Professional Visualization & Introduction to Machine Learning

From Data to Insights to Predictions

Learning Objectives

By the end of this session, you will be able to:

- Create publication-quality visualizations for research and presentations
- Build interactive visualizations to explore complex datasets
- Understand how machine learning works and train your first model
- Choose the right visualization tool for your analysis needs

---

Part 1: The Anscombe Revelation - Why Visualization Matters

The Hidden Truth Demo

The Anscombe Quartet demonstrates a critical principle: four datasets with identical statistical properties (mean, variance, correlation, regression line) that represent completely different relationships. This code loads the built-in Anscombe dataset from Seaborn and creates a 2x2 subplot grid to reveal how Dataset I shows a linear relationship, Dataset II is perfectly quadratic, Dataset III has an outlier affecting the regression, and Dataset IV has no real relationship at all. We use `regplot()` to overlay the (misleading) regression lines that would be identical if calculated numerically.

Four datasets with identical statistics tell completely different stories:

```python
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np


# Load Anscombe's quartet
anscombe = sns.load_dataset("anscombe")
```

```
print("All four datasets have:")
print("• Same mean (x=9.0, y=7.5)")
print("• Same variance and correlation (0.816)")
print("• Same regression line: y = 0.5x + 3")

# Reveal the truth through visualization
fig, axes = plt.subplots(2, 2, figsize=(10, 8))
fig.suptitle("Same Statistics, Completely Different Stories", fontsize=16, fontweight='bold')

datasets = ['I', 'II', 'III', 'IV']
colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728']

for i, (dataset, color) in enumerate(zip(datasets, colors)):
    ax = axes[i//2, i%2]
    data = anscombe[anscombe['dataset'] == dataset]
    sns.scatterplot(data=data, x='x', y='y', ax=ax, s=100, color=color)
    sns.regplot(data=data, x='x', y='y', ax=ax, scatter=False, color='red')
    ax.set_title(f'Dataset {dataset}')
    ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

The Revelation:

- Dataset I: Normal linear relationship
- Dataset II: Perfect parabola (quadratic, not linear!)
- Dataset III: Perfect line with one outlier
- Dataset IV: No relationship - one point creates fake correlation

Real-world impact: Medical researchers might miss that a drug only works for certain age groups. Climate scientists could overlook critical changes. Your research might hide important patterns without visualization!

Success Check: You should see four completely different patterns despite identical statistics!

Part 2: Seaborn - Statistical Visualization Made Beautiful

Building on the Matplotlib skills from Chapter 15, Seaborn adds statistical intelligence to your plots. This section demonstrates six essential plot types using the `tips` dataset: histograms with KDE overlays for distribution analysis, regression plots with automatic confidence intervals, bar plots with error bars, box plots for quartile visualization, violin plots that combine distribution and statistics, and correlation heatmaps. Each uses Seaborn's built-in statistical calculations that would require many lines of NumPy code to replicate manually.

Your Statistical Toolkit

We load the tips dataset using `sns.load_dataset('tips')`, which fetches a DataFrame with 244 restaurant visits from Seaborn's data repository. This real dataset from the 1990s contains total_bill, tip, sex, smoker, day, time, and size columns. The `.load_dataset()` function handles downloading and caching, returning a pandas DataFrame ready for analysis. We'll use this throughout Part 2 to demonstrate Seaborn's statistical visualization capabilities.

python

```python
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np  # For numerical operations

# Load real restaurant data
# NOTE: This is REAL data collected from restaurant visits, not simulated!
tips = sns.load_dataset("tips")
print(f"Restaurant Dataset: {len(tips)} real restaurant visits")
print(f"Columns: {', '.join(tips.columns)}")
print("This is actual tipping data from the 1990s - real patterns from real people!")
```

Essential Statistical Plots

This code creates a 2x3 subplot figure demonstrating Seaborn's statistical capabilities. The `histplot()` with `kde=True` overlays a kernel density estimate on the histogram. The `regplot()` automatically calculates and displays confidence intervals. The `barplot()` computes means and 95% confidence intervals for each category. The `boxplot()` shows median, quartiles, and outliers. The `violinplot()` combines a box plot with a KDE. The `heatmap()` with `annot=True` displays correlation coefficients. Each plot type answers different statistical questions about the tips dataset.

python

```python
# Comprehensive statistical analysis
fig, axes = plt.subplots(2, 3, figsize=(15, 10))
fig.suptitle('Restaurant Tipping Patterns', fontsize=16, fontweight='bold')
```

```
# 1. Distribution with KDE
sns.histplot(data=tips, x="total_bill", kde=True, ax=axes[0,0])
axes[0,0].set_title("Bill Distribution")

# 2. Regression with confidence
sns.regplot(data=tips, x="total_bill", y="tip", ax=axes[0,1])
axes[0,1].set_title("Bill vs Tip Relationship")
corr = tips['total_bill'].corr(tips['tip'])
axes[0,1].text(0.05, 0.95, f'r = {corr:.3f}', transform=axes[0,1].transAxes)

# 3. Category comparisons
sns.barplot(data=tips, x="day", y="tip", ax=axes[0,2])
axes[0,2].set_title("Average Tips by Day")

# 4. Box plots
sns.boxplot(data=tips, x="time", y="total_bill", ax=axes[1,0])
axes[1,0].set_title("Lunch vs Dinner Bills")

# 5. Violin plot
sns.violinplot(data=tips, x="day", y="tip", ax=axes[1,1])
axes[1,1].set_title("Tip Distribution by Day")

# 6. Correlation heatmap
corr_matrix = tips[['total_bill', 'tip', 'size']].corr()
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', center=0, ax=axes[1,2])
axes[1,2].set_title("Correlations")

plt.tight_layout()
plt.show()
```

Key Insights:

- **Histogram**: Shows distribution shape (normal? skewed?)
- **Regression**: Reveals linear relationships with confidence intervals
- **Box plots**: Display quartiles and outliers
- **Violin plots**: Combine distribution shape with statistics
- **Heatmap**: Correlation strength at a glance

Part 3: Plotly - Interactive Exploration

Unlike the static plots from Matplotlib and Seaborn, Plotly creates HTML-based interactive visualizations. This section shows three key techniques: basic interactive scatter plots using `px.scatter()` with hover data and dynamic filtering, multi-subplot dashboards using `make_subplots()` to combine different chart types, and 3D visualization with `px.scatter_3d()` for exploring three continuous variables simultaneously. The interactivity happens automatically - Plotly adds zoom, pan, hover tooltips, and legend filtering without extra code.

From Static to Interactive

This code transforms a static scatter plot into an interactive web element. The `px.scatter()` function automatically generates HTML with JavaScript event handlers. The `hover_data` parameter adds columns that appear only on mouseover, reducing visual clutter. The `color` parameter creates a legend that becomes an interactive filter - clicking legend items toggles visibility of that category. The `size` parameter maps a continuous variable to point size, creating a bubble chart effect.

python

```python
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots

# Interactive scatter plot
fig = px.scatter(tips, x="total_bill", y="tip",
                 color="day", size="size",
                 hover_data=['sex', 'smoker', 'time'],
                 title="Interactive Tipping Explorer")

fig.update_layout(width=900, height=600)
fig.show()

print("Try: Hover for details | Click legend to filter | Drag to zoom")
```

Research Dashboard (Not Business-Focused)

This code uses `make_subplots()` to create a 2x2 grid combining different Plotly chart types. The `specs` parameter defines subplot types (histogram, box, scatter, violin). We iterate through categories to add multiple traces to single subplots, creating grouped visualizations. The `go.Histogram()`, `go.Box()`, `go.Scatter()`, and `go.Violin()` objects provide fine control over

appearance. The dashboard updates all at once with `fig.update_layout()`, maintaining consistent styling across subplots.

python

```python
# Multi-chart research dashboard
fig = make_subplots(
    rows=2, cols=2,
    subplot_titles=('Distribution', 'Statistical Comparison',
                    'Correlation Analysis', 'Pattern Discovery'),
    specs=[[{"type": "histogram"}, {"type": "box"}],
           [{"type": "scatter"}, {"type": "violin"}]]
)

# Add interactive charts
fig.add_trace(go.Histogram(x=tips['total_bill'], name="Bills"), row=1, col=1)

for day in ['Thur', 'Fri', 'Sat', 'Sun']:
    day_tips = tips[tips['day'] == day]['tip']
    fig.add_trace(go.Box(y=day_tips, name=day), row=1, col=2)

fig.add_trace(go.Scatter(x=tips['total_bill'], y=tips['tip'],
                         mode='markers', name="Data"), row=2, col=1)

for time in ['Lunch', 'Dinner']:
    time_data = tips[tips['time'] == time]['tip']
    fig.add_trace(go.Violin(y=time_data, name=time), row=2, col=2)

fig.update_layout(height=700, title_text="Statistical Research Dashboard")
fig.show()
```

3D Visualization

The `px.scatter_3d()` function maps three continuous variables to x, y, and z axes, with color as a fourth dimension. Plotly automatically adds 3D controls: click-and-drag rotates the plot, scroll zooms, and double-click resets the view. The `scene` parameter in `update_layout()` customizes axis labels and camera angle. This creates an interactive 3D scatter plot rendered using WebGL for smooth performance even with hundreds of points.

python

```
# Explore three dimensions at once
fig_3d = px.scatter_3d(tips, x="total_bill", y="tip", z="size",
                       color="day", title="3D Analysis: Bill × Tip × Party Size")

fig_3d.update_layout(width=900, height=600)
fig_3d.show()

print("Click and drag to rotate | Scroll to zoom")
```

Part 4: Machine Learning - Teaching Computers to Recognize Patterns

This section implements supervised learning using scikit-learn, the same library you'll see in industry. We'll compare two algorithms: Linear Regression (which fits a plane through the data) and Random Forest (which builds 100 decision trees and averages their predictions). The code follows the standard ML workflow: prepare features (X) and target (y), split data 80/20 for training/testing using `train_test_split()`, fit models with `.fit()`, make predictions with `.predict()`, and evaluate using R^2 score (variance explained) and MAE (average dollar error).

Understanding Machine Learning

The Concept: Machine learning is like teaching someone to recognize patterns through examples.

Imagine teaching a friend to predict tips:

1. Show them 100 past restaurant receipts (training)
2. They notice patterns: bigger bills = bigger tips
3. Now they can predict tips for new customers

Three Types of Machine Learning

This code simply prints the three categories to establish context. Supervised learning uses labeled data (we know the correct answers for training examples). Unsupervised learning finds patterns without labels using techniques like clustering. Reinforcement learning optimizes actions through reward signals. We focus on supervised learning because it's most common in data science - the `sklearn.supervised` module contains dozens of algorithms following the same fit/predict pattern we'll use.

python

```
print("MACHINE LEARNING TYPES")
```

```
print("1. SUPERVISED (today's focus): Learn from labeled examples")
print("   Example: Bill amount → Tip amount")
print("2. UNSUPERVISED: Find hidden patterns")
print("   Example: Grouping similar customers")
print("3. REINFORCEMENT: Learn through trial and error")
print("   Example: Learning to play games")
```

Building Your First Prediction Model

This implementation compares Linear Regression and Random Forest regressors on the same prediction task. The code uses `train_test_split()` with `test_size=0.2` to hold out 20% of data for testing, ensuring we evaluate on unseen examples. The models dictionary allows iterating through different algorithms with identical workflows. We evaluate using R^2 (proportion of variance explained, where 1.0 is perfect) and MAE (mean absolute error in dollars). The Random Forest's `feature_importances_` attribute reveals which inputs most influence predictions.

python

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression  # For linear model
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import r2_score, mean_absolute_error

print("MACHINE LEARNING WORKFLOW")
print("1. Prepare Data → 2. Split → 3. Train → 4. Predict → 5. Evaluate")

# Step 1: Prepare data
X = tips[['total_bill', 'size']].values  # Features
y = tips['tip'].values                   # Target

print(f"\nDataset: {len(X)} restaurant visits")

# Step 2: Split data (like flashcards for studying)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
print(f"Training: {len(X_train)} | Testing: {len(X_test)}")

# Step 3: Train models
```

```
models = {
    'Linear Regression': LinearRegression(),
    'Random Forest': RandomForestRegressor(n_estimators=100, random_state=42)
}

for name, model in models.items():
    # Train
    model.fit(X_train, y_train)

    # Predict
    predictions = model.predict(X_test)

    # Evaluate
    r2 = r2_score(y_test, predictions)
    mae = mean_absolute_error(y_test, predictions)

    print(f"\n{name}:")
    print(f"  R² Score: {r2:.3f} ({r2*100:.1f}% variance explained)")
    print(f"  Average Error: ${mae:.2f}")
```

Understanding Model Performance

These visualizations reveal how well our model works. The scatter plot compares actual vs predicted values - perfect predictions would fall on the red diagonal line. Points above the line indicate over-prediction, below means under-prediction. The feature importance bar chart, extracted from the Random Forest's `feature_importances_` attribute, shows which inputs the model relies on most. We use horizontal bars with percentage labels for clarity.

python

```
import matplotlib.pyplot as plt  # For visualizations

# Visualize predictions
best_model = models['Random Forest']
predictions = best_model.predict(X_test)

plt.figure(figsize=(10, 5))

# Plot 1: Predictions vs Actual
```

```
plt.subplot(1, 2, 1)
plt.scatter(y_test, predictions, alpha=0.6)
plt.plot([0, 10], [0, 10], 'r--', label='Perfect Prediction')
plt.xlabel('Actual Tip ($)')
plt.ylabel('Predicted Tip ($)')
plt.title('Prediction Accuracy')
plt.legend()

# Plot 2: Feature Importance
plt.subplot(1, 2, 2)
importance = best_model.feature_importances_
features = ['Total Bill', 'Party Size']
plt.barh(features, importance)
plt.xlabel('Importance')
plt.title('What Determines Tips?')

plt.tight_layout()
plt.show()

# Make predictions for new customers
new_customers = [[25, 2], [75, 6], [15, 1]]
for bill, party in new_customers:
    tip = best_model.predict([[bill, party]])[0]
    print(f"Bill: ${bill}, Party of {party} → Predicted tip: ${tip:.2f}")
```

Success Check: Your model should achieve 70-90% accuracy!

Part 5: Hands-On Workshop

This workshop integrates all three libraries on a single dataset. We generate 300 students with correlated features (study hours affect GPA, stress affects satisfaction) using NumPy's random distributions with controlled relationships. You'll create static visualizations with Seaborn, interactive explorers with Plotly, and predictive models with scikit-learn - the same workflow used in real data science projects. The simulated data ensures reproducibility (everyone debugs the same issues) while teaching techniques that work identically on real datasets.

Important Note About This Workshop: We're using simulated (randomly generated) data to learn these visualization and machine learning tools. Why?

- **Consistency**: Everyone gets the same results, making it easier to help each other
- **Control**: We know what patterns should exist, so we can verify our tools work
- **Safety**: No privacy concerns while learning
- **Focus**: Learn the techniques without getting distracted by data cleaning issues

Remember: These exact same tools and techniques will work on real data. In your upcoming projects, you'll apply everything you learn here to actual datasets from research, APIs, or your field of study!

Creating Your Campus Analytics Platform

This code generates a 300-student dataset using NumPy's random distributions. We use `np.random.gamma()` for right-skewed distributions (study hours), `np.random.normal()` for bell curves (sleep), and `np.random.choice()` for categories. The relationships between variables are created through linear combinations: `stress_level` increases with study hours (multiplied by 0.15), and `gpa` increases with both study hours (0.03 coefficient) and sleep (0.1 coefficient). The `.clip()` method ensures realistic bounds, and `round(2)` makes values human-readable.

python

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import plotly.express as px

# IMPORTANT NOTE: We're using simulated data to learn these tools safely and consistently.
# Everyone gets the same results, and we can focus on learning the techniques.
# In your projects and future work, you'll apply these exact same tools to REAL data!

print("NOTE: Using simulated data for learning purposes")
print("   The tools and techniques you learn here work exactly the same on real data!")
print("   Your next project will use real datasets from actual research/sources.\n")

# Generate campus dataset
np.random.seed(42)  # This ensures everyone gets the same "random" data
n_students = 300

campus_data = pd.DataFrame({
```

```
    'student_id': range(1, n_students + 1),
    'year': np.random.choice(['Freshman', 'Sophomore', 'Junior', 'Senior'], n_students),
    'major': np.random.choice(['STEM', 'Humanities', 'Business', 'Arts'], n_students),
    'hours_studying': np.random.gamma(2, 8, n_students).clip(5, 50),
    'hours_socializing': np.random.gamma(1.5, 4, n_students).clip(2, 25),
    'hours_sleeping': np.random.normal(7, 1.5, n_students).clip(4, 11),
    'campus_involvement': np.random.choice(['Low', 'Medium', 'High'], n_students),
})

# Create realistic relationships
campus_data['stress_level'] = (
    campus_data['hours_studying'] * 0.15 +
    np.random.normal(4, 1.5, n_students)
).clip(1, 10)

campus_data['gpa'] = (
    2.0 + campus_data['hours_studying'] * 0.03 +
    campus_data['hours_sleeping'] * 0.1 +
    np.random.normal(0, 0.3, n_students)
).clip(1.0, 4.0)

campus_data['satisfaction'] = (
    10 - campus_data['stress_level'] * 0.5 +
    campus_data['hours_socializing'] * 0.1 +
    np.random.normal(0, 1, n_students)
).clip(1, 10)

# Round for realism
for col in ['hours_studying', 'hours_socializing', 'hours_sleeping',
            'stress_level', 'gpa', 'satisfaction']:
    campus_data[col] = campus_data[col].round(2)

print(f"Dataset created: {len(campus_data)} students")
```

Lab 1: Statistical Analysis

This code creates a 2x2 grid analyzing campus_data from multiple statistical angles. The `violinplot()` shows GPA distributions across majors with density shapes. The scatter plot with `regplot()` overlay demonstrates correlation between study hours and GPA, with the correlation coefficient calculated using `.corr()`. The `boxplot()` reveals how campus involvement relates to stress levels. The `heatmap()` displays a correlation matrix with `mask=np.triu()` to show only the lower triangle, avoiding redundancy.

python

```python
# Create analysis dashboard
fig = plt.figure(figsize=(12, 8))
fig.suptitle('Campus Life Analysis', fontsize=16, fontweight='bold')

# GPA by Major
ax1 = plt.subplot(2, 2, 1)
sns.violinplot(data=campus_data, x='major', y='gpa', ax=ax1)
ax1.set_title('GPA Distribution by Major')

# Study vs GPA
ax2 = plt.subplot(2, 2, 2)
sns.scatterplot(data=campus_data, x='hours_studying', y='gpa',
                hue='year', alpha=0.6, ax=ax2)
sns.regplot(data=campus_data, x='hours_studying', y='gpa',
            scatter=False, color='red', ax=ax2)
ax2.set_title('Study Hours Impact')

# Stress Analysis
ax3 = plt.subplot(2, 2, 3)
sns.boxplot(data=campus_data, x='campus_involvement', y='stress_level', ax=ax3)
ax3.set_title('Stress by Involvement')

# Correlations
ax4 = plt.subplot(2, 2, 4)
corr = campus_data[['hours_studying', 'hours_sleeping', 'gpa', 'satisfaction']].corr()
sns.heatmap(corr, annot=True, cmap='RdBu_r', center=0, ax=ax4)
ax4.set_title('Key Correlations')

plt.tight_layout()
plt.show()
```

Lab 2: Interactive Explorer

This code demonstrates Plotly's interactivity features. The first plot uses `px.scatter()` with multiple aesthetic mappings (color for major, size for satisfaction) and `hover_data` to show additional variables on mouseover. The `add_hline()` and `add_vline()` methods add reference lines to create quadrants. The second plot uses `animation_frame` to create an animated visualization showing stress changes across 15 weeks, simulating midterm and finals peaks using conditional multipliers.

python

```
# Multi-dimensional scatter
fig1 = px.scatter(campus_data,
                  x='hours_studying', y='gpa',
                  color='major', size='satisfaction',
                  hover_data=['stress_level', 'year'],
                  title='Student Performance Explorer')

# Add balance zones
fig1.add_hline(y=3.0, line_dash="dash", opacity=0.5)
fig1.add_vline(x=20, line_dash="dash", opacity=0.5)

fig1.show()

# Animated semester progression
weeks = list(range(1, 16))
animated_data = []

for week in weeks:
    week_data = campus_data.copy()
    # Stress peaks at midterms (week 8) and finals (week 15)
    stress_mult = 1.5 if week in [8, 15] else 1.0
    week_data['current_stress'] = (campus_data['stress_level'] * stress_mult).clip(1, 10)
    week_data['week'] = f'Week {week}'
    animated_data.append(week_data)

semester_data = pd.concat(animated_data)

fig2 = px.scatter(semester_data, x='current_stress', y='satisfaction',
```

```
            color='major', animation_frame='week',
            title='Semester Stress Journey',
            range_x=[0, 11], range_y=[0, 11])
fig2.show()
```

Lab 3: Machine Learning for Student Success

This implementation uses RandomForestRegressor with 100 trees to predict GPA from four features. The model's `feature_importances_` attribute returns an array showing each feature's contribution to predictions, which we sort and display as percentages. The prediction section demonstrates the trained model's `.predict()` method on new data, using list comprehension to test three student archetypes. The R^2 score indicates the proportion of GPA variance our model explains.

python

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score

# Prepare features
X = campus_data[['hours_studying', 'hours_sleeping', 'hours_socializing', 'stress_level']]
y = campus_data['gpa']

# Train model
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
model = RandomForestRegressor(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Evaluate
predictions = model.predict(X_test)
r2 = r2_score(y_test, predictions)
print(f"Model R² Score: {r2:.3f}")

# Feature importance
importance = pd.DataFrame({
    'feature': X.columns,
    'importance': model.feature_importances_
}).sort_values('importance', ascending=False)
```

```
print("\nWhat determines GPA?")
for _, row in importance.iterrows():
    print(f"  {row['feature']}: {row['importance']*100:.1f}%")

# Predict for different student types
profiles = {
    'Night Owl': [30, 5, 10, 8],      # Study lots, sleep little
    'Balanced': [20, 8, 15, 5],       # Balanced approach
    'Social Butterfly': [10, 7, 30, 4] # Social focus
}

print("\nGPA Predictions:")
for name, values in profiles.items():
    gpa = model.predict([values])[0]
    print(f"  {name}: {gpa:.2f}")
```

Summary: Your New Data Science Powers

What You've Mastered

Statistical Visualization (Seaborn)

- Publication-ready plots
- Reveal hidden patterns
- Professional styling

Interactive Exploration (Plotly)

- Research dashboards
- 3D visualizations
- Animated data stories

Machine Learning Basics

- Train predictive models
- Evaluate performance
- Make predictions

When to Use Each Tool

Goal	Tool	Why
Statistical analysis	Seaborn	Beautiful statistical plots
Interactive exploration	Plotly	User-driven discovery
Predictions	Scikit-learn	Machine learning power

Quick Reference

python

```
# SEABORN
sns.violinplot(x='category', y='value')
sns.regplot(x='x', y='y')
sns.heatmap(corr_matrix, annot=True)

# PLOTLY
fig = px.scatter(df, x='x', y='y', animation_frame='time')
fig.show()

# MACHINE LEARNING
from sklearn.ensemble import RandomForestRegressor
X_train, X_test, y_train, y_test = train_test_split(X, y)
model = RandomForestRegressor()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
```

Remember: You're revealing truths hidden in data!