Information Visualization I

School of Information, University of Michigan

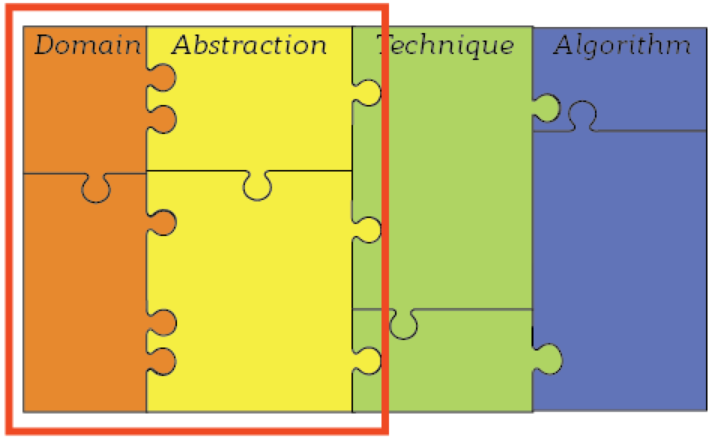
Week 1:

- Domain identification vs Abstract Task extraction
- Pandas Review

Assignment Overview

The objectives for this week are for you to:

- Review, reflect, and apply the concepts of Domain Tasks and Abstract Tasks. Specifically, given a real context, identify the expert's goals and then abstract the visualization tasks.



- Review and evaluate the domain of Pandas (https://pandas.pydata.org/) as a tool for reading, manipulating, and analyzing datasets in Python.

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

- Case study reflection: Car congestion and crash rates (20 points)
- Pandas programming exercise (80 points)

Resources:

- We're going to be recreating parts of this article by CMAP (https://www.cmap.illinois.gov/) available online (https://www.cmap.illinois.gov/updates/all/-/asset_publisher/UIMfSLnFfMB6/content/crash-scans-show-relationship-between-congestion-and-crash-rates) (CMAP, 2016)
- We'll need the datasets from the city of Chicago. We have downloaded a subset to the local folder /assets (assets/)
    - If you're curious, the original dataset can be found on Chicago Data Portal (https://data.cityofchicago.org/)
        - Chicago Traffic Tracker - Historical Congestion Estimates by Segment - 2011-2018 (https://data.cityofchicago.org/Transportation/Chicago-Traffic-Tracker-Historical-Congestion-Esti/77hq-huss)

    - [Traffic Crashes - Crashes (https://data.cityofchicago.org/Transportation/Traffic-Crashes-Crashes/85ca-t3if)](https://data.cityofchicago.org/Transportation/Traffic-Crashes-Crashes/85ca-t3if)
- Pandas
    - This assignment is partially a warm-up/reminder of how to use Pandas. We've also created an optional lab for you (see Coursera) if you need more help remembering how to do things in Pandas.
- Altair
    - We will use a python library called [Altair (https://altair-viz.github.io/)](https://altair-viz.github.io/) for the visualizations. Don't worry about understanding this code. You will only need to prepare the data for the visualization in Pandas. If you do it correctly, our code will produce the visualization for you.
    - If you're interested, we made a short [7-minute video (https://www.youtube.com/watch?v=Tg41r3lAYoQ)](https://www.youtube.com/watch?v=Tg41r3lAYoQ) explaining the very basics of how Grammar of Graphics/Altair works and why we need to transform the data as we do.

Important notes:

1) Grading for this assignment is entirely done by a human grader. They will be running tests on the functions we ask you to create. This means there is no autograding (submitting through the autograder will result in an error). You are expected to test and validate your own code.

2) Keep your notebooks clean and readable. If your code is highly messy or inefficient you will get a deduction.

3) Pay attention to the inputs and return types of your functions. Sometimes things will look right but fail later if you return the wrong kind of object (e.g., Array instead of Series). *Do not* hard-code variables into your functions. *Do not* modify our function definitions.

4) Follow the instructions for submission on Coursera. You will be providing us a generated link to a read-only version of your notebook and a PDF. 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)](https://youtu.be/PiO-K7AoWjk).

Part 1. Domain identification vs Abstract Task extraction (20 points)

Read the following article by CMAP [Crash scans show the relationship between congestion and crash rates (https://www.cmap.illinois.gov/updates/all/-/asset_publisher/UIMfSLnFfMB6/content/crash-scans-show-relationship-between-congestion-and-crash-rates)](https://www.cmap.illinois.gov/updates/all/-/asset_publisher/UIMfSLnFfMB6/content/crash-scans-show-relationship-between-congestion-and-crash-rates) and answer the following questions. Think of this as the output report produced by the analyst.

Remember: Domain tasks are questions an analyst might want to answer and/or they might be insights (answers) the analyst wants to communicate to someone else. For example, a retail analyst might want to know: how many fruit did we sell? or what's the relationship between temperature and fruits rotting? A learning analyst would have the domain task: how often do students pass the class? or how does study time correlate with grade? An advertising analyst would ask: how many people clicked on an ad? or what's the relationship between time of day and click through rate?

Abstract tasks are generic: What's the sum of a quantitative variable? or what's the correlation between two variables? Notice we gave two examples for each analyst type and these roughly map to the two abstract questions. You should not use domain language (e.g., accidents) when describing abstract tasks.

1.1 Briefly describe who you think performed this analysis. What is their expertise? What is their goal for the article? Give 3 examples of domain tasks featured in the article. (10 points)

1.1 Answer

This analysis was likely performed by a CMAP analyst, whose specialty is to work with vehicle crash data in order to derive insights which would help in the development of strategies to preclude or mitigate road crashes. Their goal for the analysis was to assess vehicle crash rates in different situations by means of visual aids using data obtained from the Illinois Department of Transportation (IDOT). The article is likely a report of their findings on analysing possible relationships between potential elements involved in crash rates in different scenarios, the results of which would assist in formulating practical strategies as part of the GO TO 2040 development plan.

3 examples of Domain Tasks (questions the analyst wants to answer) featured in the article are:

1. Is the crash rate higher in certain locations (intersections, merging/diverging areas, etc.) than others?

2. Are road congestion and crash frequency closely related?
3. Is there a difference between crash rates on expressways versus arterials?

1.2 For each domain task describe the abstract task (10 points)

1.2 Answer

For each Domain Task described above, the corresponding Abstract Task is as follows:

1. Find the sums of a quantitative variable with respect to another qualitative variable.
2. Find the correlation between two variables.
3. Find the sums of a quantitative variable with respect to another qualitative variable.

Part 2. Pandas programming exercise (80 points)

We have provided some code to create visualizations based on these two datasets:

1. [Historic Congestion (assets/Pulaski.small.csv.gz)](assets/Pulaski.small.csv.gz)
2. [Traffic Crashes (assets/Traffic.Crashes.csv.gz)](assets/Traffic.Crashes.csv.gz)

Complete each assignment function and run each cell to generate the final visualizations

In [1]:
```
import pandas as pd
import numpy as np
import altair as alt
```

In [2]:
```
# enable correct rendering
alt.renderers.enable('default')
```

Out[2]:
```
RendererRegistry.enable('default')
```

In [3]:
```
# uses intermediate json files to speed things up
alt.data_transformers.enable('json')
```

Out[3]:
```
DataTransformerRegistry.enable('json')
```

PART A: Historic Congestion (55 points)

For parts 2.1 to 2.5 we will use the Historic Congestion dataset. This dataset contains measures of speed for different segments. For this subsample, the available measures are limited to traffic on Pulaski Road in 2018.

2.1 Read and resample (15 points)

Complete the `read_csv` and `get_group_first_row` functions. Since our dataset is large we want to only grab one measurement per hour for each segment. To do this, we will resample by grouping based on some columns (e.g., month, day, hour for each segment) and then picking out the first measurement from that group. We're going to write the sampling function to be generic. Complete the `get_group_first_row` function to achieve this. Note that the file we are loading is compressed--depending on how you load the file, this may or may not make a difference ([you'll want to look at the API documents (https://pandas.pydata.org/pandas-docs/stable/reference/index.html)](https://pandas.pydata.org/pandas-docs/stable/reference/index.html)).

In [4]:
```
def read_csv(filename):
    """Read the csv file from filename (uncompress 'gz' if needed)
    return the dataframe resulting from reading the columns
    """
    # YOUR CODE HERE

    import pandas as pd

    df=pd.read_csv(filename)

    return df

#     raise NotImplementedError()
```

In [5]:
```
# Save the congestion dataframe on hist_con
hist_con = read_csv('assets/Pulaski.small.csv.gz')
print(hist_con.shape)
assert hist_con.shape == (3195450, 10)
assert list(hist_con.columns) == ['TIME','SEGMENT_ID','SPEED','STREET','DIRECTION','FROM_STREET','TO_STREET',
                                  'HOUR','DAY_OF_WEEK','MONTH']
```

```
(3195450, 10)
```

In [6]:
```
def get_group_first_row(df, grouping_columns):
    """Group rows using the grouping_columns argument and return the first row belonging to each group
    (you can look at first() for reference). We'll write this function to be more general in case
    we want to use it for a different resample.
    return a dataframe without a hierarchical index (important: return with default index)

    See the example link below if you want a better sense of what this should return
    """
    # YOUR CODE HERE

    first_rows_df=df.groupby(grouping_columns).head(1)

    return first_rows_df

#     raise NotImplementedError()
```

In [7]:
```
# test your code, we want segment_rows to be resampled version of hist_con where we've grouped by the
# properties month, day_of_week, hour, and segment_id and returned the first measure of each group
segment_rows = get_group_first_row(hist_con, ['MONTH','DAY_OF_WEEK', 'HOUR', 'SEGMENT_ID'])

# ADD YOUR TESTS HERE

segment_rows.sample(5)
```

Out[7]:

	TIME	SEGMENT_ID	SPEED	STREET	DIRECTION	FROM_STREET	TO_STREET	HOUR	DAY_OF_WEEK	MONTH
19664	12/30/2018 04:50:19 AM	25	-1	Pulaski	NB	87th	83rd	4	1	12
685494	10/27/2018 07:50:20 AM	92	23	Pulaski	SB	Irving Park	Milwaukee	7	7	10
1955913	06/27/2018 03:50:23 PM	22	-1	Pulaski	NB	99th	95th	15	4	6
665297	10/29/2018 04:50:23 AM	48	26	Pulaski	NB	North Ave	Armitage	4	2	10
1662735	07/25/2018 04:50:24 AM	22	-1	Pulaski	NB	99th	95th	4	4	7

The table should look something like this (assets/segment_rows.png).

***Note** When we show examples like this, we are sampling (e.g., `segment_rows.sample(5)`) so your table may look different.

If you want to build your own tests from our example tables, you can create an assert test for one of the rows and make sure the values match what you expect. For example we see that the row id 68592 in the example is for 8/27/2018 at 1:50:21 PM. So we could write the test:

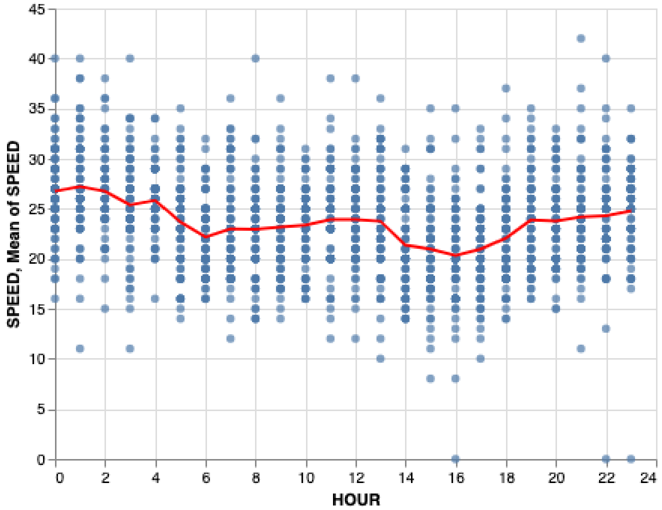
```
assert segment_rows.loc[68952].TIME == '08/27/2018 01:50:21 PM'
```

If this assertion failed, you'd get an error message.

Now let's do something a little bit interesting with this. We should now be able to test a theory that traffic speeds vary by hour of day. We're going to create a scatter plot showing hour on the x-axis and speed on the y-axis. We're going to sample only one hour per segment to keep things simple. So for each segment (we have 78 of them) we're going to find the first speed measure for 12am, 1am, 2am, etc. The result will be roughly 1872 points (plus or minus, we have some missing data). On top of that, we will add a line for the mean speed for each hour. To plot this, we need our data to look roughly like this:

	HOUR	SEGMENT_ID	SPEED
1651	21	32	25
1210	15	60	23
1271	16	42	13
1048	13	53	31
1049	13	54	28

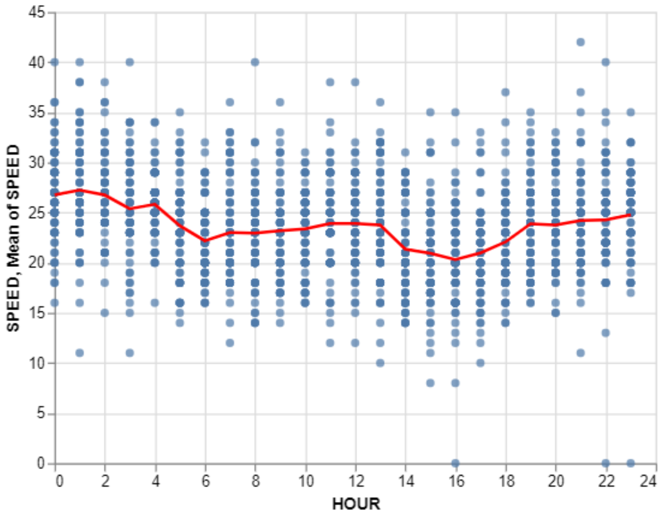
This will allow the encoding system to read row by row, and create a point for each where the X is the hour and Y is the speed. If everything works, you'll see:



Notice the dip in speeds around morning and afternoon rush hours.

In [8]:
```python
def create_mean_speed_vis(indf):
    # input: indf -- the input frame (in style of hist_con above)

    # take the history of congestion data and only keep rows where speed > -1 (-1 being missing data)
    srows = indf[indf.SPEED>-1]

    # sub-sample for hour/segment
    srows = get_group_first_row(srows, ['HOUR', 'SEGMENT_ID'])

    # grab the only columns we care about (strictly speaking, we only need HOUR and SPEED)
    srows = srows[['HOUR','SEGMENT_ID','SPEED']]

    # create the scatter plot using this data
    distr = alt.Chart(srows).mark_circle().encode(
        x='HOUR:Q',  # x is the HOUR
        y='SPEED:Q'  # y is the speed
    )

    # create the line chart on top, we could calculate the means in either Pandas or Altair
    mean = distr.mark_line(color='red').encode(
        # this "extends" distr, so x is still encoding HOUR
        y='mean(SPEED):Q' # y should now encode the mean of SPEED (at each hour)
    )

    # combine the scatter plot and line chart
    return distr+mean

create_mean_speed_vis(hist_con)
```

Out[8]:



2.2 Basic Bar Chart Visualization (10 points)

We want to create a bar chart visualization for the *average speed* of each segment (across all the samples). Our encoder is going to want the data so we that we have one row per segment, with a segment id column (we'll use this for the X placement of the bars) and the average speed (we'll use this for the length of the bar). So something like this:

	SEGMENT_ID	SPEED
62	82	12.830468
32	51	13.075874
12	31	11.920569
76	96	21.659751
46	66	14.857143

To do this, we're going to want to group by each segment and calculate the average speed on each. Complete this code on the `average_speed_per_segment` function. Make sure your function returns a ***series***.

In [9]:
```
def average_speed_per_segment(df):
    """Group rows by SEGMENT_ID and calculate the mean of each
    return a *series* where the index is the segment id and each value is the average speed per segment
    """
    # YOUR CODE HERE

    speed_mean_df=df.groupby(["SEGMENT_ID"])["SPEED"].mean()

    return speed_mean_df

#     raise NotImplementedError()
```

In [10]:
```
# reset to a "clean" segment_rows
segment_rows = get_group_first_row(hist_con, ['MONTH','DAY_OF_WEEK', 'HOUR', 'SEGMENT_ID'])

# calculate the average speed per segment
average_speed = average_speed_per_segment(segment_rows)

# ADD YOUR TESTS HERE
assert type(average_speed) == pd.core.series.Series

# check what's in average_speed
average_speed
```

Out[10]:
```
SEGMENT_ID
19    12.251926
20    15.274452
21    12.141079
22    12.346769
23    12.716657
        ...
93    13.503260
94    14.560759
95    14.959099
96    21.659751
97    18.714286
Name: SPEED, Length: 78, dtype: float64
```

If you got things right, the **series** should look something like [this (assets/average_speed.png)](assets/average_speed.png). You might want to write a test to make sure you are returning the expected type. For example:

```
assert type(average_speed) == pd.core.series.Series
```

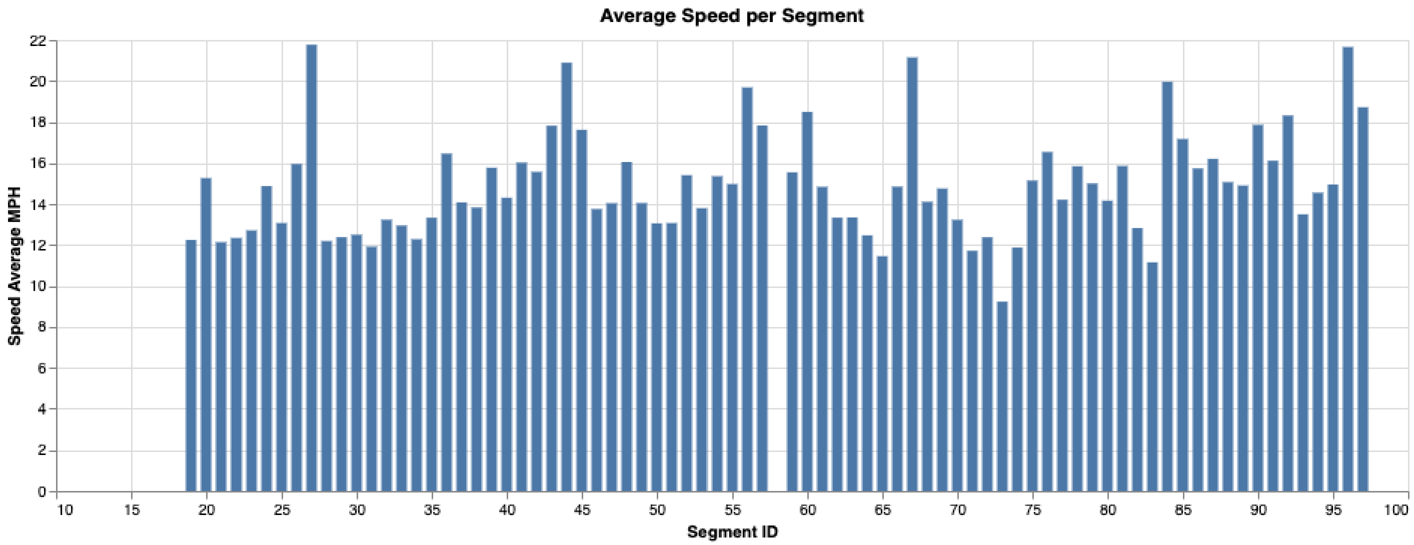
In [11]:
```
# make a dataframe from the average_speed
def get_average_speed_df(indf):
    # input: indf the input data frame (like hist_con)
    # reset segment rows
    segment_rows = get_group_first_row(indf, ['MONTH','DAY_OF_WEEK', 'HOUR', 'SEGMENT_ID'])

    # calculate the average speed
    average_speed = average_speed_per_segment(segment_rows)
    # create the data frame
    asdf = pd.DataFrame(average_speed).reset_index()
    #return the frame
    return asdf
```

In [12]:
```
# see what's inside
average_speed_df = get_average_speed_df(hist_con)

# ADD YOUR TESTS HERE
assert round(average_speed_df.iloc[70,1],2)==17.85

# print a sample
average_speed_df.sample(5)
```

Out[12]:

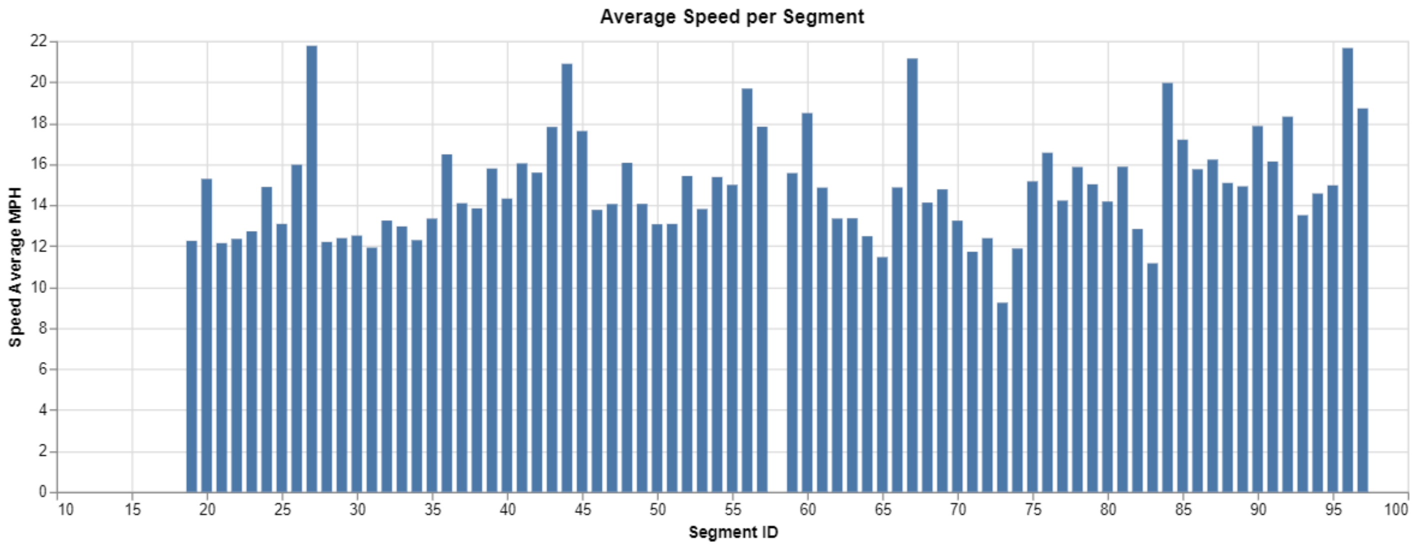
	SEGMENT_ID	SPEED
24	43	17.809129
38	57	17.820391
60	80	14.169532
74	94	14.560759
73	93	13.503260

Ok, now we can build our visualization. If your code is correct, you should seem something like:



In [13]:
```python
# let's generate the visualization

def create_average_speed_per_segment_vis(visdf):
    # visdf: frame to visualize

    # create a chart
    base = alt.Chart(visdf)

    # we're going to "encode" the variables, more on this next assignment
    encoding = base.encode(
        x= alt.X(                # encode SEGMENT_ID as a quantiative variable on the X axis
                'SEGMENT_ID:Q',
                title='Segment ID',
                scale=alt.Scale(zero=False)   # we don't need to start at 0
        ),
        y=alt.Y(
                'sum(SPEED):Q',   # encode the sum of speed for the segment as a quantitative variable on Y
                title='Speed Average MPH'
        ),
    )

    # we're going to use a bar chart and set various parameters (like bar size and title) to make it readable
    return encoding.mark_bar(size=7).properties(title='Average Speed per Segment',height=300, width=900)

create_average_speed_per_segment_vis(average_speed_df)
```

Out[13]:



2.3 Create a basic pivot table (10 points)

For the next visualization, we need a more complex transformation that will allow us to see the average speed for each month. We're going to use a heatmap style calendar visualization (think GitHub) check-in history. Our encoder is going to make a square for each segment/month. The segment id will tell us where on the x-axis to put the square and the month value will tell us where on the y-axis. We will also want the mean speed as a column (for that month/segment) which we'll encode using color. What we're working towards is a dataframe that looks something like:

	SEGMENT_ID	MONTH	SPEED
630	77	5	13.089286
421	57	5	17.178571
327	48	10	16.434524
49	23	7	12.267857
776	90	8	18.220238

We're going to do part of this for you. First, we need you to use a pivot table to get us part way there. For the pivot table we want a table where the index is the month, and each column is a segment id. We will put the average speed in the cells.

Complete the `create_pivot_table` function for this. The table you output should look something like [this (assets/pivot_table.png)](assets/pivot_table.png)

In [14]:
```
def create_pivot_table (df):
    """return a pivot table where:
    each row i is a month
    each column j is a segment id
    each cell value is the average speed for the month i in the segment j
    """
    # YOUR CODE HERE

    pivot_df=df.groupby(["MONTH","SEGMENT_ID"])["SPEED"].mean().unstack()

    return pivot_df

#     raise NotImplementedError()
```

In [15]:
```
# go back to our original sample for segment_rows
segment_rows = get_group_first_row(hist_con, ['MONTH','DAY_OF_WEEK', 'HOUR', 'SEGMENT_ID'])
```

In [16]:
```python
# run the code and see what's in the table
pivot_table = create_pivot_table(segment_rows)
pivot_table
```

Out[16]:

SEGMENT_ID	19	20	21	22	23	24	25	26	27	28	...	88	89	90	91	92	93	9
MONTH																		
2	6.857143	16.142857	13.571429	19.571429	18.285714	15.857143	11.285714	10.142857	25.000000	20.571429	...	17.000000	14.714286	19.000000	17.857143	20.857143	12.000000	16.85714
3	10.773810	14.863095	11.696429	11.815476	13.583333	16.244048	12.398810	15.529762	21.779762	12.422619	...	15.130952	16.470238	17.744048	16.095238	18.095238	13.994048	15.87500
4	11.744048	14.958333	11.791667	12.071429	13.208333	16.779762	14.136905	18.339286	22.232143	11.589286	...	14.958333	14.642857	17.702381	15.386905	18.488095	14.250000	14.80357
5	11.357143	14.738095	11.369048	11.916667	12.023810	13.220238	11.505952	15.095238	22.857143	11.892857	...	14.154762	12.553571	16.184524	15.130952	17.952381	12.607143	12.97619
6	11.630952	14.583333	13.011905	12.279762	12.428571	14.678571	12.690476	15.244048	22.309524	12.619048	...	16.089286	14.869048	17.511905	15.220238	19.035714	14.071429	14.31547
7	11.755952	13.595238	10.880952	12.238095	12.267857	14.321429	13.232143	14.964286	22.232143	11.958333	...	17.220238	15.511905	19.476190	15.630952	18.666667	13.630952	14.85714
8	12.988095	15.446429	12.303571	13.315476	13.023810	15.827381	12.988095	16.946429	22.244048	12.535714	...	14.863095	13.880952	18.220238	15.196429	17.994048	13.648810	13.66666
9	13.970238	17.059524	14.398810	13.452381	12.017857	14.869048	12.571429	15.630952	21.571429	12.464286	...	16.178571	14.916667	17.922619	14.101190	16.833333	11.952381	13.00595
10	13.708333	15.666667	12.434524	13.041667	12.422619	13.714286	13.613095	15.922619	20.333333	13.119048	...	14.291667	15.351190	18.059524	19.273810	18.119048	13.244048	15.16071
11	12.970238	16.107143	11.922619	11.476190	12.125000	15.607143	14.327381	16.815476	20.553571	12.910714	...	13.422619	15.821429	18.250000	16.357143	17.922619	12.434524	15.60714
12	11.845238	15.690476	11.541667	11.559524	13.833333	13.523810	13.375000	15.404762	21.446429	10.113095	...	14.380952	15.101190	17.404762	18.755952	19.898810	15.261905	15.24404

11 rows × 78 columns

As before, we can write a "test" based on this example. For example, [here (assets/pivot_table.png)](assets/pivot_table.png) we see that in March (Month 3) segment 21 had a value of ~11.696, so we could write the test:

```python
assert round(pivot_table.loc[3,21],3) == 11.696
```

In [17]:
```python
# add your tests here
assert round(pivot_table.loc[3,21],3) == 11.696
```

In [18]:
```python
# we're going to implement a transformation to put the pivot table into a 'long form' because it
# is easier to specify the visualization.
def make_long_form(sourceTable):
    # sourceTable: the original table to modify
    hm_pivot_table = sourceTable.copy().unstack().reset_index()
    hm_pivot_table['SPEED'] = hm_pivot_table[0]
    hm_pivot_table.drop(0,axis=1,inplace=True)
    return(hm_pivot_table)
```

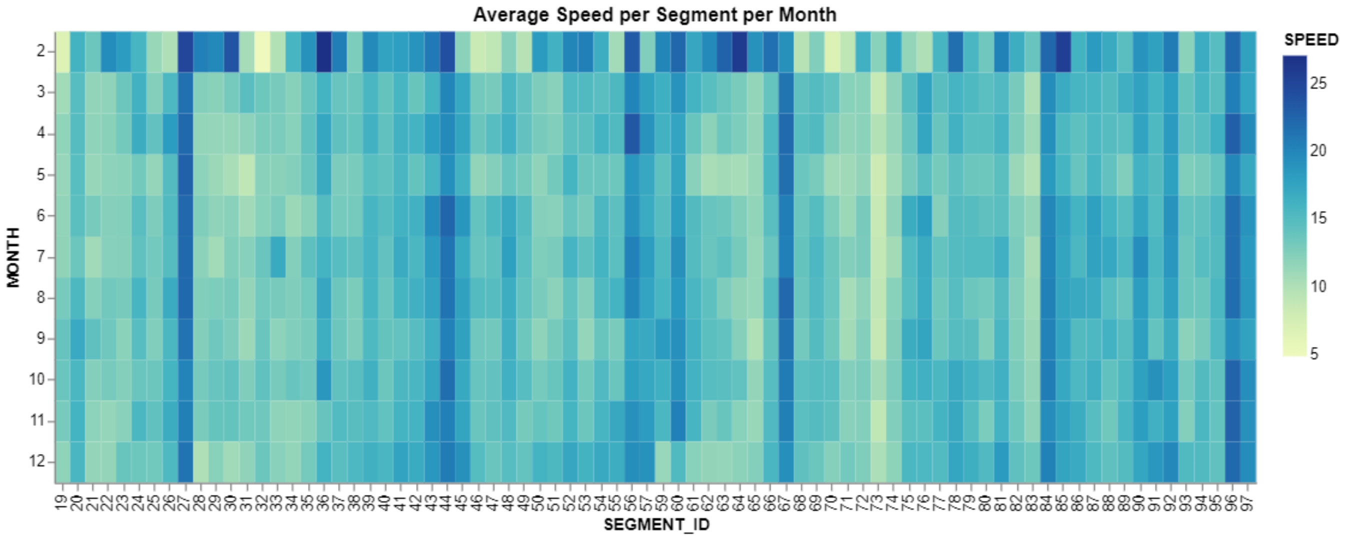
In [19]:
```
# you can see what's inside the long form
longformASSM = make_long_form(pivot_table)
longformASSM.sample(5)
```

Out[19]:

	SEGMENT_ID	MONTH	SPEED
457	61	8	15.714286
662	80	4	14.744048
381	53	9	11.488095
11	20	2	16.142857
733	86	9	15.160714

In [20]:
```python
# create the visualization. We're going to use rectangles (a heat map of sorts). We'll use the segment_id to
# figure out the horizontal placement (x), the month as the vertical (y) and use color to encode the speed.
def create_speed_month_segment_vis(visframe):
    # visframe: the frame to visualize

    # using rectangles
    encoding = alt.Chart(visframe).mark_rect().encode(
        x='SEGMENT_ID:O',   # segments on the x axis, ordinal encoding so ordered
        y='MONTH:O',        # month, ordinal encoding so ordered
        color='SPEED:Q'     # color based on speed, quantitative encoding
    )

    # adjust title, height, width and return
    return encoding.properties(title='Average Speed per Segment per Month',height=300, width=800)

create_speed_month_segment_vis(longformASSM)
```

Out[20]:



2.4 Sorting, Transforming, and Filtering (20 points)

Without telling you too much about the visualization we want to create next (that's part of the bonus below), we need to get the data into a form we can use. In the end, we'll want something roughly like:

	DIRECTION	FROM_STREET	TIME_HOURS	SPEED
19604	NB	Chicago	2018-04-25 05:00:00	27
10197	SB	26th	2018-03-30 04:00:00	25
129400	SB	Bryn Mawr	2018-12-28 20:00:00	28
97132	NB	Roosevelt	2018-10-30 15:00:00	21
4786	NB	Grand	2018-03-27 07:00:00	24

To do this:

- We're going to need to sort the dataframe by one or more columns (this is the `sort_by_col` function).
- We'll want to create a derivative column that is the time of the measurement rounded to the nearest hour (`time_to_hours`)
- We need to "facet" the data into groups to generate different visualizations.
- We need a function that selects part of the dataframe that matches a specific characteristic (`filter_orientation`)

In [21]:
```
def sort_by_col(df, sorting_columns):
    """Sort the rows of df by the columns (sorting_columns)
    return the sorted dataframe
    """
    # YOUR CODE HERE

    sorted_df=df.sort_values(sorting_columns)

    return sorted_df

#     raise NotImplementedError()
```

In [22]:
```
# test it out
segment_rows = sort_by_col(segment_rows, ['SEGMENT_ID'])
segment_rows.sample(5)
```

Out[22]:

	TIME	SEGMENT_ID	SPEED	STREET	DIRECTION	FROM_STREET	TO_STREET	HOUR	DAY_OF_WEEK	MONTH
990931	09/29/2018 01:50:25 AM	36	-1	Pulaski	NB	I-55 Expy	31st	1	7	9
2899324	03/28/2018 08:50:34 AM	31	20	Pulaski	NB	59th	55th	8	4	3
2911102	03/27/2018 03:50:21 AM	82	-1	Pulaski	SB	Washington	Van Buren	3	3	3
636628	10/31/2018 09:50:23 PM	68	27	Pulaski	SB	67th	71st	21	4	10
1604451	07/30/2018 02:50:25 PM	49	21	Pulaski	NB	Armitage	Fullerton	14	2	7

In [23]:
```
segment_rows.head()
```

Out[23]:

	TIME	SEGMENT_ID	SPEED	STREET	DIRECTION	FROM_STREET	TO_STREET	HOUR	DAY_OF_WEEK	MONTH
1591550	07/31/2018 08:50:18 PM	19	-1	Pulaski	NB	111th	107th	20	3	7
2606056	04/26/2018 04:50:22 PM	19	-1	Pulaski	NB	111th	107th	16	5	4
1977337	06/25/2018 02:50:25 PM	19	23	Pulaski	NB	111th	107th	14	2	6
667739	10/28/2018 09:50:24 PM	19	-1	Pulaski	NB	111th	107th	21	1	10
379923	11/26/2018 12:50:21 PM	19	27	Pulaski	NB	111th	107th	12	2	11

In [24]:
```
def time_to_hours(df):
    """ Add a column (called TIME_HOURS) based on the data in the TIME column and rounded up
    the value to the nearest hour.  For example, if the original TIME row said:
    ‘02/28/2018 05:40:00 PM’ we want ‘2018-02-28 18:00:00’
    (the change is that 5:40pm was rounded up to 6:00pm and the TIME_HOUR column is
    actually a proper datetime and not a string).The column should be a datetime type.
    """
    # YOUR CODE HERE

    # convert TIME column data type to datetime
    df.TIME=pd.to_datetime(df.TIME)

    # create TIME_HOURS column to round off the time to the nearest hour
    df["TIME_HOURS"]=df.TIME.dt.round('H')

    return df

#     raise NotImplementedError()
```

In [25]:
```
# we can test this out
segment_rows = time_to_hours(segment_rows)
segment_rows.sample(5)
```

Out[25]:

	TIME	SEGMENT_ID	SPEED	STREET	DIRECTION	FROM_STREET	TO_STREET	HOUR	DAY_OF_WEEK	MONTH	TIME_HOURS
1655408	2018-07-25 21:50:23	96	-1	Pulaski	SB	Bryn Mawr	Foster	21	4	7	2018-07-25 22:00:00
1948372	2018-06-28 08:40:13	74	24	Pulaski	SB	43rd	47th	8	5	6	2018-06-28 09:00:00
1347577	2018-08-26 21:50:20	33	36	Pulaski	NB	Archer	47th	21	1	8	2018-08-26 22:00:00
1951700	2018-06-28 00:50:25	80	-1	Pulaski	SB	Roosevelt	16th	0	5	6	2018-06-28 01:00:00
1608191	2018-07-30 06:50:21	53	19	Pulaski	NB	Irving Park	Elston	6	2	7	2018-07-30 07:00:00

In [26]:
```
def filter_orientation(df, traffic_orientation):
    """ Filter the rows according to the traffic orientation
    return a df that is a subset of the original with the desired orientation
    df: original traffic data frame
    traffic_orientation: a string, one of "SB" or "NB"
    """
    # YOUR CODE HERE

    df=df.loc[df.DIRECTION==traffic_orientation]

    return df

#     raise NotImplementedError()
```

In [27]:
```
# let's filter down to a south bound and north bound table
sb = filter_orientation(segment_rows, 'SB')
nb = filter_orientation(segment_rows, 'NB')
```

The sb table should look like [this (assets/sb.png)](assets/sb.png)

In [28]:
```
# let's look at a sample. You might want to build some assert tests here
sb.sample(5)
```

Out[28]:

	TIME	SEGMENT_ID	SPEED	STREET	DIRECTION	FROM_STREET	TO_STREET	HOUR	DAY_OF_WEEK	MONTH	TIME_HOURS
1648931	2018-07-26 11:50:20	75	27	Pulaski	SB	I-55 Expy	43rd	11	5	7	2018-07-26 12:00:00
1362276	2018-08-25 11:50:21	80	35	Pulaski	SB	Roosevelt	16th	11	7	8	2018-08-25 12:00:00
1044939	2018-09-24 01:50:20	69	-1	Pulaski	SB	63rd	67th	1	2	9	2018-09-24 02:00:00
339945	2018-11-30 06:50:30	79	24	Pulaski	SB	16th	Cermak	6	6	11	2018-11-30 07:00:00
2920941	2018-03-26 05:50:23	69	25	Pulaski	SB	63rd	67th	5	2	3	2018-03-26 06:00:00

In [29]:
```python
# let's put it all together to generate our table
def get_sbnb(indf):
    # input: indf, a hist_con shaped data frame

    # go back to our original sample for segment_rows
    segment_rows = get_group_first_row(indf, ['MONTH','DAY_OF_WEEK', 'HOUR', 'SEGMENT_ID'])

    # use our new functions
    segment_rows = sort_by_col(segment_rows, ['SEGMENT_ID'])
    segment_rows = time_to_hours(segment_rows)
    sb = filter_orientation(segment_rows, 'SB')
    nb = filter_orientation(segment_rows, 'NB')

    # we're going to remove speeds of -1 (no data)
    sb = sb[sb.SPEED > -1]
    nb = nb[nb.SPEED > -1]

    # now append the columns and just select the sub columns we care about
    sbnb = sb.append(nb)[['DIRECTION','FROM_STREET','TIME_HOURS','SPEED']]
    return(sbnb)
```

In [30]:
```python
# let's see what's inside
sbnb = get_sbnb(hist_con)
sbnb.sample(5)
```

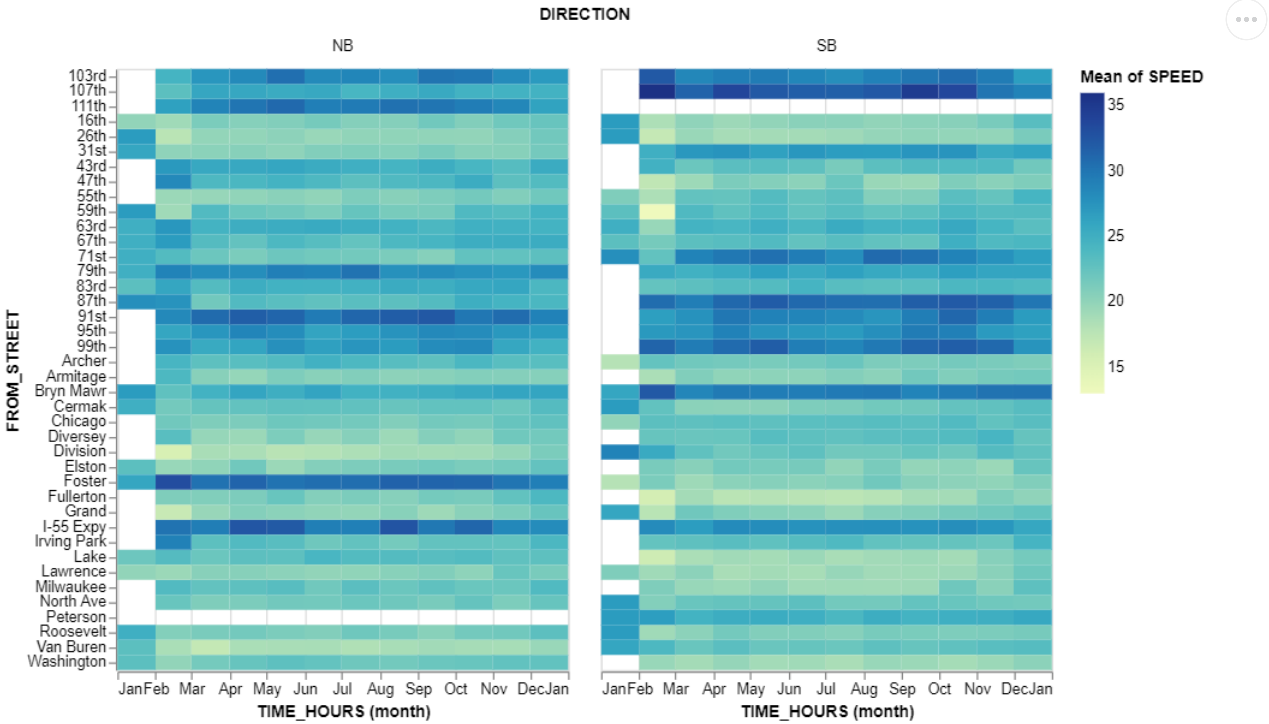
Out[30]:

	DIRECTION	FROM_STREET	TIME_HOURS	SPEED
1300515	SB	Roosevelt	2018-08-31 05:00:00	26
1044889	SB	Chicago	2018-09-24 02:00:00	20
2632261	SB	71st	2018-04-24 06:00:00	16
1972286	NB	Van Buren	2018-06-26 04:00:00	20
69229	SB	16th	2018-12-25 13:00:00	24

In [31]:

```python
# create the visualization, but it's your bonus (2.5) to describe what's going on
def create_speed_direction_vis(visdf):
    alt.data_transformers.disable_max_rows()
    return alt.Chart(visdf).mark_rect().encode(
        x='month(TIME_HOURS):T',
        y='FROM_STREET:N',
        color='mean(SPEED):Q',
        facet='DIRECTION:N'
    ).properties(
        width=300,
        height=400
    )

create_speed_direction_vis(sbnb)
```

Out[31]:



2.5 (Bonus) Traffic heatmap visualization (up to 2 points)

Looking at the visualization above (the one showing Northbound versus Southbound facets), what domain/abstract tasks are fulfilled by this visualization? List at least one domain task and the corresponding abstract task.

2.5 Answer

Domain Task: Is there a difference between average speeds per month for Northbound versus Southbound vehicles?

Abstract Task: Find the means of a quantitative variable grouped by other qualitative variables.

PART B: Crashes (25 points)

For parts 2.6 and 2.7 we will use the Crashes dataset. This dataset contains crash entries recording the time of the accident, the street, and the street number where the accident occurred. You will work with accidents recorded on Pulaski Road

In [32]:
```
# load the crash data
crashes = read_csv('assets/Traffic.Crashes.csv.gz')

# just grab the pulaski road data
crashes_pulaski = crashes[crashes.STREET_NAME == 'PULASKI RD']
```

2.6 Calculate summary statistics for grouped streets (15 points)

We want to get a few summary visualizations like where crashes are happening on Pulaski Road (by which house number). We're going to bin the records by house number to start. Think of bins as vaguely representing "street blocks" (it's obviously not quite right).

- Group the streets every 300 units (street numbers). Hint: You can use the `pd.cut` function

The second visualization will tell us around which houses accidents are happening.

- Calculate the number of accidents (count rows) and the total of injuries (sum injuries total) for each of these 300-chunk road segments. Do this *for each direction*.

Complete `bin_crashes` and `calculate_group_aggregates` functions for this

In [33]:

```python
def bin_crashes(df):
    """ Assign each crash instance a category (bin) every 300 house number units starting from 0
    Return a new dataframe with a column called BIN where each value is the start of the bin
    i.e. 0 is the label for records with street number n, where 1 <= n <= 300
    300 is the label for records with n at 301 <= n <= 600, and so on.
    """
    # YOUR CODE HERE

    df["BIN"]=pd.cut(x=df['STREET_NO'], bins=[x for x in range(0, 11701, 300)], labels=[x for x in range(0, 11700, 300)])

    return df

#     raise NotImplementedError()
```

In [34]:

```python
binned_df = bin_crashes(crashes_pulaski)

# sample the values to see what's in your new DF (we only care about street no and bin)
binned_df.sample(5)[['STREET_NO','BIN']]
```

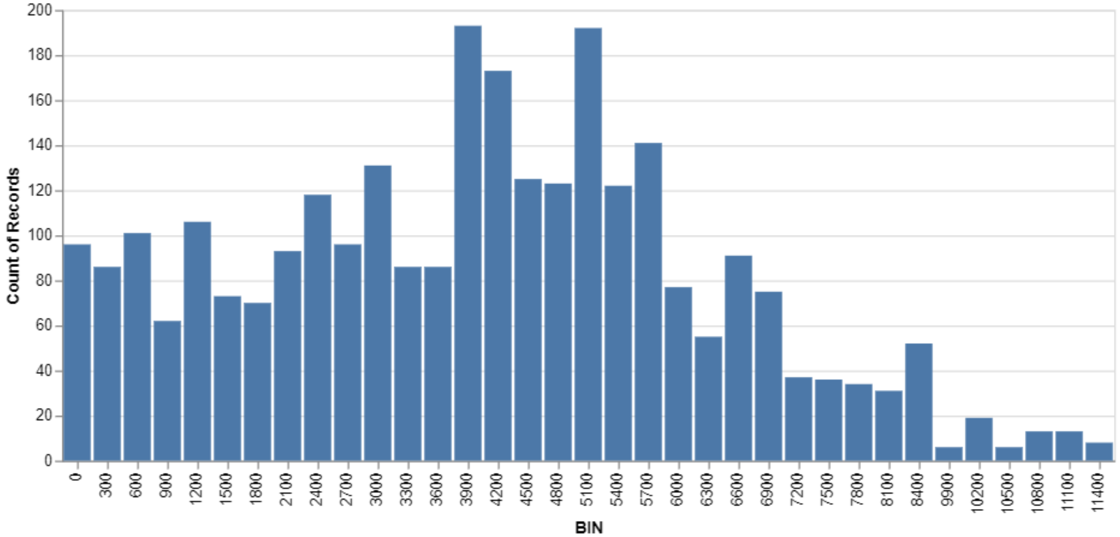
Out[34]:

	STREET_NO	BIN
78180	3030	3000
85920	4745	4500
72219	6703	6600
91336	3514	3300
102625	3741	3600

A sample of the relevant columns from the table would look something like [this (assets/binned_df.png)](assets/binned_df.png). We can also create a histogram of street numbers to see which are the most prevalent. It should look something like [this (assets/street_no.png)](assets/street_no.png).

In [35]:
```
def create_street_histogram_vis(visf):
    # create this vis
    return alt.Chart(binned_df).mark_bar().encode(
        alt.X('BIN'),
        alt.Y('count()')
    )

create_street_histogram_vis(bin_crashes(crashes_pulaski))
```

Out[35]:



In [36]:
```
def calculate_group_aggregates(df):
    """
    There are *accidents* and *injuries* (could be 0 people got hurt, could be more).
    There’s one row per accident at the moment, so we want to know how many accidents
    happened in each BIN/STREET_DIRECTION (this will be the count) and how many injuries (which will be the sum).

    Return a df with the count of accidents in a column named 'ACCIDENT_COUNT' (how many accidents happened in each
    bin (the count) and how many injuries (the sum) in a column named 'INJURIES_SUM'

    Replace NaN with 0
    """
    # YOUR CODE HERE

    # group by BIN and STREET_DIRECTION columns, get accident count and injury sum
    acc_inj_df=df.groupby(["BIN", "STREET_DIRECTION"]).agg({'RD_NO': 'count', 'INJURIES_TOTAL': 'sum'}).reset_index()

    # rename column names to match those specified in the question
    acc_inj_df.rename(columns={"RD_NO":"ACCIDENT_COUNT", "INJURIES_TOTAL":"INJURIES_SUM"}, inplace=True)

    return acc_inj_df

#     raise NotImplementedError()
```

In [37]:
```
aggregates = calculate_group_aggregates(binned_df)

# check the data
#aggregates.head(15)

aggregates.sample(15)
```

Out[37]:

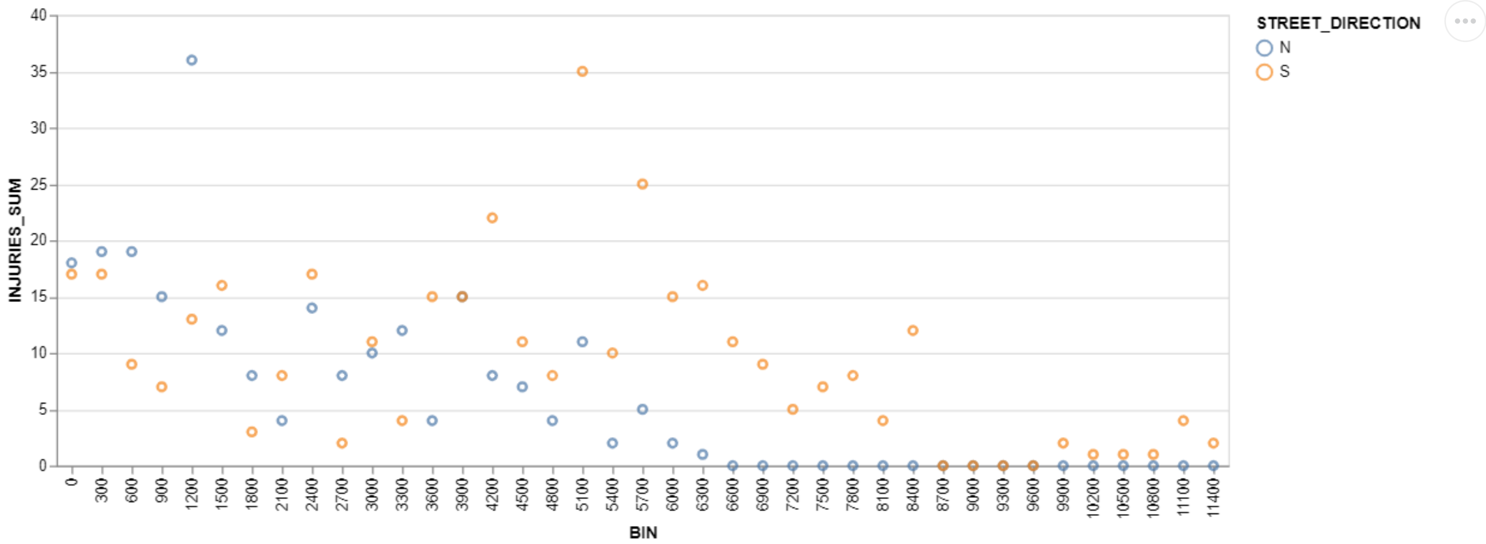
	BIN	STREET_DIRECTION	ACCIDENT_COUNT	INJURIES_SUM
48	7200	N	0	0.0
8	1200	N	76	36.0
69	10200	S	19	1.0
57	8400	S	52	12.0
38	5700	N	40	5.0
9	1200	S	30	13.0
60	9000	N	0	0.0
21	3000	S	44	11.0
47	6900	S	75	9.0
63	9300	S	0	0.0
11	1500	S	31	16.0
55	8100	S	31	4.0
31	4500	S	84	11.0
4	600	N	70	19.0
32	4800	N	44	4.0

The table should look like [this (assets/2.6_aggregate_1.png)](assets/2.6_aggregate_1.png)

Just for fun, here's a plot of injuries in the North and South directions based on bin. This may also help you debug your code. Depending on whether you removed N/A or if you hardcoded things, you may see slight differences. Here's what it might [look like (assets/direction_injuries.png)](assets/direction_injuries.png)

In [38]:
```
def create_injuries_sum_chart_vis(visdf):
    return alt.Chart(visdf).mark_point().encode(
        alt.Color('STREET_DIRECTION'),
        alt.X('BIN'),
        alt.Y('INJURIES_SUM')
    )

create_injuries_sum_chart_vis(aggregates)
```

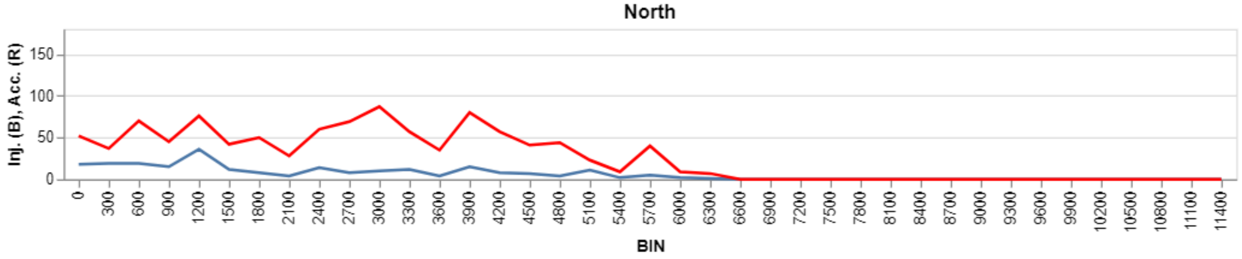
Out[38]:

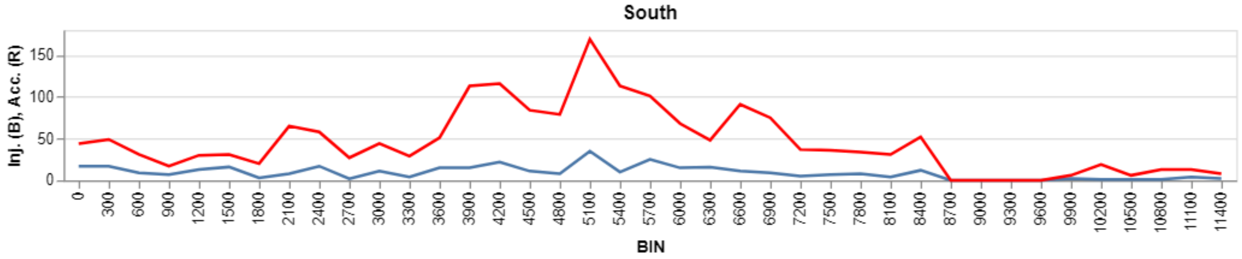


In [39]:

```python
# we can also look at the differences between injuries and accidents for a direction. We can plot
# both directions so you can see the difference

def create_injuries_vs_accident_vis(visdf,chart_title):
    c1 = alt.Chart(visdf).mark_line().encode(
        alt.X('BIN'),
        alt.Y('INJURIES_SUM',scale=alt.Scale(domain=(0, 170)), title='Inj. (B)')
    )

    c2 = c1.mark_line(color='red').encode(
        alt.Y('ACCIDENT_COUNT',scale=alt.Scale(domain=(0, 170)), title='Acc. (R)')
    )
    return (c1+c2).properties(title=chart_title,height=100)

def create_compound_i_vs_a_vis(visdf):
    north = create_injuries_vs_accident_vis(aggregates[aggregates.STREET_DIRECTION == "N"],"North")
    south = create_injuries_vs_accident_vis(aggregates[aggregates.STREET_DIRECTION == "S"],"South")
    return north & south

create_compound_i_vs_a_vis(aggregates)
```

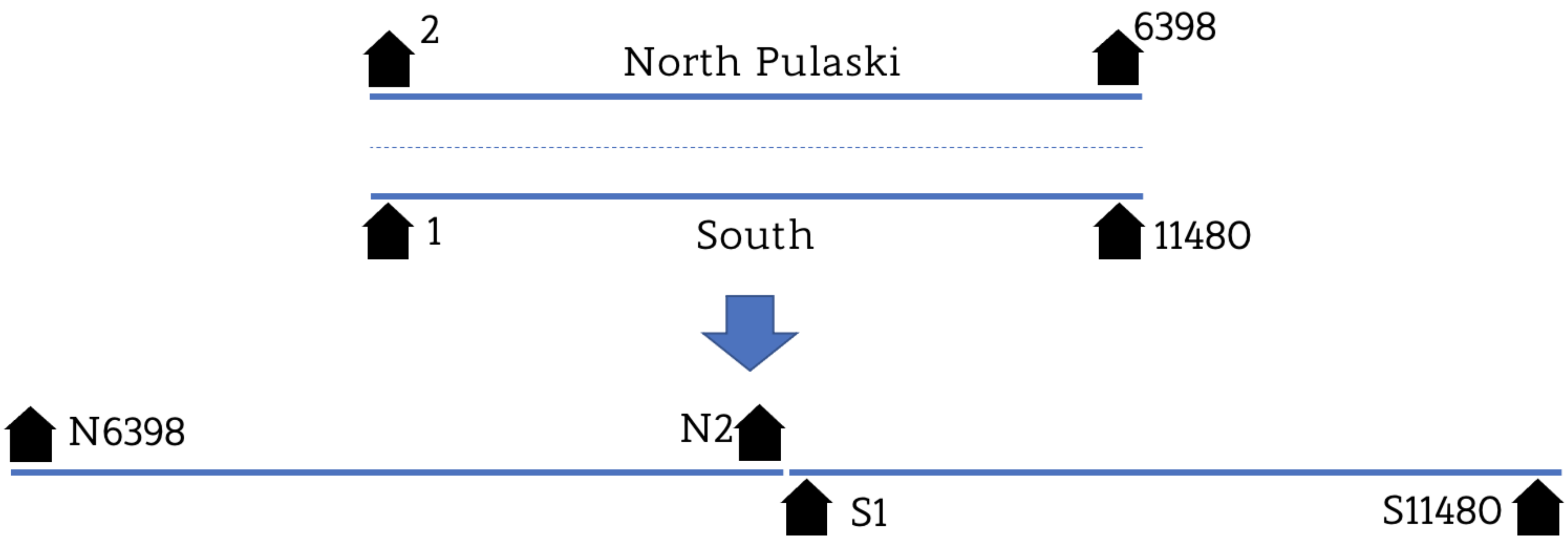
Out[39]:





2.7 Sort the street ranges (10 points)

Because the street has both North and South addresses we are going to "stretch" it so the bins range from the highest North street value down to the lowest and then going from lowest South to highest South. Something like this (but we're going to used the binned values, instead of the "raw" house numbers, in the end):



Altair will use the sort order in the dataframe so if we sort the frame this way, that's what we'll have.

- Sort the dataframe so North streets are in descending order and South streets are in ascending order
- You are provided with a 'pulaski_sort_order' arrray that contains this desired order. Use a categorical (pd.Categorial) column to order the dataframe according to this array.

In [40]:
```
# pulaski_sort_order will be a useful way for you to bin
crashed_range = list(range(0, crashes_pulaski.STREET_NO.max()+1000, 300))
pulaski_sort_order = ['N ' + str(s) for s in crashed_range[::-1]] + ['S ' + str(s) for s in crashed_range]
```

In [41]:
```
def categorical_sorting(df, sorder):
    """ Create a column called ORDER_LABEL that contains a concatenation of the street direction and the street range
    Set the sort order of this column to the provided sort array (sorder: the elements of this column should be in
    the same order of the array, we can pass in pulaski_sort_order as below)
    Sort the dataframe (df) by this column
    """
    # YOUR CODE HERE

    # import the CategoricalDtype functionality
    from pandas.api.types import CategoricalDtype

    # create the ORDER_LABEL column based on the specifications in the question
    df["ORDER_LABEL"]=df["STREET_DIRECTION"]+" "+df["BIN"].astype("str")

    # establish order of input categories
    categorical_sort_order = CategoricalDtype(pulaski_sort_order, ordered=True)

    # convert data type of ORDER_LABEL column to custom category
    df["ORDER_LABEL"]=df["ORDER_LABEL"].astype(categorical_sort_order)

    # sort values based on the ORDER_LABEL column
    df.sort_values("ORDER_LABEL", inplace=True)

    return df

#     raise NotImplementedError()
```

In [42]:
```
sorted_groups = categorical_sorting(aggregates, pulaski_sort_order)

# check the values
sorted_groups.sample(15)
```

Out[42]:

	BIN	STREET_DIRECTION	ACCIDENT_COUNT	INJURIES_SUM	ORDER_LABEL
2	300	N	37	19.0	N 300
40	6000	N	9	2.0	N 6000
3	300	S	49	17.0	S 300
34	5100	N	23	11.0	N 5100
21	3000	S	44	11.0	S 3000
41	6000	S	68	15.0	S 6000
38	5700	N	40	5.0	N 5700
6	900	N	45	15.0	N 900
69	10200	S	19	1.0	S 10200
24	3600	N	35	4.0	N 3600
70	10500	N	0	0.0	N 10500
63	9300	S	0	0.0	S 9300
64	9600	N	0	0.0	N 9600
62	9300	N	0	0.0	N 9300
18	2700	N	69	8.0	N 2700

In [43]:
```
sorted_groups.head()
```

Out[43]:

	BIN	STREET_DIRECTION	ACCIDENT_COUNT	INJURIES_SUM	ORDER_LABEL
76	11400	N	0	0.0	N 11400
74	11100	N	0	0.0	N 11100
72	10800	N	0	0.0	N 10800
70	10500	N	0	0.0	N 10500
68	10200	N	0	0.0	N 10200

The table should look like [this (assets/sorted_groups.png)](assets/sorted_groups.png)

You can test your code a few ways. First, we gave you the sort order, so you know what the ORDER_LABEL of the first row should be:

```
assert sorted_groups['ORDER_LABEL'].iloc[0] == sort_order[1]
```

(it might be sort_order[0] depending on how you did the label)

You also know that the first item should be "greater" than the second, so you can test:

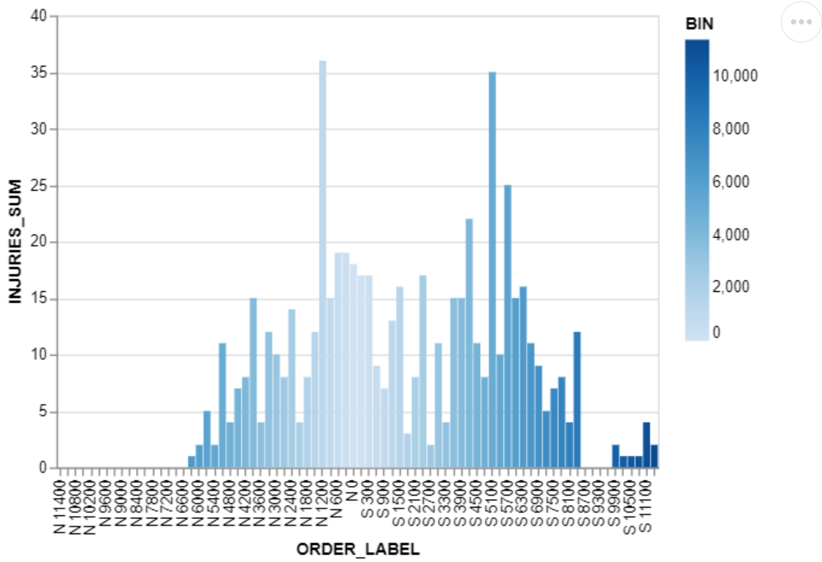
```
assert sorted_groups['ORDER_LABEL'].iloc[0] > sorted_groups['ORDER_LABEL'].iloc[1]
```

Again, just for kicks, let's see where injuries happen. We're going to color bars by the bin and preserve our ascending/descending visualization. We can probably imagine other (better) ways to visualize this data, but this may be useful for you to debug. The visualization should look something like [this (assets/order_injuries.png)](assets/order_injuries.png)

If your X axis cutoffs are a bit different, that's fine.

In [44]:
```
def create_sorted_pulaski_histogram_vis(visframe,sorder):
    # creates a histogram based on the calculated values in visframe
    # assumes an ORDER_LABEL, INJURIES_SUM, and BIN columns
    # color will double encode the bin value (which is the X)
    return alt.Chart(visframe).mark_bar().encode(
        alt.X('ORDER_LABEL:O', sort=sorder),
        alt.Y('INJURIES_SUM:Q'),
        alt.Color('BIN:Q')
    ).properties(
        width=400
    )

create_sorted_pulaski_histogram_vis(sorted_groups,pulaski_sort_order)
```

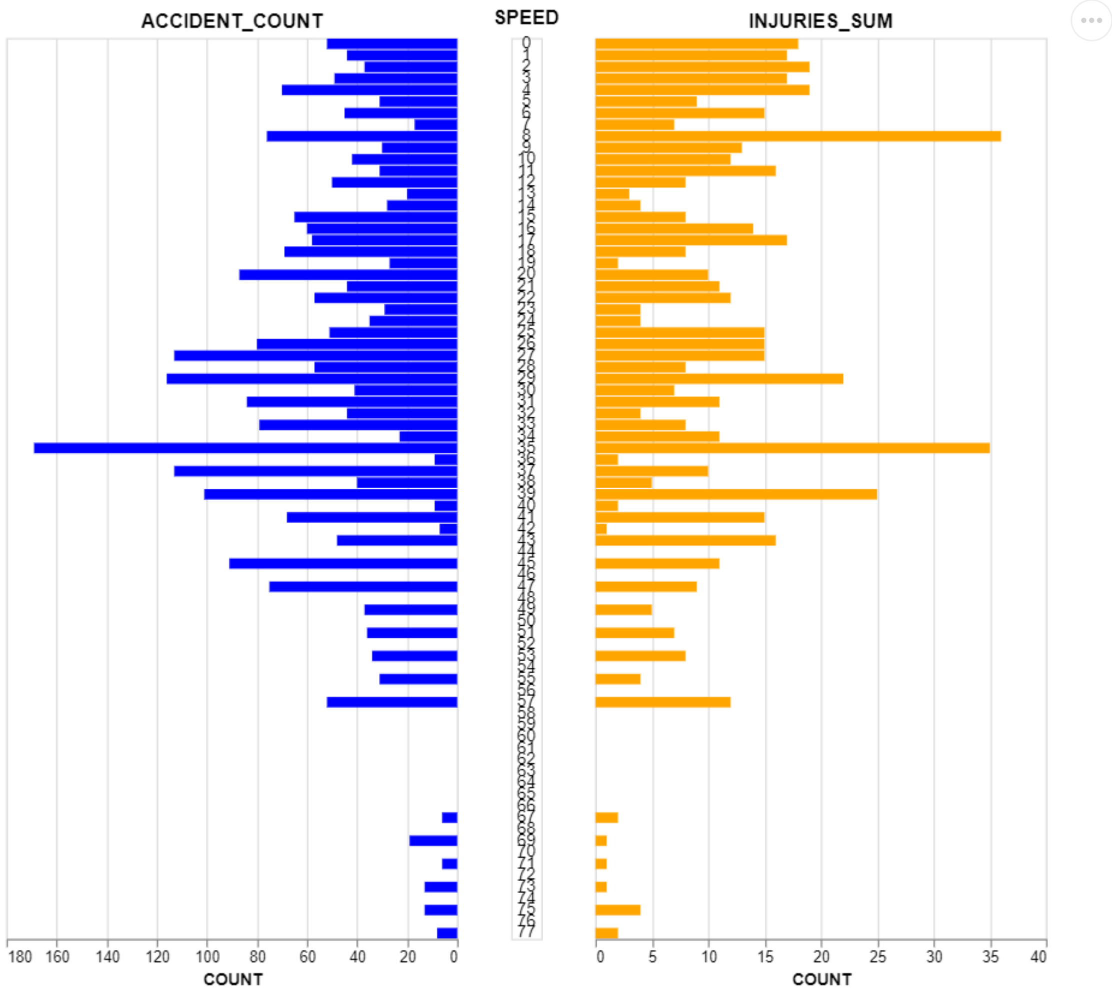
Out[44]:



Ok, let's actually make a useful visualization using some of the dataframes we've created. As a bonus, we're going to ask you what you would use this for.

In [45]:
```python
# to make the kind of chart we are interested in we're going to build it out of three different charts and
# put them together at the end

# this is going to be the left chart
bar_sorted_groups = sorted_groups[['ACCIDENT_COUNT','INJURIES_SUM']].unstack().reset_index() \
    .rename({'level_0':'TYPE','level_1':'SPEED',0:'COUNT'},axis=1)

# Note that we cheated a bit. The actual speed column (POSTED_SPEED) doesn't have enough variation for this
# example, so we're using the level_1 variable (it's an index variable) as a fake SPEED.
# Just assume this actually is the speed at which the accident happened.

a = alt.Chart(bar_sorted_groups).mark_bar().transform_filter(alt.datum.TYPE == 'ACCIDENT_COUNT').encode(
    x=alt.X('COUNT:Q',sort='descending'),
    y=alt.Y('SPEED:O',axis=None),
    color=alt.Color('TYPE:N',
                    legend=None,
                    scale=alt.Scale(domain=['ACCIDENT_COUNT', 'INJURIES_SUM'],
                                    range=['blue', 'orange']))
).properties(
    title='ACCIDENT_COUNT',
    width=300,
    height=600
)

# middle "chart" which actually won't be a chart, just a bunch of labels
b = alt.Chart(bar_sorted_groups).mark_bar().transform_filter(alt.datum.TYPE == 'ACCIDENT_COUNT').encode(
    y=alt.Y('SPEED:O', axis=None),
    text=alt.Text('SPEED:Q')
).mark_text().properties(title='SPEED',
                         width=20,
                         height=600)

# and the right most chart
c = alt.Chart(bar_sorted_groups).mark_bar().transform_filter(alt.datum.TYPE == 'INJURIES_SUM').encode(
    x='COUNT:Q',
    y=alt.Y('SPEED:O',axis=None),
    color=alt.Color('TYPE:N',
                    legend=None,
                    scale=alt.Scale(domain=['ACCIDENT_COUNT', 'INJURIES_SUM'],
                                    range=['blue', 'orange']))
).properties(
    title='INJURIES_SUM',
    width=300,
    height=600
)

# put them all together

a | b | c
```

Out[45]:



2.8 (Bonus) Accident barchart visualization (up to 2 points)

Looking at the visualization we generated above (part 2.7), what domain/abstract tasks are fulfilled by this visualization? List at least one domain task and the corresponding abstract task. See the comment in the code about "speed."

2.8 Answer

Domain Task: How do the number of accidents and injuries vary by speed?

Abstract Task: Find the sum of quantitative variables with respect to another quantitative variable.