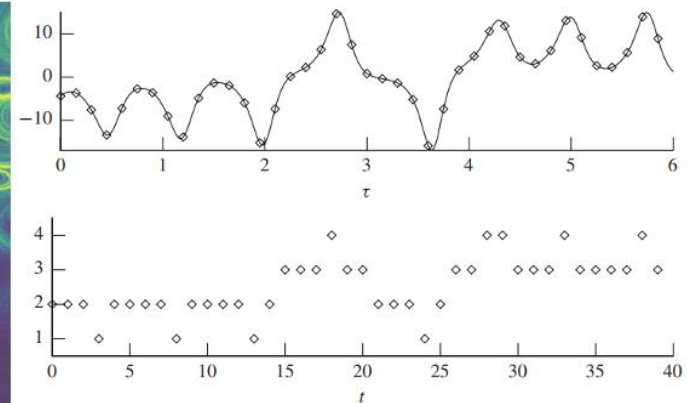
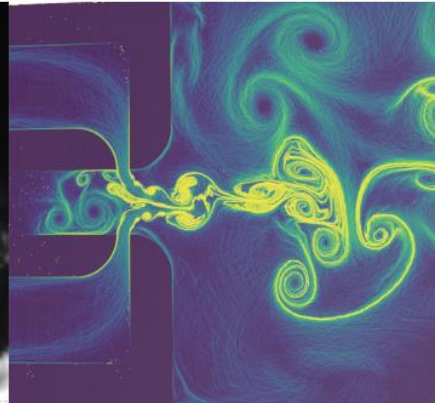


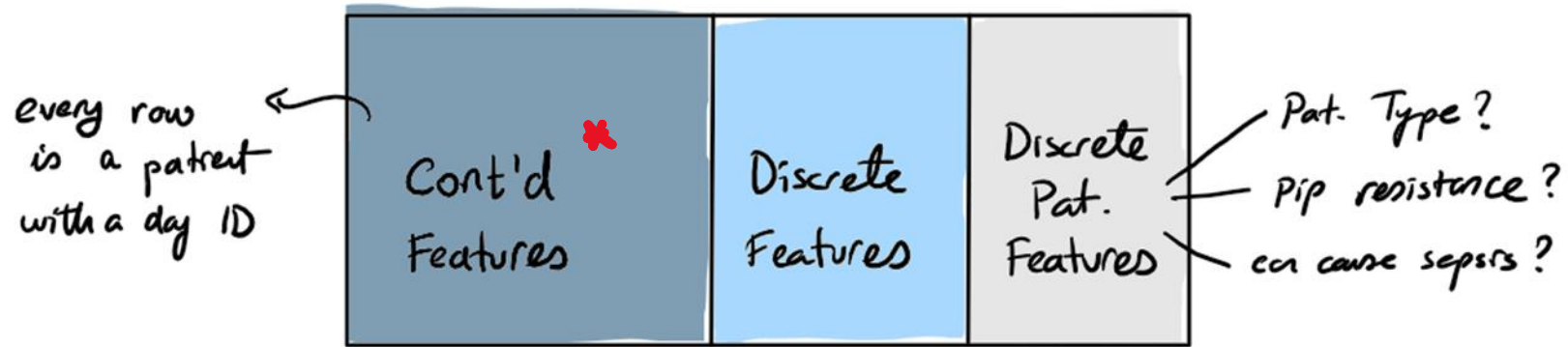
# Data Driven Engineering II: Advanced Topics

## Feature Engineering I

Institute of Thermal Turbomachinery  
Prof. Dr.-Ing. Hans-Jörg Bauer



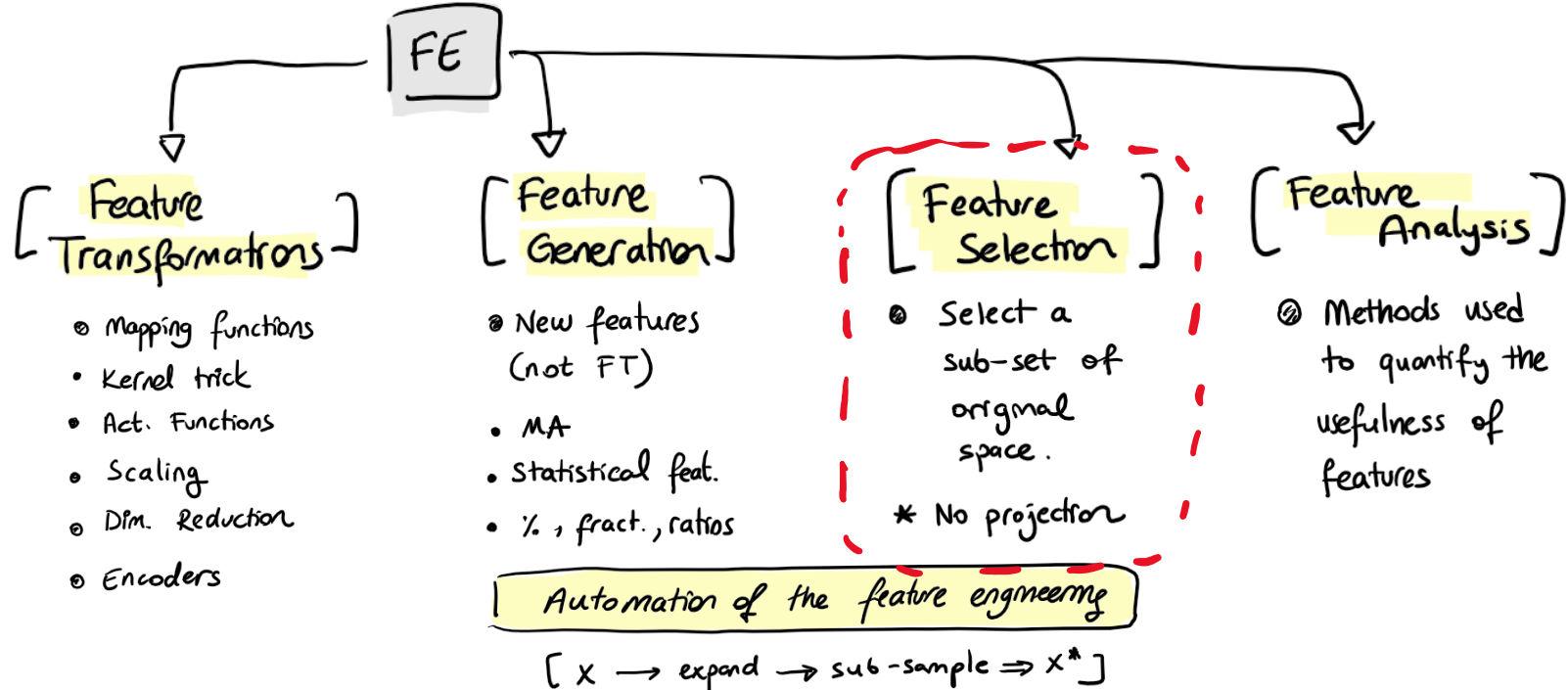
## Case Study: Th. Drug Management at ICU

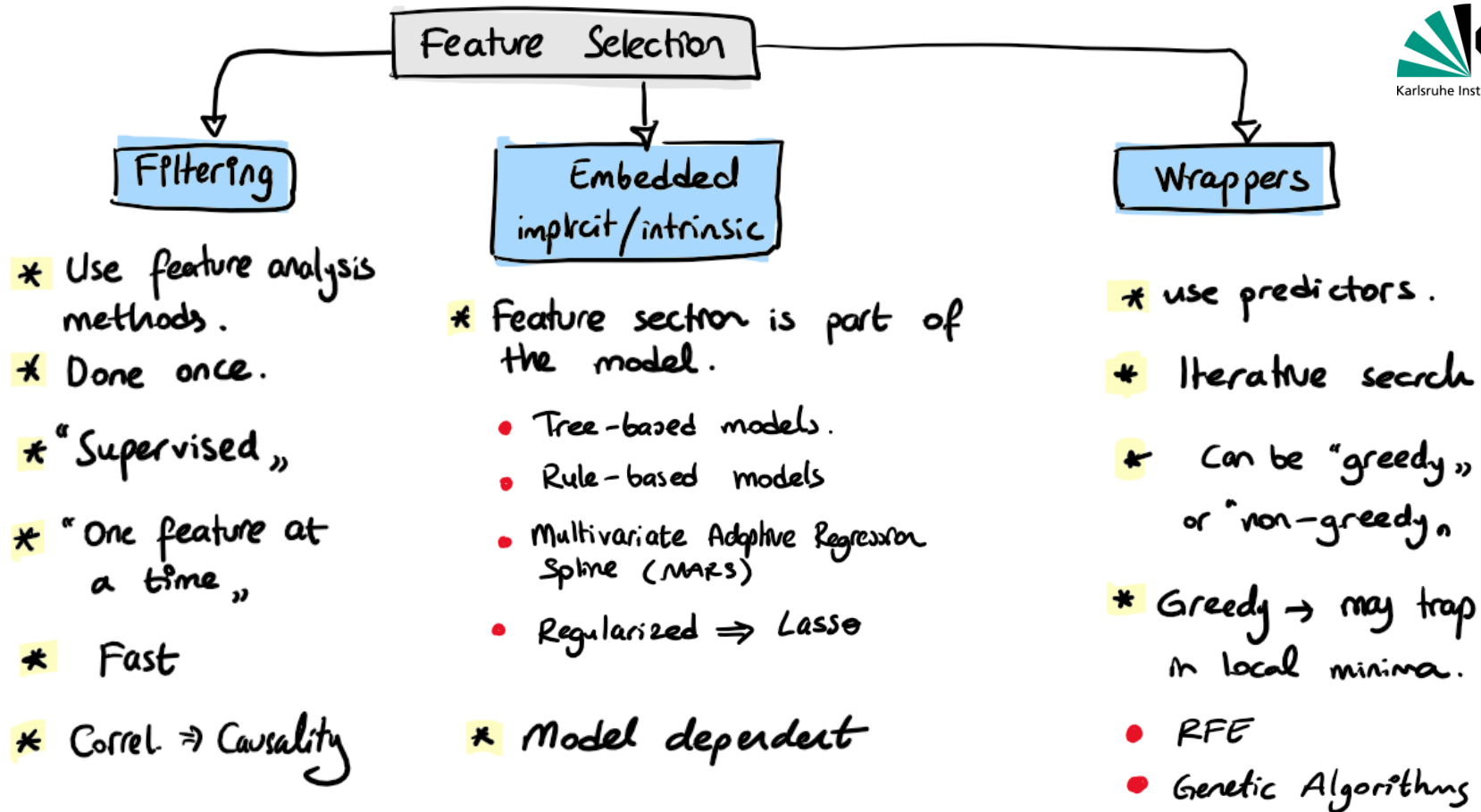


\* 2386 instances  $\Rightarrow$  day info  $\times$  patients

\*  $[37 + 54 + 108]$  features

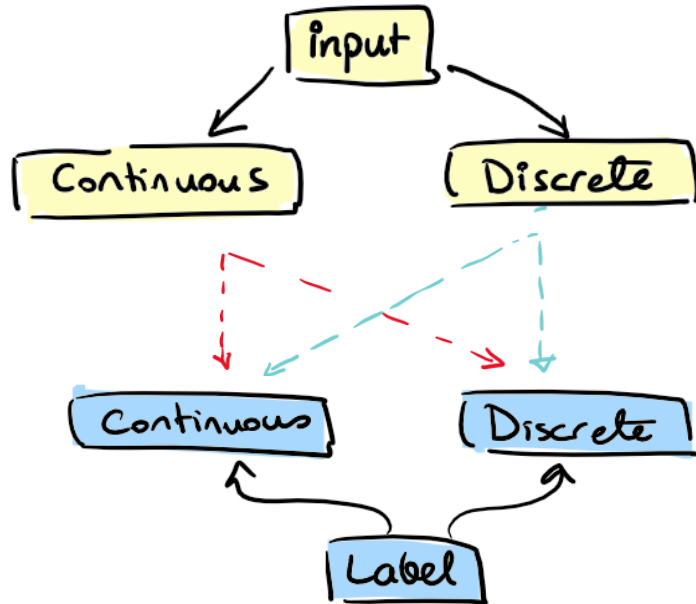
□ Operations we perform on object vectors during the whole pipeline.





# Feature Selection

- Goal := parsimonious model  $\Rightarrow$  Reduce model complexity



- ① Feature selection methods can depend on data type.
- ② Special care is needed for heterogeneous feature space.

A pipeline refers to a concept from the Scikit-Learn library in Python, used for building machine learning models.

A pipeline sequentially applies a list of transformations and a final estimator. Intermediate steps of the pipeline must be 'transforms', that is, they must implement fit and transform methods. The final estimator only needs to implement fit.

In your code:

```
fs_mi = SelectKBest(score_func=mutual_info_regression, k=k)
pipeline = Pipeline(steps=[('mi',fs_mi), ('LR', model)])
```

You're constructing a pipeline with two steps. The first step is a feature selection process using the SelectKBest function with mutual\_info\_regression as the score function. This step is named 'mi'. The second step ('LR') is to fit a model (which model is used is not specified in the given code snippet).

This pipeline can then be used like a regular Scikit-Learn estimator. When you call pipeline.fit(X, y), it will first apply fit\_transform of 'mi' on X and y, reducing the feature space of X to the k best features. Then, it will fit 'LR' on this transformed X and y.

Similarly, if you later call pipeline.predict(X\_new), it will first apply transform of 'mi' on X\_new, reducing the feature space of X\_new to the k best features seen in X during fitting, and then call predict of 'LR' on this transformed X\_new.

The advantage of using pipelines is that they help to prevent "data leakage" in your model by ensuring that data processing steps like transformations and feature selection are confined to each fold of your cross-validation procedure.

## Case I: Numerical $\Rightarrow$ Numerical

### Pearson Correlation

- \* What we have in base Corr. matrix.
- \* Lin. Corr. between feature & label.

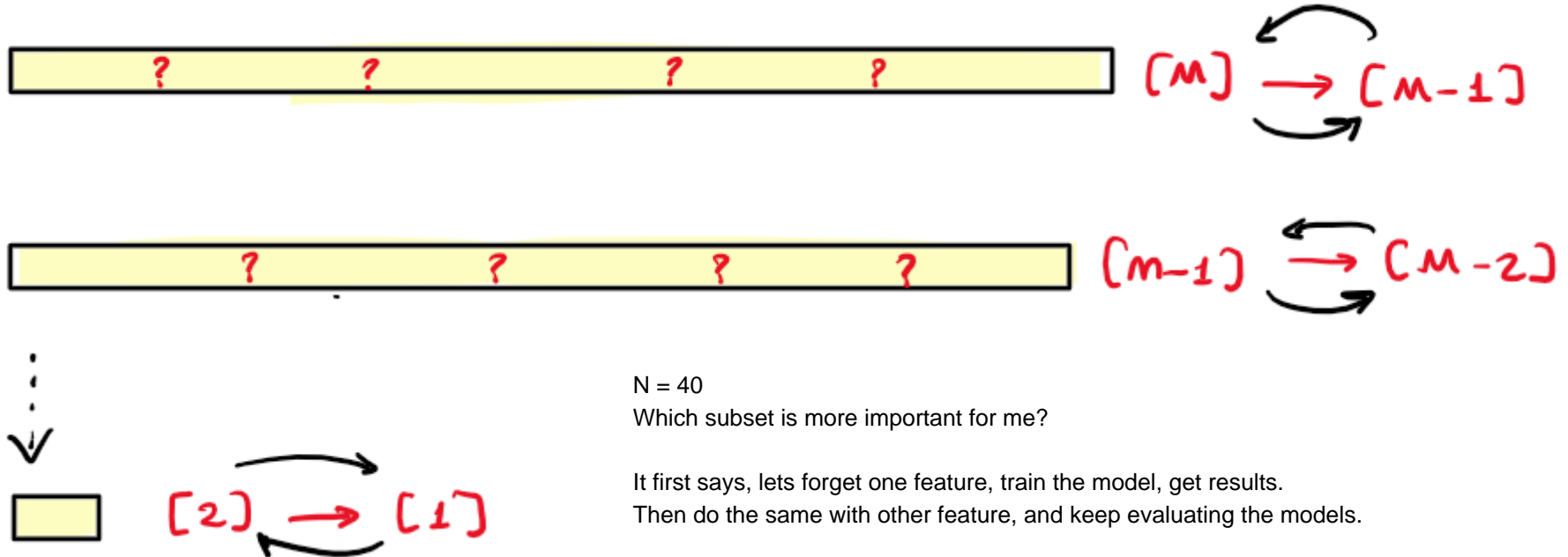
Here in both cases we are comparing labeled correlation.  
With mutual information we are looking at how well distributed is our data.

### Mutual Information

- $MI_{ij} = \text{Entropy}(i) - \text{Entropy}(i|j)$
- Used for "feature & label" couple
- Generalize well to multiclass

Notebook

# Wrappers -I : Greedy Approach



$N = 40$

Which subset is more important for me?

It first says, let's forget one feature, train the model, get results.

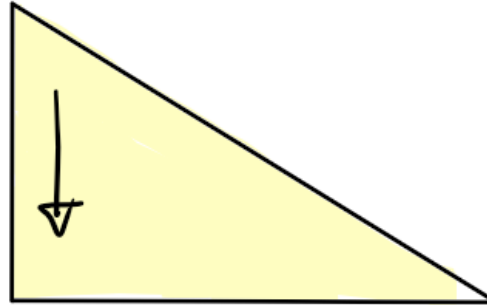
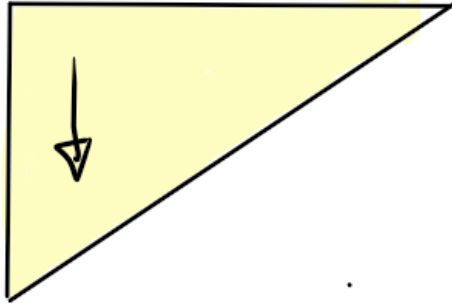
Then do the same with other feature, and keep evaluating the models.

Recursive feature elimination. It looks at all permutations and elimination.



## Wrappers -I : Greedy Approach

Its like PCA, once you fix one axis, the rest remain on them.



{ NP-hard }

- \* Execute many times on random subsets  $\Rightarrow$  Check freq. of selection
- \* Stochastic search; simulated annealing; gradient decent; ...

Notebook

In feature selection for machine learning, the two primary types of greedy wrapper methods are:

1. **Forward Selection:** This process begins with an empty set of features. The predictor is trained on each individual feature, and the one that produces the best performing model, according to a pre-defined evaluation criterion, is selected. The process is repeated, and at each subsequent iteration, one feature is added to the set of selected features. The feature to add is the one that, in conjunction with the already selected features, gives the best improvement of the performance metric. This process continues until adding more features does not improve the performance, or all features are included.

2. **Backward Elimination (or Backward Selection):** This is the reverse process of Forward Selection. The procedure starts with all candidate features. At each iteration, the least important feature is removed from the set. The removed feature is the one that, when removed from the current set of features, results in the smallest decrease in the performance metric. The process is repeated until no further improvement can be made.

The Recursive Feature Elimination (RFE) method is a type of backward elimination method with a slight variation. It works by fitting the model, ranking the features by importance, discarding the least important features, and re-fitting the model. This process is repeated until a specified number of features remain.

As for the "increasing number of features" approach, it's a forward selection method. You start with one feature and add one feature at a time that improves the performance of the model the most until adding more features does not improve the performance. It's a type of greedy algorithm because at each step it makes the locally optimal choice with the hope that these local decisions will lead to a global optimum.

It's important to note that these methods can be computationally expensive, especially when the number of features is large. They also may not produce the global optimal set of features because they don't consider all possible feature combinations. However, in practice, they often work quite well.

To make these methods more efficient, you could:

1. Use more efficient algorithms for the initial ranking in RFE.
2. Use regularization methods to promote sparsity, effectively performing feature selection (like Lasso).
3. Use a randomized search instead of an exhaustive one.
4. Implement early stopping criteria.
5. Use a subset of the data or an estimate of the performance metric.
6. Integrate domain knowledge if possible to reduce the search space.

# Genetic Algorithm for Feature Selection

Input, 40 dimensions

$X, y$

Assume we just randomly pick some.

# features → initial random subsets

Some are better than the others. The features that work better,

$\sim 50 + 150$

Fit. Indv.

Mating Mutations

$\sim 200$

$X^*, y^*$

Randomly select 10 subsets, 200 times.

population  $\sim 200$

$\sim 200$

fitness  
"Score"

Estimator

It checks the accuracy of the "200" 10 subsets models and check the results.

# Genetic Algorithm for Feature Selection

Fully automated way of selecting the features.

It usually works relatively well comparing with other methods.

A Genetic Algorithm (GA) is a search heuristic that is inspired by the process of natural selection. Genetic algorithms are used to find approximate solutions to optimization and search problems.

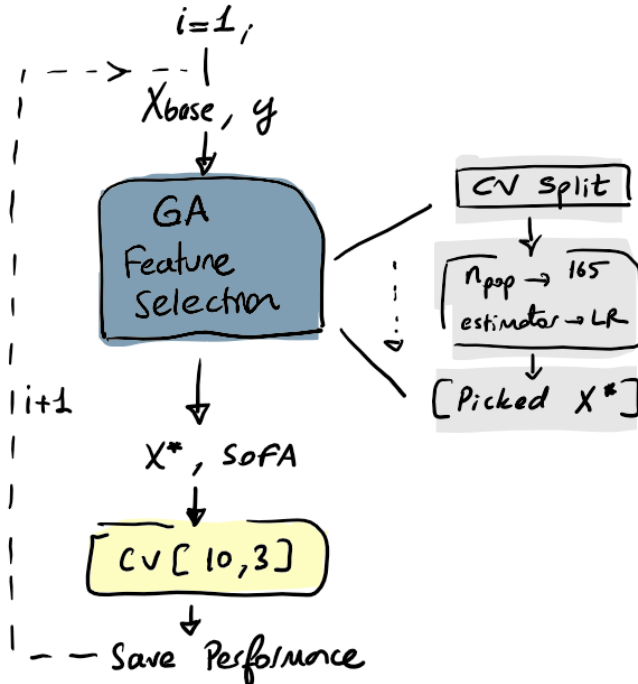
In the context of feature selection, GAs can be used to identify the subset of features that optimize a certain fitness function, which is typically the performance of a machine learning model.

Here's a high-level overview of how a genetic algorithm for feature selection works:

1. Initialization: Start with a population of chromosomes. In the context of feature selection, each chromosome is a possible solution, i.e., a subset of features. Each gene in a chromosome represents a feature and can be either 0 (feature is not selected) or 1 (feature is selected).
2. Fitness Function: The fitness function evaluates how good each solution (chromosome) is. In feature selection, it can be the performance of a machine learning model (accuracy, precision, recall, F1 score, etc.) that is trained on the selected features. A high fitness score means a better solution.
3. Selection: Chromosomes are selected to form a new generation. The probability of a chromosome being selected is proportional to its fitness score. This mimics the survival of the fittest mechanism in natural selection.
4. Crossover (Recombination): Pairs of chromosomes (parents) are selected and cross over at a random point to produce new offspring, combining genes from both parents. This mimics the genetic recombination in natural reproduction.
5. Mutation: With a small probability, some genes in the offspring chromosomes are randomly flipped (0 becomes 1 or vice versa). This introduces variability in the population and prevents the algorithm from being stuck in local optima.
6. Evolution: Steps 3 to 5 are repeated for a fixed number of iterations/generations or until some stopping criteria are met (no improvement in the fitness score, for example). The best chromosome at the end of this process is selected as the optimal solution.

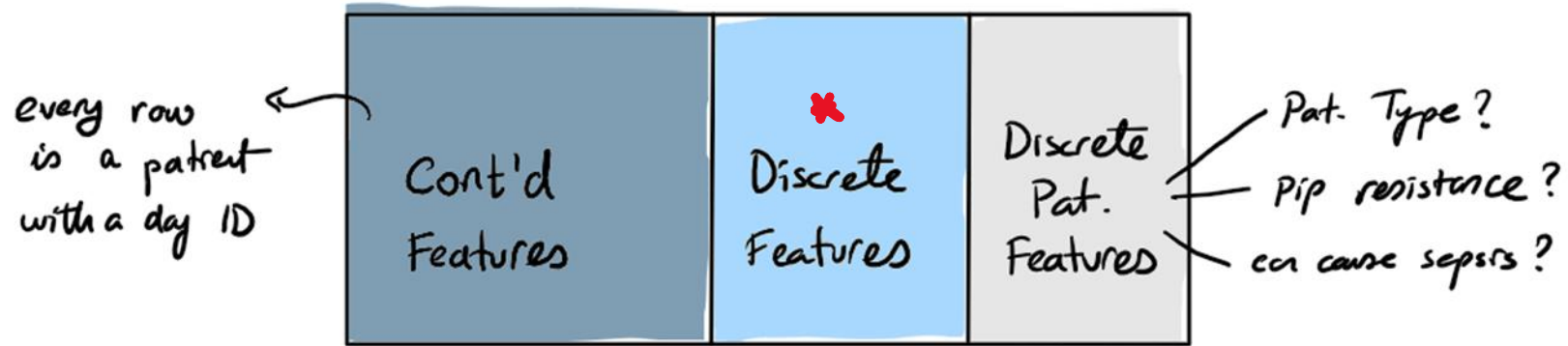
Cross-validation (CV) in the context of GAs for feature selection typically refers to the process of estimating the fitness function. In other words, instead of using the entire dataset to evaluate the performance of the model (and thus the fitness of a chromosome), the dataset is divided into a training set and a validation set. The model is trained on the training set and the performance is evaluated on the validation set. This process is repeated multiple times with different partitions of the data, and the average performance across all iterations is used as the fitness score. This helps to prevent overfitting and gives a more robust estimate of the model's true performance.

GAs are stochastic and population-based methods, making them less likely to get stuck in local optima than deterministic, greedy methods. They also naturally support multi-objective optimization, which can be useful in feature selection when you want to optimize for both model performance and simplicity (fewer features). However, GAs can be computationally intensive, especially with a large number of features. Also, setting the right parameters for the GA (population size, mutation rate, crossover rate, etc.) can be tricky and may require some experimentation.



However, GAs can be computationally intensive, especially with a large number of features. Also, setting the right parameters for the GA (population size, mutation rate, crossover rate, etc.) can be tricky and may require some experimentation.

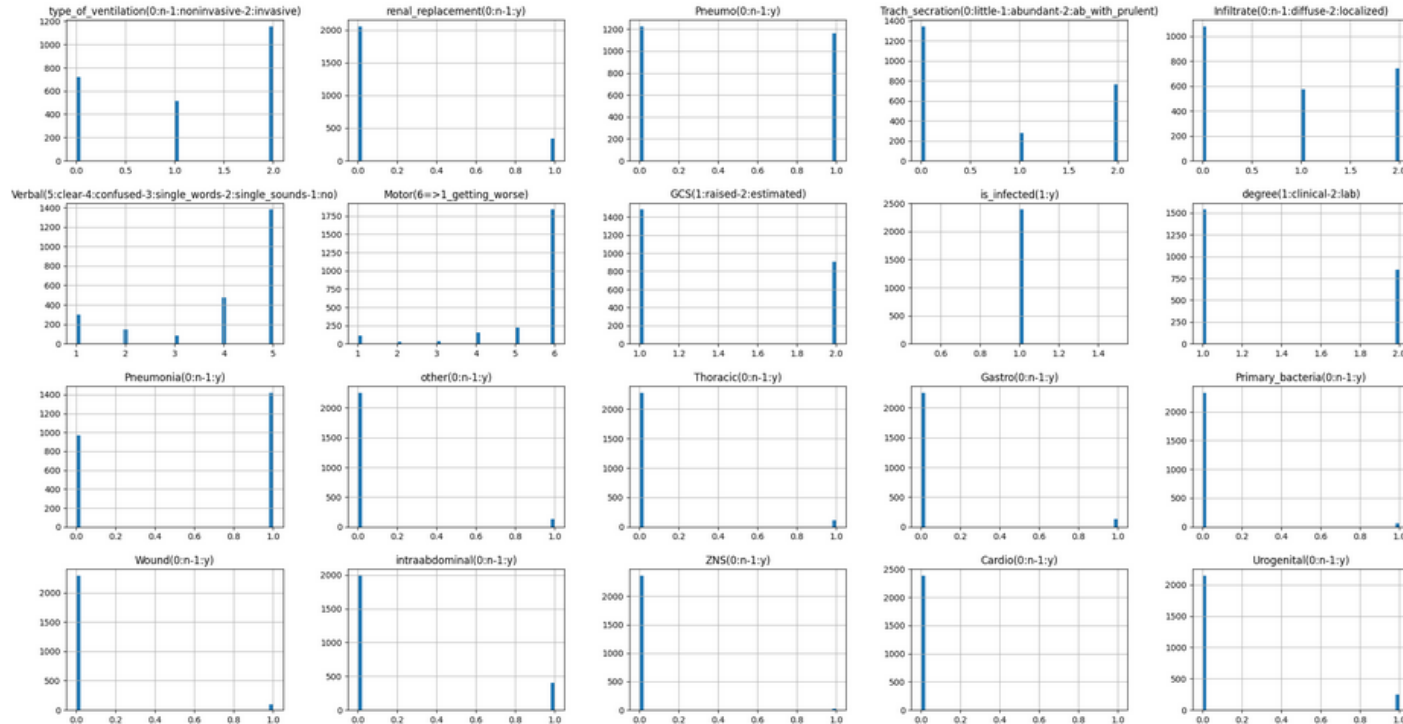
# Case Study: Th. Drug Management at ICU



\* 2386 instances  $\Rightarrow$  day info  $\times$  patients

\*  $[37 + 54 + 108]$  features

# Case Study: Th. Drug Management at ICU



## Case Study: Th. Drug Management at ICU

22	intraabdominal(0:n-1:y)	2386	non-null	float64
23	ZNS(0:n-1:y)	2386	non-null	float64
24	Cardio(0:n-1:y)	2386	non-null	float64
25	Urogenital(0:n-1:y)	2386	non-null	float64
26	other_2(0:n-1:y)	2386	non-null	float64
27	Brain(0:n-1:y-9:unknown)	2386	non-null	int64
28	Thromb(0:n-1:y-9:unknown)	2386	non-null	int64
29	Hypox(0:n-1:y-9:unknown)	2386	non-null	int64
30	Hypot(0:n-1:y-9:unknown)	2386	non-null	int64
31	Dysfu(0:n-1:y-9:unknown)	2386	non-null	int64
32	Azid(0:n-1:y-9:unknown)	2386	non-null	int64
33	Schock(0:n-1:y-9:unknown)	2386	non-null	int64
34	INFNEU_B	2386	non-null	float64
35	Clinical_cure(1:healing-2:improvement-3:failure-9:na)	2386	non-null	float64
36	Microbio_cure(1=>7-9)	2386	non-null	float64
37	Drainage(0:n-1:y)	2386	non-null	float64
38	Drainage_number	2386	non-null	float64
39	App_type(1=>6)	2386	non-null	float64
40	pathogen_resistance(1,2,3,8,9)	2386	non-null	float64
41	pip_give(0:n-1:y)	2386	non-null	float64
42	pip_give_type(1:empric-2:targetted)	2386	non-null	float64
43	Relus_delivery(0:n-1:y)	2386	non-null	float64

## Discrete Data : how to handle ?

Q1. Do we have single value features?

↳ Do we have low variance features?

Q2. Can I access domain knowledge to decide → categorical val.  
→ Ordinal val.

Q3. Check the data format ⇒ strings ?



Domain knowledge refers to having an understanding of the specific area of interest where you are applying your data analysis or machine learning algorithms. In the case of data science, this might mean understanding the specific meanings and relationships of the data you're working with.

"Categorical" and "ordinal" are two types of data that you might encounter in a dataset, and understanding them is part of the domain knowledge.

1. **Categorical Variables:** Categorical variables contain a finite number of categories or distinct groups. Categorical data might not have a logical order. For example, categorical predictors include gender, material type, and payment method. Usually, categorical variables need to be preprocessed before they can be used in many machine learning algorithms. One common way is one-hot encoding, where each category of a categorical variable is converted into a new binary variable (0 or 1).
2. **Ordinal Variables:** Ordinal variables are a type of categorical data with a set order or scale to it. This would include something like rating scores from 1-5. While they may seem like numbers, treating them as numeric variables could lead to some unintended consequences. For example, the difference between a 1 and a 2 rating might not mean the same as the difference between a 3 and a 4 rating. Depending on the specifics of the data and the use case, ordinal data may be treated as categorical (ignoring the order) or numeric (preserving the order), or some other strategy may be used to encode the ordinal information.

Domain knowledge can be very helpful when deciding how to handle different types of data. Understanding the context and the semantics of the data can guide the preprocessing steps and the choice of the model, leading to more accurate and interpretable results.

# Feature Encoding options

Discrete and continuous are two types of quantitative data that can be used in data analysis.

**Discrete Data:** Discrete data can only take on certain values. These values are distinct and separate, and can't be broken down further. It often represents counts, like the number of employees in a company, the number of cars in a household, or the number of times a particular event occurs. For example, you can't have 3.5 cars or 7.2 occurrences of an event - these values must be whole numbers.

**Continuous Data:** Continuous data, on the other hand, can take on any value within a certain range. It's often measurements or quantities that could potentially have a decimal or fractional value. Examples of continuous data include height, weight, time, temperature, or distance. For instance, a person could weigh 70.5 kg, or a race could be completed in 45.32 seconds.

It's important to understand the difference between these types of data, as the type of data can determine what kind of analysis can be done or what kind of machine learning model can be used. For example, certain statistical tests can only be used with continuous data, while others are designed specifically for discrete data. Similarly, some machine learning models require the input data to be in a certain format, depending on whether it's treating it as a continuous or discrete variable.

⚠ if domain access exists  $\Rightarrow$  use it

Q.1 # unique elements in the feature ?

- ⊗ "Low", number of discrete values
- ⊗ "high", number of discrete values

} depends on  
whole feature  
space

## Relatively low: One hot encoding

- Easiest to implement
- Accurate if used well
- useable in online learning
- Comp. inefficient  $\rightarrow$  # dimension  $\uparrow\uparrow\uparrow\uparrow$
- ~ practically  $\Rightarrow$  only for simple linear models

One-Hot encoding is independently of the Online learning.

Some methods require that we look at the statistics of the columns.  
Take a portion of it for feature engineering.

One-hot encoding is a process of converting categorical variables into a form that could be provided to machine learning algorithms to improve prediction. It works by creating a binary variable for each category of the data.

The degree of freedom in a statistical or mathematical model is the number of independent ways by which a dynamic system can move, without violating any constraint imposed on it. In terms of a dataset, it is the number of values in the final calculation of a statistic that are free to vary.

When you apply one-hot encoding to a categorical variable with  $n$  categories, you create  $n$  new binary variables (columns). This would seemingly give you  $n$  degrees of freedom. However, because of the way one-hot encoding works, one of these columns is completely determined by the others.

For example, if you have a variable "color" with three categories "red", "blue", "green", and you one-hot encode it, you'll get three new columns "color\_red", "color\_blue", "color\_green". But if an observation is not "red" and not "blue", it has to be "green". So, the "color\_green" column does not give you any new information that you didn't already have from the "color\_red" and "color\_blue" columns. In other words, the "color\_green" column is completely determined by the "color\_red" and "color\_blue" columns.

For this reason, you actually have  $n - 1$  degrees of freedom, not  $n$ . This is why in some cases, to avoid multicollinearity (perfect correlation between variables), one of the binary variables resulting from the one-hot encoding is dropped. This is known as "dummy variable trap" and dropping one variable is known as "using dummy encoding" or "creating dummy variables".

So, one-hot encoding can increase the apparent degrees of freedom in your data, but the effective degrees of freedom remain the same as with the original categorical variable.

If using a linear model, is better to drop a column? Why

If we are using SVM it doesn't matter erasing the columns

## Relatively low: One hot encoding

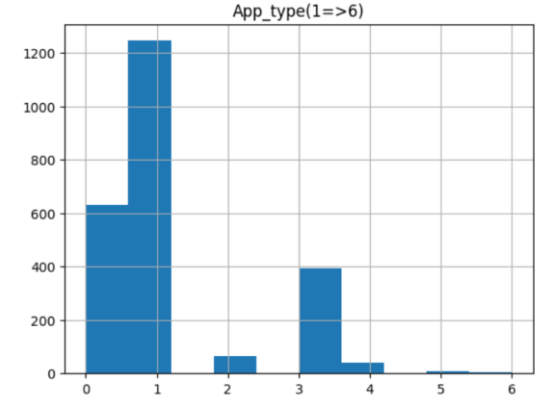
color	color-w	color-g	color-b	color-p
white	1	0	0	0
gold	0	1	0	0
black	0	0	1	0
pink	0	0	0	1

Notebook

## Single Feature Encoding

Label encoding  $\Rightarrow$   $\begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} 0 \\ 1 \\ \vdots \\ n-1 \end{bmatrix}$

If we do a label ordering text, do not use the automated label encoding, because then is disorder.



## Rare Label Encoding

Check # unique  $\Rightarrow$  ~ few value discrete values  
 $\rightarrow$  can we group these "rare," values as a single value

Notebook

Rare Label Encoding, also known as Rare Category Imputation, is a feature engineering technique used to handle categorical variables that have multiple categories but one or more of those categories occur very infrequently in the dataset.

If a category in a categorical variable only appears in a small fraction of the observations, it's considered a rare label. The problem with rare labels is that they may cause overfitting, particularly in certain machine learning algorithms. This is because the algorithm may try to fit the rare category that doesn't generalize well to unseen data.

Rare Label Encoding is used when:

1. There are many categories in a variable, and a few of them are represented by a small number of observations.
2. You want to prevent overfitting caused by rare labels.
3. You want to reduce the dimensionality of the data after one-hot encoding.

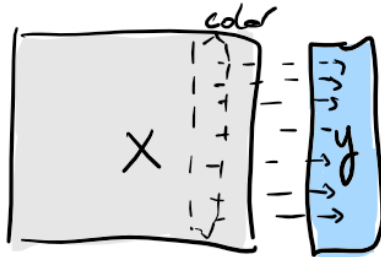
The most common method of Rare Label Encoding is to group all the rare labels under a new category, usually labelled as 'Other' or 'Rare'. This reduces the number of distinct categories, and hence reduces the number of new binary variables created by one-hot encoding.

It's important to note that Rare Label Encoding may result in loss of information, so it should be used judiciously, and its impact on model performance should be assessed.

## Count / Frequency Encoding

- Check uniqueness statistics  $\rightarrow$  # counts  $\left. \begin{array}{l} \rightarrow \% \text{ of counts} \end{array} \right\} \begin{array}{l} \text{after fit;} \\ \text{it is fixed!} \end{array}$

## Mean (label encoding)

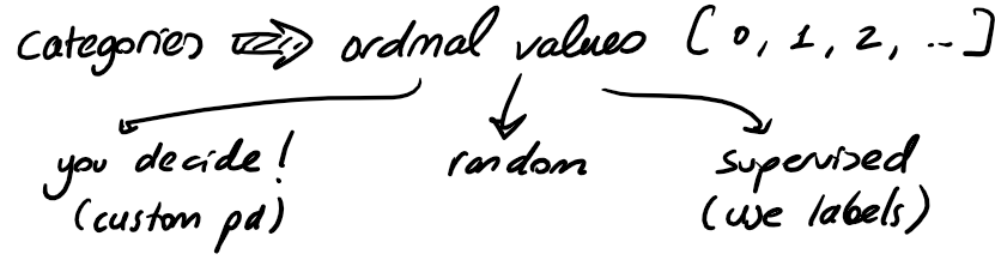


color

blue  $\leftarrow \text{mean}(y_{\text{blue}})$   
red  $\leftarrow \text{mean}(y_{\text{red}})$   
white  $\leftarrow \text{mean}(y_{\text{white}})$



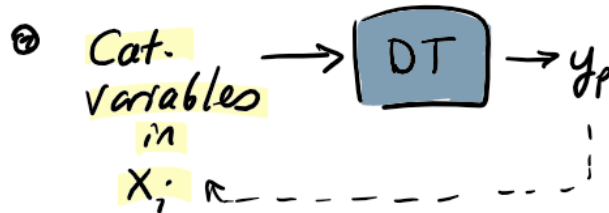
## Ordinal (label) encoding



## Decision Tree Encoder

It can be continuous, it can be discrete, it doesn't matter.

Here we can leak information



② One tree per feature

Here we train a model to decide which value to take

**Mean Label Encoding:** Mean label encoding, also known as target encoding or likelihood encoding, is a method where each category in a categorical variable is assigned a value calculated from the mean of the target variable. This is a form of encoding that uses information from both the independent variables (the category) and the dependent variable (the target).

The process works as follows: for each category, we calculate the mean of the target variable, and replace the category with that mean value. For example, if we have a categorical feature 'city' and our target variable is 'house price', we calculate the average house price for each city and replace the city name with that average price.

It's important to note that mean label encoding can introduce a problem known as target leakage. This happens because we're using the target variable to create new features. To avoid this issue, it's common to perform mean encoding on the training dataset and map those means to the categories in the validation/test datasets.

**Ordinal Label Encoding:** Ordinal label encoding is a method where each unique category value is assigned an integer value. This type of encoding is most suitable for ordinal variables, which are categorical variables with a natural ordered relationship between the categories.

The process works by first identifying the natural ordered relationship in the categories. Then, you assign integer labels that respect this order. For example, if you have a categorical feature 'size' with categories 'small', 'medium', and 'large', you could assign the numbers 1, 2, and 3 to them respectively.

For ordinal label encoding, it's important that the order assigned to the categories makes sense, because machine learning algorithms will interpret the categories as having an ordered relationship.

In both cases, the goal is to convert categorical data into a form that can be better understood by machine learning algorithms. The choice of encoding method depends on the specific dataset, the nature of the categorical variable, and the type of machine learning algorithm being used.

Notebook

# Catboost (label) encoder

- Cat. data  $\Rightarrow$  Numerical data
- again target based  $\sim \frac{\text{Target Sum} + \text{prior}}{\text{Feature Count} + 1}$

red	1
blue	3
blue	2
green	4
red	4
red	2
blue	1

$$\text{prior} = (1+3+2+4+4+2+1) / 7 = 17/7$$

$$\text{Target sum}_{\text{red}} = 1 + 4 + 2 = 7$$

$$\text{Feature count} = 3 \text{ (for red)}$$

$$\sim \left( \frac{7 + 17/7}{4} \right)$$

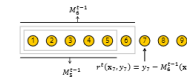


Figure 1: Ordered boosting principle, examples are ordered according to  $\sigma$ .

**Algorithm 1: Ordered boosting**  
**input** :  $\{(x_i, y_i)\}_{i=1}^n, I;$   
 $\sigma \leftarrow$  random permutation of  $[1, n];$   
**for**  $t \leftarrow 1$  **to**  $I$  **do**  
    **for**  $i \leftarrow 1$  **to**  $n$  **do**  
         $r_i \leftarrow y_i - M_{\sigma(i-1)}(x_i);$   
    **for**  $j \leftarrow 1$  **to**  $n$  **do**  
         $\Delta M \leftarrow$   
         $\text{LearnModel}((x_j, r_j) : \sigma(j) \leq i);$   
         $M_i \leftarrow M_i + \Delta M;$   
**return**  $M_n$

**Algorithm 2: Building a tree in CatBoost**  
**input** :  $M, \{(x_i, y_i)\}_{i=1}^n, \alpha, L, \{\sigma_i\}_{i=1}^n, \text{Mode}$   
 $\text{grad} \leftarrow \text{CalcGradient}(L, M, y);$   
 $r \leftarrow \text{random}(1, \sigma);$   
**if**  $\text{Mode} = \text{Plain}$  **then**  
     $G \leftarrow (\text{grad}_{\sigma_i}(i) \text{ for } i = 1..n);$   
**if**  $\text{Mode} = \text{Ordered}$  **then**  
     $G \leftarrow (\text{grad}_{\sigma_i, \sigma_i(i)-1}(i) \text{ for } i = 1..n);$   
 $T \leftarrow$  empty tree;  
**foreach** step of top-down procedure **do**  
    **foreach** candidate split  $c$  **do**  
         $T_c \leftarrow$  add split  $c$  to  $T;$   
        **if**  $\text{Mode} = \text{Plain}$  **then**  
             $\Delta(i) \leftarrow \text{avg}(\text{grad}_{\sigma_i}(p) \text{ for } p : \text{leaf}_{\sigma_i}(p) = \text{leaf}_{\sigma_i}(i)) \text{ for } i = 1..n;$   
        **if**  $\text{Mode} = \text{Ordered}$  **then**  
             $\Delta(i) \leftarrow \text{avg}(\text{grad}_{\sigma_i, \sigma_i(i)-1}(p) \text{ for } p : \text{leaf}_{\sigma_i}(p) = \text{leaf}_{\sigma_i}(i), \sigma_i(p) < \sigma_i(i)) \text{ for } i = 1..n;$   
         $\text{loss}(T_c) \leftarrow \text{cost}(\Delta, G);$   
     $T \leftarrow \arg \min_p (\text{loss}(T_c));$   
**if**  $\text{Mode} = \text{Plain}$  **then**  
     $M_{\sigma}(i) \leftarrow M_{\sigma}(i) - \alpha \text{avg}(\text{grad}_{\sigma_i}(p) \text{ for } p : \text{leaf}_{\sigma_i}(p) = \text{leaf}_{\sigma_i}(i) \text{ for } r' = 1..s, i = 1..n);$   
**if**  $\text{Mode} = \text{Ordered}$  **then**  
     $M_{\sigma}(i) \leftarrow M_{\sigma}(i) - \alpha \text{avg}(\text{grad}_{\sigma_i, \sigma_i}(p) \text{ for } p : \text{leaf}_{\sigma_i}(p) = \text{leaf}_{\sigma_i}(i), \sigma_i(p) \leq j) \text{ for } r' = 1..s, i = 1..n, j \geq \sigma_i(i) - 1;$   
**return**  $T, M$

Notebook


The CatBoost encoder is a type of target encoder developed as part of the CatBoost gradient boosting library. It's designed to handle categorical features, and it's especially useful when you have categorical variables with many categories.

The CatBoost encoder works by calculating the mean of the target variable for each category, similar to Mean or Target Encoding. However, it includes a special handling to avoid overfitting and target leakage by using a form of "leave-one-out" encoding.

Here's how it works:

1. For each row, the target mean is calculated using only the rows before it (in terms of dataset order). The current row is left out in this mean calculation to avoid leakage. This implies that row order could impact the encoding.
2. The encoder also incorporates a prior value, which is the global mean of the target variable over the whole training set.
3. These two values – the prior and the mean target of the category calculated without the current row – are then combined into a weighted sum. The more examples of a category in the data, the more weight is given to the category's mean target; conversely, the fewer examples, the more weight goes to the prior.

This approach takes care of rare categories and prevents over-fitting caused by noisy mean encoding of categories with few examples.

```
python  Copy code

import category_encoders as ce
import pandas as pd

# Assume we have the following dataframe
df = pd.DataFrame({
    'color': ['blue', 'green', 'red', 'blue', 'green', 'green'],
    'outcome': [1, 1, 0, 0, 0, 1]
})

# Create the CatBoost encoder
encoder = ce.CatBoostEncoder(cols=['color'])

# Fit and transform the data
df_encoded = encoder.fit_transform(df['color'], df['outcome'])

print(df_encoded)
```

Note that the real CatBoost library applies this encoding on-the-fly during training, which means the order of the training examples will matter, and it can even be randomized to achieve more robust results. The example given here is a simplified illustration of the encoding process.

# Multiple Correspondence Analysis (MCA)

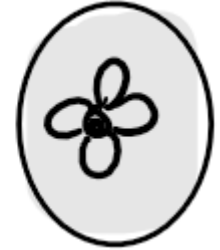
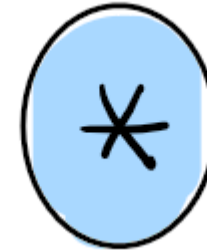
It assumes that the data dimensionality is projected in a plane, linearly or non linear. But the point to know is that is applied for continuous data.

□ PCA for categorical data

Don't use it if there are discrete values

□ Extension of CA

color	shape	obj
red	square	flower
blue	oval	star
gray	oval	flower



Notebook

# Feature Selection

① cat.  $\Rightarrow$  cat.

chi-squared  
MI

● cat  $\Rightarrow$  numerical

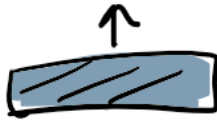
Statistical Method, not so used really.

More robust, more used.

② numerical  $\Rightarrow$  numerical

Pearson (corr. matrix)  
MI  
RFE  
GA

Model Selection



! Cat. variables

Notebook