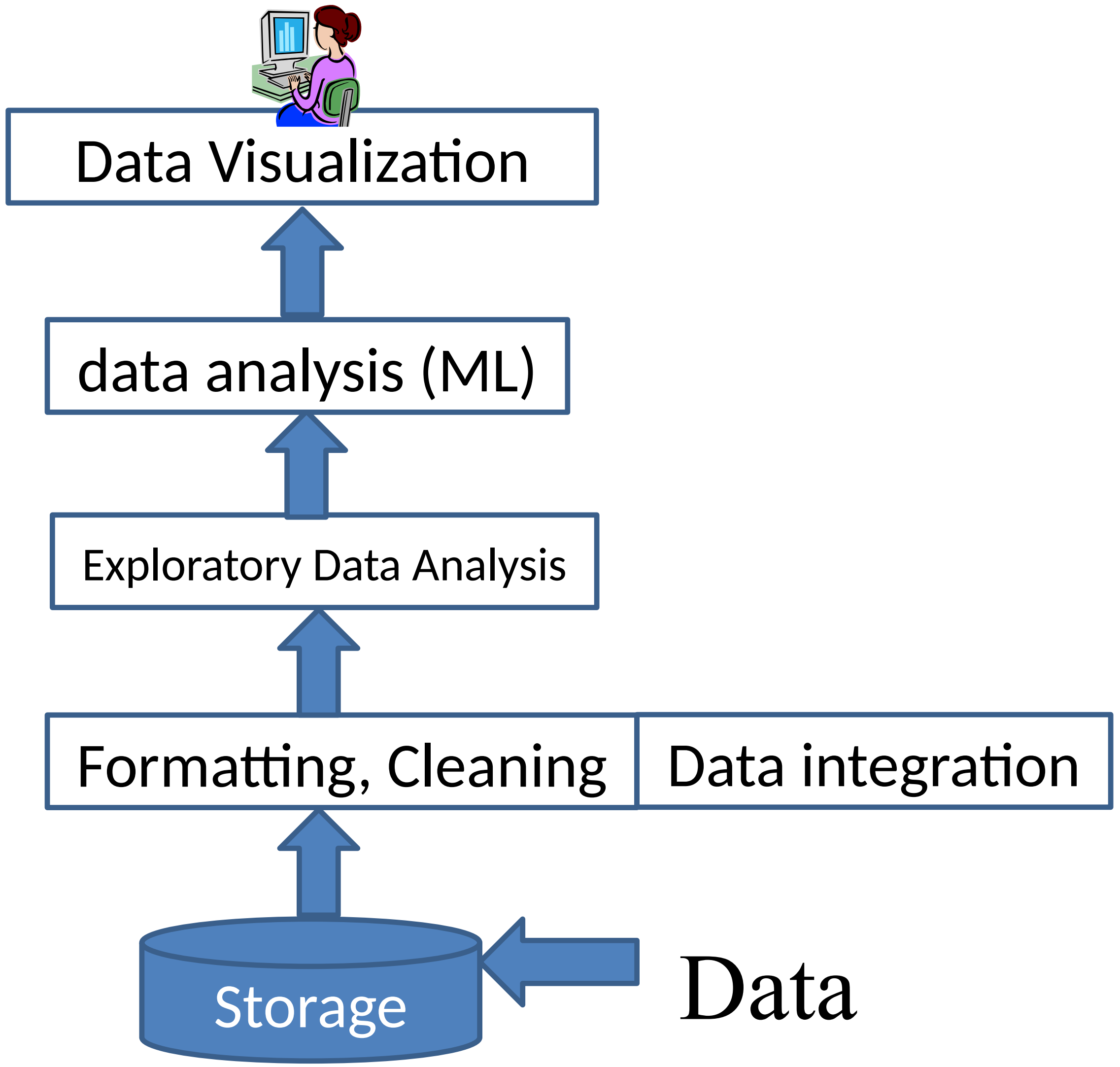
# CPSC 483
# Machine Learning (ML) Pipeline

Anand Panangadan
apanangadan@fullerton.edu

## What we will cover today

- Introductions
- What is Machine Learning?
- Activity: some issues to think about
- Types of ML approaches
- A ML project example
- Course Outline (syllabus overview)

# The Data Science Process



## Exploratory Data Analysis (EDA)

- First step to building a model
- Data-driven (model-free)
- Goal:  get a general sense of the data
  - Getting your hands "dirty" with the data
- Interactive and visual
  - Humans are good at pattern recognition

## Why EDA?

- To gain intuition about the data
- Sanity checking
  - Is the data as expected?
  - Are the ranges of variables as expected?
- detect outliers     (e.g. assess data quality)
- test assumptions (e.g. normal distributions or skewed?)
- identify useful data features & transforms (e.g. log(x))
- Is there missing data?
- What is the scale of the data?
  - Is it Big Data?

# Data pipeline

- A sequence of data processing components is called a *data pipeline*
- Pipeline components run *asynchronously*
  - Each component reads data, processes it, and outputs the result in another data store.
  - Another pipeline reads this data (at some later time) and produces its own output.
  - Each component is self-contained: the interface between components is the data store.
  - Different components can be developed and maintained independently.
  - If a component breaks down, the downstream components can often continue to run normally (at least for a while) by just using the last output from the broken component.
- Robust architecture
- Cons: a broken component can go unnoticed. The data gets stale and the overall system's performance drops.

## Main steps

- Look at the big picture
- Get the data
- Explore and visualize the data to gain insights
- Prepare the data for ML algorithms
- Select a model and train it
- Fine-tune your model
- Present your solution
- Launch, monitor, and maintain your system

# Data repositories

- OpenML.org
- Kaggle.com datasets
- PapersWithCode.com
- UC Irvine Machine Learning Repository
- Amazon’s AWS datasets
- TensorFlow datasets
- *Meta*-portals (lists of open data repositories):
  - DataPortals.org
  - OpenDataMonitor.eu
  - Data.gov
- Others …
  - Wikipedia’s list of machine learning datasets
  - Quora.com
  - The datasets subreddit

## Look at the big picture

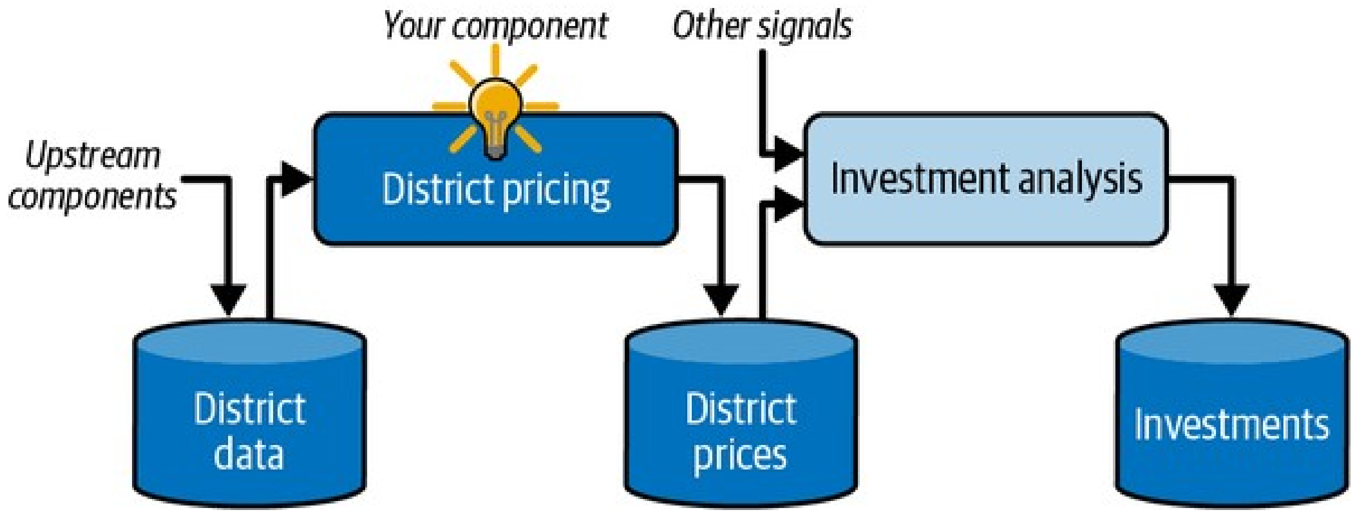
- Understand the context of the problem
  - Housing example

*"Use California census data to build a model of housing prices in the state. This data includes features such as the population, median income, and median housing price for each block group in California. Block groups ("districts") are the smallest geographical unit for which the US Census Bureau publishes sample data. A block group typically has a population of 600 to 3,000 people. The model should learn from this data and be able to predict the median housing price in any district, given all the other features."*

## Look at the big picture

- Two questions to ask
  1. What is the overall objective?
     - Building a ML model is not the end goal!
     - Determines which performance measure to use to evaluate the model
  2. What does the current solution look like (if any)?
     - Baseline performance

# The overall system



## Type of ML

- Type of training supervision
  - supervised
  - unsupervised
  - semi-supervised
  - self-supervised
  - reinforcement learning
- Type of output
  - classification
  - regression
  - clustering
- Volume and rate of data
  - batch learning
  - online learning

## Type of ML

- Type of training supervision
  - **supervised**
  - unsupervised
  - semi-supervised
  - self-supervised
  - reinforcement learning
- Type of output
  - **classification**
  - regression
  - clustering
- Volume and rate of data
  - **batch learning**
  - online learning

## Performance measures

- Given problem is regression
- Root mean square error (RMSE)

$$\mathrm{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m} h\left(\mathbf{x}^{(i)}\right) - y^{(i)})^2}$$

- Mean absolute error (MAE)

$$\mathrm{MAE}(\mathbf{X}, h) = \frac{1}{m}\sum_{i=1}^{m} h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\Big|$$

## Get the data

- Textbook code
- Textbook code on Google Colab

# Class work

- Open `02_end_to_end_machine_learning_project.ipynb` on Google Colab
- Textbook code on Google Colab
- Run first few cells
  - Up to “Download the data”

## Exploratory Data Analysis (EDA)

- Systematically going through the data
  - Generating summary statistics
  - Evaluating data quality
  - Visualizing each variable
  - Visualizing pairwise relationships

# Summary Statistics

- *not* visual

- sample statistics of data X
  - mean
  - median
  - quartiles of sorted **X**: Q1 value = $\mathbf{X}_{0.25n}$ , Q3 value = $\mathbf{X}_{0.75\,n}$
    - interquartile range:   value(Q3) - value(Q1)
    - IQR()
  - Variance

- `mydata.describe() # mydata is a Pandas dataframe`

## Data quality

- Is there missing data?
  - Count the NA/Nulls
  - Possible that missing values are not stored as NA
    - Instead, 0 or -1 or something that cannot be a valid value
- `pd.isna()`
- `mydata.head()`
- `mydata.info()`
- Are there outliers?
- Check for each variable

# Numerical vs Categorical Variables

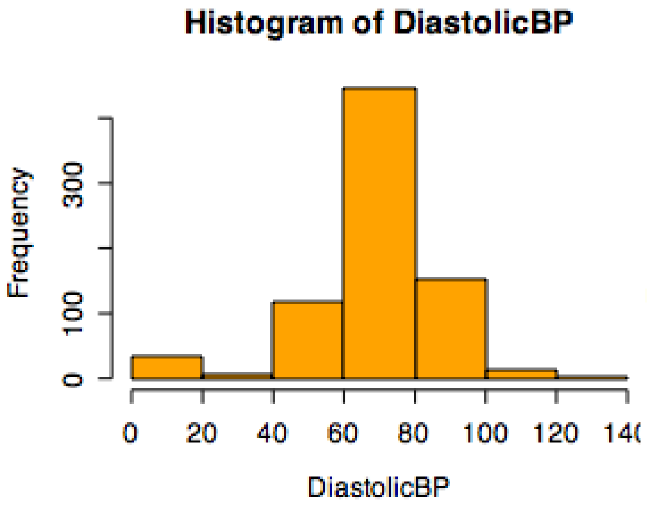
- Numerical variables
  - Values can be ordered, i.e., there is a natural ordering between the values
  - E.g., Age (younger/old), Weight (lighter/heavier)
  - Can calculate averages
  - Also called Ordinal variables
- Categorical variables
  - There is no natural ordering between the values
  - E.g., Marital status: *Single*, *Married*, *Divorced*
  - E.g., Color: *red*, *blue*, *green*, …
  - Cannot compute averages
  - Also called Nominal variables

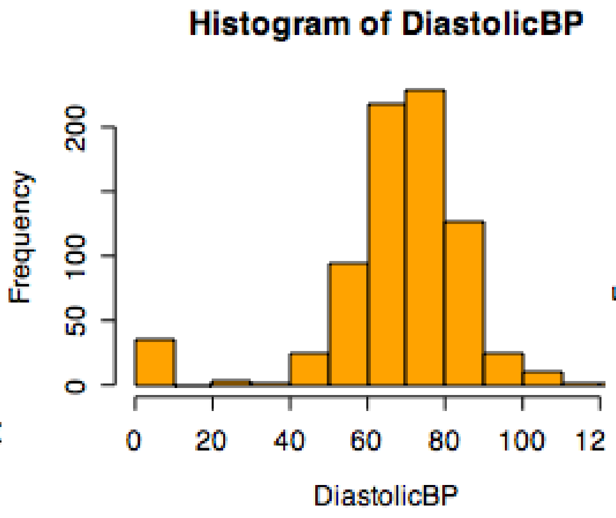
## Single Variable Visualization

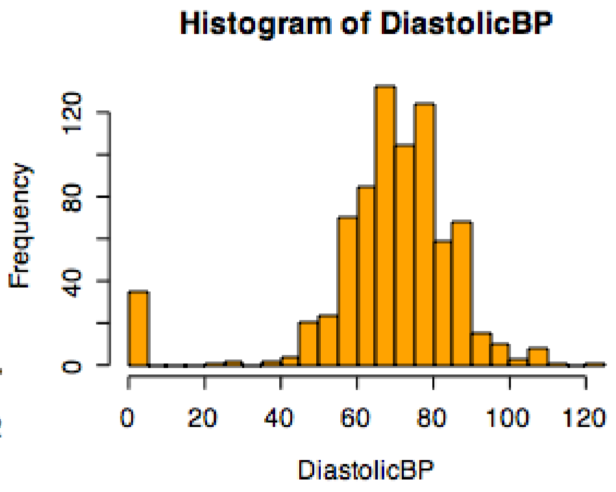
- Numerical variable:
  - Histogram
  - Box plot
- Categorical variable:
  - Bar graph

## Single Variable Visualization

- Histogram:
  - Shows center, variability, skewness, modality, outliers, or strange patterns.
  - Bin width and position matter
  - Beware of real zeros







CALIFORNIA STATE UNIVERSITY FULLERTON

## Class work

- Continue working with `02_end_to_end_machine_learning_project.ipynb`

on Google Colab

- Print summary statistics and visualize variables
  - What do you notice?

## Create a Test Set

- Split data into test-train subsets
- Do exploratory data analysis only on the **train** subset
  - Avoid *data snooping* bias
- We want the train subset to be as large as possible
  - Better ML models with more data
- We want test subset to be large enough to be *representative* of the whole dataset
  - Different features appear in the same proportion
- Typically, 80% for training, remaining 20% for testing

## Random sampling

- Simple but inconsistent when the dataset grows

```
from sklearn.model_selection import
train_test_split
train_set, test_set = train_test_split(myfulldata,
test_size=0.2, random_state=42)
```

- How to ensure that as new data instances arrive, the new test split will be consistent with the previous split
  - Split based on a unique identifier
  - If no unique identifier, create one for the purposes of splitting
  - E.g.: id = longitude * 1000 + latitude

## Stratified sampling

- Random sampling does not guarantee that test subset has features in the same proportion as the full data
- Stratified sampling methods take in an additional column parameter and ensures test and full dataset have the same proportion of the column values

```
train_set, test_set = train_test_split(myfulldata,
test_size=0.2, stratify= myfulldata["col_name"],
random_state=42)
```

CALIFORNIA STATE UNIVERSITY FULLERTON

## Relationships between variables

- Visually
  - Plot multiple variables in one plot
  - Two numerical variables: Scatterplot
  - Additional variables by size and color
- Plotting in Python for ML
  - `Pandas.plotting`
  - `matplotlib`

  -
    ```
    mydata.plot(kind="scatter",
    x="median_income", y="median_house_value",
                    alpha=0.1, grid=True)
    ```

## Relationships between variables

- Numerically
  - Calculate correlation between numerical variables
    - Especially between feature variables and output variable
  - High negative correlation is also a **strong** correlation
  - Values close to 0 indicate **no** correlation
  - Correlation coefficient quantifies **only linear** relationships

```
corr_matrix = mydata.corr(numeric_only=True)
corr_matrix["output_var"].sort_values(ascendin
g=False)
```

## Feature transforms

- Apply a numerical function to a feature
  - Example: log(), sqrt(), …

```
housing["log_total_rooms"] =
np.log(housing["total_rooms"]
```

## Feature transforms

- Combine features to get a new feature that is more related to the target variable
  - Example: Ratio of two variables

```
housing["rooms_per_house"] =
housing["total_rooms"] / housing["households"]
housing["bedrooms_ratio"] =
housing["total_bedrooms"] /
housing["total_rooms"]
```

# Class work

- Continue working with
  `02_end_to_end_machine_learning_project.ipynb`

on [Google Colab]

- Create some new features that are combinations of two other features
  - E.g.: `housing["rooms_per_house"] = housing["total_rooms"] / housing["households"]`
- Calculate the correlation of your new features with the target variable
- What is the highest correlation you can get?

## Feature transforms

- More fancy implantation
- Using scikit-learn’s FunctionTransformer

```
FunctionTransformer(np.log,
     feature_names_out="log transform")
```

## Feature transforms

- More fancy implantation
- Using scikit-learn's FunctionTransformer

```
def column_ratio(X):
    return X[:, [0]] / X[:, [1]]

FunctionTransformer(column_ratio,
feature_names_out="ratio"))
```

# Handling missing data

- Missing data is common in the real-world
- But most ML algorithms cannot work with missing features
- Some options:
  1. Remove instances (entire rows) with missing data
  2. Remove a feature (entire column) that has *any* missing value
  3. Set the missing values to some value (zero, mean, median, etc.): *Imputation*

## Handling missing data

- Some options:

```
mydata.dropna(subset=["bad_feature"],
inplace=True)

mydata.drop("bad_feature", axis=1)

median = mydata["bad_feature"].median()
mydata["bad_feature"].fillna(median,
inplace=True)
```

## Converting categorical features to numerical

1. Assign an integer to every unique categorical value
   – Color = [Red, Blue, Green, Blue, …]
   – Color_encoded = [0, 1, 2, 1, …]

```
from sklearn.preprocessing import OrdinalEncoder
ordinal_encoder = OrdinalEncoder()
Color_encoded =
ordinal_encoder.fit_transform(Color)
```

## Converting categorical features to numerical

1. One-hot encoding
   - Create new columns, one for each unique value
   - Assign 0/1 to each column depending on the value
   - Sparse matrix
   - Color = [Red, Blue, Green, Blue, …]
   - `Color_Red   = [1, 0, 0, 0, …]`
   - `Color_Blue  = [0, 1, 0, 1, …]`
   - `Color_Green = [0, 0, 1, 0, …]`

```
from sklearn.preprocessing import OneHotEncoder
cat_encoder = OneHotEncoder()
Color_onehot = cat_encoder.fit_transform(Color)
```

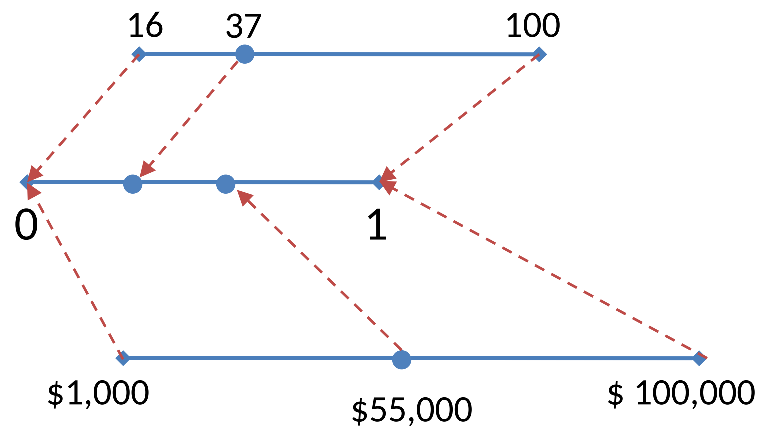
## Feature Scaling

- Most ML algorithms don’t perform well when the input numerical attributes have very different scales
  - will be biased toward ignoring the smaller unit features and focus more on the larger unit features
- Two common ways to get all attributes to have the same scale

# Normalization/Min-Max scaling

- Normalization:
  - Convert each feature independently into comparable ranges (e.g., 0… 1)
  - Percentile:
    - Proportion of the Min to Max range
  - Example:

$$\frac{37 - min}{max - min} = \frac{37 - 16}{100 - 16} = 0.25$$



## Normalization/Min-Max scaling

```
from sklearn.preprocessing import MinMaxScaler

min_max_scaler = MinMaxScaler(feature_range=(-
1, 1))
feature_scaled =
min_max_scaler.fit_transform(original_feature)
```

## Scaling: Standardization

- Min-Max scaling is impacted by even one outlier
- Standardization:
  - subtract the mean value
  - then divide by the standard deviation
- Standardized values have zero mean and standard deviation equal to 1
- But range is not limited (can be very large positive or negative values)

## Scaling: Standardization

```
from sklearn.preprocessing import
StandardScaler

std_scaler = StandardScaler()
features_scaled =
std_scaler.fit_transform(original_features)
```

CALIFORNIA STATE UNIVERSITY
FULLERTON

## Scikit-learn pipelines

- Can combine previous steps into one long pipeline of actions
- Smaller pipelines can be combined into a more complex pipeline
- Typically, define a smaller pipeline for similar features
  - Eg. A log-pipeline for all columns to be transformed using the log function
  - Eg., a one-hot encoding pipeline for all categorical features

```
cat_pipeline = make_pipeline(
    SimpleImputer(strategy="median"),
    OneHotEncoder(handle_unknown="ignore"))
```

- Combine smaller pipelines using a ColumnTransformer

# Class work

- Create a copy of the Python notebook
- Retain only the following steps (i.e., delete the others):
  - Keep aside a test dataset that is 20% of the full dataset using random sampling
  - Replace missing values with median values
  - Replace categorical features using one-hot-encoding
  - Compute a few transformed variables (ratios of two other variables)
  - Standardize all numerical variables
- Ideally, the code will be one pipeline

## Acknowledgement

- Content based on “Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow,” Aurélien Géron, 3rd Edition (October 2022), O'Reilly Media, Inc.