Lab 1: Data Visualization Using Altair

Welcome to the first lab of DSC 106! In this lab, we'll learn about using the Altair Python library to create and customize plots.

About Labs in DSC 106

Every lab in DSC 106 will serve as a worked example. Labs will teach you how to implement visualizations using code by **giving you the code to copy** into your notebook or code editor. However, we strongly recommend that you avoid copy-pasting code directly and type the code yourself to increase the amount that you learn from these labs, since you will need the concepts from labs for your projects. To get credit for labs, there are no autograder tests. Instead, you'll show your completed lab to a TA during office hours, who will give you full credit if you've completed the entire lab. In other words, for the most part you're in charge of your own learning — welcome to the upper division!

Why Altair?

In DSC 10 and 80, you've used plotting tools like `matplotlib` and `plotly` to create and customize plots. These tools support rapid prototyping of data visualizations through single function calls like `plt.bar()`, or `px.scatter()`. In the first week of DSC 106, we'll learn about a *grammar of graphics* that enables us to generate all of these common visualizations as well as express new, custom-built visualizations.

Vega-Altair, or just Altair for short, is a modern Python library that expresses this grammar of graphics. It's a bit more verbose than `plotly` for basic plots, but makes it much easier to adjust, layer, and customize interactive visualizations.

Getting Started

For this lab, we'll be using DataHub[1]. Go to https://datahub.ucsd.edu/, then Log In. Start the server titled: `DSC106_WI24_A00 - Samuel Lau [WI24]`.

Once your notebook server has started, create a new Python 3 (ipykernel) notebook. (Note that choosing Python 3 (clean) will not allow you to install packages.)

[1] We know that we moved away from DataHub in DSC 80. But since we're going to switch to JavaScript programming next week, we didn't feel that it was worth it to create an entirely new conda environment just for this lab.

Installing Altair

To get started with Altair we first need to install the library. Copy this code into the first cell of your notebook.

```
!pip install -U altair==5.2.0 vega_datasets==0.9.0
```

Loading data

Next, copy this code into a new code cell to import the libraries we'll use. If the import code errors, restart your kernel and try again.

```
import pandas as pd
import altair as alt
from vega_datasets import data
```

Run the following code to ensure that `altair` is the correct version:

```
# Should output '5.2.0'
alt.__version__
```

The `vega_datasets` package has a variety of useful datasets preloaded. These datasets can be loaded directly into `pandas` dataframes. Copy the code below into your notebook to see an example.

```
data.list_datasets()
```

```
birdstrikes = data.birdstrikes() # This loads the data as a dataframe
```

Take a peek at the dataset:

```
birdstrikes.head()
```

Altair Fundamentals

Let's start by creating basic plots. In the process, we'll learn about fundamental concepts in `altair`.

Our first plot

The `Chart` object in `altair` takes a dataframe/dataset as an argument and has methods associated with it to generate plots that we would like to build. Copy over this code into a new code cell in your notebook to make a simple bar plot. Take note of how the `Chart` object is initialized and the subsequent `mark_bar` and `encode` calls.

```
df = pd.DataFrame({"x": ["A", "B", "C", "D", "E"], "y": [5, 3, 6, 7, 2]})
alt.Chart(df).mark_bar().encode(
   x="x",
   y="y",
)
```

Specifying data types

Notice how in the example above, `altair` was automatically able to detect that the x-axis plotted a categorical variable while the y-axis plotted a numeric variable. It did this by using the data types of the dataframe columns themselves (e.g. using `df.dtypes`).

But in many cases, we'll want to specify the data types more precisely by appending a type qualifier to the field. For example:

- `field_name:N` indicates a nominal type (unordered, categorical data),
- `field_name:O` indicates an ordinal type (rank-ordered data),
- `field_name:Q` indicates a quantitative type (numerical data with meaningful magnitudes), and
- `field_name:T` indicates a temporal type (date/time data)

Encodings and Marks

Now that we have a chart object, let's move on to the next concept - Encodings and marks.

Encodings

The key to creating meaningful visualizations is to map properties of the data to visual properties in order to effectively communicate information. In Altair, this mapping of visual properties to data columns is referred to as an encoding, and is most often expressed through

the Chart.encode() method. The encodings method can be used to signify the variable to plot onto the x-axis, y-axis, size of the points, color of points, etc. You can explore all the options that can be passed to this method in this link.

Marks

We can also choose how we would like to indicate the graphical mark using the mark_* methods. You can find the entire list of mark methods here. The mark_* methods determine the kind of plot you end up with, i.e., bar plot vs pie chart, etc.

Let's now make use of these two core concepts and build a slightly more complicated plot for this class.

For this plot, we will use the stocks dataset. Copy the following code into your notebook, and think about how each plot maps data points to marks in the plot.

```
stocks_df = data.stocks()
stocks_df.head()  # Take a peek at the dataset
```

```
alt.Chart(stocks_df).mark_point().encode(
    x="price",
    y="symbol",
)

# Altair also has an alternate syntax for encoding: alt.X and alt.Y.
# This syntax is useful when we need more customization
alt.Chart(stocks_df).mark_point().encode(
    alt.X("price"),
    alt.Y("symbol"),
)
```

Be prepared to show your TA the two plots below and explain why the second one is more informative than the first.

```
alt.Chart(stocks_df).mark_line().encode(
    x="date:T",
    y="price",
)
```

```
alt.Chart(stocks_df).mark_line().encode(
    x="date:T",
    y="price",
    color="symbol",
)
```

Data Transformation

Transforming data before plotting can be crucial. Typically, we prefer to perform our transformations (e.g. subsetting, filtering, grouping, etc.) using `pandas` before passing in the transformed data into `altair`. But sometimes, it's convenient to transform the data while constructing a plot using `altair`. This is especially useful for aggregating values, e.g. taking the mean. Copy over the following two code cells and compare the two plots.

```
alt.Chart(stocks_df).mark_point().encode(
    x=alt.X("price"),
    y=alt.Y("symbol"),
)
```

```
alt.Chart(stocks_df).mark_point().encode(
    x=alt.X("price").aggregate('mean'),
    y=alt.Y("symbol"),
)
```

Another use for aggregation is the common histogram, which can be thought of as an aggregation of 1-dimensional point plots. Copy over the following two code cells and be prepared to explain how the histogram aggregates points in the point plot together

```
msft = stocks_df.query('symbol == "MSFT" and date > 2005')

# point plot
alt.Chart(msft).mark_point().encode(
   alt.X("price"),
)
```

```
# histogram
alt.Chart(msft).mark_bar().encode(
   alt.X("price").bin(),
   y="count()",
)
```

There are a lot more things that we can do with data transformation, some useful links are provided below. We strongly encourage you to go through some of them as it might come in handy later on.

- https://altair-viz.github.io/user_guide/transform/index.html
- https://uwdata.github.io/visualization-curriculum/altair_data_transformation.html

- Mark methods: https://uwdata.github.io/visualization-curriculum/altair_marks_encoding.html

Customizing Plots

After prototyping a plot, we typically want to customize it to prepare it for publication. For example, we might want to change the axis titles or axis limits.

Let's start by changing the axis titles. Copy the following code into your notebook.

```
alt.Chart(msft).mark_bar().encode(
   alt.X("price").title("Price (USD)").bin(),
   alt.Y("count()").title("Number of days observed"),
)

# Notice the change in axis labels
```

We can also change the axis limits:

```
alt.Chart(msft).mark_bar().encode(
   (alt.X("price")
    .title("Price (USD)")
    .bin()
    .scale(domain=(10, 40))
   ),
   alt.Y("count()").title("Number of days observed"),
)
```

Of course, there are many more types of plot customizations possible. For a full list of all the capabilities refer the following links:

- Customizing Visualizations — Vega-Altair 5.2.0 documentation
- 4. Scales, Axes, and Legends — Visualization Curriculum
- Scale and Guide Resolution — Vega-Altair 5.2.0 documentation

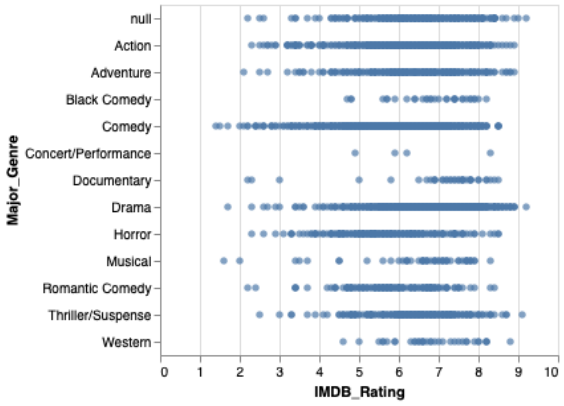
It's now time for some in-depth tasks to demonstrate the concepts we have learnt so far.

Task 1

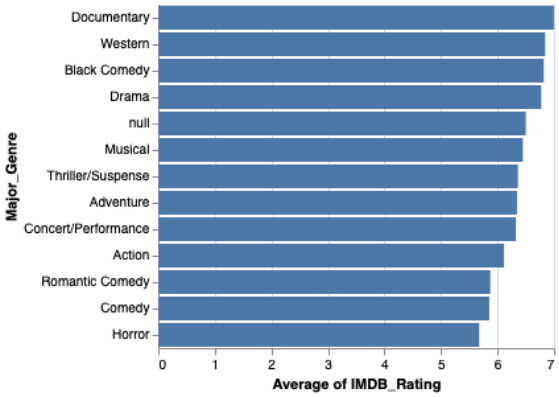
For this task we will use the movies dataset.

Things to do:

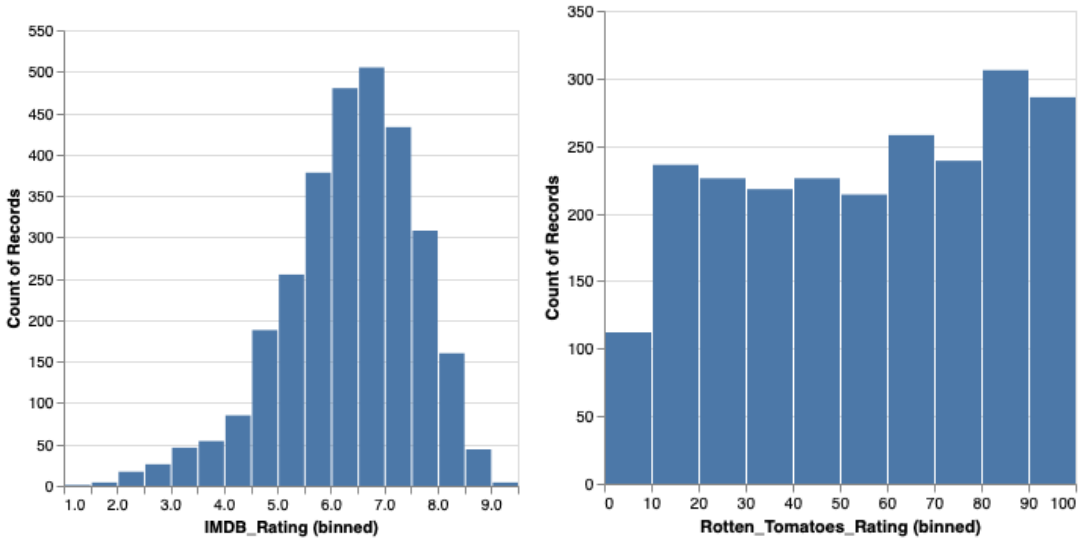
- Part 1: Point plot of the IMDB rating vs Major Genre:



- Part 2: Bar Plot of Average IMDB Rating



- Part 3: A distribution (bar charts) of ratings (IMDB and Rotten Tomatoes).



Let's start by loading the data.

```
movies_df = data.movies()
movies_df.head(2)  # take a peek at the data
```

Part 1: To begin looking at the distribution of IMDB ratings by genre, we might consider a basic point plot like the following:

```
alt.Chart(movies_df).mark_point().encode(
    alt.X("IMDB_Rating"),
    alt.Y("Major_Genre"),
)
```

Part 2: Note that this point plot makes it hard to directly compare ratings across genres. We could make this easier by averaging the ratings within each genre and using a bar plot:

```
ratings_by_genre = (
    movies_df
    .groupby('Major_Genre')
    ['IMDB_Rating']
    .mean()
    .sort_values()
    .reset_index()
)

alt.Chart(ratings_by_genre).mark_bar().encode(
   alt.X("IMDB_Rating"),
```

```
   alt.Y("Major_Genre").sort('-x'),
)
```

Part 3: To complete this task, we'll create histograms to show the distributions of IMDB and Rotten Tomatoes ratings using the following two code cells:

```
# Plot IMDB ratings in increments of 0.5
alt.Chart(movies_df).mark_bar().encode(
    alt.X("IMDB_Rating").bin(step=0.5),
    alt.Y("count()"),
)
```

```
# Plot Rotten Tomatoes Rating in increments of 10
alt.Chart(movies_df).mark_bar().encode(
    alt.X("Rotten_Tomatoes_Rating").bin(step=10),
    alt.Y("count()"),
)
```

Aside: Compound Charts

Altair allows us to combine information from several charts together. This can be done using the following operators / methods.

Operation	**Operator**	**Method**
Layering	+	alt.layer()
Horizontal Concatenation	|	alt.hconcat()
Vertical Concatenation	&	alt.vconcat()

Let now see some examples.

```
# Using the cars data:
cars = data.cars()
cars.head()
```

```
# Plotting average miles per gallon based on year of manufacture
alt.Chart(cars).mark_line().encode(
    alt.X("Year"),
    alt.Y("average(Miles_per_Gallon)"),
)
```

Now, let's layer the individual points on top of the line plot:

```
line = alt.Chart(cars).mark_line().encode(
    alt.X("Year"),
    alt.Y("average(Miles_per_Gallon)"),
)

points = alt.Chart(cars).mark_circle().encode(
    alt.X("Year"),
    alt.Y("average(Miles_per_Gallon)"),
)

line + points
```

We can also visualize the horsepower as a side-by-side plot:

```
hp = alt.Chart(cars).mark_line().encode(
    alt.X("Year"),
    alt.Y("average(Horsepower)"),
)

(
   (line + points) | (hp + hp.mark_circle())
)
```

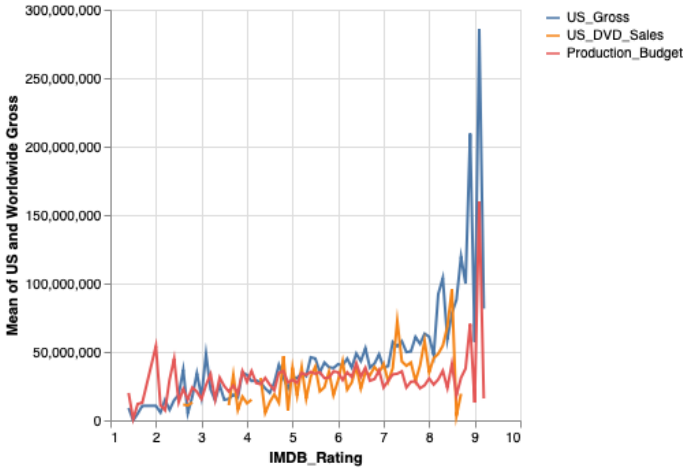
We'll use compound charts for Task 2.

Task 2

For this task we will continue using the movies dataset that we used for Task 1. We would like to do the following:

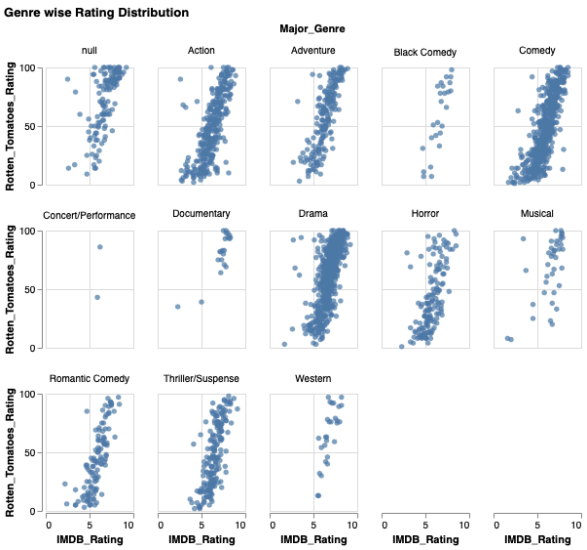
- See how the movie rating (IMDB) affects the gross collection in the US by plotting the US_Gross, US_DVD_Sales and Production_Budget with the IMDB_Rating. We would like to have the IMDB Rating on the X Axis and all the mentioned features on the Y Axis.

  Expected Plot:



- See how the rating distribution is for each Genre in a single plot. We would like to have IMDB on the X axis and Rotten Tomatoes on the Y Axis and generate a plot with subplots for each Genre.

  Expected Plot:



Part 1: We can use the `repeat` method to quickly layer plots of *multiple columns*:

```
alt.Chart(movies_df).mark_line().encode(
   alt.X("IMDB_Rating"),
   (alt.Y(alt.repeat("layer"))
    .aggregate("mean")
    .title("Mean of US and Worldwide Gross")
   ),
   alt.ColorDatum(alt.repeat("layer")),
).repeat(layer=["US_Gross", "US_DVD_Sales", "Production_Budget"])
```

Part 2: We can use the `facet` method to create separate plots for individual values within a *single column*.

```
alt.Chart(movies_df).mark_circle().encode(
   x="IMDB_Rating",
   y=alt.Y("Rotten_Tomatoes_Rating").axis(format="~s"),
   facet=alt.Facet("Major_Genre").columns(5),
).properties(title="Genre wise Rating Distribution", width=90, height=120)
```

What insights can you gather from these two plots?

Aside: Interactivity

It is possible to make the plots interactive by using the interactive method. Lets see how to do this

```
(line + points).interactive() | (
   hp + hp.mark_circle()
).interactive()  # the charts can be zoomed / panned separately.
```

```
(
   (line + points) | (hp + hp.mark_circle())
).interactive()  # the charts can be zoomed / panned simultaneously

# Notice how the parenthesis are structured differently
```

We can also create interactive tooltips:

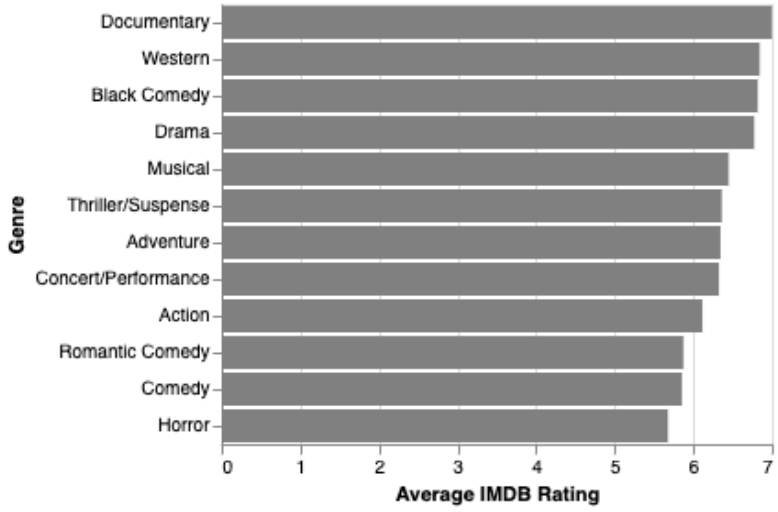
```
line = (
   alt.Chart(cars)
   .mark_line()
   .encode(
       alt.X("Year"),
       alt.Y("average(Miles_per_Gallon)"),
       tooltip=["Year", "average(Miles_per_Gallon)"],
   )
)

point = (
   alt.Chart(cars)
   .mark_circle()
   .encode(
       alt.X("Year"),
       alt.Y("average(Miles_per_Gallon)"),
       tooltip=["Year", "average(Miles_per_Gallon)"],
   )
)

hp = (
   alt.Chart(cars)
   .mark_line()
   .encode(alt.X("Year"),
alt.Y("average(Horsepower)"), tooltip=["Year", "average(Horsepower)"])
)

((line + point) | (hp + hp.mark_circle())).interactive()
```

You can find a comprehensive documentation of all the interactions that you can create using Altair here.

Task 3

Lets now play around with interactions and Plot Customizations. We'll recreate the bar plot of Average IMDB Rating, changing the bar colors to gray, and renaming the axis from "Major_Genre" to "Genre" and "Average of IMDB_Rating" to "Average IMDB Rating". Then, we'll add tooltips to show the exact average IMDB rating for each bar on hover.

Expected Plot:



Copy this code into your notebook:

```
alt.Chart(ratings_by_genre).mark_bar(color='gray').encode(
    alt.X("IMDB_Rating").title('Average IMDB Rating'),
    alt.Y("Major_Genre").sort('-x').title('Genre'),
    tooltip=["average(IMDB_Rating)"],
)
```

Exporting HTML

It is possible to export the chart as a HTML as well. Once the HTML is generated, take a peek at it, it might come in handy later in the course.

```
sample_chart = (
    alt.Chart(ratings_by_genre).mark_bar(color='gray').encode(
        alt.X("IMDB_Rating").title('Average IMDB Rating'),
        alt.Y("Major_Genre").sort('-x').title('Genre'),
        tooltip=["average(IMDB_Rating)"],
    )
)

sample_chart.save("chart.html")
```

Task 4

This task is an open ended task. You are free to choose any of the datasets that we have explored in this notebook and come up with an interesting visualization (limit yourself to 1 plot). Make sure that the visualization has the following characteristics:

- At least one aggregation
- At least one customization
- Keep it simple, but insightful. **Write one sentence** about why you think the visualization is interesting.

Checkoff

- [ ] Date - Price Plot Explanation

- Task 1
  - [ ] Scatter Plot
  - [ ] Distribution Plot
  - [ ] Bar Plot

- Task 2
  - [ ] Repeat Plot
  - [ ] Facet Plot

- Task 3
  - [ ] Customization
  - [ ] Interactivity

- Task 4
  - [ ] Aggregation
  - [ ] Customization
  - [ ] Insight