CC5067NI-Smart Data Discovery

60% Individual Coursework

2023-24 Spring

Student Name: Shirshak Aryal

London Met ID: 22085620

College ID: NP01CP4S230122

Assignment Due Date: Monday, May 13, 2024

Assignment Submission Date: Monday, May 13, 2024

Word Count: 3813

I confirm that I understand my coursework needs to be submitted online via MySecondTeacher under the relevant module page before the deadline in order for my assignment to be accepted and marked. I am fully aware that late submissions will be treated as non-submission and a marks of zero will be awarded.

Table of Contents

Table of Figures

Table of Tables

1. Data Understanding

The given dataset contains **data** relating to the **salaries** of various **jobs** in the **Data Science** field and the **factors** that affect them. It has **11 variables/columns** as shown in the table below, including work year, experience level of jobs, type of employments, job titles, salaries in different currencies and their USD equivalents, country of residence of employees, the ratio of remote workers to in-office workers, and the location and size of the company. It contains **both numeric** (such as work year, salary, salary in USD) **and string** values (such as job title, experience level, employment type).

The data collected contains **information** from **various time periods** and **locations** about **various jobs**. For example, **work years** range from **2020 to 2023**, and the **country** of **location** of **companies** as well as **employee residences** vary widely, including Canada (CA), USA (US), India (IN), etc. Initial inspection reveals that **salaries** for the **same job** appear to have **some dependency** on each of **these factors** along with the **other variables** in the dataset.

Concise descriptions of each variable in the dataset are displayed in the **table** below, along with their **data types**:

S. No.	Column	Description	Data Type
1.	work_year	Denotes the year (in AD)	Integer
2.	experience_level	Denotes the experience level of the job, i.e., senior/expert (SE), medium/intermediate (MI), entry level (EN), and executive level (EX)	String
3.	employment_type	Denotes the type of job, i.e. Full Time (FT), Contract (CT), Freelance (FL), and Part Time (PT).	String
4.	job_title	Denotes the title/name of the job	String
5.	salary	Denotes the salary of the job	Integer
6.	salary_currency	Denotes the unit of currency of the job salary	String
7.	salary_in_usd	Denotes the salary of the job after conversion into USD	Integer
8.	employee_residence	Denotes the country of residence of employee	String
9.	remote_ratio	Denotes the ratio of the number of employees working remotely to the that of employees working in-site.	Integer
10.	company_location	Denotes the country of location of the company	String
11.	company_size	Denotes the size of the company, i.e., large (L), medium (M), and small (S)	String

Table 1: Column descriptions and data types

2. Data Preparation

Before beginning the data preparation steps, the required python libraries were imported into the notebook as shown here:

```
# IMPORTING LIBRARIES
import pandas as pd # importing pandas
import numpy as np # importing numpy
import matplotlib.pyplot as plt # importing pyplot from matplotlib
import math # importing math library
```

Figure 1: Importing the required Python libraries

2.1. Write a python program to load data into pandas DataFrame.

```
# reading the csv file and converting it into a DataFrame
saldf = pd.read_csv("DataScienceSalaries.csv")
saldf.head(10)
```

	work_year	experience_level	employment_type	job_title	salary	salary_currency	salary_in_usd	employee_residence	remote_ratio	company_location	company_size
0	2023	SE	FT	Principal Data Scientist	80000	EUR	85847	ES	100	ES	L
1	2023	MI	CT	ML Engineer	30000	USD	30000	US	100	US	S
2	2023	MI	CT	ML Engineer	25500	USD	25500	US	100	US	S
3	2023	SE	FT	Data Scientist	175000	USD	175000	CA	100	CA	M
4	2023	SE	FT	Data Scientist	120000	USD	120000	CA	100	CA	M
5	2023	SE	FT	Applied Scientist	222200	USD	222200	US	0	US	L
6	2023	SE	FT	Applied Scientist	136000	USD	136000	US	0	US	L
7	2023	SE	FT	Data Scientist	219000	USD	219000	CA	0	CA	M
8	2023	SE	FT	Data Scientist	141000	USD	141000	CA	0	CA	M
9	2023	SE	FT	Data Scientist	147100	USD	147100	US	0	US	M

Figure 2: Loading the data into a pandas DataFrame

The code above **reads** the **csv file** containing the required data and **converts** it into a **pandas DataFrame**, of which the **head(10) displays** the **first 10 rows**.

2.2. Write a python program to remove unnecessary columns, i.e., salary and salary_currency.

```
### DATA PREPARATION ###

# removing the 'salary' and 'salary_currency' columns
saldf.drop(columns = ["salary","salary_currency"], inplace=True)
saldf.head(10)
```

	work_year	experience_level	employment_type	job_title	salary_in_usd	employee_residence	remote_ratio	company_location	company_size
0	2023	SE	FT	Principal Data Scientist	85847	ES	100	ES	L
1	2023	MI	CT	ML Engineer	30000	US	100	US	S
2	2023	MI	CT	ML Engineer	25500	US	100	US	S
3	2023	SE	FT	Data Scientist	175000	CA	100	CA	M
4	2023	SE	FT	Data Scientist	120000	CA	100	CA	M
5	2023	SE	FT	Applied Scientist	222200	US	0	US	L
6	2023	SE	FT	Applied Scientist	136000	US	0	US	L
7	2023	SE	FT	Data Scientist	219000	CA	0	CA	M
8	2023	SE	FT	Data Scientist	141000	CA	0	CA	M
9	2023	SE	FT	Data Scientist	147100	US	0	US	M

Figure 3: Removing the 'salary' and 'salary_currency' columns

The code here drops the columns **salary** and **salary_currency** from the DataFrame; the **inplace=True drops** the columns from the **original DataFrame** itself.

2.3. Write a python program to remove the NaN missing values from updated DataFrame.

```
# checking for null values in the dataframe
saldf.isnull().sum()
```

```
work_year             0
experience_level      0
employment_type       0
job_title             0
salary_in_usd         0
employee_residence    0
remote_ratio          0
company_location      0
company_size          0
dtype: int64
```

Figure 4: Checking if there are any NaN values in the updated DataFrame

The code shown above **checks** if any **NaN (Not a Number)** values exist in the DataFrame. The **sum()** shows that **all columns** have a value of **0**. Since the **isnull()** shows **True** if any **NaN values exist**, this means that the DataFrame has **no NaN values**.

```
# removing NaN values
saldf.dropna(inplace=True)
saldf.isnull().sum()
```

```
work_year             0
experience_level      0
employment_type       0
job_title             0
salary_in_usd         0
employee_residence    0
remote_ratio          0
company_location      0
company_size          0
dtype: int64
```

Figure 5: Removing the 'NaN' values from the DataFrame

If NaN values **did exist**, the **dropna()** could be used to drop the rows that contained those values.

2.4. Write a python program to check for duplicate values in the DataFrame.

```python
# checking for duplicate rows in the DataFrame
duplicated_rows = saldf[saldf.duplicated()]
duplicated_rows.head(10)
```

	work_year	experience_level	employment_type	job_title	salary_in_usd	employee_residence	remote_ratio	company_location	company_size
115	2023	Senior Level/Expert	FT	Data Scientist	150000	US	0	US	M
123	2023	Senior Level/Expert	FT	Analytics Engineer	289800	US	0	US	M
153	2023	Medium Level/Intermediate	FT	Data Engineer	100000	US	100	US	M
154	2023	Medium Level/Intermediate	FT	Data Engineer	70000	US	100	US	M
160	2023	Senior Level/Expert	FT	Data Engineer	115000	US	0	US	M
163	2023	Senior Level/Expert	FT	Applied Scientist	309400	US	0	US	L
164	2023	Senior Level/Expert	FT	Applied Scientist	159100	US	0	US	L
167	2023	Medium Level/Intermediate	FT	Data Engineer	162500	US	0	US	M
168	2023	Medium Level/Intermediate	FT	Data Engineer	130000	US	0	US	M
171	2023	Senior Level/Expert	FT	Data Engineer	145000	US	0	US	M

Figure 6: Checking for duplicate values in the DataFrame

The code depicted in this picture **displays all the rows** that are **duplicated** in the DataFrame, using the **duplicated()** function.

2.5. Write a python program to see the unique values from all the columns in the DataFrame.

```python
# checking for unique values in all columns of the DataFrame

for col in saldf:
    if (saldf[col].dtype == 'object'):
        print(str(col), '\n', saldf[col].unique(), '\n')
```

```
experience_level
 ['SE' 'MI' 'EN' 'EX']

employment_type
 ['FT' 'CT' 'FL' 'PT']

job_title
 ['Principal Data Scientist' 'ML Engineer' 'Data Scientist'
 'Applied Scientist' 'Data Analyst' 'Data Modeler' 'Research Engineer'
 'Analytics Engineer' 'Business Intelligence Engineer'
 'Machine Learning Engineer' 'Data Strategist' 'Data Engineer'
 'Computer Vision Engineer' 'Data Quality Analyst'
 'Compliance Data Analyst' 'Data Architect'
 'Applied Machine Learning Engineer' 'AI Developer' 'Research Scientist'
 'Data Analytics Manager' 'Business Data Analyst' 'Applied Data Scientist'
 'Staff Data Analyst' 'ETL Engineer' 'Data DevOps Engineer' 'Head of Data'
 'Data Science Manager' 'Data Manager' 'Machine Learning Researcher'
 'Big Data Engineer' 'Data Specialist' 'Lead Data Analyst'
 'BI Data Engineer' 'Director of Data Science'
 'Machine Learning Scientist' 'MLOps Engineer' 'AI Scientist'
 'Autonomous Vehicle Technician' 'Applied Machine Learning Scientist'
 'Lead Data Scientist' 'Cloud Database Engineer' 'Financial Data Analyst'
 'Data Infrastructure Engineer' 'Software Data Engineer' 'AI Programmer'
 'Data Operations Engineer' 'BI Developer' 'Data Science Lead'
 'Deep Learning Researcher' 'BI Analyst' 'Data Science Consultant'
 'Data Analytics Specialist' 'Machine Learning Infrastructure Engineer'
 'BI Data Analyst' 'Head of Data Science' 'Insight Analyst'
```

Figure 7: Seeing unique values from all columns of the DataFrame – 1

```
 'Deep Learning Engineer' 'Machine Learning Software Engineer'
 'Big Data Architect' 'Product Data Analyst'
 'Computer Vision Software Engineer' 'Azure Data Engineer'
 'Marketing Data Engineer' 'Data Analytics Lead' 'Data Lead'
 'Data Science Engineer' 'Machine Learning Research Engineer'
 'NLP Engineer' 'Manager Data Management' 'Machine Learning Developer'
 '3D Computer Vision Researcher' 'Principal Machine Learning Engineer'
 'Data Analytics Engineer' 'Data Analytics Consultant'
 'Data Management Specialist' 'Data Science Tech Lead'
 'Data Scientist Lead' 'Cloud Data Engineer' 'Data Operations Analyst'
 'Marketing Data Analyst' 'Power BI Developer' 'Product Data Scientist'
 'Principal Data Architect' 'Machine Learning Manager'
 'Lead Machine Learning Engineer' 'ETL Developer' 'Cloud Data Architect'
 'Lead Data Engineer' 'Head of Machine Learning' 'Principal Data Analyst'
 'Principal Data Engineer' 'Staff Data Scientist' 'Finance Data Analyst']

employee_residence
 ['ES' 'US' 'CA' 'DE' 'GB' 'NG' 'IN' 'HK' 'PT' 'NL' 'CH' 'CF' 'FR' 'AU'
 'FI' 'UA' 'IE' 'IL' 'GH' 'AT' 'CO' 'SG' 'SE' 'SI' 'MX' 'UZ' 'BR' 'TH'
 'HR' 'PL' 'KW' 'VN' 'CY' 'AR' 'AM' 'BA' 'KE' 'GR' 'MK' 'LV' 'RO' 'PK'
 'IT' 'MA' 'LT' 'BE' 'AS' 'IR' 'HU' 'SK' 'CN' 'CZ' 'CR' 'TR' 'CL' 'PR'
 'DK' 'BO' 'PH' 'DO' 'EG' 'ID' 'AE' 'MY' 'JP' 'EE' 'HN' 'TN' 'RU' 'DZ'
 'IQ' 'BG' 'JE' 'RS' 'NZ' 'MD' 'LU' 'MT']

company_location
 ['ES' 'US' 'CA' 'DE' 'GB' 'NG' 'IN' 'HK' 'NL' 'CH' 'CF' 'FR' 'FI' 'UA'
 'IE' 'IL' 'GH' 'CO' 'SG' 'AU' 'SE' 'SI' 'MX' 'BR' 'PT' 'RU' 'TH' 'HR'
 'VN' 'EE' 'AM' 'BA' 'KE' 'GR' 'MK' 'LV' 'RO' 'PK' 'IT' 'MA' 'PL' 'AL'
 'AR' 'LT' 'AS' 'CR' 'IR' 'BS' 'HU' 'AT' 'SK' 'CZ' 'TR' 'PR' 'DK' 'BO'
 'PH' 'BE' 'ID' 'EG' 'AE' 'LU' 'MY' 'HN' 'JP' 'DZ' 'IQ' 'CN' 'NZ' 'CL'
 'MD' 'MT']

company_size
 ['L' 'S' 'M']
```

Figure 8: Seeing unique values from all columns of the DataFrame – 2

The pictures above display **all the unique values** from **each (non-numeric) column** in the DataFrame. This is achieved using a **for loop** to **iteratively pass each**

column of the DataFrame to the **if block** below, which **only** allows the **showing** of **unique values** of those **columns** with an **'object' datatype**.

2.6. Rename the experience level columns below.

- **SE – Senior Level/Expert**
- **MI – Medium Level/Intermediate**
- **EN – Entry Level**
- **EX – Executive Level**

```
# renaming the three experience level values
saldf["experience_level"] = saldf["experience_level"].replace(["SE","MI","EN","EX"], ["Senior Level/Expert","Medium Level/Intermediate","Entry Level","Executive Level"]
saldf.head(10)
```

	work_year	experience_level	employment_type	job_title	salary_in_usd	employee_residence	remote_ratio	company_location	company_size
0	2023	Senior Level/Expert	FT	Principal Data Scientist	85847	ES	100	ES	L
1	2023	Medium Level/Intermediate	CT	ML Engineer	30000	US	100	US	S
2	2023	Medium Level/Intermediate	CT	ML Engineer	25500	US	100	US	S
3	2023	Senior Level/Expert	FT	Data Scientist	175000	CA	100	CA	M
4	2023	Senior Level/Expert	FT	Data Scientist	120000	CA	100	CA	M
5	2023	Senior Level/Expert	FT	Applied Scientist	222200	US	0	US	L
6	2023	Senior Level/Expert	FT	Applied Scientist	136000	US	0	US	L
7	2023	Senior Level/Expert	FT	Data Scientist	219000	CA	0	CA	M
8	2023	Senior Level/Expert	FT	Data Scientist	141000	CA	0	CA	M
9	2023	Senior Level/Expert	FT	Data Scientist	147100	US	0	US	M

Figure 9: Renaming the values of the 'experience_level' column

The code shown here **replaces** the **values** of the **experience_level column** in the DataFrame as per the requirement stated above, using the **replace()** function.

3. Data Analysis

3.1. Utility Functions

3.1.1. To verify that the input column exists in the DataFrame and is numeric

```python
### DATA ANALYSIS ###

### UTILITY FUNCTIONS ###

# defining a function to verify that the input column exists in the DataFrame and is numeric

def verify_input_column(input_stmt):
    """
    Purpose:
        Checks if the user-input column exists and is of int64 datatype

    Parameters:
        input string

    Returns:
        column
    """

    numeric_cols = [col for col in saldf.columns if saldf[col].dtype=='int64'] # list of all numeric columns of the dataframe
    input_stmt += " from among these ones:\n" + ", ".join(numeric_cols) + "\n\n" # displaying the numeric columns with the input statement
    error_msg = "\nInvalid input\n" # error message

    while True:
        col = input(input_stmt) # taking input from the user

        if(col in numeric_cols): break # breaking the loop if the entered column exists and is numeric
        else: print(error_msg) # printing error message if column does not exist

    return col # returning the verified column
```

Figure 10: Defining a function to verify that the input column exists in the DataFrame and is numeric

The function above is **defined** to **verify** that the **column entered** by the user **exists** in the **DataFrame** and is **numeric**. The first line of the function **stores** the **list** of **all numeric columns** in the **DataFrame** in a variable **using list comprehension**. The next line then **defines** the **string** to be **added to the input statement** set in **other functions**, which **shows** the **list** of '**valid' numeric columns** to the user. The third line **sets the error message** to be shown **if** a user **enters** an **invalid column**.

Next, the **while loop** consists of a line that **asks** the **user** to **enter** a **column**, for which the **input statement** is **passed** as the **argument** for **this function**. **If** the **column exists** in the **list of numeric columns**, **only then** the function **returns** that **column**; if the **column fails** to **satisfy** this **condition**, the **error message** is **displayed** and the **loop reruns** and **asks** the **user** for a **column again**.

```
verify_input_column("Enter a column")

Enter a column from among these ones:
work_year, salary_in_usd, remote_ratio

salary

Invalid input

Enter a column from among these ones:
work_year, salary_in_usd, remote_ratio

job_title

Invalid input

Enter a column from among these ones:
work_year, salary_in_usd, remote_ratio

salary_in_usd
'salary_in_usd'
```

Figure 11: Checking to see if the entered columns exist in the DataFrame and are numeric

The above picture demonstrates the working of the above function, with both valid and invalid columns.

3.1.2. To calculate the median of all values in a column

Median is the **central value** of a **distribution** whose **values** have been **sorted** in **ascending order.** If the distribution has an **odd number of values**, the **median** is the **value** at the **exact center**. If **even**, the **median** is the **average** of the **two middle numbers**. Medians are **more precise than means** since they are **less affected** by **outliers**. (The Economic Times, 2024)

```python
# defining a function to calculate the median of all values in a column

def calc_arg_median(col):
    """
    Purpose:
        Calculates median value of the argument numeric column of a DataFrame

    Parameters:
        a numeric column

    Returns:
        median
    """

    sorted_col = saldf[col].sort_values() # sorting the values of col in ascending order
    median_index = 0 # index of the median value in col

    # condition to check if col has an even or odd number of values
    if len(sorted_col)%2 == 0: # if col has an even number of values
        median_index = len(sorted_col)//2 # index of the median value is
                                          # half of the number of values
    else: # if col has an odd number of values
        median_index = (len(sorted_col) + 1)//2 # index of the median value is
                                                # half of one more than the number of values

    # median value of col
    median = sorted_col.iloc[median_index - 1] # subtracting 1 since indices start from 0
    return median # returning the median value
```

Figure 12: Defining a function to calculate the median of all values of the argument column

This function calculates the **median** of **all values** of a column. It **starts** by **sorting** the column in **ascending** order. It **initially sets** the **index** of the would-be **median** value to **0**. It then **checks** if the column has an **odd** or an **even number of values**. If **even**, it sets the **index** of the median to **half** the **number of values**. If **odd**, it **first adds 1** to the **number of values** and then sets the **index** of the **median** to **half** of **that number**. It then **sets** the **median** value as the **value** occurring **just before** the one in the **index defined above** in the column and returns it.

```python
def calc_median(): # ENTRY FUNCTION
    """
    Purpose:
        Calculates median value of a user-entered numeric column of a DataFrame

    Parameters:
        none

    Returns:
        median
    """

    # taking input column and verifying that it exists and is numeric
    col = verify_input_column("Enter a column")
    median = calc_arg_median(col) # calculating median value
    return median # returns the median value
```

Figure 13: Defining a function to calculate the median of all values of the input column

This function asks the user for a numeric column, verifies it using the utility function, and passes it to the function above to calculate the median, then returns it.

```python
col = 'salary_in_usd'

print("Median:", calc_median()) # input-taking function
print("Median:", calc_arg_median(col)) # argument-taking function
print("Median:", saldf[col].median()) # built-in function
```

```
Enter a column from among these ones:
work_year, salary_in_usd, remote_ratio

salary_in_usd
Median: 135000
Median: 135000
Median: 135000.0
```

Figure 14: Calculating the median of a column using the input-taking, argument-taking, and built-in functions

The picture above shows the workings of both the functions above, comparing their outputs to that of the in-built median function. It shows that the **median** value of the **salary_in_usd** column is **$135,000**.

3.2. Write a Python program to show summary statistics of sum, mean, standard deviation, skewness, and kurtosis of any chosen variable.

3.2.1. Sum

```python
# defining functions to calculate the sum of all values in a column

def calc_arg_sum(col):
    """
    Purpose:
        Calculates sum of all values of the argument numeric column of a DataFrame

    Parameters:
        a numeric column

    Returns:
        sum
    """

    sum = 0 # summation of all values of col
    for values in saldf[col]:
        sum += values # sum of all values of col
    return sum # returning the sum
```

Figure 15: Defining a function to calculate the sum of all values of the argument column

The function here **calculates the sum** of **all values** in a column. It **first sets** the **initial value** of the **sum** to **0**, then **iteratively adds** each value of the column and returns the result.

```python
def calc_sum():  # ENTRY FUNCTION
    """
    Purpose:
        Calculates sum of all values of a user-entered numeric column of a DataFrame

    Parameters:
        none

    Returns:
        sum
    """

    # taking input column and verifying that it exists and is numeric
    col = verify_input_column("Enter a column: ")
    sum = calc_arg_sum(col) # calculating the sum
    return sum # returning the sum
```

Figure 16: Defining a function to calculate the sum of all values of the input column

This function asks the user for the numeric column, verifies it, and passes it to the sum function above to calculate the sum and returns it.

```
col = 'salary_in_usd'

print("Sum:", calc_sum()) # input-taking function
print("Sum:", calc_arg_sum(col)) # argument-taking function
print("Sum:", saldf[col].sum()) # built-in function
```

```
Enter a column:  from among these ones:
work_year, salary_in_usd, remote_ratio

salary_in_usd
Sum: 516576814
Sum: 516576814
Sum: 516576814
```

Figure 17: Calculating the sum of a column using the input-taking, argument-taking, and built-in functions

This screenshot displays the workings of the functions above and compares their outputs to that of the in-built sum function. As seen here, the **sum** of **all salaries** in the **salary_in_usd** column is **$516,576,814.**

3.2.2. Mean

```python
# defining functions to calculate the mean of all values in a column

def calc_arg_mean(col):
    """
    Purpose:
        Calculates mean value of the argument numeric column of a DataFrame

    Parameters:
        a numeric column

    Returns:
        mean
    """

    col_sum = calc_arg_sum(col) # sum of all values of col
    mean = col_sum/len(saldf[col]) # mean value of col
    return mean # returning the mean value
```

Figure 18: Defining a function to calculate the mean of all values of the argument column

The picture above shows the defining of a function that calculates the **mean of all values of a column**. First, it uses the **sum function** above to **calculate the sum**, then **divides** it by the **length of the column** (which is the **same as** the **number of values** in it) and returns the mean.

```python
def calc_mean():  # ENTRY FUNCTION
    """
    Purpose:
        Calculates mean value of a user-entered numeric column of a DataFrame

    Parameters:
        none

    Returns:
        mean
    """

    # taking input column and verifying that it exists and is numeric
    col = verify_input_column("Enter a column")
    mean = calc_arg_mean(col) # calculating mean value
    return mean # returning the mean value
```

Figure 19: Defining a function to calculate the mean of all values of the input column

This function passes the verified user-input column to the function above to calculate the mean and returns it.

```
col = 'salary_in_usd'

print("Mean:", calc_mean()) # input-taking function
print("Mean:", calc_arg_mean(col)) # argument-taking function
print("Mean:", saldf[col].mean()) # built-in function
```

```
Enter a column from among these ones:
work_year, salary_in_usd, remote_ratio

salary_in_usd
Mean: 137570.38988015978
Mean: 137570.38988015978
Mean: 137570.38988015978
```

Figure 20: Calculating the mean of a column using the input-taking, argument-taking, and built-in functions

The workings of the above functions are shown clearly here, with the comparison of their outputs to that of the built-in mean function. The mean of the **salary_in_usd** column is **137570.389**, meaning that on average, **salaries** tend to be **close** to **$137,570**.

3.2.3. Standard Deviation

Standard deviation is a **measure** of **how much the values** of a distribution are **spread with respect to** the **mean** value. The **greater the spread**, the **higher** the value of **standard deviation** and **vice versa**. (The Economic Times, 2024)

```python
# defining a function to calculate the standard deviation of all values in a column

def calc_arg_stdev(col):
    """
    Purpose:
        Calculates standard deviation of the argument numeric column of a DataFrame

    Parameters:
        a numeric column

    Returns:
        standard deviation
    """

    mean = calc_arg_mean(col) # mean value of col
    sum_of_squared_diffs = 0 # summation of (x - X)^2,
                             # where x is each value of col,
                             # and X is the mean value of col

    # iteratively adding (x - X)^2
    for value in saldf[col]:
        sum_of_squared_diffs += (value - mean)**2

    var = sum_of_squared_diffs/(len(saldf[col])-1) # variance value of col
    stdev = (var)**0.5 # standard deviation of col
    return stdev # returning the stnadard deviation
```

Figure 21: Defining a function to calculate the standard deviation of all values of the argument column

This function calculates the **standard deviation of a column**. First, it calculates the **mean** of the column using the function above and **initializes** the **sum of the squared differences** of each **value** and the **mean** to **0.** The for loop then **iteratively adds** those values to that variable. Next, it **calculates the variance** by **dividing the sum of squared differences** by the **length** of the **column minus one** (since this is **sample data**, not population data).

```python
def calc_stdev(): # ENTRY FUNCTION
    """
    Purpose:
        Calculates standard deviation of a user-entered numeric column of a DataFrame

    Parameters:
        none

    Returns:
        standard deviation
    """

    # taking input column and verifying that it exists and is numeric
    col = verify_input_column("Enter a column")
    stdev = calc_arg_stdev(col) # calculating standard deviation value
    return stdev # returns the standard deviation value
```

Figure 22: Defining a function to calculate the standard deviation of all values of the input column

This function simply asks for and verifies the column input by the user to send it to the function above to calculate the standard deviation and returns it.

```python
col = 'salary_in_usd'

print("Standard Deviation:", calc_stdev()) # input-taking function
print("Standard Deviation:", calc_arg_stdev(col)) # argument-taking function
print("Standard Deviation:", saldf[col].std()) # built-in function
```

```
Enter a column from among these ones:
work_year, salary_in_usd, remote_ratio

salary_in_usd
Standard Deviation: 63055.625278224084
Standard Deviation: 63055.625278224084
Standard Deviation: 63055.6252782241
```

Figure 23: Calculating the standard deviation of a column using the input-taking, argument-taking, and built-in functions

This illustrates the workings of both functions above while comparing their outputs to that of the in-built standard deviation function. Since the **standard deviation** for the **salary_in_usd** column is **63055.62**, it indicates that the salaries have a **relatively wider spread** of values **around the mean salary**, meaning that **no single value or narrow range of salary is significantly frequent.**

3.2.4. Skewness

Skewness is the **measure of the asymmetry** of a **distribution**. It is mainly of 2 types:

- **Positive skewness:** A distribution is said to be **positively** skewed if the skewness is **greater than 0,** meaning that the **peak** is **towards** the **right** of the **graph** because the **mean** is **higher than** the **median**.
- **Negative skewness:** A distribution is said to be **negatively** skewed if the skewness is **lesser than 0**, meaning that the **peak** is **towards** the **left** of the graph because the **mean** is **lower than** the **median**. (StudySmarter, 2024)

```
# defining functions to calculate the skewness of all values in a column

def calc_arg_skew(col):
    """
    Purpose:
        Calculates skewness value of the argument numeric column of a DataFrame

    Parameters:
        a numeric column

    Returns:
        skewness
    """

    mean = calc_arg_mean(col) # mean value of col
    median = calc_arg_median(col) # median value of col
    std = calc_arg_stdev(col) # standard deviation of col
    sum_of_cubed_diffs = 0 # summation of (x - X)^3,
                           # where x is each value of col
                           # and X is the mean value of col

    # iteratively adding (x - X)^3 together
    for value in saldf[col]:
        sum_of_cubed_diffs += (value - mean)**3

    denom = (len(saldf[col]) - 1) * (std**3) # denominator of the skewness formula
    skewness = sum_of_cubed_diffs/denom # skweness value of col
    return skewness # returning the skewness value
```

Figure 24: Defining a function to calculate the skewness of all values of the argument column

This function calculates the **skewness** of the values in a **column** by first **calculating the mean, median, and standard deviation** values of the column. It then **initializes the sum of cubed differences** of the **values** and the **mean** to **0**, after which the for **loop iteratively adds** those values to it. It then **calculates the denominator** for the **skewness formula** by **multiplying** the **length** of the **column subtracted by 1** and the **cube** of the **standard deviation**. It finally **divides** the **sum of cubed differences** by that **denominator** and returns it as the **skewness**.

```python
def calc_skew(): # ENTRY FUNCTION
    """
    Purpose:
        Calculates skewness value of a user-entered numeric column of a DataFrame

    Parameters:
        none

    Returns:
        skewness
    """

    # taking input column and verifying that it exists and is numeric
    col = verify_input_column("Enter a column")
    stdev = calc_arg_skew(col) # calculating skewness value
    return stdev # returns the skewness value
```

Figure 25: Defining a function to calculate the skewness of all values of the input column

This function asks for and verifies the user-input column before passing it to the function above to calculate the skewness and return it.

```python
col = 'salary_in_usd'

print("Skewness:", calc_skew()) # input-taking function
print("Skewness:", calc_arg_skew(col)) # argument-taking function
print("Skewness:", saldf[col].skew()) # built-in function
```

```
Enter a column from among these ones:
work_year, salary_in_usd, remote_ratio

salary_in_usd
Skewness: 0.5361154662823615
Skewness: 0.5361154662823615
Skewness: 0.5364011659712974
```

Figure 26: Calculating the skewness of a column using the input-taking, argument-taking, and built-in functions

This picture demonstrates the workings of both the above functions, as well as the comparison of their outputs with that of the in-built skewness function. Since the **skewness** of the **salary_in_usd** column is **0.53**, it suggests that the distribution is **positively skewed**. I.e., **more salaries** are on the **higher end** of the **distribution**.

3.2.5. Kurtosis

Kurtosis is a **measure of** the **frequency** of occurrences of **outliers** in a dataset. It is mainly categorized into **3 types**:

- **Leptokurtic**: A distribution is said to be leptokurtic if it has frequent outliers, or if it has a kurtosis value of more than 3.
- **Mesokurtic**: A distribution is said to be mesokurtic if it has a moderate number of outliers, or if it has a kurtosis value of approximately 3.
- **Platykurtic**: A distribution is said to be platykurtic if it has very little outliers, or if it has a kurtosis value of less than 3. (Turney, 2024)

```
# defining functions to calculate the kurtosis of all values in a column

def calc_arg_kurt(col):
    """
    Purpose:
        Calculates kurtosis value of the argument numeric column of a DataFrame

    Parameters:
        a numeric column

    Returns:
        kurtosis
    """

    std = calc_arg_stdev(col) # standard deviation of col
    mean = calc_arg_mean(col) # mean value of col
    sum_of_4th_powers = 0 # summation of (x - X)^4,
                          # where x is each value of col1
                          # and X is the mean value of col1

    # iteratively adding (x - X)^4
    for value in saldf[col]:
        sum_of_4th_powers += (value - mean)**4

    fourth_moment = sum_of_4th_powers/len(saldf[col]) # fourth moment value of col
    kurt = fourth_moment/(std**4) # kurtosis value of col
    return kurt-3 # returning the kurtosis value
```

Figure 27: Defining a function to calculate the kurtosis of all values of the argument column

This function **calculates the kurtosis** of all values of a column. It starts by **calculating the standard deviation** and **mean** of the column and **initializes** the **sum** of **4th-power differences** of the **values** and the **mean** to **0**, to which the for loop iteratively adds those values. It then **calculates the fourth moment** by **dividing** the **sum of differences** by the **length** of the **column** and **calculates the kurtosis** by **dividing** the **fourth moment** by the **standard deviation** raised to the **4th power**. It then **subtracts 3** from that value (in terms of **excess kurtosis**) before returning it.

```python
def calc_kurt(): # ENTRY FUNCTION
    """
    Purpose:
        Calculates kurtosis value of a user-entered numeric column of a DataFrame

    Parameters:
        none

    Returns:
        kurtosis
    """

    # taking input column and verifying that it exists and is numeric
    col = verify_input_column("Enter a column")
    kurt = calc_arg_kurt(col) # calculating kurtosis value
    return kurt # returns the kurtosis value
```

Figure 28: Defining a function to calculate the kurtosis of all values of the input column

This function asks for and verifies the user-input column, then passes it to the function above to calculate the kurtosis and returns it.

```python
col = 'salary_in_usd'

print("Kurtosis:", calc_kurt()) # input-taking function
print("Kurtosis:", calc_arg_kurt(col)) # argument-taking function
print("Kurtosis:", saldf[col].kurt()) # built-in function
```

```
Enter a column from among these ones:
work_year, salary_in_usd, remote_ratio

salary_in_usd
Kurtosis: 0.8292585346115979
Kurtosis: 0.8292585346115979
Kurtosis: 0.8340064594833612
```

Figure 29: Calculating the kurtosis of a column using the input-taking, argument-taking, and built-in functions

This clearly illustrates the workings of the functions above and compares their outputs to that of the in-built kurtosis function. The kurtosis of the **salary_in_usd** column being **0.83** indicates that the **distribution** is **leptokurtic**, i.e. there are a **lot of extreme values** of **salaries** on **either side** of the **distribution**.

3.3. Write a Python program to calculate and show correlation of all variables.

3.3.1. Correlation

Correlation is the **measure** that shows the **rate of change** of a **variable** with respect to **another**. Its value ranges from **-1 to 1**, where **-1** indicates an **inverse** relationship (one variable **decreases** at the **same rate** that the other **increases**), while **1** indicates a **direct** relationship. The **closer** the correlation is to **0**, the more **likely** it is that the two variables are **weakly related**. (JMP Statistical Discovery, 2024)

```python
# defining functions to find the correlation between two numeric columns

def calc_arg_corr(col1, col2):
    """
    Purpose:
        Calculates correlation value of the two argument numeric columns of a DataFrame

    Parameters:
        two numeric columns

    Returns:
        correlation
    """

    col1_mean = calc_arg_mean(col1) # mean value of col1
    col2_mean = calc_arg_mean(col2) # mean value of col2

    col1_stdev = calc_arg_stdev(col1) # standard deviation of col1
    col2_stdev = calc_arg_stdev(col2) # standard deviation of col2

    sum_of_products = 0 # summation of (x - X) * (y - Y),
                        # where x, y = each value of col1, col2 respectively,
                        # and X, Y are the mean values of col1, col2 respectively

    # iteratively adding (x - X) * (y - Y)
    for i in range(len(saldf[col1])):
        sum_of_products += (saldf[col1].iloc[i] - col1_mean) * (saldf[col2].iloc[i] - col2_mean)

    covar = sum_of_products/(len(saldf[col1]) - 1) # covariance between col1 and col2
    corr = covar/(col1_stdev * col2_stdev) # correlation between col1 and col2
    return corr # returning the correlation value
```

Figure 30: Defining a function to calculate the correlation between the two argument columns

This function calculates the **correlation between two columns** of a DataFrame. First, it **calculates the means** and **standard deviations** of **both columns** and initializes the **sum of the products** of the **differences** of the **values** of each column with their respective **mean**, to be added to iteratively by the following for loop. It then calculates the **covariance of the two columns** by **dividing** the **sum** of those products by the **length** of the column minus one (sample mean). It calculates the **correlation** by then **dividing** the **covariance** by the **product** of the two **standard deviations** and returns it.

```python
def calc_corr(): # ENTRY FUNCTION
    """
    Purpose:
        Calculates correlation value of two user-entered numeric columns of a DataFrame

    Parameters:
        none

    Returns:
        correlation
    """

    # taking input column and verifying that it exists and is numeric
    col1 = verify_input_column("Enter the first column")
    col2 = verify_input_column("Enter the second column")
    corr = calc_arg_corr(col1, col2) # calculating correlation value
    return corr # returns the correlation value
```

Figure 31: Defining a function to calculate the correlation between the two input columns

This function asks the user for two numeric columns, verifies them, and passes them to the function above to calculate their correlation and return it.

```python
col1 = 'salary_in_usd'
col2 = 'remote_ratio'

print("Correlation:", calc_corr()) # input-taking function
print("Correlation:", calc_arg_corr(col1, col2)) # argument-taking function
print("Correlation:", saldf[col1].corr(saldf[col2])) # built-in function
```

```
Enter the first column from among these ones:
work_year, salary_in_usd, remote_ratio

salary_in_usd
Enter the second column from among these ones:
work_year, salary_in_usd, remote_ratio

remote_ratio
Correlation: -0.06417098519057539
Correlation: -0.06417098519057539
Correlation: -0.06417098519057558
```

Figure 32: Calculating the correlation between two columns using the input-taking, argument-taking, and built-in functions

This picture depicts the workings of each function above while comparing their outputs to that of the in-built correlation function. Here, since the correlation between the **salary_in_usd** and **remote_ratio** columns is **negative 0.06**, it suggests that the **higher the salaries, the lesser the remote ratio**. I.e., all employees that work **in-site** have **higher salaries** than those that work **remotely**.

4. Data Exploration

4.1. Write a python program to find out the top 15 jobs.

```
# finding out the top 15 jobs
top_jobs = saldf['job_title'].value_counts().head(15).sort_values()
top_jobs
```

```
job_title
Data Analytics Manager          22
Data Science Consultant         24
Machine Learning Scientist      26
Data Manager                    29
ML Engineer                     34
Research Engineer               37
Data Science Manager            58
Applied Scientist               58
Research Scientist              82
Data Architect                 101
Analytics Engineer             103
Machine Learning Engineer      289
Data Analyst                   612
Data Scientist                 840
Data Engineer                 1040
Name: count, dtype: int64
```

Figure 33: Finding out the top 15 jobs

This code finds out the **top 15 jobs** in the DataFrame by **first selecting** the **'job_title' column** and **sorting** the **frequencies** of each of the **values** in it in **descending** order using the **value_counts()** function. The **head(15)** then **selects** the **top 15 values** from the result, which are then **further sorted** in **ascending** order by **sort_values()** for clarity.

```
[46] top_jobs.plot(kind = "barh")
     plt.ylabel("Jobs")
     plt.xlabel("Frequency")
     plt.title("Top 15 Jobs")

Text(0.5, 1.0, 'Top 15 Jobs')
```

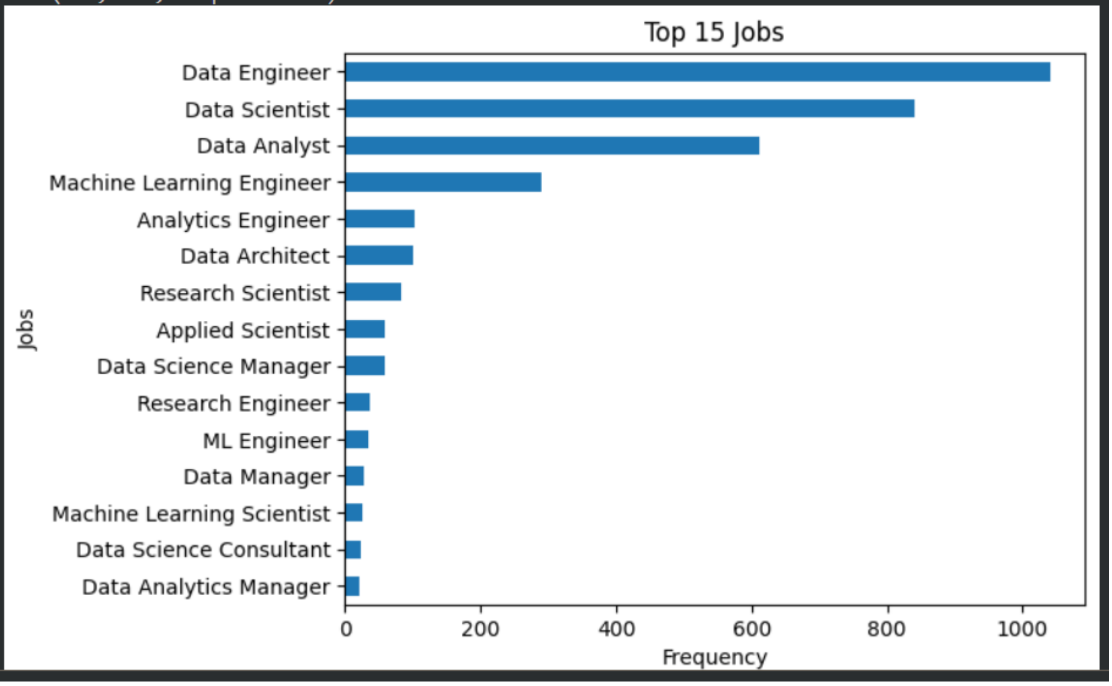

Figure 34: Plotting the bar graph of the top 15 jobs

The **top_jobs.plot(kind = 'barh')** plots a horizontal bar graph of the top 15 jobs calculated above. The **plt.ylabel('Jobs')** and **plt.xlabel('Frequency')** sets the **y-axis** and **x-axis labels** to **Jobs** and **Frequency respectively**, while the **plt.title('Top 15 Jobs')** sets the **title** of the bar graph to **Top 15 Jobs**.

This graph shows that the most common job is **Data Engineer (more than 1000)**, followed by jobs such as **Data Scientist, Data Analyst, Machine Learning Engineer**, etc.

4.2. Which job has the highest salaries? Illustrate with bar graph.

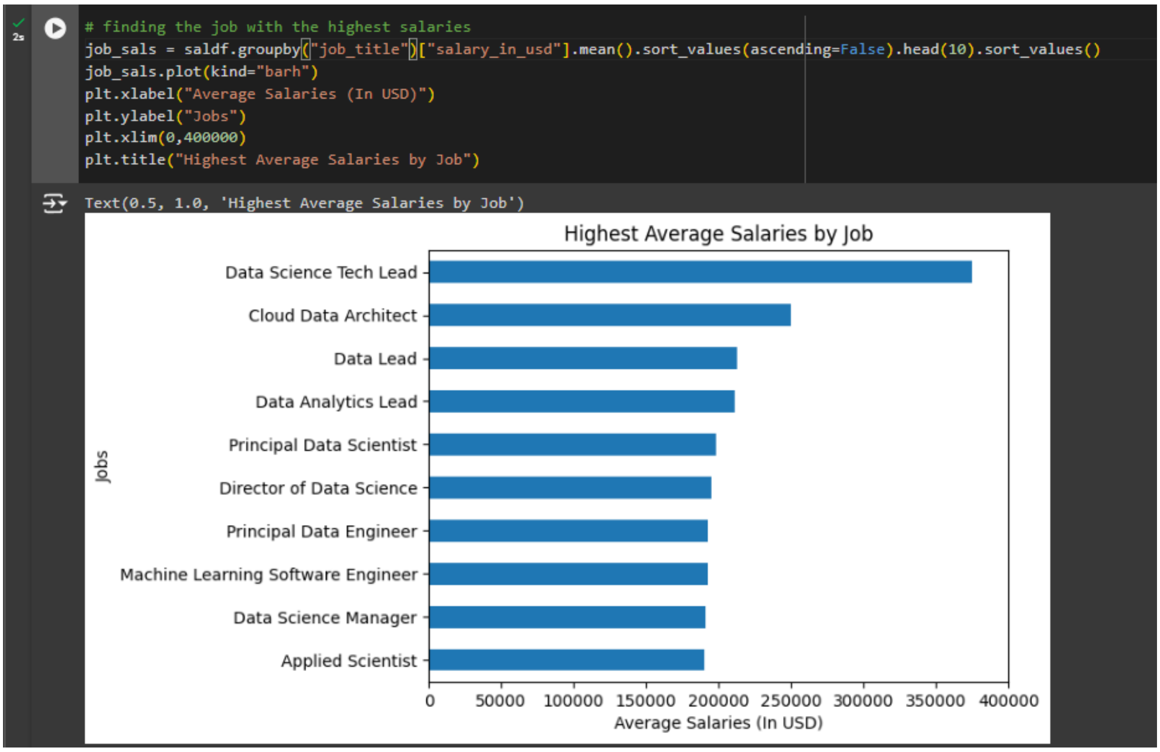

Figure 35: Plotting the bar graph of the jobs with the highest salaries

This code displays the jobs with the highest salaries. First, the **saldf.groupby('job_title') groups** the entire **DataFrame** by the column **job_title**, out of which the following **['salary_in_usd']** selects only that **specific column**. The **mean()** then calculates the **mean** of each value of the **salary_in_usd** column **per each value of job_title**. The **sort_values(ascending=False)** then **sorts** the values in **descending** order, out of which the **head(10)** selects only the **top 10** ones which are **further sorted** for clarity by **sort_values().**

The **job_sals.plot(kind = 'barh')** plots a horizontal bar graph. The **plt.xlabel('Average Salaries (in USD)')** and **plt.ylabel('Jobs')** set the **labels** for the **x and y-axes** respectively. The **plt.xlim(0, 400000)** sets the **lower** and **upper limits** of the **x-axis** to **0** and **400,000** respectively. The **plt.title('Highest Average Salaries by Job')** then sets the **title** of the **entire bar graph.**

The bar graph above shows that on average, **Data Science Tech Lead** jobs have the **highest salaries** by far among all other jobs **(above $350,000)**, followed by jobs such as **Cloud Data Architect, Data Lead, Data Analytics Lead**, etc.

4.3. Write a python program to find out salaries based on experience level. Illustrate it through a bar graph.

```
# finding out the salaries based on the experience levels
explvl_sals = saldf.groupby(saldf["experience_level"])["salary_in_usd"].mean().sort_values(ascending=False).head(10).sort_values()
plt.figure(figsize = (6,2))
explvl_sals.plot(kind="barh")
plt.ylabel("Experience Levels")
plt.xlabel("Salaries")
plt.title("Average Salaries by Experience Level")
```

```
Text(0.5, 1.0, 'Average Salaries by Experience Level')
```

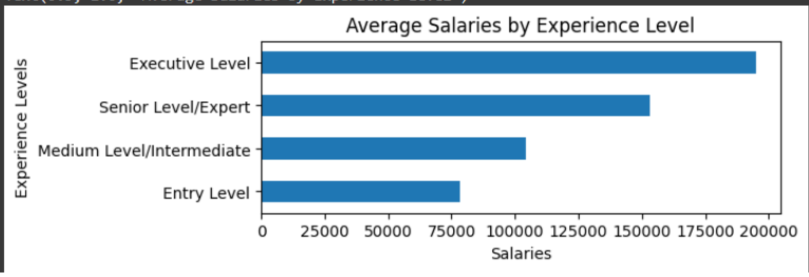

Figure 36: Plotting the bar graph of salaries based on experience level

This code displays a bar graph of salaries based on experience level. It starts by **grouping** the entire DataFrame by the **experience_level** column using the **groupby()** function. The **['salary_in_usd']** then selects that specific column from the result, of which the **mean** values are calculated **per experience level** by the **mean().** The values are then sorted in **descending** order by the **sort_values(ascending=False),** out of which the **head(10)** selects the **top 10 values only,** further sorted by **sort_values().**

The **plt.figure(figsize = (6, 2))** then sets the **length** and **height** of the plot to **6 and 2 inches** respectively. The **plot(kind = 'barh')** then plots a horizontal bar graph. The **plt.xlabel('Salaries'), plt.ylabel('Experience Levels'),** and **plt.title('Average Salaries by Experience Level')** then set the **x-axis label, y-axis label**, and the **title** of the plot respectively.

The bar graph shows that **on average**, jobs with **Executive** experience level have the **highest salaries**, while **entry level** jobs have the **lowest**.

4.4. Write a Python program to show histogram and box plot of any chosen different variables. Use proper labels in the graph.

4.4.1. Histogram of Salaries in USD

A histogram is a **graphical representation** of a **distribution** of **continuous** values of data (**unlike** a **bar graph** where the **values** are **discrete**), where the **height** of each ‘**bin’** (which are the **individual streaks/bars**) represents the **frequency** of that **value**. A **bin** represents a **class interval** (which is the **range of data** into which the values fall), and **all bins** of a histogram have **equal width**. (Jaspersoft, 2024)

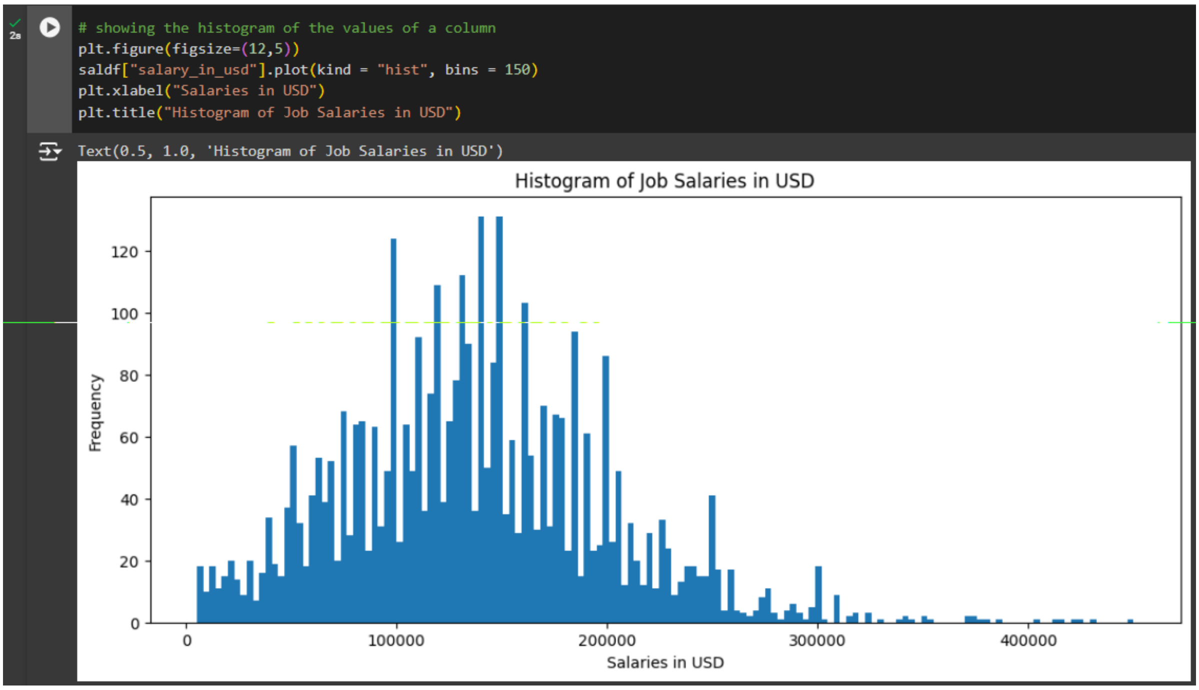

Figure 37: Plotting the histogram of job salaries in USD

This code plots the histogram of the values of the **salary_in_usd** column. It uses **plt.figure(figsize = (12, 5))** to set the size of the figure to **12 inches by 5 inches**. The **plot(kind = ‘hist’, bins = 150)** plots the histogram with **150 bins,** which are the individual streaks of blue seen here. The **plt.xlabel(‘Salaries in USD’)** and **plt.title(‘Histogram of Job Salaries in USD’)** set the **x-axis label** and **title** of the plot respectively.

The histogram shows that **most** of the salaries fall in the range of **$100,000** and **$200,000**, with the rest falling in other ranges, and very few salaries above **$300,000** and fewer yet beyond **$400,000**.

4.4.2. Boxplot of Work Years

Boxplots are **graphical representations** of a **distribution** of data in terms of **5 key numbers:** the **minimum** value, **first quartile**, **second quartile/median**, **third quartile**, and the **maximum** value, along with **outliers**. The **minimum** value of a distribution is represented by the **left/bottom whisker** of a boxplot, while the **maximum** value is represented by the **right/top whisker**. The **first quartile** is represented by the **left/bottom edge** of the **box**, and the **third quartile** by the **right/top edge**. The **median** is represented by a **line inside the box.** The **actual box** represents the **interquartile range (IQR)** of the distribution. (Galarnyk, 2023)

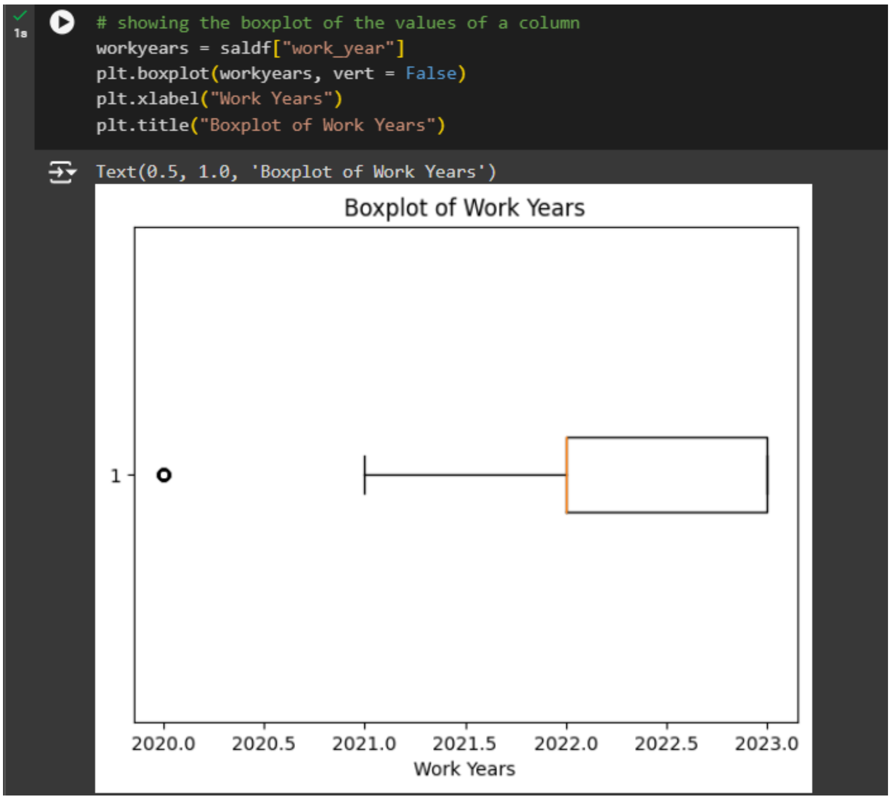

Figure 38: Plotting the boxplot of work years

This code generates a box plot of the **work_year** column. The **plt.boxplot(workyears, vert=False)** generates the actual plot, with **vert=False** making the plot horizontal. The **plt.xlabel('Work Years')** and **plt.title('Boxplot of Work Years')** then set the x-axis label and title of the plot respectively.

The **left whisker** in the figure shows that the **minimum value** of the **work_year** column is **2021**, with the **maximum value** being **2023** as shown by the **right whisker** (overlapped by the **third quartile**). The **median** value appears to be **2022**, while **2020** is seen as an **outlier value.** The **box** from **2022** to **2023** shows the **interquartile range** and that the **third quartile is 2023**, i.e., **75% of the values lie below 2023**. Since **2022** is the **median** as well as the **first quartile**, **25%** as well as **50%** of the data falls below it.

References

Galarnyk, M., 2023. *Understanding Boxplots.* [Online] Available at: https://builtin.com/data-science/boxplot [Accessed 12 May 2024].

Jaspersoft, 2024. *What is a Histogram Chart?.* [Online] Available at: https://www.jaspersoft.com/articles/what-is-a-histogram-chart [Accessed 12 May 2024].

JMP Statistical Discovery, 2024. *Correlation.* [Online] Available at: https://www.jmp.com/en_ca/statistics-knowledge-portal/what-is-correlation.html
[Accessed 12 May 2024].

StudySmarter, 2024. *Skewness.* [Online] Available at: https://www.studysmarter.co.uk/explanations/math/statistics/skewness/ [Accessed 12 May 2024].

The Economic Times, 2024. *What is 'Median'.* [Online] Available at: https://economictimes.indiatimes.com/definition/median [Accessed 12 May 2024].

The Economic Times, 2024. *What is 'Standard Deviation'.* [Online] Available at: https://economictimes.indiatimes.com/definition/standard-deviation [Accessed 12 May 2024].

Turney, S., 2024. *What Is Kurtosis? | Definition, Examples & Formula.* [Online] Available at: https://www.scribbr.com/statistics/kurtosis/ [Accessed 12 May 2024].