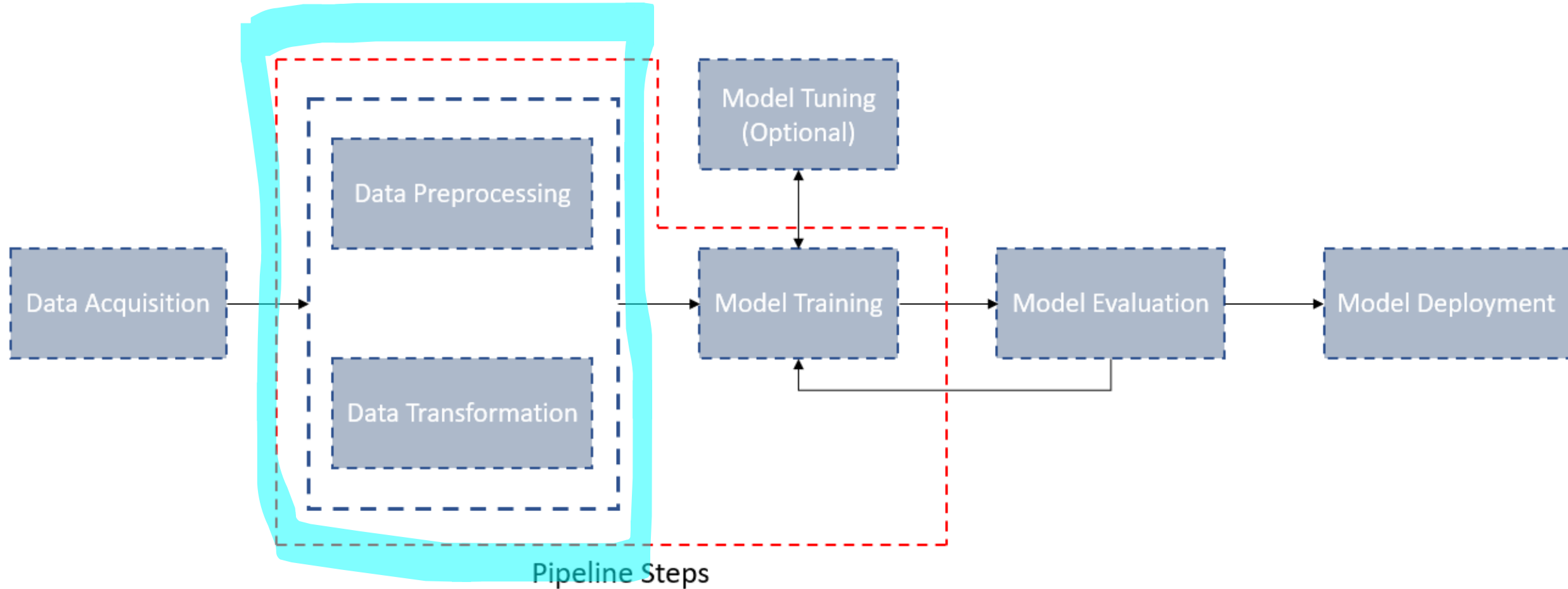


Data pre-processing Feature Engineering and Model Fitting

How to make
your ML experiments work on real data

In part from: [LINK](#)

The ML pipeline



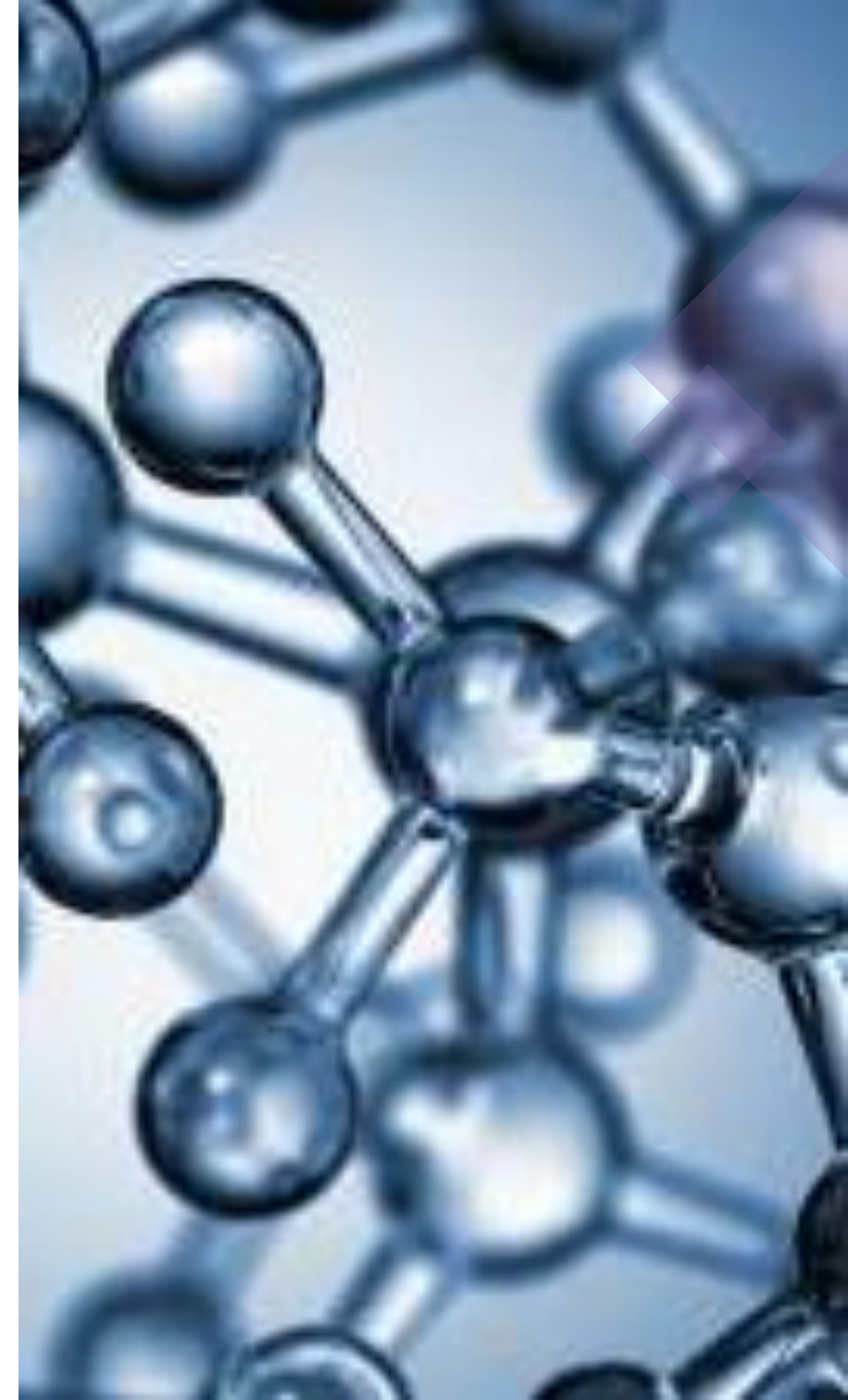
Issues with data

What data? Where can I find it?

What type of data? Several classification dimensions (sequential/non sequential; symbolic/continuous; structured/unstructured)

How to represent/transform my data (especially for unstructured data, such as images and texts)?
→ feature transformation and engineering

How to «clean» my data: noise, missing elements, unbalanced distributions, feature selection and/or augmentation



Data pre-processing

So far we assumed input data to be available in some form (feature vectors, pixel matrixes, graphs, sequences of continuous or discrete values)

However finding the appropriate data for a given problem is a relevant task, not always data is available. Furthermore, data can be “not ready” for analysis and needs lots of cleaning and transformation (a process often called ETL, extraction transformation and load)

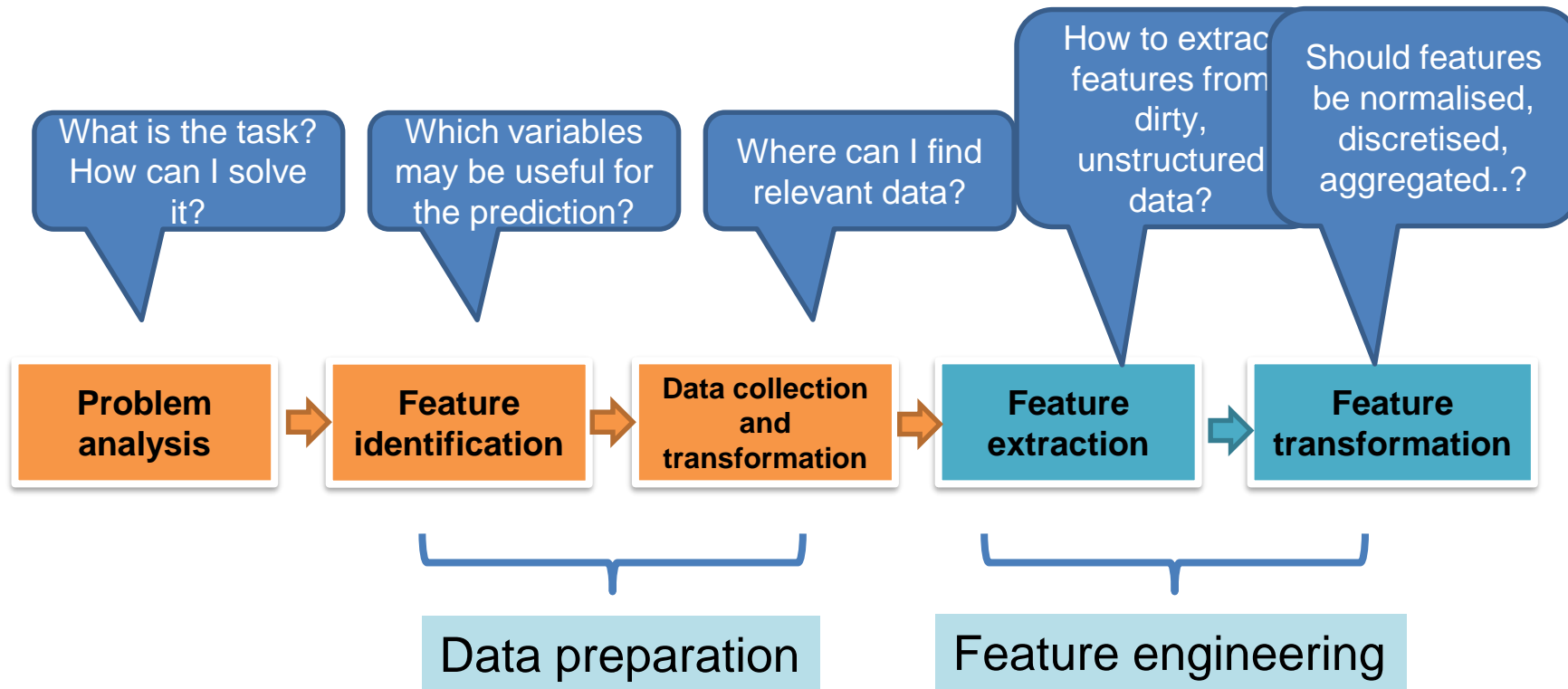
Therefore the responsibility of a machine learning engineer is:

- Setting up the correct problem to be solved/optimized (this is far from straightforward in the real world, often requires expert knowledge depending on application domain)
- **Identifying/Designing/extracting/representing relevant variables (features) to predict the unknown variable(s)**
- **Finding relevant data from which features can be extracted**
- Choosing a learning algorithm (or a family of algorithms)

From scratch
or from
available
data?

- **Ex 1:** you have a problem (e.g., Huber wants to optimise the use of electric bikes in a city) but you don't know which types of data can be useful for your prediction and where to find them;
- **Ex 2:** you are given a dataset (e.g. patients health records on diabetes in a country) and a problem (early prediction of coronary complications). Are the attributes in your EHR dataset all useful? Do you need more/less data?

The data workflow



Problem analysis and feature identification

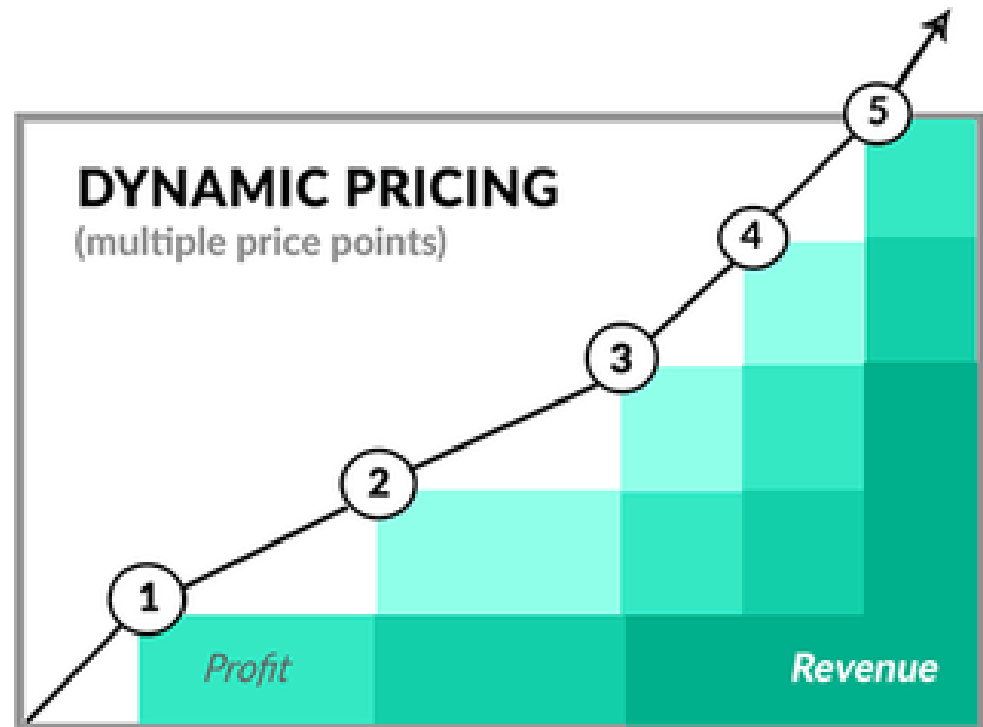
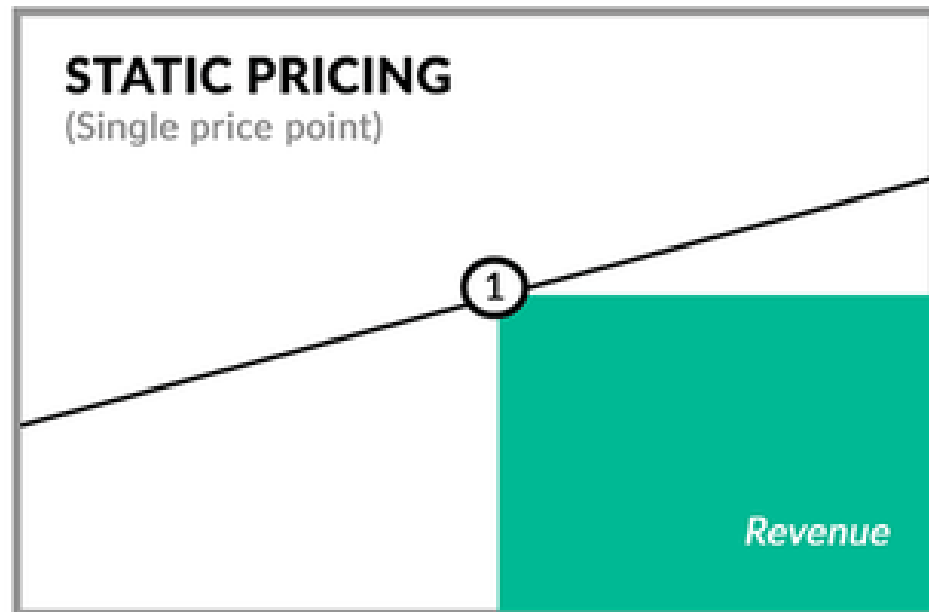
- Given a ML problem (e.g. image classification, patient disease prediction, predicting successful football players..) often the very first issue is: **Which kind of information would be helpful to accomplish the task? And where can I find it?**
- Often this is **preliminary** and more crucial than finding and processing the information, once available.
- Often there is not a SINGLE source of information, data may come from different sources, data can be
 - heterogeneous (structured, unstructured; csv, images, texts, signals)
 - semantically diverse (business data, social, meteorological, traffic, health...) and
 - available from different sources (legacy data, open data, web data..)

Example problem analysis 1: Uber bike sharing optimisation

Task: Surge pricing- These algorithms monitor traffic conditions and journey times in real-time to predict and suggest prices as demand for rides changes, and traffic conditions mean journeys are likely to take longer.

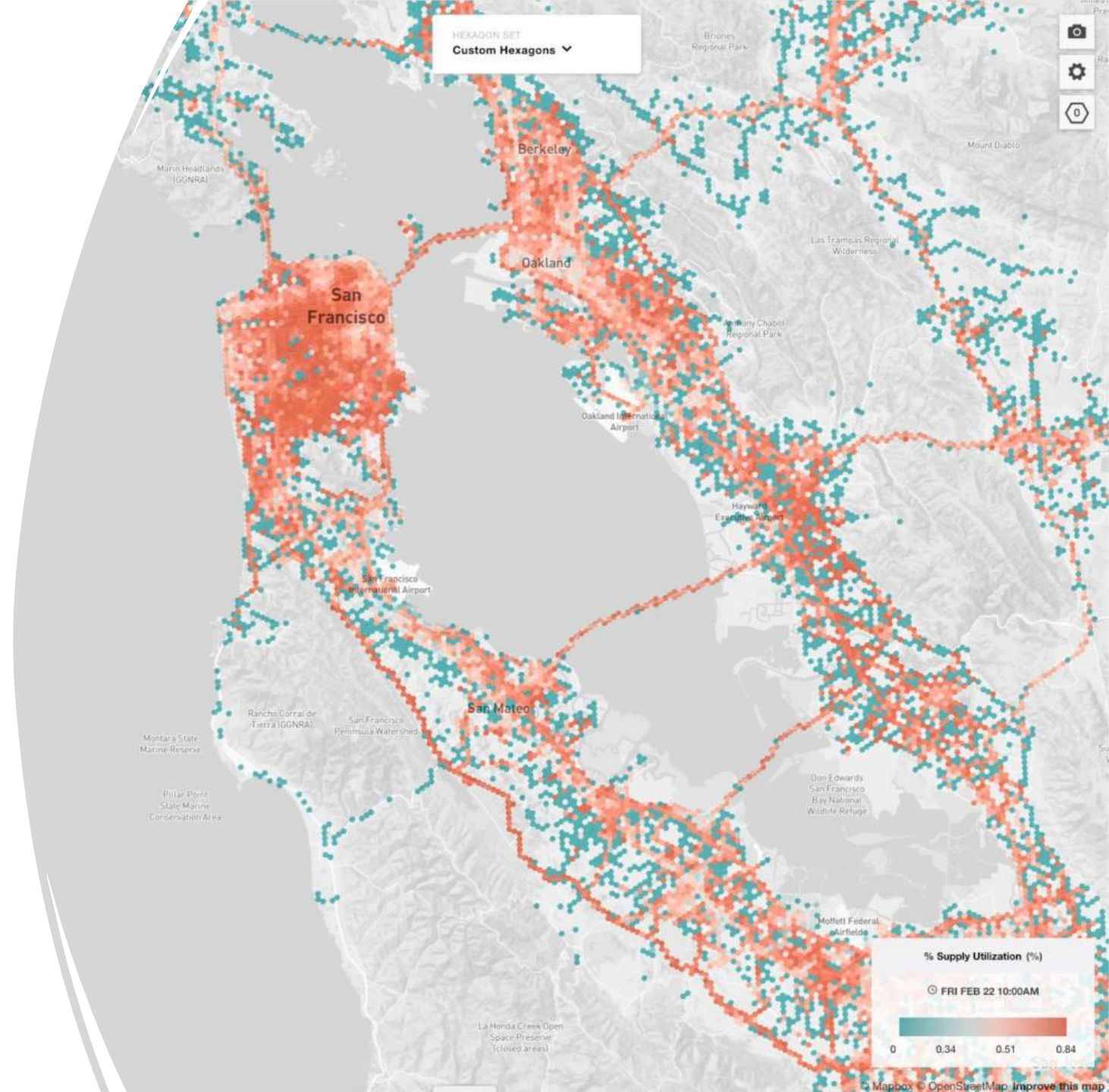
Which variable should be predicted?
Task is predicting the best price dynamically, optimizing global revenue

Dynamic pricing



Which data?

- Uber uses a mixture of legacy and external **data** to dynamically generate fares:
 - street traffic **data**,
 - GPS **data** (preferred routes by users)
 - external **data** like public transport routes and weather data
 - Historical data on users (from apps)
 - User opinions (from social networks)
- **Heterogeneous data**, from multiple sources: need preprocessing and integration before they can be transformed into some of the «known» formats (vectors, matrixes, sequences..)



Example problem analysis 2

- The **transfer fees** of football players are getting higher and higher each year.
- Can ML help in predicting these values?
 - **Problem 1 (feature identification):**
Which information can support the decision system?
 - **Problem 2 (data collection):**
Where do I find the data?
 - **Problem 3 (feature engineering):**
Are my data ready for processing?

Let's play: which features would you use?



Using domain
knowledge
and
open data
sources we
may come out
with this
(possible) list
of relevant
features:

- Features (for each player):
 - **Player's basic information:** team, age, height, weight.
 - **Market information:** transfer fee, former team, duration of the contract, when the player joined the team, . . .
 - **Performance information:** on-pitch time, actions at the ball, fouls, scores.
- But now the question is: **Where can we find this data?**
- Data sources: Transfer Market, WhoScored, European Football Database, and Garter

Finding good data sources

- Data can be legacy data of a company/entity
- **Lots of open source data are available on the web**
- Lots of data repositories on line (Kaggle perhaps the most well known, but many other)
- In the era of big data finding data is not the main problem. But we have another problem: real world data are NOT ready for use!

Using on-line info (ex, web data) is far from being easy (often you need scarpers..)

The screenshot shows the Whoscored website interface. At the top is a dark blue navigation bar with links: NEWS, TRANSFERS & RUMOURS, MARKET VALUES, COMPETITIONS, FORUMS, MY TM, and LIVE (with a red notification bubble containing the number 3). A LOGIN button is on the right. Below this is a filter bar with icons and dropdown menus for Home, England, Competition, Club, and Player. A large grey box in the center contains the text "Ad closed by Google" and two buttons: "Stop seeing this ad" and "Why this ad?". Below the ad is a "SPOTLIGHT" section with the subtext "Latest transfers · What's happening today?". The first article, dated "05.10.2018 - 12:26", is titled "Foden, Nelson and Barnes named in England Under-21 squad" and features a photo of Phil Foden. The text of the article states: "Manchester City midfielder Phil Foden has been handed his first England Under-21 call up. The 18-year-old is joined by Arsenal's Reiss Nelson – on loan at Hoffenheim – and Leicester's Harvey Barnes, currently on loan at West Brom, as squad first-timers. The trio are ...". A "Read More" link is visible. The second article, dated "04.10.2018 - 15:26", is titled "Dortmund Youngster" and features a photo of a young man. To the right of the articles is a video player showing a crowd of people, with a "W HOTELS" logo and the text "LIVE FROM #WakeUpCallFest" at the bottom.

Whoscored: <https://www.google.com/>

Raw data: The dream

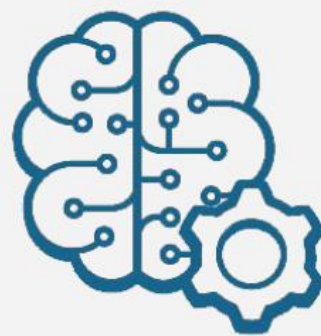
The Dream...



Raw data



Dataset

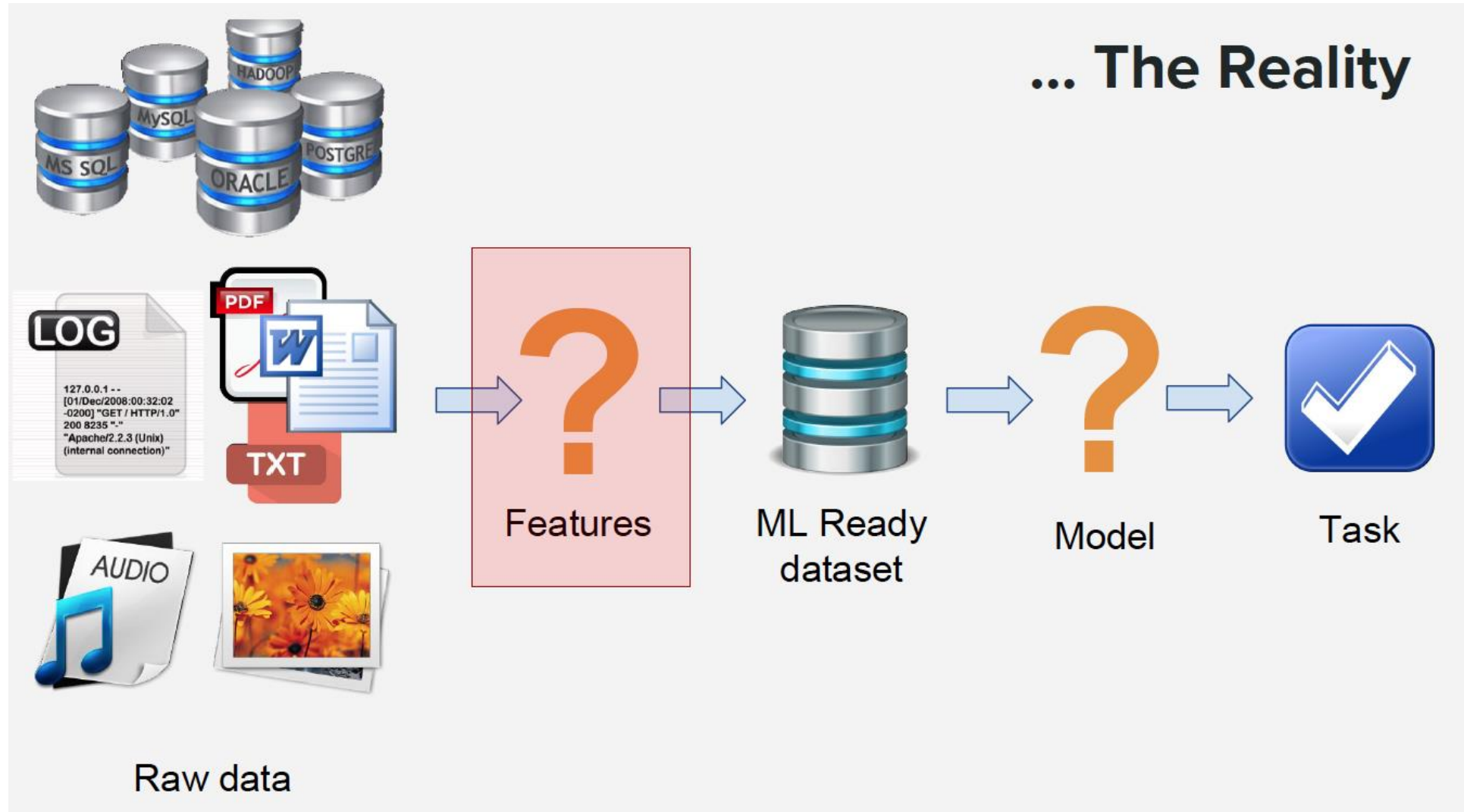


Model

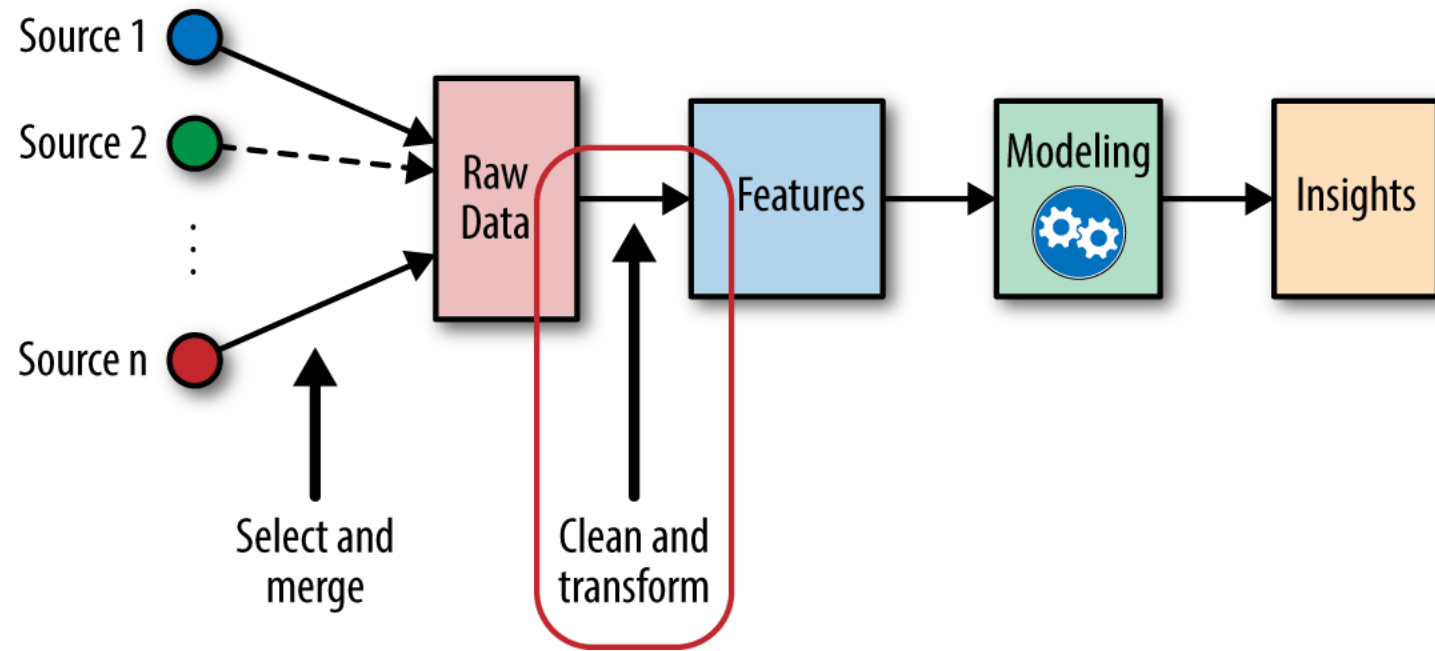


Task

Raw data: The reality



A lot of cleaning and transformation is needed before data are truly useful



Machine learning systems are only as good as the data you feed it

Feature Engineering: 3 related tasks

Feature Extraction:
Transformation of raw data (e.g, text) into features suitable (e.g., numbers) for processing

Feature Transformation:
Transformation of data to improve the accuracy of the algorithm (e.g., normalization, scaling..)

Feature Selection:
Removing unnecessary features

Feature extraction

- In practice, data rarely comes in the form of ready-to-use feature-value matrices (as for the decision-tree and perceptron examples).
- That's why every task begins with **feature extraction**. Sometimes, it can be enough to read the CSV file and convert it into an array, but this is a rare exception.
- Popular types of data from which features can be extracted:
 - Texts
 - Images
 - Geospatial data
 - Date and time
 - Time series, web data, etc.

1. Text

- **Text is pervasive: web pages, social media messages, news, diagnostic reports, release notes..**
- The first step is **tokenization**, i.e., splitting the text into units (hence, tokens).
 - “Before working with text, one must tokenize it”
before,working,with,text,one,must,tokenize,it
- Next, stemming or lemmatization to **normalize** tokens (recent approaches avoid stemming):
 - befor,work,with,text,one,must,stem,it
- Finally, **text encoding** (bag of words is the simplest):
 - Build a vocabulary over all words in all documents (now *dense* word representations are used, called embeddings)
 - Encode every document in a **sparse vector** d_i where $d_{ij}=1$ iff word j of vocabulary is in d_i , else $d_{ij}=0$

1. Text:

Example of text encoding

"This is how you get ants."

tokenizer

From free text to a list of tokens

['this', 'is', 'how', 'you', 'get', 'ants']

Create a vocabulary ordering all tokens

Build a vocabulary over all docum

['aardvak', 'amsterdam', 'ants', ..., 'you', 'your', 'zyxst']

Sparse matrix encoding

The «simplest» document encoding strategy assigns a binary value for presence absence of a word in the document

aardvak	ants	get	you	zyxst
[0, ..., 0, 1, 0, ... , 0, 1 , 0, ..., 0, 1, 0,	0]			

1. Text:

single words can be assigned a vector representation, so-called **one hot encoding**

The cat sat on the mat

The: [0 1 0 0 0 0 0]

cat: [0 0 1 0 0 0 0]

sat: [0 0 0 1 0 0 0]

on: [0 0 0 0 1 0 0]

the: [0 0 0 0 0 1 0]

mat: [0 0 0 0 0 0 1]

Dimension of the vector equals the dimension of the vocabulary. Here we assume a simple 7-words vocabulary

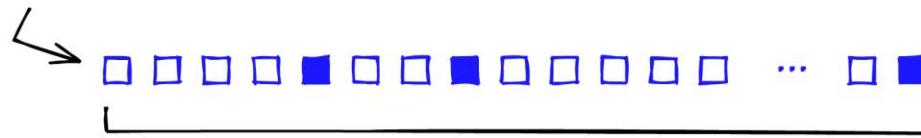
1. Text: Embeddings

- State of the art approach for text representation is **word embeddings**
- More in NLP courses, however, the idea is that **words**, rather than being represented as a binary value (or a real value, or a binary vector) in a “sparse” document space with $|V|$ dimensions, are represented as “dense” numeric vectors in a “reduced” semantic space
- Words with “close” vectors are (semantically) similar
- A [survey](#) on word embedding methods

Sparse and dense representations

sparse

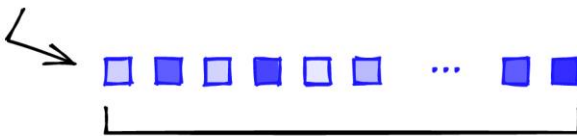
$[0, 0, 0, 1, 0, \dots 0]$



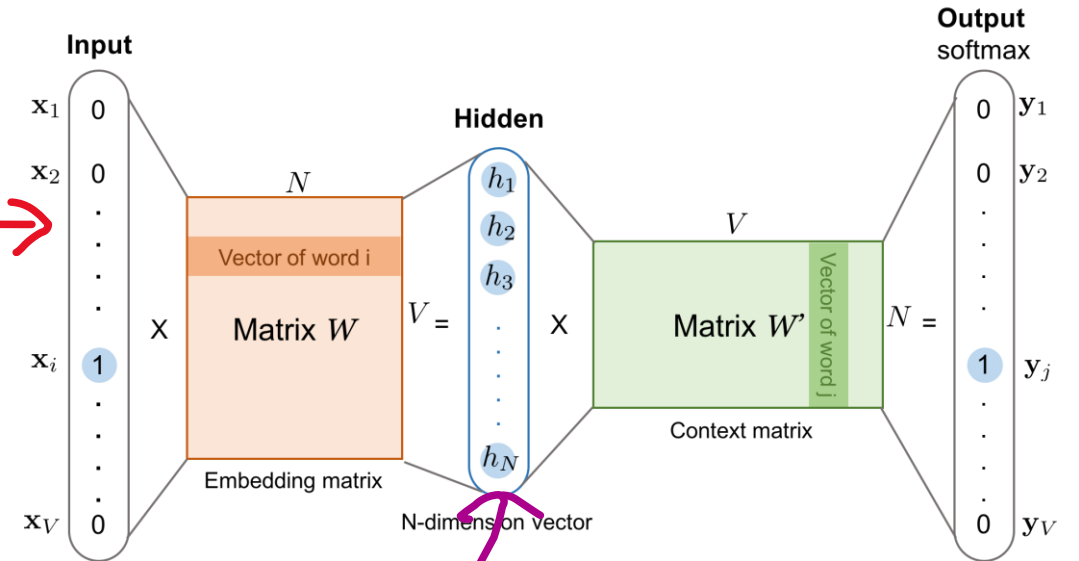
30K+

dense

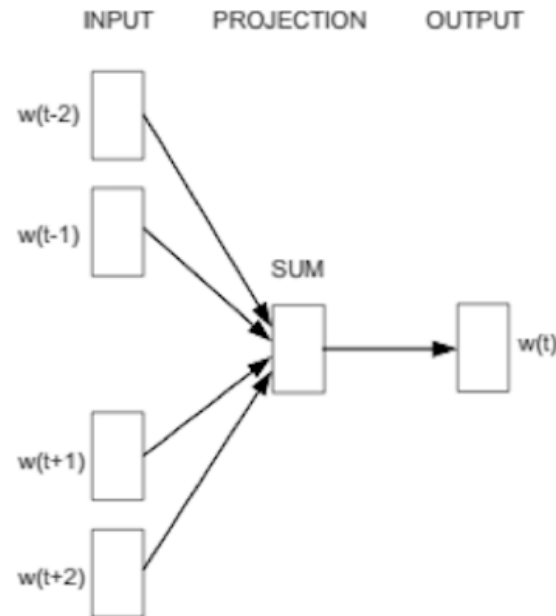
$[0.2, 0.7, 0.1, 0.8, 0.1, \dots 0.9]$



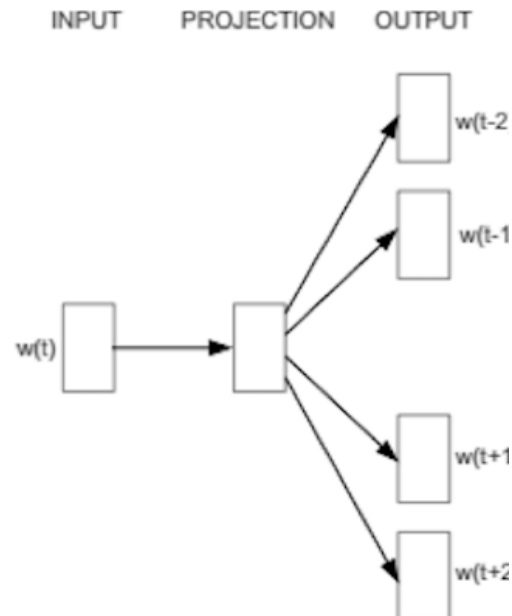
784



Word2vect embeddings: 2 training methods

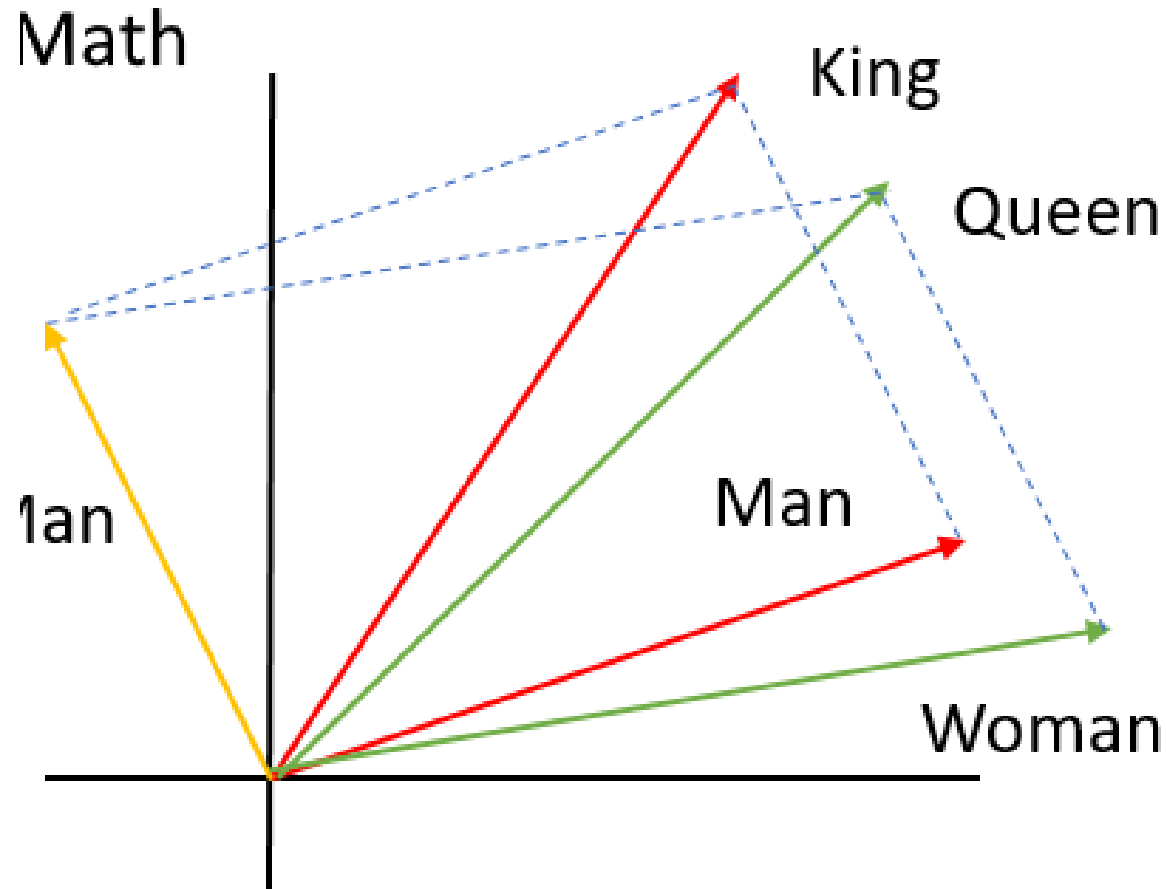


CBOW



Skip-gram

- CBOW: given a context, learn predicting a word;
- Skip-gram: given a word, learn predicting the context
- In both cases, the result of training is an «encoder» able to associate to word's one-hot vectors a «dense» representation, named **embedding**.



Dense
representations
capture the
«essence» of
word meanings

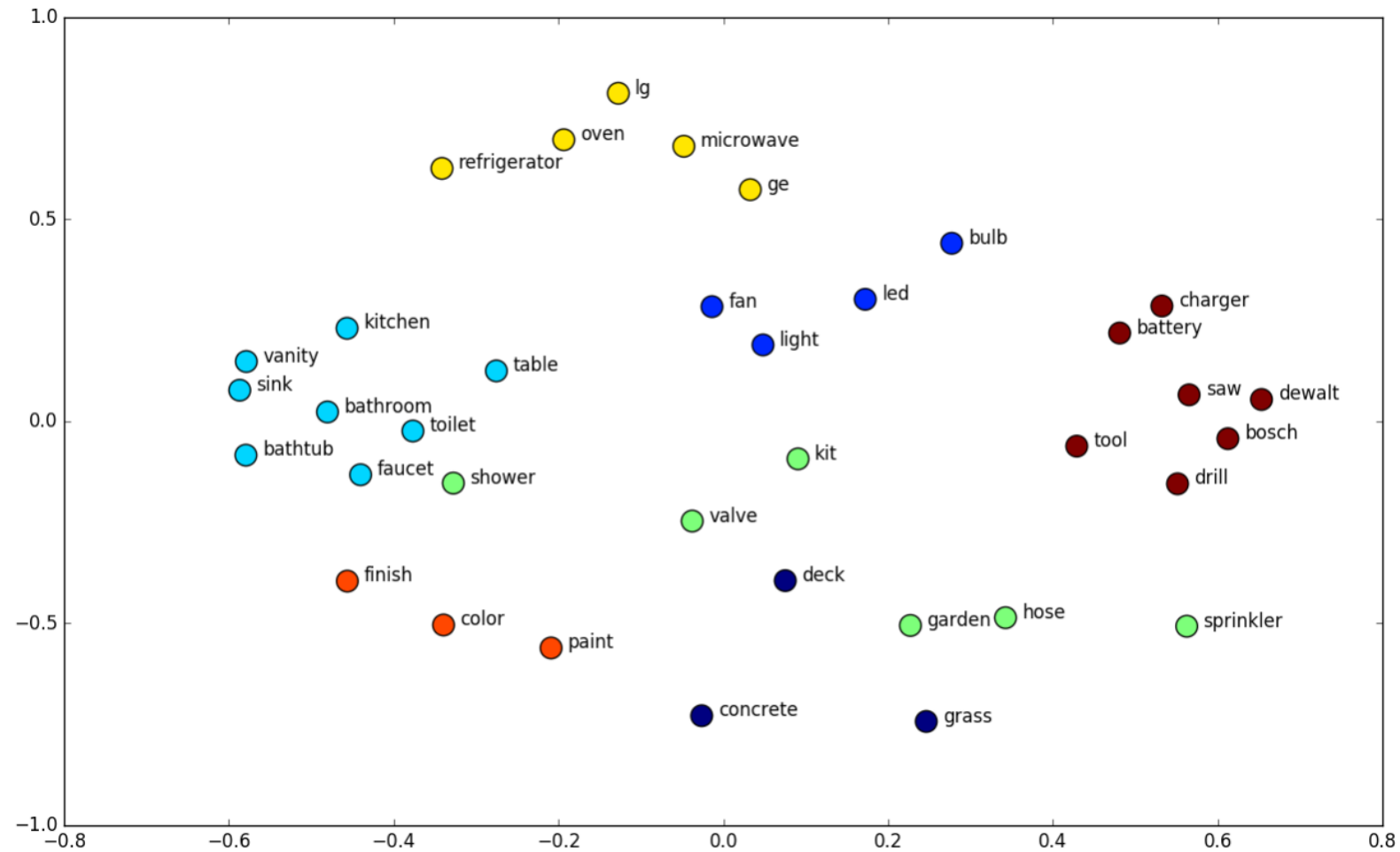
Semantically similar words
are closer in a “latent” space

Words are “projected” onto latent semantic spaces

- The «dense» dimensions are “latent” (hidden) and learned by looking at word contexts. However, the meaning of dimensions is not explicit! (black boxes)
- **NOTE:** We are **unaware** that dimension 1 is, e.g., “royalty”.
- Learned dimensions depend on **the source texts used for learning** – kings, queen, and princesses have different vectors if learned from fairy tales or gossips newspapers!



Example of embeddings

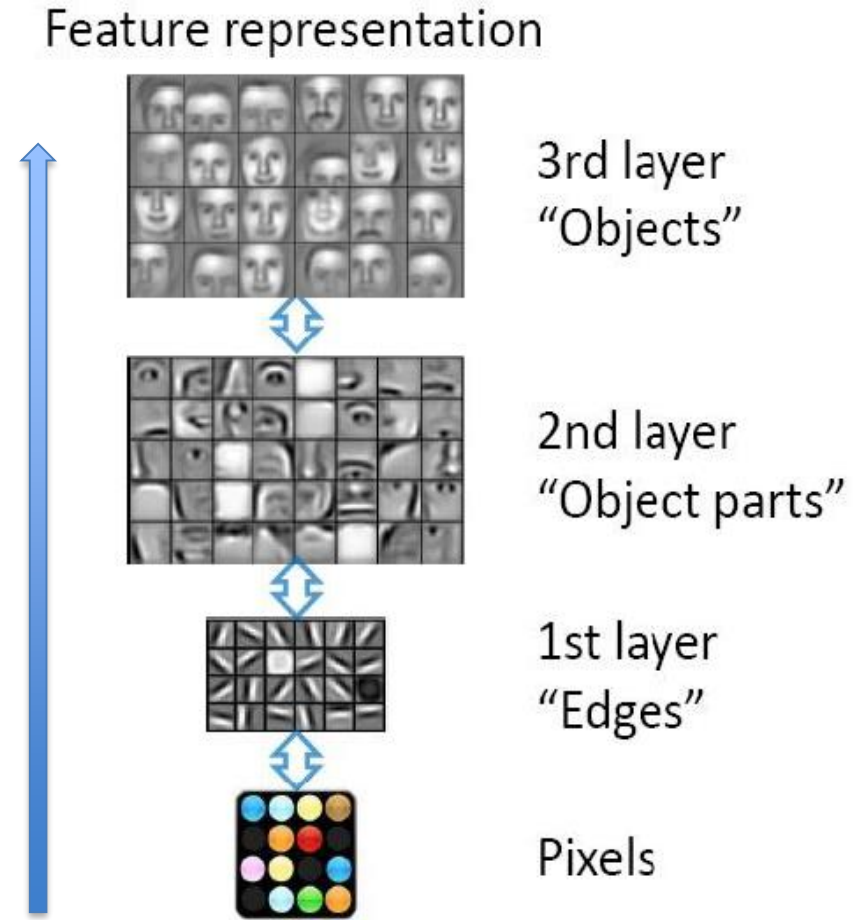


More on embeddings

- Embeddings are now widely applied not only to textual data but to any vectorial representation of data (including images and graphs)
- They capture «latent» similarities in the data and allow better generalization during the learning phase
- They also «compress» the representation of data items since they project surface features into a «denser» semantic space (see later on dimensionality reduction)
- More [here](#)

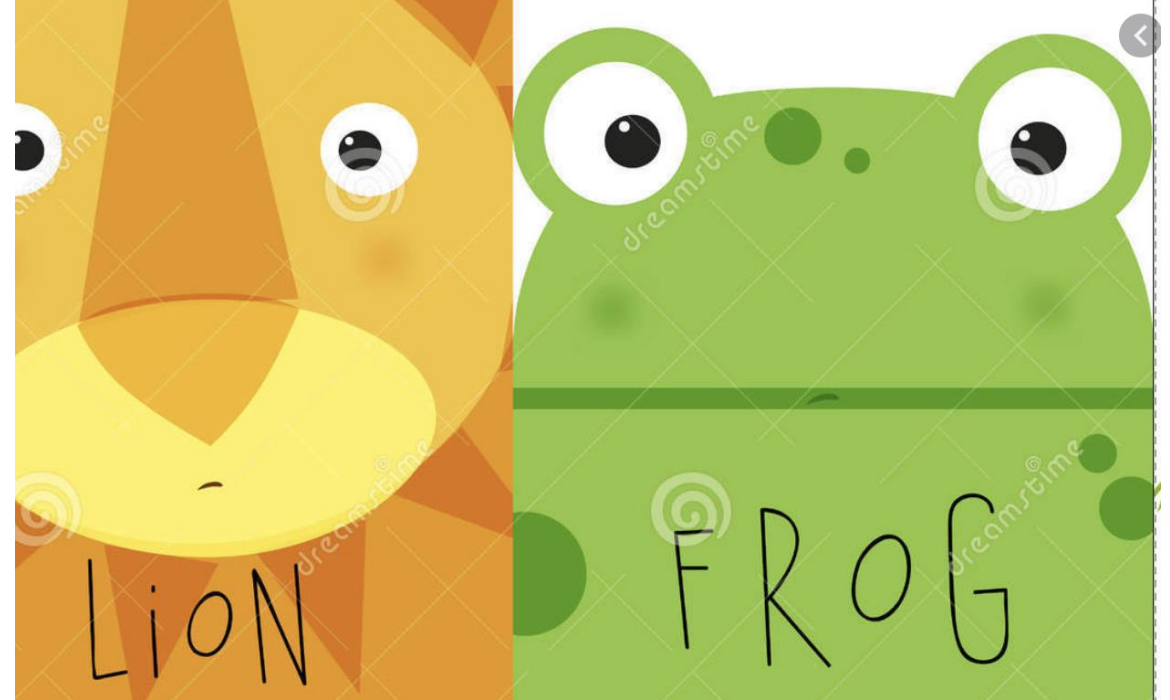
2. Images

- Images are usually represented at the pixel level
- Note that the entire pipeline of a CNN (except for the final classification layer) can be regarded as a way to compress the features of an image into a more compact representation.
- Even in this case, the latent “semantics” discovered by hidden layers is not available!



2. Images

- Nevertheless, we should not focus too much on neural network techniques. Simpler features are still very useful for image representation
- For example, to «predict» if an image represents a lion or a frog, a chromaticity histograms is more than enough!





3. Geospatial data

3. Geospatial Data

- Geospatial data are very useful in many applications where location is relevant (e.g. recommender systems, and many other location-dependent optimization problems, e.g., supply chain management)
- Geospatial data is often presented in the form of addresses or coordinates (latitude, longitude)
- Depending on the task, you may need two mutually-inverse operations:
 - Geocoding (recovering a point of interest from an address)
 - Reverse geocoding (recovering an address from a point).
- Both operations are accessible in practice via external APIs from Google Maps or OpenStreetMap.

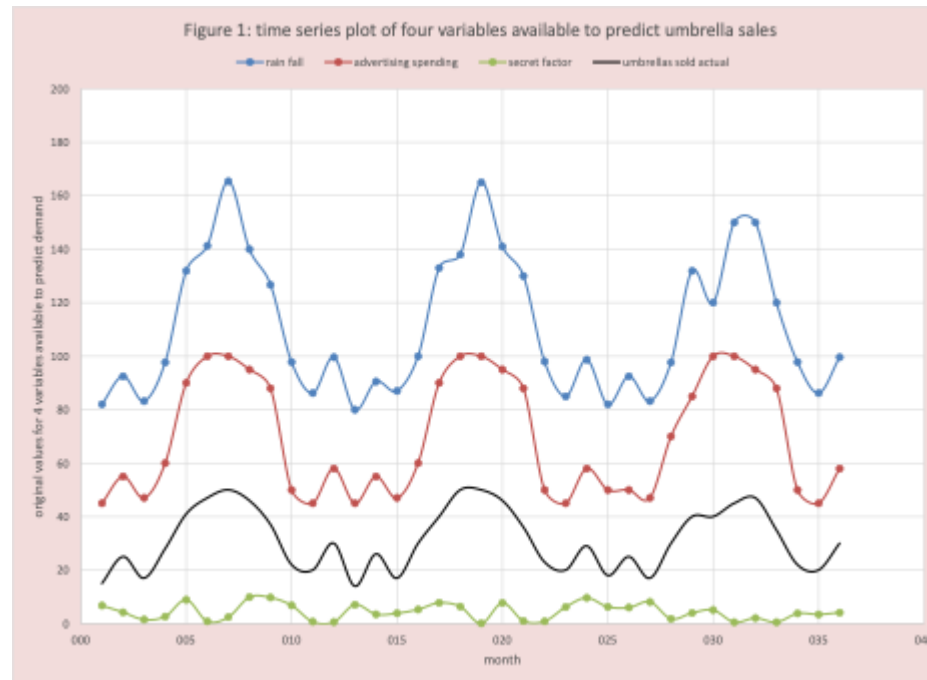
3. Geospatial Data

CAVEAT:

- Textual addresses may contain typos, which makes the data cleaning step necessary (see later).
- Coordinates contain fewer misprints, but its position can be incorrect due to **GPS noise** or bad accuracy in places like tunnels, downtown areas, etc.
- If the data source is a mobile device, the geolocation **may not be determined by GPS** but by WiFi networks in the area. While traveling along in Manhattan, there can suddenly be a WiFi location from Chicago.

4. Time series

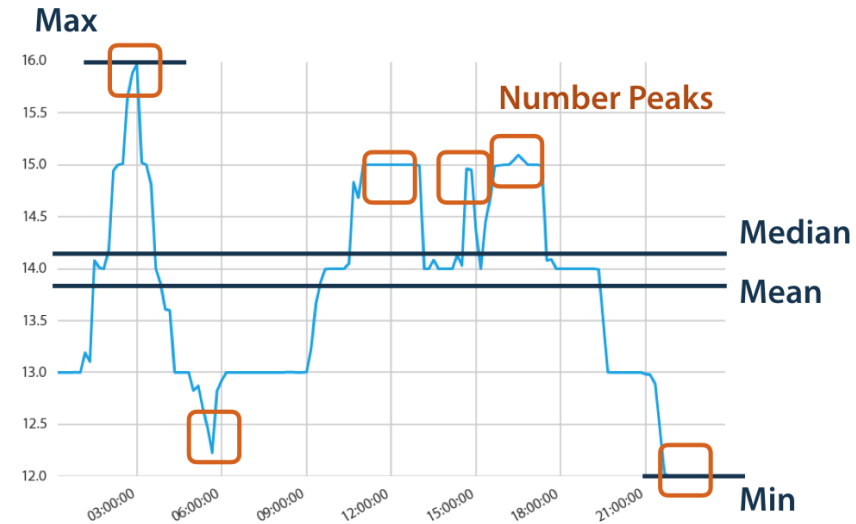
- **Sequential data are quite common** (e.g. stock market data, personal patient records (trajectories), sensor data..)
- In many cases, before you can feed your ML system with time series data, you need to **eliminate trends**, to average over selected time spans, and to **normalize** (especially if you have different types of time series in your data).



Example: these time series of «umbrella sales» behave similarly, however without normalization, the similarity cannot be captured.

4. Times series: extracting features

- There are libraries ([link](#)) to automatically extract a large number of time series *features*.



5. Other domains

In other domains, you can come up with your features based on intuition about the nature of the data, based on available information, and the classification/regression task that has been set

But almost NEVER your data are «ready-to-use»!



Feature Engineering: 3 related tasks

Feature Extraction:
Transformation of raw data (e.g, text) into features suitable (e.g., numbers) for processing

Feature Transformation:
Transformation of data to improve the accuracy of the algorithm (e.g., normalization, scaling..)

Feature Selection:
Removing unnecessary features

Feature transformation methods



1. Normalization

- Scaling and centering, Change of bases, Categorical to numeric...

2. Missing values

- Removal, regression imputation, k-neares neighbours...

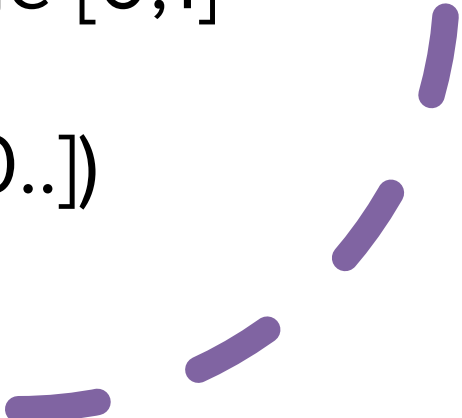
3. Data augmentation (add more features)

4. Imbalanced categories

- Oversampling, undersampling, smote, anomaly detection, cost-sensitive learning

A large red circle on the left side of the slide, partially cut off by the edge.

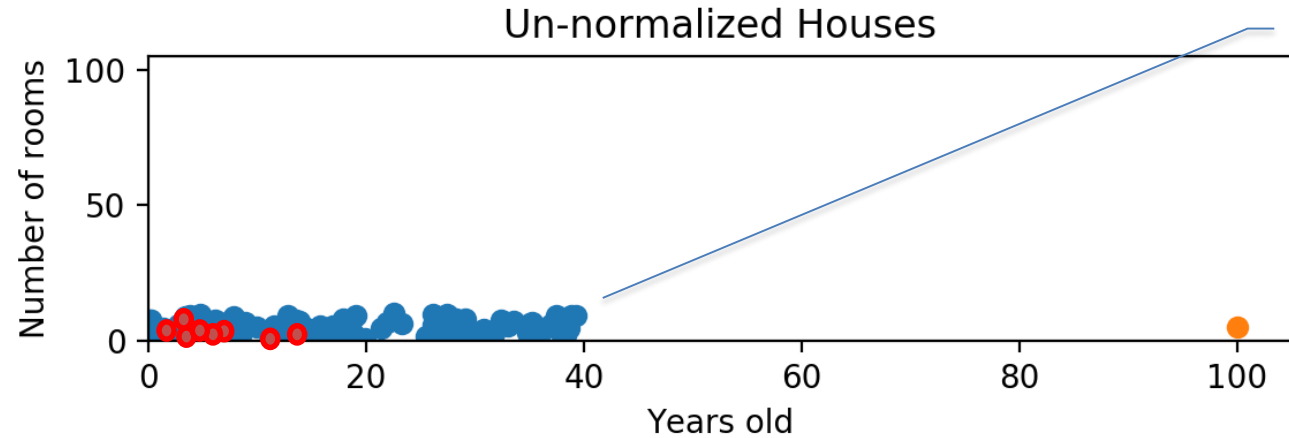
Normalization and changing distribution

- Certain algorithms –and platforms– require a specific format for data (E.g., **decision trees allow for categorical data**, other methods do not)
 - Similarly, some algorithms suffer for **unbalanced** scaling of features (e.g. one feature with range $[0,1]$ and others with range $[-10000.. +10000..]$)
- 
- A decorative purple dashed line in the bottom right corner of the slide.

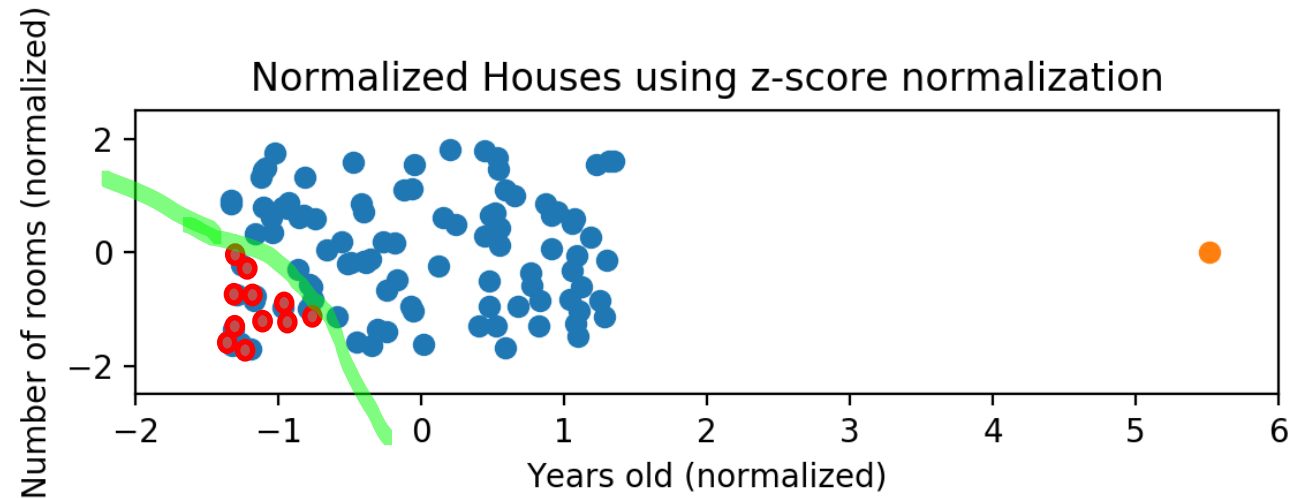
Normalization: Scaling & Centering

- The reason for centering and scaling is that it places all features **on equal standing**.
- Some ML algorithms project instances onto a multi-dimensional space and examine the distances between different data points (e.g., clustering). In such methods, features **with large absolute differences** in values will be more important (will “affect” more than others the computation of distance).
- Yet generally such absolute differences in values **reflects nothing more than the metric chosen to measure the variable**, and *a priori* it is unreasonable that one variable should be more important than others

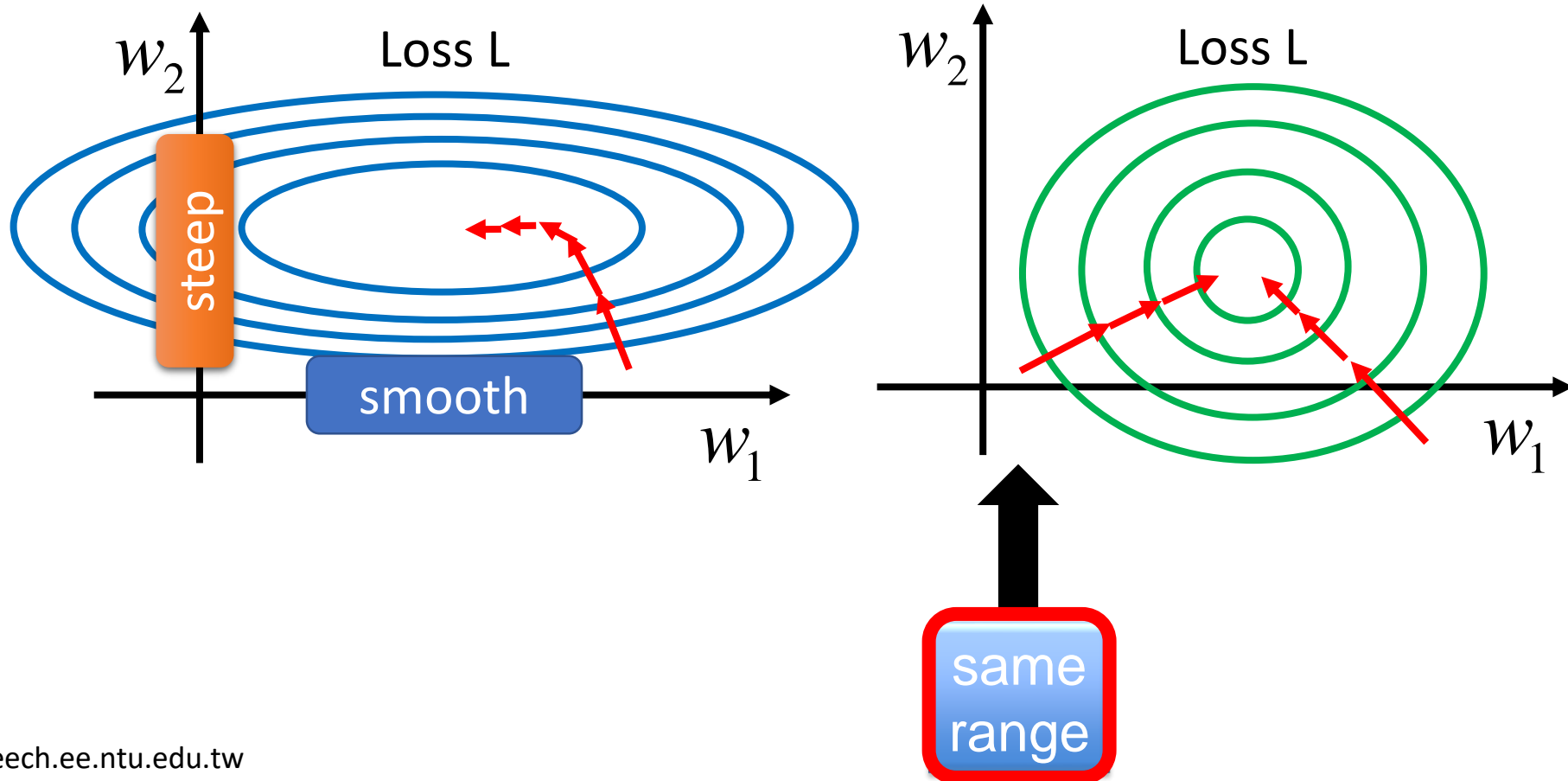
Example: predicting the sell price of houses



Here, the feature “number of rooms” does not allow any useful separation between datapoints



Feature normalization also helps gradient descent converge faster!



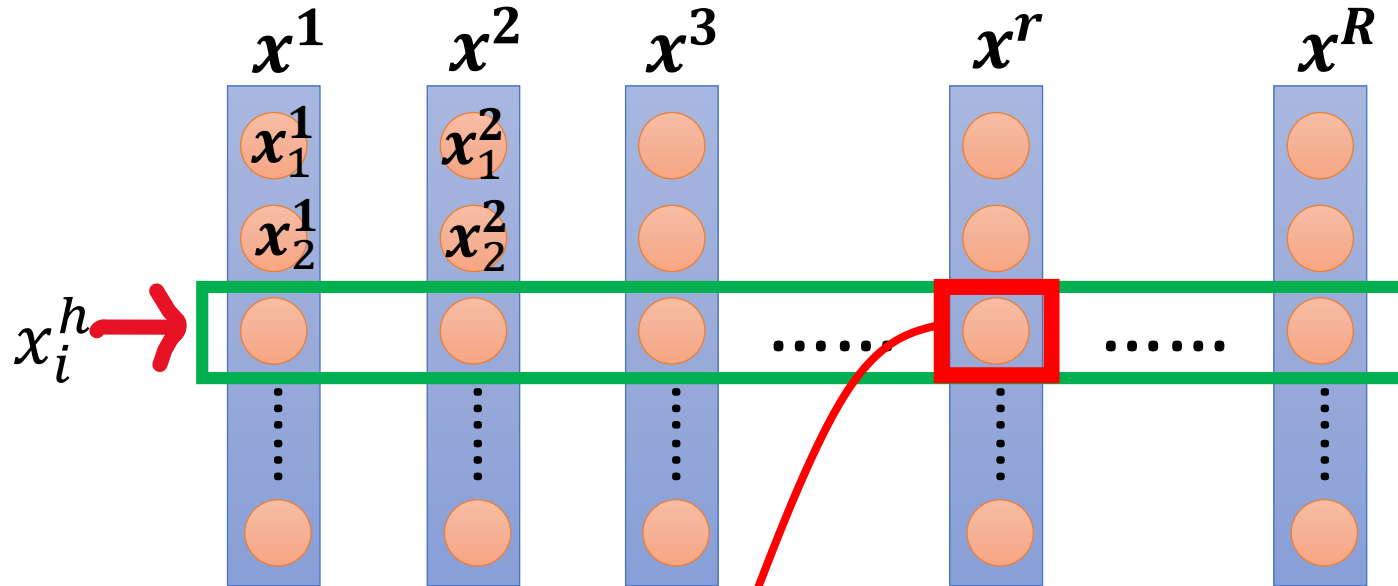
Why gradient converges faster with normalized features

- Remember the «basic» weight updating rule:

$$\Delta w_{ij} = \eta \frac{\partial Loss}{\partial w_{ij}} = \eta \delta_j \frac{\partial (net_j)}{w_{ij}} = \eta \delta_j x_i$$

- Larger signals travelling on a synaptic connection cause greater updates
- This applies **both** to input features and to the features computed in the internal layers of a deep NN

Feature Normalization



For all feature values x_i^h in D :

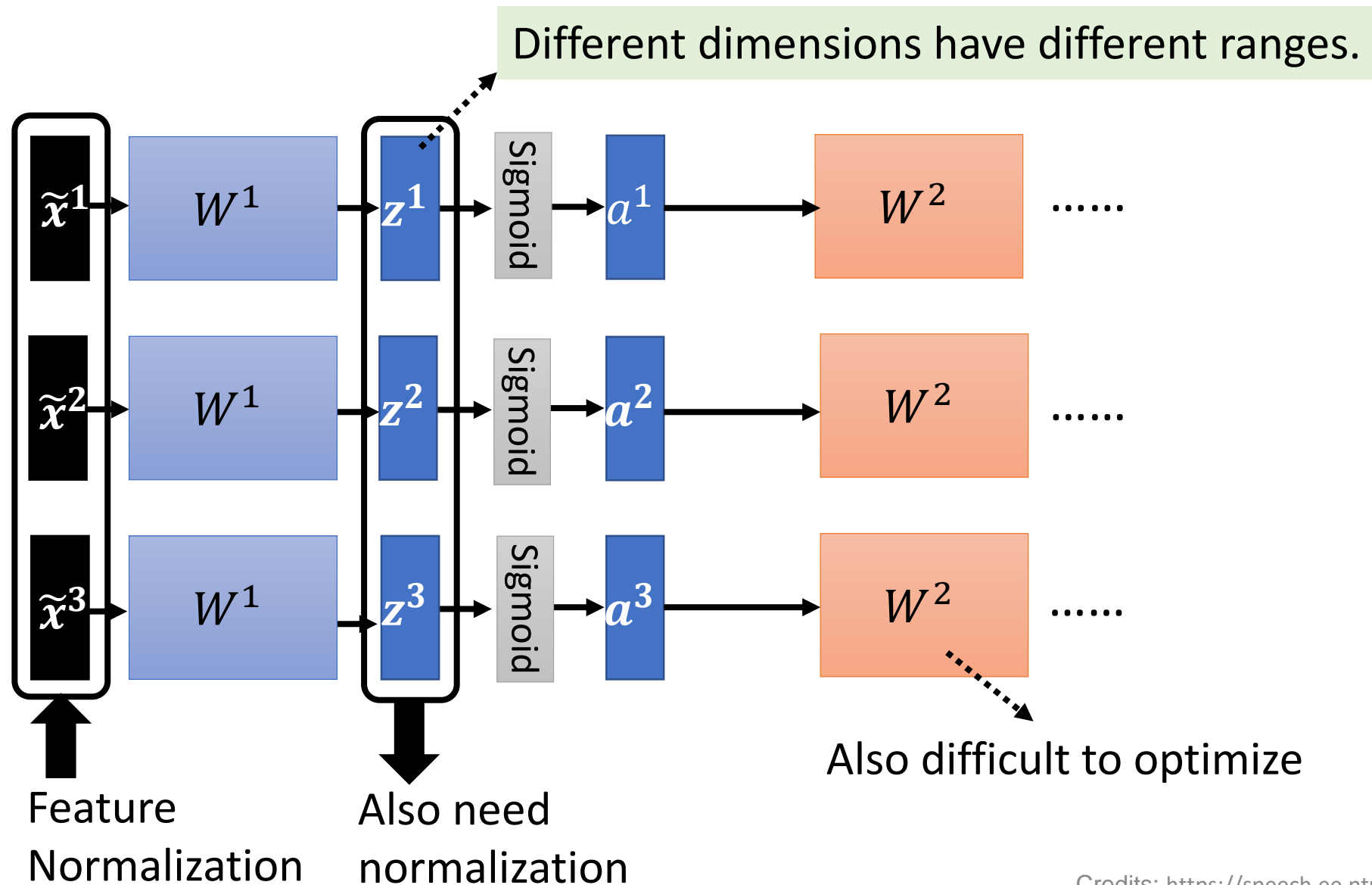
- compute mean: μ_i
- compute standard deviation: σ_i

$$\tilde{x}_i^h \leftarrow \frac{x_i^h - \mu_i}{\sigma_i}$$

With normalized values, the means of all dimensions i are 0, and the variances are all 1

Note: we are «forcing» a gaussian distribution of all features

In deep architectures normalization is needed also in the internal layers

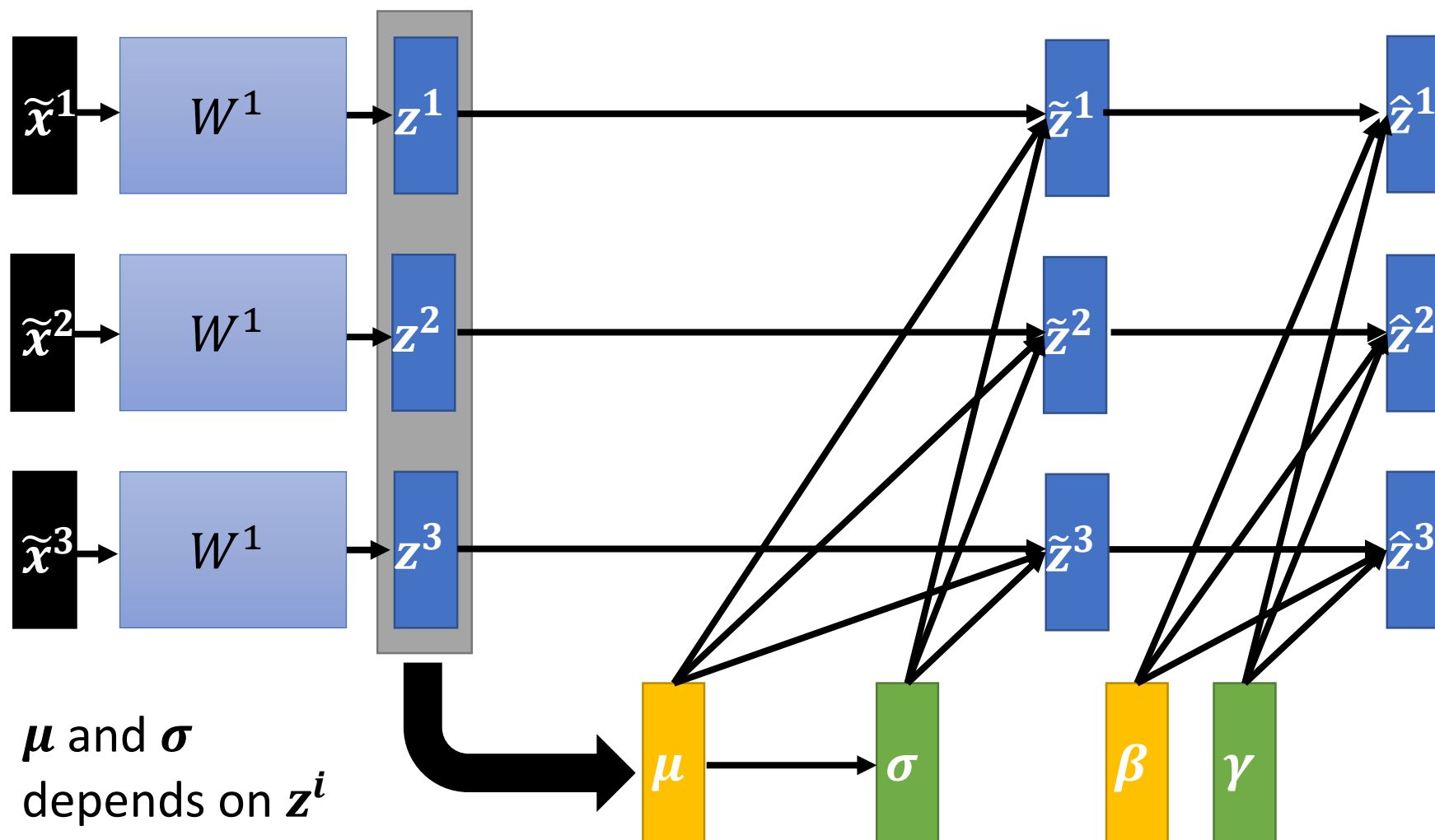


Batch normalization (2 steps)

$$\tilde{z}^i = \frac{z^i - \mu}{\sigma}$$

β and γ hyperparameters

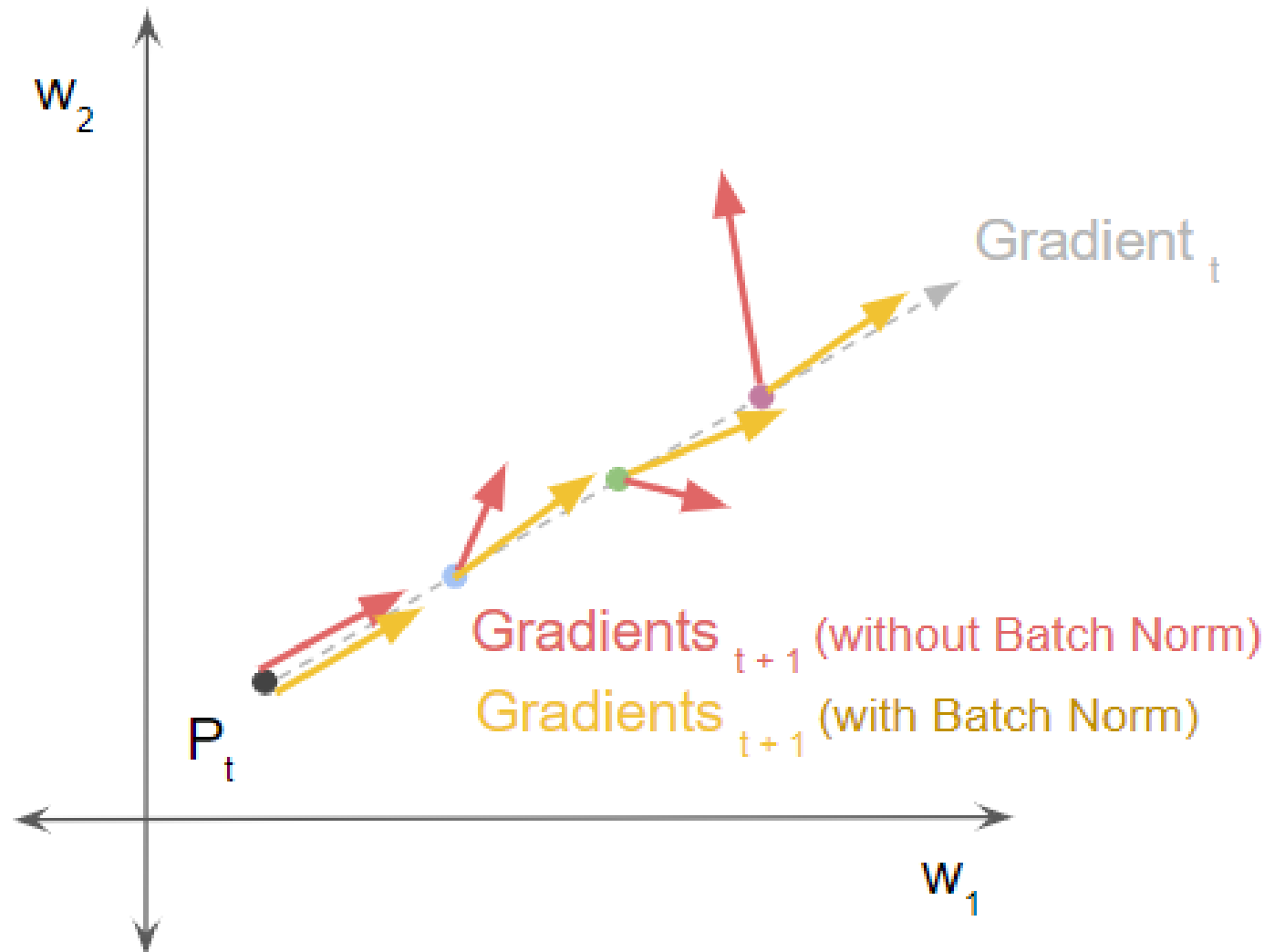
$$\hat{z}^i = \gamma \tilde{z}^i + \beta$$



Why batch normalization works

- It normalizes not only the input features but also further values in the **hidden units** to take **on a similar range of values** that can speed up learning (faster gradient descent, as we have seen).
- The second reason why **batch norm** works, is it makes weights, later or deeper in the network you have, more *robust to changes of weights* in earlier layers of the neural network (eg. in layer one).
- What batch norm ensures is that no matter how the parameters of the neural network update, **their mean and variance will at least stay the same** mean and variance, causing the input values to become more stable

Gradients
with batch
norm are
smoother

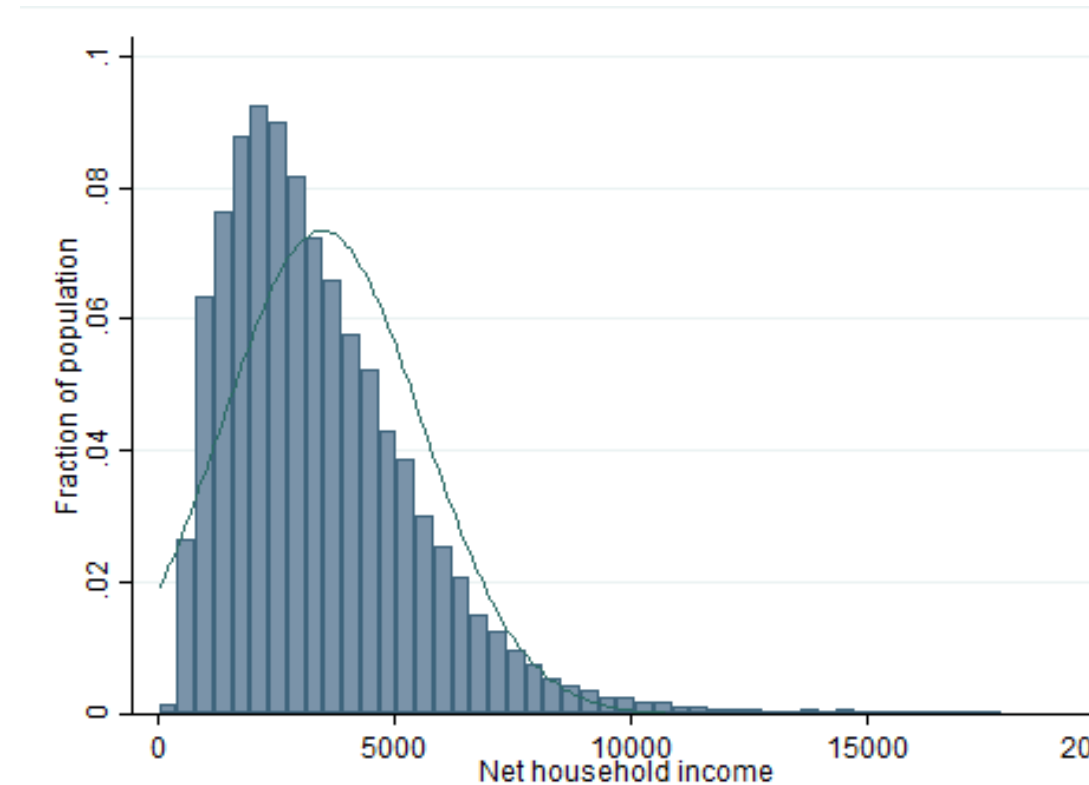


Normalization: Changes of Bases

- Input features values are usually distributed according to some distribution, e.g., a normal (Gaussian) distribution for continuous variables
- The “skewness” is an **asymmetry** in a statistical distribution, in which the curve appears distorted or skewed either to the left or to the right
- Skewness can be quantified to define **the extent to which a distribution differs from a normal (Gaussian) distribution**

Skewed data example


- If data are «skewed» the tail region may act as an «outlier» for the model and outliers may adversely affect the model's performance, especially regression-based models - since the «most common» data may no longer be around the mean.



Outliers: instances that are significantly "diverse" from average population

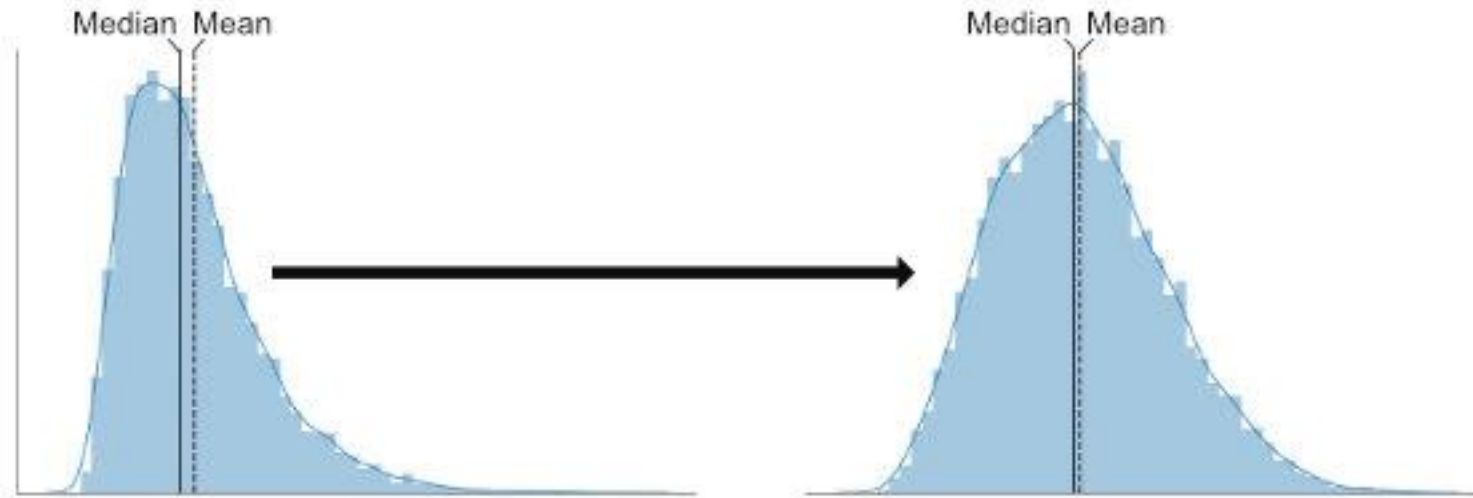
A large red circle on the left side of the slide, partially cut off by the edge.

Normalization: Changes of Bases

- To reduce the **skewness of the distribution of a feature's values in a dataset**, we can perform a log transformation
 - For more “sparse” distributions, other more complex methods are possible (e.g. *qqnorm*).
- 
- A decorative purple dashed line in the bottom right corner, consisting of several short, curved segments.

Example

Transforming non-normal data



The original distribution of values
and the distribution after applying
a log transformation

Normalization: Categorical into numeric

- Certain features can take categorical values (e.g., Spotify dataset: *artist, track name*..).
- Categories may be nominal (sport, politics, finance..) or ordinal (e.g., dates or weekdays). Ordinal levels follow a logical order. In nominal categories often there is no order (e.g., city names)
- Some algorithms **do not accept categorical data**, therefore we need some transformation.

Normalization:

Categorical into numeric

- **One-hot encoding** is the default way of turning categorical data into numeric. With this method, we encode the categorical variable as a **one-hot vector**, i.e. a vector where only one element is non-zero, or "*hot*".
- With one-hot encoding, a categorical feature becomes an array whose size is the number of possible choices for those features. With N values, the dimension of the vectors is N

One hot encoding

color	color_red	color_blue	color_green
red	1	0	0
green	0	0	1
blue	0	1	0

Normalization: Categorical into numeric

- However, if N is large, one-hot encoding may be a bad idea.
- Another approach to encoding categorical values is to use a technique called **label encoding**. Label encoding is simply converting each categorical **value** to a number
- But, in those algorithms where the “weight” of each attribute value matters (regressors), label encoding introduces an ***unjustified bias towards higher values***
- An intermediate alternative is **label binarization** which introduces $\log_2(N)$ values.

Example

(label and binary encoding)

The diagram illustrates the process of encoding categorical data. It features a table with two main sections: 'Categorical Feature' and 'Binary Encoded'. The 'Categorical Feature' section lists names and their corresponding label-encoded values. The 'Binary Encoded' section shows the binary representation of these labels. Two orange boxes with arrows point to the table: 'label encoding' points to the label values, and 'Binary encoding' points to the binary columns.

			Binary Encoded			
Categorical Feature =			x1	x2	x4	x8
Louise	=>	1	1	0	0	0
Gabriel	=>	2	0	1	0	0
Emma	=>	3	1	1	0	0
Adam	=>	4	0	0	1	0
Alice	=>	5	1	0	1	0
Raphael	=>	6	0	1	1	0
Chloe	=>	7	1	1	1	0
Louis	=>	8	0	0	0	1
Jeanne	=>	9	1	0	0	1
Arthur	=>	10	0	1	0	1

Feature transformation methods

1. Normalization

- Scaling and centering, Change of bases, Categorical to numeric...

2. Missing values

- Removal, regression imputation, k-nearest neighbours...

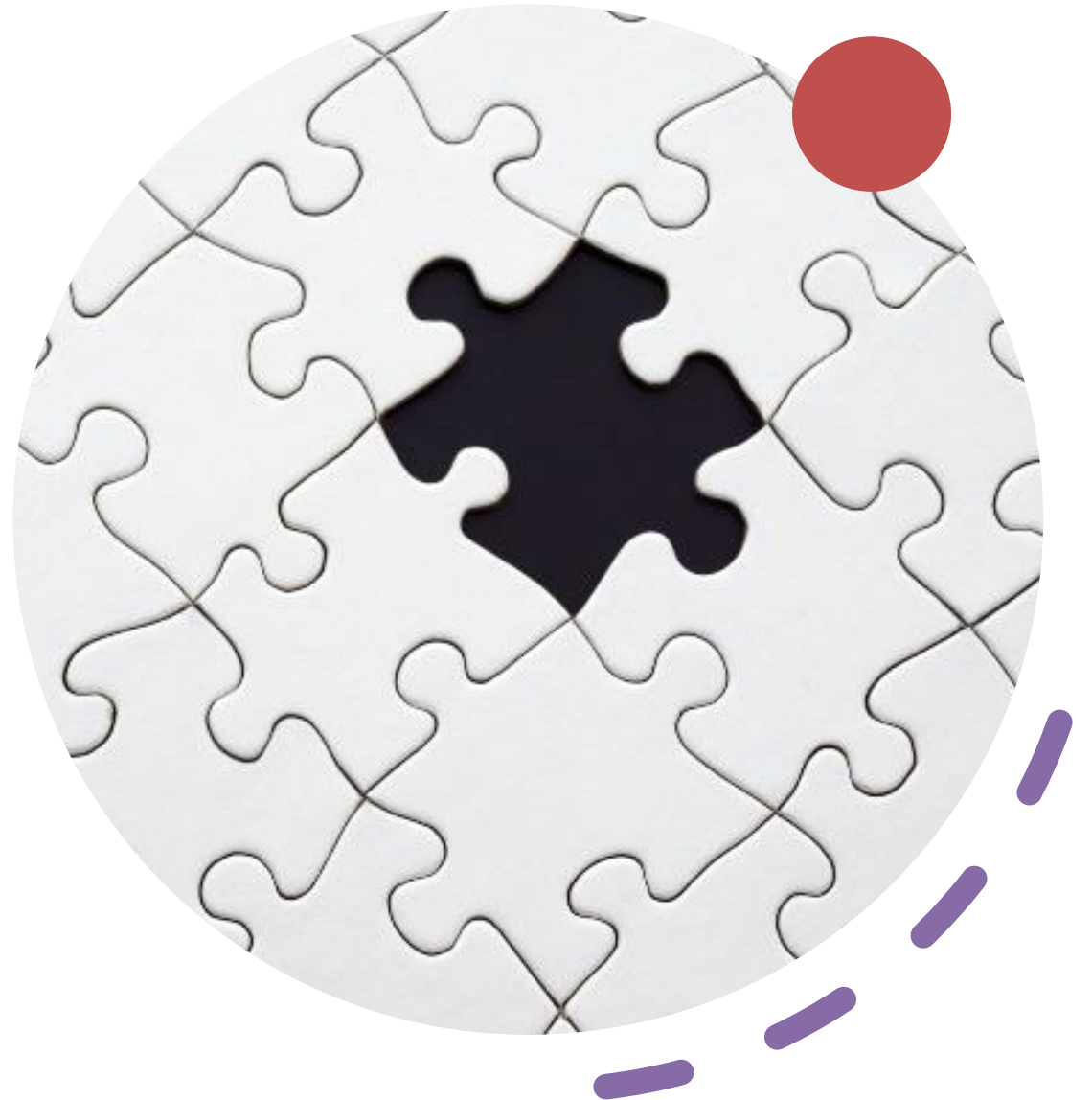
3. Data augmentation (add more features)

4. Imbalanced categories

- Oversampling, undersampling, smote, anomaly detection, cost-sensitive learning

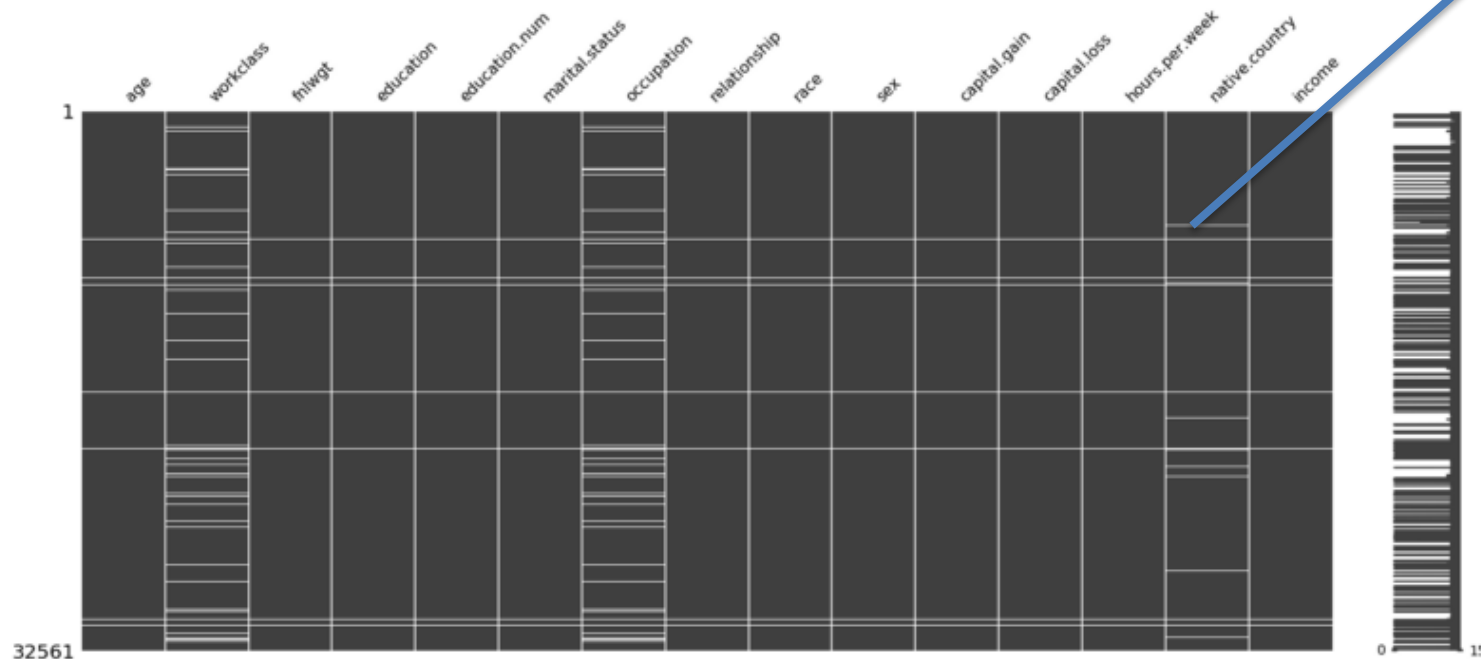
Missing values

- Real-world data often has missing values
- Data can have missing values for several reasons such as observations that were not recorded or data corruption
- Handling missing data is important since many machine learning algorithms do not support data with missing values (or they perform worst, or a particular feature is useful and we would like to recover the most of what we have)



Missing values

The first thing to do is to count how many missing values you have and try to visualize their distributions (methods are provided e.g. see the ***missingno*** package in Python).



The white spaces are missing values



Missing values

- The simplest thing to do is the **removal** of instances with missing values (if missing $<10\%$), or removal of the attribute (if missing $>50\%$). This is rather brute-force, since we lose information.
- For **numerical** values, a standard and often very good approach is to replace the missing values with **mean, median or mode** in the entire distribution of values for a given feature
- With **categorical** values, the standard is to replace with the most probable value
(although it might be dangerous..)

Missing values: better methods

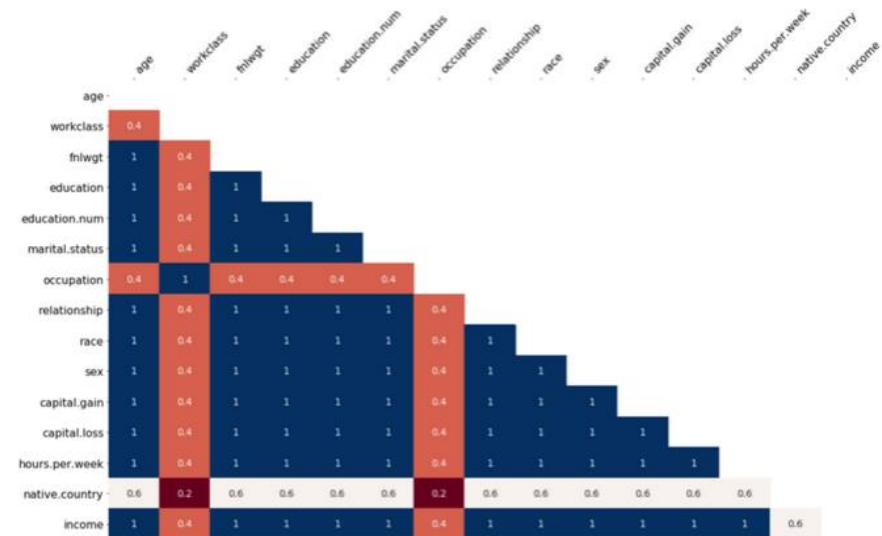
Regression imputation:

A regression model is estimated to predict the observed values of a feature x_j based on other features ($x_k \dots x_n$), and the model is then used to impute values where that variable is missing. (we can use a NN)

$$x_j = w_k x_k + \cdots w_n x_n$$

- **Correlation matrices**

among features can help to design the regression model (what are the most helpful features that could predict the missing value of a feature)

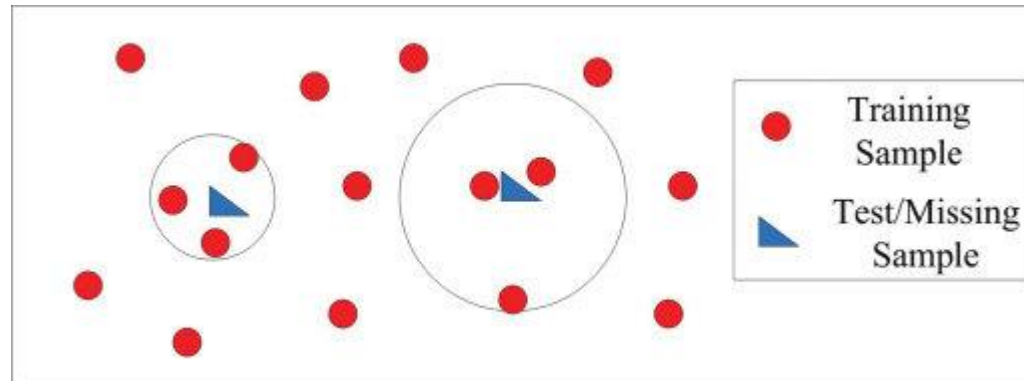


Missing values: other methods

- Imputation with **K-Nearest Neighbours**:

If j -th feature x_j^i is missing on instance \mathbf{x}_i , we can consider the K most similar instances **that have no missing value in j -th feature**.

- Then impute the missing value with the most frequent value (the *mode*) amongst the j -th features of these K instances.



Feature transformation methods



1. Normalization

- Scaling and centering, Change of bases, Categorical to numeric...

2. Missing values

- Removal, regression imputation, k-nearest neighbours...

3. Data augmentation (add more features)

4. Imbalanced categories

- Oversampling, undersampling, smote, anomaly detection, cost-sensitive learning

Feature Augmentation

- Feature augmentation refers to methods that **add more features** to available data
- The objective is **enhancing the quality of models** by adding informative features to the original data
- For image data-sets, you can rotate, scale, translate, interpolate
- For other types of datasets, you can add new features that can be **inferred** from other features
- For example, in a database of football matches, you may want to add for each team the time elapsed between the current match and the last victorious match
- Note: **adding new features is different from adding more data**, an issue that we consider next

Feature transformation methods

1. Normalization

- Scaling and centering, Change of bases, Categorical to numeric...

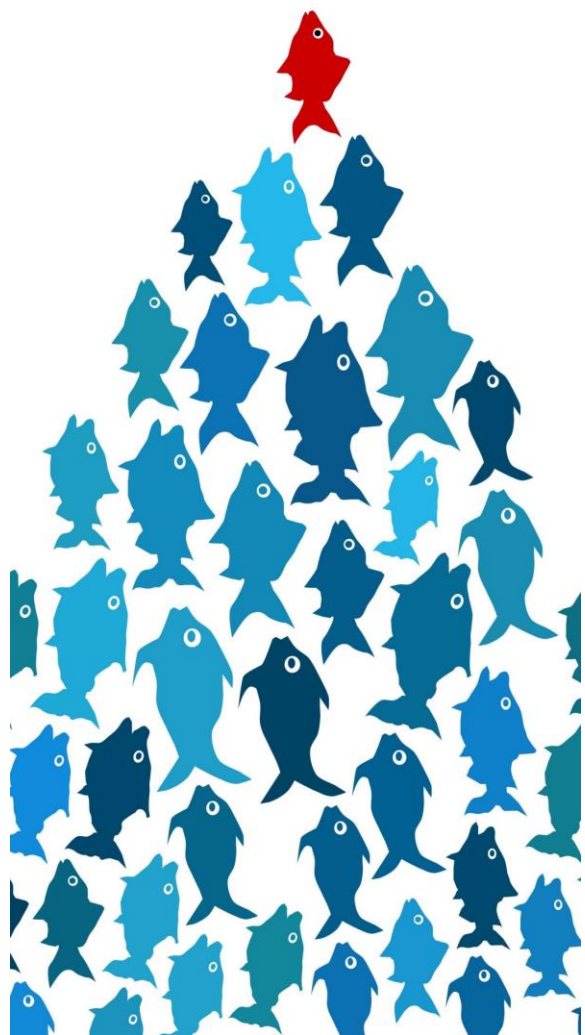
2. Missing values

- Removal, regresion imputation, k-neares neighbours...

3. Data augmentation (add more features)

4. Imbalanced categories

- Oversampling, undersampling, smote, anomaly detection, cost-sensitive learning



Imbalanced Categories

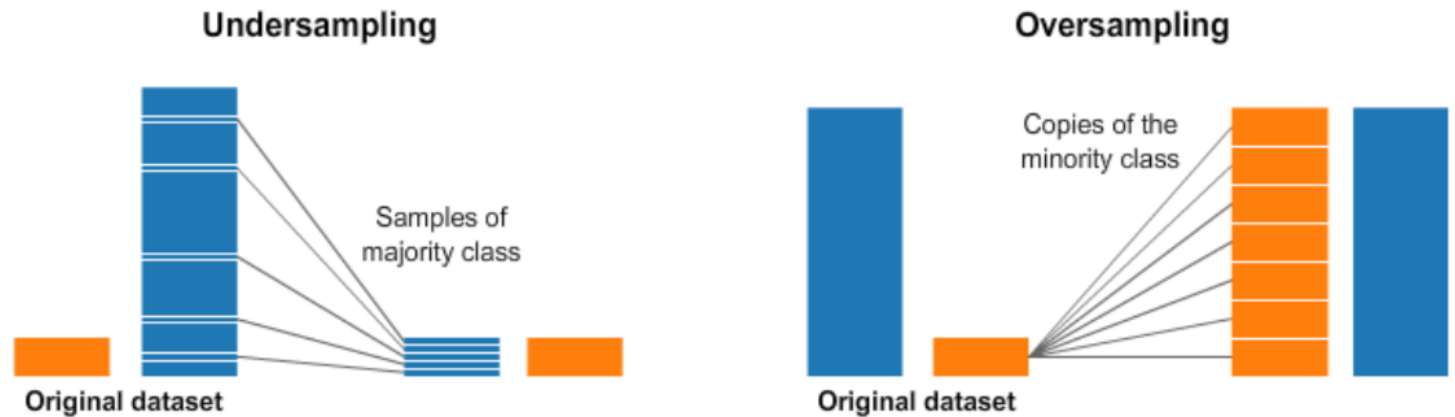
- Class imbalance is when each class does not make up an equal portion of your data-set
- For example, suppose you have two classes—A and B
- Class A is 90% of your data-set and class B is the other 10%, but you are most interested in identifying instances of class B
- You can reach an accuracy of 90% by simply predicting class A every time, but this provides a useless classifier for your intended use case

Why imbalance is a critical issue

- Receiving significantly more examples from one or more classes, the model **could be biased** towards those particular classes;
- In some cases, models trained on unbalanced datasets could actually **completely ignore** the minority classes.
- There are cases where we are actually interested in predicting the minority class, e.g. risk prediction (in health, fraud, and other applications) and in all anomaly detection applications (behavioural anomalies, fake news detection..)

Imbalanced Categories: Sampling

Sampling: A simple way to fix imbalanced data-sets is simply to balance them, either by **oversampling** instances of the minority class or **undersampling** instances of the majority class

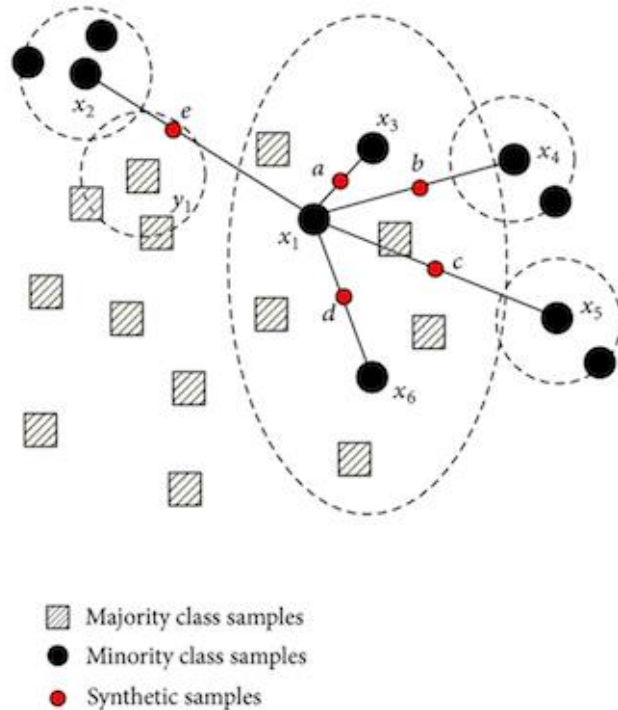


undersampling and oversampling

Disadvantages of under/over sampling

- Undersampling may discard potentially useful data;
- Oversampling creates exact copies of existing examples and **may cause overfitting**;
- Another disadvantage of oversampling is that increasing the number of training examples also increases the learning time.

Imbalanced Categories: SMOTE

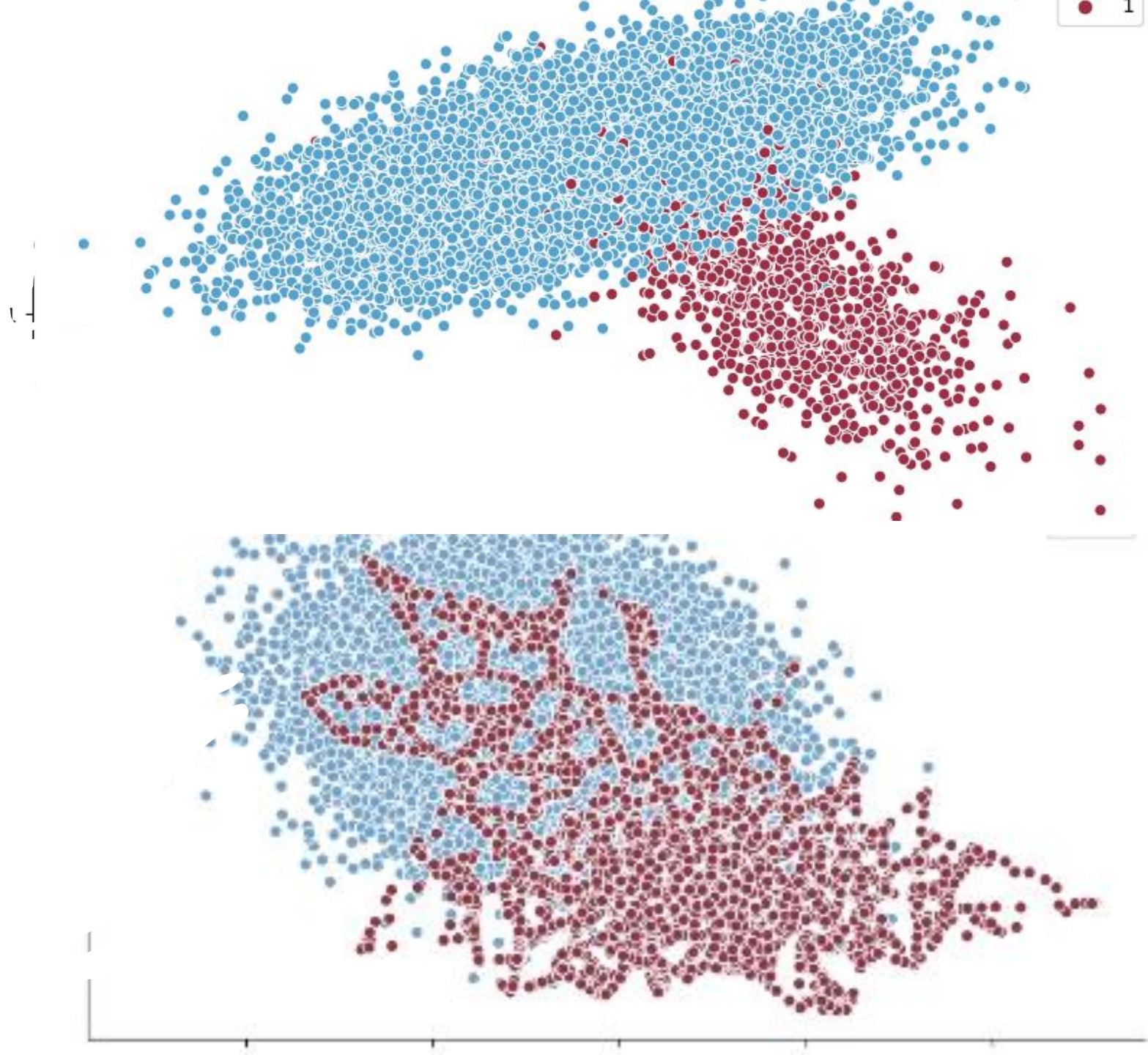


A more powerful sampling method is **SMOTE** Synthetic Minority Oversampling Technique , which creates new instances of the minority class by forming convex combinations of neighboring instances ([link](#)).

- As the graphic shows, it effectively draws lines between minority points in the feature space and samples along these lines.
- If features are categorical, **SMOTE can't be used**. Recent data augmentation approaches relate on **generative methods** such as [GANs](#) that may be also applied to sequential data.

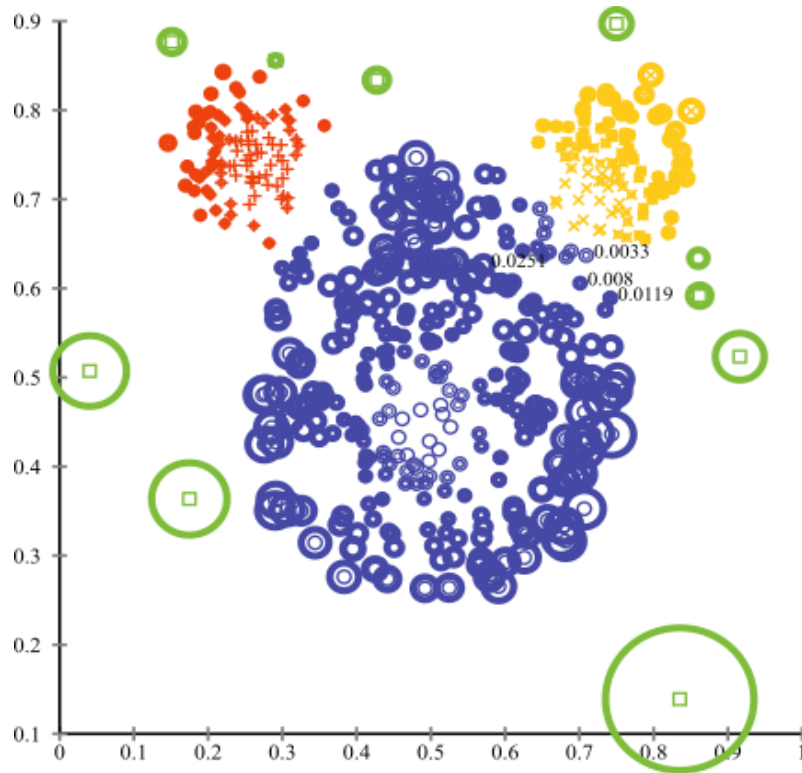
convex combination is a linear combination of points where all coefficients are non-negative and sum to 1.

Example of re-sampling with SMOTE



Imbalanced Categories: Anomaly Detection

Anomaly Detection: we assume that there is a “normal” distribution(s) of data-points, and anything that sufficiently deviates from that distribution(s) is an anomaly.



- When we reframe our classification problem into an anomaly detection problem (see lesson on denoising autoencoders) where we treat the majority class as the “normal” distribution of points, and the minority as anomalies
- We can also simply **ignore** anomalies (however, it depends on the application: if anomalies are, e.g. fraudulent behaviors, then this is exactly what we may be looking for!)

Imbalanced Categories: Cost-Sensitive Learning

In regular learning, we treat all misclassifications equally (regardless of the class which is misclassified), which causes issues in imbalanced classification problems, as there is no **extra reward** for identifying the minority class over the majority class.

- **Cost-sensitive Learning:** Cost-sensitive learning changes this, and uses a function **$C(p, t)$** (usually represented as a matrix) that specifies the **cost of misclassifying** an instance of class **t** as class **p** .

	Actual Positive $y_i = 1$	Actual Negative $y_i = 0$
Predicted Positive $c_i = 1$	C_{TP_i}	C_{FP_i}
Predicted Negative $c_i = 0$	C_{FN_i}	C_{TN_i}

- The algorithm, *in the attempt of minimizing the cost of wrong decisions*, will pay more attention to the minority elements

Feature Engineering: 3 related tasks

Feature Extraction:
Transformation of raw data (e.g, text) into features suitable (e.g., numbers) for processing

Feature Transformation:
Transformation of data to improve the accuracy of the algorithm (e.g., normalization, scaling..)

Feature Selection:
Removing unnecessary features

Feature Selection

How many? Are there enough? Are there too many?

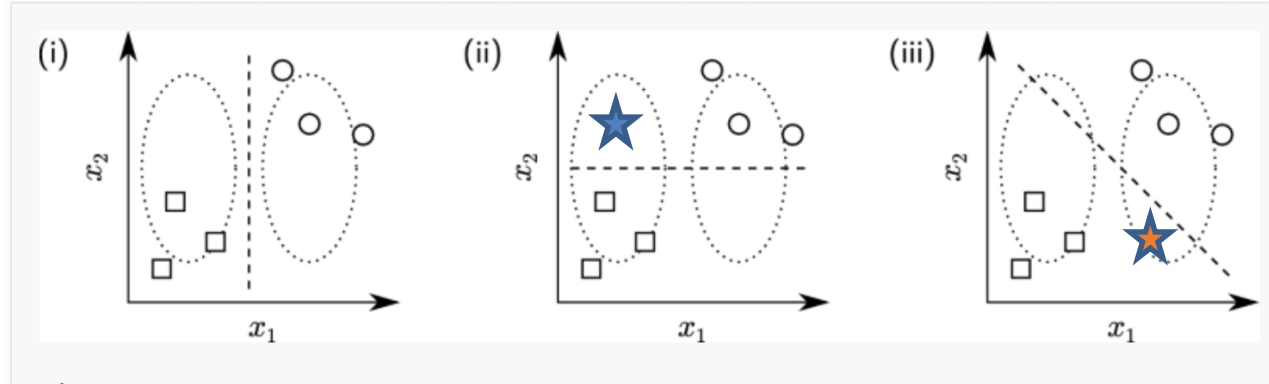
- For any ML task, we can easily come up with thousands of features and extract them from various external sources.
- However, the **number** and **complexity** of **needed** features often depend on the specific task addressed
- For example, if you need to distinguish city landscapes from mountain landscapes you don't need pixel features (a color histogram would do)



Feature Selection

- In many practical cases, one may come out with very many– potentially useful features (so the “**too many**” is the most frequent case)
- Not easy to say what is truly useful, nor if some features are correlated:
 - Adding many *potentially correlated* features **can decrease model performance**
 - “Too many” features make models less **interpretable** and less **generalizable**
- So, we need **automatic tools for feature selection (filtering)**

Feature selection: why it is important



- Ovals represent the (hidden, i.e. unknown) space of positive (squares) and negative (circles) examples
- Dashed lines are the "models" (classification functions learned from available data, that separate positive examples from negative)
- In the reality, as shown by the figures above, *only feature x_1 is useful to predict the class value* of examples (Figure i) but given the examples, a ML algorithm may come out with any of the 3 models (i) (ii) and (iii). However, model (ii) and (iii) **would NOT generalize on unseen instances**
- For example, instance ★ will be mistakenly predicted as negative by model (ii) and instance ★ would be mistakenly predicted as positive by model (iii)

Feature Selection

Since the exhaustive search for an optimal feature subset is infeasible in most cases, many search strategies have been proposed in the literature, often classified in three types:

- Filter Methods (A)
- Wrapper Methods (B)
- Embedded and hybrid methods (C)
- To learn more: [link](#)

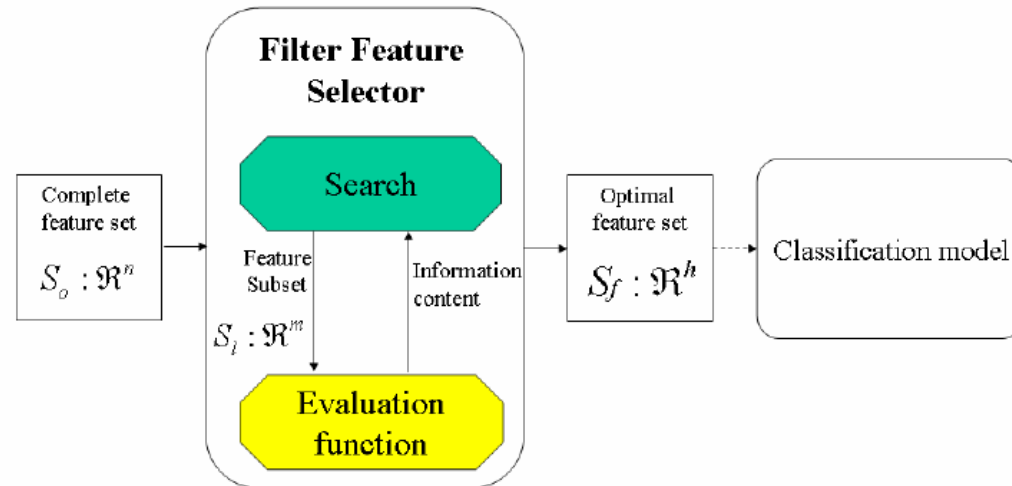
Feature Selection vrs dimensionality reduction

- Feature selection is basically a process that selects and excludes some features **without modifying** them at all.
- The other strategy is Dimensionality reduction that **modifies or transforms** features into a lower dimension, creating a whole new feature space that looks approximately like the first one, but smaller in terms of dimensions.
- You will familiarize with some «classic» dimensionality reduction strategies (e.g., matrix factorization, principal component analysis) in other courses
- As far as ML methods are concerned, **Deep encoders** are a way to reduce the dimensions of a feature set, projecting input vectors onto a latent space.
- Note that also ensembles inherently limit the negative effect of irrelevant features, but they do not explicitly remove or replace them.

Feature Selection:

A) Filter Methods

- Filter methods select features based on a performance measure **regardless of, and prior to, the employed data classification algorithm**
- Only **after** the best features are found, the ML algorithms can use them



Feature Selection: A) Filter Methods

- We can roughly classify the developed measures for **feature filtering** into:
information, distance, consistency, similarity, and statistical measures
- Furthermore:
 - *univariate* filters evaluate (and usually rank) a **single feature**
 - *multivariate* filters evaluate an entire feature subset

A list of common filter methods

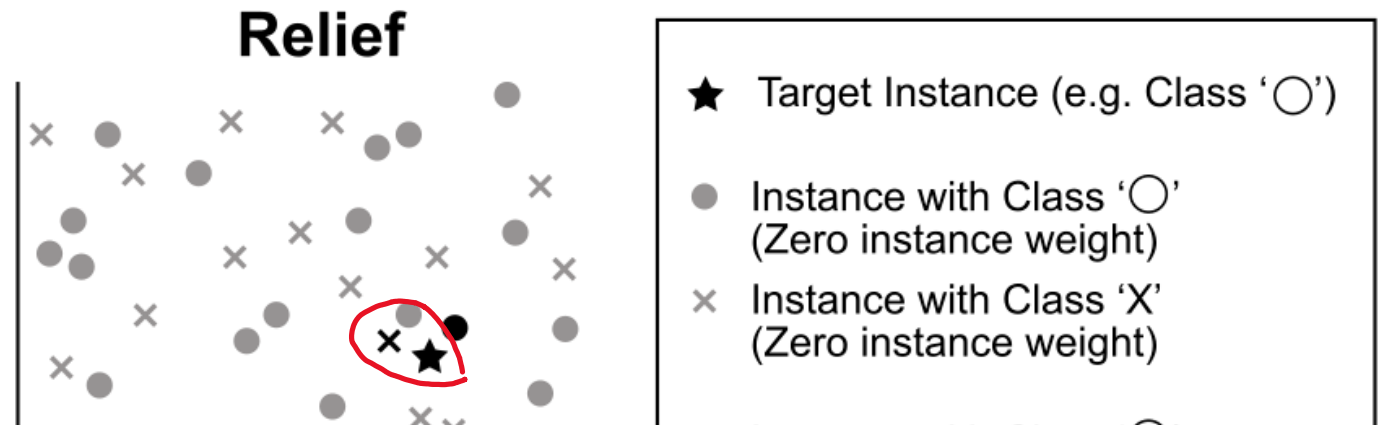
Fast correlation-based filter (FCBF)	multivariate, information	classification
Fisher score	univariate, statistical	classification
Relief and ReliefF	univariate, distance	classification, regression
Spectral feature selection (SPEC) and Laplacian Score (LS)	univariate, similarity	classification, clustering
Feature selection for sparse clustering	multivariate, similarity	clustering
Localized Feature Selection Based on Scatter Separability (LFSBSS)	multivariate, statistical	clustering
Multi-Cluster Feature Selection (MCFS)	multivariate, similarity	clustering
Feature weighting K-means	multivariate, statistical	clustering
ReliefC	univariate, distance	clustering

Name	Filter class	Applicable to task
Information gain	univariate, information	classification
Gain ratio	univariate, information	classification
Symmetrical uncertainty	univariate, information	classification
Correlation	univariate, statistical	regression
Chi-square	univariate, statistical	classification
Inconsistency criterion	multivariate, consistency	classification
Minimum redundancy, maximum relevance (mRmR)	multivariate, information	classification, regression
Correlation-based feature selection (CFS)	multivariate, statistical	classification, regression

Examples of filters: RELIEF

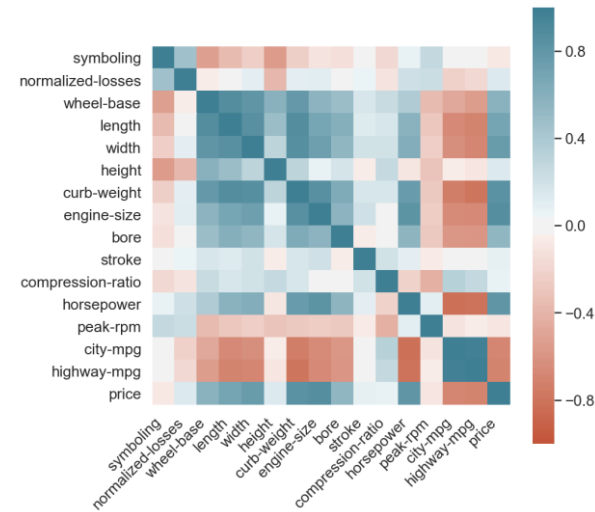
- **Information Gain** (information, univariate: we have seen it in DT)
- **Relief(F)** (distance, univariate): consider all features as **independent ones** and estimate the relevance (quality) of each feature based on its ability to distinguish instances located **near** each other, but **belonging to different classes**:
 - The algorithm iteratively selects a random **instance x** and then searches for its **two nearest neighbors** in D: the **nearest hit** (from the **same class**, e.g., negative) and the **nearest miss** (from a different class).
 - **For each feature value x_i of \mathbf{x}** , the estimation of the quality of the i -th feature (weight W_i) is updated depending on the differences between the current instance and its nearest hit and **along the corresponding feature i axis**.
$$W_i = W_i - (x_i - \text{nearHit}_i)^2 + (x_i - \text{nearMiss}_i)^2$$
 - The weight W_i increases if the value of the near miss is «far» and the value of the near hit is «close»
 - The rationale is: **to what extent this feature is able to differentiate two instances belonging to different classes?**
 - Several measures to compute difference (euclidean distance, Manhattan distance..)
 - Only good for numeric features

Relief Example



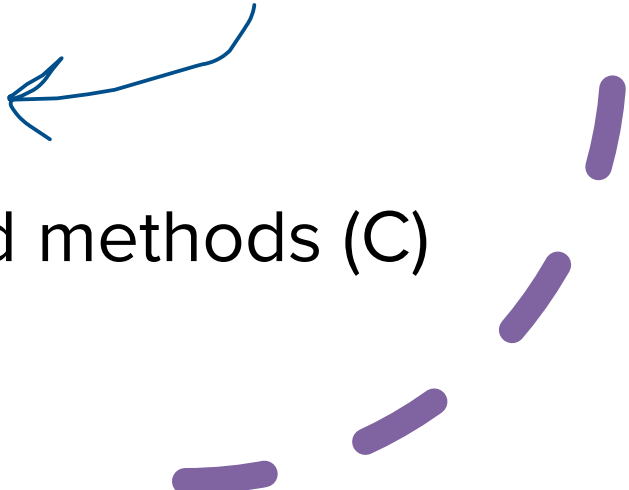
Correlation-based feature selection

- Based on the following principles: If two or more variables are correlated, only one can be selected
- [Spearman correlation, \$\chi\$ – square test](#) are common methods to identify correlated variables (and remove the dependent variable)
- Heatmaps can graphically identify correlation between variables



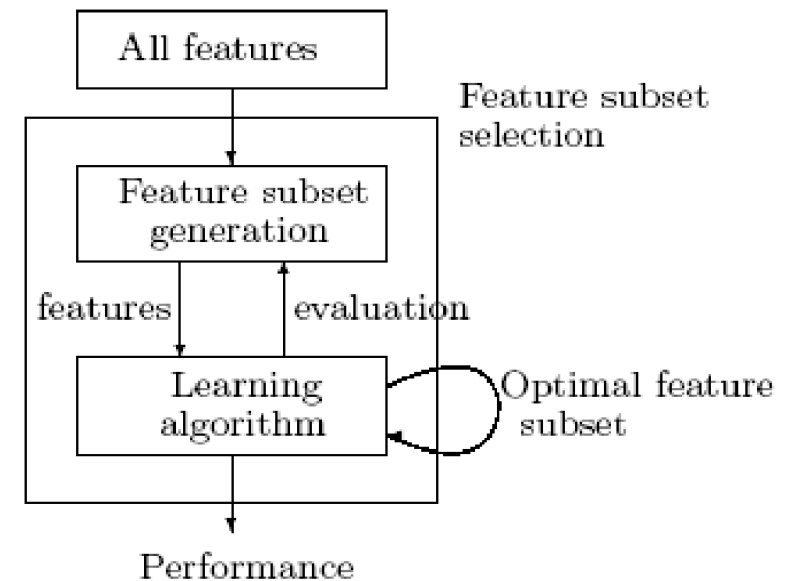
Feature Selection

Since the exhaustive search for optimal feature subset is infeasible in most cases, many greedy search strategies have been proposed in the literature, often classified in three types:

- Filter Methods (A)
- **Wrapper Methods** (B) 
- Embedded and hybrid methods (C)
- To learn more: [link](#)

Feature Selection: B) Wrappers

- Wrappers evaluates **feature subsets** by the quality of the performance on a specific ML algorithm, which is taken as a “**black box**” **evaluator**



(b) Wrapper method

Feature Selection:

B) Wrappers

- Thus, for classification tasks, a wrapper will evaluate subsets of features based on a ML method performance (e.g. Regression Trees or Neural Networks)
- **The evaluation is repeated for each subset**, and the subset generation is dependent on the search strategy, in the same way as with filters (e.g., random)
- **Wrappers are much slower than filters** in finding sufficiently good subsets because they depend on the considered algorithm

Feature Selection:

B) Wrappers Methods

- Recursive feature elimination
- **Sequential** feature selection algorithms
- Genetic algorithms

Feature Selection:

Sequential feature selection algorithm

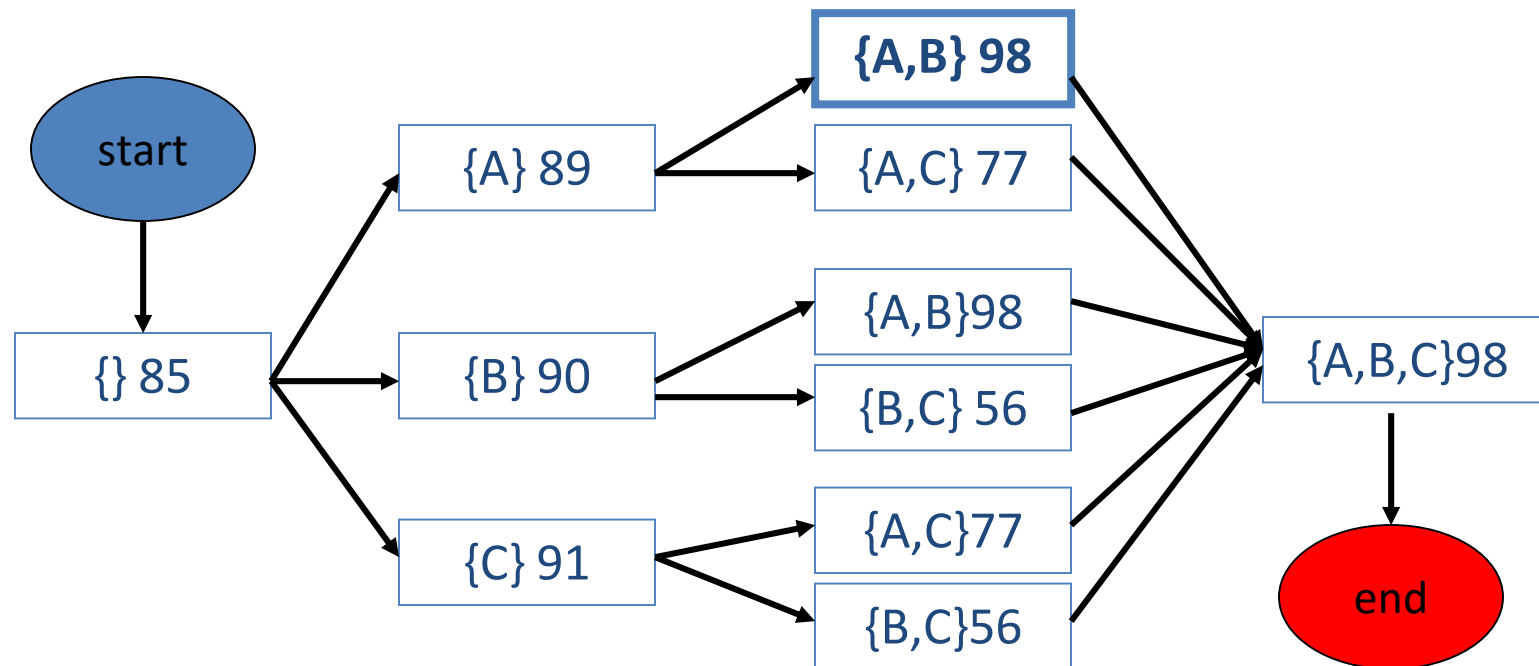
The task: Say we have features A, B, C, and a classifier M . We want to predict T (the class) given the smallest possible subset of features $\{A, B, C\}$ while achieving maximal performance (accuracy)

FEATURE SET	CLASSIFIER	PERFORMANCE
{A,B,C}	M	<u>98%</u>
<u>{A,B}</u>	M	<u>98%</u>
{A,C}	M	77%
{B,C}	M	56%
{A}	M	89%
{B}	M	90%
{C}	M	91%
{.}	M	85%

Feature Selection:

Sequential feature selection algorithm

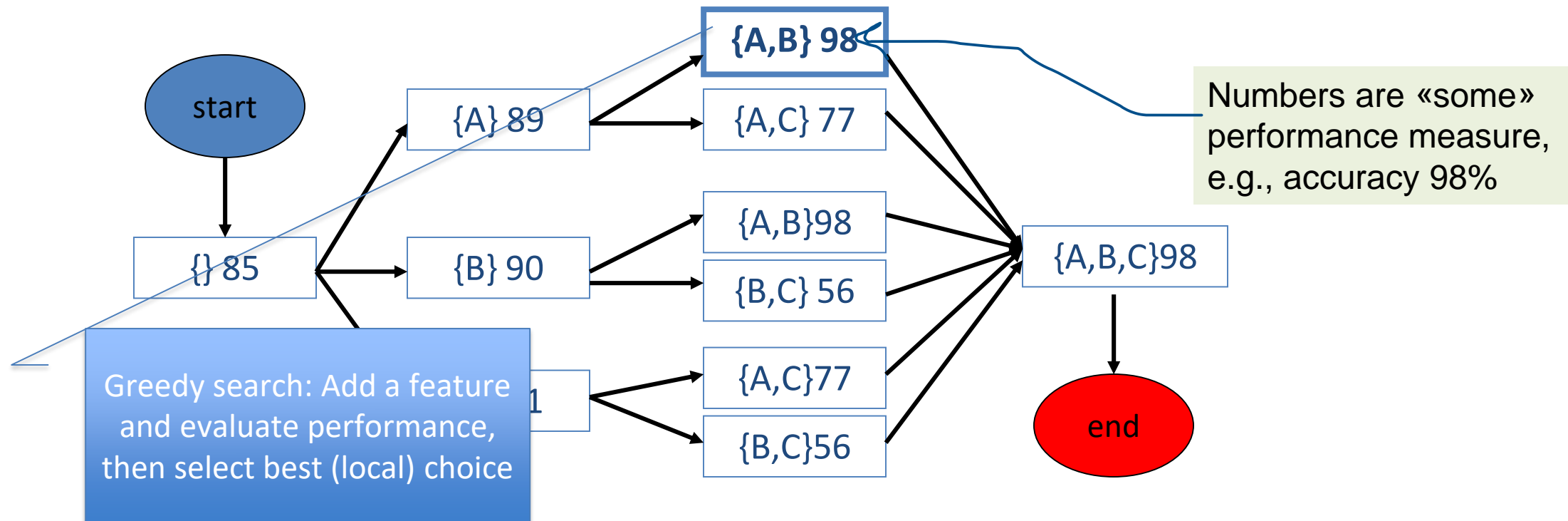
The set of all subsets of features is the *power set* and its size is $2^{|V|}$. Hence for large V , **we cannot do this procedure exhaustively**; instead, we rely on a *heuristic search of the space* of all possible feature subsets.



Feature Selection:

Sequential feature selection algorithm

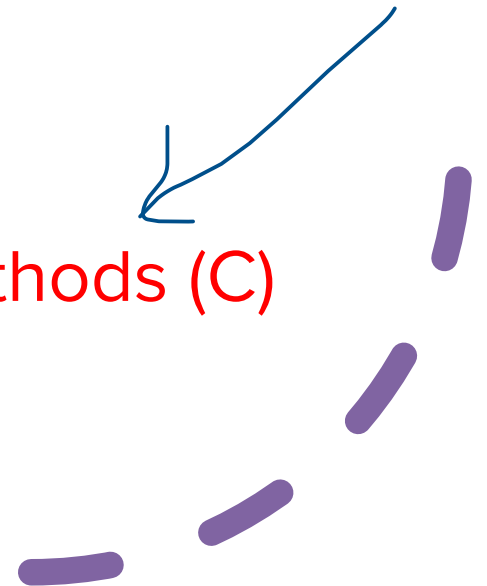
A common example of heuristic search is *hill climbing*: keep adding features one at a time until no further improvement can be achieved. Evaluation is based, e.g., on a lookahead of one step.



Feature Selection

Since the exhaustive search for optimal feature subset is infeasible in most cases, many search strategies have been proposed in the literature, often classified in three types:

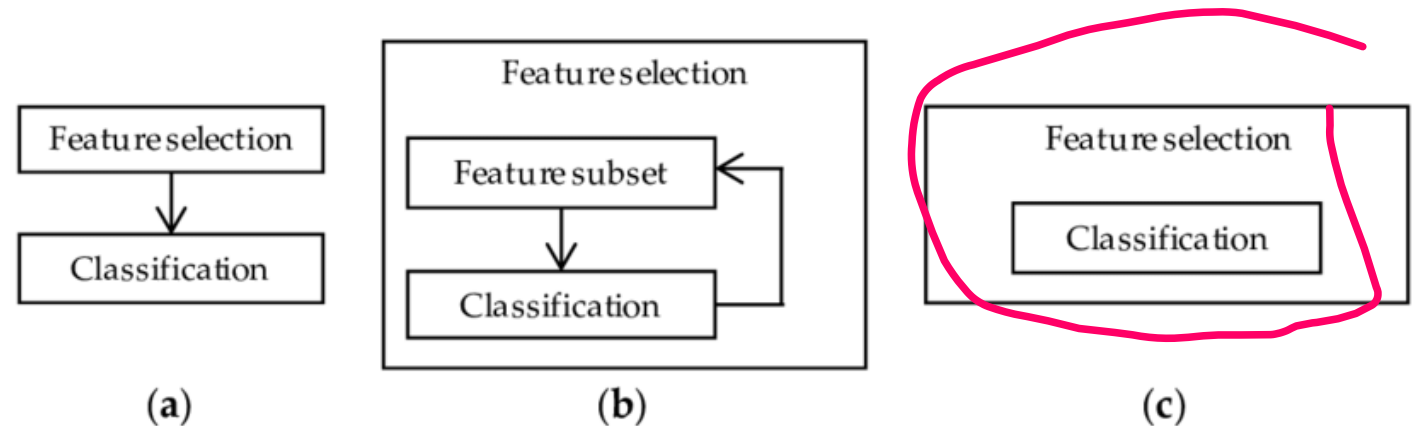
- Filter Methods (A)
- Wrapper Methods (B)
- Embedded and hybrid methods (C)
- To learn more: [link](#)



Feature Selection: C) Embedded methods

Embedded methods perform feature selection **during the execution of the ML algorithm.**

- In contrast with filter (a) and wrapper (b) approaches, in embedded methods (c) the features selection part **can not be separated from the learning part.**
- Most embedded methods are model-dependent, i.e. they are specifically designed for the class of ML algorithms chosen



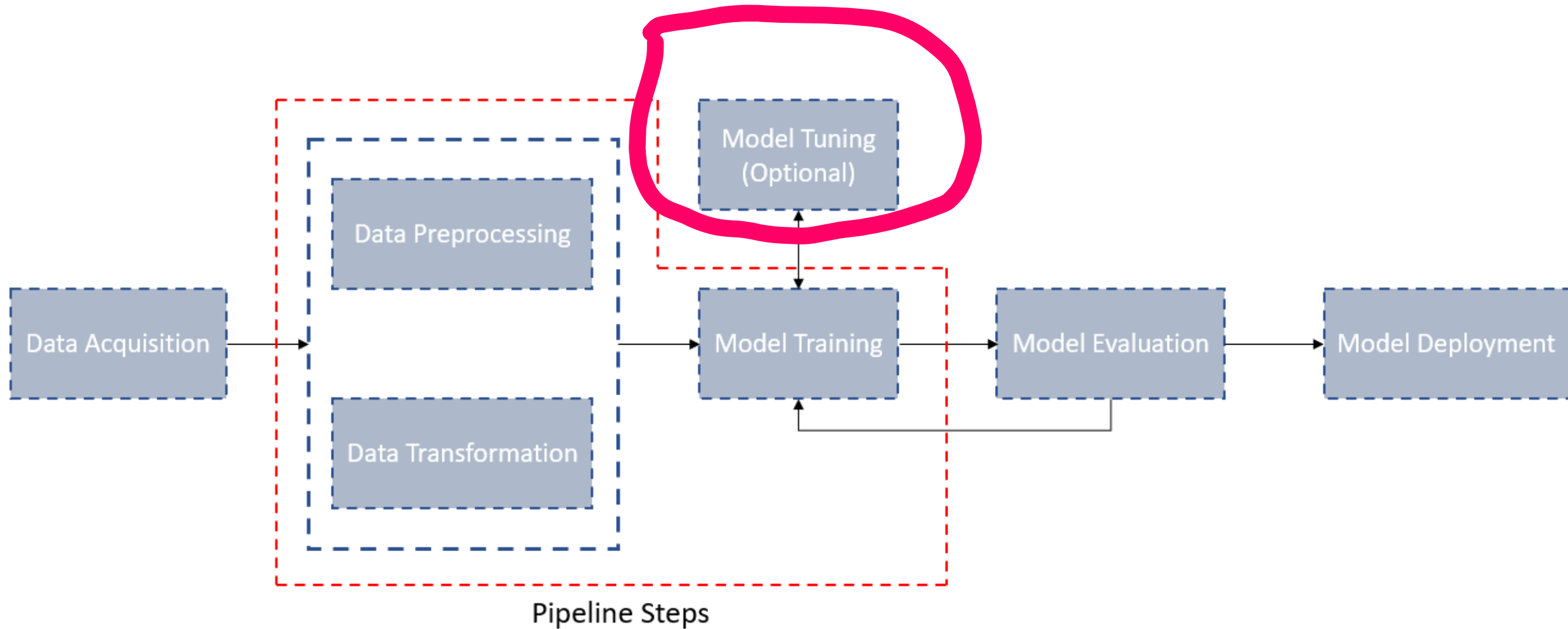
Feature Selection: C) Embedded methods

- Any and all embedded methods work as follows:
 - First, these methods train a machine learning model.
 - They then derive feature importance from this model, which is a measure of how much is each feature important when making a prediction.
 - Finally, they remove non-important features using the derived feature importance.

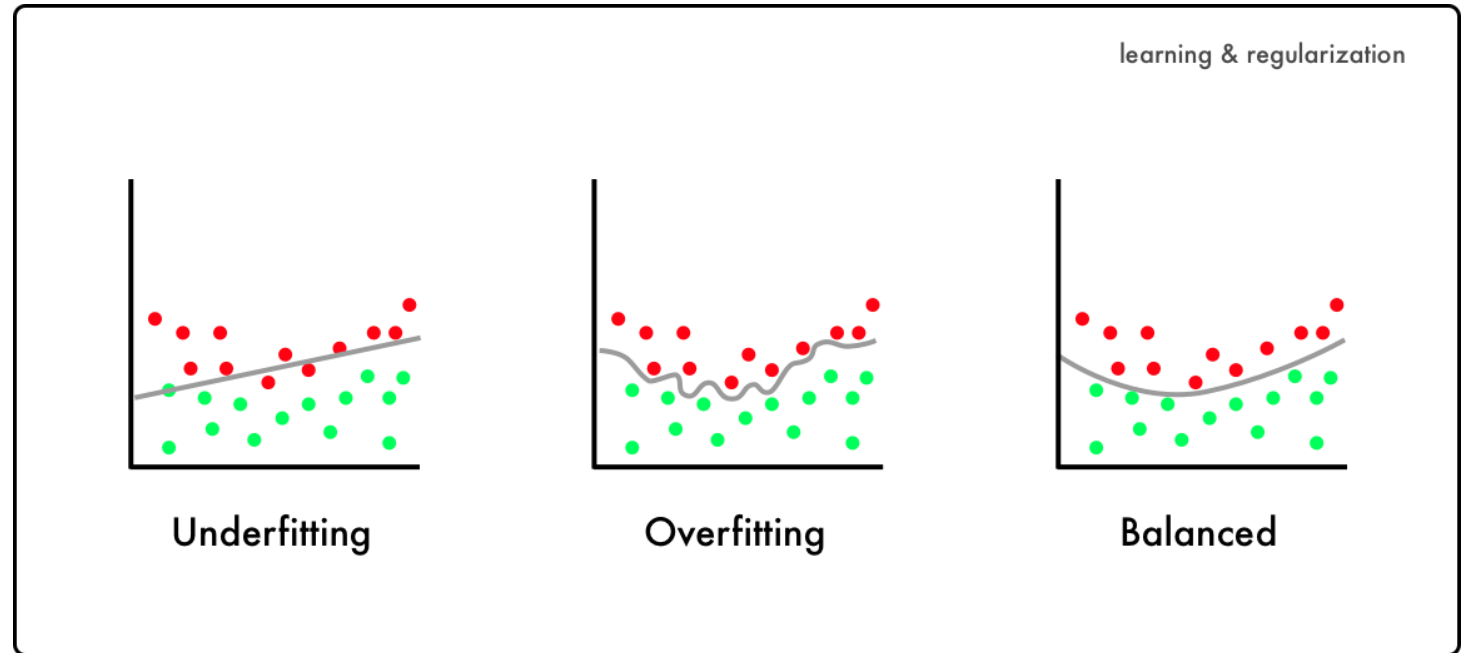
Embedded methods (2)

1. The most Common embedded technique are the tree-based algorithms like Random Forest.
2. Tree-based algorithms select a feature in each recursive step of the tree growth process and divide the sample set into smaller subsets. Topmost features in the tree are the most relevant, as we have already learned.
3. Other Embedded Methods are regularization methods, such as the LASSO with the L1 penalty and Ridge with the L2 penalty for constructing a linear model. These two methods shrink many features to zero or almost near to zero. We discuss later regularization methods, since they are part of the “model fitting” strategies.

The ML pipeline: model tuning (fitting)



Model tuning (fitting)



- Model fitting is a measure of how well a machine learning model generalizes to similar data to that on which it was trained.
- A model that is well-fitted produces more accurate outcomes. A model that is overfitted matches the data too closely.
- An underfitted model makes bad predictions

Overfitting and underfitting strategies

- techniques

Reducing Overfitting

- Increase Training Data
- Reduce Model Complexity
- Early stopping during Training Phase
- L1 and L2 regularization
- Dropouts for Neural Network

Reducing Underfitting

- Increase Training Data
- Increase complexity of Model
- Increase no. of features
- Remove Noise from data
- Increase no. of training epochs

(some ways of) Preventing overfitting

- Approach 1: Get more data!
 - Almost always the best bet if data is cheap and you have enough compute power to train on more data.
- Approach 2: Average many different models.
 - Ensembles (see previous lessons)
- Approach 3: Early stopping
 - Start with small weights and stop the learning before it overfits.

Approach 4: Regularization methods

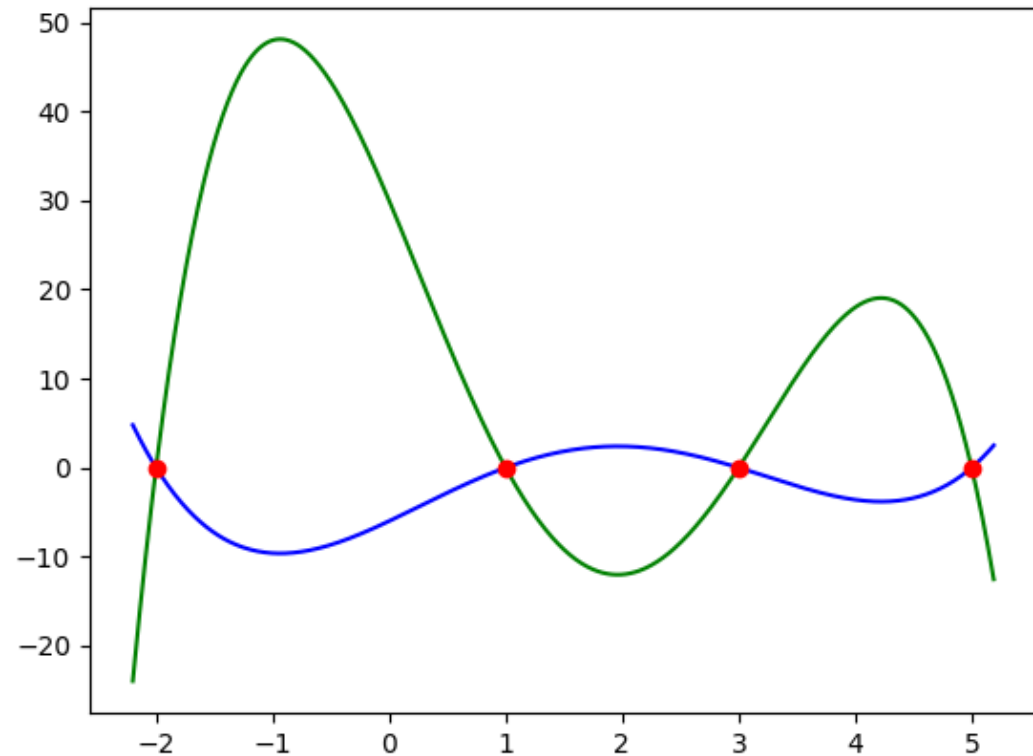
- Lasso, Ridge regression, Drop-out

Regularization

- [Regularization](#) adds a penalty to the different parameters of a model to reduce its freedom in the selection of best parameters.
- This penalty is applied to the coefficient that multiplies each of the features (e.g., the weights in a linear convolutional model), and is done to avoid overfitting, make the model robust to noise, and to improve its generalization.
- The simplest regularization method is [Lasso](#) (L1) for linear models – non linear models use other regularization methods, e.g., Ridge regression

What is regularization

- Red points are «examples» in the training dataset
- The green curve is an overfitting example. In mathematical terms, the green curve has «too large» coefficients
- Regularization aims at reducing the coefficients (e.g. the weight of a single convolutional layer) such as in the blue line



Lasso regularization

- As we said, embedded methods are strictly dependent on the selected prediction model
- Lasso (L1) regularization is **only applicable to algebraic linear models (regressors, perceptron..)** that model the output as a linear combination of input features x_{ji} :

$$y_i = w_0 + \sum_{j=1}^m w_j x_{ji}$$

The output value y_i for an input x_i is predicted as a linear combination of input features x_{ji}

As we said, learning a predictive model requires estimating the coefficients w_i , based on the known $\langle x_i, y_i \rangle$ pairs in the training set (as we have seen for the perceptron model)

Lasso regularization (2)

- We know that learning a model (e.g. learning a linear model) always imply to **define an optimization problem to minimize some error function** (called Loss function). Model **parameters** (the w_i in our current linear model) are adjusted to minimize the error of predictions
- In linear models, a possible Loss function is **Residual Sum of Squares**:

$$RSS = \sum (y_j - \hat{y}_j)^2 = \sum_{j=1}^n (y_j - \sum_{i=1}^m w_i x_{ji})^2$$

Where x_{ji} is the i -th **feature** of input j of the dataset, and y_j is the (known) true value of the output function

The optimization problem is to find all w_i such that RSS is minimised

Lasso regularization (3)

- The Lasso regularization problem can be stated as follows:

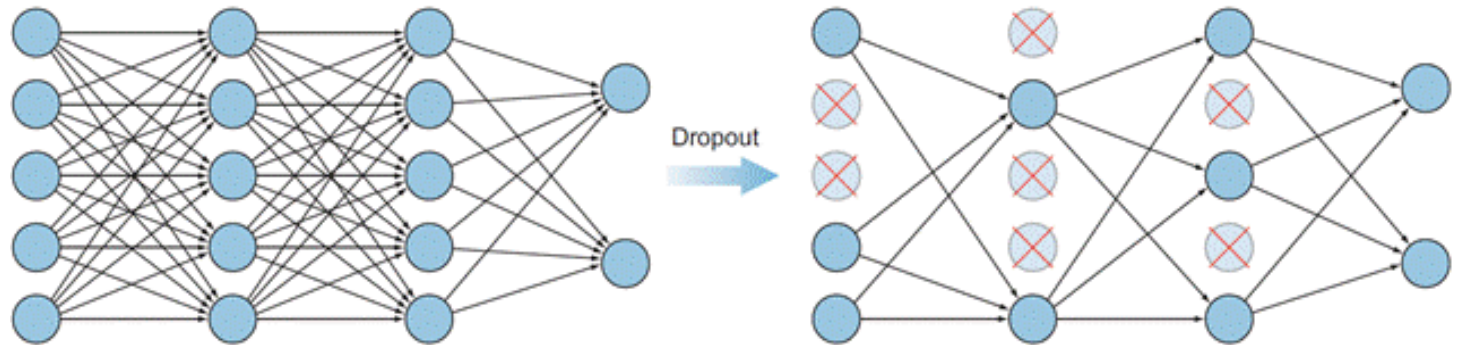
MINIMIZE

$$\text{RSS} = \sum_{j=1}^n (y_j - \sum_{i=1}^m w_i x_{ji})^2 + \lambda \sum_{i=1}^m |w_i|$$

- The red part is called ***l1* penalty** (since it **increases** the RSS), and has the effect of forcing some of the coefficients w_i to be **exactly zero** – in the attempt of minimazing RSS - when the λ parameter is sufficiently large, so it performs feature selection.
- This process is bit “extreme” since it essentially **eliminates** those features from the model instead of minimizing their impacts.
- Similar to the Lasso regression, Ridge regression puts a similar constraint on the coefficients by introducing a penalty factor. However, while Lasso regression takes the magnitude of the coefficients, [Ridge regression](#) takes the square.

Drop out

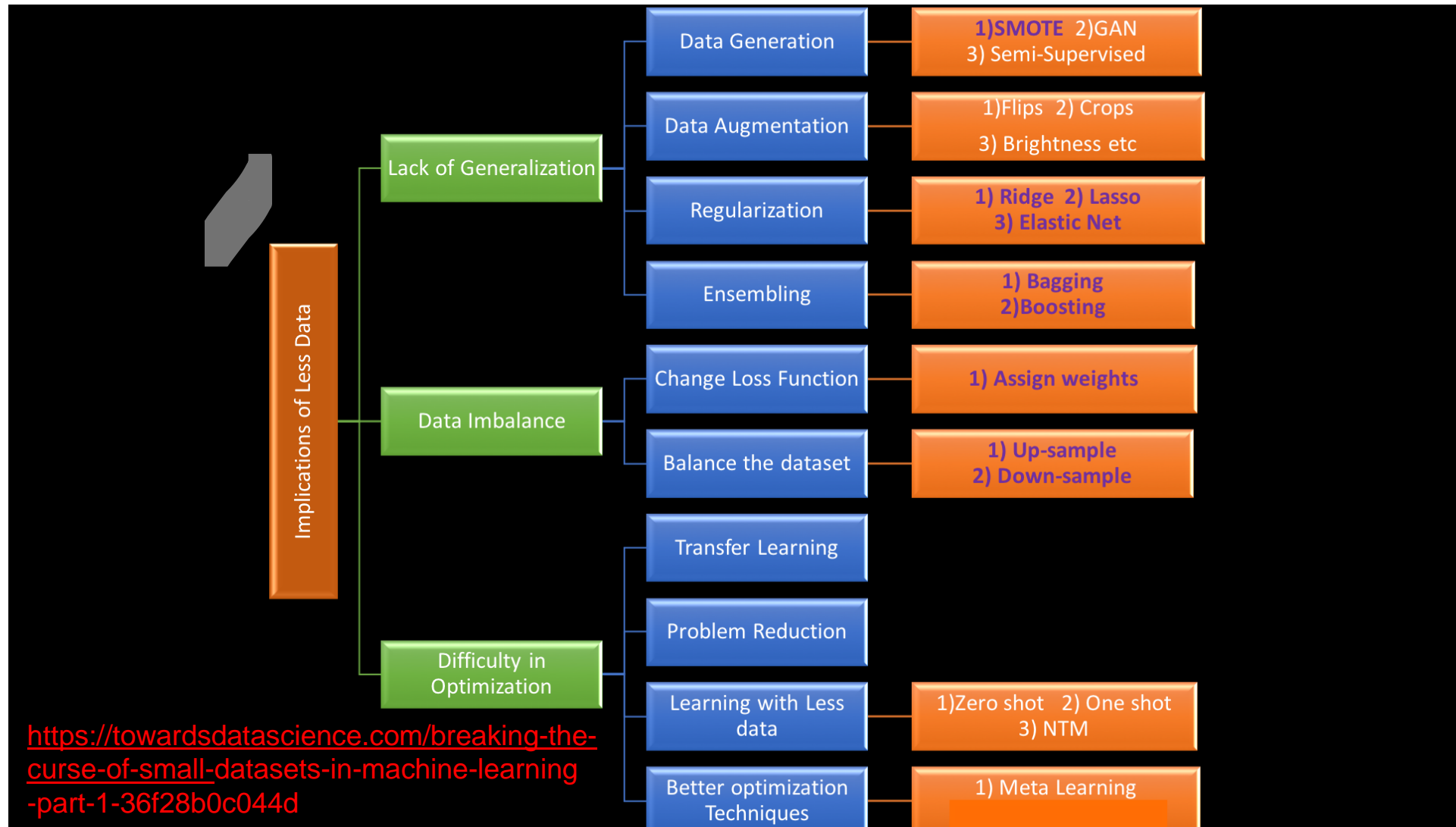
- Dropout is another regularization method that approximates training a large number of neural networks with different architectures in parallel.
- During training, some number of layer outputs are randomly ignored or “*dropped out*.” This has the effect of making the layer treated like a **layer with a different number of nodes** and connectivity to the prior layer.
- Each update to a layer during training is performed with a different “*view*” of the configured layer.



Why dropout works

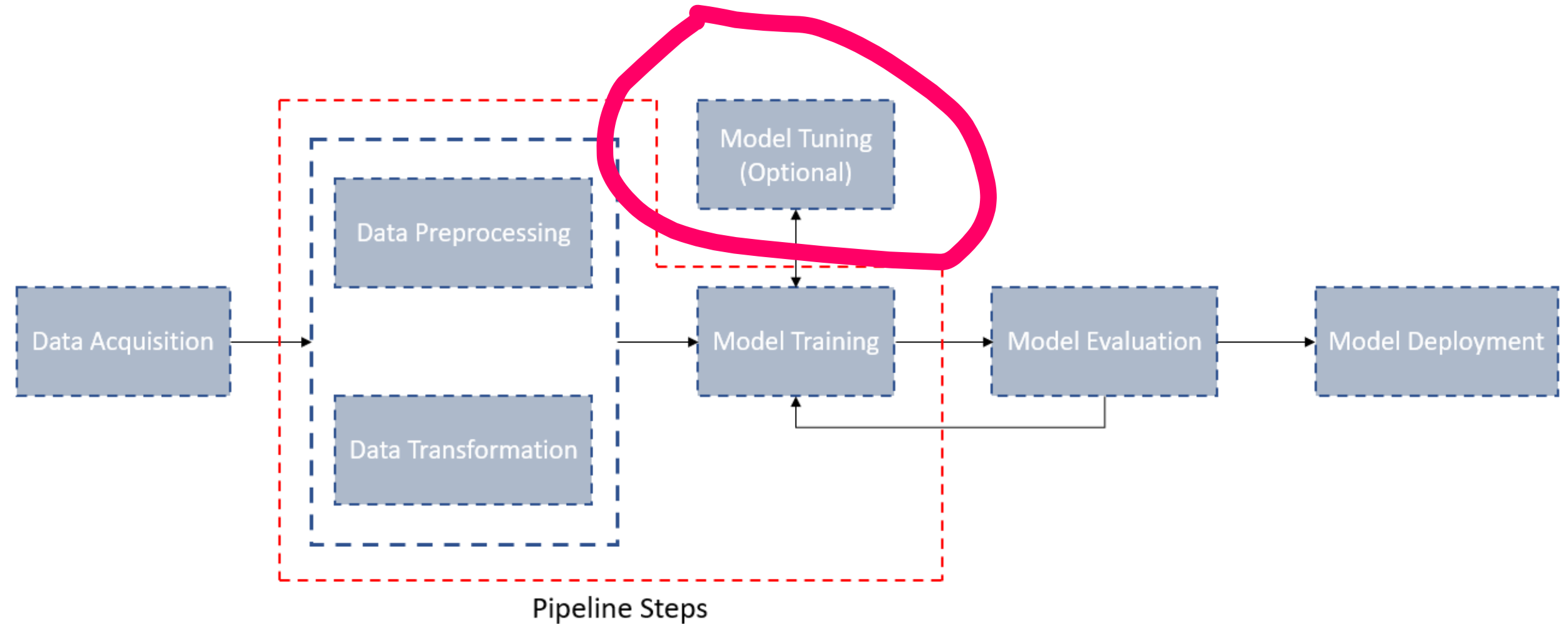
- When training a NN, a neuron's synaptic weights may change in a way that “fixes up” the mistakes of the other units (remember the backpropagation formula!).
- This leads to complex **co-adaptations among weights**, which in turn leads to the overfitting problem because this complex co-adaptation may fail to generalise on the unseen dataset.
- Dropout prevents these units to fix up the mistake of other units (since these are occasionally removed during training), thus preventing “strong” co-adaptation, as in every iteration the presence of a unit is highly unreliable.
- By randomly dropping a few units (nodes), we relax the dependence of each layer from the others.

Underfitting is when we have **too few data**
(some of the already seen methods to avoid overfitting or
to cope with unbalanced data may help with this)



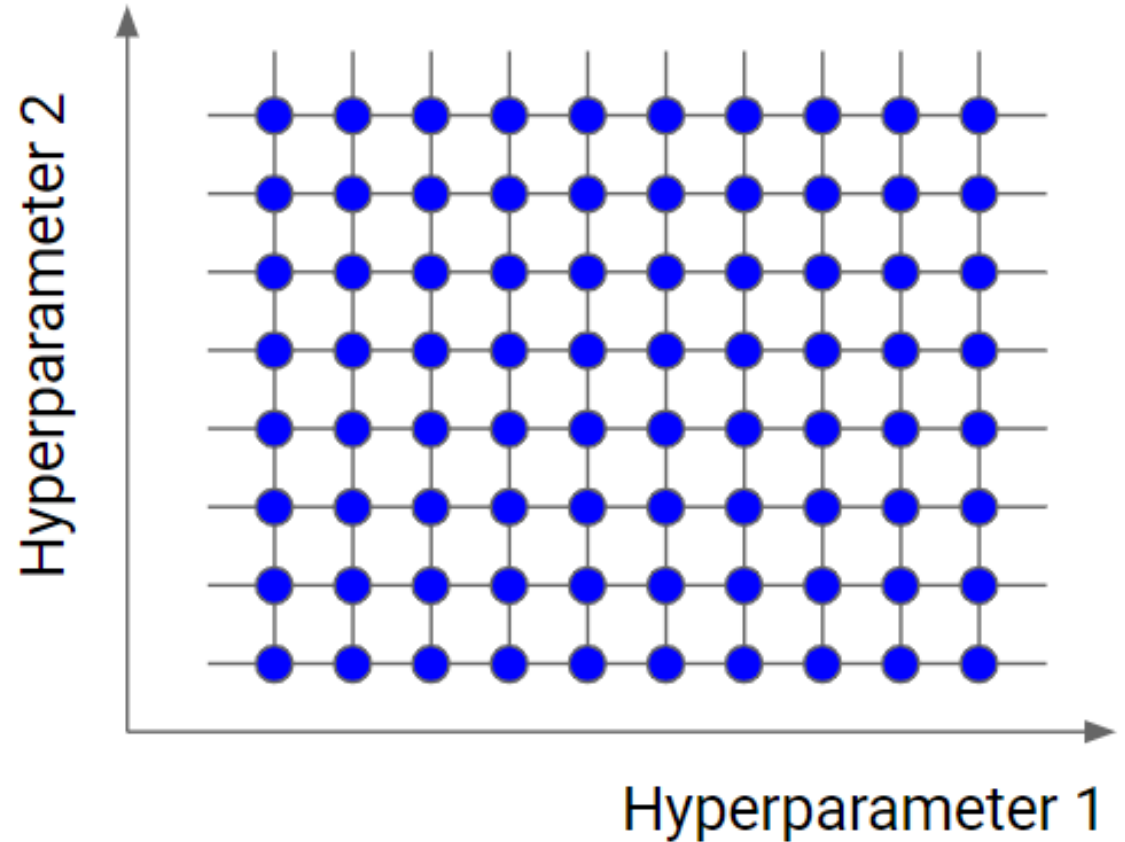
<https://towardsdatascience.com/breaking-the-curse-of-small-datasets-in-machine-learning-part-1-36f28b0c044d>

The ML pipeline: Hyperparameter tuning



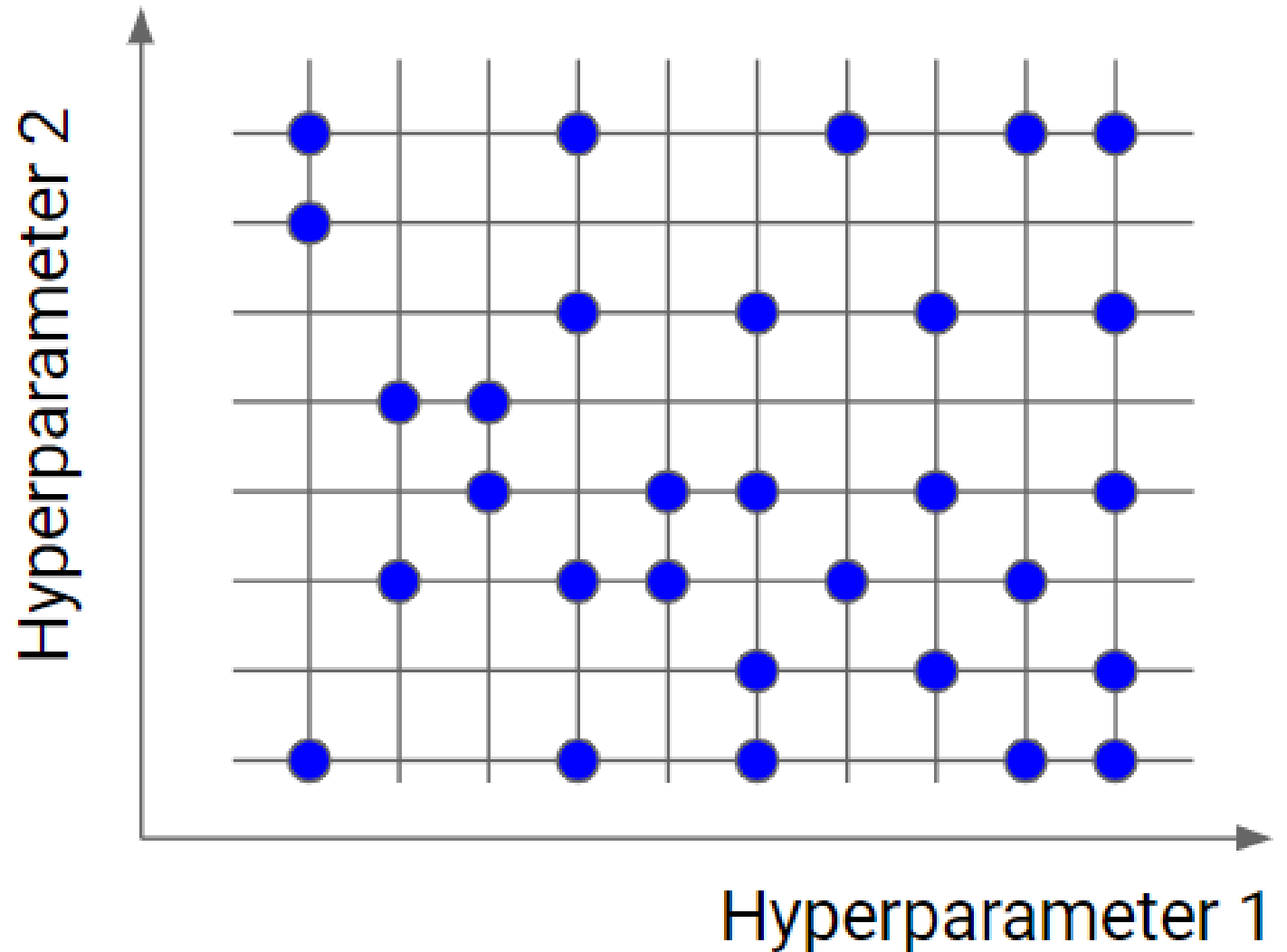
Grid search

- The grid search is an exhaustive search through a set of manually specified set of values of hyperparameters.
- It means you have a set of models (which differ from each other in their parameter values, which lie on a grid).
- We train each of the models and evaluate it. We then select the one that performed best.
- For many hyperparameters, **exhaustive grid search is not feasible**



Random search

- Set up a grid of hyperparameter values and select *random* combinations to train the model and score it.
- The number of search iterations is set based on time/resources.



Automated Hyperparameter Tuning

- [Bayesian Optimization](#) (see the link for a very clear description)
- Adam optimizer is also very popular
<https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/>
- [AutoML](#) aims to automatize **the entire ML workflow**, from feature pre-processing to model optimization and hyperparameter tuning.
- Many AI companies have created and publicly shared such systems (e.g., [Cloud AutoML](#) by Google) to help people with little or no ML knowledge to build high-quality custom models.