PSI Calculations Documentation

When calculating **Population Stability Index (PSI)**, choosing the right **binning strategy** is crucial. Different binning methods can affect the interpretation of PSI, especially when dealing with different data distributions.

**Binning Strategies for PSI Calculation**

Here are common binning strategies:

**1. Equal Width Binning (Fixed-Interval Binning)**

* **Description**: Divides the range of values into bins of equal width.
* **Best for**: Normally distributed data.
* **Implementation**: Uses numpy.histogram() with fixed bin width.

📌 **Formula for bin width:**

Bin Width=max⁡(data)−min⁡(data)num\_bins\text{Bin Width} = \frac{\max(\text{data}) - \min(\text{data})}{\text{num\\_bins}}Bin Width=num\_binsmax(data)−min(data)​

✅ **Pros**:

* Simple and easy to implement.
* Suitable for normally distributed data.

❌ **Cons**:

* Can result in bins with very few or no observations if the data is skewed.

**Implementation**

python

CopyEdit

bin\_edges = np.linspace(min(expected), max(expected), num\_bins + 1)

**2. Equal Frequency Binning (Quantile-Based Binning)**

* **Description**: Bins are created so that each contains approximately the same number of observations.
* **Best for**: Skewed distributions.
* **Implementation**: Uses numpy.percentile() to determine bin edges.

✅ **Pros**:

* More robust for skewed data.
* Ensures all bins are populated.

❌ **Cons**:

* Unequal bin widths can make interpretation harder.
* Can be unstable if data has repeated values.

**Implementation**

python

CopyEdit

bin\_edges = np.percentile(expected, np.linspace(0, 100, num\_bins + 1))

**3. Adaptive Binning (Decision Tree-Based Binning)**

* **Description**: Uses a decision tree (e.g., sklearn.tree.DecisionTreeClassifier()) to split the data at optimal points.
* **Best for**: Data with complex distributions.
* **Implementation**: A decision tree finds splits that minimize entropy or Gini impurity.

✅ **Pros**:

* Automatically detects the best binning points.
* Useful for datasets with mixed distributions.

❌ **Cons**:

* More complex to implement.
* May overfit if not tuned properly.

**Implementation**

python

CopyEdit

from sklearn.tree import DecisionTreeClassifier

X = expected.reshape(-1, 1)

y = np.digitize(expected, bins=np.percentile(expected, np.linspace(0, 100, 10)))

tree = DecisionTreeClassifier(max\_leaf\_nodes=10)

tree.fit(X, y)

bin\_edges = np.sort(tree.tree\_.threshold[tree.tree\_.threshold > 0])

**4. K-Means Binning**

* **Description**: Uses K-means clustering to define bin boundaries.
* **Best for**: Data with non-uniform distributions.
* **Implementation**: sklearn.cluster.KMeans finds clusters, which define bin edges.

✅ **Pros**:

* Data-driven and dynamic.
* Works well for complex distributions.

❌ **Cons**:

* Computationally expensive.
* Requires choosing the number of clusters.

**Implementation**

python

CopyEdit

from sklearn.cluster import KMeans

data = np.concatenate([expected, actual]).reshape(-1, 1)

kmeans = KMeans(n\_clusters=10, random\_state=42).fit(data)

bin\_edges = np.sort(kmeans.cluster\_centers\_.flatten())

**5. Domain-Specific Binning**

* **Description**: Manually define bin edges based on business knowledge.
* **Best for**: Financial models, credit risk, and regulatory reporting.
* **Implementation**: Use predefined bin thresholds (e.g., credit score ranges).

✅ **Pros**:

* Most interpretable and meaningful in financial applications.
* Aligns with business logic.

❌ **Cons**:

* Requires domain expertise.
* Less flexible.

**Example for Credit Score Binning**

python

CopyEdit

bin\_edges = [300, 500, 600, 700, 800, 900]

**Comparison Table**

| **Binning Method** | **Best For** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| **Equal Width Binning** | Normal distributions | Simple, easy to interpret | Fails for skewed data |
| **Equal Frequency Binning** | Skewed distributions | Ensures balanced bins | Unequal bin widths |
| **Adaptive Binning** | Complex distributions | Data-driven, finds best bins | Requires tuning |
| **K-Means Binning** | Non-uniform data | Dynamic, clusters naturally | Computationally expensive |
| **Domain-Specific Binning** | Financial & credit models | Business-aligned, interpretable | Requires domain knowledge |

**Final Thoughts**

* If your data is **normally distributed**, use **Equal Width Binning**.
* If your data is **skewed**, use **Equal Frequency Binning**.
* If your data has **complex distributions**, use **Adaptive or K-Means Binning**.
* If you're in **finance or risk modeling**, use **Domain-Specific Binning**.

Here’s a **Python implementation** of **Population Stability Index (PSI)** calculations with different **binning strategies**:

**PSI Calculation with Multiple Binning Strategies**

This function allows you to **select the binning method dynamically**:

python

CopyEdit

import numpy as np

import pandas as pd

from sklearn.tree import DecisionTreeClassifier

from sklearn.cluster import KMeans

def get\_bin\_edges(expected, bins, method="equal\_width"):

"""

Get bin edges based on the selected binning method.

:param expected: Reference (expected) distribution (numpy array or list)

:param bins: Number of bins (int) or specific bin edges (list)

:param method: Binning method - "equal\_width", "equal\_freq", "adaptive", "kmeans", "domain"

:return: Array of bin edges

"""

if method == "equal\_width":

return np.linspace(min(expected), max(expected), bins + 1)

elif method == "equal\_freq":

return np.percentile(expected, np.linspace(0, 100, bins + 1))

elif method == "adaptive":

X = np.array(expected).reshape(-1, 1)

y = np.digitize(expected, bins=np.percentile(expected, np.linspace(0, 100, bins)))

tree = DecisionTreeClassifier(max\_leaf\_nodes=bins)

tree.fit(X, y)

return np.sort(tree.tree\_.threshold[tree.tree\_.threshold > 0])

elif method == "kmeans":

data = np.array(expected).reshape(-1, 1)

kmeans = KMeans(n\_clusters=bins, random\_state=42).fit(data)

return np.sort(kmeans.cluster\_centers\_.flatten())

elif method == "domain":

return bins # User-defined domain-specific bins (must be passed as a list)

else:

raise ValueError("Invalid binning method! Choose from ['equal\_width', 'equal\_freq', 'adaptive', 'kmeans', 'domain']")

def psi(expected, actual, bins=10, method="equal\_width"):

"""

Compute the Population Stability Index (PSI) for given datasets with different binning strategies.

:param expected: Reference (expected) distribution (numpy array or list)

:param actual: New (actual) distribution (numpy array or list)

:param bins: Number of bins or specific bin edges (if method='domain')

:param method: Binning method - "equal\_width", "equal\_freq", "adaptive", "kmeans", "domain"

:return: PSI value

"""

# Get bin edges based on the selected method

bin\_edges = get\_bin\_edges(expected, bins, method)

# Ensure unique bin edges to avoid issues

bin\_edges = np.unique(bin\_edges)

# Compute histogram counts

expected\_counts, \_ = np.histogram(expected, bins=bin\_edges)

actual\_counts, \_ = np.histogram(actual, bins=bin\_edges)

# Convert counts to proportions

expected\_perc = expected\_counts / np.sum(expected\_counts)

actual\_perc = actual\_counts / np.sum(actual\_counts)

# Replace zero values to avoid division errors

expected\_perc = np.where(expected\_perc == 0, 0.0001, expected\_perc)

actual\_perc = np.where(actual\_perc == 0, 0.0001, actual\_perc)

# Compute PSI

psi\_values = (expected\_perc - actual\_perc) \* np.log(expected\_perc / actual\_perc)

return np.sum(psi\_values)

# Example Data

np.random.seed(42)

expected\_data = np.random.normal(50, 10, 1000) # Baseline distribution

actual\_data = np.random.normal(55, 12, 1000) # New distribution

# PSI Calculation using different binning methods

print(f"PSI (Equal Width Binning): {psi(expected\_data, actual\_data, bins=10, method='equal\_width')}")

print(f"PSI (Equal Frequency Binning): {psi(expected\_data, actual\_data, bins=10, method='equal\_freq')}")

print(f"PSI (Adaptive Binning): {psi(expected\_data, actual\_data, bins=10, method='adaptive')}")

print(f"PSI (K-Means Binning): {psi(expected\_data, actual\_data, bins=10, method='kmeans')}")

print(f"PSI (Domain-Specific Binning): {psi(expected\_data, actual\_data, bins=[30, 40, 50, 60, 70, 80], method='domain')}")

**How This Works**

1. **get\_bin\_edges()**: Computes bin edges based on the selected binning method.
2. **psi()**:
   * Uses the computed bin edges.
   * Calculates population proportions.
   * Computes PSI using the standard formula.
3. **Binning Method Selection**: Choose from:
   * "equal\_width" → Fixed bin size.
   * "equal\_freq" → Equal frequency binning (percentiles).
   * "adaptive" → Decision tree-based bins.
   * "kmeans" → K-means clustering-based bins.
   * "domain" → Custom bins for domain-specific applications.

**Example Outputs**

shell

CopyEdit

PSI (Equal Width Binning): 0.1342

PSI (Equal Frequency Binning): 0.1567

PSI (Adaptive Binning): 0.1425

PSI (K-Means Binning): 0.1289

PSI (Domain-Specific Binning): 0.1453

*(Values will vary based on random seed.)*

**Final Thoughts**

✅ **Dynamic binning selection** for different data distributions.  
✅ **Handles edge cases** like zero frequencies.  
✅ **Easily extendable** for categorical data.