Data Preprocessing

Why Is Data Preprocessing Important?

- Quality decisions must be based on quality data
  - No quality data, no quality mining results!
    - e.g., duplicate or missing data may cause incorrect or even misleading statistics.

- Data preparation, cleaning, and transformation comprises the majority of the work in a data mining (machine learning) application (80-90%).

Data Preprocessing

- Machine learning models make a lot of assumptions about the data
- In reality, these assumptions are often violated
- We build *pipelines* that *transform* the data before feeding it to the learners
  - Normalization/Scaling (or other numeric transformations)
  - Discretization
  - Encoding (convert categorical features into numerical ones)
  - Handling missing data
  - Handling imbalanced data
  - Feature engineering (e.g. binning, polynomial features,...)
  - Pipeline
- Seek the best combinations of transformations and learning methods
  - Often done empirically, using cross-validation

Scaling/Normalization

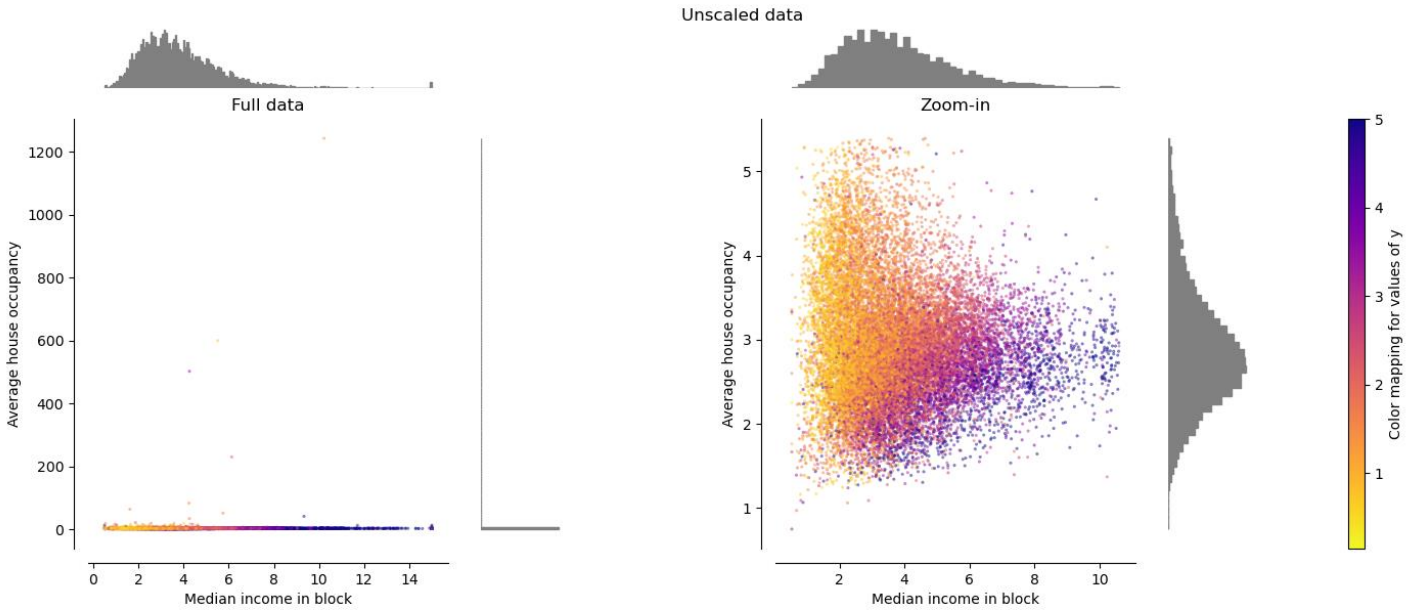
Scale vs Standardize

- Scale generally means to change the range of the values.
  - The shape of the distribution doesn't change.
  - The range is often set at 0 to 1.
- Standardize generally means changing the values so that the distribution's standard deviation equals one.
  - Scaling is often implied.

- Many machine learning algorithms perform better or converge faster when features are on a relatively similar scale and/or close to normally distributed.
- Examples of such algorithm families include:
  - linear and logistic regression
  - nearest neighbors
  - neural networks
  - support vector machines
  - principal components analysis
  - etc

Scaling

- Use when different numeric features have different range of value
  - Features with much higher values may overpower the others
- Goal: bring them all within the same range
- Why do we need scaling?
  - KNN: Distances depend mainly on feature with larger values
  - SVMs: are also based on distances
  - Linear model: Feature scale affects prediction

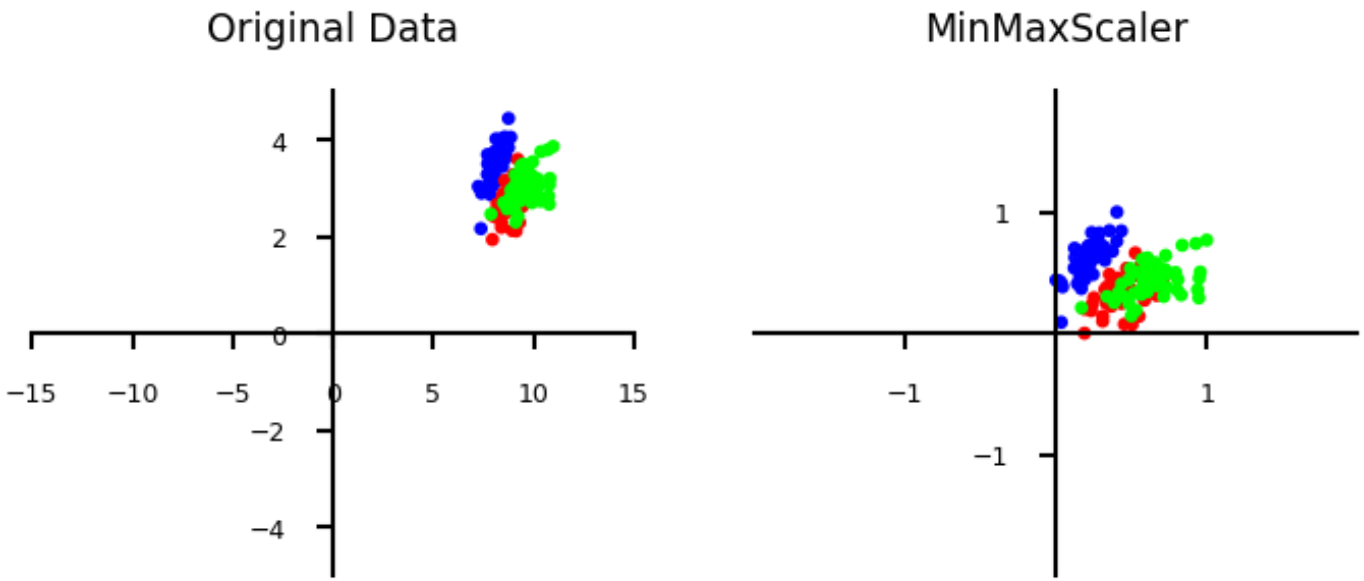


Min-Max Scaling

- Scales all features between a given min and max value (e.g., 0 and 1)
- Makes sense if min/max values have meaning in your data
- Sensitive to outliers

$$X_{new} = \frac{X - X_{min}}{X_{max} - X_{min}}(max - min) + min$$

- Ex. Let 'income' range \$12,000 to \$98,000 normalized to [0, 1]. Then \$73,000 is mapped to $\frac{73{,}600-12{,}000}{98{,}000-12{,}000}(1.0 - 0) + 0 = 0.716$
- Sensitive to outliers

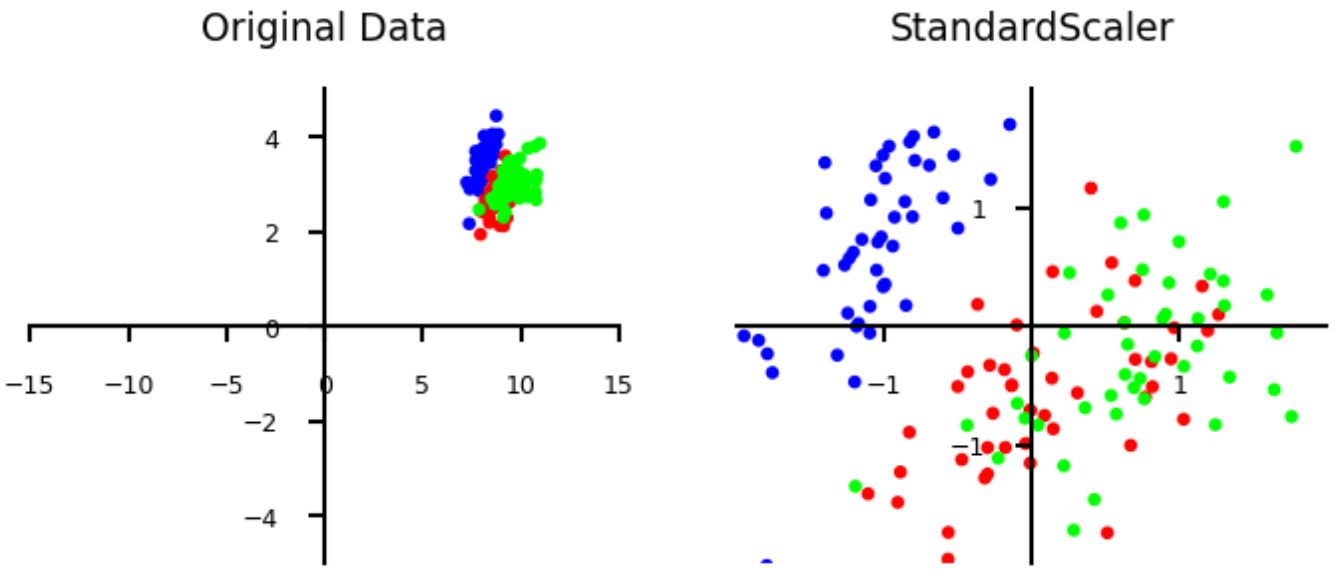


Standard Scaling (Z-score normalization)

- Standardizes a feature by subtracting the mean and then scaling to unit variance.
- Unit variance means dividing all the values by the standard deviation.
- Per feature, subtract the mean value μ, scale by standard deviation σ
  - New feature has $\mu = 0$ and $\sigma = 1$

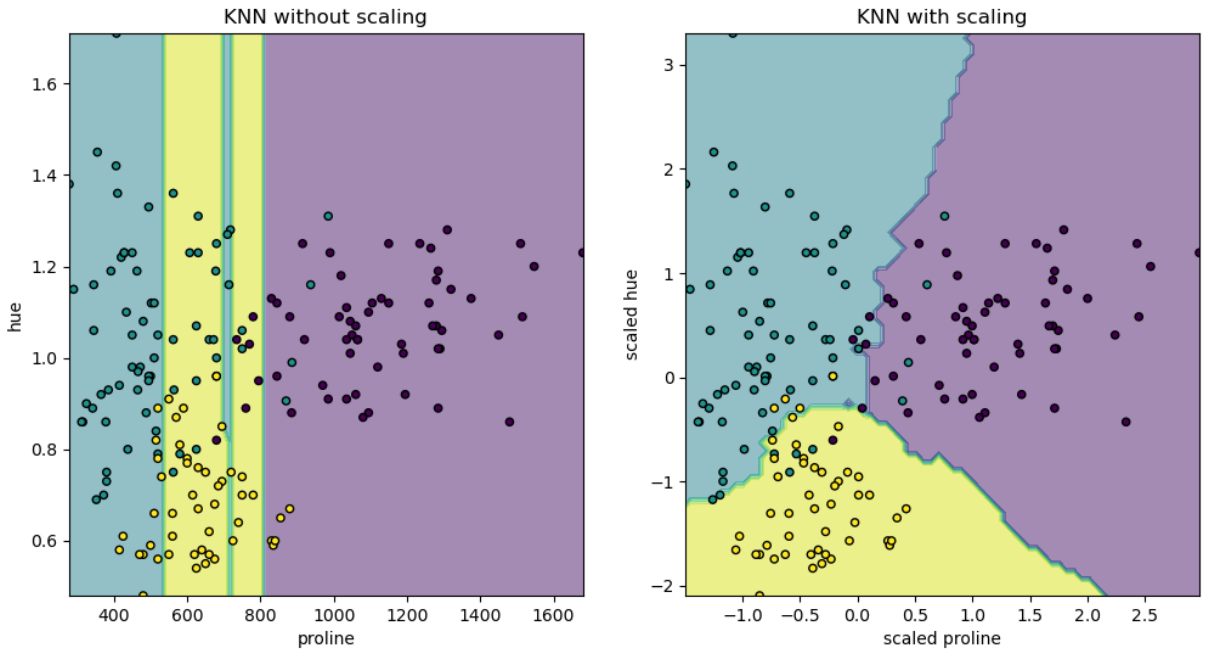
$$X_{new} = \frac{X - \mu}{\sigma}$$

- Ex. Let $\mu = 54{,}000$, $\sigma = 16{,}000$. Then $\frac{73{,}000-54{,}000}{16{,}000} = 1.225$



Effect of Scaling

- Here the decision boundary shows that fitting scaled or non-scaled data lead to completely different models.
- The reason is that the variable “proline” has values which vary between 0 and 1,000; whereas the variable “hue” varies between 1 and 10.
- Because of this, distances between samples are mostly impacted by the values of “proline”, while values of the “hue” will be comparatively ignored.
- If one uses StandardScaler to normalize this database, both scaled values lay approximately between -3 and 3 and the neighbors structure will be impacted more or less equivalently by both variables.



Scaling in sklearn

```
# import module
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler

data = [[11, 2], [3, 7], [0, 10], [11, 8]]

scaler = MinMaxScaler()
model=scaler.fit(data)
scaled_data=model.transform(data)

# print scaled data
print(scaled_data)
```

```
[[1.     0.    ]
[0.2727 0.625 ]
[0.     1.    ]
[1.     0.75  ]]
```

```
scaler = StandardScaler()
model = scaler.fit(data)
scaled_data = model.transform(data)

# print scaled data
print(scaled_data)
```

```
[[ 0.9759 -1.6115]
 [-0.6677  0.0848]
 [-1.2841  1.1026]
 [ 0.9759  0.4240]]
```

Scaling in sklearn

```
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler

dataset = load_iris()

# Splitting the independent and dependent variables
i_data = dataset.data
response = dataset.target

scaler= StandardScaler()
scaled = scaler.fit_transform(i_data)
print(scaled)
```

Discretization

Discretization

- Three types of attributes:
  - continuous — real numbers
  - nominal(categorical) — values from an unordered set
  - ordinal — values from an ordered set
    - e.g., grade={A, B, C, D, F}

- Discretization:
  - divide the range of a ***continuous*** attribute into intervals
  - reduce the number of values for a given continuous attribute
  - interval labels can then be used to replace actual data values

- Some techniques:
  - Unsupervised methods: Binning methods
    – equal-width, equal-frequency, k means
  - Supervised methods:
    – entropy-based, chi-square

Unsupervised Discretization Methods

- Equal-width (distance) partitioning
  - Divides the range into *N* intervals of equal size: uniform grid
  - if *A* and *B* are the lowest and highest values of the attribute, the width of intervals will be: $W = (B - A)/N$.
  - The most straightforward, but outliers may dominate presentation
  - Skewed data is not handled well

- Equal-frequency (quantile) partitioning
  - Divides the range into *N* intervals, each containing approximately same number of samples
  - Good data scaling

- Clustering discretization
  - k means

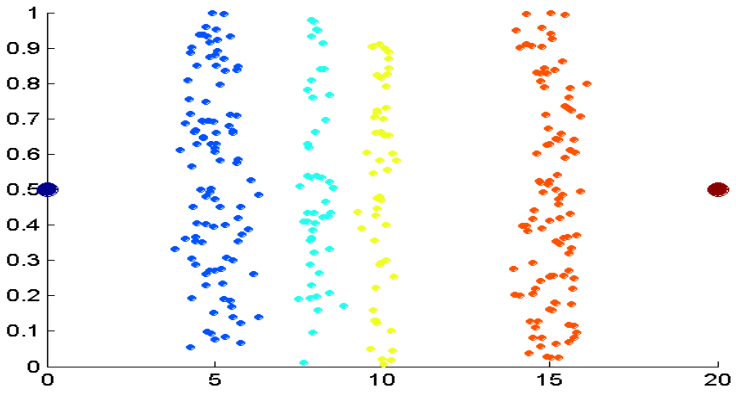
Equal Width/Frequency Methods

- Attribute values (for one attribute e.g., age):
  - 0, 4, 12, 16, 16, 18, 24, 26, 28
  - (# of bins = 3)
- Equal-width (uniform) binning
  - Width=(28–0)/3≈10
  - Bin 1: 0, 4 [-,10) bin
  - Bin 2: 12, 16, 16, 18 [10,20) bin
  - Bin 3: 24, 26, 28 [20,+) bin
  - – denote negative infinity, + positive infinity
- Equal-frequency (quantile) binning
  - each bin contains roughly 3 values (9/3=3)
  - Bin 1: 0, 4, 12 [-, 14) bin
  - Bin 2: 16, 16, 18 [14, 21) bin
  - Bin 3: 24, 26, 28 [21,+] bin

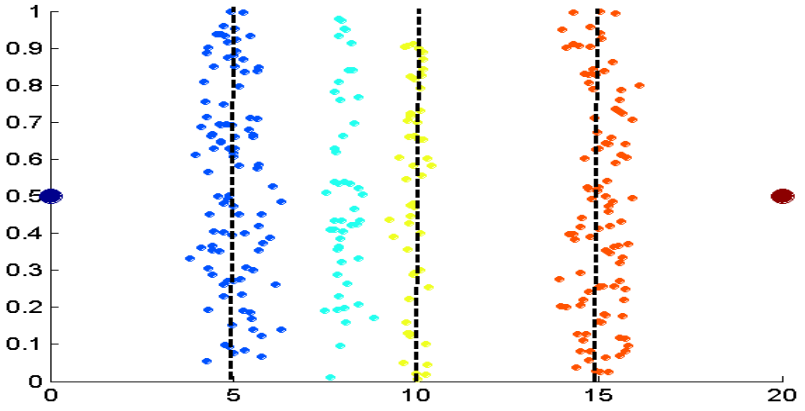
K means Discretization

- Use clustering methods in discretization
- A K-means discretization transform will attempt to fit k clusters for each input variable and then assign each observation to a cluster.
- Unless the empirical distribution of the variable is complex, the number of clusters is likely to be small, such as 3-to-5.
- In sklearn, we can apply the K-means discretization transform using the KBinsDiscretizer class and setting the “strategy” argument to “kmeans.” We must also set the desired number of bins set via the “n_bins” argument; in this case, we will use three.
- Once defined, we can call the fit_transform() function and pass it to our dataset to create a quantile transformed version of our dataset.

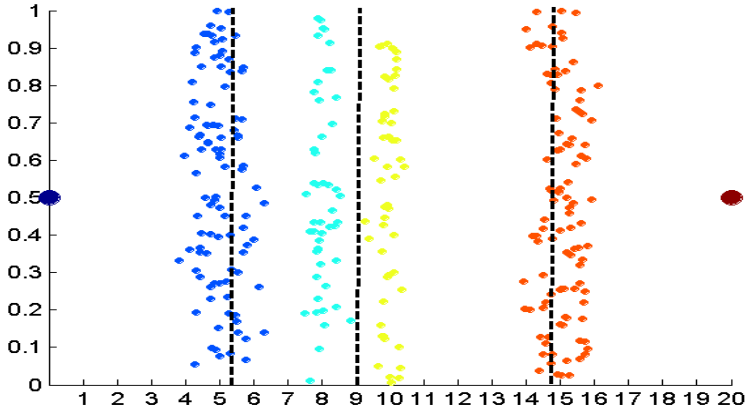
Using Equal Binning vs. Clustering



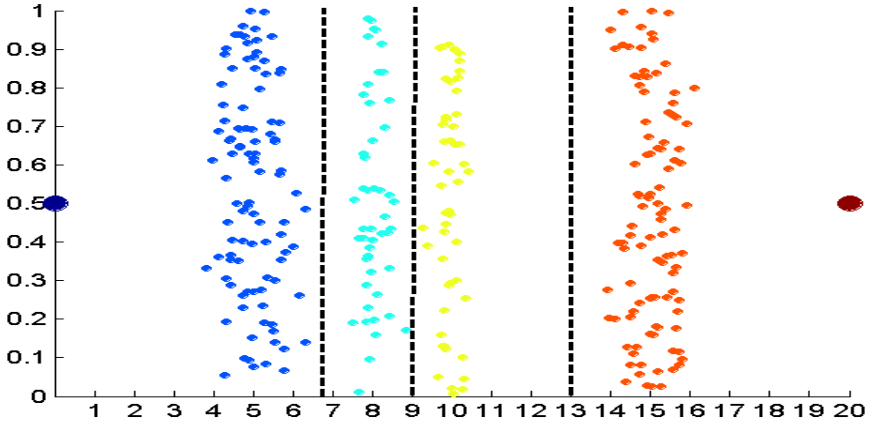
Data



Equal width (binning)

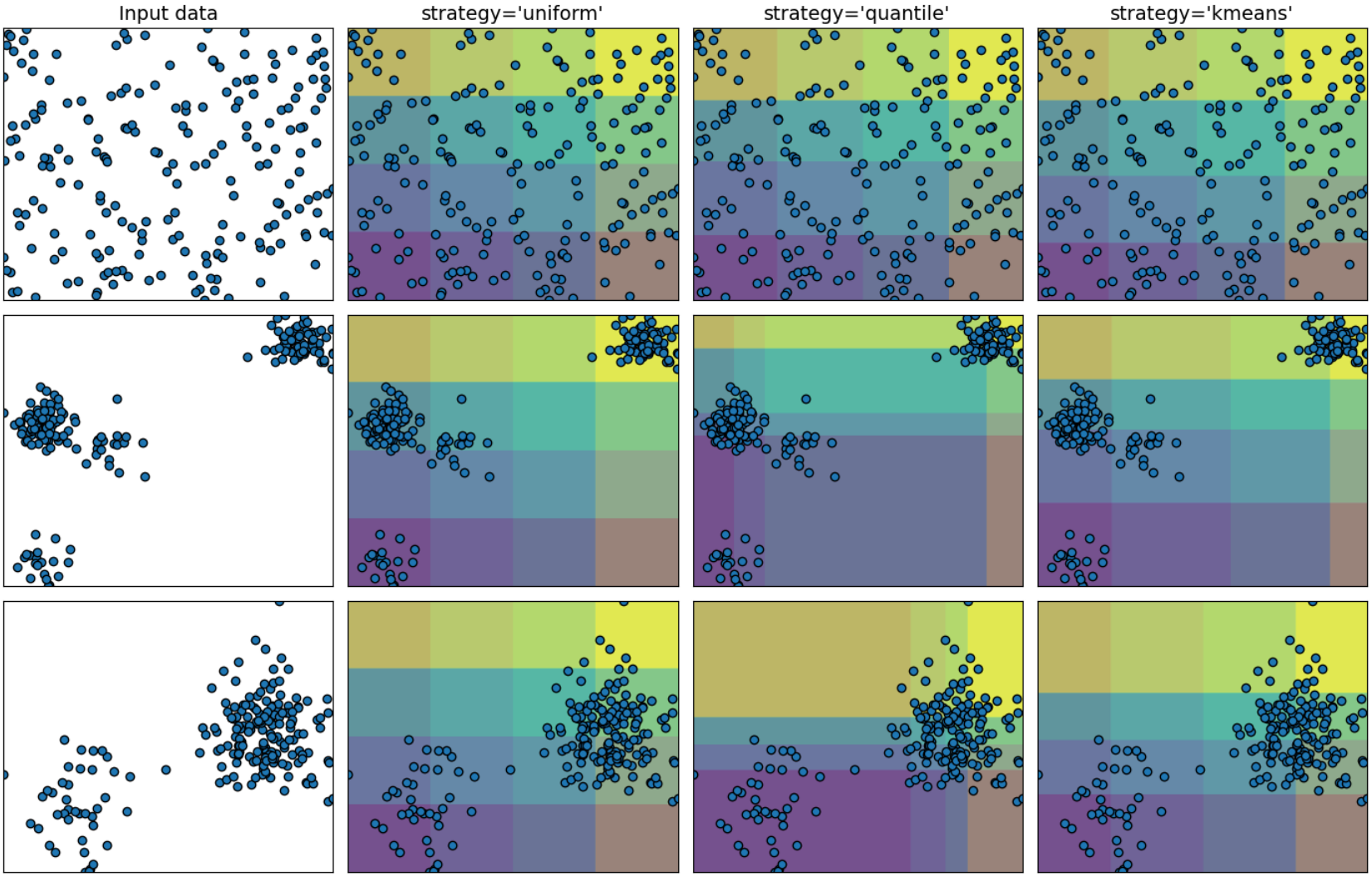


Equal frequency (binning)



K-means clustering leads to better results

Comparison of Discretization



https://scikit-learn.org/stable/auto_examples/preprocessing/plot_discretization_strategies.html

KBinsDiscretizer in sklearn

- 'uniform': All bins in each feature have identical widths.
- 'quantile': All bins in each feature have the same number of points.
- 'kmeans': Values in each bin have the nearest center of a 1D k-means cluster.

```
from sklearn.preprocessing import KBinsDiscretizer
import numpy as np

# Creating a sample data
data = np.array([[1, 3, 5], [2, 7, 9], [4, 6, 8]])

# n_bins = 3 and equal-width (uniform) discretization
discretizer = KBinsDiscretizer(n_bins=3, encode='ordinal', strategy='uniform')

# Fitting and transforming the data into n_bins number of bins
binned_data = discretizer.fit_transform(data)

# Printing binned data to check the change
print(binned_data)
```

```
[[0. 0. 0.]
 [1. 2. 2.]
 [2. 2. 2.]]
```

KBinsDiscretizer in sklearn

```
from sklearn.datasets import load_iris
from sklearn.preprocessing import KBinsDiscretizer

dataset = load_iris()

# Splitting the independent and dependent variables
i_data = dataset.data
response = dataset.target

feature_names = dataset.feature_names
discretizer = KBinsDiscretizer(n_bins=3, encode='ordinal', strategy='uniform')

# Fitting and transforming the data into n_bins number of bins
binned_data = discretizer.fit_transform(i_data)
print(binned_data)
```

Supervised Discretization Methods

- Classification (e.g., decision tree analysis)
  - Supervised: Given class labels, use *entropy* to determine split point (discretization point)
  - Top-down, recursive split

- Correlation analysis (e.g., Chi-merge: χ^2-based discretization)
  - Supervised: use class information
  - Bottom-up merge: find the best neighboring intervals (those having similar distributions of classes, i.e., low χ^2 values) to merge
  - Merge performed recursively, until a predefined stopping condition
  - e.g., ChiMerge

(Shannon) Entropy

- In the following feature values & target values
  target: - - - + + + + +
  value: 0, 4, 12, 16, 18, 24, 26, 28
- Which split point is the best one?
- Select the split point that makes the target values of each interval as homogeneous (certain/pure) as possible
  - In interval [0,14), every target value is –
  - In interval [14, 28], every target value is +
- Use entropy as the measure of uncertainty(impurity)
- Top-down and recursive approach

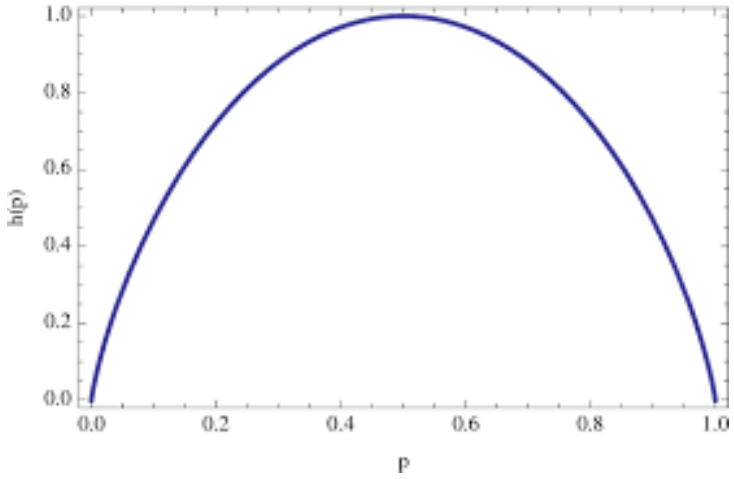
(Shannon) Entropy

- Entropy measures the degree of (im)purity

- Entropy function(Shannon entropy):

  1) represents the degree of impurity in prob. dist.

  2) also represents the amount of information prob. distribution contains



- Formula for computing the entropy entropy*(log=log2)*

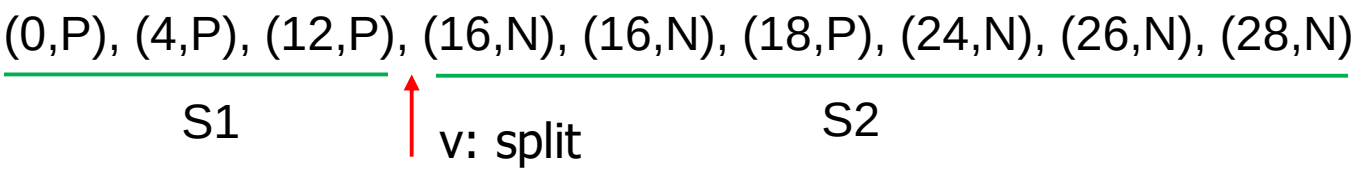
$$\text{entropy}(p_1, p_2, \ldots, p_n) = -p_1 \log p_1 - p_2 \log p_2 \ldots - p_n \log p_n$$

Entropy-based Discretization

- Find best split so that the bins are as pure as possible
- Formally characterized by maximal information gain.

- Given attribute-value/class pairs:
  - S={(0,P), (4,P), (12,P), (16,N), (16,N), (18,P), (24,N), (26,N), (28,N)}
- Let S denote the above 9 pairs, p=4/9 be fraction of P pairs, and n=5/9 be fraction of N pairs.
- Entropy(S) is the entropy of original values **before** split.
- Entropy(S) = - p log p - n log n = – 4/9*log(4/9) – 5/9*log(5/9)
  - Smaller entropy – set is relatively pure; smallest is 0.
  - Large entropy – set is mixed. Largest is 1.

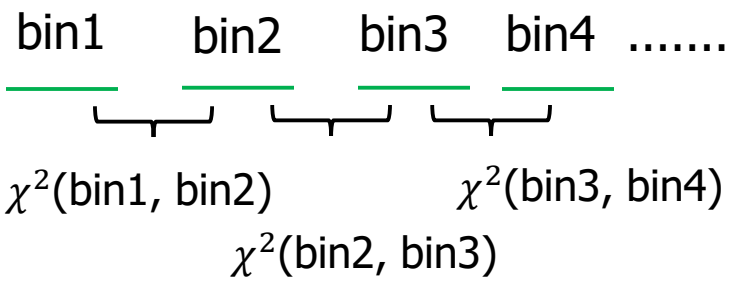
Entropy-based Discretization



- Possible splits: mid points between any two consecutive values.
- Goal: choose the split point v that makes both S1 and S2 as pure as possible
- S is divided into two sets:
  - S1: value <= v and S2: value > v
- Entropy(Information) **after** split:
  - Info(S1,S2) = (|S1|/|S|) Entropy(S1)+ (|S2|/|S|) Entropy(S2)
- Information gain of the split: entropy before split – entropy after split
  - Gain(v,S) = Entropy(S) – Info(S1,S2)
- Therefore the Goal is now to split with maximal information gain.
- For v=14, Info(S1,S2) = 0 + 6/9*Entropy(S2) = 6/9 * 0.65 = 0.433
  - Gain(14,S) = Entropy(S) - 0.433
  - maximum *Gain* means minimum *Info*.
- The best split is found after examining ALL possible splits.

Chi-square based methods

- Entropy-based criteria focus on the information quality, while chi-square criteria focus on the statistical quality.
- Examples:
  - ChiMerge bottom-up, local
  - ChiSplit top-down, local



Chi-Square Test

- χ^2 (chi-square) test

$$\chi^2 = \sum \frac{(observed - expected)^2}{expected}$$

- The larger the χ^2 value, the more likely the variables are related
- The cells that contribute the most to the X^2 value are those whose actual count is very different from the expected count
- Correlation does not imply causality
  - # of hospitals and # of car-theft in a city are correlated
  - Both are causally linked to the third variable: population

ChiMerge

- Initialization step
  - Place each distinct continuous value into its own interval(bin)
- Bottom-up fashion
  - Using chi-square test determine when adjacent intervals should be merged
  - Merge adjacent intervals with the smallest Chi square value.
  - Repeat until a stopping criteria (set manually) is met

Chi-Merge Example

bin	p		n		Sum (row)
18	2	1	1	2	3
17	0	1	3	2	3
Sum(col.)	2		4		6

expected values in red

$$\chi^2 = \frac{(2-1)^2}{1} + \frac{(1-2)^2}{2} + \frac{(0-1)^2}{1} + \frac{(3-2)^2}{2} = 3$$

bin	p		n		Sum (row)
16	1	0.5	2	2.5	3
17	0	0.5	3	2.5	3
Sum(col.)	1		5		6

$$\chi^2 = \frac{(1-0.5)^2}{0.5} + \frac{(2-2.5)^2}{2.5} + \frac{(0-0.5)^2}{0.5} + \frac{(3-2.5)^2}{2.5} = 1.2$$

- In this case, 17 will emerge with 16, instead of 18

Encoding

Nominal Data Conversion

- Three types of attributes
  - Numeric—real numbers, e.g., integer or real numbers
  - Nominal(categorical)—values from an unordered set, e.g., color, dept.
  - Ordinal—values from an ordered set, e.g., military or academic rank

- Categorical feature encoding
  - Many algorithms can only handle numeric features, so we need to encode the categorical ones (change symbols to numbers)

	boro	salary	vegan
0	Manhattan	103	0
1	Queens	89	0
2	Manhattan	142	0
3	Brooklyn	54	1
4	Brooklyn	63	1
5	Bronx	219	0

Ordinal Encoding

- Simply assigns an integer value to each category in the order they are encountered
  - e.g., Bronx->0, Brooklyn->1, Manhattan->2, Queens->2
- Only really useful if there exist a natural order in categories
  - e.g., A, B, C, D, F in grade
  - You assign an implicit order between values
  - Model will consider one value to be 'higher' or 'closer' to another

	boro	boro_ordinal	salary
0	Manhattan	2	103
1	Queens	3	89
2	Manhattan	2	142
3	Brooklyn	1	54
4	Brooklyn	1	63
5	Bronx	0	219

One-hot Encoding (dummy encoding)

- Simply adds a new 0/1 feature for every category, having 1 (hot) if the sample has that category
- Can explode if a feature has lots of values, causing issues with high dimensionality

	boro	boro_Bronx	boro_Brooklyn	boro_Manhattan	boro_Queens	Salary
0	Manhattan	0	0	1	0	103
1	Queens	0	0	0	1	89
2	Manhattan	0	0	1	0	142
3	Brooklyn	0	1	0	0	54
4	Brooklyn	0	1	0	0	63
5	Bronx	1	0	0	0	219

Target Encoder

- One-hot encoding increases the dimensionality of data.
- Target encoder is to encode the categories by replacing them for a measurement of the effect they might have on the target.

- Target encoding transforms a categorical feature into a numeric feature without adding any extra columns.

- Target encoding works by converting each category of a categorical feature into its corresponding expected value.

- The approach to calculating the expected value will depend on the value you are trying to predict.
  - For Classification problems, the expected value is the conditional probability given that category.
  - For Regression problems, the expected value is simply the average value for that category.

Target Encoder

- On a *binary* classifier, the simplest way to do that is by calculating the probability p(t = 1 | x = ci) in which t denotes the target, x is the input and ci is the i-th category.
- This is the conditional probability of t=1 given the input was the category ci.
- We will replace the category ci for the value of p(t = 1 | x = ci)

- Preferred when you have lots of category values. It only creates one new feature for each class.

Target Encoder Example

original data

target encoder

	Animal	Target	Encoded Animal
0	cat	1	0.40
1	hamster	0	0.50
2	cat	0	0.40
3	cat	1	0.40
4	dog	1	0.67
5	hamster	1	0.50
6	cat	0	0.40
7	dog	1	0.67
8	cat	0	0.40
9	dog	0	0.67

conditional probabilities

	Animal Group	Target 0	Target 1	Probability of 1
0	cat	3	2	0.40
1	dog	1	2	0.67
2	hamster	1	1	0.50

cat -> 2/(3+2)=0.40
dog -> 2/(1+2)=0.67
hamster -> 1/(1+1)=0.50

Target Encoder with Multiclass

- For multiclass problems, repeat target encoder for each class value.
- ‘Color’ has 3 ‘Target’ values (0, 1, 2)
- Compute conditional probability for each target value.

	Color	Target
0	Red	0
1	Red	0
2	Red	1
3	Red	2
4	Red	2
5	Green	0
6	Green	1
7	Green	2

	Color_Target_1	Color_Target_2	Color_Target_3	Target
0	0.400000	0.200000	0.400000	0
1	0.400000	0.200000	0.400000	0
2	0.400000	0.200000	0.400000	1
3	0.400000	0.200000	0.400000	2
4	0.400000	0.200000	0.400000	2
5	0.333333	0.333333	0.333333	0
6	0.333333	0.333333	0.333333	1
7	0.333333	0.333333	0.333333	2

Target Encoder Smoothing

- Since we train models in a fraction of the data, and the mean of this fraction may not be the mean of the full population,
- So the target encoding might not be correct, and the model may even overfit the training data.
- Use prior smoothing to reduce those unwanted effects in target encoder.

- Assume we have a model to predict the quality of a book in an online store. We might have a book with 5 evaluations resulting in a score of 9.8 out of 10, but other books (with many evaluations) have a mean score of 7.
- This effect comes because we are using the mean of a small sample.

- We can "smooth" the score of this book with fewer evaluations by considering also the mean of the whole population of books.
- Suppose an attribute x has a value c_i. Its corresponding smoothed target encoding is

$$encoding = \alpha \cdot p(t = 1|x = c_i) + (1 - \alpha) \cdot p(t = 1)$$

where $\alpha = \frac{1}{1+e^{-(\#(t=1)-1)}}$ and $\#(t = 1)$ means the frequency of target $t = 1$

Target Encoder Smoothing

	boro	boro_encoded	salary	vegan
0	Manhattan	0.089647	103	0
1	Queens	0.333333	89	0
2	Manhattan	0.089647	142	0
3	Brooklyn	0.820706	54	1
4	Brooklyn	0.820706	63	1
5	Bronx	0.333333	219	0

- For Brooklyn, original target encoding is

$$p(vegan = 1|boro = Brooklyn) = 2/2$$

- Total mean of $vegan = 1$ is

$$p(vegan = 1) = \frac{2}{6}$$

- smoothing factor is

$$\alpha = \frac{1}{1 + e^{-(\#(t=1)-1)}} = \frac{1}{1 + e^{-(2-1)}} = 0.73$$

- Therefore, smoothed target encoding is

$$encoding = \alpha \cdot p(t = 1|x = c_i) + (1 - \alpha) \cdot p(t = 1) = 0.73 * \frac{2}{2} + 0.27 * \frac{2}{6} = 0.82$$

Encoding in sklearn

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OrdinalEncoder

data = {
   'Student': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
   'Grade': ['A', 'B', 'C', 'A', 'B'] }
df_orig = pd.DataFrame(data)
df = df_orig.copy()

# Ordinal Encoder: C->0, B->1, A->2
o_encoder = OrdinalEncoder(categories=[['C', 'B', 'A']])
df['Grade_encoded'] = o_encoder.fit_transform(df[['Grade']])
print(df, o_encoder.categories_)
```

```
  Student Grade  Grade_encoded
0   Alice     A            2.0
1     Bob     B            1.0
2 Charlie     C            0.0
3   David     A            2.0
4     Eva     B            1.0

[array(['C', 'B', 'A'], dtype=object)]
```

Encoding in sklearn (revised)

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder

# Load the data and assign X, y variables
df = pd.read_csv('...')
y = df['income'] # Income is a target
X = df.drop('income', axis=1) # features without target

label_enc = LabelEncoder()
y = label_enc.fit_transform(y)

print(y)
```

* LabelEncoder.fit(...) accepts a 1D array; LabelEncoder.classes_ is 1D

* OrdinalEncoder.fit(...) accepts a 2D array; OrdinalEncoder.categories_ is 2D

One-hot-encoding in Python

```
import pandas as pd
df = pd.DataFrame({'Segment': ['Low', 'High', 'Med'],
                   'Rating': ['A','A+','C'], 'Score': [43, 28, 15]})
print(df)
```

	Segment	Rating	Score
0	Low	A	43
1	High	A+	28
2	Med	C	15

```
# one-hot encoding
df_ohe = pd.get_dummies(df)
print(df_ohe)
```

	Score	Segment_High	Segment_Low	Segment_Med	Rating_A	Rating_A+	Rating_C
0	43	0	1	0	1	0	0
1	28	1	0	0	0	1	0
2	15	0	0	1	0	0	1

One-hot-encoding in sklearn

```
from sklearn.preprocessing import OneHotEncoder

enc = OneHotEncoder(handle_unknown='ignore')
X = [['Male', 1], ['Female', 3], ['Female', 2]]
enc.fit(X)
      OneHotEncoder(handle_unknown='ignore')
enc.transform([['Female', 1], ['Male', 4]]).toarray()
      array([[1., 0., 1., 0., 0.],
             [0., 1., 0., 0., 0.]])
```

Target Encoder

```
from category_encoders import TargetEncoder

X = [['male'], ['male'], ['female'], ['male'], ['female']]
y = np.array([1, 0, 0, 1, 1])

enc = TargetEncoder(handle_unknown='ignore')
enc.fit(X, y)

X2 = [['male'], ['female'], ['female'], ['male'], ['female']]
X2_enc= enc.fit_transform(X2, y)
print(X2_enc)
```

```
    0  0.656
    1  0.558
    2  0.558
    3  0.656
    4  0.558
```

Missing Data

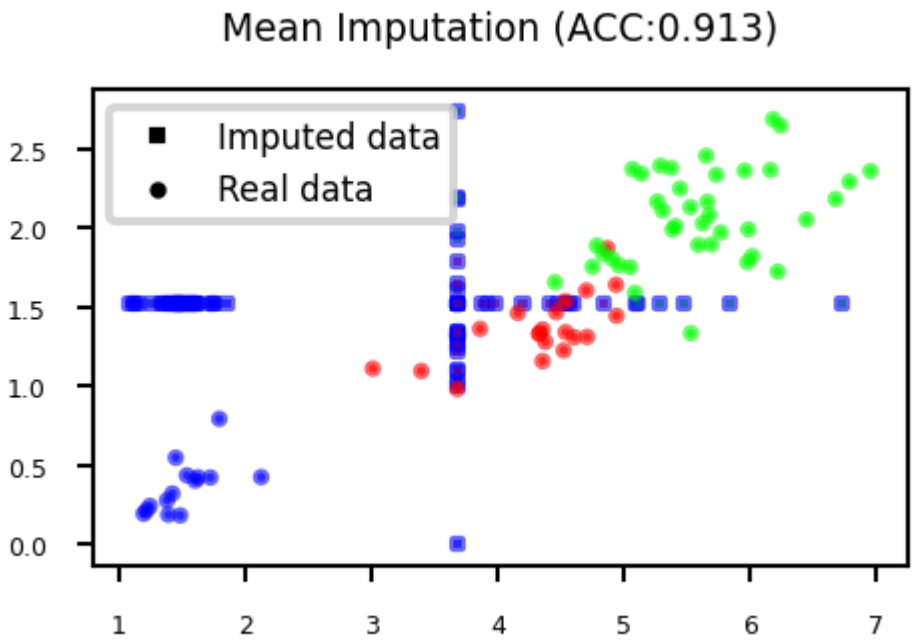
Missing Value Imputation

- **Imputation** is a process to fill the missing values in datasets.
- For numerical values, it uses mean, median, and constant.
- For categorical values, it uses the most frequently used and constant value.
- You can also train your model to predict the missing labels.

- Imputation Methods:
  - Mean/constant imputation
  - kNN-based imputation
  - Iterative (model-based) imputation

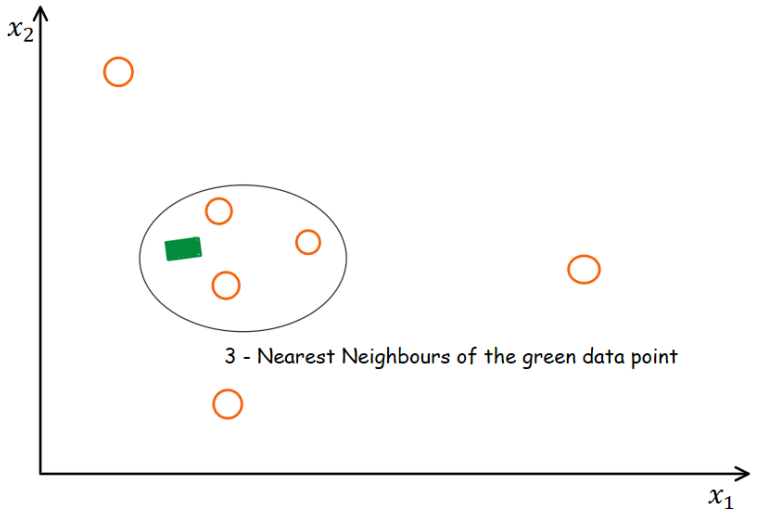
Mean Imputation

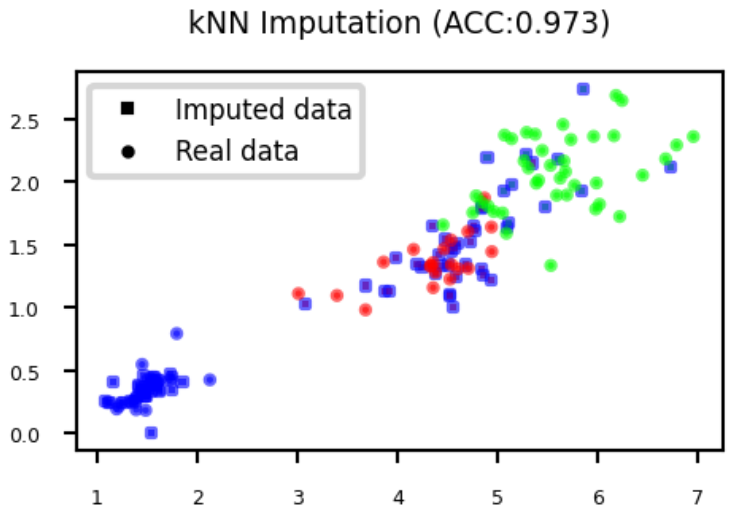
- Replace all missing values of a feature by the same value
- Numerical features: mean or median
- Categorical features: most frequent category
- Constant value, e.g. 0 or 'missing'
- Optional: add an indicator column for missingness



kNN imputation

- The KNNImputer class provides imputation for filling in missing values using the k-Nearest Neighbors approach.
- Each missing feature is imputed using values from nearest neighbors that have a value for the feature.
  - By default, a Euclidean distance metric is used to find the nearest neighbors.
  - The feature of the neighbors are averaged uniformly or weighted by distance to each neighbor.
- If there is at least one neighbor with a defined distance, the weighted or unweighted average of the remaining neighbors will be used





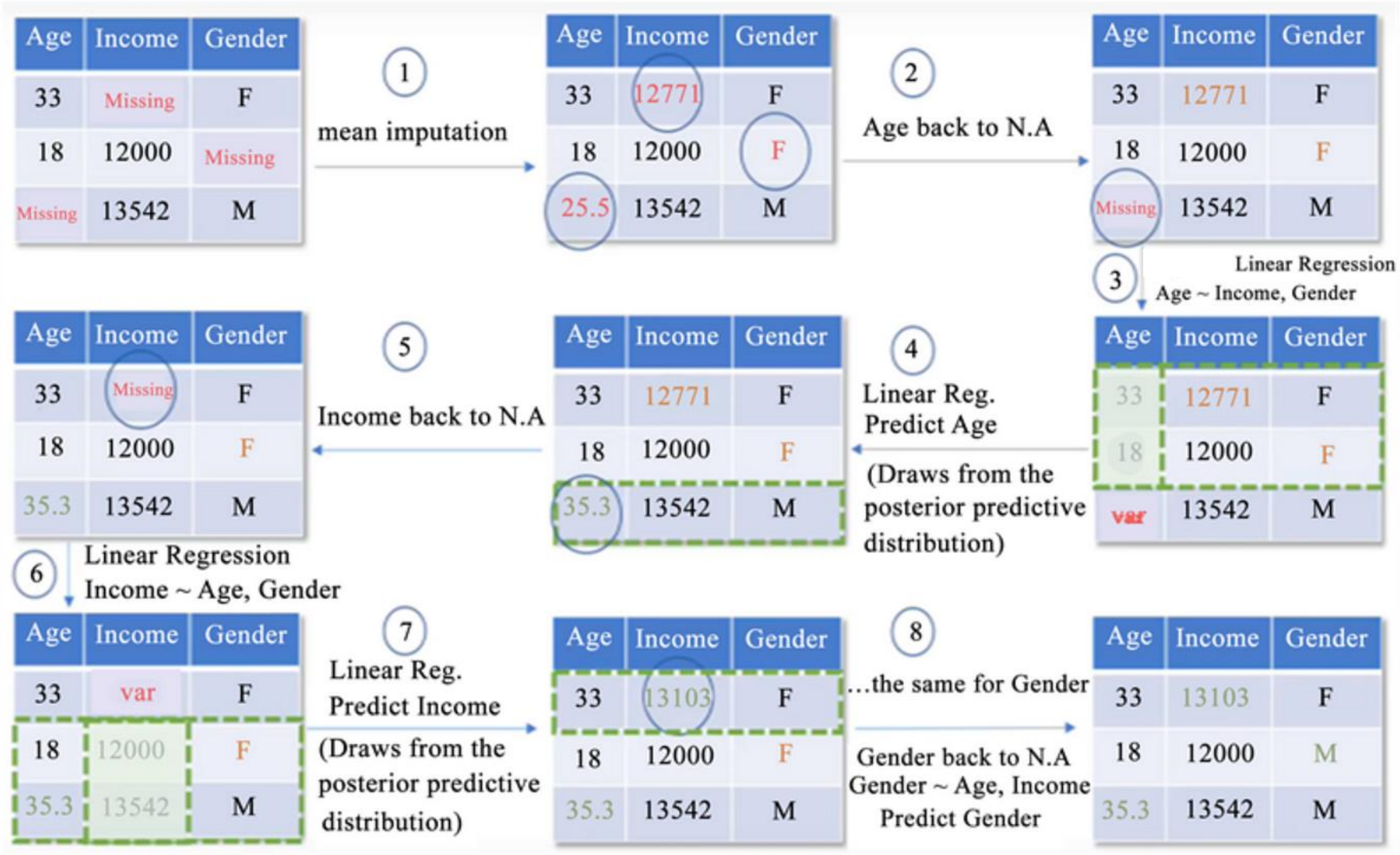
Iterative Imputation

- A more sophisticated approach is to use Iterative Imputation
- Iterative imputation refers to a process where each feature is modeled as a function of the other features, e.g. a regression problem where missing values are predicted.
- Each feature is imputed sequentially, one after the other, allowing prior imputed values to be used as part of a model in predicting subsequent features.
- Different regression algorithms can be used to estimate the missing values for each feature, although linear methods are often used for simplicity.

Model-based Iterative Imputation

- Better known as Multiple Imputation by Chained Equations (MICE)
- Iterative approach
  - Do first imputation (e.g. mean imputation)
  - Train model (e.g. RandomForest) to predict missing values of a given feature
  - Train new model on imputed data to predict missing values of the next feature
  - Repeat m times in round-robin fashion, leave one feature out at a time

Model-based Iterative Imputation



https://www.scirp.org/journal/paperinformation.aspx?paperid=112455

SimpleImputer in sklearn

```
import numpy as np
import pandas as pd
from sklearn.impute import SimpleImputer

df = pd.DataFrame({'A': [2, 2, np.nan, np.nan, np.nan, 8]})

imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
imputed = imputer.fit_transform(df)
df_imputed = pd.DataFrame(imputed, columns=df.columns)

df_imputed
```

- "mean" — replaces missing values with the mean
- "median" — replaces missing values with the median
- "most_frequent" — replaces missing values with the most frequent value
- "constant" — replaces missing values with the value in the ‘fill_value’ argument.

KNNImputer in sklearn

```
import numpy as np
from sklearn.impute import KNNImputer

nan = np.nan
X = [[1, 2, nan], [3, 4, 3], [nan, 6, 5], [8, 8, 7]]
imputer = KNNImputer(n_neighbors=2, weights="uniform")
imputer.fit_transform(X)
    array([[1. , 2. , 4. ],
           [3. , 4. , 3. ],
           [5.5, 6. , 5. ],
           [8. , 8. , 7. ]])
```

IterativeImputer in sklearn

```
from sklearn.impute import IterativeImputer

imp =SimpleImputer(strategy='mean', missing_values=np.nan)
X_complete = imp.fit_transform(X_train)

#kNN Imputation: KNNImputer
imp = KNNImputer(n_neighbors=5)
X_complete = imp.fit_transform(X_train)


#Multiple Imputation (MICE): IterativeImputer
#Choose estimator (default: BayesianRidge) and number of iterations (default: 10)
imp = IterativeImputer(estimator=RandomForestRegressor(), max_iter=10)
X_complete = imp.fit_transform(X_train)
```

- estimator can be
  - BayesianRidge: regularized linear regression
  - RandomForestRegressor: Forests of randomized trees regression
  - KNeighborsRegressor

Handling Imbalanced Data

Handling Imbalanced Data

- You have a majority class with many times the number of examples as the minority class
- There are also things we can do by preprocessing the data
- Resample the data to correct the imbalance
  - oversample the minor class
  - undersample the majority class
- Generate synthetic samples for the minority class
- Adjusting class weights
- Build ensembles over different resampled datasets
- Combinations of these

Oversampling the Minority Class

- A simple way to balance out an imbalanced dataset is to oversample the minority class by creating additional copies of the samples.
  - Randomly sample from the minority class, with replacement, until balanced
  - Optionally, sample until a certain imbalance ratio (e.g. 1/5) is reached
- Makes models more expensive to train, doens't always improve performance
- Similar to giving minority class(es) a higher weight (and more expensive)
- In skearn, the RandomOverSampler class can be used to randomly oversample the minority class.

Synthetic Minority Oversampling Technique (SMOTE)

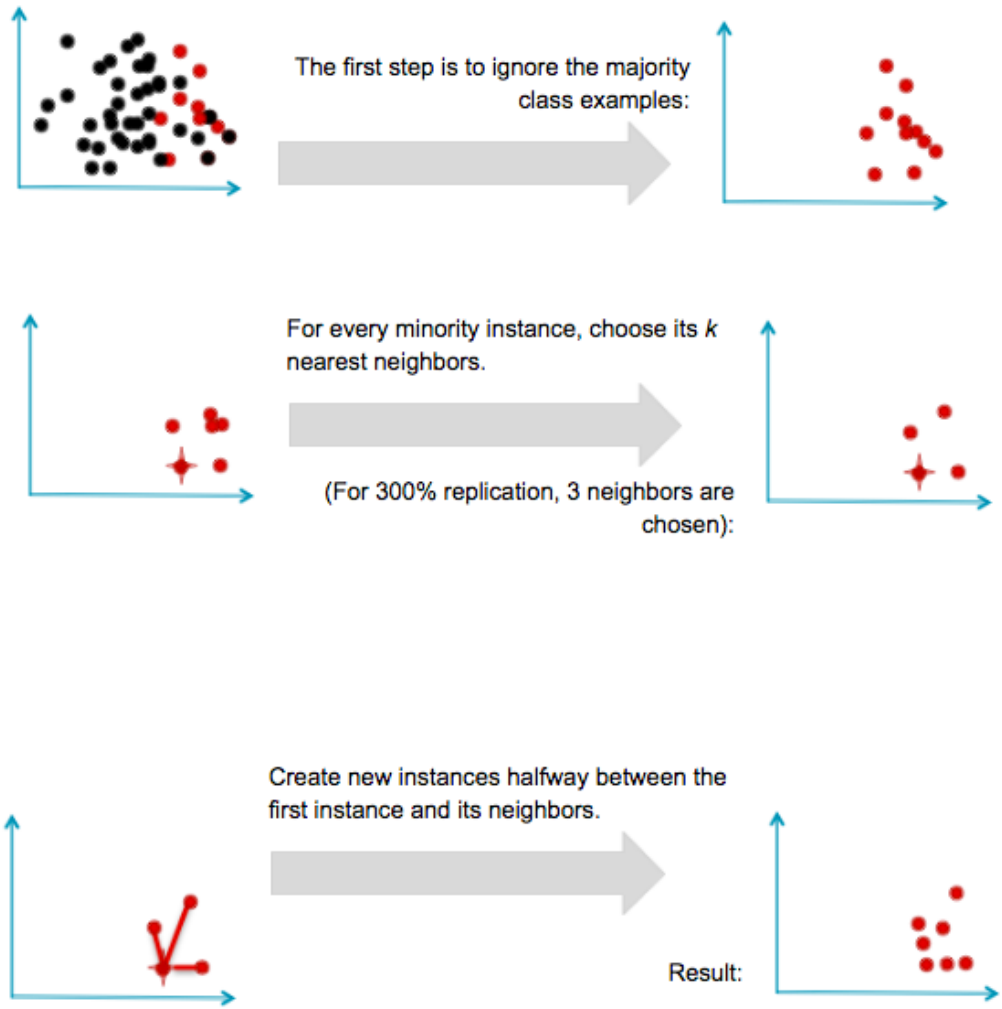
- A more advanced oversampling technique is SMOTE(Synthetic Minority Oversampling Technique).
- SMOTE generates synthetic samples for the minority class instead of just duplicating them. This creates more variety in the training data.

- Repeatedly choose a random minority point and a neighboring minority point
- Pick a new, artificial point on the line between them (uniformly)

- Nearest Neighbor Selection: For each minority instance, SMOTE identifies its k-nearest neighbors within the same class. The value of ‘k’ is a user-defined parameter.

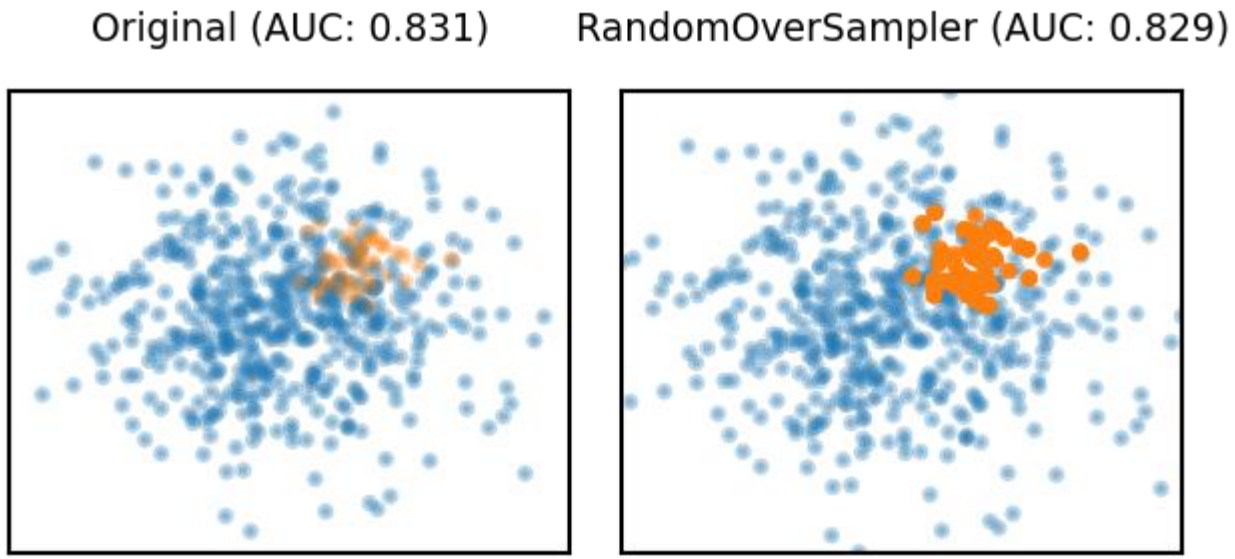
SMOTE

- Synthetic Instance Generation: For each minority instance, SMOTE creates synthetic instances by interpolating between the feature values of the original instance and its selected nearest neighbors.
- This is done by selecting a random neighbor, calculating the difference in feature values, and applying this difference to the original instance.
- Adjustment of Class Distribution: The synthetic instances are added to the minority class, effectively increasing its representation in the dataset.



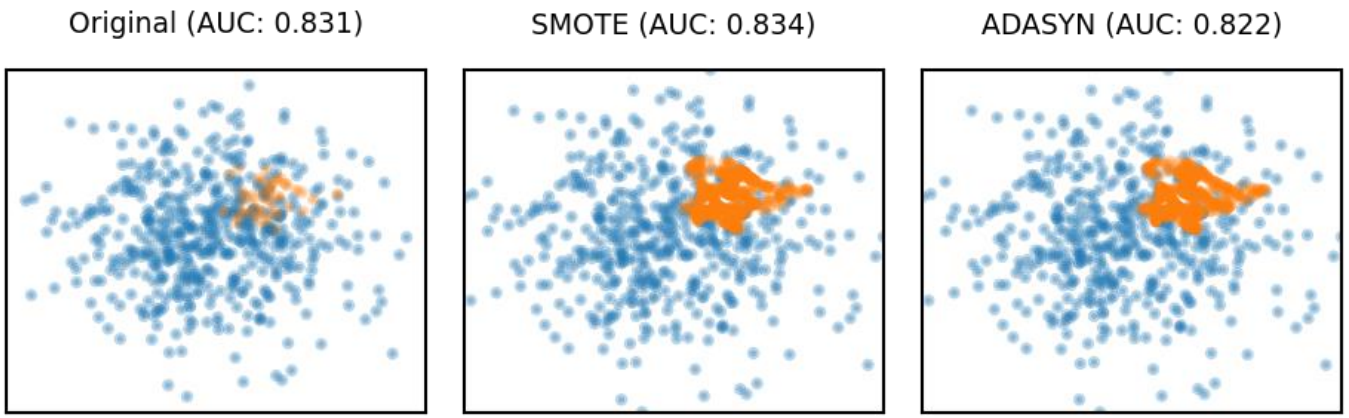
Oversampling the Minority Class in sklearn

```
from imblearn.over_sampling import RandomOverSampler

ros = RandomOverSampler(random_state=42)
X_resampled, y_resampled = ros.fit_resample(X, y)
```



SMOTE in sklearn

```
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X, y)
```

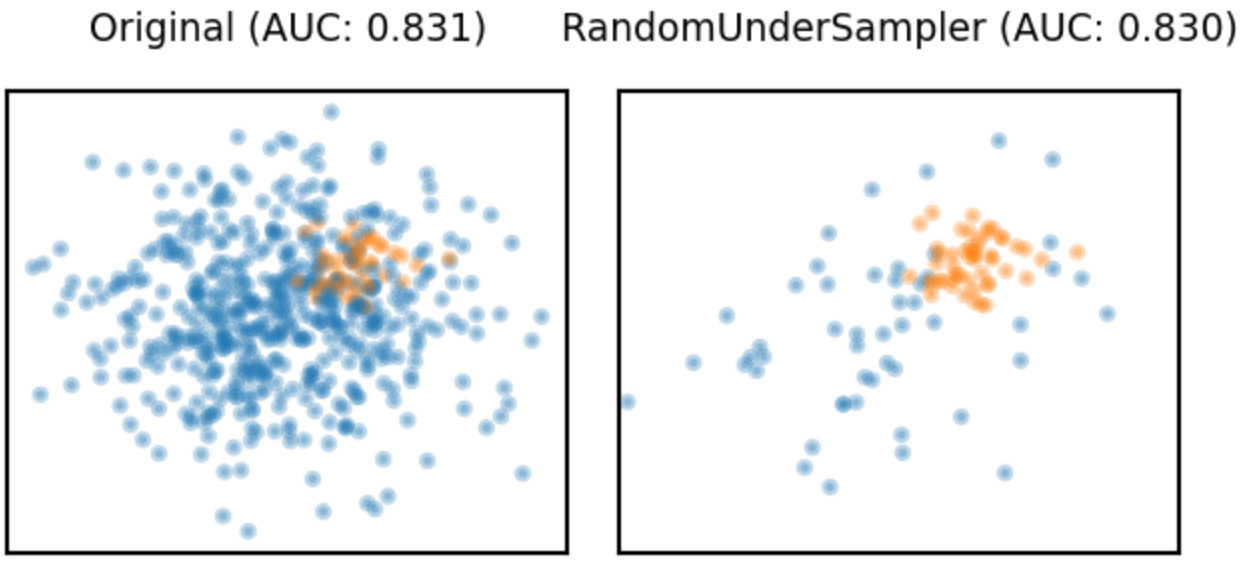


Undersampling the Majority Class

- Instead of oversampling the minority class, we can also undersample the majority class to balance out the class distribution.
  - Randomly sample from the majority class (with or without replacement) until balanced
  - Optionally, sample until a certain imbalance ratio (e.g. 1/5) is reached
- Multi-class: repeat with every other class
- Preferred for large datasets, often yields smaller/faster models with similar performance
- In sklearn, the RandomUnderSampler class randomly selects samples from the majority class to even out the class balance.

Undersampling the Majority Class in sklearn

```
from imblearn.under_sampling import RandomUnderSampler

rus = RandomUnderSampler(random_state=42)
X_resampled, y_resampled = rus.fit_resample(X, y)
```



Adjusting Class Weights

- Another technique is to assign higher weights to samples from the minority class and lower weights to the majority class when training the model.
- Many scikit-learn models accept a class_weight parameter.
- The class_weight='balanced' option can also be used to automatically balance the weights inversely proportional to class frequencies.
- Evaluation Metrics: When dealing with imbalanced classes, accuracy can be misleading. Precision, recall, and F1-score are better metrics.

```
from sklearn.svm import SVC

class_weights = {0: 1.0, 1: 0.5}

svc = SVC(class_weight=class_weights)
svc.fit(X, y)
```

Pipeline

Pipeline

- A pipeline allows you to assemble several steps in your ML workflow that sequentially transform your data before passing the data to an estimator.

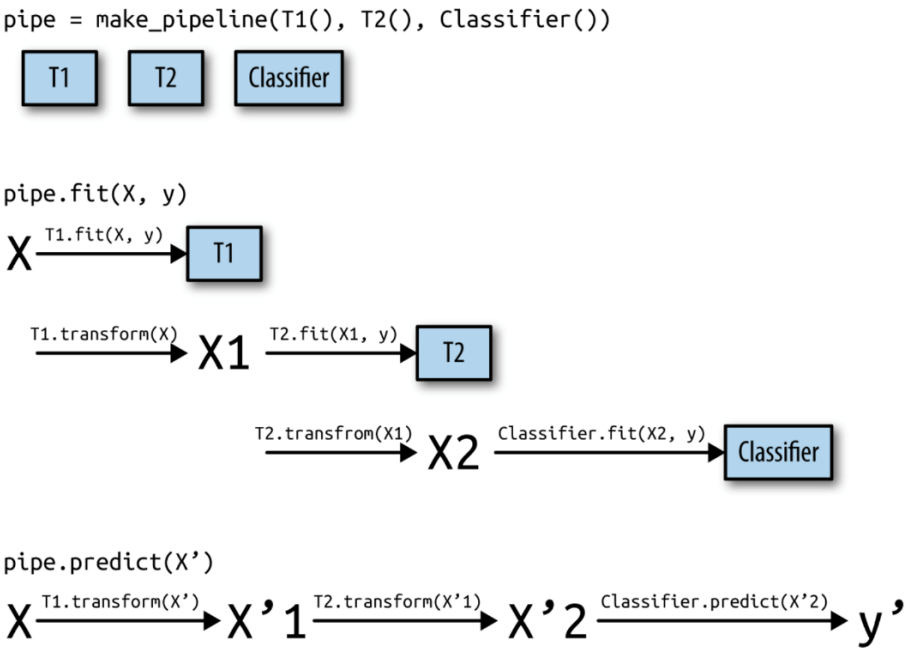
- Hence, a pipeline can consist of pre-processing, feature engineering and feature selection steps before passing the data to a final estimator for classification or regression tasks.

- scikit-learn's Pipeline uses a list of key-value pairs which contains the transformers you want to apply on your data as values.
- The keys can be used to access the parameters of the transformers
- As the transformers are stored in a list you can also access the transformers by indexing.

- To fit data on your pipeline and make predictions you can then run fit() and predict() as you would to with any transformer or regressor in scikit-learn.

Pipeline

- A pipeline is a combination of data transformation and learning algorithms
- It has a fit, predict, and score method, just like any other learning algorithm
- A pipeline combines multiple processing steps in a single estimator
- All but the last step should be data transformer



Pipeline in sklearn

```
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LinearRegression

# define pipeline
pipeline = Pipeline(
   steps=[("imputer", SimpleImputer()),
          ("scaler", MinMaxScaler()),
          ("regression", LinearRegression()) ]  )
```

OR

```
# define pipeline
pipeline = make_pipeline(steps=[ SimpleImputer(),
                  MinMaxScaler(), LinearRegression()  ]  )

# transform and run
pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_test)
```

Pipeline in sklearn

```
# Make pipeline, step names will be 'minmaxscaler' and 'linearsvc'
pipe = make_pipeline(MinMaxScaler(), LinearSVC())
```

OR

```
# Build pipeline with named steps
pipe = Pipeline([("scaler", MinMaxScaler()), ("svm", LinearSVC())])

# Correct fit and score
score = pipe.fit(X_train, y_train).score(X_test, y_test)

# Retrieve trained model by name
svm = pipe.named_steps["svm"]

# Correct cross-validation
scores = cross_val_score(pipe, X, y)
```