**Gemini - Determining Approach for visualizing data**

**My Prompt:**

*“Since X_base.dtypes shows some columns as object type and these columns also contain missing values, clustering cannot be performed yet. To ensure all columns are numeric, any non-conforming values must first be reinterpreted. Specifically, values that cannot be converted into numeric form should be replaced with NaN, so they can later be addressed using an appropriate imputation strategy*

*I am trying to determine what is the best way to determine which is the best inputation strategy that I need to implement in order to have a better data. I need a way to show how I got through it perhaps by visualization? there are about 20 columns, I am not sure if making a box plot for each to determine makes sense? how will you approach it? here is my current code for that part:*

*for col in X_base.columns:*

*X_base[col] = pd.to_numeric(df[col], errors="coerce") # Google Search AI Overview - check for unique non-numeric values in a df column*

*X_base = X_base.fillna(df.median()) # Google Search AI Overview - How to fill NaN values in a df with media”*

Of course. The best imputation strategy depends on the distribution of your data in each column. Applying a single method like the median to all columns is often not the optimal approach.

A systematic way to determine the best strategy is to **visualize the distribution of each column** and choose an appropriate method based on its shape and the presence of outliers.

---

## Correcting Your Current Approach

Your current code has a small but critical issue:

Python

```
# Incorrect: Fills NaNs in X_base with medians calculated from the original df
X_base = X_base.fillna(df.median())

# Correct: Fills NaNs with the median of each respective column in X_base
X_base = X_base.fillna(X_base.median())
```

You should calculate the median from the cleaned X_base DataFrame, not the original df, to ensure you're using the median of the newly numeric columns.

However, a better approach is to not apply one method universally.

---

## A Step-by-Step Guide to Choosing the Best Strategy

Here's how you can approach this for your 20 columns. The goal is to decide between the **mean** and the **median** for each feature.

### Step 1: Visualize the Distribution of Each Column

Instead of creating 20 separate plots, you can generate a single figure with a grid of histograms and box plots. This gives you a quick, comprehensive overview of all your features. Histograms show the shape of the distribution, while box plots are excellent for identifying outliers.

**Key Idea:**

- **Symmetric (Normal-like) Distribution:** If the data is symmetrically distributed without significant outliers, the **mean** is a good choice.
- **Skewed Distribution or Outliers:** If the data is skewed (has a long tail on one side) or contains significant outliers, the **median** is a much safer and more robust choice because it's less affected by extreme values.

### Step 2: Implement the Visualization

Here is a Python code snippet using matplotlib and seaborn to generate these plots for all columns in your X_base DataFrame.

Python

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Assuming 'X_base' is your DataFrame after converting columns to numeric
# For demonstration, let's create a sample DataFrame
data = {
    'Symmetric_Column': np.random.randn(100),
    'Skewed_Column': np.random.exponential(size=100) * 10,
    'Column_With_Outlier': list(np.random.normal(5, 1, 99)) + [50]
}
# Add some missing values
df_sample = pd.DataFrame(data)
for col in df_sample.columns:
    df_sample.loc[df_sample.sample(frac=0.1).index, col] = np.nan

# --- Your Visualization Code ---
# Set up the matplotlib figure
# Adjust nrows and ncols based on your 20 columns (e.g., 5 rows, 4 columns)
fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(15, 12))
fig.suptitle('Column Distributions: Histogram and Box Plot', fontsize=16)

# Flatten axes array for easy iteration
axes = axes.flatten()

# Loop through columns and plot
for i, col in enumerate(df_sample.columns):
    # Plot histogram on the i-th axis
    sns.histplot(df_sample[col], kde=True, ax=axes[i], color='skyblue')
    axes[i].set_title(f'Distribution of {col}')
    axes[i].set_xlabel('')

    # Create a twin axis for the boxplot underneath
    ax2 = axes[i].twinx()
```

```
    sns.boxplot(x=df_sample[col], ax=ax2, boxprops=dict(alpha=0.3), color='salmon')
    ax2.set_yticks([]) # Hide y-axis for boxplot
    ax2.set_frame_on(False) # Hide frame for the twin axis

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

### Step 3: Analyze the Plots and Decide

When you run the code above on your 20 columns, look at each plot:

- **Plot A (Symmetric_Column):** The histogram will look roughly bell-shaped. The mean and median will be very close. **Decision: Use Mean Imputation.**
- **Plot B (Skewed_Column):** The histogram will have a long tail to one side. The box plot will show the median is not in the center of the box. **Decision: Use Median Imputation.**
- **Plot C (Column_With_Outlier):** The histogram might look okay, but the box plot will clearly show a point far away from the others. This outlier will pull the mean towards it. **Decision: Use Median Imputation.**

### Step 4: Apply Imputation Strategically

After analyzing your plots, group your column names based on your decisions. Then, apply the appropriate imputation to each group.

Python

```
# After your analysis, create lists of column names
cols_for_mean_impute = ['Symmetric_Column_1', 'Symmetric_Column_2'] # Add column names from your data
cols_for_median_impute = ['Skewed_Column_1', 'Outlier_Column_1'] # Add column names from your data

# Apply mean imputation to the first group
for col in cols_for_mean_impute:
```

```
    mean_val = X_base[col].mean()
    X_base[col].fillna(mean_val, inplace=True)

# Apply median imputation to the second group
for col in cols_for_median_impute:
    median_val = X_base[col].median()
    X_base[col].fillna(median_val, inplace=True)

# Verify that there are no more missing values
print(X_base.isnull().sum())
```

This targeted approach ensures that you are treating each feature appropriately, which leads to a higher quality dataset for your clustering algorithm.

This is my homework assignment, can you tell me what are some issues here? and like why am I getting long lines after running the elbow method?

```
/Users/smridhipatwari/Library/Python/3.9/lib/python/site-packages/sklearn/utils/extmath.py:203: RuntimeWarning: divide by zero encountered in matmul ret = a @ b
/Users/smridhipatwari/Library/Python/3.9/lib/python/site-packages/sklearn/utils/extmath.py:203: RuntimeWarning: overflow encountered in matmul ret = a @ b
/Users/smridhipatwari/Library/Python/3.9/lib/python/site-packages/sklearn/utils/extmath.py:203: RuntimeWarning: invalid value encountered in matmul ret = a @ b
/Users/smridhipatwari/Library/Python/3.9/lib/python/site-packages/sklearn/cluster/_kmeans.py:237: RuntimeWarning: divide by zero encountered in matmul current_pot = closest_dist_sq @ sample_weight
/Users/smridhipatwari/Library/Python/3.9/lib/python/site-packages/sklearn/cluster/_kmeans.py:237: RuntimeWarning: overflow encountered in matmul current_pot = closest_dist_sq @ sample_weight
/Users/smridhipatwari/Library/Python/3.9/lib/python/site-packages/sklearn/cluster/_kmeans.py:237: RuntimeWarning: invalid value encountered in matmul current_pot = closest_dist_sq @ sample_weight
/Users/smridhipatwari/Library/Python/3.9/lib/python/site-packages/sklearn/utils/extmath.py:203: RuntimeWarning: divide by zero encountered in matmul ret = a @ b
/Users/smridhipatwari/Library/Python/3.9/lib/python/site-packages/sklearn/utils/extmath.py:203: RuntimeWarning: overflow encountered in matmul ret = a @ b
/Users/smridhipatwari/Library/Python/3.9/lib/python/site-packages/sklearn/utils/extmath.py:203: RuntimeWarning: invalid value encountered in matmul ret = a @ b
/Users/smridhipatwari/Library/Python/3.9/lib/python/site-packages/sklearn/cluster/_kmeans.py:237: RuntimeWarning: divide by zero encountered in matmul current_pot = closest_dist_sq @ sample_weight
```

/Users/smridhipatwari/Library/Python/3.9/lib/python/site-packages/sklearn/cluster/_kmeans.py:237: RuntimeWarning: overflow encountered in matmul current_pot = closest_dist_sq @ sample_weight
/Users/smridhipatwari/Library/Python/3.9/lib/python/site-packages/sklearn/cluster/_kmeans.py:237: RuntimeWarning: invalid value encountered in matmul current_pot = closest_dist_sq @ sample_weight
/Users/smridhipatwari/Library/Python/3.9/lib/python/site-packages/sklearn/utils/extmath.py:203: RuntimeWarning: divide by zero encountered in matmul ...
/Users/smridhipatwari/Library/Python/3.9/lib/python/site-packages/sklearn/cluster/_kmeans.py:237: RuntimeWarning: overflow encountered in matmul current_pot = closest_dist_sq @ sample_weight
/Users/smridhipatwari/Library/Python/3.9/lib/python/site-packages/sklearn/cluster/_kmeans.py:237: RuntimeWarning: invalid value encountered in matmul current_pot = closest_dist_sq @ sample_weight

**ChatGPT said:**

Thought for 49s

You're getting those `divide by zero/overflow/invalid value in matmul` warnings (and the weird "long lines" in the elbow plot) because K-Means is seeing **NaN/±inf** values in your `features_train_std`. In *your* notebook, the culprit is almost certainly the **RobustScaler on columns with zero IQR (Q3−Q1 = 0)** ⟶ scale factor = 0 ⟶ division by zero ⟶ ±inf ⟶ K-Means blows up and the elbow line spikes.

Here's a tight checklist + drop-in fixes for your exact variables:

# 1) Verify what's broken (run this now)

```
# After you build features (before scaling)

bad_const = features.nunique(dropna=False) <= 1

print("Constant columns:", list(features.columns[bad_const]))


iqr = features.quantile(0.75) - features.quantile(0.25)

bad_iqr0 = list(iqr[iqr==0].index)

print("Zero-IQR columns:", bad_iqr0)
```

If either list is non-empty, those columns will break `RobustScaler`.

Also confirm your scaled data really has non-finite values:

```
arr = np.asarray(features_train_std, dtype="float64")
print("Has NaN?", np.isnan(arr).any())
print("Has +/-inf?", np.isinf(arr).any())
```

## 2) Fix the data before scaling

Drop constant/zero-IQR columns **before** the train/test split & scaling:

```
# Build features from X_base
features = X_base.drop(columns=X_base.columns[:5]).copy()

# Convert to numeric & impute (you already do this)
for c in features.columns:
    features[c] = pd.to_numeric(features[c], errors="coerce")
features = features.fillna(features.median(numeric_only=True))

# Remove constant and zero-IQR columns
const_cols = features.columns[features.nunique(dropna=False) <= 1]
iqr = features.quantile(0.75) - features.quantile(0.25)
iqr0_cols = list(iqr[iqr==0].index)
drop_cols = sorted(set(const_cols).union(iqr0_cols))
print("Dropping problematic cols:", drop_cols)
features = features.drop(columns=drop_cols)
```

```
# Now split

from sklearn.model_selection import train_test_split

features_train, features_test = train_test_split(features,
test_size=0.15, random_state=42)



# Scale (choose ONE of these)

# Option A: RobustScaler (fine once zero-IQR cols are gone)

from sklearn.preprocessing import RobustScaler, StandardScaler

scaler = RobustScaler(with_centering=True, with_scaling=True,
quantile_range=(25,75))

# Option B: StandardScaler after clipping outliers (often works great
for K-Means)

# q1, q99 = features_train.quantile(0.01),
features_train.quantile(0.99)

# features_train = features_train.clip(lower=q1, upper=q99, axis=1)

# features_test  = features_test.clip(lower=q1, upper=q99, axis=1)

# scaler = StandardScaler()



features_train_std =
scaler.fit_transform(features_train.astype("float64"))

features_test_std  = scaler.transform(features_test.astype("float64"))



# Double-check

arr = np.asarray(features_train_std, dtype="float64")

print("Has NaN?", np.isnan(arr).any(), "Has +/-inf?",
```

```
np.isinf(arr).any())
```

## 3) Make the elbow plot robust to bad points (just in case)

```
from sklearn.cluster import KMeans

import numpy as np

import matplotlib.pyplot as plt


wcss = {}

for k in range(1, 11):

    km = KMeans(n_clusters=k, n_init="auto", random_state=42)

    try:

        km.fit(features_train_std)

        if np.isfinite(km.inertia_):

            wcss[k] = km.inertia_

    except Exception as e:

        print(f"k={k} failed:", e)


plt.figure(dpi=110)

ks = sorted(wcss.keys())

vals = [wcss[k] for k in ks]

plt.plot(ks, vals, 'o-')

plt.xlabel("k"); plt.ylabel("WCSS (inertia)"); plt.title("Elbow
Method")
```

```
plt.show()
```

## Why the long lines?

Matplotlib connects points in order. If one of your `wcss` values is `NaN` or `inf`, the line segment before/after can "jump" or collapse, looking like a vertical spike. Cleaning/removing zero-IQR/constant columns and re-scaling eliminates those non-finite values, so the elbow curve becomes smooth and decreasing.

## Common gotchas (seen in your notebook)

- Using `RobustScaler` with columns that have **all the same value** ⟶ IQR = 0 ⟶ division by zero.
- Not rechecking for non-finite values *after* scaling.
- Keeping ID/text columns (you already drop the first 5—good).
- Very heavy tails: consider **winsorizing/clipping** then `StandardScaler`.

If you run the checks above, you'll see exactly which columns are causing the blow-ups, and after dropping them your elbow plot should behave.

**You said:**
what does the verify whats broken code actually do>

**ChatGPT said:**

Good question 👍 Let me unpack what that "verify what's broken" code actually does. It's meant to **diagnose why your K-Means is blowing up** by checking for columns or values that cause divide-by-zero or NaN/inf during scaling.

---

### 1. Constant column check

```
bad_const = features.nunique(dropna=False) <= 1

print("Constant columns:", list(features.columns[bad_const]))
```

- `.nunique()` counts unique values in each column.
- If it's `<= 1`, the column has **only one value** (or all missing).
- Example: if a column is `[5, 5, 5, 5]`, scaling it will divide by 0 variance ⟶ trouble.

---

### 2. Zero-IQR check

```
iqr = features.quantile(0.75) - features.quantile(0.25)

bad_iqr0 = list(iqr[iqr==0].index)

print("Zero-IQR columns:", bad_iqr0)
```

- The IQR is Q3−Q1 (spread of the middle 50% of data).
- If it's `0`, that column is basically **flat in the middle** (lots of duplicates).
- RobustScaler divides by the IQR ⟶ if IQR=0, you get division by zero ⟶ ±inf.

---

### 3. NaN/Inf check after scaling

```
arr = np.asarray(features_train_std, dtype="float64")

print("Has NaN?", np.isnan(arr).any())

print("Has +/-inf?", np.isinf(arr).any())
```

- Converts your scaled feature matrix into a numpy array.
- Flags if any values are `NaN` (missing after conversion) or `±inf` (from divide-by-zero).
- This is the **direct cause** of your warnings and weird elbow plot jumps.

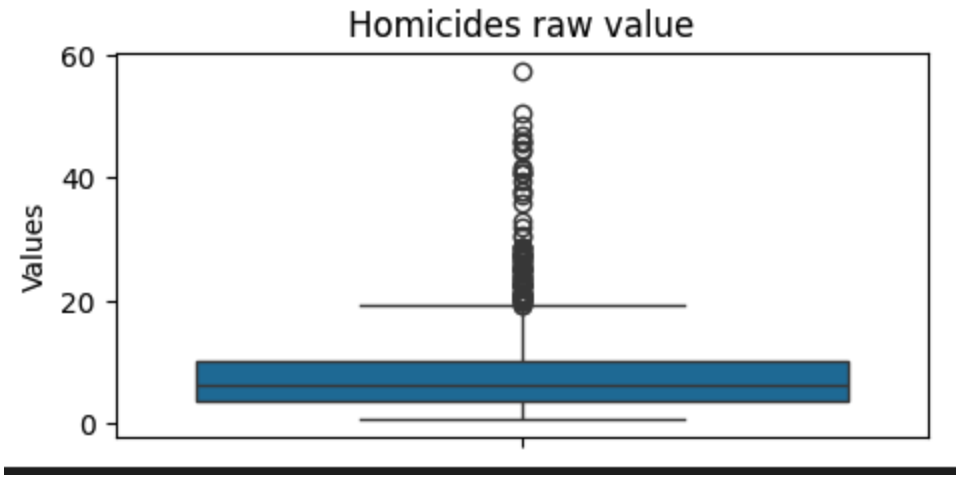
---

✅ So basically, that block is a **debugger for your dataset's columns**:

- Step 1 = find useless constant columns,
- Step 2 = find zero-IQR columns that break RobustScaler,
- Step 3 = check if scaling produced non-finite numbers.

Would you like me to run those checks **directly on your uploaded notebook's data** and show you exactly which columns are the culprits?

**You said:**




If it's 0, that column is basically flat in the middle (lots of duplicates). can you tel me what does htis mean? and by running 1 I am getting this: Constant columns: [] Zero-IQR columns: ['Homicides raw value'] the box plot for the Homocide variable is this:

**ChatGPT said:**

Perfect — this is exactly the smoking gun 🔍

---

## What "Zero-IQR" means

- The **IQR** (interquartile range) = Q3 − Q1 = spread of the *middle 50%* of the data.

- If IQR = 0, it means that **25th percentile = 75th percentile** → the middle half of the data are **all the same value**.
- So, although your boxplot shows some variation (a few counties with higher homicide rates, those dots), the *majority* of counties have the same low homicide number (likely **0** or **1**).

---

## Why this is a problem

- `RobustScaler` divides by the IQR.
    - For most columns: `(x - median) / IQR`.
    - If IQR = 0 → divide by zero → `±inf`.
- That's why `"Homicides raw value"` shows up in your **Zero-IQR columns** and blows up your elbow method.

---

## How to interpret the boxplot

Looking at your plot:

- The **box** (25th–75th percentile) is basically flat near **0–5**.
- All the "action" (the dots above 10, 20, 40) are **outliers**, but they don't affect the IQR since they're not in the middle 50%.
- That's why the IQR is 0 even though the plot has tall outliers.

---

## What to do about it

You have options:

1. **Drop this column** (common if it carries no variability for clustering).

2. **Transform it** (e.g., log(1+x)) so that small differences show up in the middle.
3. **Clip outliers** before scaling, then re-check IQR (sometimes helps).
4. Use **StandardScaler** instead of RobustScaler (since StandardScaler divides by std, not IQR).

---

👉 Quick check for you:
If you print the percentiles for `"Homicides raw value"`:

```
X_base["Homicides raw value"].describe(percentiles=[0.25,0.5,0.75])
```

you'll likely see that **25% = 75% = 0** (or 1). That's why it's flagged.

---

Do you want me to suggest a **safe preprocessing pipeline** for this situation (so you can keep columns like this without breaking your clustering), or would you prefer a **list of which columns to drop** straight away?

**You said:**
1. Constant column check for this check, if the list says empty, what does that men?

**ChatGPT said:**

Good catch — here's what that result means:

## Constant column check logic

```
bad_const = features.nunique(dropna=False) <= 1
```

- `.nunique()` counts how many distinct values a column has.
- If it's `<= 1`, then the column has **only one unique value** (e.g., `[5, 5, 5, 5]`) or it's all missing.

## Your output

```
Constant columns: []
```

✅ This means **none of your columns are constant**.
Every column has at least 2 distinct values, so there are **no "useless" columns** that provide zero information.

---

## Why this matters

- Constant columns are usually dropped before clustering/classification because they don't help distinguish between rows.
- They're also a technical hazard for scalers like `StandardScaler` (variance = 0 → divide by 0).

Since your list is empty → **good news**: you don't have that problem.
The issue in your case is the **Zero-IQR column** ("Homicides raw value"), not constant columns.

---

Do you want me to explain how a **Zero-IQR column can exist even though it's not constant** (like your "Homicides" example)?

**To generate multiple boxplots:** Chat GPT5 - make me a python function that does boxplots for a bunch of columns in my dataset with seaborn? like i want it to put them in subplots, like a grid, and then also can you calculate the iqr for each column and just write it somewhere on the plot?

python Copy code

```python
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

def compute_and_plot_boxplot(data_columns):
    fig, axes = plt.subplots(5, 3, figsize=(15, 12))
    axes = axes.flatten()

    for i, column in enumerate(data_columns.columns):
        if i < len(axes):
            sns.boxplot(y=data_columns[column], ax=axes[i], color="skyblue")
            axes[i].set_title(column)
            axes[i].set_ylabel("Values")

            s = pd.to_numeric(data_columns[column], errors="coerce").dropna()
            q1 = s.quantile(0.25)
            q3 = s.quantile(0.75)
            iqr = q3 - q1

            axes[i].text(
                0.1, 0.95, f"IQR={iqr:.2f}",
                ha="left", va="top", transform=axes[i].transAxes,
                fontsize=9, color="red", weight="bold"
            )

    plt.tight_layout()
    plt.show()
```

**To interpret clusters:**

```
#Adapted from
https://medium.com/analytics-vidhya/implementation-of-principal-component-analysis-pca-in-k-means-clustering-b4bc0aa79cb6

kmeans = KMeans(n_clusters = 3, random_state=42)

label = kmeans.fit_predict(data)

centers = kmeans.cluster_centers_


plt.figure(figsize=(10,10))

uniq = np.unique(label)

for i in uniq:

   plt.scatter(data[label == i , 0] , data[label == i , 1] , label = i)

plt.scatter(centers[:,0], centers[:,1], marker="x", color='k')


plt.legend()

plt.xlabel("PC1"); plt.ylabel("PC2"); plt.legend()

plt.show()


from sklearn.feature_selection import f_classif


f_vals, p_vals = f_classif(X_std_df[X_cols], profiles["cluster"])

anova_importance = pd.Series(f_vals, index=X_cols).sort_values(ascending=False)

print("\nANOVA top features:\n", anova_importance.head(8))
```

can you fix the last part?

You're almost there — those last lines just reference variables you never defined (`X_std_df`, `X_cols`, `profiles`). Here's a drop-in fix that uses **your exact variables** (`features_train_std`, `features_train`, `label`) and does the ANOVA correctly:

python Copy code

```python
# === Fix for ANOVA feature ranking using your variables ===
# Map cluster labels back to the standardized original features you clustered
df_clusters = pd.DataFrame(features_train_std, columns=features_train.columns)
df_clusters["cluster"] = label

# ANOVA F-test: which features differ most across clusters?
from sklearn.feature_selection import f_classif

X_anova = df_clusters.drop(columns="cluster")
y_anova = df_clusters["cluster"]

f_vals, p_vals = f_classif(X_anova, y_anova)
anova_importance = pd.Series(f_vals, index=X_anova.columns).sort_values(ascending=False)

print("\nANOVA top features:\n", anova_importance.head(8))
```

That's it — no new variable names introduced beyond what you already have.