: https://wikitable2csv.ggor.de/ (https://wikitable2csv.ggor.de/). Simply paste the URL of the URL into wikitable2csv and leave the default options as they are.

Then click download on Table 1, and you should download a file called `table-1.csv` .

Upload this file to your Jupyterhub folder (where this ipynb is) and rename it " `presidential_elections.csv` ". Then run the cell below to make sure that you did everything properly.

In [22]:
```
df = pd.read_csv("presidential_elections.csv")
df.head(5)
```

Out[22]:

	State	1789	1792	1796	1800	Unnamed: 5	1804	1808	1812	1816	...	1988	1992	1996
0	Alabama	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	R	R	R
1	Alaska	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	R	R	R
2	Arizona	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	R	R	D
3	Arkansas	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	R	D	D
4	California	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	R	D	D

5 rows × 66 columns

Question 2b: Clean the data

The data in this table is pretty messy. Create a clean version of this table called `df_1972_to_2016`. It should contain exactly 51 rows (corresponding to the 50 state plus Washington DC) and 12 columns (one for each of the election year from 1972 to 2016, to include only republican `R` and democratic `D` votes in each state).

The index of this dataframe should be the state name. The name you pick for the index doesn't matter.

The column names contain **only** the numerical values: i.e., only the *numerical value for the year*, NO extraneous symbols.

Hint: Our solution uses `iloc`, `drop` (twice), `rename`, and `set_index`.

Note: Feel free to open your csv file in Excel or Google Slides to explore the data if you find that easier. However, we require that you **do your actual data cleaning in pandas**, i.e. don't just delete and rename columns in Excel.

Note: In your personal projects, it is sometimes more convenient to manually do your data cleaning using Excel or Google Sheets. The downside of doing this is that you have no record of what you did, and if you have to redownload the data, you have to redo the manual data cleaning process.

Hint: It will be easiest for you to start by extracting the last 14 columns and begin cleaning from there (remember that Python allows negative indexing, which you can use in `iloc`).

Hint: Remember that `.drop` allows you to drop rows as well as columns (using the `axis` parameter).

In [23]:
```
df_1972_to_2016 = df.iloc[:,-14:].dropna(axis=1)
df_1972_to_2016 = df_1972_to_2016.drop(index=[25,52])
df_1972_to_2016 = df_1972_to_2016.rename(columns={"2000 ‡":2000, "2016
 ‡":2016 })
df_1972_to_2016 = df_1972_to_2016.set_index('State.1')
df_1972_to_2016.shape
```

Out[23]:
```
(51, 12)
```

Question 2c: get numerical values

To perform PCA, we need to convert our data into being numerical. To do this, replace all of the "D" characters with the number 0, and all of the "R" characters with the number 1. Store the resulting dataframe in a new variable `df_1972_to_2016_num`.

Hint: Use `df.replace` (which by default *returns* the modified dataframe without affecting the original).

In [24]:
```
df_1972_to_2016_num = df_1972_to_2016.replace("D", 0)
df_1972_to_2016_num = df_1972_to_2016_num.replace("R", 1)
```

Question 2d: center and scale the data

Now **center the data** so that the mean of each column is 0 and **scale the data** so that the variance of each column is 1. Store your result in `df_1972_to_2016`.

Hint: Remember that `np.mean` and `np.std` allow you to use the `axis` parameter to compute the mean/standard deviation of the columns (instead of the flattened array).

In [25]:
```
df_mean = np.mean(df_1972_to_2016_num, axis=0)
df_std = np.std(df_1972_to_2016_num, axis=0)
df_1972_to_2016 = ((df_1972_to_2016_num - df_mean)/df_std)
```

Question 2e: SVD

We now have our data in a nice and tidy centered and scaled format, phew. We are now ready to do PCA.

Now, **create a new dataframe** `first_2_pcs` that contains exactly the first two columns of the principal components matrix. The first column should be labeled `pc1` and the second column should be labeled `pc2`. Store your result in `first_2_pcs`.

Hint: Just like you did before, use `np.linalg.svd`. **Do not overwrite** `u, s, vt` that you defined earlier, **use `u1, s1, vt1`** for the result of `np.linalg.svd`.

Hint: You can use Python's slicing method to extract the first two columns.

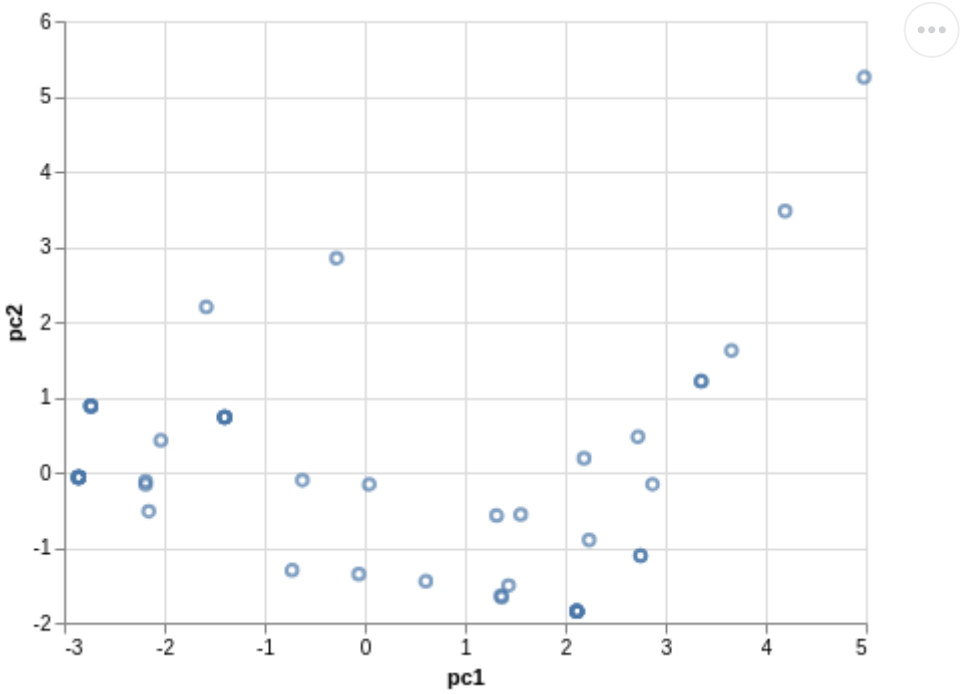
In [26]:
```
u1, s1, vt1 = np.linalg.svd(df_1972_to_2016, full_matrices=False)
first_2_pcs = np.dot(df_1972_to_2016, vt1.T)
first_2_pcs = pd.DataFrame(first_2_pcs)
first_2_pcs = first_2_pcs.rename(columns = {0: "pc1", 1: "pc2"}).iloc
[:, 0:2]
```

Question 2f: plot the first two principal components

The cell below plots the 1st and 2nd principal components of our 50 states + Washington DC.

In [27]:
```
alt.Chart(first_2_pcs).mark_point().encode(
    x = "pc1",
    y = "pc2"
)
```

Out[27]:



Unfortunately, we have two problems:

1. There is a lot of overplotting, with only 27 distinct dots. This means that at least some states voted exactly alike in these elections.
2. We don't know which state is which, because the points are unlabeled.

Let's start by addressing problem 1.

In the cell below, create a new dataframe `first_2_pcs_jittered` with a small amount of random noise added to each principal component. In this same cell, create a scatterplot.

The amount of noise you add should not significantly affect the appearance of the plot, it should simply serve to separate overlapping observations.

Hint: Use `np.random.normal` with the mean of 0 and a standard deviation of 0.1, and an appropriate value for the `size` parameter. Note that you want to *add a different value to each element* stored in the `first_2_pcs`, but you should avoid using loops and instead use the `size` parameter.

In [28]:
```
normal_noise = np.random.normal(size=first_2_pcs.shape)
first_2_pcs_jittered = first_2_pcs + normal_noise
```

Question 2g: visualize the data using the first principal components

To label the points on a scatter plot of your **jittered data**, the best option is to use Altair's `mark_text()` and give it a list of state names as the `text` encoding.

Hint: You can get a list of the state names with `list(df_1972_to_2016.index)`.

Select **one option** from the two given below to create a scatter plot of your jittered data. (We recommend starting with Option 1 and getting it to work, and then, if you have time, come back to figure out Option 2.)

Option 1: Altair will let you zoom and pan around to look at the data if you add the `.interactive()` at the end of the chart definition.

Option 2: *Alternatively* (or "additionally", depending how curious you are), you can create a cool linked chart, which would automatically update the list of states whose names you have selected on the scatter plot.

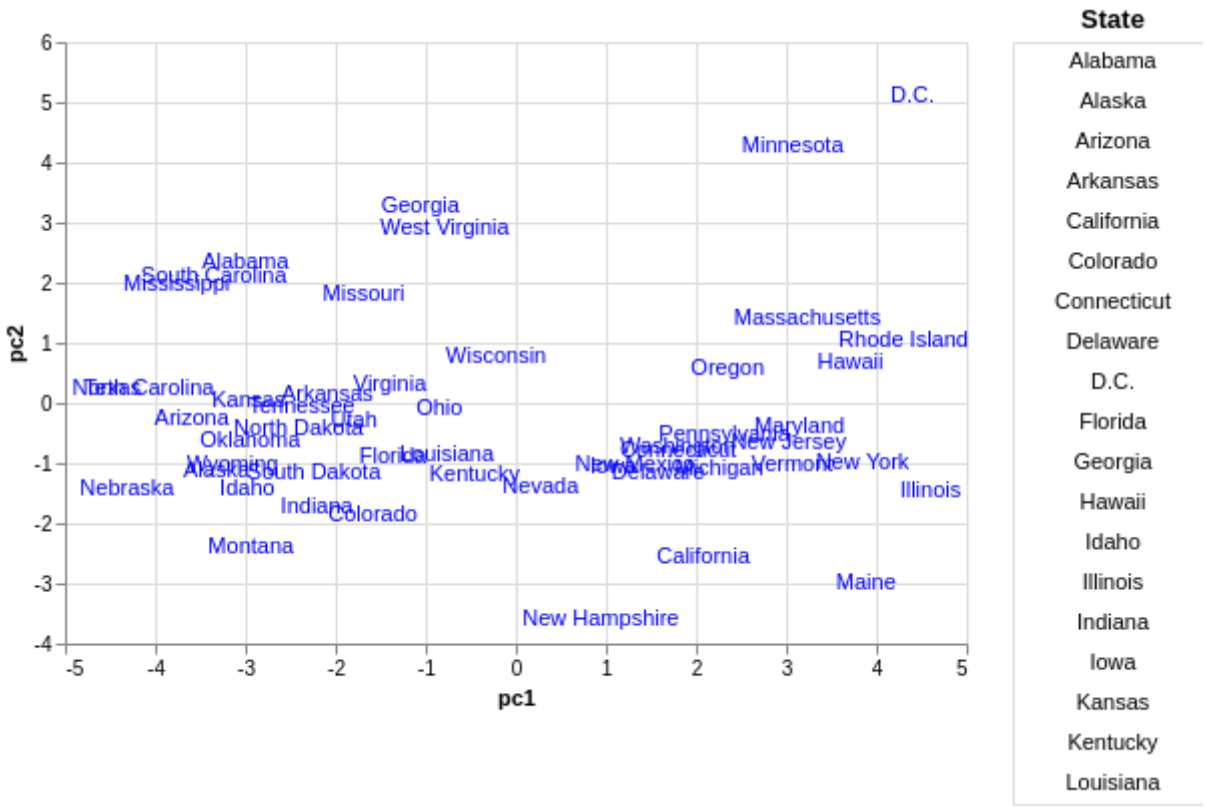
One important skill as a user of modern tools is using existing documentation and examples to get the plot you want. Using the example given on this page as a guide (https://altair-viz.github.io/gallery/scatter_linked_table.html), create a scatter plot of your jittered presidential election data.

Workflow productivity hint: Find a shortcut to quickly comment/uncomment a highlighted selection inside a code cell: at the top bar, go to `Help => Keyboard Shortcuts`, scroll to the section "Edit Mode" and find the shortcut for "comment" (note, it is the same shortcut to uncomment a selection).

In [29]:

```
state_names = list(df_1972_to_2016.index)
first_2_pcs_jittered['state'] = state_names

################################################

### OPTION 1
# alt.Chart(...).mark_text().encode(
#     x = ...,
#     y = ...,
#     text = ...
# ).properties(width=450)...

################################################
## OPTION 2

# Brush for selection
brush = alt.selection(type='interval')

labels = alt.Chart(first_2_pcs_jittered).mark_text().encode(
    x = "pc1",
    y = "pc2",
    text = "state",
    color=alt.condition(brush, alt.value('blue'), alt.value('grey'))
).properties(width=450).add_selection(brush)


# Base chart for data tables
ranked_text = alt.Chart(first_2_pcs_jittered).mark_text().encode(
    y = alt.Y('row_number:O',axis=None)
).transform_window(
    row_number='row_number()'
).transform_filter(
    brush
).transform_window(
    rank='rank(row_number)'
).transform_filter(
    alt.datum.rank<20
)

# Data Tables
origin = ranked_text.encode(text='state:N').properties(title='State', w
idth=100)

# Build chart
alt.hconcat(
    labels,
    origin
)
```

Out[29]:



Give an example of a cluster of states that vote a similar way (remember what the 0 and 1 originally stood for?). Does the composition of this cluster surprise you? If you're not familiar with U.S. politics, it's fine to just say '*No, I'm not surprised because I don't know anything about U.S. politics.*'.

No, I'm not surprised because I don't know anything about U.S. politics.

In the cell below, write down anything interesting that you observe by looking at this plot. You will get credit for this as long as you write something reasonable that you can take away from the plot.

Northern states are pretty well separated overall by Southern states. I don't think that should surprise us.

Question 2h: plot the columns' contributions to PCs

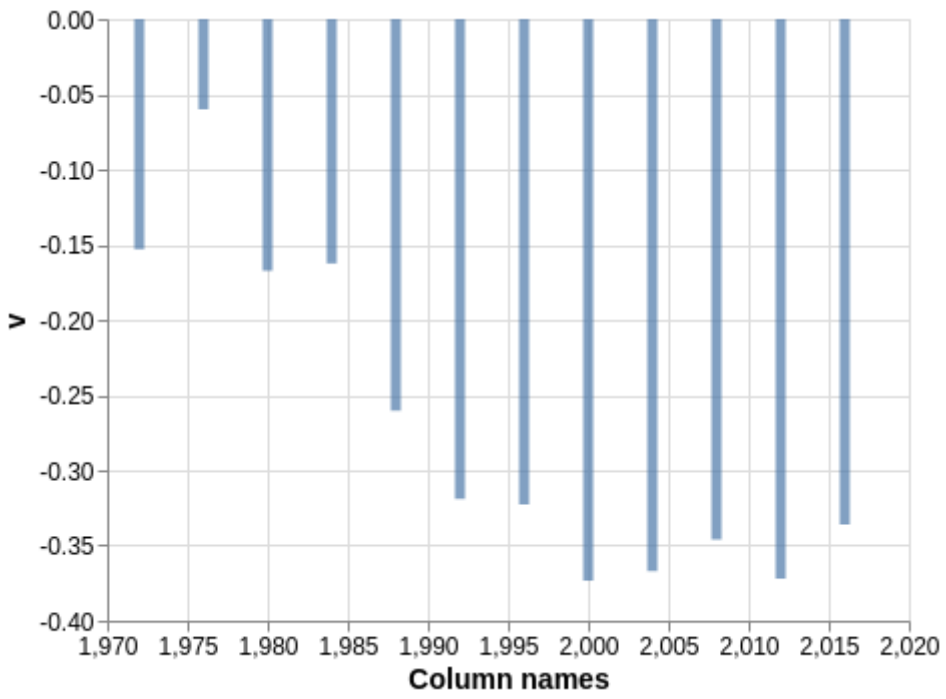
We can also look at the contributions of each year's elections results on the values for our principal components. Below, we will define and use the `plot_pc` function to plot the 1st row of V^T in the cell below.

Here by "1st row" we mean the row that is used to generate `pc1`, and by "2nd row" we mean the row that is used to generate `pc2`.

In [30]:
```
def plot_pc(col_names, vt, k):
    """
    Plot how much each column of our data contributes
    to each principal component and labels the rows of V^T.
    """
    df = pd.DataFrame({'v':vt[k, :], 'Column names':col_names})
    chart = alt.Chart(df).mark_bar().encode(
        x='Column names',
        y='v',
        opacity=alt.value(0.7)
    ).configure_axis(
        labelFontSize=12,
        titleFontSize=14
    ).configure_axisX(
        labelAngle = 0
    ).properties(width = 400)
    return chart
```

In [31]:
```
### If you get a ValueError: arrays must all be same length
### Go back to question 2e and read the note about NOT overwriting
### `u, s, vt` that you defined earlier.

plot_pc(list(df_1972_to_2016_num.columns), vt1, 0)
```

Out[31]:

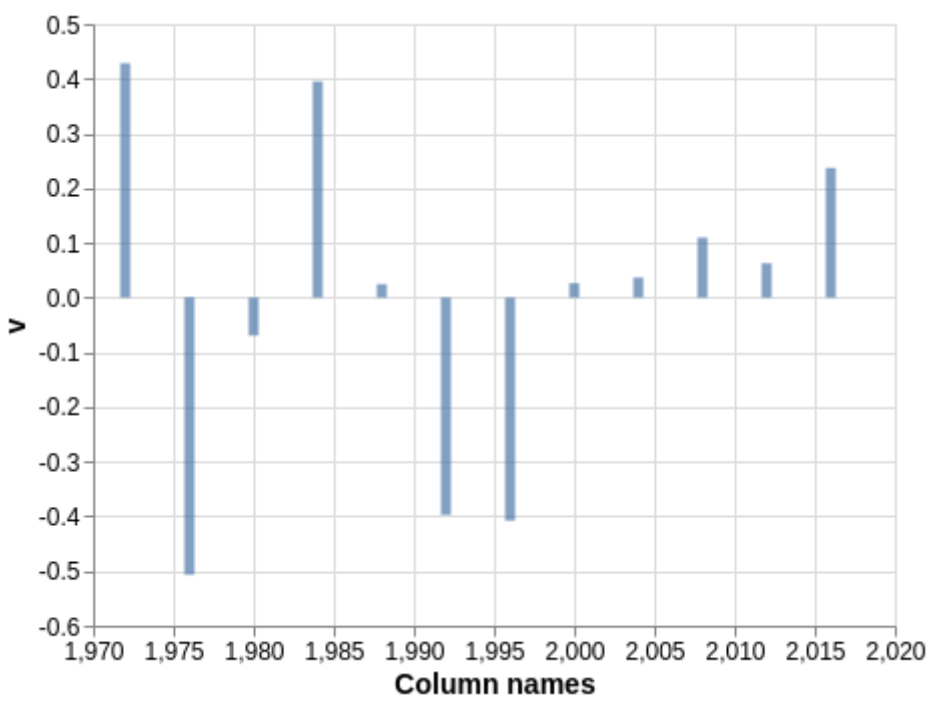


In the cell below, plot the the 2nd row of V^T.

Hint: You are just copying and pasting code from the cell above and then changing one number.

In [32]:
```
plot_pc(list(df_1972_to_2016_num.columns), vt1, 2)
```

Out[32]:



Question 2i

Using your plots from Question 2h as well as the original table, give a description of what it means to have a relatively large positive value for `pc1` (right side of the 2D scatter plot), and what it means to have a relatively large positive value for `pc2` (top side of the 2D scatter plot).

In other words, what is generally true about a state with relatively large positive value for `pc1` ? For a large positive value for `pc2` ?

Note: `pc2` is pretty hard to interpret, and we don't really have a concensus on what it means either. We'll be nice when grading: we just want to see your best attempt at an explanation.

Note: Principal components beyond the first are often hard to interpret (but not always, see question 1 earlier in this homework).

SOLUTION: high value for pc1 means information encoded by the specific feature can explain the variance observed in the dataset well. they significantly contribute to the relationships/patterns observed in the dataset in a direct way.

pc2 could just be equivalent to pc1 but encoding much less information. pc2 can also be linked to the patterns linked to the data by a second order statistic.

In [33]:
```
# feel free to use this cell for scratch work.
# If you need more scratch space, add cells *below* this one.

# Make sure to put your actual answer in the cell ABOVE, next to the wo
rd SOLUTION:
```

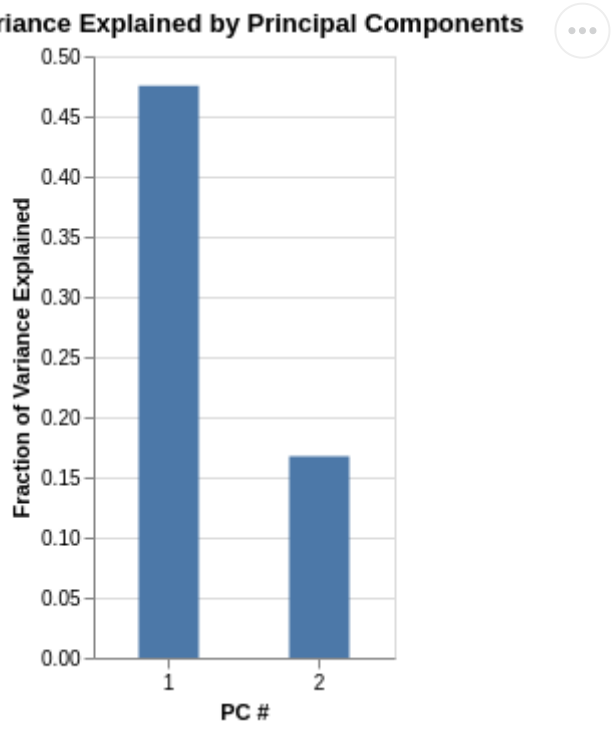
Question 2j

To get a better sense of whether our 2D scatterplot captures the whole story, create a scree plot for this data. On the y-axis plot the fraction of the total variance captured by the ith principal component. You should see that the first two principal components capture quite a bit of the variance. It is partially for this reason that the 2D scatter plot was so much more useful for this dataset.

Hint: Your code will be very similar to the scree plot from problem 1d. Be sure to label your axes appropriately!

In [34]:
```
explained_var = pd.DataFrame({
    'PC #': [1, 2],
    'Fraction of Variance Explained' : [(s1[0]**2/(sum(s1**2))), (s1[1]
**2/(sum(s1**2)))]})

# Draw your Altair visualization
alt.Chart(explained_var,
          title="Variance Explained by Principal Components"
).mark_bar(size=30).encode(
    alt.X('PC #:O'),
    alt.Y('Fraction of Variance Explained:Q')
).configure_axisX(labelAngle=0).properties(width=150)
```

Out[34]:



Congratulations! You are finished with this homework on Principal Component Analysis.

Running Built-in Tests

1. All tests are in `tests` directory
2. Each python file in `tests` is a test
3. `grader.check('testname')` runs test `'testname'`, e.g. `'q1'`
4. `grader.check_all()` runs all visible tests

In [35]:
```
# Run built-in checks
grader.check_all()
```

q1a

All tests passed!

q1b

All tests passed!

q1c

All tests passed!

q1d

All tests passed!

q1e

All tests passed!

q2a

All tests passed!

q2b

All tests passed!

q2c

All tests passed!

q2d

0 of 1 tests passed

Tests failed:

- **./tests/q2d.py**

  Test code:

```
>>> round(df_1972_to_2016_aligned.sum(axis=1)["D.C."], 5) < round(d
f_1972_to_2016_centered.sum(axis=1)["D.C."], 5)
True
```

  Test result:

```
Trying:
    round(df_1972_to_2016_aligned.sum(axis=1)["D.C."], 5) < round(d
f_1972_to_2016_centered.sum(axis=1)["D.C."], 5)
Expecting:
    True
**********************************************************************
***
Line 1, in ./tests/q2d.py 0
Failed example:
    round(df_1972_to_2016_aligned.sum(axis=1)["D.C."], 5) < round(d
f_1972_to_2016_centered.sum(axis=1)["D.C."], 5)
Exception raised:
    Traceback (most recent call last):
      File "/opt/conda/lib/python3.7/doctest.py", line 1329, in __r
un
        compileflags, 1), test.globs)
      File "", line 1, in
        round(df_1972_to_2016_aligned.sum(axis=1)["D.C."], 5) < rou
nd(df_1972_to_2016_centered.sum(axis=1)["D.C."], 5)
    NameError: name 'df_1972_to_2016_aligned' is not defined
```

q2e

All tests passed!

In [ ]:

```
# Generate pdf in classic notebook (does not work in JupyterLab)
import nb2pdf
nb2pdf.convert('hw4.ipynb')

# To generate pdf using command-line, run in terminal,
# nb2pdf hw4.ipynb
```

Submission Checklist

1. Check filename is 'hw4.ipynb'
2. Save file to confirm all changes are on disk
3. Run *Kernel* > *Restart & Run All* to execute all code from top to bottom
4. Check `grader.check_all()` output
5. Save file again to write any new output to disk
6. Check generated pdf that all responses are displayed correctly
7. Submit to Gradescope