# Data Prep and EDA With Python

## Data Science Workflow

The “Data Science Workflow” is a general framework to be followed on data science projects. It is not a linear process and will require inter-iterations.

1. Scoping a Project
2. Gathering Data
3. Cleaning Data
4. Exploring Data
5. Modeling Data
6. Sharing Insights

## Scoping a Project

Projects don’t start with data, they start with a clearly defined scope.

* Who are the end users and stakeholders?
* What business problems are you trying to solve?
* Is this a supervised or unsupervised learning problem?
  + Is data science the correct approach? Statistics?
* What data do you need to perform the analysis?

### Steps

1. **Think like and end user** – stakeholder – who will benefit from the results of the analysis.
2. **Brainstorm problems** – not necessarily a data science problem, sky is the limit.
3. **Brainstorm solutions** – data science is only one potential solution.
4. **Determine techniques** – is this a supervised learning or unsupervised learning.
5. **Identify data requirements** – what data is required to solve the analysis. Dat requirements may differ for supervised and unsupervised problems. Do we start with an MVP, minimum viable product.
6. **Summarize the scope and objectives** – What techniques and data will we use? What specific impact are we trying to make? What is the goal?

## Gathering Data

Data is the foundation of the project, it is important to collect the right data. Data can come from many different sources:

* Files (flat, spreadsheets, etc.
* Databases
* Websites
* APIs

Pandas Axis = 0 targets a row, 1 targets a column.

### Reading Data Into a Dataframe

Assumes that Pandas is loaded as pd.

#### Flat File

pd.read\_csv(file\_path, sep, header) – load a typical flat file and you can specify the separator.

#### Spreadsheet

pd.read\_excel(file\_path, sheet\_name) – to load data from a spreadsheet, sheet name is the zero based sheet number of the work book.

#### JSON Data

pd.read\_json(file\_path) – read in JSON data

#### Database

pd.read\_sql() – read in data from a SQL database. First import a database driver, connect to the database and then use pd.read\_sql() to query the database.

**From SQLite3**

import sqlite3

conn = sqlite3.connect(‘maven.db’)

pd.read.sql(‘SELECT \* FROM courses’, conn)

df.head(), df.shape, df.describe()

## Cleaning Data

Garbage in, garbage out. Data must be properly cleaned and prepared. This is less fun but very important to the data science workflow, 50 to 80% of the project is spent here. Getting raw data into a format ready for analysis.

* Correcting data types
* Imputing missing data
* Dealing with data inconsistencies
* Reformatting the data
* Outliers

### Checking for Duplicate Values

If your project requires no duplicate values in a given field the following code should be used. The first line compares the number of unique values and the shape of dataframe.

**df[‘enrolleea\_id’].nunique() = df.shape[0]**

**df[‘duplicated’] = df.drop (columns = [‘enrollee\_id ’]).duplicated()**

**df[‘duplicted’].value\_counts()**

**df.drop(df[df[‘duplicated’] == True].index, inplace=True)**

**df.head()**

**print(f”New df size is {df.shape}.”)**

**df.describe()**

#### Special Symbol Clean-up

**df[‘company\_size’] = df[‘company\_size’].apply(lambda x: ’10-49’ if x == ‘10/49’ else x)**

#### Simple Visualizations

**df[‘training\_hours’].hist()**

**sns.violinplot(data=df, x=’education\_level’, y=’training\_hours’, hue=’target’, gridsize=200)**

### Correcting Data Types

**.dtypes** – used to list and inspect the data types in a data frame, i.e numbers must be converted numbers. You can also use **.info()** on a column

#### Converting to a Datetime Column

df.Birthdate = pd.to\_datetime(df.Birthdate)

pd.to\_datetime(dt\_col, format=’%Y-%M-%D’) – if you need to set the order of the date time column conversion.

#### Converting to a Numeric

clean\_income = df.Income.str.replace(‘$’, ‘’).str.repalce(‘,’, ’’)

df.Income = pd.to\_numeric(clean\_income)

**Example**

df.[‘Warm Up Time’] = pd.to\_numeric(df.[‘Warm Up Time’].astype(‘str’).str.repalce(‘ min’), ‘’)

#### Converting a Boolean to an Integer

df.Rain = df.Rain.astype(‘int’)

### Missing Data

**np.NaN** or **np.NA** are missing values in Pandas. **None** in Python.

**Finding Missing Data**

Using **.isna()** or **.info()** or **.value\_counts(dropna=False)**

**df.isna().sum()**

**df[df.isna().any(axis=1)]**

**Variable = df.isna().sum()** to count missing values by column – True values are missing

To see rows with missing values – **df[df.isna().any(axis=1)]**

**df.column\_name.value\_counts()** – will show unique values in a column and the counts for each value.

**df.column\_name.value\_counts(dropna=False) –** will add a Nan count.

**Dropping Missing Data**

**df.dropna()** will remove rows with missing data. Add **.reset\_index** to renumber the rows.

**df.dropna(thresh=2)** will keep rows that have 2 or more missing values – this sounds backwards

**df.dropna(subset=[‘colum\_name’])** will remove rows that have a NaN in that column.

Adding inplace=True updates the current dataframe.

df.dropna(subset=[‘price’], axis=0, inplace=True)

**Replace Missing Data with the Mean**

mean = df[“column\_name”].mean()

df[“column\_name”].replace(np.nan, mean, inplace=True)

**Imputing Missing Data**

The **.fillna()** method – df[‘income’] = df[‘income’].fillna(df[‘income’].median(), inplace=True)

df.year = np.where(df.year.isna(), ‘Freshman’, df.year)

**List other imputing approaches**

**Missing Data Evaluation Function – IBM Data Science**

This function, actually just a for loop for now but plan on making it a function. Assume the dataset has been read into a Pandas dataframe, df.

missing\_data = df.isnull()

missing\_data.head()

for column in missing\_data.columns.values.tolist():

print(column)

print(missing\_data[column].value\_counts()

print(“ ”)

**Replace Missing Data With Mean**

avg\_bore = df[“bore”].astype(‘float’).mean(axis=0)

df[‘bore’].replace(np.nan, avg\_bore, inplace=True)

df.income = df.income.fillna(df[‘income’].median())

Missing Values

missing\_values = df.isna().any(axis=1)

print(missing\_values)

index\_to\_drop = 2

df.drop(index\_to\_drop, inplace=True)

df\_cleaned = df.dropna()

**Unique Counts**

df.column. value\_counts(dropna=False)

### Inconsistent Text & Typos

As an example, full state name instead of Two-letter – categorical data

**df.[“Column\_Name”].value\_counts()** – this provides a list of unique values and their count in a given column.

* **df[df.Class.isin([‘Exploratory Data Analysis’, ‘EDA’])]** – shows potentially same class.

For numeric data your can try **df[“Age”].describe()** to see if the descriptive values make sense.

* **df[df.Grade > 100]** – shows students with class grades over 100, the max value.

The line below corrects a state feature from the state ‘Utah’ spelled out to the 2-letter abbreviation ‘UT’. In general it reads, for the field df.state, if the value equals Utah, please replace with UT, otherwise leave it alone. Using to get all values in state feature to a two letter abbreviation.

**df.state = np.where(df.state == ‘Utah’, ‘UT’, df.state**

The line of code below is used to correct the name of a class to be consistent. The line of codes reads, if a value in the class field reads ‘EDA’ change it to Exploratory Data Analysis, if not then leave it alone.

**df.class = np.where(df.class == ‘EDA’, ‘Exploratory Data Analysis’, df.class)**

Similar to the other examples, this one is trying to cap grades at 100, the max score in this case. If the Grade field is more than 100 change it to 100, if not over 100 leave it alone.

**df.Grade = np.where(df.Grade > 100, 100, df.Grade)**

### Data Formatting – IBM Data Science

This example converts US based, standard mpg to metric liters per something. I have not checked the conversion factors. This is only an example.

df[“city-mpg”] = 235 / df[“city-mpg”]

df.rename(columns=”city=mpg”:”city-L/100kl”, inplace=True)

### Update Values Based on Logical Condition

Using np.where (this is a Numpy function). There is also a Pandas where approach. The Numpy way is vectorized and is faster.

np.where(condition, if true, if false) – basic syntax

df.State = np.where(df.State == ‘Uath’, ‘UT’, df.State) to change a state.

### Mapping Values

Use this method to change values in a dataframe.

state\_mapping = {‘AL’ : ‘AL’, ‘Alabama’: ‘AL’, ‘NY’: ‘NY’, ‘New York’}

df[‘State\_Clean’] = df.State.map[state\_mapping]

### Data Type Correction

df.dtype() # Will show column types for review

df[“price”] = df[“price”].astype(“int”)

### Cleaning Text Data

run\_times.Location.str.strip(‘” ”’).str.lower(),replace(‘the ’, ‘’)

### Duplicate Data

Repeated records and how to deal with them.

**dataframe.duplicated().sum()** – how many rows are duplicated

**dataframe.duplicated(keep=False)** – display duplicated rows

**dataframe.drop\_duplicates()** – drop duplicate rows, can rest the index if needed

### Outliers

Values that are too high or too low to be correct, one way to identify is to use plot. Use histograms, box plots, or standard deviation.

**Histogram**

Import seaborn as sns

df.hist(bins=55)

sns.histplot(df, binwidth=1);

**Box Plots**

sns.boxplot(x=df.Grade);

import numpy as np

q25, q50, q75 = np.percentile(df.Grade, (25, 50, 75))

iqr = q75 – q25

min\_grade = q25 – (1.5 \* iqr)

max\_grade = q75 + (1.5 \* iqr)

min\_grade, q25, q50, q75, max\_grade \* to show grades

df[df.Grade < min\_grade]

df[df.Grade > max\_grade]

**Standard Deviation**

Anything more than 3 standard deviations from the mean can be considered an outlier. Can be adjusted if needed.

mean = np.mean(df.grade)

sd = np.std(df.grade)

[grade for grade in df.Grade if (grade < mean - 3\*sd) or (grade > mean + 3\*sd) ]

***Handling Outliers***

**df[df.Grade < 50]**

**df.drop([list row numbers]) # drop the entire row – sloppy**

**df[df.Grade >= 60]**

**df[df.Grade < 60]**

**min\_grade = df[df.Grade >= 60].Grade.min()**

**min\_grade**

**np.where(df.Grade < 60, min\_grade, df.Grade)**

**Detecting Outliers With IQR**

# Calculate the interquartile range

**q25, q50, q75 = np.percentile(data[‘field’], [25, 50, 75])**

**iqr = q75 – q25**

# Calculate the min and max limits to define outliers

**min = q25 – 1.5\*(iqr)**

**max = q75 + 1.5\*(iqr)**

**print(min, q25, q50, q75, max)**

# identify outlier points

**[x for x in data[‘unemployment’] if x > max]**

**[x for x in data[‘unemployment’] if x < min]**

# something is missing here

### Outlier Detection and Correction

## Common Imports and Setup

# Importing

from sklearn.datasets import load\_diabetes

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

# Load the dataset

diabetes = load\_diabetes()

# Create the dataframe

column\_name = diabetes.feature\_names

df\_diabetes = pd.DataFrame(diabetes.data)

df\_diabetes .columns = column\_name

df\_diabetes .head()

print("Old Shape: ", df\_diabetes.shape)

## Outlier Detection with a Box Plot

sns.boxplot(df\_diabetes['bmi'])

def removal\_box\_plot(df, column, threshold):

sns.boxplot(df[column])

plt.title(f'Original Box Plot of {column}')

plt.show()

removed\_outliers = df[df[column] <= threshold]

sns.boxplot(removed\_outliers[column])

plt.title(f'Box Plot without Outliers of {column}')

plt.show()

return removed\_outliers

threshold\_value = 0.12

no\_outliers = removal\_box\_plot(df\_diabetes, 'bmi', threshold\_value)

## Outlier Removal using IQR

# IQR

# Calculate the upper and lower limits

Q1 = df\_diabetes['bmi'].quantile(0.25)

Q3 = df\_diabetes['bmi'].quantile(0.75)

IQR = Q3 - Q1

lower = Q1 - 1.5\*IQR

upper = Q3 + 1.5\*IQR

# Create arrays of Boolean values indicating the outlier rows

upper\_array = np.where(df\_diabetes['bmi'] >= upper)[0]

lower\_array = np.where(df\_diabetes['bmi'] <= lower)[0]

# Removing the outliers

df\_diabetes.drop(index=upper\_array, inplace=True)

df\_diabetes.drop(index=lower\_array, inplace=True)

# Print the new shape of the DataFrame

print("New Shape: ", df\_diabetes.shape)

### Outlier Caping and Flooring

df\_diabetes.info()

df\_diabetes.describe()

**Upper Capping**

uv = np.percentile(df\_diabetes.bmi, [99])[0]

print(uv)

df\_diabetes[(df\_diabetes.bmi>uv)]

df\_diabetes.bmi[(df\_diabetes.bmi > 3\*uv)] = 3\*uv

df\_diabetes[(df\_diabetes.bmi>uv)]

**Lower Flooring**

lv = np.percentile(df\_diabetes.bmi, [1])[0]

print(lv)

df\_diabetes[(df\_diabetes.bmi<lv)]

df\_diabetes.bmi[(df\_diabetes.bmi < 0.3\*lv)] = 0.3\*lv

df\_diabetes[(df\_diabetes.bmi<lv)]

df\_diabetes.head(11)

Update a Value

df.loc[row\_number, ‘Grade’] = 74

**Review Dataset**

# Missing Data

df[df.isna().any(axis=1]

# Inconsistent Text and Typos

df.Class.value\_counts()

# Duplicate Date

df[df.duplicated()]

# Outliers

sns.histplot(df)

### Creating Numeric Columns

**Calculating a Percentage**

The dataframe, shopping\_list, has the columns; Category, Item, and Price.

# Calculate the total spent

shopping\_list[‘Total Spend’] = shopping\_list[‘Price’].sum()

shopping\_list

# Calculate the percent spent

shopping\_list[‘Percent Spent’] = (shopping\_list[‘Price’] / shopping\_list[‘Total Spend’]) \* 100

shopping\_list

**Adding Taxes to Current Column**

We want to add taxes to an existing fee column using np.where()

run\_time[‘Fee with Tax’] = np.where(run\_time.location == ‘gym’, run\_time.Fee \* 1.08, run\_time.Fee )

### Extracting DateTime Components

The following cab be used to extract information from datetime fields in a dataframe (run\_times) and could be saved to a variable or a new field.

**run\_times[‘Run Date’].dt.day**

**run\_times[‘Run Date’].dt.dayofweek**

**run\_times[‘Run Date’].dt.time**

### DateTime Calculations

Use pd.to\_timedelta() to add or subtract time frames

run\_times[‘Run Date’] - run\_times[‘Run Date’]

# Add two weeks to the race date

**run\_times[‘Run Date’] + pd.to\_timedelta(2, unit=’W’ )**

### Extracting Text

Three ways to extract information from text fields.

.str[start:end]

**run\_notes.notes.str[:6]**

.str.split

**two\_fields = run\_notes.notes.str.split(‘-’)**

**pd.Dateframe(two\_fields.to\_list(), columns=[‘Day’, ‘Notes’])**

.str.contains()

**run\_notes[‘Contains final’] = run\_notes.str.contains(‘final’)** # new Boolean field

**run\_notes[‘Contains final’] = run\_notes.str.contains(‘great | congrats’, regex=True)** # new Boolean field

## Exploratory Data Analysis

EDA is all about exploring and understanding the projects data before building models. Should yield insights in the data and provide more understanding of the data.

* Slicing and dicing the data
* Summarizing the data
* Visualizing the data

### Filtering Data

**tea.loc[tea.type == ‘herbal’]**

**mask = (tea.loc[tea.type == ‘herbal’] | tea.temp >= 200) # | ‘or’ or & ‘and’**

**tea.loc[mask]**

Using .loc or [] to select rows

**tea.loc[tea[‘Low Inventory’] == ‘Low Inventory’]** # show all rows with a low inventory

**tea.loc[tea[‘Low Inventory’] == ‘Low Inventory’, [‘Price’, ‘Inventory’]]** # shows only rows and columns that match

**tea.FieldName = tea.FieldName.str.strip()** # removes blank spaces

### Sorting Data

tea.sort\_values(‘name’) # can also pass in ascending=False to reserve sort

### Grouping Data

tea\_temps = tea.groupby(‘type’)[‘temp].mean().reset\_index()

tea.groupby(‘type’)[‘temp].agg([‘min’, ‘max’, ‘count’, ‘nunique’]).reset\_index()

**Example**

df\_test = df[[‘A’, ‘B’, ‘Price’]]

df\_grp = df\_test.groupby([‘A’, ‘B’], as\_index=False).mean()

df\_grp

df\_pivot = df\_grp.pivot(index=’A’, columns=’B’)

df\_pivot

plt.pcolor(df\_pivot, cmap=’RdBu’)

plt.colorbar()

plt.show()

## Visualizing Data

Using Pandas to visualize data. This example uses tea\_temps, a dataframe from a previous exercise.

**tea\_temps.plot.barh();**

From a dataframe

**dataframe.plot.line(x=’year’, y=’score’);**

***Bar Plots***

**tea.groupby([‘field’])[‘Price’].mean().sort\_values().plot.bar();** # provide a simple bar plot

**tea.groupby([‘field’])[‘Price’].mean().sort\_values().plot.barh();** # provide a simple horizontal bar plot

***Line Plot***

**hap[hap.country\_name == ‘Mexico’].iloc[:, 1:3].plot.line(x=’year’, y=’other\_field’ )**

**(hap[hap.country\_name**

**.isin([‘Canada’, ‘Mexico’, ‘United States’])]**

**.iloc[:, :3]**

**.pivot(index=’year’, columns=’country\_name’, values=’happiness\_score’)**

**.plot.line())**

***Pair Plots - Seaborn***

**sns.pairplot(data=dataframe)**

***Scatter Polts – Seaborn***

**sns.scatterplot(data=df, x=’column1’, y=’colunn2’)**

***Correlation***

Correlation does not imply causation

**df.corr()**

### EDA Tips

* Remind yourself of the original questions – don’t get lost in the data
* Keep a running list of observations and questions
* Apply steps in any that makes sense for the project
* You may need to go back to earlier steps or collect more data
* Have you already answered any of your questions

## Descriptive Statistics- IBM Data Science

df.describe() # Provides general descriptive statistics about a dataframe named df.

df.value\_counts() # Provides distinct value counts.

df[‘A’].unique() # unique values in a column

**Example**

new\_var\_count = df[‘new\_var’].value\_counts()

new\_var\_count.rename(columns=(‘new\_var’:’value\_counts’), inplace=True)

new\_var\_count.index.name=’New Var’

sns.boxplot(data=df, x=’A’, y=’price’)

Need to include code for a Seaborn box plot

### Data Correlation – IBM Data Science

sns.regplot(x=’A’, y=’Price’, data=df)

plt.ylim(0,)

# requires statsmodels

person\_coef, p\_value = stats.personr(df[‘A’], df[‘Price’])

# need to include a heat map

## Modeling Data

Involves structing and preparing the data for specific modeling techniques, and applying algorithms to build a model to make predictions or discover patterns.

* Restructuring the data
* Feature engineering (adding new fields)
* Applying machine learning algorithms

### Data Preparation for Modeling

**Steps**

1. Create a **single table**
2. Set the correct **row granularity**
3. Ensure each **column** is non-null and numeric
4. **Engineer features** for the model as appropriate

#### Create a Single Table

**Appending** – adding rows to a table, must have the exact same columns.

**pd.concat()** – to append to a table, columns must be identical – **pd.concat(df\_1, df\_2).reset\_index()**

**Joining** – adding columns to a table based on common values, the tables must have at least one common column.

**left\_df.merge(right\_df, how=’left’, left\_on=”field1”, right\_on=”field1”)** – The left\_on and right\_on are the common column.

* **inner** – records that exist in both records
* **left** – includes all records from the left table and only matching records from the right table
* **right** – includes all records from the right table and only matching records from the left table
* **outer** – includes all records from both tables

### Preparing Rows for Modeling

Aggregate individual customer data into a single row, use groupby() to collapse to get each row to an aggregate for each customer.

### Preparing Columns for Modeling

There can be no null values for modeling.

**Dummy Variables**

To convert text data to numeric. Also known as “One-Hot Encoding.”

pd.get\_dummies(column\_name, drop\_first=True)

**Preparing DateTime Columns**

You should not use month numbers because it will rank them by number.

month\_dummies = pd.get\_dummies(df.month)

**monthly\_purchases = pd.concat([df.customers, month\_dummies, axis=1)**

**Calculate Days From a Date**

**today = purchases.purchase\_date.max()** # Will this give the date or the last in data set?

**last\_purchase[‘days\_passed’] = (today – last\_purchase.purcahse\_date).dt.days**

### Feature Engineering

Will be covered in more depth in future courses

**Transformations**

Log Transformation – **np.log()**

Feature Scaling

* **Normalization** – all values between 0 and 1, good with unknown or odd distributions
* **Standardization** – all values between -3 and 3, good with normal distributions

I need some code examples for standardization

**Proxy Variables**

Used to deal with non-numbers like zip codes, use **median income** instead of zip code. Probably from a different data source and then joining to the dataframe.

**Feature Engineer Tips**

* Should domain expertise to create relevant features
* Data set should be long not wide, more rows than columns
* Loop back through the process.

## Regression

This will be the section on regression.

Regression Plot

sns.residplot(df[‘A’], df[‘Price’])

## Sharing Insights

Summarizing key findings for end users and stakeholders

* Reiterate the problem
* Interpret the results of your analysis
* Share recommendations and next steps
* Deploy model if required
* Focus on the potential impact of the analysis, not technical details

## Creating Dummy Variables with Pandas and a For Loop

**df\_model** # A dataframe that is read for dummy variables

**cat\_cols = []**

**for col in df\_model.columns:**

**print(‘\n’)**

**print(f“Column {col} is of data type {df\_model[col].dtype}”)**

**if df\_model[col].dtype == ‘object’**

**print(‘Creating Dummies for it’)**

**cat\_cols.append(col)**

**df\_model = pd.get\_dummies(df\_model, columns=cat\_cols)**

### Target Variable Correction

Converting a target, Y, variable from a float to an integer

**df\_model[‘target’] = pd.to\_numeric(df\_model[‘target’])**

**df\_model[‘target’] = df\_model[‘target'].astype(int)**

next cell to verify conversion and counts

**df\_model[‘target’].value\_counts()**

Covert Data Types to Float

1. for col in X\_test.columns:
2. X\_test[col] = X\_test[col].astype(float)

## Feature Correlations - Heatmap

## Imports

import seaborn as sns

import matplotlib.pyplot as plt

import numpy as np

df = sns.load\_dataset('mpg')

df.head()

# Drop non-numeric features

df.drop('origin', axis=1, inplace=True)

df.drop('name', axis=1, inplace=True)

df.head()

df.corr()

my\_corr\_mat = df.corr()

## Heatmap Examples

sns.heatmap(my\_corr\_mat, cmap='plasma')

plt.figure(figsize=(12,6), dpi=250)

sns.heatmap(my\_corr\_mat, cmap='plasma', vmin=-1, vmax=1);

plt.figure(figsize=(12,6), dpi=250)

sns.heatmap(my\_corr\_mat, cmap='plasma', center=0.5, vmin=0.5, vmax=1);

plt.figure(figsize=(12,6), dpi=250)

sns.heatmap(my\_corr\_mat, cmap='plasma', center=0, vmin=-0.25, vmax=0.25, annot=True);

plt.figure(figsize=(12,6), dpi=250)

sns.heatmap(my\_corr\_mat, annot=True, cmap='magma', vmin=-1, vmax=1);

plt.figure(figsize=(12,6), dpi=250)

sns.heatmap(my\_corr\_mat, annot=True, cmap='magma', vmin=-1, vmax=1, linewidths=2, linecolor='red');

plt.figure(figsize=(12,6), dpi=250)

sns.heatmap(my\_corr\_mat, annot=True, cmap='magma', vmin=-1, vmax=1, square=True);

plt.figure(figsize=(12,6), dpi=250)

sns.heatmap(my\_corr\_mat, annot=True, cmap='magma', vmin=-1, vmax=1, square=True, mask=np.triu(my\_corr\_mat));

plt.figure(figsize=(12,6), dpi=250)

sns.heatmap(my\_corr\_mat, annot=True, cmap='mako', vmin=-1, vmax=1, square=True);

plt.figure(figsize=(12,6), dpi=250)

sns.heatmap(my\_corr\_mat, annot=True, cmap='vlag', vmin=-1, vmax=1, square=True);

plt.figure(figsize=(12,6), dpi=250)

sns.heatmap(my\_corr\_mat, annot=True, cmap='icefire', vmin=-1, vmax=1, square=True);

plt.figure(figsize=(12,6), dpi=250)

sns.heatmap(my\_corr\_mat, annot=True, cmap='icefire', fmt=".2f", vmin=-1, vmax=1, square=True, mask = np.triu(my\_corr\_mat));

plt.figure(figsize=(12,6), dpi=250)

sns.heatmap(my\_corr\_mat, annot=True, cmap='coolwarm', fmt=".2f", vmin=-1, vmax=1, square=True);

plt.figure(figsize=(12,6), dpi=250)

sns.heatmap(my\_corr\_mat, annot=True, cmap='Spectral', fmt=".2f", vmin=-1, vmax=1, square=True);

plt.figure(figsize=(12,6), dpi=250)

sns.heatmap(my\_corr\_mat, annot=True, cmap='Spectral', fmt=".2f", vmin=-1, vmax=1, square=True, mask = np.triu(my\_corr\_mat));

### Missing Value Imputers

**SimpleImputer:** a basic imputer for missing values.

**IterativeImputer:** This imputer estimates the missing values by modeling each feature with missing values as a function of other features and then estimating the missing values iteratively. It's particularly useful for handling missing values in datasets with complex dependencies between features.

**KNNImputer:** This imputer imputes missing values by finding the nearest neighbors with non-missing values and averaging their values. It's based on the k-nearest neighbors algorithm.

**IterativeSVD:** This imputer uses Singular Value Decomposition (SVD) to estimate missing values iteratively. It's suitable for handling missing values in high-dimensional datasets.

**MatrixFactorization:** This imputer uses matrix factorization techniques, such as Singular Value Decomposition (SVD) or Non-negative Matrix Factorization (NMF), to estimate missing values.

**MissingIndicator:** This transformer adds binary indicators for missing values in the dataset, allowing machine learning algorithms to handle missing values more effectively.

### Missing Value Operations

**Drop a Data Frame Row**

import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],

'Age': [25, 30, 35, 40]}

df = pd.DataFrame(data)

print("Original DataFrame:")

print(df)

# Drop a row by index

index\_to\_drop = 2

df.drop(index\_to\_drop, inplace=True)

print("\nDataFrame after dropping a row:")

print(df)

**Drop Rows with Missing Values**

# Create a sample dataframe with missing values

data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],

'Age': [25, None, 35, 40]} # 'None' represents missing value

df = pd.DataFrame(data)

# Display the original dataframe

print("Original DataFrame:")

print(df)

# Find missing values

missing\_values = df.isna().any(axis=1)

print(missing\_values)

# Drop rows with missing values

df\_cleaned = df.dropna()

# Display the dataframe after dropping rows with missing values

print("\nDataFrame after dropping rows with missing values:")

print(df\_cleaned)

**Replace Missing Values with the Mean**

# Create a sample dataframe with missing values

data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],

'Age': [25, None, 35, 40]} # 'None' represents missing value

df = pd.DataFrame(data)

# Display the original dataframe

print("Original DataFrame:")

df.drop('Name', axis=1, inplace=True)

print(df)

# Fill missing values with the mean of the column

df\_filled = df.fillna(df.mean())

# Display the dataframe after imputing missing values

print("\nDataFrame after imputing missing values:")

print(df\_filled)

**Replace Missing Missing Values with Simple Imputer**

import pandas as pd

from sklearn.impute import SimpleImputer

# Create a sample dataframe with missing values

data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],

'Age': [25, None, 35, 40]} # 'None' represents missing value

df = pd.DataFrame(data)

print(df)

# Create a SimpleImputer object

imputer = SimpleImputer(strategy='mean')

# other strategy parameters: mean, median, most\_frequency, consatant

# Impute missing values

df\_imputed\_array = imputer.fit\_transform(df[['Age']])

# Convert the imputed array back to a dataframe

df\_imputed = pd.DataFrame(df\_imputed\_array, columns=['Age'])

# Update the dataframe with imputed values

df['Age'] = df\_imputed\_array

print(df)

**Replacing Missing Values with KKNImputer**

import pandas as pd

from sklearn.impute import KNNImputer

# Create a sample dataframe with missing values

data1 = {'Feature1': [1, 2, None, 4, 5],

'Feature2': [None, 2, 3, 4, 5],

'Feature3': [1, 2, 3, 4, None]}

df = pd.DataFrame(data1)

# Display the original dataframe

print("Original DataFrame:")

print(df)

# Create a KNNImputer object

imputer = KNNImputer(n\_neighbors=2)

# Impute missing values

df\_imputed\_array = imputer.fit\_transform(df)

# Convert the imputed array back to a dataframe

df\_imputed = pd.DataFrame(df\_imputed\_array, columns=df.columns)

# Display the dataframe after imputing missing values

print("\nDataFrame after imputing missing values with KNNImputer:")

print(df\_imputed)

**Replacing Missing Values with IterativeImputer**

import pandas as pd

from sklearn.experimental import enable\_iterative\_imputer # Required to enable IterativeImputer

from sklearn.impute import IterativeImputer

# Create a sample dataframe with missing values

data2 = {'Feature1': [1, 2, None, 4, 5],

'Feature2': [None, 2, 3, 4, 5],

'Feature3': [1, 2, 3, 4, None]}

df = pd.DataFrame(data2)

# Display the original dataframe

print("Original DataFrame:")

print(df)

# Create an IterativeImputer object

imputer = IterativeImputer()

# Impute missing values

df\_imputed\_array = imputer.fit\_transform(df)

# Convert the imputed array back to a dataframe

df\_imputed = pd.DataFrame(df\_imputed\_array, columns=df.columns)

# Display the dataframe after imputing missing values

print("\nDataFrame after imputing missing values with IterativeImputer:")

print(df\_imputed)

### Reshaping Data

The df.pivot function puts all of each class on one row.

df.pivot(index=’Customer’, Columns=’Genre’, Values=’# Songs’).fillna(0).reset\_index())

The df.melt function does the reverse of the df.pivot function.

The df.groupby function sums all of one classs into one observation

(df.groupby(‘Customer’)[‘# Songs’].sum().reset\_index())

#### Prepare Columns for Modeling

* **Ensure all values are non-null -** We can use **df.info()** or **df.isna()**
* All values must be numeric **– np.where()**
* Convert categorical fields into numeric with dummy variables

### Missing Data

Find missing values with **df.info()** or **df.isna()**

Example: **df.isna().any(axis=1)**

Keep a raw copy of the data import

df\_customers = customers\_raw.copy()

df\_customers.info() or customers.isna()

df\_customers.isna().any(axis=1)

df\_customers[df\_customers.isna().any(axis=1)]

The second line provides a list of rows with NaN values

Drop rows with NaN values

df\_customers.dropna().reset\_index()

**Fill in missing data**

df\_customers[‘Age’] = df\_customers.Age.fillna(df\_customers.Age.median())

df\_customers[‘Followers’ = df\_customers.Followers.fillna(0)]

**Converting to Numeric**

df\_customers[‘Income’] = df\_customers.Income.str.replace(‘$’, ‘’).replace(‘,’, ‘’)

df\_customers[‘Income’] = pd.to\_numeric(df\_customers.Income)

**Converting to DateTime Then to Numeric**

df\_customers[‘Sign Up Date’] = pd.to\_datetime(df\_customers[‘Sign Up Date’], format=’%m/%d/%y’)

df\_customers[‘Sign Up Month’] = df\_customers[‘Sign Up Date’].dt.match

df\_customers[‘Sign Up Day’] = df\_customers[‘Sign Up Date’].dt.dayofweek

df\_customer[‘Weekend’] = np.where(df\_customer[‘Sign Up Date’].isin([5,6]), 1, 0) # Binning

df\_customer.drop(Columns=[‘Sign Up Day’] # then drop ‘Sign Up Day’

**Converting Binary Categorical Field to Numeric**

df\_customers[‘Discount’] = np.where(df\_customers[‘Discount’] == ‘Yes’, 1, 0)

**Dummy Variables** – For Features with more than two values

dummies\_edu = pd.get\_dummies(df\_customers[‘Education’]).astype(int)

df\_customer = pd.concat([customers, dummies\_edu], axis=1)

df\_customer = df\_customers.drop(columns=[‘Education’])

**Applying Calculations**

df\_customer[‘Pct Pop’] = df\_customer[‘Pop’] / df\_customer[‘# Songs’]

**Excluding Identifiers**

names = df\_customer.Name

# Save the names column, can’t put into model but will need for later analysis, will tie together with indexes

df\_customer = df\_customer.drop(columns=[‘Name’])

### Feature Scaling

**Normalization**

Used on data set with unknown distributions – between 0 and 1

from sklearn.preporcessing import MinMaxScaler

mm\_scaler = MinMaxScaler

model\_df\_subset = model\_df[‘Age’, ‘# Songs’, ‘Pct\_Pop’]

normalized = mm\_scaler.fit\_transform(model\_df\_subset)

pd.DataFrame(normalized, columns= model\_df\_subset)

concat back to main data frame

**Standardization**

Use on normally distrubuted – based on standard deviations

from sklkearn.preprocessing import StandardScaler

std\_scaler = StgandardScaler()

standardized = std\_scaler.fit\_transform(model\_df\_subset)

pd.DataFrame(standardized, columns= model\_df\_subset)

concat back to main data frame

The End