EDA in Health Care Sector

**Exploratory Data Analysis (EDA) for data science and ML**

## ****Table of Contents****[¶](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Table-of-Contents)

1. [Objectives](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Objectives)
2. [Setup](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Setup)
   1. [Install required libraries](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Install-required-libraries)
   2. [Import required libraries](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Import-required-libraries)
3. [Regression](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Regression)
   1. [Load the diabetes data set](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Load-the-diabetes-data-set)
   2. [Add some missing values](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Add-some-missing-values)
   3. [Initial data preprocessing](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Initial-data-preprocessing)
      1. [One-hot encoding](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#One-hot-encoding)
      2. [Make a train-test split](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Make-a-train-test-split)
   4. [Perform EDA](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Perform-EDA)
      1. [A look at the beginning and end of the data set](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#A-look-at-the-beginning-and-end-of-the-data-set)
         1. [Exercise 1](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Exercise-1)
      2. [Describe the DataFrame](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Describe-the-DataFrame)
         1. [Exercise 2](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Exercise-2)
      3. [Missing values](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Missing-values)
         1. [Exercise 3](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Exercise-3)
         2. [Exercise 4](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Exercise-4)
      4. [Drop missing observations](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Drop-missing-observations)
      5. [Fill in missing values with the mean](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Fill-in-missing-values-with-the-mean)
      6. [Fill in missing values with the median](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Fill-in-missing-values-with-the-median)
         1. [Exercise 5](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Exercise-5)
      7. [Histograms and boxplots](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Histograms-and-boxplots)
         1. [Exercise 6](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Exercise-6)
      8. [Correlation matrix](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Correlation-matrix)
      9. [Pair plots](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Pair-plots)
      10. [A simple function to perform EDA](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#A-simple-function-to-perform-EDA)
4. [Classification](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Classification)
   1. [Import the iris data set](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Import-the-iris-data-set)
   2. [Perform EDA on the iris data set](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Perform-EDA-on-the-iris-data-set)
      1. [Exercise 7](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Exercise-7)
      2. [Exercise 8](https://jupyterlabnext-0-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/saitejapuppa/lab/tree/GPXX0HE1EN/EDA%20lab.ipynb#Exercise-8)

## Objectives

After completing this lab you are able to:

* Perform EDA using a set of very simple and easy-to-memorize Python commands
* Interpret key EDA plots and statistics
* Improve a prediction model by using information obtained through EDA
* Perform basic feature engineering
* Detect and handle outliers
* Deal with missing data

For this lab, we use the following libraries:

* [pandas](https://pandas.pydata.org/?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMML0187ENSkillsNetwork31430127-2021-01-01) for managing the data
* [NumPy](https://numpy.org/?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMML0187ENSkillsNetwork31430127-2021-01-01) for mathematical operations
* [SciPy](https://scipy.org/) for additional mathematical operations beyond those provided by numpy
* [sklearn](https://scikit-learn.org/stable/?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMML0187ENSkillsNetwork31430127-2021-01-01) for machine learning and machine learning pipeline-related functions
* [seaborn](https://seaborn.pydata.org/?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMML0187ENSkillsNetwork31430127-2021-01-01) for visualizing the data
* [Matplotlib](https://matplotlib.org/?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMML0187ENSkillsNetwork31430127-2021-01-01) for additional plotting tools
* [MissingNo](https://github.com/ResidentMario/missingno) for missing data analysis and visualizations
* [fasteda](https://github.com/Matt-OP/fasteda) for fast and easy EDA

## Load the diabetes data set

The following code loads the diabetes data set from sklearn and prints out a description of the data.

Clearly, you have 10 features and a target column. As was mentioned previously, the target is a quantitative measure of diabetes progression one year after a baseline diabetes level was established. Your task is to predict this measure of disease progression on the basis of the observed features.

Most of the features are numeric, except for `sex`, which is a categorical feature with `1.0` representing either female or male (it is not clear from the data description) and `2.0` representing the opposite sex. Variables `s1` through `s6` are as defined in the data description.

# Add some missing values

The original data set does not contain any missing values. To verify this, run the following code, which outputs True if there are missing values and False otherwise.

To derive the most benefit from this EDA exercise, you introduce some missing values. Specifically, you select 3 feature columns at random, and set them to missing for 10% of randomly selected rows.

## Initial data preprocessing

In this step, you perform some initial data preprocessing necessary to use the data in a prediction model.

In a typical workflow, data preprocessing is usually conducted after EDA. After all, how can one determine the necessary preprocessing steps without thoroughly examining the data first? However, in this case, you're moving these initial preprocessing steps earlier to allow for building multiple prediction models based on insights gained from EDA, aiming to enhance the predictive performance of each successive model. Note that some level of preprocessing is essential for building any useful model. Therefore, you'll carry out this initial preprocessing here, with the understanding that in a standard workflow, these steps would typically occur after EDA.

### One-hot encoding

Recall that the sex column encodes female and male with 1.0 and 2.0, where it is not clear from the data set documentation which code belongs to which sex. If you were to keep this column encoded as is, then prediction models would interpret this column as having a particular ordering, which is not what you want because sexes have no inherent order. Rather, it would be better to [one-hot](https://en.wikipedia.org/wiki/One-hot) encode this data, where you split the sex column into two columns: one for each category. The data transformation afforded by one-hot encoding is best illustrated by an example:

The previous table shows the original sex column, and the two columns into which the sex column was split into using one-hot encoding: sex1 and sex2. Note that when sex takes the value of 1.0, the sex1 column takes the value of 1.0 and the sex2 column takes the value of 0.0. Likewise, when sex takes the value of 2.0, the sex1 column takes the value of 0.0 and the sex2 column takes the value of 1.0. So one-hot encoding essentially splits a categorical variable into a series of indicator columns, where the indicator is equal to 1.0 if the category is present and 0.0 otherwise.

Now, instead of keeping both the sex1 and sex2 columns in your new data set, you only keep one column, say sex1, because after you have one of these columns, the other column does not add any useful information to the model. In other words, when sex1 is equal to 1.0, then sex2 must be equal to 0.0, and vice versa. Essentially, this is equivalent to saying that, if you knew that an individual is female, then you know that they are not male, and vice versa. Consequently, you would need to keep only one of the columns, either the one for female or the one for male, because the other column does not give you any useful information after you have the first column. In this case, keep the first column, sex1, and drop sex and sex2 from the data set.

# Drop 'sex' and 'sex2'

diabetes = diabetes.drop(['sex', 'sex2'], axis=1)

# Reorder columns to have 'sex1' where 'sex' used to be

diabetes = diabetes.loc[:, ['age', 'sex1', 'bmi', 'bp', 's1', 's2', 's3', 's4', 's5', 's6', 'target']]

# Print a sample of 5 rows

diabetes.sample(5)

You see that the original sex column is now encoded with an indicator variable: wherever sex was equal to 1.0, sex1 is 1.0, and whenever sex was equal to 2.0, sex1 is equal to 0.0. This no longer implies ordered data, but rather the presence of a particular category: when sex was 1.0, sex1 is 1.0 (in other words, the sex == 1 category is present), and when sex was 2.0, sex1 is 0.0 (in other words, the sex == 1 category was absent).

### Make a train-test split

Before actually performing EDA, split the data set into a training set and a test set. The training set is what you train your prediction models on, and the test set is the unseen data on which you make predictions. You evaluate the performance of your models by using its ability to predict the unseen test set. Run the following code to randomly assign 33% of the rows to the test set, and the remaining 67% of the rows to the training set.

# Make a train-test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(diabetes.iloc[:, :-1], diabetes.iloc[:, [-1]], test\_size=0.33, random\_state=2024)

The resulting objects contain the following data:

* X\_train are the feature columns in the training set.
* X\_test are the feature columns in the test set.
* y\_train is the target column for the training set.
* y\_test is the target column for the test set.

## Perform EDA

Finally, you are ready to perform EDA on the diabetes data set!

### A look at the beginning and end of the data set

EDA typically starts with a look at the beginning (the head) and the end (the tail) of a data set. Run the following code to look at the beginning of the data set.

### Describe the DataFrame

After examining both the start and end of the data set, it is typically beneficial to generate statistical summaries, including the mean and standard deviation, for your columns. Proceed with the following exercise to statistically describe the data set.

#### Exercise 2

Given that the describe method of a pandas.DataFrame describes a data set, write code in the following cell that describes the diabetes data set.

# Type your solution here:

diabetes.describe()

Click here for solution

The description table provides many useful statistics regarding the data set:

* The count row indicates the number of non-missing observations in each column.
* The mean row provides, for each column, the [arithmetic mean](https://en.wikipedia.org/wiki/Arithmetic_mean) for all non-missing observations.
* The std row provides, for each column, the [standard deviation](https://en.wikipedia.org/wiki/Standard_deviation) for all non-missing observations.
* The min row provides, for each column, the minimum value for all non-missing observations.
* The 25%, 50%, and 75% rows provide [quartiles](https://en.wikipedia.org/wiki/Quartile) for each column.
* The max row provides, for each column, the maximum value for all non-missing observations.

The description table offers initial hints about potential anomalies in the data. For example, if a column's maximum value is unusually high compared to its mean and standard deviation, it might suggest the presence of an outlier or measurement error, requiring further investigation.

### Missing values

The next step is to look for missing values.

#### Exercise 3

Consider the following facts:

* diabetes is a pandas.DataFrame.
* isna() is a pandas.DataFrame method that returns another pandas.DataFrame with True where values are missing and False otherwise.
* sum() is a method that returns the sum across all rows for every column in a pandas.DataFrame.
* When summing binary data in Python, True takes the interger value 1, and False takes the interger value 0.

Given the previous information, write one line of code that shows the number of missing values for every column in the diabetes DataFrame.

# Type your solution here:

diabetes.isna().sum()

Click here for solution

The previous exercise indicated that the bmi, s1, and s4 columns had 44 observations missing, with no missing observations for any of the other columns. Note that this aligns with your expectations. After all, you randomly selected bmi, s1, and s4 and introduced some missing values into these columns manually.

Now that you have established the presence of missing values in the data set, it is crucial to determine whether these absences occur randomly or if there is a pattern associated with observations containing missing values. One straightforward yet often effective method to assess this is by examining the missing value matrix, which you'll delve into in the next exercise.

#### Exercise 4

The missingno package, imported as msno, has a function named matrix(), which accepts a pandas.DataFrame as the first positional argument. Call the matrix() function on the diabetes DataFrame.

# Type your solution here:

msno.matrix(diabetes)

Click here for solution

The missing value matrix arranges all observations vertically, starting from the first row in the diabetes DataFrame at the top and descending to the last row at the bottom. Due to the compact display of 442 rows, there's no visible gap between them. However, rows with missing values are distinguishable. For example, it's noticeable that the initial rows, such as the first 25, contain no missing data. This can be confirmed by identifying the index number of the first row with a missing bmi value:

diabetes.index[diabetes['bmi'].isna()][0]

You can see that, in fact, the first 29 rows do not have missing values.

But the missing value matrix provides you with even more information in a very clear and concise way. It is easy to verify, thanks to this matrix, that whenever bmi is missing, s1 and s4 are also missing, and whenever bmi is not missing, s1 and s4 are not missing! This is confirmed by the **sparkline** on the right, which indicates the number of columns for which data is not missing for each observation. Here, you can clearly see that, for all observations, there are either 11 columns with non-missing data, or 8 columns with non-missing data (this applies to observations that have bmi, s1, and s4 missing). If you saw a data set like this without knowing how the missing values were created, you would immediately think that the observations are not missing at random. In other words, you would think that there is something about the observations with missing data that sets them apart in a systematic way from the observations that do not have missing data.

A well-trained data scientist might conclude, on the basis of this missing value matrix and associated sparkline, that the missing observations likely came from a different data collection system, such as a different hospital that did not record bmi, s1 and s4 for its observations. This is because it would be extremely unlikely for s1 and s4 to be missing if and only if bmi was missing by random chance alone. Hence, there must be something systematic that sets those observations apart from the rest. Consequently, such a data scientist would not believe that the observations with missing data belong to the same group as those without missing data, and would consequently employ various strategies for dealing with grouped data.

However, you have the luxury of knowing exactly how the data for these observations was set to missing. You know that you did this at random, first selecting 3 columns at random, then 10% of the rows at random, and then assigning missing values to the intersections of these rows and columns. As such, you know that, by construction, the data is missing at random. Therefore, you proceed forward on the basis of this knowledge. However, this exercise highlights the usefulness of the missing value matrix, which provides you with not only basic statistics regarding missing values, but also a visual structure of the missingness of the data.

There are typically three different strategies for dealing with data that is missing at random:

* Dropping the observations with missing data
* Filling the observations with missing data with the mean
* Filling the observations with missing data with the median

### Drop missing observations

The simplest strategy for dealing with rows that contain missing values is to drop them from the data set. This is the most conservative approach because you do not speculate about the value of missing data, you just disregard observations with missing feature values entirely. The following code trains a [linear regression](https://en.wikipedia.org/wiki/Linear_regression) model on a data set that had dropped the rows with missing data from the data set. To evaluate the performance of this model, you also calculate the [root mean squared error](https://en.wikipedia.org/wiki/Root-mean-square_deviation) (RMSE) of the prediction from the true value for the test set:

# linear regression dropping NANs

# Get NAN indices

nonnan\_train\_indices = X\_train.index[~X\_train.isna().max(axis=1)]

nonnan\_test\_indices = X\_test.index[~X\_test.isna().max(axis=1)]

# Fit an instance of `LinearRegression`

reg = LinearRegression().fit(X\_train.loc[nonnan\_train\_indices], y\_train.loc[nonnan\_train\_indices])

# Generate predictions

pred = reg.predict(X\_test.loc[nonnan\_test\_indices])

# Calculate the root mean squared error

root\_mean\_squared\_error(y\_test.loc[nonnan\_test\_indices],pred)

You end up with an RMSE of around 56.03.

How good is an RMSE of 56.03? Well, you won't really know unless you compare the performance of this model to a few different model specifications. See if you can improve on the performance of this baseline prediction model using the information obtained by EDA.

### Fill in missing values with the mean

One alternative to dropping all of the rows that contain missing data is to fill in missing values using either the mean or the median for the non-missing values in the column. Let's do exactly that, and run a regression model on the resulting data set.

# linear regression with mean fill

# Get NAN indices

nonnan\_train\_indices = X\_train.index[~X\_train.isna().max(axis=1)]

nonnan\_test\_indices = X\_test.index[~X\_test.isna().max(axis=1)]

# Initialize the simple imputer

imp\_mean = SimpleImputer(missing\_values=np.nan, strategy='mean')

# Fit the simple imputer using the training data

imp\_mean.fit(X\_train)

# Actually mean fill the training data

X\_train\_mean\_filled = imp\_mean.transform(X\_train)

# Fit an instance of `LinearRegression`

reg = LinearRegression().fit(X\_train\_mean\_filled, y\_train)

# Generate predictions

pred = reg.predict(X\_test.loc[nonnan\_test\_indices])

# Calculate the root mean squared error

root\_mean\_squared\_error(y\_test.loc[nonnan\_test\_indices],pred)

First, it's important to recognize that the 'SimpleImputer' instance 'imp\_mean' was exclusively trained on the training data. Training the imputer on the entire data set would lead to a leakage of the test set into the imputer's mean calculation. However, the test set should represent unseen data, not just for your linear regression model but also for the imputer. Therefore, it's crucial to restrict the training of the mean imputer to the training data only.

Second, it's essential to note that you chose not to impute missing values in the test set. This decision was made to ensure that you predict the exact same rows using both your mean imputation strategy and the strategy of dropping missing values. Imputing missing values in the test set would effectively expand its size, contaminating the comparison between the two models' performances.

Finally, it is worth highlighting that the mean imputation strategy resulted in a lower RMSE compared to simply dropping rows with missing data. In this specific scenario, employing a mean imputation strategy enhanced prediction accuracy. However, it is important to recognize that this outcome will not always occur: mean filling does not guarantee improved performance over dropping rows with missing data, and the data scientist must experiment with different strategies to minimize the generalization error.

### Fill in missing values with the median

Instead of filling in the missing value with the mean, you can also fill it in using the median. Complete the following exercise to fill in missing values using the median.

#### Exercise 5

Replace the part that says ### REPLACE WITH YOUR CODE ### with your code to fill in missing values with the median. The ### REPLACE WITH YOUR CODE ### part is found under the # Initialize the simple imputer header.

# linear regression with median fill

# Get NAN indices

nonnan\_train\_indices = X\_train.index[~X\_train.isna().max(axis=1)]

nonnan\_test\_indices = X\_test.index[~X\_test.isna().max(axis=1)]

# Initialize the simple imputer

imp\_median = ### REPLACE WITH YOUR CODE ###

# Fit the simple imputer using the training data

imp\_median.fit(X\_train)

# Actually median fill the training data

X\_train\_median\_filled = imp\_median.transform(X\_train)

# Fit an instance of `LinearRegression`

reg = LinearRegression().fit(X\_train\_median\_filled, y\_train)

# Generate predictions

pred = reg.predict(X\_test.loc[nonnan\_test\_indices])

# Calculate the root mean squared error

root\_mean\_squared\_error(y\_test.loc[nonnan\_test\_indices],pred)

Click here for solution

As observed, median filling yields an RMSE of 55.73, which is practically identical (allowing for rounding errors up to 2 decimal places) to the RMSE obtained with the mean fill strategy. Given their comparable performance, the choice between these strategies is subjective and lies with the data scientist. Considering that median filling is less influenced by extreme values, you opt for median filling in this instance. However, it's worth noting that choosing mean filling would have been equally acceptable.

### Histograms and boxplots

The following code generates [histograms](https://en.wikipedia.org/wiki/Histogram) and [boxplots](https://en.wikipedia.org/wiki/Box_plot) for all numerical columns in the data set (that is, for all columns except sex1). Note that the following code can take more than 10 seconds to produce the plots, and that a significant number of plots will be created.

for idx, col in enumerate([i for i in diabetes.columns if i != 'sex1']):

fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (14, 6))

sns.histplot(diabetes, x = diabetes[col], kde = True,

color = sns.color\_palette("hls", len([i for i in diabetes.columns if i != 'sex1']))[idx], ax = ax1)

sns.boxplot(x = diabetes[col], width = 0.4, linewidth = 3, fliersize = 2.5,

color = sns.color\_palette("hls", len([i for i in diabetes.columns if i != 'sex1']))[idx], ax = ax2)

fig.suptitle(f"Histogram and Boxplot of {col}", size = 20, y = 1.02)

plt.show()

Histograms, shown towards the left, bin the data into equally sized bins and then show you the counts of the data in each bin. In addition to the histogram, each of the diagrams on the left provides a smoothed estimate of the density using a curve. These histograms and smoothed curves provide data scientists with a lot of information regarding the distribution of the individual variables. For instance, using histograms, you can visually confirm whether a variable is more or less normally distributed, or whether the data is skewed in a particular direction.

The associated boxplots to the right indicate the ranges in which the bulk of the data lives (the box), the extent of the data excluding outliers (the whiskers), and the outliers (the fliers sometimes seen beyond the whiskers). Whether an extreme point is considered an outlier is determined using the [interquartile range](https://en.wikipedia.org/wiki/Interquartile_range). Note that for some of the plots there are no outliers. Boxplots can be used to help identify the presence of outliers and to indicate whether the data is skewed in a particular direction.

Use the information obtained using the histograms to try to improve your prediction model. In particular, focus on the s3 variable, which is a measure of the high-density lipid (HDL) cholesterol in the bloodstream. For simplicity, this histogram is reproduced in the following example.

for idx, col in enumerate(['s3']):

fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (14, 6))

sns.histplot(diabetes, x = diabetes[col], kde = True,

color = sns.color\_palette("hls", len(['s3']))[idx], ax = ax1)

sns.boxplot(x = diabetes[col], width = 0.4, linewidth = 3, fliersize = 2.5,

color = sns.color\_palette("hls", len(['s3']))[idx], ax = ax2)

fig.suptitle(f"Histogram and Boxplot of {col}", size = 20, y = 1.02)

plt.show()

Note that the distribution appears far from normal. Notably, there is a long right tail indicating some patients with considerably higher HDL readings than the median patient. Additionally, according to basic statistical principles, a normal distribution extends from negative infinity to positive infinity. Therefore, for s3 to follow a normal distribution, it would need to theoretically permit negative readings. However, because s3 cannot be negative due to the nature of cholesterol measurement, this distribution cannot be considered normal. A glance at the boxplot reinforces the view that s3 deviates from normality. Finally, to confirm your suspicions, you could employ a statistical test of normality, such as the [Shapiro-Wilk test](https://en.wikipedia.org/wiki/Shapiro%E2%80%93Wilk_test).

# normality test on s3

stat, p = shapiro(X\_train['s3'])

print('Statistics=%.3f, p=%.3f' % (stat, p))

# interpret

alpha = 0.05

if p > alpha:

print('Sample looks normally distributed (fail to reject H0)')

else:

print('Sample does not look normally distributed (reject H0)')

Thus, you can clearly see that s3 is not normally distributed using a combination of:

* EDA (your analysis of the histogram and boxplot)
* Domain knowledge about what this variable represents
* A statistical test

However, the histogram does indicate that the distribution might be [log-normal](https://en.wikipedia.org/wiki/Log-normal_distribution), meaning that a natural logarithm of s3 might be normally distributed. Let's confirm whether this is likely the case using the Shapiro-Wilk test in the following exercise.

#### Exercise 6

Replace the part that says ### REPLACE WITH YOUR CODE ### to run the Shapiro-Wilk test on the natural logarithm of s3. Note that a natural logarithm could be taken using the log() function from the NumPy package.

# normality test on log of s3

stat, p = shapiro.log((X\_train['s3']))

print('Statistics=%.3f, p=%.3f' % (stat, p))

# interpret

alpha = 0.05

if p > alpha:

print('Sample looks normally distributed (fail to reject H0)')

else:

print('Sample does not look normally distributed (reject H0)')

Click here for solution

In this case, you see that, according to the Shapiro-Wilk test, the log of s3 is normally distributed!

Now, to benefit from this information, let's provide some context that is slightly outside the scope of this project. For a linear regression model to be valid, you need the residuals resulting from the prediction to be normally distributed. A violation of this normality assumption could occur if the data is not linearly related or if some of the features (or the target itself) are not normally distributed. Using this fact and the knowledge that you just obtained through EDA, you can realize that you can probably improve your prediction model by simply taking the log of s3. Note that you will continue to fill in missing values with the median, as you have done before.

# linear regression with median fill, and using the log of 's3'

X\_train['s3'] = np.log(X\_train['s3'])

X\_test['s3'] = np.log(X\_test['s3'])

# Get NAN indices

nonnan\_train\_indices = X\_train.index[~X\_train.isna().max(axis=1)]

nonnan\_test\_indices = X\_test.index[~X\_test.isna().max(axis=1)]

# Initialize the simple imputer

imp\_median = SimpleImputer(missing\_values=np.nan, strategy='median')

# Fit the simple imputer using the training data

imp\_median.fit(X\_train)

# Actually median fill the training data

X\_train\_median\_filled = imp\_median.transform(X\_train)

# Fit an instance of `LinearRegression`

reg = LinearRegression().fit(X\_train\_median\_filled, y\_train)

# Generate predictions

pred = reg.predict(X\_test.loc[nonnan\_test\_indices])

# Calculate the root mean squared error

root\_mean\_squared\_error(y\_test.loc[nonnan\_test\_indices],pred)

Just as suspected, you have an improvement in performance! Simply replacing s3 with the natural logarithm of s3 caused the RMSE to drop from 55.73 to 55.54. Thus, you have been able to incrementally improve your prediction model thanks to your keen eye when looking at the histogram, your knowledge of the log-normal distribution, and your knowledge of the inner workings of the linear regression model!

Can you do even better? Of course you can! For instance, you can use the box plots to eliminate observations that contain outliers from the training set. For example, look at the histogram and boxplot for s2:

for idx, col in enumerate(['s2']):

fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (14, 6))

sns.histplot(diabetes, x = diabetes[col], kde = True,

color = sns.color\_palette("hls", len(['s2']))[idx], ax = ax1)

sns.boxplot(x = diabetes[col], width = 0.4, linewidth = 3, fliersize = 2.5,

color = sns.color\_palette("hls", len(['s2']))[idx], ax = ax2)

fig.suptitle(f"Histogram and Boxplot of {col}", size = 20, y = 1.02)

plt.show()

Here, notice that you have extreme outliers that are very far from the bulk of the observations. Let's remove some of these outliers from the training set to see if you can improve your prediction further. The following code removes just one of those outliers, the most extreme observation.

# linear regression with median fill, using the log of 's3', and removal of outlier from 's2':

# Outlier removal

X\_train\_nonoutlier\_idx = X\_train.index[X\_train.s2 < X\_train.s2.quantile(0.999)]

X\_train = X\_train.loc[X\_train\_nonoutlier\_idx]

y\_train = y\_train.loc[X\_train\_nonoutlier\_idx]

# Get NAN indices

nonnan\_train\_indices = X\_train.index[~X\_train.isna().max(axis=1)]

nonnan\_test\_indices = X\_test.index[~X\_test.isna().max(axis=1)]

# Initialize the simple imputer

imp\_median = SimpleImputer(missing\_values=np.nan, strategy='median')

# Fit the simple imputer using the training data

imp\_median.fit(X\_train)

# Actually median fill the training data

X\_train\_median\_filled = imp\_median.transform(X\_train)

# Fit an instance of `LinearRegression`

reg = LinearRegression().fit(X\_train\_median\_filled, y\_train)

# Generate predictions

pred = reg.predict(X\_test.loc[nonnan\_test\_indices])

# Calculate the root mean squared error

root\_mean\_squared\_error(y\_test.loc[nonnan\_test\_indices],pred)

Once again, you were able to improve the accuracy of the prediction, this time from 55.54 to 55.37. Note that, prior to engaging in EDA, your RMSE was 56.02 with the baseline model that simply eliminated rows with missing data. Throughout this entire analysis, you did not change the model itself or tune any hyperparameters. Rather, you took advantage of your increasing knowledge of the data set to incrementally improve your prediction model by transforming the data and by subsetting the training set to those observations that are most likely to make a model that generalizes well.

### Correlation matrix

The following code generates a correlation matrix.

plt.figure(figsize = (12, 8))

sns.heatmap(diabetes.corr(), annot = True, cmap = "Spectral", linewidths = 2, linecolor = "#000000", fmt='.3f')

plt.show()

A correlation matrix shows the [Pearson correlation coefficient](https://en.wikipedia.org/wiki/Correlation_coefficient#Pearson) between various pairs of variables. The Pearson correlation coefficient shows the strength and direction of the linear relationship between two variables, with -1 indicating a perfect negative linear relationship and +1 indicating a perfect positive linear relationship. To enhance readability, the previous correlation matrix is presented in the form of a heatmap, with high correlations being blue, and low correlations being red, as indicated by the right-hand bar beside the correlation matrix.

The previous correlation matrix indicates that s1 and s2 have a very strong positive linear relationship: their correlation is 0.901. Moreover, you see that neither s1 nor s2 are very correlated with the target, as both of their correlations are close to zero. This raises the question about whether including both s1 and s2 is necessary within the same model Perhaps the inclusion of one of these variables is sufficient. Let's test this assumption by excluding s1 from the analysis.

# linear regression with median fill, using the log of 's3', removal of outlier from 's2', and the dropping of 's1':

"""

# Outlier removal (commented out because `X\_train` and `y\_train` are already updated)

X\_train\_nonoutlier\_idx = X\_train.index[X\_train.s2 < X\_train.s2.quantile(0.999)]

X\_train = X\_train.loc[X\_train\_nonoutlier\_idx]

y\_train = y\_train.loc[X\_train\_nonoutlier\_idx]

# Get NAN indices (commented out because these indices did not change)

nonnan\_train\_indices = X\_train.index[~X\_train.isna().max(axis=1)]

nonnan\_test\_indices = X\_test.index[~X\_test.isna().max(axis=1)]

"""

# Get column names that do not include 's1'

cols\_no\_s1 = [i for i in X\_train.columns if i != 's1']

# Initialize the simple imputer

imp\_median = SimpleImputer(missing\_values=np.nan, strategy='median')

# Fit the simple imputer using the training data

imp\_median.fit(X\_train.loc[:, cols\_no\_s1])

# Actually median fill the training data

X\_train\_median\_filled = imp\_median.transform(X\_train.loc[:, cols\_no\_s1])

# Fit an instance of `LinearRegression`

reg = LinearRegression().fit(X\_train\_median\_filled, y\_train)

# Generate predictions

pred = reg.predict(X\_test.loc[nonnan\_test\_indices, cols\_no\_s1])

# Calculate the root mean squared error

root\_mean\_squared\_error(y\_test.loc[nonnan\_test\_indices],pred)

You can see that the performance of the model without s1 is only slightly worse than the performance of the model with s1. Consequently, you can draw the conclusion that s1 does not significantly improve the predictive capabilities of the linear regression model when s2 is present.

This information could be useful for you for a number of reasons. For instance, if s1 was a variable that is frequently missing or expensive to collect, it might be desirable to design a model that excludes it entirely if it does not contribute much to the predictive power of the model. Additionally, simpler models are easier to understand, and if interpretability were a goal, s1 would be a great candidate to drop from the analysis. None of these insights would have been easily obtained without looking at the correlation matrix.

### Pair plots

The following code outputs the pair plots for the diabetes data set.

# This code may take a minute or two to finish running

sns.pairplot(diabetes)

plt.show()

Pair plots offer scatter plots for all variable pairs within a data set, typically arranged in a matrix format. The main diagonal of this matrix features histograms, as plotting a variable against itself doesn't provide meaningful information.

Scatter plots located off the main diagonal visually depict the relationship between respective variables, offering insights into the data structure. For instance, examining the scatter plots for the (s1, s2) pair reveals a positive linear relationship, aligning with your expectations derived from correlation matrix analysis.

As an additional example, the previous plots give you a very quick and easy indication that sex1 is a binary variable. Of course, in this particular instance, you knew this beforehand. However, if you were looking at this data set for the very first time and the variable had a non-descriptive name, such as ab\_1112, then you would not have known this fact without performing some EDA.

### A simple function to perform EDA

The fasteda package includes a function called fast\_eda that performs all of the previous analysis using one easy-to-remember command.

# This code may take a minute or two to finish running

fast\_eda(diabetes)

The order and formatting of the items produced using fast\_eda might differ from what you derived manually, but the information contained in all plots and matrices produced by fast\_eda is identical.

# Classification

Conducting EDA for a classification task closely resembles that of a regression task. The key disparity lies in the presence of predefined classes within classification data sets. These classes can serve as visual cues in various EDA plots, offering insights into data distribution and relationships. Let's import a classification data set to emphasize this aspect.

## Import the iris data set

The [iris data set](https://en.wikipedia.org/wiki/Iris_flower_data_set) is a data set of iris flowers. The target variable in this data set is categorical and indicates the iris flower type. The objective is to use the numerical features to predict the iris flowers' species. Run the following code block to import this data set.

# Load the data set from sklearn

iris\_sklearn = load\_iris()

# Extract the data and target labels as a numpy array

iris\_npy = np.concatenate([iris\_sklearn['data'], np.atleast\_2d(iris\_sklearn['target']).T], axis=1)

# Define column names

col\_names = ['sepal\_length', 'sepal\_width', 'petal\_length', 'petal\_width', 'target']

# Convert the numpy array to a pandas dataframe with column names

iris = pd.DataFrame(iris\_npy, columns=col\_names)

# Print a description of the dataset

print(iris\_sklearn['DESCR'])

Take a look at the target column.

iris['target'].sample(5)

The previous example indicates that the target column, which is categorical, is already numerically encoded. Let's map this encoding to class names.

class\_names = dict(zip(list(map(float, range(len(iris\_sklearn['target\_names'])))), iris\_sklearn['target\_names']))

print(class\_names)

The target column has the following mapping: target label 0.0 is associated with the setosa flower, 1.0 is associated with the versicolor flower, and 2.0 is associated with the virginica flower.

## Perform EDA on the iris data set

Complete the following exercise to perform EDA on the iris data set.

### Exercise 7

Use the fasteda library to perform EDA on the iris data set. Note that the fast\_eda function takes a parameter called target to which you should pass your target variable when analyzing a classification data set.

### TYPE YOUR SOLUTION HERE:

Click here for solution

As seen in the previous plots, setting the target parameter in fast\_eda changes the way pair plots and histograms are drawn. For pair plots, the scatter plots off the main diagonal are color-coded by the target class, which helps in identifying pairs of features that could separate the classes well. For instance, consider the scatter plot that has petal width on the horizontal axis and sepal width on the vertical axis.

plt.axis('equal')

sns.scatterplot(iris, x='petal\_width', y='sepal\_width', hue='target', palette=sns.color\_palette("hls", iris['target'].nunique()))

plt.show()

It's evident that petal width alone can effectively distinguish the setosa class from the others. Conversely, sepal width alone isn't adequate to isolate the setosa class. However, when considering the versicolor and virginica classes, you see that higher values of both sepal width and petal width tend to correspond to the virginica class (encoded as 2.0), whereas lower values of both features are associated with the versicolor class (encoded as 1.0). Consequently, you recognize that the combination of sepal and petal widths can better differentiate these two classes compared to either feature in isolation. Specifically, higher sepal width is more likely to indicate the virginica class when accompanied by higher petal width.

Note that the histograms produced by fast\_eda are also presented by class, and this can help you identify differences in the distributions of feature values for the different classes. For instance, consider the histogram for sepal length.

for idx, col in enumerate(['sepal\_length']):

fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (14, 6))

sns.histplot(iris, x = iris[col], kde = True,

color = sns.color\_palette("hls", iris['target'].nunique()), ax = ax1, hue = 'target')

sns.boxplot(x = iris[col], width = 0.4, linewidth = 3, fliersize = 2.5,

color = sns.color\_palette("hls", iris['target'].nunique())[idx], ax = ax2)

fig.suptitle(f"Histogram and Boxplot of {col}", size = 20, y = 1.02)

plt.show()

This histogram indicates that the versicolor and virginica classes appear to be far more dispersed than the setosa class. This, in turn, indicates that it is very unlikely for you to see flowers in the setosa class having a sepal length greater than 6.5, but it is possible (though still unlikely) for the versicolor and virginica classes to have sepal lengths as low as 4.5. Therefore you can conclude that, for example, an observation with a sepal length of 5 could belong to any of the classes with a reasonably high degree of probability, but an observation with a sepal length of 7.5 will almost surely not belong to the setosa class. These types of observations would not be easily made without histograms that visually split the data by class.

The final major item to consider when dealing with a classification data set is the class balance, or the proportion of observations that belong to each class. Many prediction algorithms perform poorly in the presence of unbalanced data. In other words, when there are relatively few examples for some classes. Class balance can be seen using the countplot generated by fast\_eda, but sometimes it is better to visualize that data using a pie chart instead. Complete the following exercise to generate a pie chart that indicates the class balance of the iris data set.

### Exercise 8

Replace the part that says ### REPLACE WITH YOUR CODE ### to generate a pie chart showing the relative proportion of each class. Note that value counts can be obtained using the pandas.Series .value\_counts() method.

# Get value counts

vc = ### REPLACE WITH YOUR CODE ###

# Draw a pie chart using value counts and the `autopct\_format` format

\_ = plt.pie(vc)

Click here for solution

An even prettier pie chart, one that shows the exact percentages of each class as well as the number of observations belonging to each class, can be made using:

# Inspired by:

#https://stackoverflow.com/questions/73616867/pie-chart-from-count-of-values-showing-the-percentage/73617021#73617021

# Define a function to format value counts into percentages

def autopct\_format(values):

def my\_format(pct):

total = sum(values)

val = int(round(pct\*total/100.0))

return '{:.1f}%\n({v:d})'.format(pct, v=val)

return my\_format

# Get value counts

vc = iris['target'].value\_counts()

# Draw a pie chart using value counts and the `autopct\_format` format

\_ = plt.pie(vc, labels = vc.rename(class\_names).index, autopct=autopct\_format(vc))

There are 50 observations for each of the 3 classes, indicating that this data set is balanced.