In [55]:
```
# Initialize Otter
import otter
grader = otter.Notebook("lab5-pca.ipynb")
```

In [56]:
```
import numpy as np
import pandas as pd
import altair as alt
from sklearn.decomposition import PCA
alt.data_transformers.disable_max_rows()
```

Out[56]:
```
DataTransformerRegistry.enable('default')
```

In [3]:
```
Author: Neala Rashidfarrukhi
Collaborator: Eve McCreary
```

```
  Input In [3]
    Author: Neala Rashidfarrukhi
                  ^
SyntaxError: invalid syntax
```

Lab 5: Principal components

There are many perspectives on principal components analysis (PCA). PCA is variously described as: a dimension reduction method; a method of approximating covariance structure; a latent model; a change of basis that optimally describes covariation; and so on. How can these seemingly distinct views be compatible with a single method?

A simple answer is that PCA has a very wide range of applications in which it serves different purposes. Sometimes it is applied to find a few derived variables based on a large number of input variables -- hence, 'dimension reduction'. At others, it is used to interpret covariation among many variables -- hence, a 'covariance approximation'. With different objectives come different perspectives.

In PSTAT100, we'll try to look beyond this and focus on the core technique of PCA: *finding linear data transformations that preserve variation and covariation*. In short, we'll focus on the *principal components* (PC) part of PCA, taking the following view.

- **Principal components** are ***linear data transformations***.
- The **analysis** of principal components is ***varied depending on the application***.

We'll keep an open mind for the time being about what the analysis (A) part of PCA entails. So, what does it mean to say that 'principal components are linear data transformations'? Suppose you have a dataset with n observations and p variables. As a dataframe, this might look something like the following:

Observation	Variable 1	Variable 2	$\cdots$	Variable p
1	x_{11}	x_{12}	$\cdots$	x_{1p}
2	x_{21}	x_{22}	$\cdots$	x_{2p}
$\vdots$	$\vdots$	$\vdots$		$\vdots$
n	x_{n1}	x_{n2}	$\cdots$	x_{np}

We can represent the values as a data matrix $\mathbf{X}$ with n rows and p columns:

$$\mathbf{X} = \underbrace{\left[\begin{array}{cccc} \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_p \end{array}\right]}_{\text{column vectors}} = \left[\begin{array}{cccc} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \end{array}\right]$$

To say that the principal components are linear data transformations means that each principal component is of the form:

$$\text{PC} = \mathbf{X}\mathbf{v} = v_1\mathbf{x}_1 + v_2\mathbf{x}_2 + \cdots + v_p\mathbf{x}_p$$

In other words, a linear combination of the columns of the data matrix. In PCA, the linear combination coefficients are known as *loadings*; the PC loadings are found in a particular way using the correlations among the columns.

Objectives

In this lab, you'll focus on computing and interpreting principal components:

- finding the loadings (linear combination coefficients) for each PC;
- quantifying the variation captured by each PC;
- visualization-based techniques for selecting a number of PC's to A(nalyze);
- plotting and interpreting loadings.

In addition, you'll encounter a few ways that PCA is useful in exploratory analysis:

- describing variation and covariation;
- identifying variables that 'drive' variation and covariation;
- visualizing multivariate data.

You'll work with a selection of county summaries from the 2010 U.S. census. The first few rows of the dataset are shown below:

In [9]:
```
# import tidy county-level 2010 census data
census = pd.read_csv('data/census2010.csv', encoding = 'latin1')
census.head()
```

Out[9]:

	State	County	Women	White	Citizen	IncomePerCap	Poverty	ChildPoverty	Professional	Service	...	Transit	OtherTransp	WorkAtHome	MeanCommute	Employed	PrivateWork	SelfEmpl
0	Alabama	Autauga	51.567339	75.788227	73.749117	24974.49970	12.912305	18.707580	32.790974	17.170444	...	0.095259	1.305969	1.835653	26.500165	43.436374	73.736490	5.43
1	Alabama	Baldwin	51.151337	83.102616	75.694057	27316.83516	13.424230	19.484305	32.729943	17.950921	...	0.126621	1.443800	3.850477	26.322179	44.051127	81.282655	5.90
2	Alabama	Barbour	46.171840	46.231594	76.912223	16824.21643	26.505629	43.559621	26.124042	16.463434	...	0.495403	1.621725	1.501946	24.518283	31.921135	71.594256	7.14
3	Alabama	Bibb	46.589099	74.499889	77.397806	18430.99031	16.603747	27.197085	21.590099	17.955450	...	0.503137	1.562095	0.731468	28.714391	36.692621	76.743846	6.63
4	Alabama	Blount	50.594351	87.853854	73.375498	20532.27467	16.721518	26.857377	28.529302	13.942519	...	0.362632	0.419941	2.265413	34.844893	38.449142	81.826708	4.22

5 rows × 24 columns

The observational units are U.S. counties, and each row (observation) is a county. The values are, for the most part, percentages of the county population. You can find variable descriptions in the metadata file in the data directory (*data > census2010metadata.csv*).

---

0. Correlations

PCA identifies variable combinations that capture covariation by decomposing the correlation matrix. So, to start with, let's examine the correlation matrix for the 2010 county-level census data to get a sense of which variables tend to vary together.

The correlation matrix is a matrix of all pairwise correlations between variables. If x_ij denotes the value for the ith observation of variable j, then the entry at row j and column k of the correlation matrix $\mathbf{R}$ is:

$$r_{jk} = \frac{\sum_i (x_{ij} - \bar{x}_j)(x_{ik} - \bar{x}_k)}{S_j S_k}$$

In the census data, the `State` and `County` columns indicate the geographic region for each observation; essentially, they are a row index. So we'll drop them before computing the matrix $\mathbf{R}$:

In [10]:
```
# store quantitative variables separately
x_mx = census.drop(columns = ['State', 'County'])
```

From here, the matrix is simple to compute in pandas using `.corr()`:

In [11]:
```
# correlation matrix
corr_mx = x_mx.corr()
```

The matrix can be inspected directly to determine which variables vary together. For example, we could look at the correlations of the percentage of the population that is employed with all other variables in the dataset by extracting the `Employed` column:

In [12]:
```
# correlation between poverty and other variables
corr_mx.loc[:, 'Employed'].sort_values()
```

Out[12]:
```
ChildPoverty   -0.744510
Poverty        -0.735569
Unemployment   -0.697985
Minority       -0.439053
Service        -0.403261
MeanCommute    -0.252111
Drive          -0.215038
Carpool        -0.144336
Production     -0.136277
Citizen        -0.087343
Office         -0.014838
OtherTransp    -0.010041
FamilyWork      0.055654
Women           0.131181
Transit         0.151700
SelfEmployed    0.154107
PrivateWork     0.264826
WorkAtHome      0.303839
White           0.432856
Professional    0.473413
IncomePerCap    0.767001
Employed        1.000000
Name: Employed, dtype: float64
```

Recall that correlation is a number in the interval [-1, 1] whose magnitude indicates the strength of the relationship between variables.

- Correlations near -1 are *strongly negative*, and mean that the variables *tend to vary in opposition*
  - (large values of one coincide with small values of the other and vice-versa).
- Correlations near 1 are *strongly positive*, and mean that the variables *tend to vary together*
  - (large values coincide and small values coincide).

As a result, from examining these entries, it can be seen that the percentage of the county population that is employed is:

- strongly *negatively* correlated with child poverty, poverty, and unemployment, meaning it *tends to vary in opposition* with these variables;
- strongly *positively* correlated with income per capita, meaning it *tends to vary together* with this variable.

If instead we wanted to look up the correlation between just two variables, we could retrieve the relevant entry directly using `.loc[...]`:

In [13]:
```
# correlation between employment and income per capita
corr_mx.loc['Employed', 'IncomePerCap']
```

Out[13]:
```
0.7670009685702536
```

So across U.S. counties employment is, perhaps unsurprisingly, strongly and positively correlated with income per capita, meaning that higher employment rates tend to coincide with higher incomes per capita.

Question 0 (a)

Check your understanding by repeating this for a different pair of variables.

(i) Find the correlation between the poverty rate and demographic minority rate and store it in `pov_dem_rate`.

In [14]:
```
# correlation between poverty and percent minority
pov_dem_rate = corr_mx.loc['Poverty','Minority']
# print
pov_dem_rate
```

Out[14]:
```
0.6231625196890354
```

In [15]:
```
grader.check("q0_a_i")
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Input In [15], in <cell line: 1>()
----> 1 grader.check("q0_a_i")

File /opt/conda/lib/python3.9/site-packages/otter/check/utils.py:131, in logs_event.<locals>.event_logger.<locals>.run_function(self, *args, **kwargs)
    129 except Exception as e:
    130     self._log_event(event_type, success=False, error=e)
--> 131     raise e
    132 else:
    133     self._log_event(event_type, results=results, question=question, shelve_env=shelve_env)

File /opt/conda/lib/python3.9/site-packages/otter/check/utils.py:124, in logs_event.<locals>.event_logger.<locals>.run_function(self, *args, **kwargs)
    122 try:
    123     if event_type == EventType.CHECK:
--> 124         question, results, shelve_env = f(self, *args, **kwargs)
    125     else:
    126         results = f(self, *args, **kwargs)

File /opt/conda/lib/python3.9/site-packages/otter/check/notebook.py:179, in Notebook.check(self, question, global_env)
    176     global_env = inspect.currentframe().f_back.f_back.f_globals
    178 # run the check
--> 179 result = check(test_path, test_name, global_env)
    181 return question, result, global_env

File /opt/conda/lib/python3.9/site-packages/otter/execute/__init__.py:46, in check(nb_or_test_path, test_name, global_env)
     44     test = OKTestFile.from_file(nb_or_test_path)
     45 else:
---> 46     test = NotebookMetadataOKTestFile.from_file(nb_or_test_path, test_name)
     48 if global_env is None:
     49     # Get the global env of our callers - one level below us in the stack
     50     # The grade method should only be called directly from user / notebook
     51     # code. If some other method is calling it, it should also use the
     52     # inspect trick to pass in its parents' global env.
     53     global_env = inspect.currentframe().f_back.f_globals

File /opt/conda/lib/python3.9/site-packages/otter/test_files/metadata_test.py:76, in NotebookMetadataOKTestFile.from_file(cls, path, test_name)
     73 with open(path, encoding="utf-8") as f:
     74     nb = json.load(f)
---> 76 test_spec = nb["metadata"][NOTEBOOK_METADATA_KEY]["tests"]
     77 if test_name not in test_spec:
     78     raise ValueError(f"Test {test_name} not found")

KeyError: 'otter'
```

(ii) Interpret the correlation: is it large or small, positive or negative, and what does that mean?

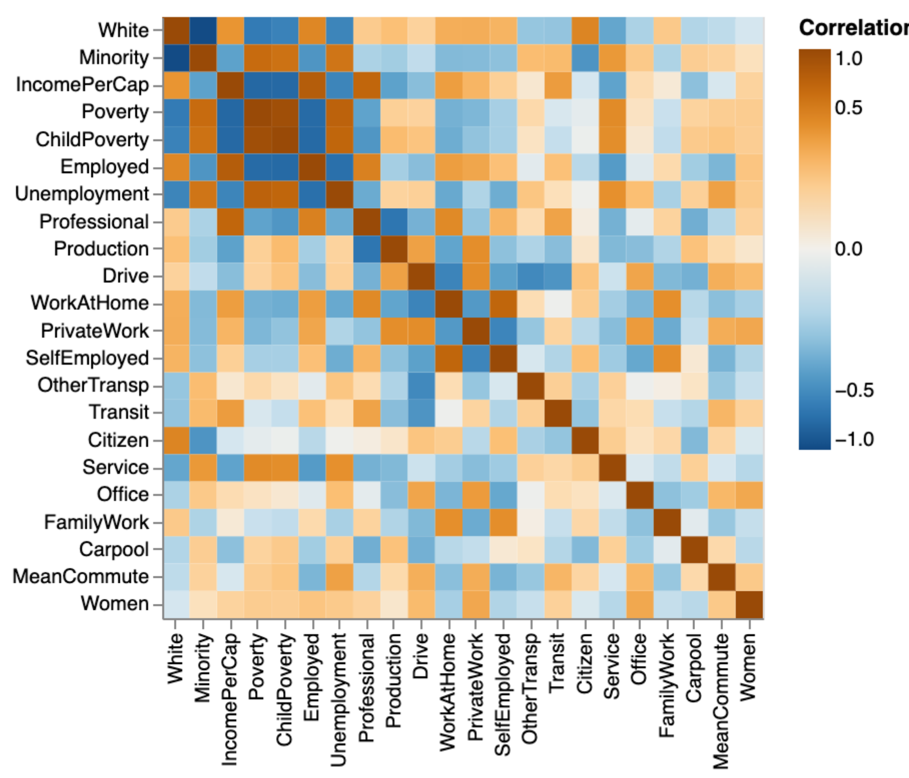
Type your answer here, replacing this text.

While direct inspection is useful, it can be cumbersome to check correlations for a large number of variables this way. A heatmap -- a colored image of the matrix -- provides a (sometimes) convenient way to see what's going on without having to examine the numerical values directly. The cell below shows one way of constructing this plot.

Notice that the color scale shows positive correlations in orange, negative ones in blue, strong correlations in dark tones, and weak correlations in light tones. This is known as a 'diverging color gradient', and should, as a rule of thumb, always be used for plots of this type.

In [16]:
```
# melt corr_mx
corr_mx_long = corr_mx.reset_index().rename(
    columns = {'index': 'row'}
).melt(
    id_vars = 'row',
    var_name = 'col',
    value_name = 'Correlation'
)

# construct plot
alt.Chart(corr_mx_long).mark_rect().encode(
    x = alt.X('col', title = '', sort = {'field': 'Correlation', 'order': 'ascending'}),
    y = alt.Y('row', title = '', sort = {'field': 'Correlation', 'order': 'ascending'}),
    color = alt.Color('Correlation',
                      scale = alt.Scale(scheme = 'blueorange', # diverging gradient
                                        domain = (-1, 1), # ensure white = 0
                                        type = 'sqrt'), # adjust gradient scale
                     legend = alt.Legend(tickCount = 5)) # add ticks to colorbar at 0.5 for reference
).properties(width = 300, height = 300)
```

Out[16]:



Question 0 (b)

Which variable is self employment rate most *positively* correlated with? Refer to the heatmap.

Answer

The Variable self employment rate corresponds with work at home.

---

1. Principal components analysis

Principal components analysis (PCA) consists in finding variable combinations that capture large portions of the variation and covariation in one's dataset.

'Variable combinations' here means *linear* combinations. That is, if again x_{ij} denotes the data value for the ith observation and the jth variable, the value of a principal component is of the form:

$$\text{PC}_i = \sum_j w_j x_{ij} \quad (\text{value of PC for observation } i)$$

The weights w_j for each variable are called the *loadings*. The loadings tell which variables are most influential (heavily weighted) in each component, and thus offer an indirect picture of which variables are driving variation and covariation in the original data.

Here we'll look at how to:

- compute the full set of principal components;
- determine the variation they capture;
- select a subset of principal components for analysis;
- and examine the loadings.

The data should be normalized before carrying out PCA. (You'll see why a little later.)

In [17]:
```
# center and scale ('normalize')
x_ctr = (x_mx - x_mx.mean())/x_mx.std()
```

Computing PC loadings

In `sklearn`, the module `PCA(...)` computes principal components, the proportion of variance captured by each one, and the loadings of each one. The syntax may be a bit different than what you're used to. First we'll configure the module with a fixed number of components to match the number of variables in the dataset and store the result under a separate name.

In [18]:
```
# compute principal components
pca = PCA(n_components = x_ctr.shape[1])
pca.fit(x_ctr)
```

Out[18]:
```
PCA(n_components=22)
```

Most quantities you might want to use in PCA can be retrieved as attributes of `pca` after `pca.fit(...)` has been run. In particular:

- `.components_` contains the loadings of the principal components;
- `.explained_variance_ratio_` contains the proportion of variation and covariation captured by each principal component.

You might find it worthwhile to open up the PCA documentation and keep the 'Attributes' section visible as you're working through the remainder of this part.

Selecting the number of PCs

The basic strategy for selecting a number of principal components to work with is to determine how many are needed to capture a large portion of variation and covariation in the original data. This can be done graphically by plotting the variance ratios.

Let's start by retrieving the variance ratios for each component. These are stored as the `.explained_variance_ratio_` attribute of `pca`:

In [19]:
```
# variance ratios
pca.explained_variance_ratio_
```

Out[19]:
```
array([2.62856309e-01, 1.51574359e-01, 1.14128491e-01, 7.66651854e-02,
       5.43445926e-02, 5.15406137e-02, 4.73175702e-02, 4.02078608e-02,
       3.66870768e-02, 3.36410429e-02, 2.63259728e-02, 2.20184669e-02,
       1.75961200e-02, 1.68408338e-02, 1.42824011e-02, 1.02390948e-02,
       7.83380242e-03, 6.30665716e-03, 4.71870613e-03, 2.66195282e-03,
       2.10073405e-03, 1.12156844e-04])
```

Notice that the components are sorted in descending order of variance ratio -- that means that the first component always captures the most variation and covariation, the second component always captures the secondmost, and so on. For plotting purposes, it will be helpful to store these in a dataframe:

In [20]:
```
# store proportion of variance explained as a dataframe
pca_var_explained = pd.DataFrame({'Proportion of variance explained': pca.explained_variance_ratio_})

# add component number as a new column
pca_var_explained['Component'] = np.arange(1, 23)

# print
pca_var_explained.head()
```

Out[20]:

	Proportion of variance explained	Component
0	0.262856	1
1	0.151574	2
2	0.114128	3
3	0.076665	4
4	0.054345	5

These values report the proportion of variance explained *individually* by each component; it is also useful to show the proportion of variance explained *collectively* by a set of components.

Question 1 (a)

Add a column to `pca_var_explained` called `Cumulative variance explained` that contains the cumulative sum of the proportion of variance explained. For the first component, this new variable should be equal to the value of `Proportion of variance explained`; for the second component, it should be equal to the sum of the values of `Proportion of variance explained` for components 1 and 2; for the third, to the sum of values for components 1, 2, and 3; and so on.

Print the first few rows.

(*Hint*: use `cumsum(...)` with an appropriate axis specification.)

In [21]:
```
# add cumulative variance explained as a new column
pca_var_explained['Cumulative variance explained']= pd.DataFrame({'Proportion of variance explained': pca.explained_variance_ratio_}).cumsum()
# print
pca_var_explained
```

Out[21]:

	Proportion of variance explained	Component	Cumulative variance explained
0	0.262856	1	0.262856
1	0.151574	2	0.414431
2	0.114128	3	0.528559
3	0.076665	4	0.605224
4	0.054345	5	0.659569
5	0.051541	6	0.711110
6	0.047318	7	0.758427
7	0.040208	8	0.798635
8	0.036687	9	0.835322
9	0.033641	10	0.868963
10	0.026326	11	0.895289
11	0.022018	12	0.917308
12	0.017596	13	0.934904
13	0.016841	14	0.951744
14	0.014282	15	0.966027
15	0.010239	16	0.976266
16	0.007834	17	0.984100
17	0.006307	18	0.990406
18	0.004719	19	0.995125
19	0.002662	20	0.997787
20	0.002101	21	0.999888
21	0.000112	22	1.000000

In [22]:
```
grader.check("q1_a")
```

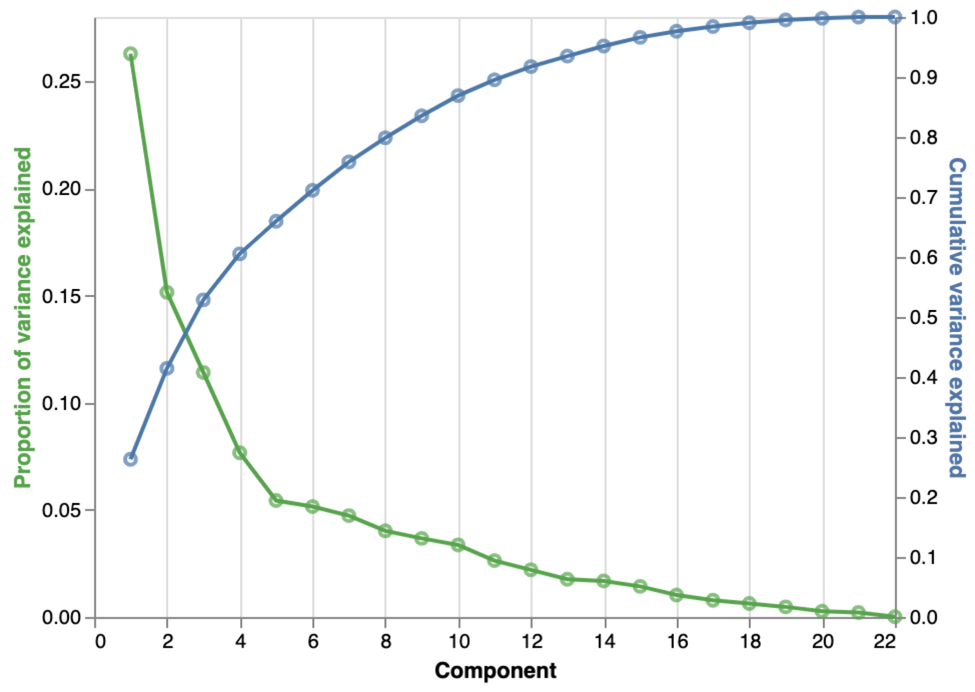
Out[22]:

q1_a passed!

Now we'll make a dual-axis plot showing, on one side, the proportion of variance explained (y) as a function of component (x), and on the other side, the cumulative variance explained (y) also as a function of component (x). Make sure that you've completed Q1(a) before running the next cell.

In [23]:
```
# encode component axis only as base layer
base = alt.Chart(pca_var_explained).encode(
    x = 'Component')

# make a base layer for the proportion of variance explained
prop_var_base = base.encode(
    y = alt.Y('Proportion of variance explained',
              axis = alt.Axis(titleColor = '#57A44C'))
)

# make a base layer for the cumulative variance explained
cum_var_base = base.encode(
    y = alt.Y('Cumulative variance explained', axis = alt.Axis(titleColor = '#5276A7'))
)

# add points and lines to each base layer
prop_var = prop_var_base.mark_line(stroke = '#57A44C') + prop_var_base.mark_point(color = '#57A44C')
cum_var = cum_var_base.mark_line() + cum_var_base.mark_point()

# layer the layers
var_explained_plot = alt.layer(prop_var, cum_var).resolve_scale(y = 'independent')

# display
var_explained_plot
```

Out[23]:



The purpose of making this plot is to quickly determine the fewest number of principal components that capture a considerable portion of variation and covariation. 'Considerable' here is a bit subjective.

In this case, we'll base that decision on the proportion of variance explained (left axis) rather than the cumulative variance explained. Notice that there are diminishing gains after a certain number of components, in the sense that adjacent components explain similar proportions of variation. Sometimes it's said that there's an 'elbow' in the plot to describe this phenomenon.

Question 1 (b)

Using the graph and table above, how many principal components explain more than 6% of total variation (variation and covariation) individually? Store this in `main_pca`.

In [24]:
```
main_pca = sum(pca_var_explained['Proportion of variance explained'] > 0.06)
#print
main_pca
```

Out[24]: 4

In [25]:
```
grader.check("q1_b")
```

Out[25]:

q1_b passed!

Question 1 (c)

How much total variation is captured collectively by the number of components you stated above? Store this exact proportion in `main_variation`.

In [26]:
```
main_variation = pca_var_explained['Cumulative variance explained'][main_pca]
#print
main_variation
```

Out[26]: 0.6595689368294221

In [27]:
```
grader.check("q1_c")
```

Out[27]:

q1_c passed!

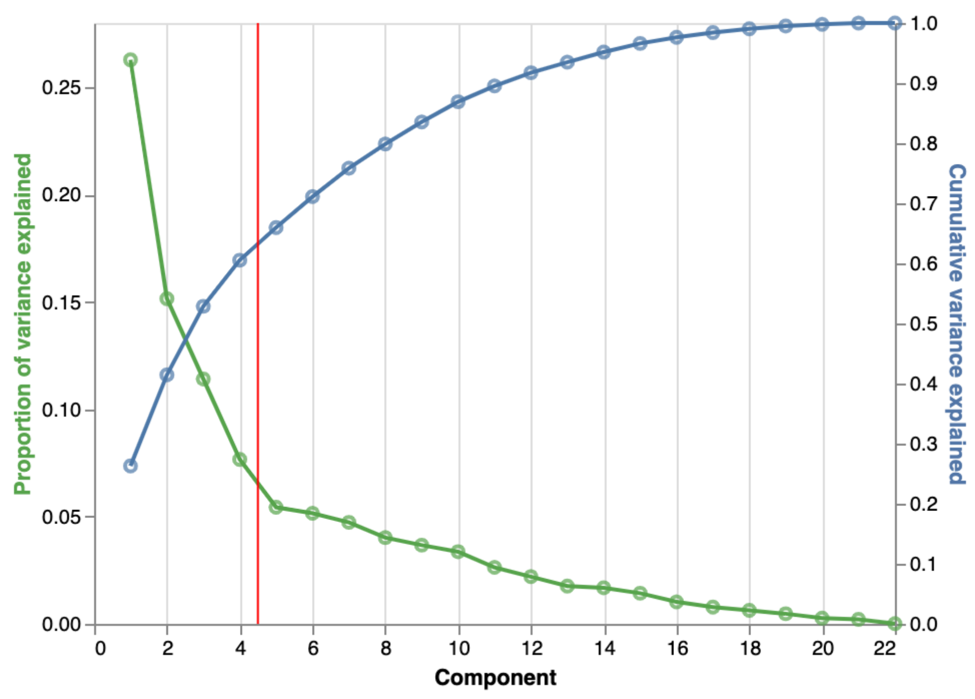
Question 1 (d)

Indicate your selected number of components (answer in Q1(b)) by adding a vertical line to the plot above. Instead of placing the line directly on your selected number of components, put it at the midpoint between your selected number and the next-largest number. Choose a color of your liking for the line. If you're not sure where to start, have a look at the week 5 lecture codes.

(*Hint*: in order to make this work in Altair, you'll need to layer the line on to either `prop_var` or `cum_var` *before* calling `alt.layer(...)`; if you try to add the line as a layer to `var_explained_plot`, Altair will throw an error.)

In [28]:
```
# add vertical line indicating number of selected pcs
line = alt.Chart(pd.DataFrame({'Component':[4.5]})).mark_rule(color = 'red').encode( x='Component')
# add line to one layer
cum_var = cum_var_base.mark_line() + cum_var_base.mark_point() + line # layer the layers
var_explained = alt.layer(prop_var, cum_var).resolve_scale(y = 'independent')
# display
var_explained
```

Out[28]:



Plotting and interpreting loadings

Now that you've chosen the number of components to work with, the next step is to examine loadings to understand just *which* variables the components combine with significant weight.

The loadings are stored as the `.components_` attribute of `pca` as an array of lists:

In [29]:
```
# loadings for first two pcs
pca.components_[0:2]
```

Out[29]:
```
array([[ 0.0200547 , -0.28961352, -0.05069832, -0.33486321,  0.36521186,
         0.36483606, -0.24013945,  0.2032542 ,  0.05216762,  0.09430653,
         0.1021974 ,  0.0791294 , -0.03023258,  0.02187119, -0.21835338,
         0.09700319, -0.3455879 , -0.03553872, -0.15530037, -0.08507737,
         0.33342029,  0.29246071],
       [-0.13995796, -0.19654947, -0.06499401, -0.02043167,  0.12017151,
         0.0810864 ,  0.17561108,  0.13971356, -0.189803  , -0.2823294 ,
        -0.40613046,  0.06374361,  0.10114242,  0.20940286,  0.33163575,
        -0.17673925, -0.05465314, -0.44192238,  0.3161738 ,  0.22113731,
         0.04304656,  0.19162849]])
```

As with the variance ratios, these will be more useful to us in a dataframe.

Question 1 (e)

Modify the code cell below to rename and select the loadings for the number of components you chose above.

In [30]:
```
# store the loadings as a data frame with appropriate names
loading_df = pd.DataFrame(pca.components_).transpose().rename(
columns = {0: 'PC1', 1:'PC2',2:'PC3',3:'PC4'} # add entries for each selected component
).loc[:, ['PC1','PC2','PC3','PC4']] # slice just components of interest
# add a column with the variable names
loading_df['Variable'] = x_mx.columns # print
loading_df
```

Out[30]:

	PC1	PC2	PC3	PC4	Variable
0	0.020055	-0.139958	0.187600	-0.176614	Women
1	-0.289614	-0.196549	-0.288902	-0.078059	White
2	-0.050698	-0.064994	-0.281904	-0.467986	Citizen
3	-0.334863	-0.020432	0.284074	-0.022197	IncomePerCap
4	0.365212	0.120172	-0.040170	-0.128231	Poverty
5	0.364836	0.081086	-0.077433	-0.098585	ChildPoverty
6	-0.240139	0.175611	0.287636	-0.258789	Professional
7	0.203254	0.139714	0.005957	-0.122145	Service
8	0.052168	-0.189803	0.281398	-0.267195	Office
9	0.094307	-0.282329	-0.285500	0.355106	Production
10	0.102197	-0.406130	-0.099229	-0.261077	Drive
11	0.079129	0.063744	-0.095696	0.457962	Carpool
12	-0.030233	0.101142	0.390869	0.052245	Transit
13	0.021871	0.209403	0.139315	0.221098	OtherTransp
14	-0.218353	0.331636	-0.116068	-0.113166	WorkAtHome
15	0.097003	-0.176739	0.135322	-0.144408	MeanCommute
16	-0.345588	-0.054653	0.157726	0.128709	Employed
17	-0.035539	-0.441922	0.158709	0.146725	PrivateWork
18	-0.155300	0.316174	-0.266798	-0.104453	SelfEmployed
19	-0.085077	0.221137	-0.203301	-0.064817	FamilyWork
20	0.333420	0.043047	0.069938	-0.125235	Unemployment
21	0.292461	0.191628	0.282231	0.074901	Minority

In [31]:
```
grader.check("q1_e")
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Input In [31], in <cell line: 1>()
----> 1 grader.check("q1_e")

File /opt/conda/lib/python3.9/site-packages/otter/check/utils.py:131, in logs_event.<locals>.event_logger.<locals>.run_function(self, *args, **kwargs)
    129 except Exception as e:
    130     self._log_event(event_type, success=False, error=e)
--> 131     raise e
    132 else:
    133     self._log_event(event_type, results=results, question=question, shelve_env=shelve_env)

File /opt/conda/lib/python3.9/site-packages/otter/check/utils.py:124, in logs_event.<locals>.event_logger.<locals>.run_function(self, *args, **kwargs)
    122 try:
    123     if event_type == EventType.CHECK:
--> 124         question, results, shelve_env = f(self, *args, **kwargs)
    125     else:
    126         results = f(self, *args, **kwargs)

File /opt/conda/lib/python3.9/site-packages/otter/check/notebook.py:179, in Notebook.check(self, question, global_env)
    176     global_env = inspect.currentframe().f_back.f_back.f_globals
    178 # run the check
--> 179 result = check(test_path, test_name, global_env)
    181 return question, result, global_env

File /opt/conda/lib/python3.9/site-packages/otter/execute/__init__.py:46, in check(nb_or_test_path, test_name, global_env)
     44     test = OKTestFile.from_file(nb_or_test_path)
     45 else:
---> 46     test = NotebookMetadataOKTestFile.from_file(nb_or_test_path, test_name)
     48 if global_env is None:
     49     # Get the global env of our callers - one level below us in the stack
     50     # The grade method should only be called directly from user / notebook
     51     # code. If some other method is calling it, it should also use the
     52     # inspect trick to pass in its parents' global env.
     53     global_env = inspect.currentframe().f_back.f_globals

File /opt/conda/lib/python3.9/site-packages/otter/test_files/metadata_test.py:76, in NotebookMetadataOKTestFile.from_file(cls, path, test_name)
     73 with open(path, encoding="utf-8") as f:
     74     nb = json.load(f)
---> 76 test_spec = nb["metadata"][NOTEBOOK_METADATA_KEY]["tests"]
     77 if test_name not in test_spec:
     78     raise ValueError(f"Test {test_name} not found")

KeyError: 'otter'
```

Again, the loadings are the *weights* with which the variables are combined to form the principal components. This is why the variable names have been appended as a separate column: each row is the weight for one variable in the dataset, and each column is a distinct set of weights.

For example, the `PC1` column tells us that this component is equal to:

$$(0.020055 \times \text{women}) + (-0.289614 \times \text{white}) + (-0.050698 \times \text{citizen}) + \dots$$

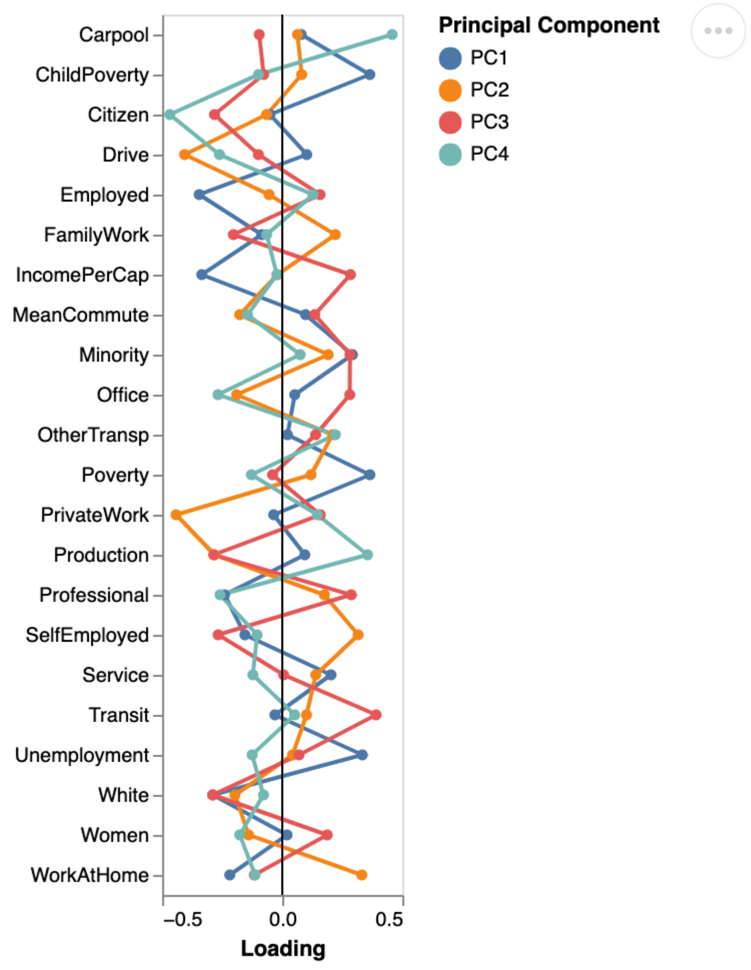
Since the components together capture over half the total variation, the heavily weighted variables in the selected components are the ones that drive variation in the original data. By visualizing the loadings, we can see which variables are most influential for each component, and thereby also which variables seem to drive total variation in the data.

Loadings are typically plotted against variable name as points connected by lines, as in the plot below. Make sure the previous question is complete before running this cell.

In [32]:
```
# melt from wide to long
loading_plot_df = loading_df.melt(
    id_vars = 'Variable',
    var_name = 'Principal Component',
    value_name = 'Loading'
)

# add a column of zeros to encode for x = 0 line to plot
loading_plot_df['zero'] = np.repeat(0, len(loading_plot_df))

# create base layer
base = alt.Chart(loading_plot_df)

# create lines + points for loadings
loadings = base.mark_line(point = True).encode(
    y = alt.X('Variable', title = ''),
    x = 'Loading',
    color = 'Principal Component'
)

# create line at zero
rule = base.mark_rule().encode(x = alt.X('zero', title = 'Loading'), size = alt.value(0.05))

# layer
```

```
loading_plot = (loadings + rule).properties(width = 120)

# show
loading_plot
```

Out[32]:



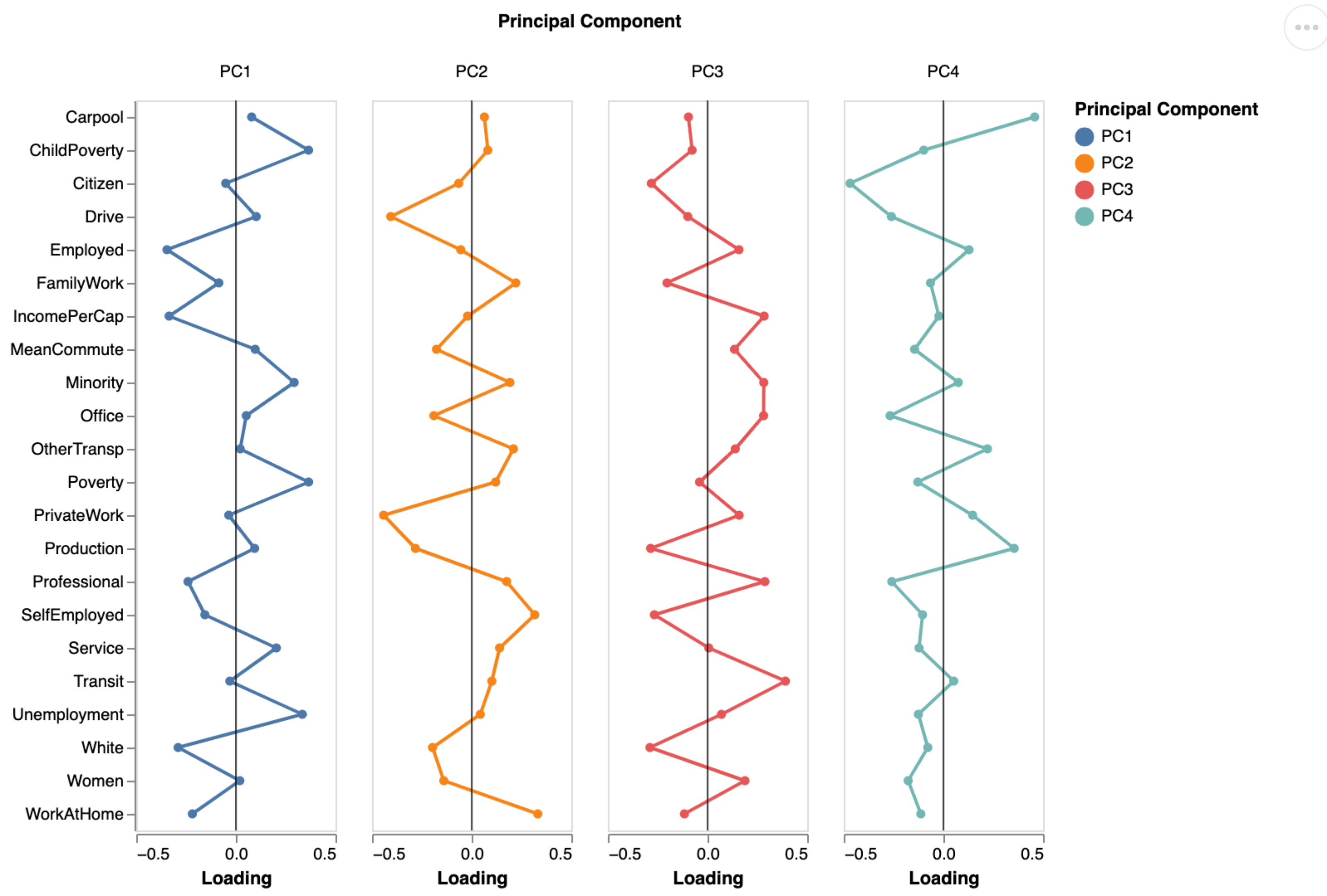
Question 1 (f)

The plot above is a bit crowded -- use `.facet(...)` to show each line separately. The resulting plot should have four adjacent panels, one for each PC.

(*Hint*: you can do this in one line by modifying `loading_plot`.)

In [33]:
```
# melt from wide to long
loading_plot.facet(column ='Principal Component')
```

Out[33]:



Great, but what do these plots have to say?

Look first at PC1: the variables with the largest loadings (points farthest in either direction from the zero line) are Child Poverty (positive), Employed (negative), Income per capita (negative), Poverty (positive), and Unemployment (positive). We know from exploring the correlation matrix that employment rate, unemployment rate, and income per capita are all related, and similarly child poverty rate and poverty rate are related. Therefore, the positively-loaded variables are all measuring more or less the same thing, and likewise for the negatively-loaded variables.

Essentially, then, PC1 is predominantly (but not entirely) a representation of income and poverty. In particular, counties have a higher value for PC1 if they have lower-than-average income per capita and higher-than-average poverty rates, and a smaller value for PC1 if they have higher-than-average income per capita and lower-than-average poverty rates.

Often interpreting principal components can be difficult, and sometimes there's no clear interpretation available! That said, it helps to have a system instead of staring at the plot and scratching our heads. Here is a semi-systematic approach to interpreting loadings:

1. Divert your attention away from the zero line.
2. Find the largest positive loading, and list all variables with similar loadings.
3. Find the largest negative loading, and list all variables with similar loadings.
4. The principal component represents the difference between the average of the first set and the average of the second set.
5. Try to come up with a description of less than 4 words.

This system is based on the following ideas:

- a high loading value (negative or positive) indicates that a variable strongly influences the principal component;
- a negative loading value indicates that
  - increases in the value of a variable *decrease* the value of the principal component
  - and decreases in the value of a variable *increase* the value of the principal component;
- a positive loading value indicates that
  - increases in the value of a variable *increase* the value of the principal component
  - and decreases in the value of a variable *decrease* the value of the principal component;
- similar loadings between two or more variables indicate that the principal component reflects their *average*;
- divergent loadings between two sets of variables indicates that the principal component reflects their *difference*.

Let's call PC1 'Income and poverty'. Here are my best stabs at the remaining ones.

PC2: Self employment. (High values come from high self employment + high work-at-home + low private sector workers.)

PC3: Urbanization. (High values come from high transit use + professional/office workers + commute + diversity + high income.)

PC4: Carpooling. (?)

You'll get some practice with this in HW3. For now, please take a moment to consider how I arrived at these interpretations by looking at the loading plots and thinking through the steps above.

Why normalize?

Data are typically normalized because without normalization, the variables on the largest scales tend to dominate the principal components, and most of the time PC1 will capture the majority of the variation.

However, that is artificial. In the census data, income per capita has the largest magnitudes, and thus, the highest variance.

In [34]:
```
# three largest variances
x_mx.var().sort_values(ascending = False).head(3)
```

Out[34]:
```
IncomePerCap    3.804072e+07
Minority        5.265263e+02
White           5.264985e+02
dtype: float64
```

When PCs are computed without normalization, the total variation is mostly just the variance of income per capita. But that's just because of the *scale* of the variable -- incomes per capita are large numbers -- not a reflection that it varies more or less than the other variables.

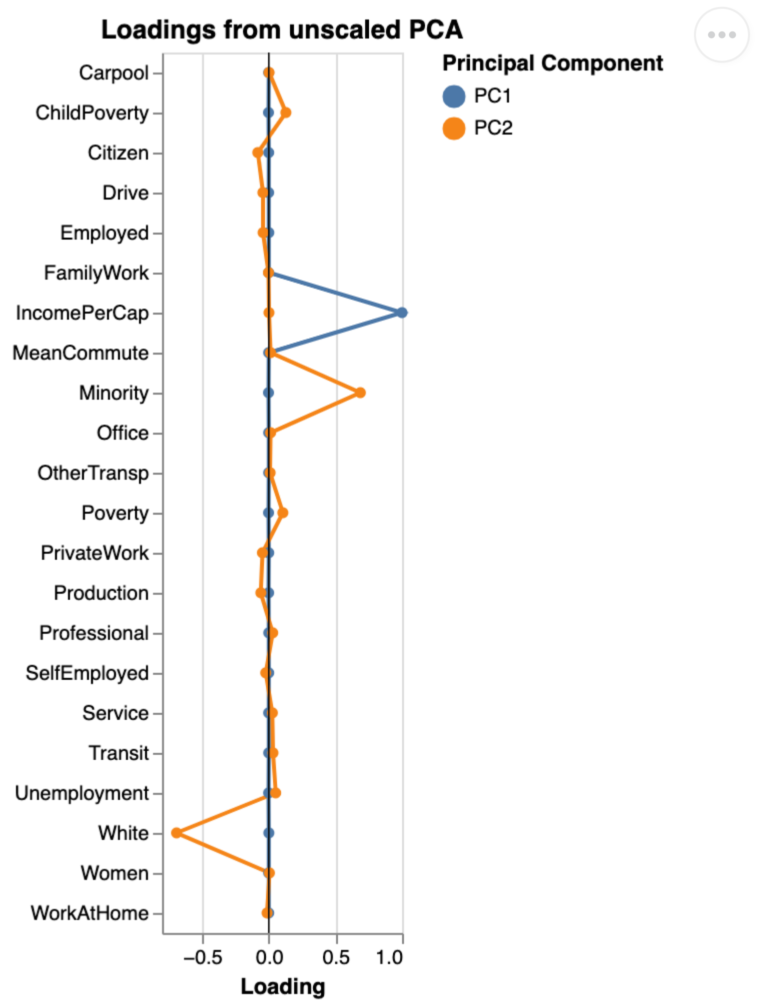
Run the cell below to see what happens to the loadings if the data are not normalized.

In [35]:
```
# recompute pcs without normalization
pca_unscaled = PCA(22)
pca_unscaled.fit(x_mx)

# show variance ratios for first three pcs
pd.Series(pca_unscaled.explained_variance_ratio_, index = range(1, 23)).head(3)
```

Out[35]:
```
1    0.999965
2    0.000025
3    0.000003
dtype: float64
```

In [36]:
```
# store the loadings as a data frame with appropriate names
unscaled_loading_df = pd.DataFrame(pca_unscaled.components_).transpose().rename(
    columns = {0: 'PC1', 1: 'PC2'} # add entries for each selected component
).loc[:, ['PC1', 'PC2']] # slice just components of interest

# add a column with the variable names
unscaled_loading_df['Variable'] = x_mx.columns.values

# melt from wide to long
unscaled_loading_plot_df = unscaled_loading_df.melt(
    id_vars = 'Variable',
    var_name = 'Principal Component',
    value_name = 'Loading'
)

# add a column of zeros to encode for x = 0 line to plot
unscaled_loading_plot_df['zero'] = np.repeat(0, len(unscaled_loading_plot_df))

# create base layer
base = alt.Chart(unscaled_loading_plot_df)

# create lines + points for loadings
loadings = base.mark_line(point = True).encode(
    y = alt.X('Variable', title = ''),
    x = 'Loading',
    color = 'Principal Component'
)

# create line at zero
rule = base.mark_rule().encode(x = alt.X('zero', title = 'Loading'), size = alt.value(0.05))

# layer
loading_plot = (loadings + rule).properties(width = 120, title = 'Loadings from unscaled PCA')

# show
loading_plot
```

Out[36]:



Notice that the variables with nonzero loadings in unscaled PCA are simply the three variables with the largest variances.

In [37]:
```
# three largest variances
x_mx.var().sort_values(ascending = False).head(3)
```

Out[37]:
```
IncomePerCap    3.804072e+07
Minority        5.265263e+02
White           5.264985e+02
dtype: float64
```

---

2. Exploratory analysis based on PCA

Now that we have the principal components, we can use them for exploratory data visualizations. The principal component values are computed via `.fit_transform(...)` in the PCA module:

In [38]:
```
# principal component values
pca.fit_transform(x_ctr)
```

Out[38]:
```
array([[-6.88065991e-02, -1.64753870e+00,  7.49835096e-01, ...,
        -1.20443293e-02, -1.37438096e-01, -1.36665529e-02],
       [-7.02814073e-01, -1.42878047e+00,  9.99713602e-01, ...,
         1.81583515e-02, -7.11870830e-02, -6.05785726e-03],
       [ 4.01339525e+00, -7.13092030e-02, -7.04349996e-01, ...,
        -2.95164824e-01, -3.54231427e-01,  3.04955232e-03],
       ...,
       [-1.07529925e+00, -3.42603384e-01,  2.92017629e-01, ...,
        -1.39651347e-01,  1.07599304e-01,  1.45073942e-02],
       [-1.17486155e+00,  1.01948369e+00, -3.31889301e-01, ...,
        -6.53668444e-02,  1.22798558e-01,  2.96030715e-03],
       [-2.19830943e+00,  1.39898518e+00, -3.55560527e-01, ...,
        -5.74535104e-01, -3.46836319e-02,  4.01882982e-02]])
```

The cell below extracts the first four PCs and stores them as a dataframe.

In [39]:
```
# project data onto first four components; store as data frame
projected_data = pd.DataFrame(pca.fit_transform(x_ctr)).iloc[:, 0:4].rename(columns = {0: 'PC1', 1: 'PC2', 2: 'PC3', 3: 'PC4'})

# add state and county
projected_data[['State', 'County']] = census[['State', 'County']]

# print
projected_data.head(4)
```

Out[39]:

	PC1	PC2	PC3	PC4	State	County
0	-0.068807	-1.647539	0.749835	-0.500517	Alabama	Autauga
1	-0.702814	-1.428780	0.999714	-1.165125	Alabama	Baldwin
2	4.013395	-0.071309	-0.704350	0.195341	Alabama	Barbour
3	1.556478	-1.080257	-1.892863	1.543793	Alabama	Bibb

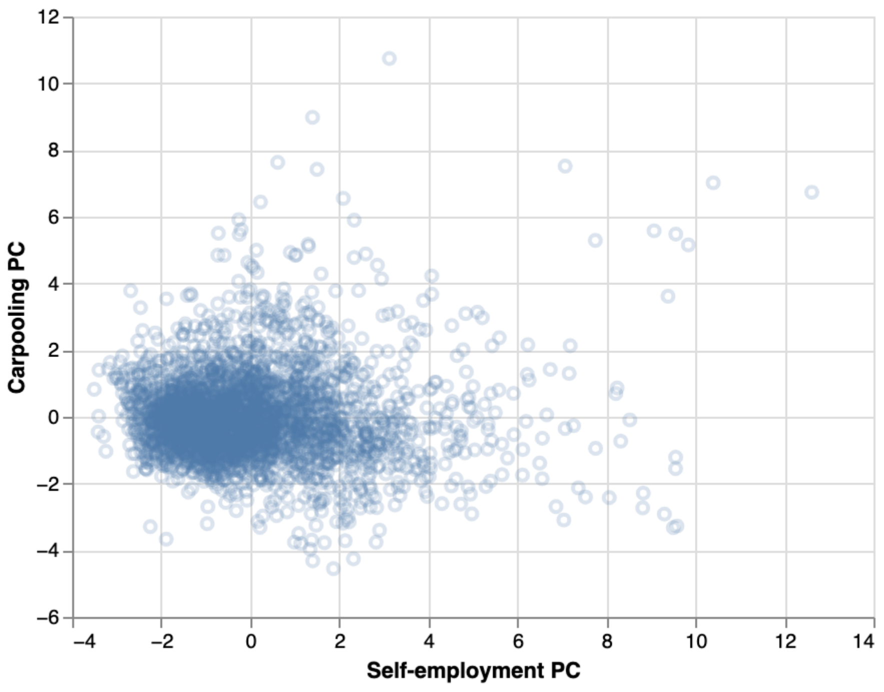
The PC's can be used to construct scatterplots of the data and search for patterns.

Outliers

The cell below plots PC2 (self-employment) against PC4 (carpooling):

In [40]:
```
# base chart
base = alt.Chart(projected_data)

# data scatter
scatter = base.mark_point(opacity = 0.2).encode(
    x = alt.X('PC2:Q', title = 'Self-employment PC'),
    y = alt.Y('PC4:Q', title = 'Carpooling PC')
)

# show
scatter
```

Out[40]:



Notice that there are a handful of outling points in the upper right region away from the dense scatter. What are those?

In order to inspect the outlying counties, we first need to figure out how to identify them. The outlying values have a large *sum* of PC2 and PC4. We can distinguish them by finding a cutoff value for the sum.

Question 2 (a)

Compute the sum of principal components 2 and 4 and sort them in descending order. Store the result in `pc_2plus4` and print the first 15 sorted values.

In [45]:
```
# find cutoff value
pc_2plus4 = (projected_data['PC2'] + projected_data['PC4']).sort_values(ascending = False)
#print
pc_2plus4.head(15)
```

Out[45]:
```
81      19.346522
82      17.417281
86      15.036571
84      14.993320
70      14.649029
85      14.587844
67      13.867717
73      13.039559
95      12.998509
739     10.377646
76       9.320243
2334     9.107116
68       8.918242
2498     8.910908
2079     8.639690
dtype: float64
```

In [46]:
```
grader.check("q2_a")
```

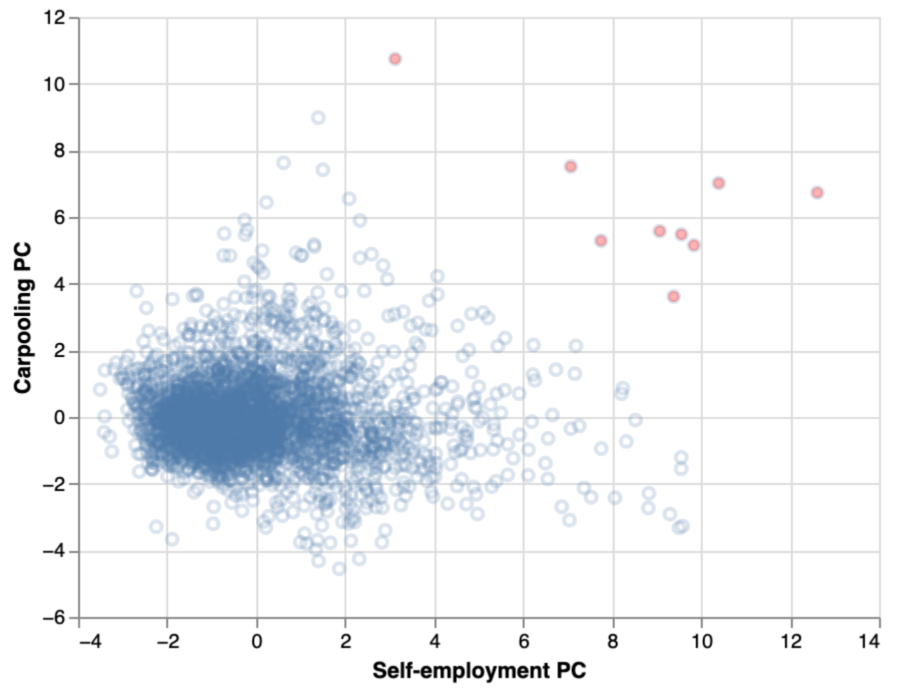
Out[46]:

q2_a passed!

Notice that there's a large jump from about 10 to about 13 (you could compare this with the typical jump using `.diff()` if you're curious); so we'll take 12 as the cutoff value. The plot below shows that this cutoff captures the points of interest.

In [47]:
```
# store outlying rows using cutoff
outliers = projected_data[(projected_data.PC2 + projected_data.PC4) > 12]

# plot outliers in red
pts = alt.Chart(outliers).mark_circle(
    color = 'red',
    opacity = 0.3
).encode(
    x = 'PC2',
    y = 'PC4'
)

# layer
scatter + pts
```

Out[47]:



Notice that all the outlying counties are remote regions of Alaska:

In [48]:

```
outliers
```

Out[48]:

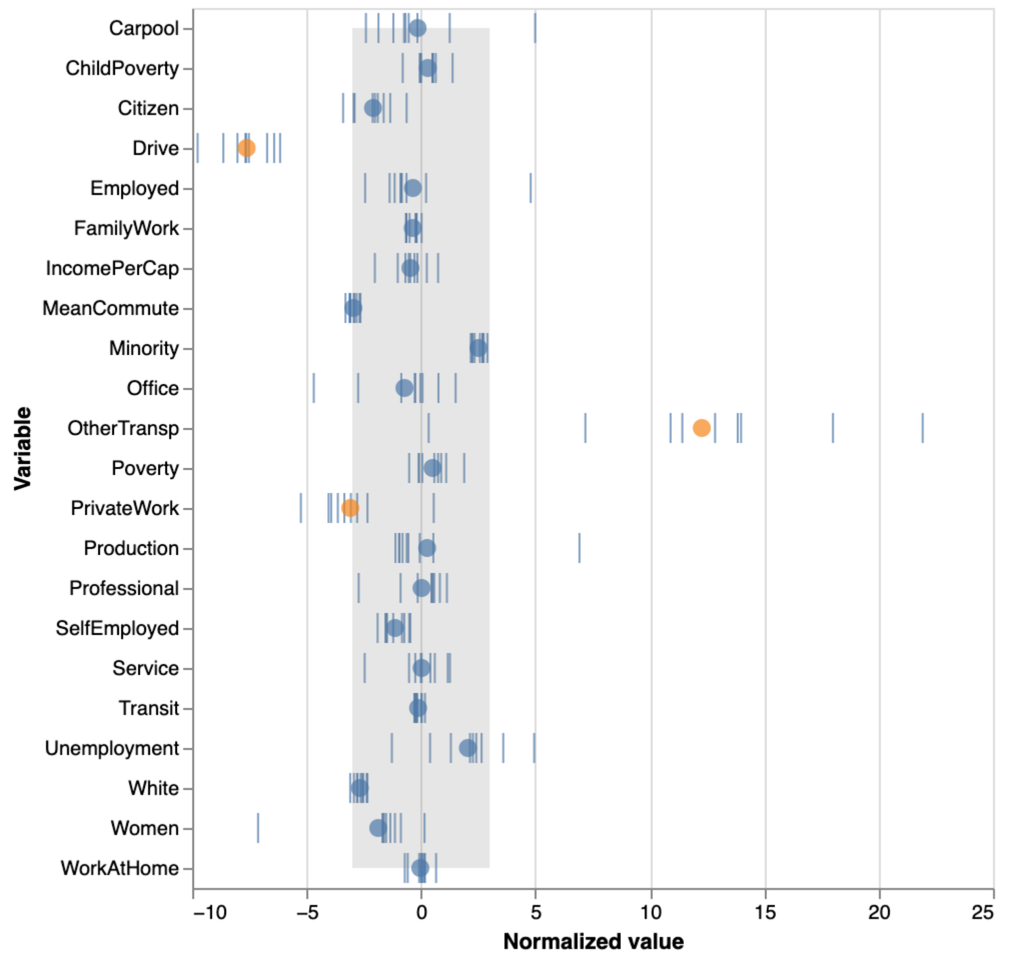
	PC1	PC2	PC3	PC4	State	County
67	-1.347609	3.125398	-1.248396	10.742319	Alaska	Aleutians East Borough
70	3.407044	9.073309	4.394921	5.575720	Alaska	Bethel Census Area
73	1.513140	7.752609	3.705396	5.286949	Alaska	Dillingham Census Area
81	5.827615	12.614982	5.394652	6.731539	Alaska	Kusilvak Census Area
82	0.943499	10.400522	3.811759	7.016760	Alaska	Lake and Peninsula Borough
84	2.882938	9.840638	3.773945	5.152682	Alaska	Nome Census Area
85	2.397895	7.074290	3.028609	7.513554	Alaska	North Slope Borough
86	3.128434	9.559099	4.164754	5.477473	Alaska	Northwest Arctic Borough
95	2.271239	9.385133	2.503918	3.613377	Alaska	Yukon-Koyukuk Census Area

What sets them apart? The cell below retrieves the normalized data and county name for the outlying rows, and then plots the normalized values of each variable for all 9 counties as vertical ticks, along with a point indicating the mean for the outlying counties. This plot can be used to determine which variables are over- or under-average for the outlying counties relative to the nation by simply locating means that are far from zero in either direction.

In [49]:

```
# retrieve normalized data for outlying rows
outlier_data = x_ctr.loc[outliers.index.values].join(
    census.loc[outliers.index.values, ['County']]
)

# melt to long format for plotting
outlier_plot_df = outlier_data.melt(
    id_vars = 'County',
    var_name = 'Variable',
    value_name = 'Normalized value'
)

# plot ticks for values (x) for each variable (y)
ticks = alt.Chart(outlier_plot_df).mark_tick().encode(
    x = 'Normalized value',
    y = 'Variable'
)

# shade out region within 3SD of mean
grey = alt.Chart(
    pd.DataFrame(
        {'Variable': x_ctr.columns,
         'upr': np.repeat(3, 22),
         'lwr': np.repeat(-3, 22)}
    )
).mark_area(opacity = 0.2, color = 'gray').encode(
    y = 'Variable',
    x = alt.X('upr', title = 'Normalized value'),
    x2 = 'lwr'
)

# compute means of each variable across counties
means = alt.Chart(outlier_plot_df).transform_aggregate(
    group_mean = 'mean(Normalized value)',
    groupby = ['Variable']
).transform_calculate(
    large = 'abs(datum.group_mean) > 3'
).mark_circle(size = 80).encode(
    x = 'group_mean:Q',
    y = 'Variable',
    color = alt.Color('large:N', legend = None)
)

# layer
ticks + grey + means
```

Out[49]:



Question 2 (b)

The two variables that clearly set the outlying counties apart from the nation are the percentage of the population using alternative transportation (extremely above average) and the percentage that drive to work (extremely below average). Why is this?

(*Hint:* take a peek at the Wikipedia page on transportation in Alaska.)

According to wikipedia, Alaska is the least connected in terms of road transporation.

Regional patterns

Are there regional patterns in the data? The cell below merges a table of U.S. census regions with the projected data.

In [50]:
```
# add US region
regions = pd.read_table('data/regions.txt', sep = ',')
plot_df = pd.merge(projected_data, regions, how = 'left', on = 'State')

# any non-matches?
plot_df.Region.isna().mean()
```

Out[50]:
```
0.024238657551274082
```

However, there are some counties that didn't get a match in the region table. In fact, all of Puerto Rico:

In [51]:
```
# inspect rows with missing region
plot_df[plot_df.Region.isna()].State.value_counts()
```

Out[51]:
```
Puerto Rico    78
Name: State, dtype: int64
```

That's an easy fix. We'll just give PR its own epynomous region.

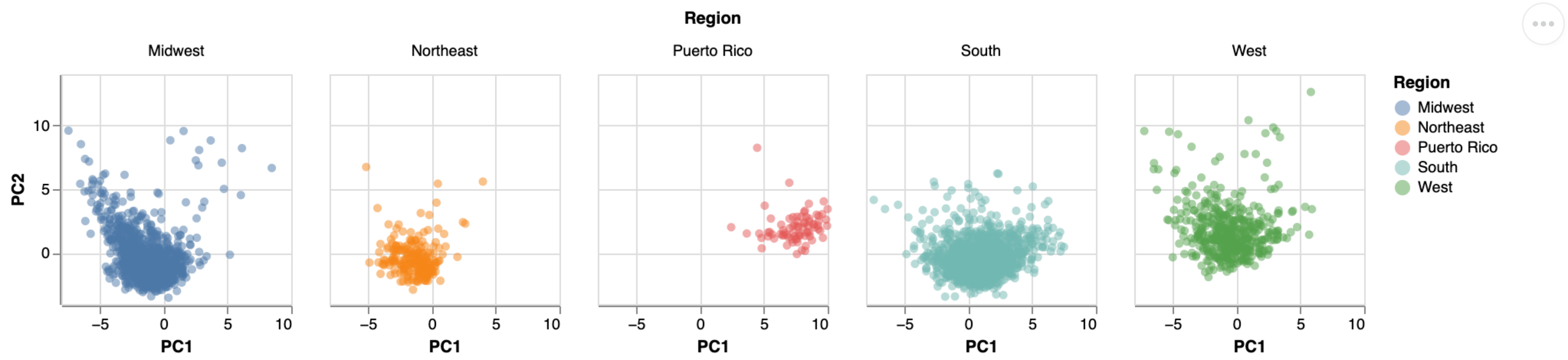
In [52]:
```
# replace NaNs
plot_df.Region = plot_df.Region.fillna('Puerto Rico')
```

Question 2 (c)

Use `plot_df` to construct a faceted scatterplot of PC2 against PC1 by region, and color the points by region.

In [53]:
```
# base chart
base = alt.Chart(plot_df).mark_circle(opacity = 0.5 ).encode(
x = alt.X('PC1'),
y = alt.Y('PC2'),
color = 'Region').properties( height = 150,
width = 150
).facet('Region')
# data scatter

# show
base
```

Out[53]:



Question 2 (d)

How does the northeast compare with the south?

(i) Describe the location of scatter along the PC axes.

(For instance, the western region scatter is centered around a PC1 value just below zero, say around -1, and a PC2 value just above zero, say around 2.)

Answer

The north east region is centered around a PC1 value just below 0, around -1, and a PC2 value around 0. The south region is centered arounf a PC1 value above 0, around 1, and a PC2 value around 0

(ii) Can you interpret the difference in location in any way?

State one qualitative difference in southern and northeastern counties that this points to.

Answer

PC1 represents income while PC2 represents self employment. The south has a higher PC1 value than the northeast which means that the south has less income and more poverty.

Submission Checklist

1. Save file to confirm all changes are on disk
2. Run *Kernel > Restart & Run All* to execute all code from top to bottom
3. Save file again to write any new output to disk
4. Select *File > Download as > HTML*.
5. Open in Google Chrome and print to PDF on A3 paper in portrait orientation.
6. Submit to Gradescope

To double-check your work, the cell below will rerun all of the autograder tests.

In [54]:
```
grader.check_all()
```

Out[54]:
```
q1_a results: All test cases passed!

q1_b results: All test cases passed!

q1_c results: All test cases passed!

q2_a results: All test cases passed!
```

In [ ]: