Machine Learning

| Feature Scaling and Transformation

Prof. Dr. Mostafa Elhosseini

https://youtube.com/drmelhosseini

Agenda

- Feature Scaling and Transformation

Feature Scaling

- Feature scaling is one of the most crucial transformations you should perform on your data
- With few exceptions, Machine Learning algorithms don't perform well when the input numerical attributes have very different scales.
- The total number of rooms ranges from about 6 to 39,320, while the median incomes only range from 0 to 15. Without any scaling, most models will be biased toward ignoring the median income and focusing more on the number of rooms.

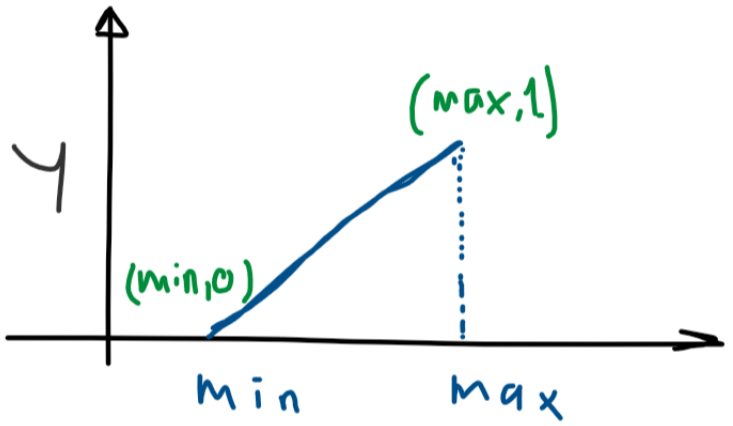
Feature Scaling

- There are two common ways to get all attributes to have the same scale: and
  - min-max scaling (Normalization)
  - standardization

Feature Scaling

min-max scaling

- Min-max scaling (**normalization**) is quite simple: values are shifted and rescaled so that they end up ranging from 0 to 1
- This is performed by subtracting the min value and dividing by the difference between the min and the max
- Scikit-Learn provides a transformer called MinMaxScaler



$$\frac{y-0}{x-min} = \frac{1-0}{max-min}$$

$$y = \frac{x-min}{max-min}$$

Feature Scaling

min-max scaling

- It has a **feature_range** hyperparameter that lets you change the range if, for some reason, you don't want 0 –1 (e.g., neural networks work best with zero-mean inputs, so a range of –1 to 1 is preferable)

```
from sklearn.preprocessing import MinMaxScaler

min_max_scaler = MinMaxScaler(feature_range=(-1, 1))
housing_num_min_max_scaled = min_max_scaler.fit_transform(housing_num)
```

- The **effect of the outlier** in min-max scaling is **evident**, as it **compresses** most of the dataset into a small range, making it difficult to **distinguish** between the original values based on their scaled versions.

Feature Scaling

Standardization

- Standardization is quite different: first it subtracts the mean value (so standardized values always have a zero mean), and then it divides by the standard deviation so that the resulting distribution has unit variance.
- $standardized\ value = \frac{value - mean}{Std\ Deviation}$

Feature Scaling

Standardization

- While the outlier still affects the standardized values, the impact is less severe. The data points remain more spread out, preserving more of their relative differences. The standardized values without the outlier are more varied, indicating that standardization maintains a better distinction among values in the presence of outliers.
- suppose a district has a median income equal to 100 (by mistake), instead of the usual 0–15. Min-max scaling to the 0–1 range would map this outlier down to 1 and it would crush all the other values down to 0–0.15, whereas standardization would not be much affected.

Feature Scaling

Standardization

- Scikit-Learn provides a transformer called StandardScaler for standardization

```
from sklearn.preprocessing import StandardScaler

std_scaler = StandardScaler()
housing_num_std_scaled = std_scaler.fit_transform(housing_num)
```

Heavy tail

- A heavy tail refers to a distribution **where the tails** (the ends of the distribution, far from the mean) **are not exponentially rare** but instead have a significant probability of occurring.
- Heavy-tailed distributions have more extreme values (outliers) than would be expected in a normal (Gaussian) distribution.
- These extreme values can significantly impact the mean and variance of the distribution and can pose challenges in statistical modeling and risk assessment because they imply a higher likelihood of extreme events.

Heavy tail

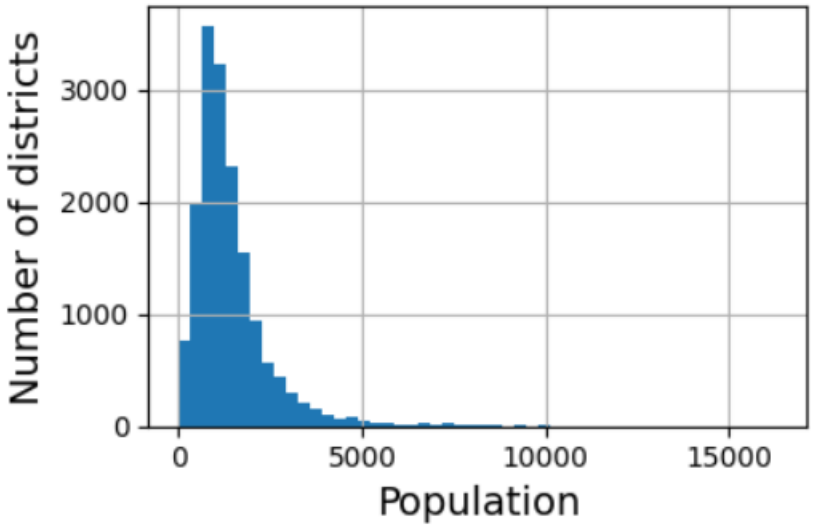
- When a feature's distribution has a heavy tail, both min-max scaling and standardization will **squash most values into a small range**.
- Machine learning models generally don't like this at all, so before you scale the feature, you should **first transform it** to shrink the heavy tail, and if possible to make the distribution roughly symmetrical.
- A **common way** to do this for **positive features with a heavy tail** to the right is to replace the feature with its **square root** (or raise the feature to a power between 0 and 1)

Heavy tail

- A **common way** to do this for **positive features with a heavy tail** to the right is to replace the feature with its **square root** (or raise the feature to a power between 0 and 1).
- If the feature has a really long and heavy tail, such as a power law distribution, then replacing the feature with its **logarithm may help**.

Heavy tail

- For example, the **population feature** roughly follows a **power law**.
- A power law is a functional relationship between two quantities, where one quantity varies as a power of another
- $P(x) \propto x^{-\alpha}$
- $P(x)$ is the frequency of an event at size x
- α is a positive constant



Heavy tail

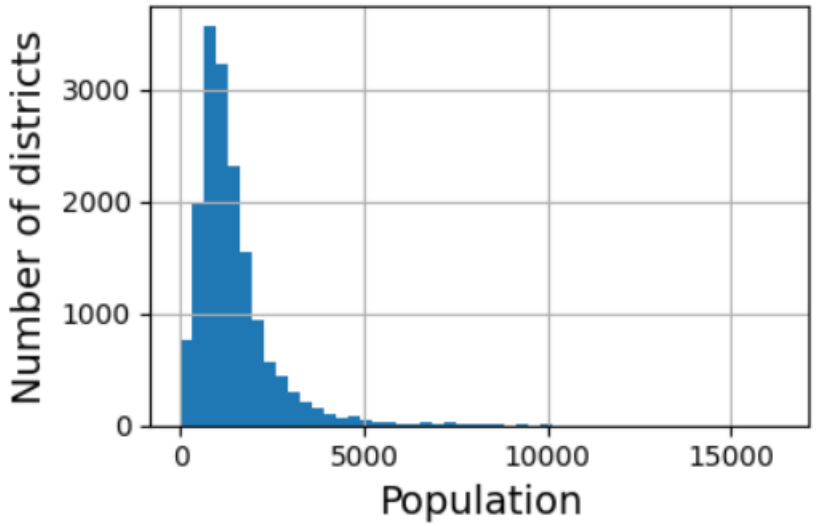
- The distribution has a **heavy or fat tail**, indicating that events much larger than the average are more common than they would be in, for example, a normal distribution
- Power laws describe a wide range of phenomena in nature and human activities, including city populations, earthquake magnitudes, and internet file sizes

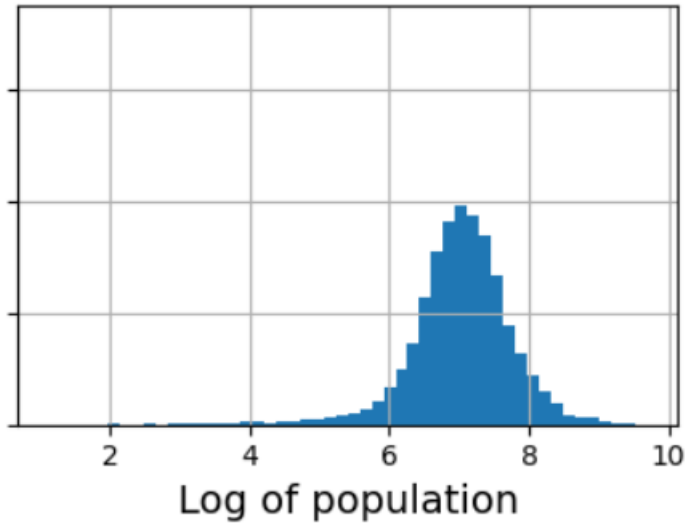
Heavy tail

- Each of these follows a pattern where there are a few large or frequent occurrences and many more small or infrequent ones
- They imply that small events are extremely common, whereas large events are rare but not as rare as one would expect with an exponential decay.
- For example, in city sizes, a few cities are very large (like New York or Tokyo), but there are many more small cities and towns

Heavy tail

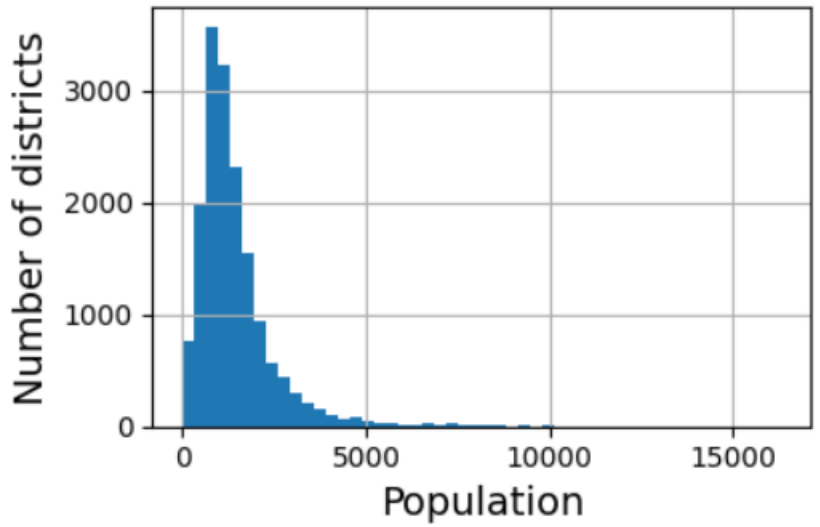
- For example, the **population feature** roughly follows a **power law**: districts with 10,000 inhabitants are only 10 times less frequent than districts with 1,000 inhabitants, not exponentially less frequent
- Compute its **log**: it's very close to a Gaussian distribution (i.e., bell-shaped).

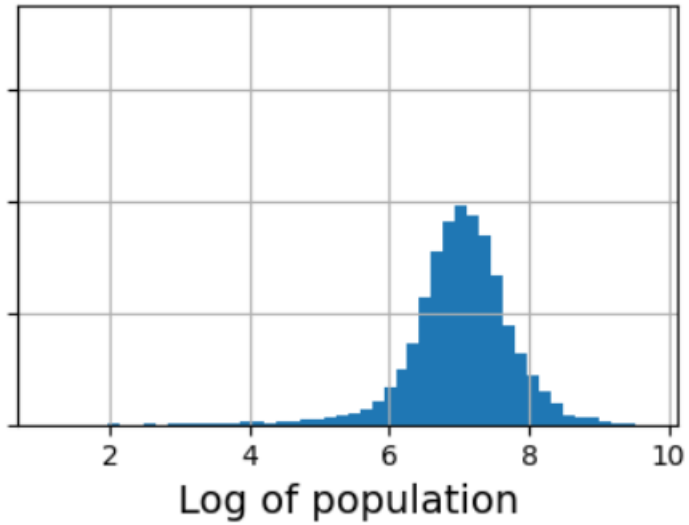




Heavy tail

```
# extra code – this cell generates Figure 2–17
fig, axs = plt.subplots(1, 2, figsize=(8, 3), sharey=True)
housing["population"].hist(ax=axs[0], bins=50)
housing["population"].apply(np.log).hist(ax=axs[1], bins=50)
axs[0].set_xlabel("Population")
axs[1].set_xlabel("Log of population")
axs[0].set_ylabel("Number of districts")
save_fig("long_tail_plot")
plt.show()
```





Bucketizing

- Another approach to handle heavy-tailed features consists in bucketizing the feature.
- Dividing the range of the feature into a set number of intervals (or buckets) and then replacing each value with the bucket index it falls into.
- you could replace each value with its **percentile**. Bucketizing with equal-sized buckets results in a feature with an almost **uniform distribution**, so there's **no need for further scaling**, or you can just **divide** by the number of buckets to force the values to the **0 –1 range**.

Multimodal distribution

- Many machine learning algorithms are based on statistical assumptions that are best met by unimodal distributions, typically Gaussian.
- In a multimodal distribution, the presence of multiple peaks might lead to a situation where **one mode dominates the feature**, skewing the learning process and potentially causing the algorithm to **miss patterns associated with the less dominant modes**.
- For clustering algorithms, such as K-means, or for classification tasks, the presence of multiple modes within a single feature can lead to misclassification or poor cluster assignment if the algorithm interprets each mode as a separate class or cluster.

Multimodal distribution

- Algorithms that use gradient descent for optimization can struggle with multimodal distributions because the multiple peaks can create local minima, making it harder for the algorithm to find the global minimum.
- Machine learning models aim to generalize from the training data to unseen data. Multimodal distributions might reflect complex underlying structures or multiple subpopulations within the data, which can be challenging for a model to generalize if not properly addressed during the training.
- Features with multimodal distributions can be harder to interpret, as the different modes might imply different behaviors or characteristics of the data points. Without transforming these features, it might be difficult to understand their influence on the prediction of the model.

Multimodal distribution

consider the weight distribution:

- Professional athletes: We have 100 athletes with weights ranging from 200 to 220 pounds due to muscle mass.
- Office workers: We have 100 office workers with weights ranging from 120 to 140 pounds.
- Here's what might happen with this multimodal distribution:
- **Skewed Average**: The average weight of the community would be around 170 pounds, which does not represent either subpopulation accurately.
- **Misleading Range**: The range of weights is from 120 to 220 pounds, which suggests high variance and could lead to assuming that individuals who weigh around 170 pounds are "average", when in reality, very few people in this community actually weigh that amount.

Multimodal distribution

consider the weight distribution:

- Clustering Issue: If we apply a clustering algorithm like K-means with k=2, aiming to categorize into 'lighter' and 'heavier' individuals, it might not work well because the means of the two clusters would not align well with the centers of the two actual subpopulations.
- Classification Problem: If we're trying to classify people into high risk and low risk of heart disease based on weight, the algorithm might get confused because the weights don't follow a single trend but rather have two peaks (multimodal). It may not be clear how to draw a boundary between high risk and low risk.

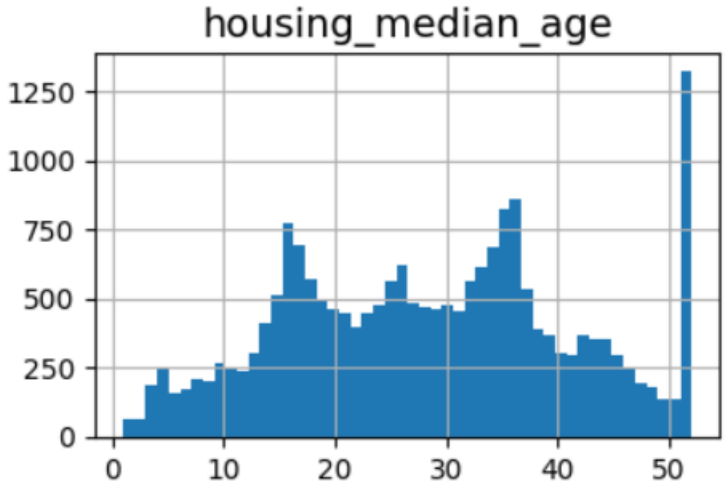
Multimodal distribution

consider the weight distribution:

- Feature Scaling: When we scale this feature for machine learning, if we use standard scaling methods like min-max scaling, both subpopulations are squashed into the same range, potentially losing the distinction between the two different groups.

Bucketizing

- When a feature has a **multimodal distribution** (i.e., with two or more clear peaks, called modes), such as the **housing_median_age** feature, it can also be helpful to bucketize it, but this time **treating the bucket IDs as categories**, rather than as numerical values.



Bucketizing

- This means that the bucket indices must be encoded, for example using a **OneHotEncoder** (so you usually don't want to use too many buckets). This approach will allow the regression model to more easily learn different rules for different ranges of this feature value.

Multimodal distributions

- Another approach to transforming multimodal distributions is to add a feature for each of the modes (at least the main ones), representing the similarity between the housing median age and that particular mode.
- The similarity measure is typically computed using a **radial basis function** (RBF)—any function that depends only on the distance between the input value and a fixed point. The most commonly used RBF is the **Gaussian RBF**, whose output value decays exponentially as the input value moves away from the fixed point

Multimodal distributions

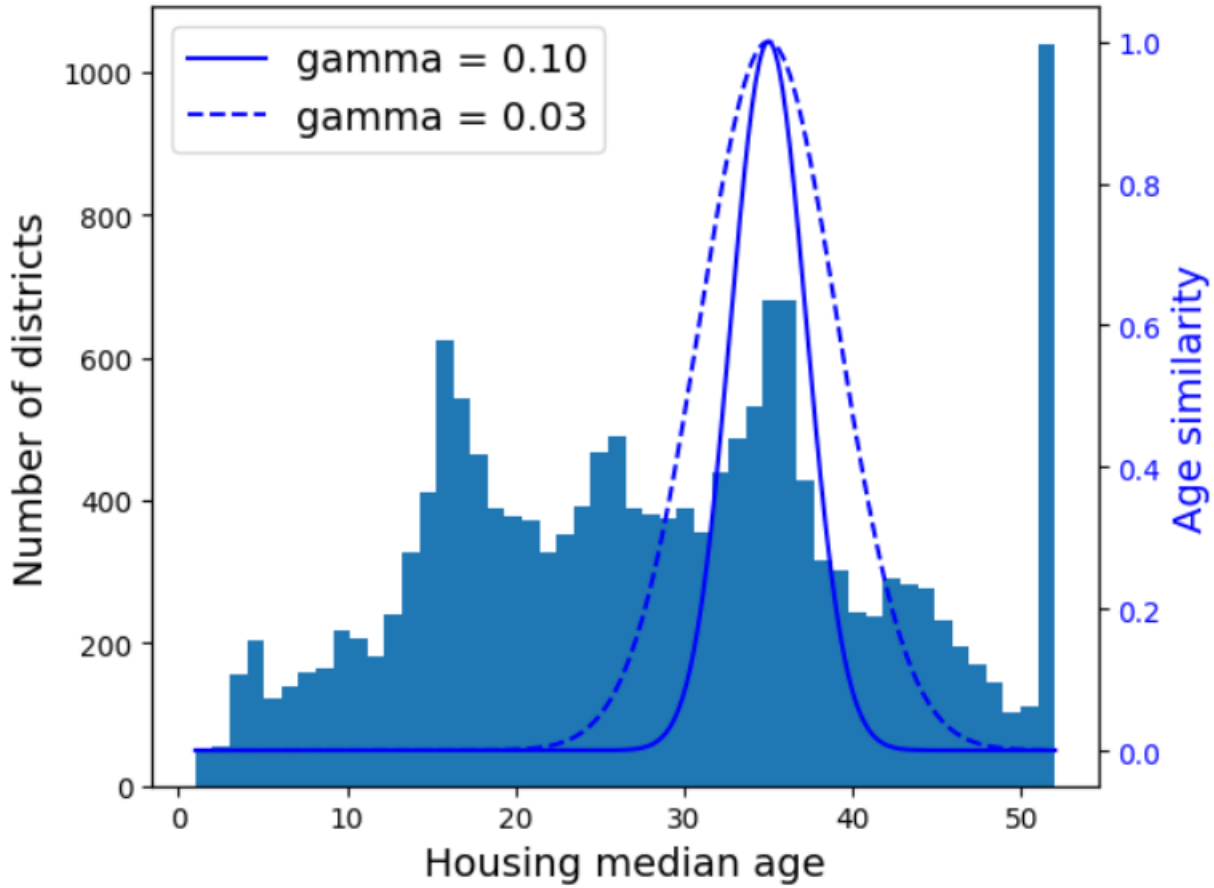
- For example, the Gaussian RBF similarity between the housing age x and 35 is given by the equation $\exp(-\gamma(x-35)^2)$.
- The hyperparameter γ determines how quickly the similarity measure decays as x moves away from 35.

```
from sklearn.metrics.pairwise import rbf_kernel

age_simil_35 = rbf_kernel(housing[["housing_median_age"]], [[35]], gamma=0.1)
```

Multimodal distributions

- Figure shows this new feature as a function of the housing median age (solid line). It also shows what the feature would look like if you used a smaller gamma value. As the chart shows, the new age similarity feature peaks at 35, right around the spike in the housing median age distribution: if this particular age group is well correlated with lower prices, there's a good chance that this new feature will help.



Multimodal distributions

- So far we've only looked at the input features, but the **target values** may also need to be transformed.
- Most of Scikit-Learn's transformers have an inverse_transform() method, making it easy to compute the inverse of their transformations.

```
from sklearn.linear_model import LinearRegression

target_scaler = StandardScaler()
scaled_labels = target_scaler.fit_transform(housing_labels.to_frame())

model = LinearRegression()
model.fit(housing[["median_income"]], scaled_labels)
some_new_data = housing[["median_income"]].iloc[:5]  # pretend this is new data

scaled_predictions = model.predict(some_new_data)
predictions = target_scaler.inverse_transform(scaled_predictions)
```

Multimodal distributions

- So far we've only looked at the input features, but the **target values** may also need to be transformed.
- Most of Scikit-Learn's transformers have an inverse_transform() method, making it easy to compute the inverse of their transformations.

```
from sklearn.linear_model import LinearRegression

target_scaler = StandardScaler()
scaled_labels = target_scaler.fit_transform(housing_labels.to_frame())

model = LinearRegression()
model.fit(housing[["median_income"]], scaled_labels)
some_new_data = housing[["median_income"]].iloc[:5]  # pretend this is new data

scaled_predictions = model.predict(some_new_data)
predictions = target_scaler.inverse_transform(scaled_predictions)
```

```
predictions
```

```
array([[131997.15275877],
       [299359.35844434],
       [146023.37185694],
       [138840.33653057],
       [192016.61557639]])
```

Multimodal distributions

- But a simpler option is to use a TransformedTargetRegressor

```
from sklearn.compose import TransformedTargetRegressor

model = TransformedTargetRegressor(LinearRegression(),
                                   transformer=StandardScaler())
model.fit(housing[["median_income"]], housing_labels)
predictions = model.predict(some_new_data)
```

```
predictions
```

```
array([131997.15275877, 299359.35844434, 146023.37185694, 138840.33653057,
       192016.61557639])
```