template

December 10, 2024

[98]:
```
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

[99]:
```
# %pip install tabulate
```

1 Outage Severity and People Affected

Name(s): Quy-Dzu Do

Website Link: https://krazykats.github.io/Outages_and_Stats/

[100]:
```
import pandas as pd
import numpy as np
import os

import plotly.express as px
pd.options.plotting.backend = 'plotly'

from dsc80_utils import * # Feel free to uncomment and use this.

LOCAL_DIR = os.getcwd()
website_folder = os.path.join(LOCAL_DIR, "..", "Website_Resources")
```

[101]:
```
from final_proj import *
```

1.1 Step 1: Introduction

I chose to do the Outage Dataset as I felt the data to be more relevant to me as I hav lived in California for most of my life have gone through many outages due to High Winds, fires and other reasons. The League data does not interest me since I have no experience with the game so I have no expertise in the field and any conclusions I couls draw would most likely lack context needed to properly extrapolate the data. As for the recepies, I would not be opposed to working with it but I find the subject of outages to be more compelling than data on cooking and nutritional facts.

For the outages dataset, I am most interested in looking at the relation between frequency and duration of outages with the corresponding population affected. I expect there to be a difference as most companies would be more concerned with population centers and centers of commerce getting an outage than a rural population but I would like to see how disproporionate is the result.

1.2 Step 2: Data Cleaning and Exploratory Data Analysis

We shall begin by pulling the data into a Pandas Dataframe skipping the first 5 rows and the seventh row as they do not contain data and we set the `"OBV"` Column to the index as it IDs the rows of data. We will also convert the `"OUTAGE.START.TIME"` and `"OUTAGE.RESTORATION.TIME"` to timedelta objects, and `"OUTAGE.START.DATE"` and `"OUTAGE.RESTORATION.DATE"` to timestamp objects so they can be combined into a single date-time value stores as a Pandas Timestamp in a new dataframe with `"OUTAGE.START.DATE"` and `"OUTAGE.RESTORATION.DATE"` containing the new objects and the Time columns dropped.

[102]:
```
data_set = os.path.join(LOCAL_DIR, "outage.xlsx")
data = pd.read_excel(data_set, skiprows=lambda x: x in [0,1,2,3,4,6],
                     dtype={"OUTAGE.START.TIME": str,
                            "OUTAGE.RESTORATION.TIME": str})
data = data.drop("variables", axis=1).set_index("OBS")
data_time= time_to_datetime(data)
data_time.head()
```

[102]:
```
     YEAR  MONTH U.S._STATE POSTAL.CODE  … AREAPCT_UC PCT_LAND  \
OBS                                      …
1    2011    7.0  Minnesota          MN  …        0.6    91.59
2    2014    5.0  Minnesota          MN  …        0.6    91.59
3    2010   10.0  Minnesota          MN  …        0.6    91.59
4    2012    6.0  Minnesota          MN  …        0.6    91.59
5    2015    7.0  Minnesota          MN  …        0.6    91.59

     PCT_WATER_TOT PCT_WATER_INLAND
OBS
1             8.41             5.48
2             8.41             5.48
3             8.41             5.48
4             8.41             5.48
5             8.41             5.48

[5 rows x 55 columns]
```

[ ]:

[103]:
```
fig1 = data_time["OUTAGE.DURATION"]\
    .hist(bins=100, title="Outage Duration Distribution", labels={"value":␣
 ↪"Duration (minutes)", "index": "Frequency"})
fig1.update_layout(width=800, height=600)
fig1.show()
```

[104]:
```
fig2 = data_time.loc[data_time["U.S._STATE"] == "California", "OUTAGE.
↪DURATION"]\
    .hist(bins=100, title="Outage Duration Distribution in California",␣
↪labels={"value": "Duration (minutes)", "index": "Frequency"})
fig2.update_layout(width=800, height=600)
fig2.show()
```

[105]:
```
fig1.write_html(os.path.join(website_folder, "uni_1_1.html"),␣
↪include_plotlyjs='cdn')
fig2.write_html(os.path.join(website_folder, "uni_1_2.html"),␣
↪include_plotlyjs='cdn')
```

[106]:
```
fig = data_time.groupby("YEAR")["OUTAGE.DURATION"].agg("mean")\
    .plot.line(title="Average Outage Duration by Year")
fig.update_layout(width=800, height=600)
fig.show()
```

[107]:
```
fig.write_html(os.path.join(website_folder, "uni_2_1.html"),␣
↪include_plotlyjs='cdn')
```

[108]:
```
# Create the scatter plot
fig = px.scatter(data_time, x='OUTAGE.DURATION',
                 y='CUSTOMERS.AFFECTED',
                 title="Customers Affected vs. Outage Duration",
                 labels={'OUTAGE.DURATION': 'Outage Duration (minutes)',
                         'CUSTOMERS.AFFECTED': 'Customers Affected'})

# Show the plot
fig.update_layout(width=800, height=600)
fig.show()
```

[109]:
```
fig.write_html(os.path.join(website_folder, "bi_1.html"),␣
↪include_plotlyjs='cdn')
```

[110]:
```
# Create the scatter plot
fig = px.scatter(data_time, x='TOTAL.CUSTOMERS',
                 y='CUSTOMERS.AFFECTED',
                 title="Customers Affected vs. Total Customers",
                 labels={'TOTAL.CUSTOMERS': 'Number of Customers in the State',
                         'CUSTOMERS.AFFECTED': 'Customers Affected'})

# Show the plot
fig.update_layout(width=800, height=600)
fig.show()
```

[111]:
```
fig.write_html(os.path.join(website_folder, "bi_2.html"),␣
↪include_plotlyjs='cdn')
```

[112]:
```
table = pd.pivot_table(data_time, values=['OUTAGE.DURATION',
                                          "CUSTOMERS.AFFECTED",
                                          "DEMAND.LOSS.MW"],
                         index=["U.S._STATE"],
                        aggfunc="mean")
table
```

[112]:
```
                CUSTOMERS.AFFECTED  DEMAND.LOSS.MW  OUTAGE.DURATION
U.S._STATE
Alabama                   94328.80          291.50          1152.80
Alaska                    14273.00           35.00              NaN
Arizona                   64402.67         1245.70          4552.92
…                              …              …              …
West Virginia            179794.33          362.00          6979.00
Wisconsin                 45876.00          161.00          7904.11
Wyoming                   11833.33           26.75            33.33

[50 rows x 3 columns]
```

[ ]:

1.3 Step 3: Assessment of Missingness

[113]:
```
# assess missingness of Outage Duration in realtion to other columns
missing_data = data_time.copy()
missing_data["MISSING_LABEL"] = (missing_data["OUTAGE.DURATION"].isna()).
 ↪astype(str)
missing_data
```

[113]:
```
      YEAR  MONTH    U.S._STATE POSTAL.CODE  … PCT_LAND PCT_WATER_TOT  \
OBS                                          …
1     2011    7.0     Minnesota          MN  …    91.59          8.41
2     2014    5.0     Minnesota          MN  …    91.59          8.41
3     2010   10.0     Minnesota          MN  …    91.59          8.41
…     …    …            …         …   …        …            …
1532  2009    8.0  South Dakota          SD  …    98.31          1.69
1533  2009    8.0  South Dakota          SD  …    98.31          1.69
1534  2000    NaN        Alaska          AK  …    85.76         14.24

      PCT_WATER_INLAND MISSING_LABEL
OBS
1                 5.48         False
2                 5.48         False
3                 5.48         False
…                …            …
1532              1.69         False
1533              1.69         False
```

```
1534              2.90          True

[1534 rows x 56 columns]
```

[114]:
```
# conduct  test againt Month
stats, obs = permutation_test(missing_data, 'MONTH', 'MISSING_LABEL', tvd)
np.mean(stats >= obs)
```

[114]:
```
np.float64(0.132)
```

[115]:
```
fig = px.histogram(stats)
fig.add_vline(x=obs, line_width=3, line_dash="dash", line_color="red")
fig.update_layout(width=800, height=600)
fig.show()
```

[116]:
```
fig.write_html(os.path.join(website_folder, "missing_MCAR.html"),␣
 ↪include_plotlyjs='cdn')
```

[117]:
```
#
stats, obs = permutation_test(missing_data, 'NERC.REGION', 'MISSING_LABEL', tvd)
np.mean(stats >= obs)
```

[117]:
```
np.float64(0.001)
```

[118]:
```
obs
```

[118]:
```
np.float64(0.3153910849453322)
```

[119]:
```
fig = px.histogram(stats)
fig.add_vline(x=obs, line_width=3, line_dash="dash", line_color="red")
fig.update_layout(width=800, height=600)
fig.show()
```

[120]:
```
fig.write_html(os.path.join(website_folder, "missing_MAR.html"),␣
 ↪include_plotlyjs='cdn')
```

[ ]:
```

```

[121]:
```
missing_data
```

[121]:
```
      YEAR  MONTH    U.S._STATE POSTAL.CODE  … PCT_LAND PCT_WATER_TOT  \
OBS                                          …
1     2011    7.0     Minnesota          MN  …    91.59          8.41
2     2014    5.0     Minnesota          MN  …    91.59          8.41
3     2010   10.0     Minnesota          MN  …    91.59          8.41
…     …      …           …         …   …     …            …
1532  2009    8.0  South Dakota          SD  …    98.31          1.69
1533  2009    8.0  South Dakota          SD  …    98.31          1.69
```

```
1534  2000    NaN        Alaska          AK  …    85.76         14.24

      PCT_WATER_INLAND MISSING_LABEL
OBS
1                 5.48         False
2                 5.48         False
3                 5.48         False
…                …            …
1532              1.69         False
1533              1.69         False
1534              2.90          True

[1534 rows x 56 columns]
```

1.4 Step 4: Hypothesis Testing

diff_medians is a custom function found in one of the auxilary python scripts that takes the difference in the medians of a value column grouped by a 2 label label column.

[122]:
```
permutation_data = data_time.copy()
permutation_data["Is_California"] = (permutation_data["U.S._STATE"] ==␣
↪"California").astype(str)
```

[123]:
```
diff_medians(permutation_data, "OUTAGE.DURATION", "Is_California")
```

[123]:
```
np.float64(-581.5)
```

[124]:
```
n = 1000
medians_diff = []
observed_diff = diff_medians(permutation_data, "OUTAGE.DURATION",␣
↪"Is_California")
for _ in range(n):
    permutation_data["shuffled_labels"] = np.random.
↪permutation(permutation_data["Is_California"])
    medians_diff.append(diff_medians(permutation_data, "OUTAGE.DURATION",␣
↪"shuffled_labels"))

np.mean([diff <= observed_diff for diff in medians_diff])
```

[124]:
```
np.float64(0.0)
```

[125]:
```
plot_ser = pd.Series(medians_diff)
plot_ser.name = "median differences"
fig = px.histogram(plot_ser, title="Permutation Test for Difference in␣
↪Medians", labels={"value": "Difference in Medians", "0": "Median␣
↪Difference"})
fig.add_vline(x=observed_diff, line_width=3, line_dash="dash", line_color="red")
fig.update_layout(width=800, height=600)
```

```
fig.show()
```

[126]:
```
fig.write_html(os.path.join(website_folder, "hypothesis_test.html"),␣
↪include_plotlyjs='cdn')
```

1.5 Step 5: Framing a Prediction Problem

While Outage Duration is a good classifier for how extreme an outage is, most companies would be more interested in the effects it has on customers and how whether they are more likely to complain. Thus, we will be using the given data and predicting how many customers are affected. This will help the companies to identify events that are more likely to affect more people. From there, the companies may seek to create new methods to counter act on these specific predictive variables.

Looking at inital models, I saw that much of the data seems very much skewed by extremely low values that happen very often and a few outlier high values. This has resulted in very high RMSE values and overall a bad predictor for what is creates a high amount of affected customers. To remedy this, I will do a classification model and define a new variable "High_Risk_Customers" as a Binary classification of whether a certain event will have more than a certain number of affected customers thus the model will focus on identifying these communities and risk factors rather than trying to accurately predicting the values.

1.6 Step 6: Baseline Model

[127]:
```
base_pred_model = data_time.copy()
base_pred_model = base_pred_model[["NERC.REGION",
                                   "MONTH",
                                   "CLIMATE.REGION",
                                   "CLIMATE.CATEGORY",
                                   "ANOMALY.LEVEL",
                                   "CAUSE.CATEGORY",
                                   "RES.PERCEN",
                                   "COM.PERCEN",
                                   "IND.PERCEN",
                                   "RES.CUSTOMERS",
                                   "COM.CUSTOMERS",
                                   "IND.CUSTOMERS",
                                   "POPULATION",
                                   "POPPCT_URBAN",
                                   "CUSTOMERS.AFFECTED"]]
base_pred_model.dropna(inplace=True)
base_pred_model["CUSTOMERS.AFFECTED"] = base_pred_model["CUSTOMERS.AFFECTED"].
↪apply(lambda x: int(x>150_000))
```

[128]:
```
base_pred_model["CUSTOMERS.AFFECTED"].sum()
```

[128]: np.int64(261)

[129]:
```
data_time.plot.scatter(x="RES.PERCEN", y="CUSTOMERS.AFFECTED")
```

[130]:
```
data_time.plot.scatter(x="ANOMALY.LEVEL", y="CUSTOMERS.AFFECTED")
```

[131]:
```
def RMSE(y_true, y_pred):
    return np.sqrt(np.mean((y_true - y_pred) ** 2))


from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.preprocessing import FunctionTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer

import random
random.seed(10)

X = base_pred_model.drop("CUSTOMERS.AFFECTED", axis=1)
y = base_pred_model["CUSTOMERS.AFFECTED"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)

model = DecisionTreeClassifier(max_depth = 4)
log_transformer = FunctionTransformer(np.log)
exp_transformer = FunctionTransformer(np.exp)
square_transformer = FunctionTransformer(np.square)
preproc = ColumnTransformer(
    transformers=[
        ('cat', OneHotEncoder(), ['NERC.REGION',
                                  'MONTH', 'CLIMATE.REGION',
                                  'CLIMATE.CATEGORY', 'CAUSE.CATEGORY'])
    ],
    remainder='passthrough',
    force_int_remainder_cols=False,
)

pipe = Pipeline([('preproc', preproc),
                  ('model', model)])

pipe.fit(X_train, y_train)
(pipe.predict(X_test) == y_test).mean()
```

[131]: np.float64(0.7276119402985075)

[132]:
```
y_pred = pipe.predict(X_test)

TP = 0
FP = 0

for i in range (len(y_test)):
    if y_pred[i] == 1:
        if y_test.iloc[i] == 1:
            TP += 1
        else:
            FP += 1

TP/(TP+FP)
```

[132]: 0.2857142857142857

1.7 Step 7: Final Model

For this final Model, let us try to use a SVC model and see it can do better than our base model. We will apply some column transformations to some of the columns as there may be some information to gain by changing them. One of the columns changed is the Anomoly Level which is now squared. This is due to the extremes of the anaomoly level not having any high values and thus are squared so that there can be an easier split in the middle rather than at the sides where there might need to be multiple splits. We will also apply an exponential transformer to separate the higher values of the percentages which are where there are more high customers affected but there is still a lot of low class points there too so we are hoping to introduce changes within that axis so that there are greater distances in the higher values are more spread and thus will allow the support vector to split the data at a higher value so that we can get more True Positives and less False Positives.

We will also search along the tolerance levels of the model to observe what tolerances fit the model the best and to see if increasing the tolerance will affect the models performance

[144]:
```
log_transformer = FunctionTransformer(np.log)
exp_transformer = FunctionTransformer(np.exp)
square_transformer = FunctionTransformer(np.square)
preproc = ColumnTransformer(
    transformers=[
        ('num', exp_transformer, ['RES.PERCEN']),
        ('square', square_transformer, ['ANOMALY.LEVEL']),
        ('cat', OneHotEncoder(), ['NERC.REGION',
                                  'MONTH', 'CLIMATE.REGION',
                                  'CLIMATE.CATEGORY', 'CAUSE.CATEGORY'])
    ],
    remainder='passthrough',
    force_int_remainder_cols=False,
)
```

```
hyper_param = {}

for j in range(1000):
    model = SVC(degree = 3, gamma = 'auto', tol = 1e-3 * (1 + 50*j))
    pipe = Pipeline([('preproc', preproc),
                     ('model', model)])

    pipe.fit(X_train, y_train)

    y_pred = pipe.predict(X_test)

    y_pred = pipe.predict(X_test)

    TP = 0
    FP = 0

    for i in range (len(y_test)):
        if y_pred[i] == 1:
            if y_test.iloc[i] == 1:
                TP += 1
            else:
                FP += 1

    hyper_param[j] = ((pipe.predict(X_test) == y_test).mean() , TP/(TP+FP))

# hyper_param
```

[134]:
```
# Final model

model = SVC(degree = 3, gamma = 'auto', tol = 1e-3)
pipe = Pipeline([('preproc', preproc),
                     ('model', model)])

pipe.fit(X_train, y_train)

y_pred = pipe.predict(X_test)

TP = 0
FP = 0

for i in range (len(y_test)):
    if y_pred[i] == 1:
        if y_test.iloc[i] == 1:
            TP += 1
        else:
            FP += 1
```

```
TP/(TP+FP)
```

[134]: 0.5714285714285714

1.8 Step 8: Fairness Analysis

Let us now look at the fairness of the model. To do this we will be looking at the if the model predicts similarly across region in the US. We will divide the data set into 2 lables, “True” for states in the West and “False” for the States in the East and then run a permutation test seeing if the difference in the precision is statistically different between the 2 groups.

• Null Hypothesis: Our model is fair. Its precision for East and West States are roughly the same, and any differences are due to random chance. • Alternative Hypothesis: Our model is unfair. Its precision for West States is not the same as its precision for East States.

[135]:
```
# prep the data set
base_pred_model = data_time.copy()
base_pred_model = base_pred_model[["U.S._STATE",
                                   "NERC.REGION",
                                   "MONTH",
                                   "CLIMATE.REGION",
                                   "CLIMATE.CATEGORY",
                                   "ANOMALY.LEVEL",
                                   "CAUSE.CATEGORY",
                                   "RES.PERCEN",
                                   "COM.PERCEN",
                                   "IND.PERCEN",
                                   "RES.CUSTOMERS",
                                   "COM.CUSTOMERS",
                                   "IND.CUSTOMERS",
                                   "POPULATION",
                                   "POPPCT_URBAN",
                                   "CUSTOMERS.AFFECTED"]]
base_pred_model.dropna(inplace=True)
base_pred_model["CUSTOMERS.AFFECTED"] = base_pred_model["CUSTOMERS.AFFECTED"].
  ↪apply(lambda x: int(x>150_000))
```

[136]:
```
# Categorize the states as in West or not
base_pred_model["Is_West_Missippi"] = (base_pred_model["U.S._STATE"].isin([
                                                                "Alaska",
                                                                "Arizona",
                                                                "Arkansas",
                                                                "California",
                                                                "Colorado",
                                                                "Hawaii",
                                                                "Idaho",
                                                                "Iowa",
```

```
                                                                    "Kansas",
                                                                    "Louisiana",
                                                                    "Minnesota",
                                                                    "Missouri",
                                                                    "Montana",
                                                                    "Nebraska",
                                                                    "Nevada",
                                                                    "New Mexico",
                                                                    "North␣
 ↪Dakota",
                                                                    "Oklahoma",
                                                                    "Oregon",
                                                                    "South␣
 ↪Dakota",
                                                                    "Texas",
                                                                    "Utah",
                                                                    "Washington",
                                                                    "Wyoming"
                                                                   ]))
base_pred_model.drop("U.S._STATE", axis=1, inplace=True)
```

[137]:
```
X_data = base_pred_model.drop("CUSTOMERS.AFFECTED", axis=1)
y_data = base_pred_model["CUSTOMERS.AFFECTED"]
```

[138]:
```
def split_data(X, y):
    True_data_X = X[X["Is_West_Missippi"] == True]
    True_data_y = y[X["Is_West_Missippi"] == True]
    False_data_X = X[X["Is_West_Missippi"] == False]
    False_data_y = y[X["Is_West_Missippi"] == False]
    return True_data_X, True_data_y, False_data_X, False_data_y

def get_precision (y_test, y_pred):
    TP = 0
    FP = 0

    for i in range (len(y_test)):
        if y_pred[i] == 1:
            if y_test.iloc[i] == 1:
                TP += 1
            else:
                FP += 1
    if TP == 0 and FP == 0:
        return 0

    return TP/(TP+FP)

def find_diff_precision(True_data_X, True_data_y, False_data_X, False_data_y):
```

```
    True_data_y_pred = pipe.predict(True_data_X)
    False_data_y_pred = pipe.predict(False_data_X)

    True_data_precision = get_precision(True_data_y, True_data_y_pred)
    False_data_precision = get_precision(False_data_y, False_data_y_pred)

    return abs(True_data_precision - False_data_precision)



True_data_X, True_data_y, False_data_X, False_data_y = split_data(X_data,␣
 ↪y_data)
observed_diff = find_diff_precision(True_data_X, True_data_y, False_data_X,␣
 ↪False_data_y)
observed_diff
```

[138]: 0.05656565656566

[139]:
```
# Premutation test

permutation_inner_data = X_data.copy()
stats = []

for _ in range(100):
    permutation_inner_data["Is_West_Missippi"] = \
        np.random.permutation(permutation_inner_data["Is_West_Missippi"])

    True_data_X, True_data_y, False_data_X, False_data_y = \
        split_data(permutation_inner_data, y_data)

    stats.append(find_diff_precision(True_data_X,
                                     True_data_y,
                                     False_data_X,
                                     False_data_y))

(np.array(stats) > observed_diff).mean()
```

[139]: np.float64(0.25)

[140]:
```
fig = px.histogram(stats, labels={"value": "Difference in Precision",
                                  "count": "Frequency"})
fig.add_vline(x=observed_diff, line_width=3, line_dash="dash", line_color="red")
fig.show()
```

[141]:
```
fig.write_html(os.path.join(website_folder, "fairness_analysis.html"),␣
 ↪include_plotlyjs='cdn')
```

2 Auxilary Code in python files

[142]:
```
################################################################################
# Functions used in Project
################################################################################
import numpy as np
import pandas as pd
from plotly import express as px

def time_date_to_timestamp(data:pd.DataFrame) -> pd.DataFrame:
    data = data.copy()
    data["OUTAGE.START.TIME"] = pd.to_timedelta(data["OUTAGE.START.TIME"],)
    data["OUTAGE.RESTORATION.TIME"] = pd.to_timedelta(data["OUTAGE.RESTORATION.
 ↪TIME"])
    return data

def time_to_datetime(data:pd.DataFrame) -> pd.DataFrame:
    data = time_date_to_timestamp(data)
    data["OUTAGE.START.DATE"] = data["OUTAGE.START.DATE"] + data["OUTAGE.START.
 ↪TIME"]
    data["OUTAGE.RESTORATION.DATE"] = data["OUTAGE.RESTORATION.DATE"] +␣
 ↪data["OUTAGE.RESTORATION.TIME"]
    return data

def diff_medians(data:pd.DataFrame, val_col: str, label_col :str) -> float:
    return data.groupby(label_col)[val_col].median().diff().iloc[-1]
```

[143]:
```
"""
Imports and helpful functions that we use in DSC 80 lectures. Use `make
setup-lec` to copy this (and custom-rise-styles.css) to the lecture folders.

Usage:

from dsc80_utils import *
"""
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib_inline.backend_inline import set_matplotlib_formats
from IPython.display import display, IFrame, HTML

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

```python
import plotly.io as pio
pio.renderers.default = "notebook"

# DSC 80 preferred styles
pio.templates["dsc80"] = go.layout.Template(
    layout=dict(
        margin=dict(l=30, r=30, t=30, b=30),
        autosize=True,
        width=600,
        height=400,
        xaxis=dict(showgrid=True),
        yaxis=dict(showgrid=True),
        title=dict(x=0.5, xanchor="center"),
    )
)
pio.templates.default = "simple_white+dsc80"

set_matplotlib_formats("svg")
sns.set_context("poster")
sns.set_style("whitegrid")
plt.rcParams["figure.figsize"] = (10, 5)

# display options for numpy and pandas
np.set_printoptions(threshold=20, precision=2, suppress=True)
pd.set_option("display.max_rows", 7)
pd.set_option("display.max_columns", 8)
pd.set_option("display.precision", 2)

# Use plotly as default plotting engine
pd.options.plotting.backend = "plotly"


def display_df(
    df, rows=pd.options.display.max_rows, cols=pd.options.display.max_columns
):
    """Displays n rows and cols from df"""
    with pd.option_context(
        "display.max_rows", rows, "display.max_columns", cols
    ):
        display(df)


def dfs_side_by_side(*dfs):
    """
    Displays two or more dataframes side by side.
    """
    display(
```

```
        HTML(
            f"""
        <div style="display: flex; gap: 1rem;">
        {''.join(df.to_html() for df in dfs)}
        </div>
    """
        )
    )

from pathlib import Path

# The stuff below is for Lecture 7/8.
def create_kde_plotly(df, group_col, group1, group2, vals_col, title=''):
    fig = ff.create_distplot(
        hist_data=[df.loc[df[group_col] == group1, vals_col], df.
 ↪loc[df[group_col] == group2, vals_col]],
        group_labels=[group1, group2],
        show_rug=False, show_hist=False
    )
    return fig.update_layout(title=title)

def multiple_hists(df_map, histnorm="probability", title=""):
    values = [df_map[df_name]["child"].dropna() for df_name in df_map]
    all_sets = pd.concat(values, keys=list(df_map.keys()))
    all_sets = all_sets.reset_index()[["level_0", "child"]].rename(
        columns={"level_0": "dataset"}
    )
    fig = px.histogram(
        all_sets,
        color="dataset",
        x="child",
        barmode="overlay",
        histnorm=histnorm,
    )
    fig.update_layout(title=title)
    return fig


def multiple_kdes(df_map, title=""):
    values = [df_map[key]["child"].dropna() for key in df_map]
    labels = list(df_map.keys())
    fig = ff.create_distplot(
        hist_data=values,
        group_labels=labels,
        show_rug=False,
        show_hist=False,
        colors=px.colors.qualitative.Dark2[: len(df_map)],
```

```python
    )
    return fig.update_layout(title=title).update_xaxes(title="child")

def multiple_describe(df_map):
    out = pd.DataFrame(
        columns=["Dataset", "Mean", "Standard Deviation"]
    ).set_index("Dataset")
    for key in df_map:
        out.loc[key] = df_map[key]["child"].apply(["mean", "std"]).to_numpy()
    return out

def make_mcar(data, col, pct=0.5):
    """Create MCAR from complete data"""
    missing = data.copy()
    idx = data.sample(frac=pct, replace=False).index
    missing.loc[idx, col] = np.NaN
    return missing


def make_mar_on_cat(data, col, dep_col, pct=0.5):
    """Create MAR from complete data. The dependency is
    created on dep_col, which is assumed to be categorical.
    This is only *one* of many ways to create MAR data.
    For the lecture examples only."""

    missing = data.copy()
    # pick one value to blank out a lot
    high_val = np.random.choice(missing[dep_col].unique())
    weights = missing[dep_col].apply(lambda x: 0.9 if x == high_val else 0.1)
    idx = data.sample(frac=pct, replace=False, weights=weights).index
    missing.loc[idx, col] = np.NaN

    return missing


def make_mar_on_num(data, col, dep_col, pct=0.5):
    """Create MAR from complete data. The dependency is
    created on dep_col, which is assumed to be numeric.
    This is only *one* of many ways to create MAR data.
    For the lecture examples only."""

    thresh = np.percentile(data[dep_col], 50)

    def blank_above_middle(val):
        if val >= thresh:
            return 0.75
        else:
```

```
            return 0.25

    missing = data.copy()
    weights = missing[dep_col].apply(blank_above_middle)
    idx = missing.sample(frac=pct, replace=False, weights=weights).index

    missing.loc[idx, col] = np.NaN
    return missing

def permutation_test(data, col, group_col, test_statistic, N=1000):
    """
    Conduct a permutation test to compare two groups based on a given test
    statistic.

    This function computes the observed test statistic for the two groups in the
    dataset, and then generates a distribution of permuted test statistics by
    repeatedly shuffling the group labels and calculating the test statistic on
    the shuffled data. The result is a distribution of permuted statistics and
    the observed statistic for comparison.

    Parameters
    ----------
    data : pd.DataFrame
        The input DataFrame containing the data to be tested, along with the
        group labels.
    col : str
        The name of the column in `data` that contains the data values to be
        compared between the two groups.
    group_col : str
        The name of the column in `data` that contains the group labels. There
        should be exactly two unique groups in this column.
    test_statistic : function
        A function that calculates the test statistic based on the data column
        and the group column. This function must accept three arguments: the
        data DataFrame, the name of the data column, and the name of the group
        column.
    N : int, optional (default=1000)
        The number of permutations to perform in the test.

    Returns
    -------
    shuffled_stats : np.ndarray
        An array of test statistics computed from the permuted datasets.
    obs : np.floating
        The observed test statistic calculated from the original data.

    Example
```

```
    -------
    >>> import numpy as np
    >>> import pandas as pd
    >>> def mean_diff(data, col, group_col):
    ...     group_means = data.groupby(group_col)[col].mean()
    ...     return np.abs(group_means.iloc[0] - group_means.iloc[1])
    >>> data = pd.DataFrame({
    ...     'value': [1, 2, 3, 4, 5, 6],
    ...     'group': ['A', 'A', 'A', 'B', 'B', 'B']
    ... })
    >>> perm_stats, obs_stat = permutation_test(data, 'value', 'group',␣
↪mean_diff)
    """
    obs = test_statistic(data, col, group_col)
    shuffled = data.copy()

    shuffled_stats = []
    for _ in range(N):
        shuffled[col] = np.random.permutation(shuffled[col])
        shuffled_stat = test_statistic(shuffled, col, group_col)
        shuffled_stats.append(shuffled_stat)

    shuffled_stats = np.array(shuffled_stats)

    return shuffled_stats, obs

def diff_in_means(data, col, group_col):
    """
    Compute the difference in means between two groups.

    This function calculates the difference in means of the values in the
    specified column between two groups defined by the group column.

    Parameters
    ----------
    data : pandas.DataFrame
        The input DataFrame containing the data and group labels.
    col : str
        The name of the column in `data` that contains the numeric data for
        which the mean will be computed.
    group_col : str
        The name of the column in `data` that contains the group labels. There
        should be exactly two unique groups in this column.

    Returns
    -------
    float
```

```
        The difference in means between the two groups. The result is
        calculated as mean(group2) - mean(group1), where the group ordering is
        based on their appearance in the DataFrame.

    Example
    -------
    >>> import pandas as pd
    >>> data = pd.DataFrame({
    ...     'value': [1, 2, 3, 4, 5, 6],
    ...     'group': ['A', 'A', 'A', 'B', 'B', 'B']
    ... })
    >>> diff_in_means(data, 'value', 'group')
    np.float64(3.0)
    """
    return data.groupby(group_col)[col].mean().diff().iloc[-1]


def tvd(data, col, group_col):
    """
    Compute the Total Variation Distance (TVD) between two categorical
    distributions.

    The Total Variation Distance (TVD) measures the difference between the
    distributions of a categorical variable across two groups. It is defined as
    half the sum of the absolute differences between the group-wise proportions
    for each category.

    Parameters
    ----------
    data : pandas.DataFrame
        The input DataFrame containing the data and group labels.
    col : str
        The name of the column in `data` that contains the categorical data.
    group_col : str
        The name of the column in `data` that contains the group labels. There
        should be exactly two unique groups in this column.

    Returns
    -------
    float
        The Total Variation Distance (TVD) between the two distributions. A
        value of 0 indicates that the distributions are identical, while a value
        of 1 indicates that they are completely disjoint.

    Example
    -------
    >>> import pandas as pd
```

```python
    >>> data = pd.DataFrame({
    ...     'category': ['X', 'X', 'Y', 'Y', 'Z', 'Z'],
    ...     'group': ['A', 'A', 'B', 'B', 'B', 'A']
    ... })
    >>> tvd(data, 'category', 'group')
    np.float64(0.6666666666666666)
    """

    tvd = (
        data.pivot_table(
            index=col, columns=group_col, aggfunc="size", fill_value=0
        )
        .apply(lambda x: x / x.sum())
        .diff(axis=1)
        .iloc[:, -1]
        .abs()
        .sum()
        / 2
    )

    return tvd


def ks(data, col, group_col):
    """
    Compute the Kolmogorov-Smirnov (KS) statistic between two distributions.

    The Kolmogorov-Smirnov (KS) statistic is used to measure the distance
    between the empirical distribution functions of two samples. This function
    applies the KS test to compare the distributions of a numeric column between
    two groups.

    Parameters
    ----------
    data : pandas.DataFrame
        The input DataFrame containing the data and group labels.
    col : str
        The name of the column in `data` that contains the numeric data to
        compare.
    group_col : str
        The name of the column in `data` that contains the group labels. There
        should be exactly two unique groups in this column.

    Returns
    -------
    float
        The Kolmogorov-Smirnov (KS) statistic, which measures the maximum
```

```python
        distance between the two empirical cumulative distribution functions. A
        higher value indicates greater dissimilarity between the distributions.

    Example
    -------
    >>> import pandas as pd
    >>> data = pd.DataFrame({
    ...     'value': [1, 2, 3, 4, 5, 6],
    ...     'group': ['A', 'A', 'A', 'B', 'B', 'B']
    ... })
    >>> ks(data, 'value', 'group')
    np.float64(0.6666666666666666)
    """

    from scipy.stats import ks_2samp

    # should have only two values in column
    valA, valB = data[group_col].unique()
    ks, _ = ks_2samp(
        data.loc[data[group_col] == valA, col],
        data.loc[data[group_col] == valB, col],
    )

    return ks
```