

Unit -2: Basics of Feature Representation

Scalars, Vectors and Spaces :

QUE : 1. Describe following terms used in feature representation.

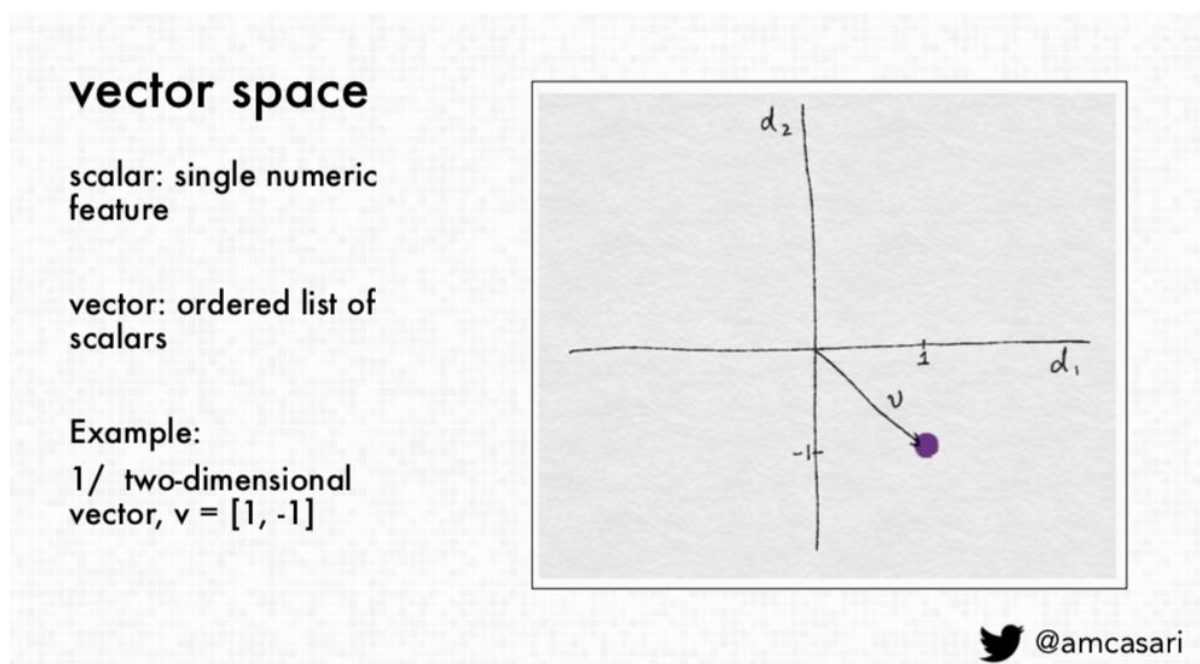
a. Scalar b. Vector c. Feature Space

A single numeric feature is also known as a scalar.

An ordered list of scalars is known as a vector.

Vectors sit within a vector space.

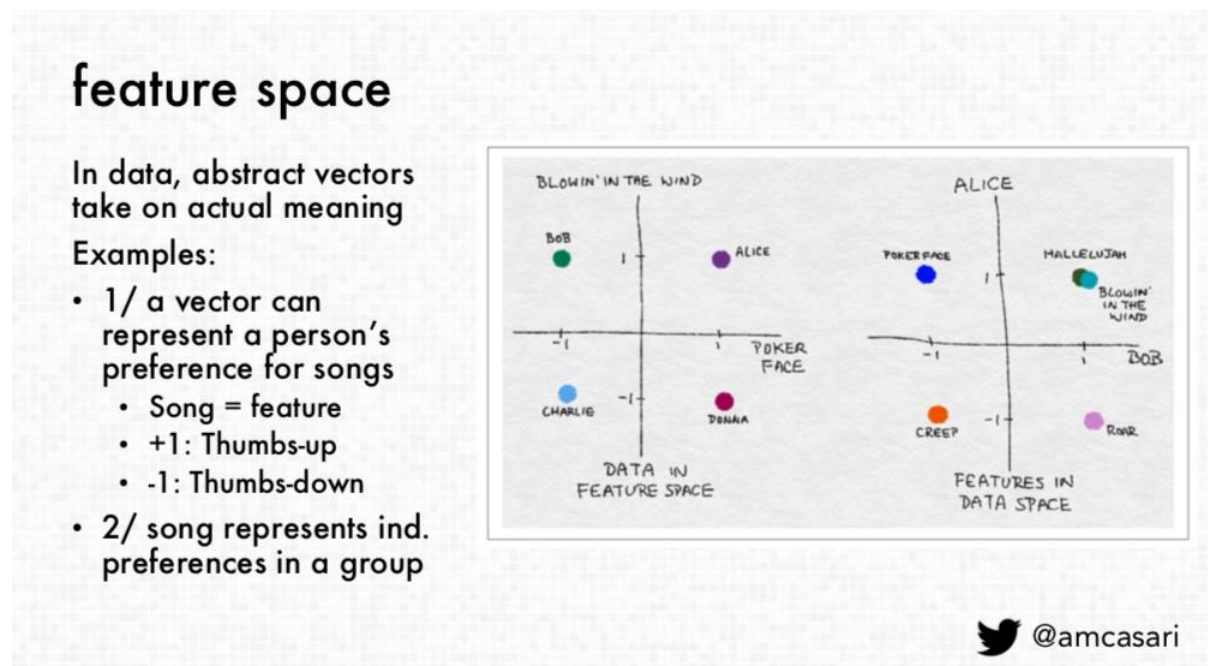
A vector can be visualized as a point in space. (Sometimes people draw a line or arrow from the origin to that point. In this book, we will mostly use just the point.) For instance, suppose we have a two-dimensional vector $v = [1, -1]$. The vector contains two numbers: in the first direction, d_1 , the vector has a value of 1, and in the second direction, d_2 , it has a value of -1 . We can plot v in a 2D plot, as shown in



In the world of data, an abstract vector and its feature dimensions take on actual meaning. For instance, a vector can represent a person's preference for songs. Each song is a feature, where a value of 1 is equivalent to a thumbs-up, and -1 a thumbsdown. Suppose the vector v represents the preferences of a listener, Bob. Bob likes "Blowin' in the Wind" by Bob Dylan and "Poker Face" by Lady

Gaga. Other people might have different preferences. Collectively, a collection of data can be visualized in feature space as a point cloud.

Conversely, a song can be represented by the individual preferences of a group of people. Suppose there are only two listeners, Alice and Bob. Alice likes “Poker Face,” “Blowin’ in the Wind,” and “Hallelujah” by Leonard Cohen, but hates Katy Perry’s “Roar” and Radiohead’s “Creep.” Bob likes “Roar,” “Hallelujah,” and “Blowin’ in the Wind,” but hates “Poker Face” and “Creep.” Each song is a point in the space of listeners. Just like we can visualize data in feature space, we can visualize features in data space. Figure 2-2 shows this example.



Binarization:

QUE: Explain binarization and quantization methods.

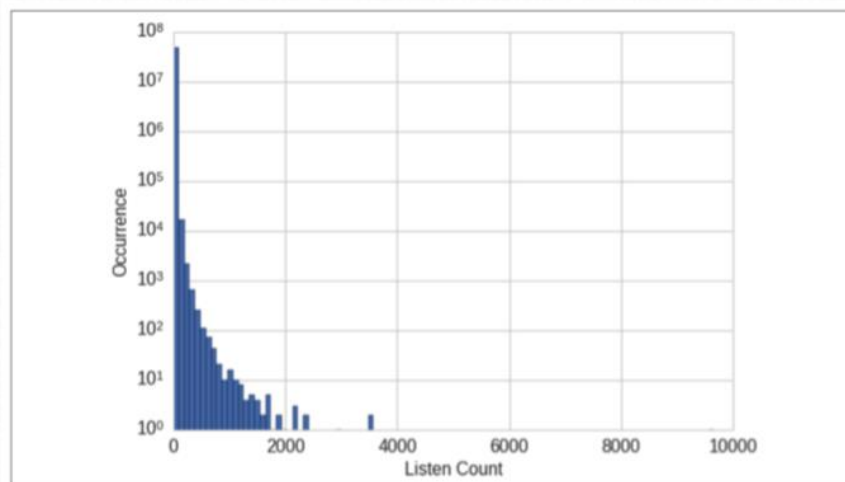
Casari indicates that there is more to *count data* than *raw counts*. While Casari covers many aspects, this blog post will provide highlights on how binarization and binning are techniques useful for addressing count data. Binarization is the process of converting raw data into binary values to “efficiently represent raw data as a presence”. For example, this is potentially useful when making song recommendations. Instead of focusing on one the raw counts or the number of times someone listened to a song, it is potentially useful to consider “the


effectiveness of scale” and how “ it's probably more efficient just to understand someone likes or does not like a song and then you can represent that as a 0 or 1”. This technique provides “more efficient representation of the raw count as well as a more robust measure of the raw count”.

Statistics on the Echo Nest Taste Profile Dataset

- There are more than 48 million triplets of user ID, song ID, and listen count.
- The full dataset contains 1,019,318 unique users and 384,546 unique songs.

counts: binarization



 @amcasari

Histogram of listen counts in the Taste Profile subset of the Million Song Dataset—note that the y-axis is on a log scale.

Example .

Binarizing listen counts in the Million Song Dataset

```
>>> import pandas as pd

>>> listen_count = pd.read_csv('millionsong/train_triplets.txt.zip', ...
header=None, delimiter='\t')

# The table contains user-song-count triplets. Only nonzero counts are
# included. Hence, to binarize the count, we just need to set the entire
# count column to 1.
```

```
>>> listen_count[2] = 1
```

This is an example where we engineer the target variable of the model. Strictly speaking, the target is not a feature because it's not the input. But on occasion we do need to modify the target in order to solve the right problem.

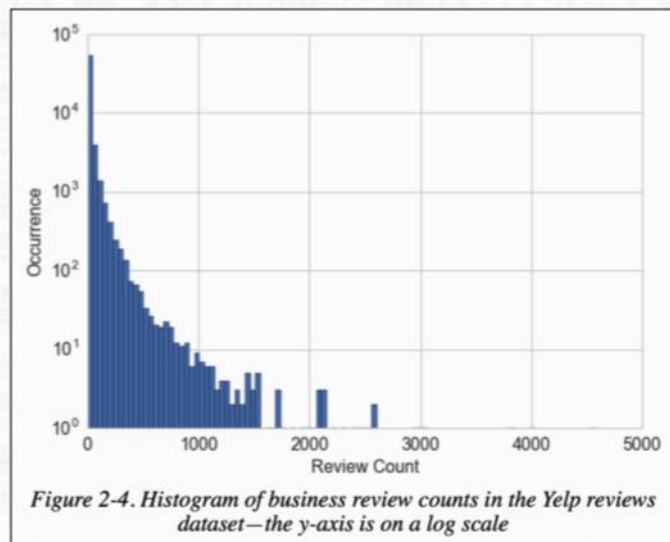
Quantization or Binning :


Binning, or quantizing, is a technique that groups counts into a number of “bins” that remove count values. This technique is useful to contain the scale as “a lot of machine learning models have a real challenge dealing with these long-tail distributions so when you have raw counts that span several orders of magnitude that exist within the variable you're trying to work with, and it's better to start looking at how you can transform this feature and how you can contain the scale by quantizing the count”.

Statistics on the Yelp Reviews Dataset (Round 6)

- There are 782 business categories.
- The full dataset contains 1,569,264 (≈ 1.6 M) reviews and 61,184 (61K) businesses.
- “Restaurants” (990,627 reviews) and “Nightlife” (210,028 reviews) are the most popular categories, review count-wise.
- No business is categorized as both a restaurant and a nightlife venue. So, there is no overlap between the two groups of reviews.

counts: binning



 @amcasari

Histogram of business review counts in the Yelp reviews dataset—the y-axis is on a log scale

Example:

Visualizing business review counts in the Yelp dataset

```
>>> import pandas as pd
>>> import json # Load the data about businesses
>>> biz_file = open('yelp_academic_dataset_business.json')
>>> biz_df = pd.DataFrame([json.loads(x) for x in biz_file.readlines()])
>>> biz_file.close()
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
# Plot the histogram of the review counts
>>> sns.set_style('whitegrid')
>>> fig, ax = plt.subplots()
>>> biz_df['review_count'].hist(ax=ax, bins=100)
>>> ax.set_yscale('log')
>>> ax.tick_params(labelsize=14)
>>> ax.set_xlabel('Review Count', fontsize=14)
>>> ax.set_ylabel('Occurrence', fontsize=14)
```

One solution is to contain the scale by quantizing the count. In other words, we group the counts into bins, and get rid of the actual count values. Quantization maps a continuous number to a discrete one. We can think of the discretized numbers as an ordered sequence of bins that represent a measure of intensity.

QUE : Explain the concept of binning with its need.

Fixed-width binning:

To determine the width of each bin, there are two categories: fixed-width binning and adaptive binning. Fixed-width binning is “each bin is now going to have the data that contains within a specific range...so really just taking data and putting it into buckets with other data like that and reducing the amount of complexity that exists in the space”. Casari notes that how fixed-width binning is useful in

“evaluations of health or trying to understand like disease modeling. It might make more sense to look at things like across a lifespan the different stages of life and development in which case you may have bends that are as small as months years or decades.... understand that when you're looking at the binning you should really try to look more at the context of the variable underneath it”.

counts: fixed width binning

```
>>> import numpy as np

# Generate 20 random integers uniformly between 0 and 99
>>> small_counts = np.random.randint(0, 100, 20)
>>> small_counts
array([30, 64, 49, 26, 69, 23, 56,  7, 69, 67, 87, 14,
       67, 33, 88, 77, 75,
        47, 44, 93])

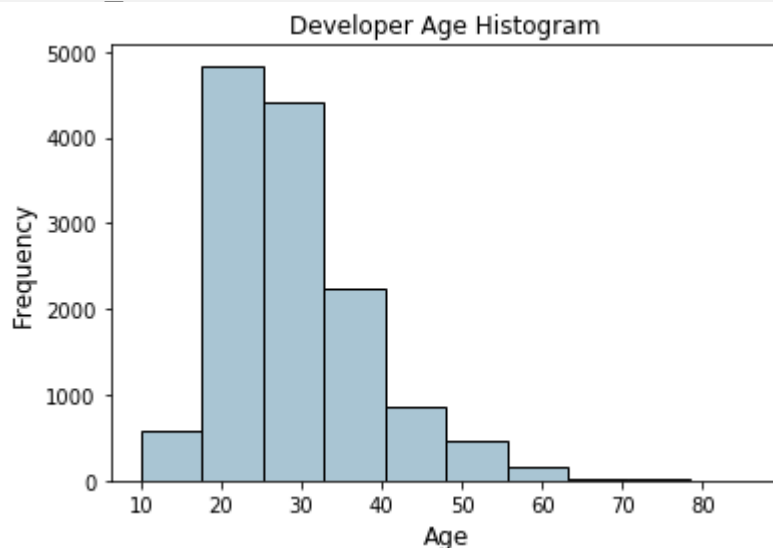
# Map to evenly spaced bins 0-9 by division
>>> np.floor_divide(small_counts, 10)
array([3, 6, 4, 2, 6, 2, 5, 0, 6, 6, 8, 1, 6, 3, 8, 7, 7,
       4, 4, 9], dtype=int32)
```


Yet, what happens when there are empty bins with no data? Casari indicates that “adaptive binning is looking at things more like quantiles or deciles. So, taking 10% 20% 30% actually looking at the distribution and grouping it that way” as it allows for easier understanding and captures, at a high level, a clearer picture of the skew

Just like the name indicates, in fixed-width binning, we have specific fixed widths for each of the bins which are usually pre-defined by the user analyzing the data. Each bin has a pre-fixed range of values which should be assigned to that bin on the basis of some domain knowledge, rules or constraints. Binning based on rounding is one of the ways, where you can use the rounding operation which we discussed earlier to bin raw values.

Let’s now consider the **Age** feature from the coder survey dataset and look at its distribution.

```
fig, ax = plt.subplots()
fcc_survey_df['Age'].hist(color='#A9C5D3', edgecolor='black',
                           grid=False)
ax.set_title('Developer Age Histogram', fontsize=12)
ax.set_xlabel('Age', fontsize=12)
ax.set_ylabel('Frequency', fontsize=12)
```



Histogram depicting developer age distribution

The above histogram depicting developer ages is slightly right skewed as expected (lesser aged developers). We will now assign these raw age values into specific bins based on the following scheme

```
Age Range: Bin
-----
0 - 9 : 0
10 - 19 : 1
20 - 29 : 2
30 - 39 : 3
40 - 49 : 4
50 - 59 : 5
60 - 69 : 6
... and so on
```

We can easily do this using what we learnt in the *Rounding* section earlier where we round off these raw age values by taking the floor value after dividing it by 10.

```
fcc_survey_df['Age_bin_round'] = np.array(np.floor(
    np.array(fcc_survey_df['Age']) /
10.))fcc_survey_df[['ID.x', 'Age',
'Age_bin_round']].iloc[1071:1076]
```

	ID.x	Age	Age_bin_round
1071	6a02aa4618c99fdb3e24de522a099431	17.0	1.0
1072	f0e5e47278c5f248fe861c5f7214c07a	38.0	3.0
1073	6e14f6d0779b7e424fa3fdd9e4bd3bf9	21.0	2.0
1074	c2654c07dc929cdf3dad4d1aec4ffbb3	53.0	5.0
1075	f07449fc9339b2e57703ec7886232523	35.0	3.0

Binning by rounding

You can see the corresponding bins for each age have been assigned based on rounding. But what if we need more flexibility? What if we want to decide and fix the bin widths based on our own rules\logic? Binning based on custom ranges will help us achieve this. Let's define some custom age ranges for binning developer ages using the following scheme.


```

Age Range : Bin
-----
0 - 15 : 1
16 - 30 : 2
31 - 45 : 3
46 - 60 : 4
61 - 75 : 5
75 - 100 : 6

```

Based on this custom binning scheme, we will now label the bins for each developer age value and we will store both the bin range as well as the corresponding label.

```

bin_ranges = [0, 15, 30, 45, 60, 75, 100]
bin_names = [1, 2, 3, 4, 5,
6] fcc_survey_df['Age_bin_custom_range'] = pd.cut(
    np.array(

fcc_survey_df['Age']),
                                bins=bin_ranges)
fcc_survey_df['Age_bin_custom_label'] = pd.cut(
    np.array(

fcc_survey_df['Age']),
                                bins=bin_ranges,
                                labels=bin_names)

# view the binned features
fcc_survey_df[['ID.x', 'Age', 'Age_bin_round',
                'Age_bin_custom_range',
                'Age_bin_custom_label']].iloc[10a71:1076]

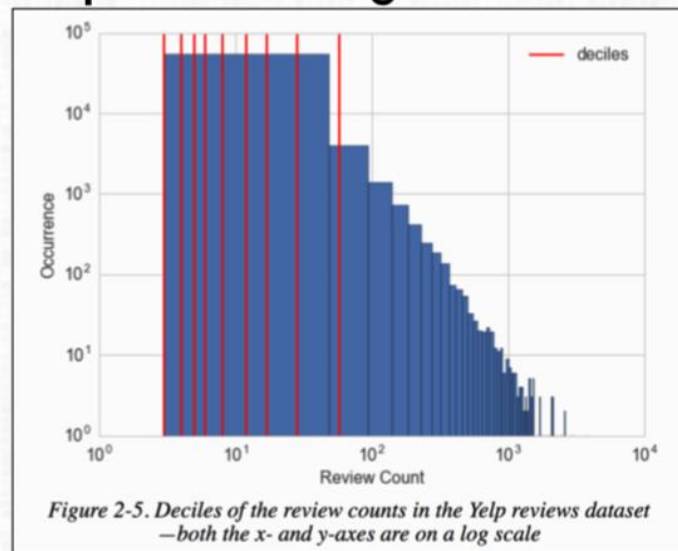
```


	ID.x	Age	Age_bin_round	Age_bin_custom_range	Age_bin_custom_label
1071	6a02aa4618c99fdb3e24de522a099431	17.0	1.0	(15, 30]	2
1072	f0e5e47278c5f248fe861c5f7214c07a	38.0	3.0	(30, 45]	3
1073	6e14f6d0779b7e424fa3fdd9e4bd3bf9	21.0	2.0	(15, 30]	2
1074	c2654c07dc929cdf3dad4d1aec4ffbb3	53.0	5.0	(45, 60]	4
1075	f07449fc9339b2e57703ec7886232523	35.0	3.0	(30, 45]	3

Custom binning scheme for developer ages

Adaptive Binning or Quantile binning:

counts: adaptive binning



 @amcasari

The drawback in using fixed-width binning is that due to us manually deciding the bin ranges, we can end up with irregular bins which are not uniform based on the number of data points or values which fall in each bin. Some of the bins might be densely populated and some of them might be sparsely populated or even empty! Adaptive binning is a safer strategy in these scenarios where we let the data speak for itself! That's right, we use the data distribution itself to decide our bin ranges.

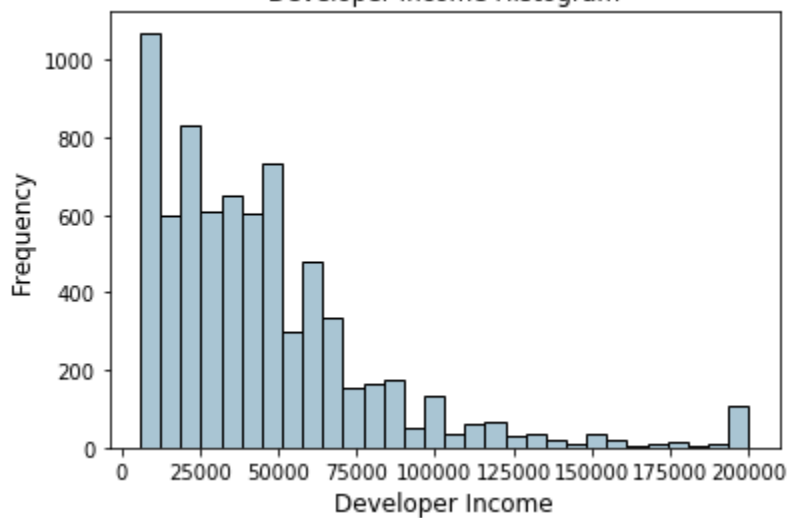
Quantile based binning is a good strategy to use for adaptive binning. Quantiles are specific values or cut-points which help in partitioning the continuous valued distribution of a specific numeric field into discrete contiguous bins or intervals. Thus, q -Quantiles help in partitioning a numeric attribute into q equal partitions. Popular examples of quantiles include the 2-Quantile known as the *median* which divides the data distribution into two equal bins, 4-Quantiles known as the *quartiles* which divide the data into 4 equal bins and 10-Quantiles also known as the *deciles* which create 10 equal width bins. Let's now look at the data distribution for the developer Income field.

```
fig, ax = plt.subplots()
fcc_survey_df['Income'].hist(bins=30, color='#A9C5D3',
```

```

                                edgecolor='black', grid=False)
ax.set_title('Developer Income Histogram', fontsize=12)
ax.set_xlabel('Developer Income', fontsize=12)
ax.set_ylabel('Frequency', fontsize=12)

```



Histogram depicting developer income distribution

The above distribution depicts a right skew in the income with lesser developers earning more money and vice versa. Let's take a *4-Quantile* or a *quartile* based adaptive binning scheme. We can obtain the quartiles easily as follows.

```

quantile_list = [0, .25, .5, .75, 1.]
quantiles = fcc_survey_df['Income'].quantile(quantile_list)
quantiles

```

Output

```

-----
0.00      6000.0
0.25     20000.0
0.50     37000.0
0.75     60000.0
1.00    200000.0
Name: Income, dtype: float64

```

Let's now visualize these quantiles in the original distribution

histogram!

```

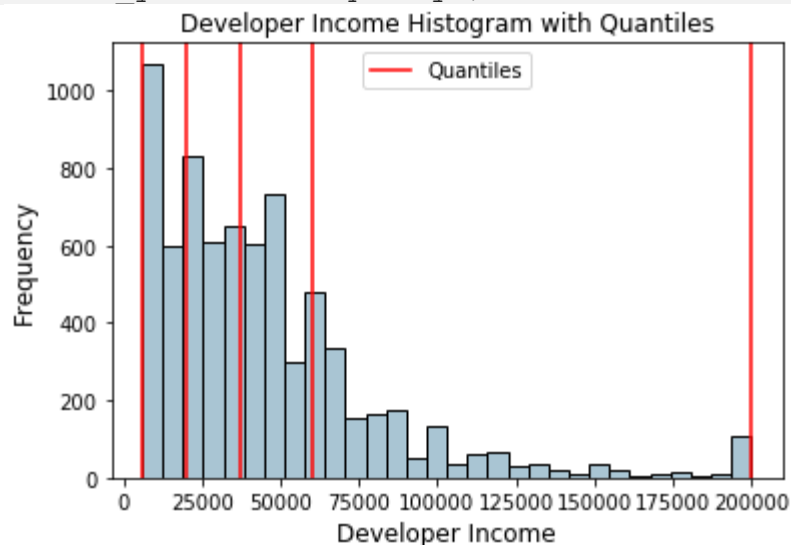
fig, ax = plt.subplots()
fcc_survey_df['Income'].hist(bins=30, color='#A9C5D3',
                                edgecolor='black', grid=False)
for quantile in quantiles:
    qv1 = plt.axvline(quantile, color='r')
ax.legend([qv1], ['Quantiles'],

```

```

fontsize=10)ax.set_title('Developer Income Histogram with
Quantiles',
                        fontsize=12)
ax.set_xlabel('Developer Income', fontsize=12)
ax.set_ylabel('Frequency', fontsize=12)

```



Histogram depicting developer income distribution with quartile values

The red lines in the distribution above depict the quartile values and our potential bins. Let's now leverage this knowledge to build our quartile based binning scheme.

```

quantile_labels = ['0-25Q', '25-50Q', '50-75Q', '75-100Q']
fcc_survey_df['Income_quantile_range'] = pd.qcut(

fcc_survey_df['Income'],
                                q=quantile_list)
fcc_survey_df['Income_quantile_label'] = pd.qcut(

fcc_survey_df['Income'],
                                q=quantile_list,

labels=quantile_labels)

fcc_survey_df[['ID.x', 'Age', 'Income', 'Income_quantile_range',
                'Income_quantile_label']].iloc[4:9]

```

	ID.x	Age	Income	Income_quantile_range	Income_quantile_label
4	9368291c93d5d5f5c8cdb1a575e18bec	20.0	6000.0	(5999.999, 20000.0]	0-25Q
5	dd0e77eab9270e4b67c19b0d6bbf621b	34.0	40000.0	(37000.0, 60000.0]	50-75Q
6	7599c0aa0419b59fd11ffede98a3665d	23.0	32000.0	(20000.0, 37000.0]	25-50Q
7	6dff182db452487f07a47596f314bddc	35.0	40000.0	(37000.0, 60000.0]	50-75Q
8	9dc233f8ed1c6eb2432672ab4bb39249	33.0	80000.0	(60000.0, 200000.0]	75-100Q

Quantile based bin ranges and labels for developer incomes

This should give you a good idea of how quantile based adaptive binning works. An important point to remember here is that the resultant outcome of binning leads to discrete valued categorical features and you might need an additional step of feature engineering on the categorical data before using it in any model. We will cover feature engineering strategies for categorical data shortly in the next part!

Data set - $X = \{1, 1, 1, 1, 1, 2, 2, 11, 11, 12, 12, 44\}$ and $k=3$

equal width binning-

Low: $0 \leq X < 15$, which contains all the data values except one

Medium: $15 \leq X < 30$, which contains no data values at all

High: $30 \leq X < 45$, which contains a single outlier

equal frequency binning –

we have $n=12$, $k=3$, and $n/k=4$

Low: Contains the first four data values, all $X=1$.

Medium: Contains the next four data values, $\{1, 2, 2, 11\}$.

High: Contains the last four data values, $\{11, 12, 12, 44\}$.

k-means - seems to be the correct partition

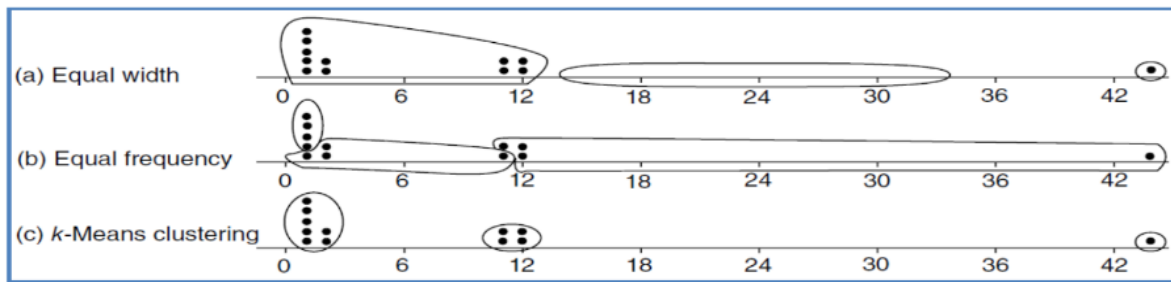


Illustration of binning methods.

Log Transform:

QUE :Discuss the power of transformation. Explain the effect of logarithmic transformation with example.

The log transform belongs to the power transform family of functions. This function can be mathematically represented as

$$y = \log_b(x)$$

which reads as *log* of x to the base b is equal to y . This can then be translated into

$$b^y = x$$

which indicates as to what power must the base b be raised to in order to get x . The natural logarithm uses $b=e$ where $e = 2.71828$ popularly known as Euler's number. You can also use base $b=10$ used popularly in the decimal system.

Log transforms are useful when applied to skewed distributions as they tend to expand the values which fall in the range of lower magnitudes and tend to compress or reduce the values which fall in the range of higher magnitudes. This tends to make the skewed distribution as normal-like as possible. Let's use log transform on our developer Income feature which we used earlier.

```
fcc_survey_df['Income_log'] = np.log((1+ fcc_survey_df['Income']))
fcc_survey_df[['ID.x', 'Age', 'Income', 'Income_log']].iloc[4:9]
```

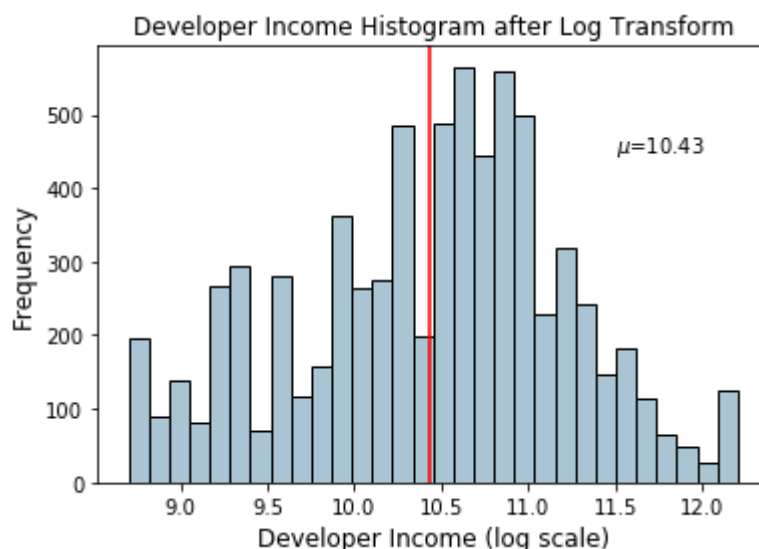
	ID.x	Age	Income	Income_log
4	9368291c93d5d5f5c8cdb1a575e18bec	20.0	6000.0	8.699681
5	dd0e77eab9270e4b67c19b0d6bbf621b	34.0	40000.0	10.596660
6	7599c0aa0419b59fd11ffede98a3665d	23.0	32000.0	10.373522
7	6dff182db452487f07a47596f314bddc	35.0	40000.0	10.596660
8	9dc233f8ed1c6eb2432672ab4bb39249	33.0	80000.0	11.289794

Log transform on developer income

The `Income_log` field depicts the transformed feature after log transformation.

Let's look at the data distribution on this transformed field now.

```
income_log_mean = np.round(np.mean(fcc_survey_df['Income_log']), 2)
fig, ax = plt.subplots()
fcc_survey_df['Income_log'].hist(bins=30, color='#A9C5D3',
                                  edgecolor='black', grid=False)
plt.axvline(income_log_mean, color='r')
ax.set_title('Developer Income Histogram after Log Transform',
             fontsize=12)
ax.set_xlabel('Developer Income (log scale)', fontsize=12)
ax.set_ylabel('Frequency', fontsize=12)
ax.text(11.5, 450, r'$\mu$='+str(income_log_mean), fontsize=10)
```

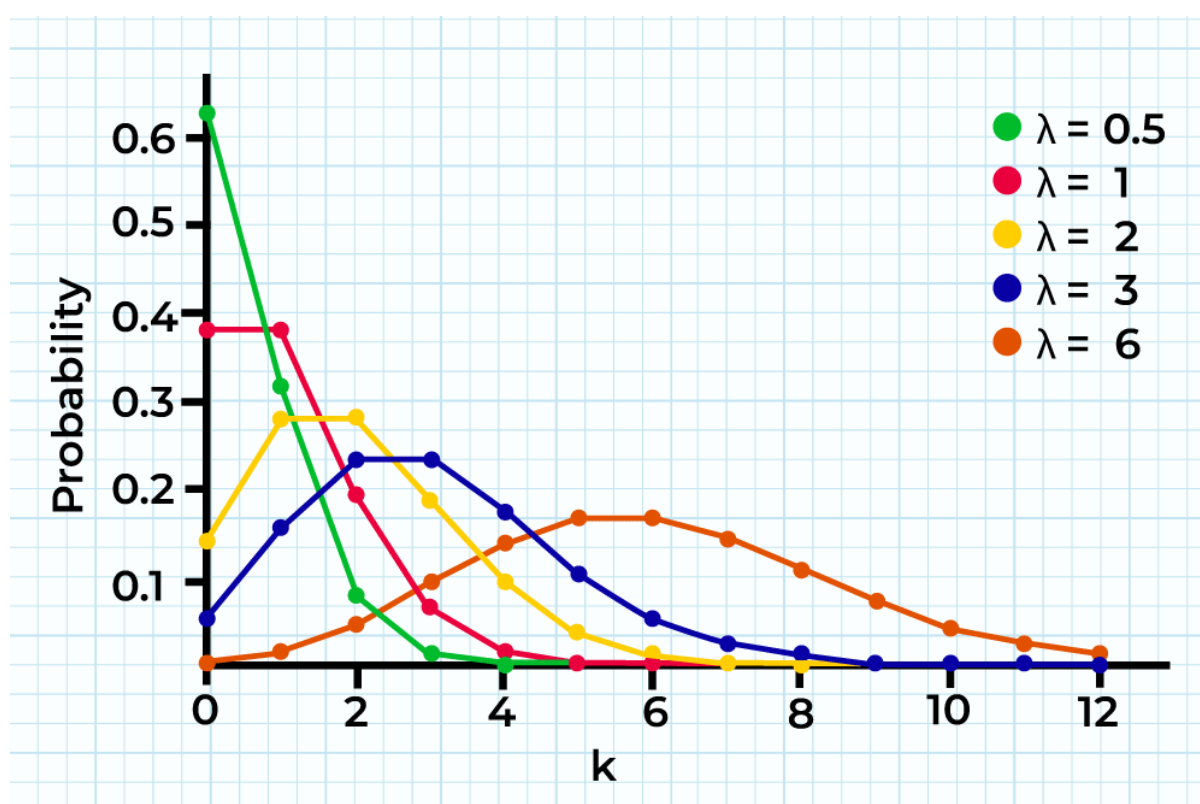


Histogram depicting developer income distribution after log transform

Based on the above plot, we can clearly see that the distribution is more normal-like or gaussian as compared to the skewed distribution on the original data.

Power Transforms: Generalization of the Log Transform

The log transform is a specific example of a family of transformations known as power transforms. In statistical terms, these are variance-stabilizing transformations. To understand why variance stabilization is good, consider the Poisson distribution. This is a heavy-tailed distribution with a variance that is equal to its mean: hence, the larger its center of mass, the larger its variance, and the heavier the tail. Power transforms change the distribution of the variable so that the variance is no longer dependent on the mean. For example, suppose a random variable X has the Poisson distribution. If we transform X by taking its square root, the variance of \sqrt{X} is roughly constant, instead of being equal to the mean.



A rough illustration of the Poisson distribution, an example distribution where the variance increases along with the mean

A simple generalization of both the square root transform and the log transform is known as the Box-Cox transform:

$$\tilde{x} = \begin{cases} \frac{x^\lambda - 1}{\lambda} & \text{if } \lambda \neq 0, \\ \ln(x) & \text{if } \lambda = 0. \end{cases}$$

Box-Cox Transform:

QUE :Explain Box Cox transformation. What is effect of Box Cox transformation?

The Box-Cox transform is another popular function belonging to the power transform family of functions. This function has a pre-requisite that the numeric values to be transformed must be positive (similar to what *log* transform expects). In case they are negative, shifting using a constant value helps. Mathematically, the Box-Cox transform function can be denoted as follows.

$$y = f(x, \lambda) = x^\lambda = \begin{cases} \frac{x^\lambda - 1}{\lambda} & \text{for } \lambda > 0 \\ \log_e(x) & \text{for } \lambda = 0 \end{cases}$$

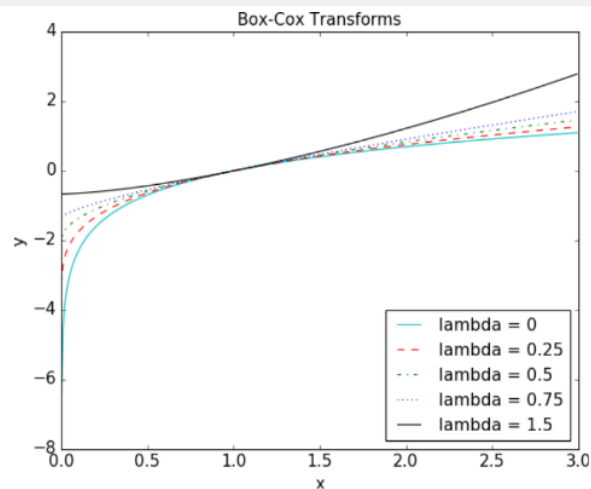
Such that the resulted transformed output y is a function of input x and the transformation parameter λ such that when $\lambda = 0$, the resultant transform is the natural *log* transform which we discussed earlier. The optimal value of λ is usually determined using a maximum likelihood or log-likelihood estimation. Let's now apply the Box-Cox transform on our developer income feature. First we get the optimal lambda value from the data distribution by removing the non-null values as follows.

```
income = np.array(fcc_survey_df['Income'])
income_clean = income[~np.isnan(income)]
l, opt_lambda = spstats.boxcox(income_clean)
print('Optimal lambda value:', opt_lambda)
```

Output

Optimal lambda value: 0.117991239456

- Box-Cox transform for $\lambda = 0$ (the log transform), $\lambda = 0.25$, $\lambda = 0.5$ (a scaled and shifted version of the square root transform), $\lambda = 0.75$, and $\lambda = 1.5$.
- Setting λ to be less than 1 compresses the higher values, and setting λ higher than 1 has the opposite effect.



Now that we have obtained the optimal λ value, let us use the Box-Cox transform for two values of λ such that $\lambda = 0$ and $\lambda = \lambda(\text{optimal})$ and transform the developer Income feature.

```
fcc_survey_df['Income_boxcox_lambda_0'] = spstats.boxcox(
    (1+fcc_survey_df['Income']),
    lambda=0)
fcc_survey_df['Income_boxcox_lambda_opt'] = spstats.boxcox(
    fcc_survey_df['Income'],
    lambda=opt_lambda)

fcc_survey_df[['ID.x', 'Age', 'Income', 'Income_log',
                'Income_boxcox_lambda_0',
                'Income_boxcox_lambda_opt']].iloc[4:9]
```

	ID.x	Age	Income	Income_log	Income_boxcox_lambda_0	Income_boxcox_lambda_opt
4	9368291c93d5d5f5c8cdb1a575e18bec	20.0	6000.0	8.699681	8.699681	15.181133
5	dd0e77eab9270e4b67c19b0d6bbf621b	34.0	40000.0	10.596660	10.596660	21.115429
6	7599c0aa0419b59fd11ffede98a3665d	23.0	32000.0	10.373522	10.373522	20.346526
7	6dff182db452487f07a47596f314bddc	35.0	40000.0	10.596660	10.596660	21.115429
8	9dc233f8ed1c6eb2432672ab4bb39249	33.0	80000.0	11.289794	11.289794	23.637178

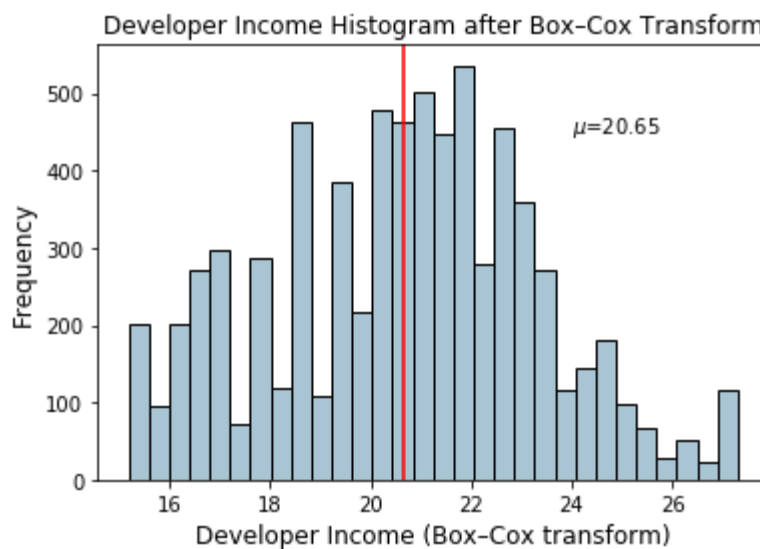
Developer income distribution after Box-Cox transform

The transformed features are depicted in the above data frame. Just like we expected, `Income_log` and `Income_boxcox_lambda_0` have the same values. Let's

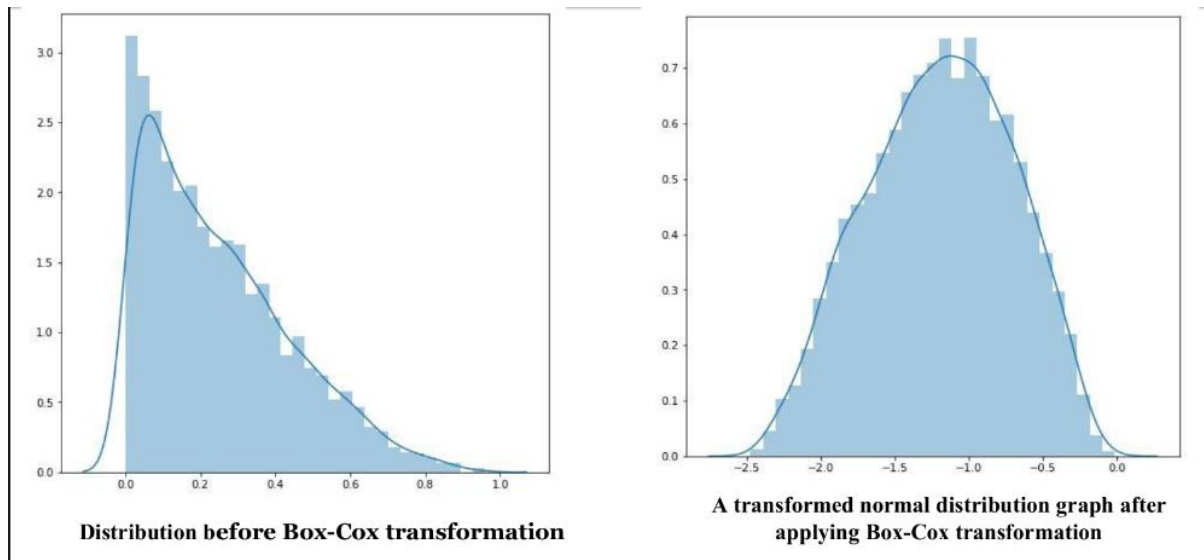
look at the distribution of the transformed Income feature after transforming with the optimal λ .

```
income_boxcox_mean = np.round(
    np.mean(

fcc_survey_df['Income_boxcox_lambda_opt']),2)fig, ax =
plt.subplots()
fcc_survey_df['Income_boxcox_lambda_opt'].hist(bins=30,
        color='#A9C5D3', edgecolor='black',
grid=False)
plt.axvline(income_boxcox_mean, color='r')
ax.set_title('Developer Income Histogram after Box-Cox
Transform',
            fontsize=12)
ax.set_xlabel('Developer Income (Box-Cox transform)',
            fontsize=12)
ax.set_ylabel('Frequency', fontsize=12)
ax.text(24, 450, r'$\mu$='+str(income_boxcox_mean), fontsize=10)
```



Histogram depicting developer income distribution after Box-Cox transform



Effect of Box-Cox Transformation:

1. **Normalization of Data:**

- The Box-Cox transformation is used to stabilize and normalize data that does not meet the assumptions of normality. It can make the distribution of the data more symmetrical and closer to a normal distribution.

2. **Stabilization of Variances:**

- It helps stabilize the variances across different levels or groups in the data. This is especially important in situations where the assumption of constant variance (homoscedasticity) is required for statistical tests or models.

3. **Handling Non-Constant Variances (Heteroscedasticity):**

- If the data exhibits heteroscedasticity (unequal variances at different levels), applying the Box-Cox transformation can often mitigate this issue, making the data more suitable for analysis.

4. **Parameterized Transformation:**

- The Box-Cox transformation is parameterized by the value of λ . Depending on the dataset, different values of λ can be chosen to achieve the desired transformation. For example, when $\lambda = 0$, the transformation is equivalent to taking the natural logarithm.

5. **Improved Interpretability:**

- The Box-Cox transformation can sometimes make the relationship between variables more interpretable. For example, in finance, it can help stabilize the variability of returns over time.

6. **Parameter Estimation:**

- The value of λ is typically estimated from the data using methods like maximum likelihood estimation. This allows the transformation to be tailored to the specific dataset.

7. **Improved Model Performance:**

- When the assumptions of normality and constant variance are met, applying the Box-Cox transformation can lead to more accurate and reliable statistical models.

8. **Caution with Interpretation:**

- While the Box-Cox transformation can be a powerful tool, it's important to remember that the transformed data should be interpreted in the context of the original data. Any conclusions drawn from the transformed data should be carefully considered.

Min-Max Scaling:

QUE :Describe min-max scaling in detail.

Let x be an individual feature value (i.e., a value of the feature in some data point), and $\min(x)$ and $\max(x)$, respectively, be the minimum and maximum values of this feature over the entire dataset. Min-max scaling squeezes (or stretches) all feature values to be within the range of $[0, 1]$. Figure 2-15 demonstrates this concept. The formula for min-max scaling is:

This method of scaling requires below two-step:

1. First, we are supposed to find the minimum and the maximum value of the column.
2. Then we will subtract the minimum value from the entry and divide the result by the difference between the maximum and the minimum value.

As we are using the maximum and the minimum value this method is also prone to outliers but the range in which the data will range after performing the above two steps is between 0 to 1.

$$\tilde{x} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

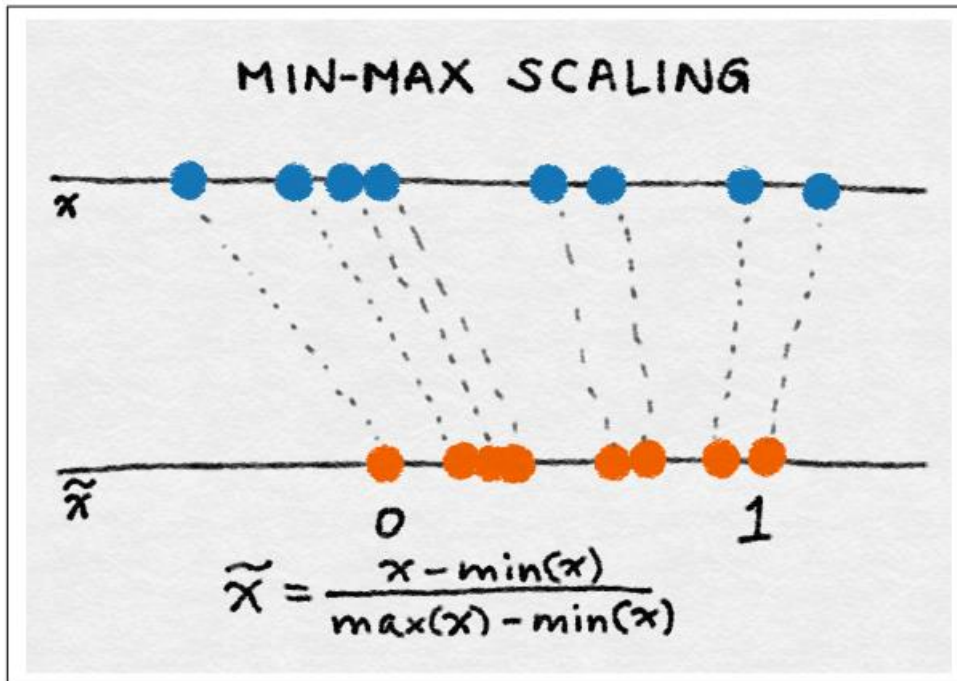


Figure 2-15. Illustration of min-max scaling

EXAMPLE :

```
from sklearn.preprocessing import MinMaxScaler
```

```
scaler = MinMaxScaler()
```

```
scaled_data = scaler.fit_transform(df)
```

```
scaled_df = pd.DataFrame(scaled_data,
```

```
                        columns=df.columns)
```

```
scaled_df.head()
```

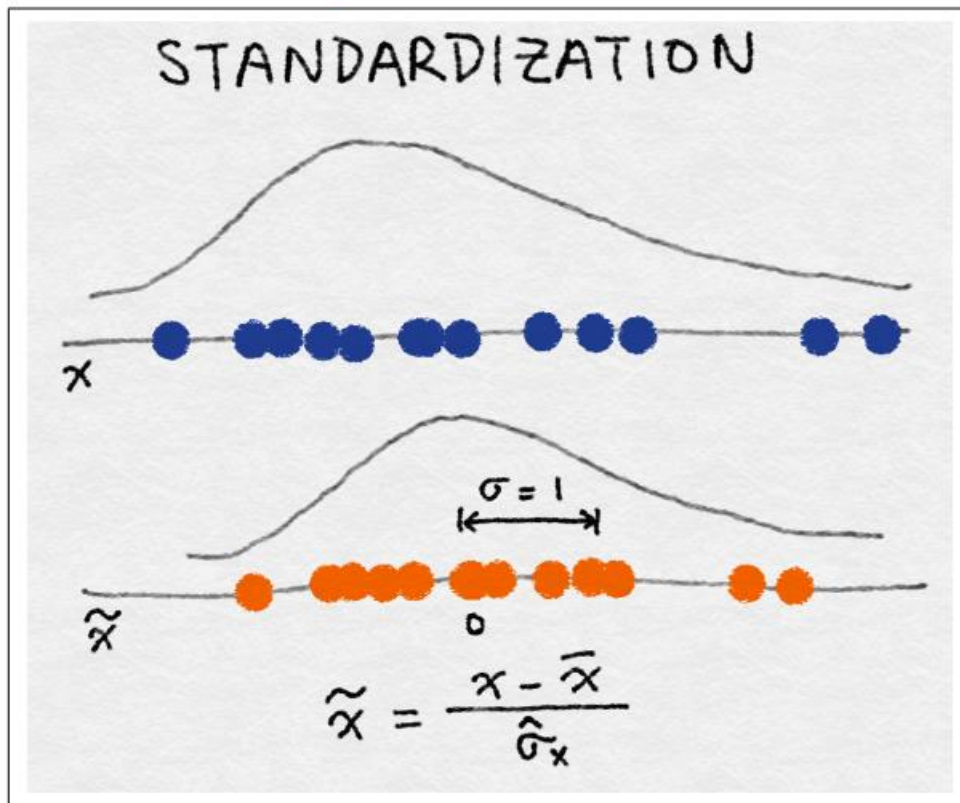
Standardization (Variance Scaling) :

Feature standardization is defined as:

$$\tilde{x} = \frac{x - \text{mean}(x)}{\text{sqrt}(\text{var}(x))}$$

It subtracts off the mean of the feature (over all data points) and divides by the variance. Hence, it can also be called variance scaling. The resulting scaled

feature has a mean of 0 and a variance of 1. If the original feature has a Gaussian distribution, then the scaled feature does too.



This method of scaling is basically based on the central tendencies and variance of the data.

1. First, we should calculate the mean and standard deviation of the data we would like to normalize.
2. Then we are supposed to subtract the mean value from each entry and then divide the result by the standard deviation.

This helps us achieve a normal distribution(if it is already normal but skewed) of the data with a mean equal to zero and a standard deviation equal to 1.

EXAMPLE

```
from sklearn.preprocessing import StandardScaler
```

```
scaler = StandardScaler()
```

```
scaled_data = scaler.fit_transform(df)
```

```
scaled_df = pd.DataFrame(scaled_data,
                          columns=df.columns)

print(scaled_df.head())
```

l2 normalization:

QUE :Discuss l2 normalization in detail.

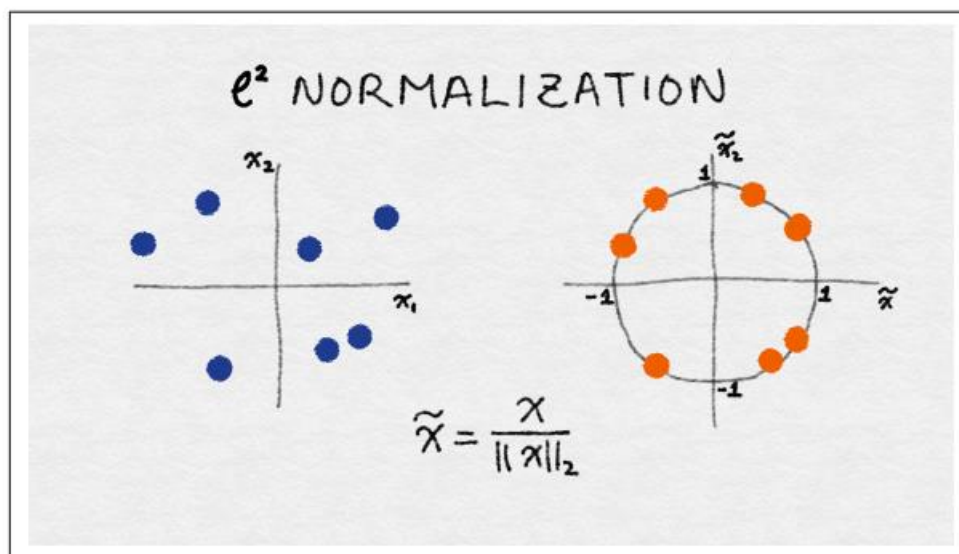
This technique normalizes (divides) the original feature value by what's known as the ℓ_2 norm, also known as the Euclidean norm. It's defined as follows:

$$\tilde{x} = \frac{x}{\|x\|_2}$$

The ℓ_2 norm measures the length of the vector in coordinate space. The definition can be derived from the well-known Pythagorean theorem that gives us the length of the hypotenuse of a right triangle given the lengths of the sides:

$$\|x\|_2 = \sqrt{x_1^2 + x_2^2 + \dots + x_m^2}$$

The ℓ_2 norm sums the squares of the values of the features across data points, then takes the square root. After ℓ_2 normalization, the feature column has norm 1. This is also sometimes called ℓ_2 scaling. (Loosely speaking, scaling means multiplying by a constant, whereas normalization could involve a number of operations.) Figure 2-17 illustrates ℓ_2 normalization.



Interaction Features :

QUE :Explain Interaction Features with suitable example. What are advantages of Interaction Features?

A simple pairwise interaction feature is the product of two features.

The analogy is the logical AND.

It expresses the outcome in terms of pairs of conditions: “the purchase is coming from zip code 98121” AND “the user’s age is between 18 and 35.”

linear models often find interaction features very helpful

A simple linear model uses a linear combination of the individual input features x_1, x_2, \dots, x_n to predict the outcome y :

$$y = w_1x_1 + w_2x_2 + \dots + w_nx_n$$

An easy way to extend the linear model is to include combinations of pairs of input features, like

$$y = w_1x_1 + w_2x_2 + \dots + w_nx_n + w_{1,1}x_1x_1 + w_{1,2}x_1x_2 + w_{1,3}x_1x_3 + \dots$$

Interaction features are very simple to formulate, but they are expensive to use.

The training and scoring time of a linear model with pairwise interaction features would go from $O(n)$ to $O(n^2)$, where n is the number of singleton features.

There are a few ways around the computational expense of higher-order interaction features. One could perform feature selection on top of all of the interaction features.

Feature selection employs computational means to select the best features for a problem.

Some feature selection techniques still require training multiple models with a large number of features.

Advantages of Interaction Features:

Capturing Non-Linear Relationships:

Interaction features allow the model to capture non-linear relationships between the original features and the target variable. This is important because many real-world relationships are not strictly linear.

Incorporating Domain Knowledge:

Domain experts often understand how different factors interact to influence outcomes. By creating interaction features, you can encode this domain knowledge directly into the model.

Improving Model Accuracy:

Interaction features can enhance the predictive power of a model. They provide additional information and context that can help the model make more accurate predictions.

Reducing Underfitting:

In situations where a model is too simplistic (underfit), interaction features can introduce complexity and allow the model to better fit the data.

Handling High-Dimensional Data:

When dealing with high-dimensional data, interactions can help reduce the "curse of dimensionality" by focusing the model's attention on meaningful combinations of features.

Enhancing Feature Importance Interpretation:

Interaction features can reveal which combinations of features are most important for prediction. This can provide valuable insights into the underlying relationships in the data.