ADVANCED DATA ANALYTICS – STUDY NOTES

Introduction to Big Data

To extract the knowledge (and value), data needs to be
- Collected
- Stored
- Managed
- Analyzed

Analytics: The science that analyze crude data to extract useful knowledge from them.
Discover patterns and models that are:
- Valid: hold on new data with some certainty
- Useful: should be possible to act on the item
- Unexpected: non-obvious to the system
- Understandable: humans should be able to interpret the pattern

Machine Learning: The study of computer algorithms that improve automatically through experience.

Data Mining: This field focuses on the discovery of unknown properties in the data.

Big Data: The use of predictive analysis or other advanced methods to extract value from data (refers seldom to a particular size of data set).
Three V's:
- Volume (amount of data being processed)
- Velocity (ability to deal with data arriving very fast)
- Variety (data from different sources and with different formats)

Data Science: The creation of models able to extract patterns from complex data and the use of these models in real-life problems.

Model: A generalization obtained from data that can be used afterwards to generate predictions for new given instances.

Method: A systematic procedure that allows to achieve an intended goal.

Algorithm: A step-by-step set of instructions easily understandable by humans.

Hyper-parameter: Values set by the user (e.g., number of clusters in k-means).

Model parameter: Values that are set by a modelling/ learning algorithm in its internal procedure (e.g., slope of a linear regression).

Programming for data analysis

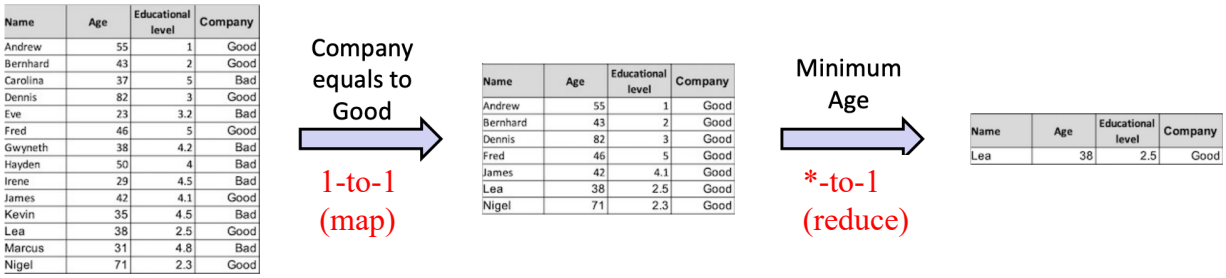
General

Many data analysis computations are performed over tabular data → pandas.
Data analysis programs are often defined as

- a sequence of transformations
- each transformation transforms one DataFrame into another DataFrame.

One-to-one transformation (map): This transformation transforms each row of the DataFrame into another row of a DataFrame. Each transformation can be performed independently.

Many-to-one (reduce) transformation: This transformation transforms multiple rows of the DataFrame into a single row of the new DataFrame. This transformation needs to get the data for all rows to be reduced.



Name	Age	Educational level	Company
Andrew	55	1	Good
Bernhard	43	2	Good
Carolina	37	5	Bad
Dennis	82	3	Good
Eve	23	3.2	Bad
Fred	46	5	Good
Gwyneth	38	4.2	Bad
Hayden	50	4	Bad
Irene	29	4.5	Bad
James	42	4.1	Good
Kevin	35	4.5	Bad
Lea	38	2.5	Good
Marcus	31	4.8	Bad
Nigel	71	2.3	Good

Name	Age	Educational level	Company
Andrew	55	1	Good
Bernhard	43	2	Good
Dennis	82	3	Good
Fred	46	5	Good
James	42	4.1	Good
Lea	38	2.5	Good
Nigel	71	2.3	Good

Name	Age	Educational level	Company
Lea	38	2.5	Good

→ The transformation from one DataFrame to another is useful, because we can use parallel computing, resulting into a shorter running time.

Pandas

General processing opertations

Load a CSV file into a DataFrame:

- pd.read_csv("filename")

Selecting rows based on conditions:

- df [df ["header"] == "condition"]

For combining multiple conditions use & (and) or | (or)

Selecting columns:

- df [["header_1", "header_2", …]]

Applying reduce/aggregation function:

- df_name ["header"].min()

Other functions: max, mean, median, std, …

Series: A series is a sequence of values with an index (like a column of a DataFrame)
Converting a series into a DataFrame:

- series.to_frame()

→ Useful, because many functions return series' instead of a DataFrame!

Applying different aggregation functions at once:

- df_name.agg({ "header_1" : "min" , "header_2" : "max", … })

Applying an aggregation function with (an) entire row(s) as an output

- df_name.nsmallest(#rows in output, ["header"])

Similar for nlargest

Grouping and applying reduce/aggregation function per group

- df [["header_1", … ,"header_i", …, "header_n"]].groupby (["header_i"]).min()

Combining DataFrames

Sometimes data to be processed is not in a single table / DataFrame. In this case, it is necessary to be able to combine multiple DataFrames. Two of the most useful operations are:

- *concat*: combines DataFrames by taking rows from each of the DataFrames.
  - pd.concat([DataFrame_1, DataFrame_2])

country	population
PT	10276617
ES	46937060
DE	83019213

+

capital	population	country
Brasilia	211049519	BR
Mexico City	127575529	MX
Montevideu	3461731	UY

→

country	population	capital
PT	10276617	NaN
ES	46937060	NaN
DE	83019213	NaN
BR	211049519	Brasilia
MX	127575529	Mexico City
UY	3461731	Montevideu

- *join*: combines DataFrames by taking columns from each of the DataFrames.
  - DataFrame_1.join(DataFrame_2, on = "header", how = "left")

The link between the two DataFrames is done by the column specified by on in DataFrame_1 and the index of DataFrame_2

country	population
PT	10276617
ES	46937060
DE	83019213

+

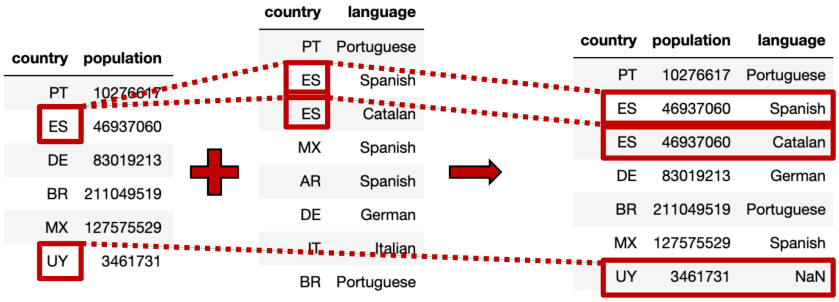
country	language
DE	German
PT	Portuguese
ES	Spanish

→

country	population	language
PT	10276617	Portuguese
ES	46937060	Spanish
DE	83019213	German

```
countries1 = population1.join(language.set_index("country"), on="country")
```

Left join: Join every row of the first table with all possible values of the second table. If no matching value exists in the second table, the value will be *NaN*.

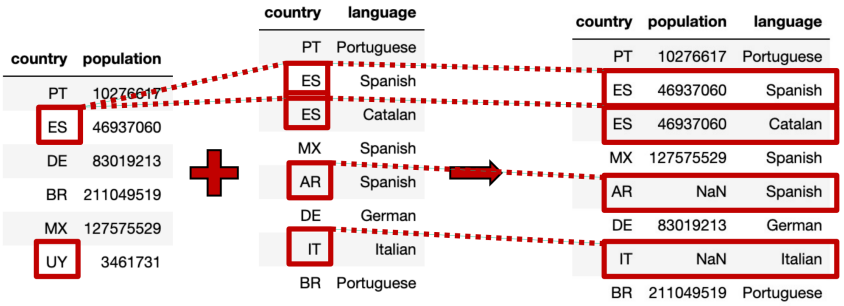


country	population
PT	10276617
ES	46937060
DE	83019213
BR	211049519
MX	127575529
UY	3461731

country	language
PT	Portuguese
ES	Spanish
ES	Catalan
MX	Spanish
AR	Spanish
DE	German
IT	Italian
BR	Portuguese

country	population	language
PT	10276617	Portuguese
ES	46937060	Spanish
ES	46937060	Catalan
DE	83019213	German
BR	211049519	Portuguese
MX	127575529	Spanish
UY	3461731	NaN

Right join: Join every row of the second table with all possible values of the first table. If no matching value exists in the first table, the value will be *NaN*.



country	population
PT	10276617
ES	46937060
DE	83019213
BR	211049519
MX	127575529
UY	3461731

country	language
PT	Portuguese
ES	Spanish
ES	Catalan
MX	Spanish
AR	Spanish
DE	German
IT	Italian
BR	Portuguese

country	population	language
PT	10276617	Portuguese
ES	46937060	Spanish
ES	46937060	Catalan
MX	127575529	Spanish
AR	NaN	Spanish
DE	83019213	German
IT	NaN	Italian
BR	211049519	Portuguese

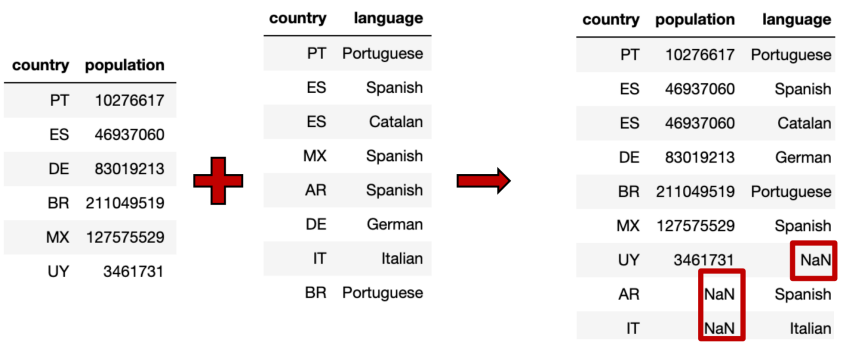
Inner join: Join every row of the first table with all possible values of the second table. If no matching value exists in the second table, the row will not appear.

country	population
PT	10276617
ES	46937060
DE	83019213
BR	211049519
MX	127575529
UY	3461731

country	language
PT	Portuguese
ES	Spanish
ES	Catalan
MX	Spanish
AR	Spanish
DE	German
IT	Italian
BR	Portuguese

country	population	language
PT	10276617	Portuguese
ES	46937060	Spanish
ES	46937060	Catalan
DE	83019213	German
BR	211049519	Portuguese
MX	127575529	Spanish

Outer join: Join every row of the one table with all possible values of the other. If no matching value exists in the other table, the value in that particular cell will be NaN.



country	population
PT	10276617
ES	46937060
DE	83019213
BR	211049519
MX	127575529
UY	3461731

country	language
PT	Portuguese
ES	Spanish
ES	Catalan
MX	Spanish
AR	Spanish
DE	German
IT	Italian
BR	Portuguese

country	population	language
PT	10276617	Portuguese
ES	46937060	Spanish
ES	46937060	Catalan
DE	83019213	German
BR	211049519	Portuguese
MX	127575529	Spanish
UY	3461731	NaN
AR	NaN	Spanish
IT	NaN	Italian

Time-series

A time-series is a series of values that have a timestamp.

a) *Univariate*: The time-series has a single feature
   E.g.: price in a stock index, mortality, birth rate
b) *Multivariate*: The time-series has multiple features
   E.g.: Weather (composed by temperature, pressure, humidity, etc.)

Moving Averages

In a time-series, it is common that the values are noisy because the observation vary abruptly from point to point (often due to an unprecise observation process)
Moving averages are useful in this situation, as they lead to a decrease of that noise.
A (simple) moving average of length *K* computes the average of *K* values.

Simple Moving Average

Normal

data["simpleMA"] = data["value"].rolling(K, center = False).mean()
→ rolling transformation allows to create groups of *count* rows, and apply a reduce/aggregation function to each group.

Centered

data["simpleMA"] = data["value"].rolling(K, center = True).mean()
Note: Pandas has no simple way of computing the centered moving average, as it should, with even number of elements.
Solution: It first computes the moving averages, which are assigned to the middle of two elements and then computes the average of the two averages.

Weighted Moving Average

Normal

The weighted moving average with length *K* of values, v_1, …, v_n, using weights w_1, …, w_k is defined by weighting the contribution of each element in the window

$$WMA_k(t) = \frac{\sum_{i=1}^{k} w_i * v_{t+i}}{\sum_{i=1}^{k} w_i}$$

data["weightedMA"] = data["value"].rolling(K).apply(lambda x: sum(weights*x)/sum(weights))
→ "weights" is a list that needs to be defined in advance (e.g.: [1, 2, …, K])

Exponential

The exponential moving average is a special weighted moving average, where the weighting factors decrease exponentially.

$$EMA(t) = \alpha v_t + \alpha(1-\alpha)v_{t-1} + \alpha(1-\alpha)^2 v_{t-2} + \cdots, \text{with } \alpha \in (0, 1]$$

data["EMA"] = data["value"].ewm(com = 0.2).mean()
→ Here, $\alpha = \frac{1}{1+com}$.

Correlations

In statistics, correlation is any statistical relationship between two random variables. Correlation is commonly used to refer to the degree to which a pair of variable are *linearly* related.
Note: Correlation does not necessarily mean causality!

Intuition:

- 1: Perfect correlation – the two variables move in tandem, i.e., in the same direction and with a related change
- 0: no correlation
- 1: Perfect negative correlation – when one variable increases, the other decreases with a related change

corrVals = dataDF[["Column1", "Column2"]].corr() →

∴	$Column1$	$Column2$
$Column1$	1	x
$Column2$	x	1

dataDF.corr() → computes the correlation between all the columns of the dataframe.

Shifted correlation

Sometimes the correlation is not immediate, but it takes some time (e.g., a person only dies from COVID several days after it has symptoms). Maybe a better correlation can be found by shifting the cases some days to align them with the deaths. → use .iloc

cases = data[["casesMA"]].iloc[:-12].reset_index(drop = True) → select all but the last 12 rows
deaths = data[["deathsMA"]].iloc[12:].reset_index(drop = True) → select all but the first 12 rows
→ shiftCorrVal = cases.corrwith(deaths["deathsMA"])

Parallel and Distributed framework

MapReduce: Overview

Sequentially read a lot of data
Map phase: Extract the important information
Group by key: Sort and shuffle the output of the map phase
Reduce phase: Aggregate, summarize, filter or transform
Write the result

→ Programmers only specify the map and reduce functions. The system is responsible for distributed execution, fault-tolerance, etc.

→ The map reduce system is responsible for the rest, e.g., partitioning, machine failures…

Both, Map and Reduce tasks can be executed in *parallel* → good for efficiency & runtime

Issues:

- Need to read and write files (on multiple machines, bc otherwise we might lose data)
- Underlying filesystem replication (for writers)

- One job must finish before the next can start

Programming with (Apache) Spark

Programs using Pandas API are translated into Sparks programs.
Programs are *optimized* to execute efficiently!
Spark programs encode the dependencies among various DataFrames (lineage graph)

Wide-Dependencies: df partition depends on multiple partitions stored on diff. nodes
e.g., groupBy, generic join, select aggregate → computationally expensive!
Narrow-Dependencies: df partition depends on data that is co-located (same node)
e.g., simple select, where, co-located join

import pyspark.pandas as ps → importing Pandas API

df = spark.read.option("header", True).option("inferSchema", True).csv(fileName)
df = df.withColumnRenamed("Educational level", "EducationalLevel")

From Spark Dataframe to Spark Pandas Dataframe:
pandas_sparkDF = sparkDF.pandas_api()

From Spark Pandas Dataframe to Spark Dataframe:
sparkDF = pandas_sparkDF.to_spark(index_col = "column")

Tables and Selection

It is possible to create a SQL view/ table from a Spark DataFrame.
Note:

- In SQL, a table stores a set of rows, each one with the same set of columns – *null* represents a value that does not exist.
- A view is the result of a query – it also contains a set of rows, each one with the same set of columns

→ Technically, Spark SQL mostly maintains views, as they are the result of queries. For simplicity, we often refer to Spark SQL views as tables

Registering a DataFrame as a SQL table:

```
# Read a CSV file into a DataFrame
dfPS = ps.read_cvs(filename)

# Convert Pandas Spark DataFrame into Spark DataFrame
dfS = dfPS.to_spark()
dfS = dfS.withColumnRenamed("Eductional level", "EducationalLevel")
dfS.createOrReplaceTempView("persons") → in SQL this DF will be known as "persons"
```

Executing SQL statements: spark.sql("statement")

Selecting columns from a table: SELECT columns FROM table (use * for all columns)
→ spark.sql("SELECT * FROM persons")

Selecting rows based on conditions: SELECT columns FROM table WHERE cond1 AND/OR cond2

→ spark.sql(“””SELECT * FROM persons
WHERE company == “Good” AND age > 30
”””””)

Create a view from a select result: CREATE OR REPLACE TEMPORARY VIEW table AS SELECT…

Ordering and Limiting the results:
SELECT cols FROM table [WHERE cond][GROUP BY cols][ORDER BY cols][LIMIT num]

- GROUP BY groups the rows by the values of the specified columns.
- ORDER BY allows to order the result by one or more columns. Use “ORDER BY col DESC” for ordering in descending order.
- LIMIT allows to return only the first N rows in the result.

Aggregation

SELECT fun(col), … FROM table [WHERE cond]…
Some functions: min, max, mean, percentile(col, 0.5), std, …

Nested Selects

It is possible to use the result of a SELECT statement in the definition of another SELECT statement. E.g., the following code computes the number of persons older than Fred.

result = spark.sql(“””SELECT count(*) FROM persons
WHERE age > (SELECT age FROM persons WHERE Name = “Fred”)
”””””)

Windows (e.g. Moving Average)

Spark SQL has no moving average function but it has support for defining windows using the following syntax:

Moving_Average = spark.sql(“””SELECT day, value, MEAN(value) OVER
(ORDER BY days ROWS BETWEEN offset PRECEDING
AND offset FOLLOWING) AS centerMA
FROM data).show()

Note: This is not exactly a moving average, as we get values for every day including the first ones

Note: offset can be UNBOUNDED PERCEDING | offset PERCEDING | CURRENT ROW | offset FOLLOWING | UNBOUNDED FOLLOWING

Windows operations can also be performed in different partitions independently – this allows to perform operations over groups

SELECT fun(col1) OVER (PARTITION BY col2 ORDER BY col3 ROWS BETWEEN offset AND offset) FROM …

→ This applies fun to values of col1 in the window defined independently in each partition of data, defined by the value of col2 and ordering the partition by col3.

Top-N for each group: → use nested SELECTs, PARTITION, ORDER BY and RANK

```
youngest = spark.sql(“””SELECT * FROM (
                        SELECT *, RANK(age)
                        OVER ( PARTITION BY company ORDER BY age ) AS
                        rank FROM persons
                        ) WHERE rank = 1”””)
```

Join

Join rows in table_left with rows in table_right using a condition. Different types of join are supported → INNER | LEFT | RIGHT | CROSS | FULL OUTER

```
join = spark.sql(“””SELECT * FROM table1 A INNER JOIN table2 B ON A.col = B.col”””)
```

→ For referring to table names, it is often convenient to use a smaller name (here A and B)

Miscellaneous

Wide and Long format

Wide: Each row is a record and each column includes the values of a given variable → pivot

Long: Each row is a value of a given variable

→ Allows to add new variable without changing the format, which can be beneficial!

→ unpivot

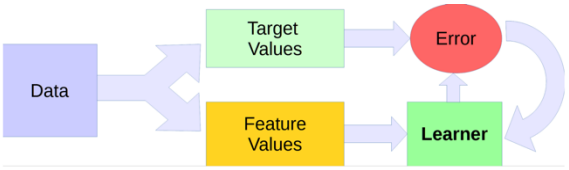
Long to wide : `df.pivot(index = cols, columns = cols, values = vals)`

Wide to long : `df.melt(id_vars = columns you don’t want to unpivot)`

Unstack

Unstack “pivots” a level of the row index to the column axis → df.unstack(col)

Introduction to Machine Learning



There are two classifications of Machine Learning:

1. Supervised learning
   → We are given a data set and already know what our correct output should look like, having the idea that there is a relationship between the input and output
   → Data includes values to predict (e.g. regression for continuous values, classification for discrete classes)

2. Unsupervised learning
   → Allows us to approach problems with little to no idea what our results should look like. We can derive structure form data where we don’t necessarily know the effect on the variables
   → Data does not include the target value. The goal is to find structure in the data

Supervised Learning

Regression

→ Trying to predict results within a continuous output, so trying to map input variables to some continuous function.

There are 2 important assumptions in linear regression:

- input variables are independent
- input variables have a linear relationship with the output variable

Linear regression aims to predict a real-valued output based on input values:

$$y = \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_{n+1},$$

where x_i is one dimension/input variable of the input space.

- with one input variable: $y = \theta_1 x_1 + \theta_2$ (θ_1 and θ_2 are called weights/parameters)

Let's assume y is a function of x plus some error: $y = F(x) + \epsilon$
We want to approximate $F(X)$ [the "true" function] with some $g(x, \theta) = \theta_1 x + \theta_2$
→ Find $g(x, \theta)$ that minimizes the error → Mean squared error
Square error is often written as: $\mathbb{E}[\theta|X] = \frac{1}{2}\sum_{t=1}^{n}\big(y_t - g(x_t|\theta)\big)^2$
Gradient Descent:
→ To find the parameters θ_1 and θ_2 for $g(x) = \theta_1 x + \theta_2$ that minimize the squared error

$$\mathbb{E}[\theta|X] = \frac{1}{2}\sum_{t=1}^{n}\big(y_t - g(x_t|\theta)\big)^2.$$

Outline:

- Start with some θ_1 and θ_2
- Keep changing θ_1 and θ_2 to reduce the squared error
- Until we end up reaching probably a minimum error

→ Linear regression, however, only makes sense if the data shows correlation
→ Otherwise: Try polynomial regression

Polynomial regression:
e.g., 3rd order polynomial regression: $y = \theta_1 x^3 + \theta_2 x^2 + \theta_3 x + \theta_4$

Overfitting

A statistical model begins to describe the random error in the data rather than the relationship between variables.
A regression model that becomes tailor-made to fit the random quirks of one sample is unlikely to fit the random quirks of another sample
→ A model that overfits too much one sample will most likely fail with unseen data

Training error: error measured in the training set.
Problem: not a good indicator of the error for new examples
Solution: Measure error outside the training set (estimate true error)

Errors Overview

Measureable errors:

- *Training error*: error measured in the training data and used to fit the parameters
- *Test error*: error measured on the test data to estimate the true error

Unmeasurable errors:

- *True error*: expected error for all data
- *Generalization error*: difference between True error and Training error

Validation and Selection

Holdout method

The dataset is separated into two sets, called the *training set* and the *test set*.

Problem: Choosing the lowest error on the training set makes the test error biased. The test error is strongly influenced by the data selected for the training and test sets

K-fold cross-validation

The original data set is divided into k sets. This results in k folds. This results in k partitions, where for each partition one of the folds is used as the test set and the remaining folds are used as training sets.
→ Average of the predictive performances (i.e., error) obtained in each partition

Advantage: By using several partitions, we reduce the chance of having the predictive performance influenced by the partition used and, as a result, have a more reliable predictive performance estimate

Bootstrapping

Bootstrapping is used to estimate uncertainty in the parameters
Idea: Our data is a representative sample of the universe of possible points. Hence, we can fake resampling that universe by resampling our data.
Method: Create replicas by resampling from the data. Create each replica of N points by picking at random from the original data. Thus, it can happen that some points will be picked more than one time, some points will be missing, but all replicas should be different. Then, we use each replica as a different experiment to collect statistics.

Wild Bootstrapping

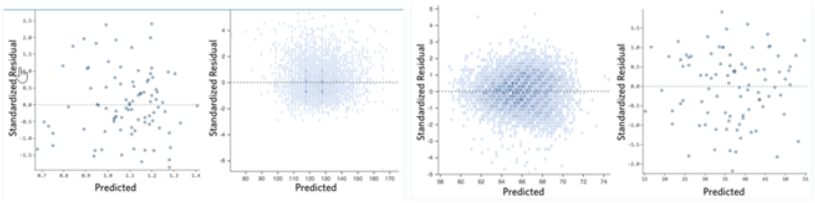
Bootstrapping cannot be used for time series, because picking years at random is nonsense
→ Wild Bootstrapping
Idea: True value = predicted value + normally distributed noise with different deviation at each point, for which we use the residuals.
Residuals: $\epsilon_i = y_i - \hat{y}_i$
Random variable n: $n \sim N(0,1)$ → standard normal distributed

→ Wild Bootstrapping points: $y_i^* = \hat{y}_i + \epsilon_i n_i$

Ideally, the Residual Plot should be normally distributed

1. Residuals are pretty symmetrically distributed, tending to cluster towards the middle of the plot.
2. they're clustered around the lower single digits of the y-axis (e.g., 0.5 or 1.5, not 30 or 150).
3. in general, there aren't any clear patterns.

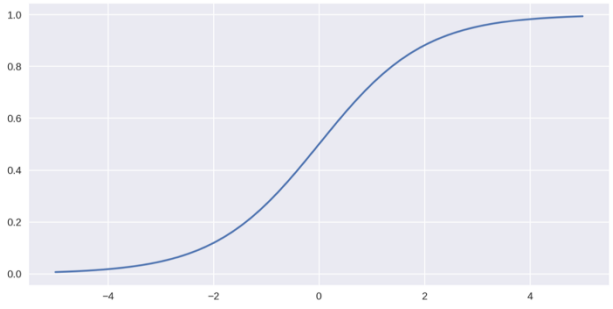
Examples of residual plots that meet previous requirements:



The logistic model

$$\text{Logistic Function: } y = \frac{L}{1 + e^{-k(x-x_0)}} + o$$

- L acts like the limit of the function
- k acts like the slope of the function → the smaller k the smoother (less steep)
- x_0 acts like the midpoint/turnpoint of the function
- o is the offset of the function, so the limit from the left



General recipe:

- Specify your cost function (e.g., MSE)
- Adjust the model parameters to minimize this cost function

Exemplary Code:

```
from scipy.optimize import minimize # Here we are going to use the minimize function

# Logistic Function
def logistic(x,x0,L,k):
    return L/(1 + np.exp(-k*(x-x0)))

# Cost function - Minimize mean squared error
def log_cost(params, data):
    x0,L,k,offset = params
```

```
    pred = logistic(data.iloc[:,0],x0,L,k) + offset
    return np.mean( (pred-data.iloc[:,1])**2)

st_params = [1980,1500,0.001,1500]
plt.figure(figsize=(10,5))
res=minimize(log_cost,st_params,args=(dataset_df))
x0,L,k,offset = res.x # Best parameters found

# Used the model found (logistic model with a set of parameters and an offset) to
predict the values
pred = logistic(dataset_df.iloc[:,0],x0,L,k) + offset

# Plot the data and the model built
plt.plot(dataset_df.iloc[:,0],dataset_df.iloc[:,1],'.')
plt.plot(dataset_df.iloc[:,0],pred)
```

Multivariate Regression

- Multiple linear regression: $y = \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n + \theta_{n+1}$
- Multiple non-linear regression: $y = \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_2 + \theta_4 x_2^2 + \theta_5$

Regression Summary

Why fit a regression model?

→ Fit a theoretical model to understand relations

- Measuring error is necessary to estimate the quality of the fit
- Bootstrapping (or wild) is useful for estimating uncertainty of the parameters

Interpolation: predict new values within the range of the data

- Need a test set for estimating error outside the training set
- If selecting model, more reliable to estimate true error with cross validation

Note: For now ignore the influence of categorial varibales.

Regularization

Change learning algorithm to reduce overfitting by constraining the model

→ Ridge regression $J(\theta) = \underbrace{\sum_{t=1}^{n}[y_t - g(x_t|\theta)]^2}_{\text{Normal regression}} + \underbrace{\lambda \sum_{j=1}^{m} \theta_j^2}_{\text{penalty term}}$

By introducing the penalty term, we force the magnitude of θ to be smaller.

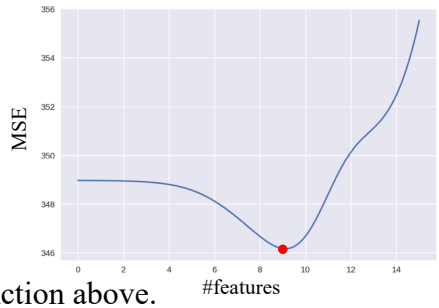
Code:

```
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score

alphas = np.linspace(0,15,100)
errors = []
for alpha in alphas:
    ridge = Ridge(alpha=np.exp(alpha))
    cv_err = cross_val_score(ridge, X, y,
                             scoring='neg_mean_squared_error',
                             cv=5)
    rmse = (-np.mean(cv_err))**0.5
    errors.append(rmse)

plt.plot(alphas,errors)
best_alpha = np.exp(alphas[np.argmin(errors)])
print(f'RMSE: {np.min(errors):.2f}, alpha: {best_alpha:.2f}')
```

Result:



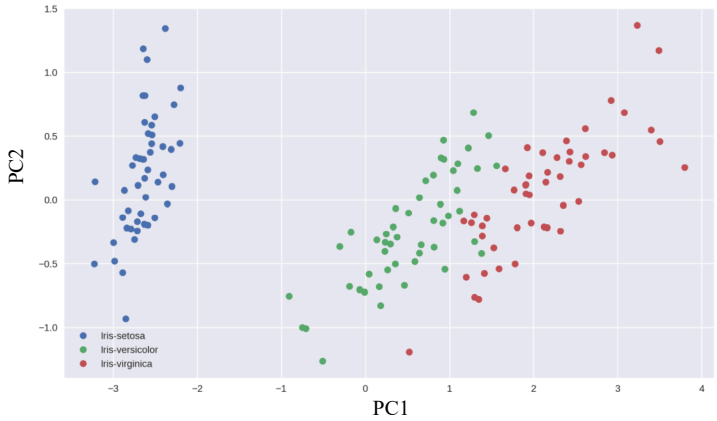
Note: in the code alpha represents the λ in the function above.
Note: The result is based on the best alpha. With this we compute the model based on training set

Dimensionality Reduction

Assume you have many features and want to categorize your data. One way would be to plot histograms and scatterplots in order to see the relation between the features for each class. This approach, however, is tedious.

Principal Component Analysis (PCA)

→ *Taking a high dimensional data set and reducing it to low dimensional graph*
PCA is a linear dimensionality reduction technique, which performs a linear mapping of the data to a lower-dimensional space in such a way that the variance of the data in the low-dimensional representation is maximized. It tries to preserve the essential parts that have more variation of the data and remove the non-essential parts with fewer variation.
Example graph:



Notes:

- The values of PC1 and PC2 do not have an interpretation
- The PC1 axis is more important than the PC2 axis in terms of similarity

→ Useful for identifying patterns and similarities in the data. By revealing how variables are correlated and how data points are grouped, PCA can highlight inherent similarities among observations in a dataset

Might not work in some cases:

- Highest variance components may not be most useful
- The number of components we want may not capture most variance
- Linear transformation may not be enough

t-Distributed Stochastic Neighbor Embedding (tSNE)

tSNE is a non-linear technique for dimensionality reduction that is particularly well suited for the visualization of high-dimensional datasets. It minimizes the divergence between two distributions: a distribution that measures pairwise similarities of the input objects and a distribution that measures pairwise similarities of the corresponding low-dimensional points.

Classification

Classes are linearly separable if they can be separated by some linear combination of feature values (a hyperplane). The frontier is a linear discriminant.

Hyperplane: A subspace whose dimension is one less than that of its ambient space, i.e. if space $\in R^2$, a hyperplane is a line, if space $\in R^3$, a hyperplane is a plane, …

Linear Discriminant:
The frontier is given by the function $y(\vec{x}) = \vec{w}^T\vec{x} + w_0$ that is positive on one side of the hyperplane and negative on the other, i.e. one class is positive, the other is negative.

Note: Fitting with the MSE does not lead to good results, because this pulls the discriminant towards distant points.

→ *Logistic Regression*:
Logistic Regression is a statistical method for predicting *binary* classes.
Assume there is a function $g(\vec{x}, \widetilde{w}) = P(C_1|\vec{x})$, where $\vec{x}$ is the vector of features and $\widetilde{w}$ represents the weights or coefficients that the logistic regression model learns during training. $P(C_1|\vec{x})$ is the probability of the instance belonging to class 1 and we want to find a discriminant so that $P(C_1|\vec{x}) = P(C_2|\vec{x})$.
→ Based on the features we choose the class with larger estimated probability.
→ The function g is typically the sigmoid function/logistic function: $\frac{1}{1+e^{-x}}$.

Assuming $g(\vec{x}, \widetilde{w}) = P(C_1|\vec{x})$, this implies that the plane is:

$$P(y = 1|x; \theta) = 1 - P(y = 0|x; \theta)$$

Further,

$$g(\vec{x}, \widetilde{w}) = \frac{1}{1 + e^{-(\vec{w}^T\vec{x} + w_0)}}$$

The parameters $\vec{w}$ are found by maximum likelihood.

Regression vs. Classification:
Regression: Fit the data as closely as possible to predict continuous values
Classification: Find discriminant between discrete classes

Maximum margin classifier

Margin: A margin is the distance between the hyperplane and the nearest data point from either class. The goal of SVM is to choose a hyperplane with the greatest possible margin between the hyperplane and any point within the training data, offering a buffer that helps correctly classify new data points.

Margin classifier: A classifier that provides the distance to the discriminant, i.e. the hyperplane separating the classes is a linear classifier.

Maximum Margin Classifier: Finds the maximum-margin discriminant maximizing the distance to the nearest points to separate.
→ Reduces overfitting by constraining the frontier

Problem: Difficult to compute because of discontinuity

Kernel Trick

Kernel: transforms an input data space into the required form.
Kernel trick: The kernel takes a low-dimensional input space and transforms it into a higher dimensional space. Within this higher-dimensional space, the classes are "linearly" separable. This separation is then transformed back to the initial lower-dimensional input space.

- Linear Kernel
- Polynomial Kernel (can distinguish curved or nonlinear input space)
- Radial Basis Function Kernel (can map input space in infinite dimensional space)

Unsupervised Learning

Classification vs. clustering

Classification

- Class labels are part of the data
- Classes used in training, generally to predict labels for additional data

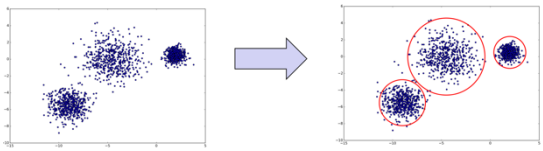
Clustering

- "Classes" (clusters) are unknown and inferred by the model
- Although descriptive of the data structure, clusters are used to predict classes

Clustering

Basic idea: Group similar examples together

- Min difference (or max similarity) within clusters
- Max difference (or min similarity) between clusters
- Clustering requires some measure of similarity



Why do we need clustering? → To help understand data and relations!

What can we do with clustering?

- Find useful classes of examples
- Find outliers
- Find relations between entities and groups (e.g. hierarchical clustering)

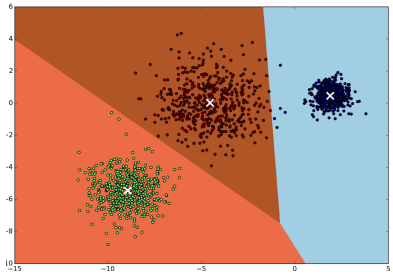
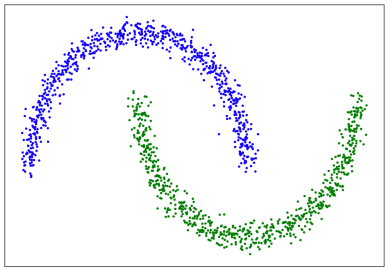
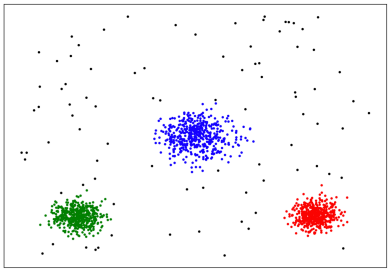
- Summarize data

Partitional Clustering: Data is divided into groups at the same level
Hierarchical clustering: Clusters are nested within larger clusters in a tree

Exclusive clustering: Each example belongs to only one cluster
Overlapping clustering: Examples may belong to more than one cluster
Fuzzy clustering: Each example belongs to clusters with $w_k = [0,1]$
Probabilistic clustering: each example belongs to clusters with $w_k = [0,1]$, where $\sum_k w_k = 1$

Complete clustering: All examples are assigned to cluster(s)
Partial clustering: Some examples remain unassigned (e.g. noise, irrelevant data, etc.)

Types of clustering

Prototype-based	**Contiguity-based**	**Density-based**
Define a (set) of prototype(s) for each cluster. Examples belong to the cluster that has the closest prototype	Closer in cluster than outside	High-density regions, discarding noise
		

Algorithms: K-means

Problem to solve: Find the best clustering of K clusters

- Define clusters by proximity to the mean of the cluster
- The number of centroids and clusters is predefined

→ Partitional, exclusive, complete and prototype based

Pseudocode:

Start with K centroids
Loop

- Assign each example to closest centroid
- Update centroid to mean of respective cluster
- Recompute clusters
- Repeat until convergence / or given number of iterations is reached

K-means initialization: What should be the initial K centroids?

- Random: Start with the coordinates of K random elements
- Forgy: Randomly assign elements to partitions and use the centroids of each partition

→ According to literature, the random approach is better

K-means distance:
Different distance measures can be used. Euclidean is usually used, but Minkowsky is more general:

$$D(x,y) = \sqrt[p]{\sum_d |x_d - y_d|^p}$$

Clustering validation: Silhouette Score

Silhouette score measures how close an element is to its own cluster (*cohesion*) compared to other clusters (*separation*).
For element i:

- $a(i)$: average of distance of i to all points in the same cluster
- $b(i)$: average of distance of i to all points in the nearest cluster (lowest average distance)

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))}$$

- $s(i) = [-1, 1]$: The higher the score, the more appropriately point i is clustered
- Silhouette coefficient is the mean of Silhouette Scores of all elements

Number of clusters

The number of cluster is a parameter for K-means
When K is unknown, the following strategy is often used:

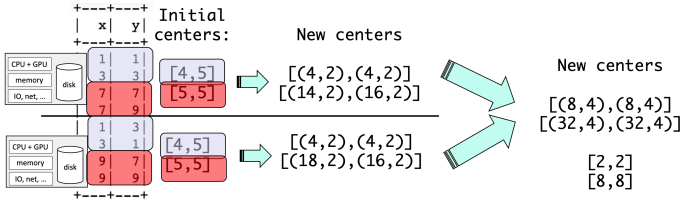
- Compute the clustering for different values of K;
- Find the K with best quality / where quality stops improving fast.

When using the Silhouette coefficient, this is often known as Silhouette analysis

Clustering @Spark

Where can distributed processing frameworks help?

- Data size
  - K-means: Computing the sum for subsets, by keeping track of size of subset [sum, count]



  - Each machine only needs to load a partition of the data – helps supporting *larger data sets*
  - Each machine only does a subset of the computations – *faster* computations
- Number of models
  - Use different algorithms for training and experiment which works better – *faster* computations

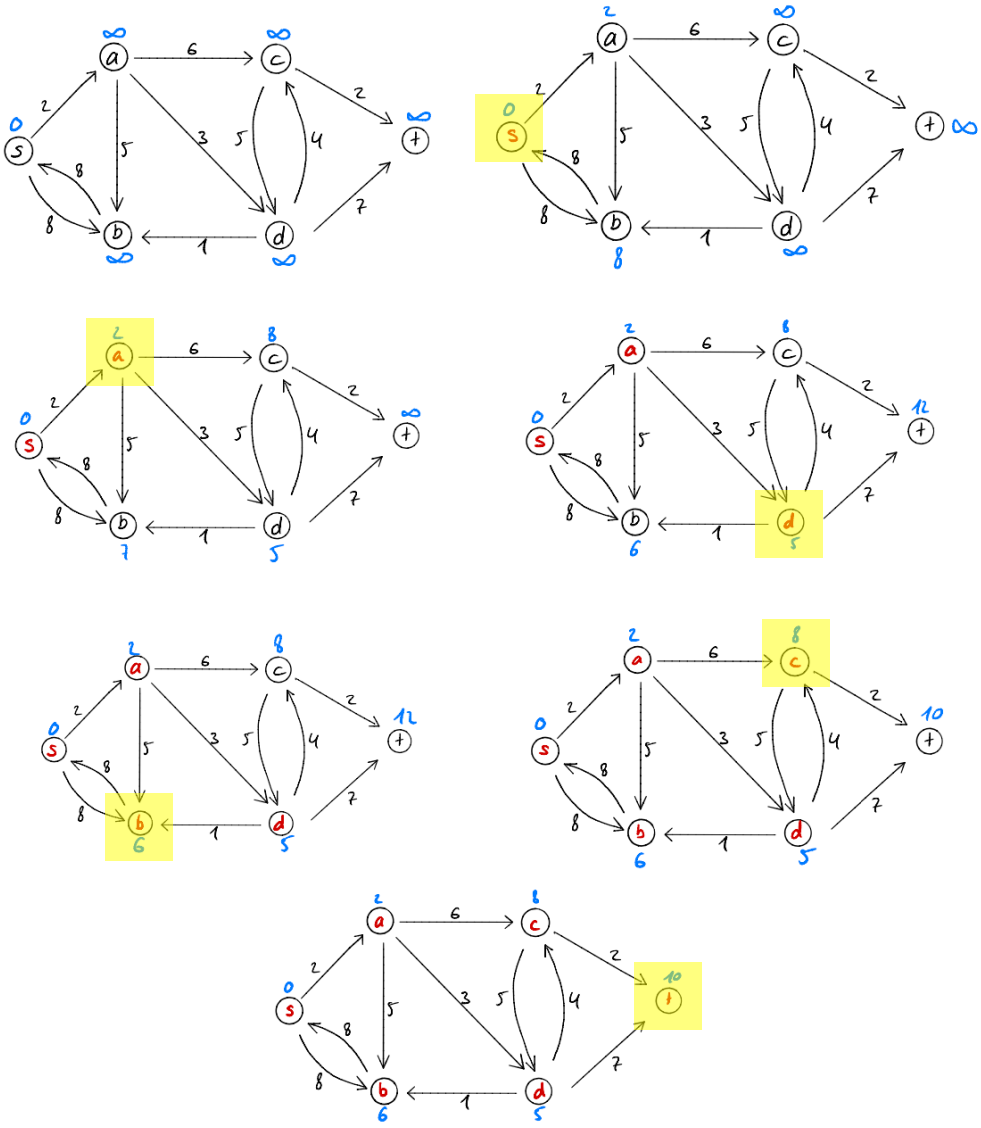
- Compute different models for the same algorithm in parallel – *faster* computations

Graph Algorithms

A graph is a structure composed by *vertices* and *edges*. Edges can be both, directed and undirected. Graphs with directed edges, *arcs*, are called *directed graphs*. Graphs, that have edges with associated values are called *weighted graphs*.

Dijkstra – Shortest Path Problem

When to use: Non-negative arc lengths with a specific starting point



→ Don't stop once you have the first value for a path to t but only stop when you look at t.

Compute importance of web pages – PageRank

Key ideas:

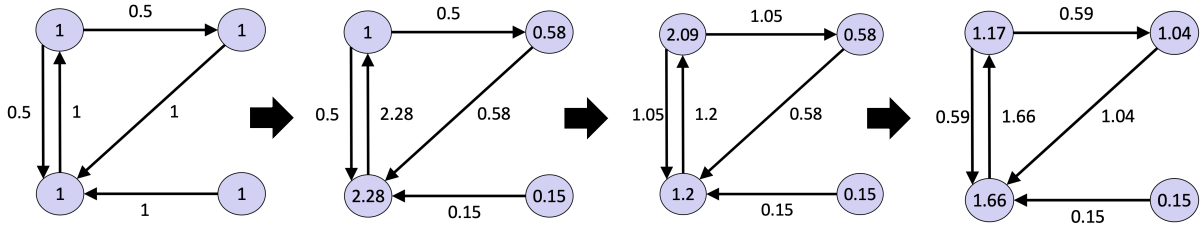
- The more links to a page, the more important it is

- Importance of links depends on the source page

Intuition: In each step, each node propagates to neighbors its page rank, divided by the number of edges.
The page rank of a node is (with $R[i]$ the page rank of page i):

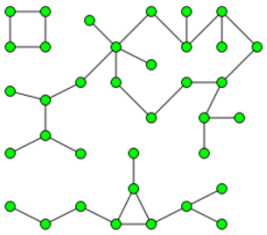
$$R[i] = 0.15 + 0.85 \sum_{j \in \text{Outlinks}(j)} \frac{R[j]}{\text{Outlinks}(j)}$$

e.g.:



→ Iterate until convergence (or until changes are small)

Connected components

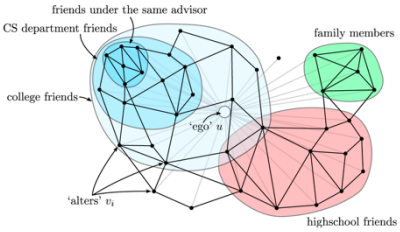


Find out connected components in a graph
Connected component: A subset pf vertices, for which any pair of vertices is connected directly or indirectly and no vertex is connected with any other node outside of the component

Use cases:

- Find related people in e-commerce sites:
  - All people using the same address, credit card, etc.

Community detection



Often networks are organized into clusters, communities
Goal: Find densely linked clusters

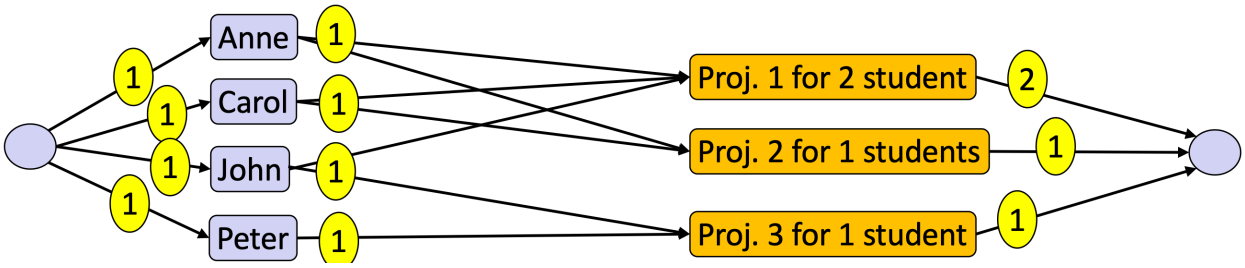
Use-cases:

- Discovering social circles
- Customer Segmentation
- Recommender systems

Maximum-Flow

Computes the maximum flow between two vertices

- Use-cases: Match making



Graphs in Spark

Creating an edge: → must have columns src, dest. Can also have additional columns that can be used in queries

```
edges = spark.sql( """SELECT
         input AS src,
         output AS dst,
         COUNT(*) AS cnt
         FROM transactions
         GROUP BY input, output""")
preprocDF.createOrReplaceTempView("edges")
```

Creating a vertex: → must have a column id(entifier). Can also have additional properties.

```
vertex = spark.sql( """SELECT FIRST(input) AS id, MIN(time) AS since
                FROM inputDF GROUP BY input
            UNION
                  SELECT FIRST(output) AS id, MIN(time) AS since
                         FROM outputDF GROUP BY output """)
```

Creating a graph:

```
from graphframes import *

g = GraphFrame(vertices, edges)

motifs = g.find("(a)-[e]->(b); (b)-[e2]->(a)")
motifs.show()
```

Introduction to cloud-based Data Analysis ecosystems

CPUs, GPUs, TPUs and data analysis

The *central processing unit (CPU)* is the main processor of a computer and runs generics instructions. It is responsible for executing the code of programs.

- A CPU runs one instruction at a time
- Modern processors include many CPUs (or cores)

A *graphics processing unit (GPU)* is a specialized processor designed to accelerate the creation of images.

- Modern GPUs may include thousands of cores, with each one executing the *same* operation at the same time on different data

A *tensor processing unit (TPU)* is a specialized processor designed by Google to accelerate AI processing, specifically neural networks machine learning. It is similar to a GPU, but optimized to do matrix operations (which are the base of neural networks)

- Recent TPUs have tens of thousands of units that execute operations in parallel.

→ Parallel processing speeds data analysis up!

Data analysis/processing models

Data analysis/processing tasks consist in executing computations over data
Two main data processing models:

1. *Batch processing* consist in performing computations over a dataset that is available when the computation starts
   → Goal: Execute computation over data and produce result. All data must be known when processing starts
   Note: For different data analysis processes data will be in different data storage systems and after processing data must be stored in some data storage system.
2. *Stream processing* consists in doing continuous computations over data that is being created/received.
   - Need to process data as it arrives
   - Need to be able to process data from multiple sources
   - Need to tolerate faults

   a. Continuous
      - Each tuple processed as it arrives
   b. Mini-batch
      - Tuples received for each X ms grouped in a mini-batch and then process mini-batches

   → When doing stream processing, it is often interesting to compute results based on data from a given interval, but compute results more frequently than this time interval.

   Stream processing systems typically load streaming data from intermediate publish-subscribe (PS) systems; results can be stores in files, databased or presented in dashboards.

Data storages

I. Blob stores
   - Used to store large data "files"
     - Immutable
     - Append-only
   - Inexpensive storage
   - Common uses: Read/write the whole file, logs, etc.

II. Relational (SQL) databases
   - Used to store structured data - tabels
   - More expensive than blob storage
   - Common uses: Queries, updates to rows in tables, store general application data

III. Key-value stores / Document databases
   - Used to store structured data – document (more scalable than SQL databases)
   - More expensive than blob stores and relational (SQL) databases
   - Multiple storage servers around the world (more secure)
   - Common uses: Queries, updates to documents, store general application data

I. Graph database → store graph-based data
II. Analytical stores → Keeps a copy of the db used by user-facing services, to allow data analysis without impacting running services.

Data processing platforms often include *specialized distributed file systems*, designed to support efficient data parallel processing.

Time-series databases:
Database specially designed to process time-series in real-time.
Goal: applications often want to detect state transitions – e.g. from working correctly to incorrectly
Challenges:
- Speed is more important than full accuracy
- Impossible to store all data

Privacy-preserving Data Analysis

Security of data in public cloud

Concerns regarding security/privacy:
- Location of data (different countries have different laws regarding data)
  - Cloud platforms typically allow users to specify which data centers to use for storing data and for executing computations (not all: e.g. free Google Colab)

Large cloud providers implement encryption at rest for customers' data, by default

Encryption at Rest: Whenever data is stored on disks, the data will be encrypted. Several layers of encryption occur.
- Impossible to access data without encryption key
- Considered a best practice to meet regulatory requirements: GDPR, HIPAA
- Security in case of remote or physical access to disks

→ When using the cloud, one needs to be careful which data centers will be used for storing data and performing computations
→ Cloud providers provide encryption at rest for improving security and privacy of data

Privacy-preserving data sharing

Anonymize: Guarantees that form the anonymized data it is not possible to identify the original data (typically worried about identifying the individuals/ subject of the data).

K-anonymity

Quasi-identifier: A set of attributes that, when combined, can lead to the identification of individuals/subjects.

A dataset is *k-anonymous* if, for each unique combination of quasi-identifiers, there are at least k-1 other individuals with the same combination.

Suppression: Removing certain values from the dataset
Generalization: Replacing specific values with more general or less precise values
Bucketization: Grouping values into buckets to achieve anonymity.
Other techniques:

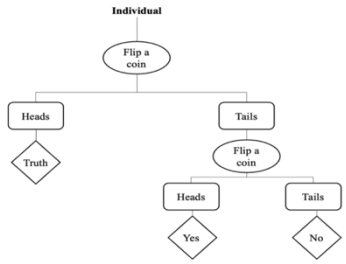
- *Data swapping*: Changing values between similar samples
- *Noise addition*: Adding noise to values (e.g. change the age of a sample from 22 to 23)

Differential privacy

Differential privacy algorithm: Guarantees that by observing a result that includes the data for some user, it is impossible to learn more about the user than by observing the result without the data from that user.
Car example (average speed): Each car adds some random noise to its speed.
Problem: If each car sends its speed multiple times, one can find out its speed.

LaPlace mechanism: Adding the noise with a pdf from the LaPlace distribution
Gaussian mechanism: Adding the noise with a pdf from the Gaussian distribution
→ Both mechanisms cannot be used with categorical variables

Randomized mechanism: Adding uncertainty to the value reported, i.e. reporting the real value with a given probability; otherwise a random value is reported.



Privacy for statistical information:

- Counts, averages, etc.,
- Histograms,
- CDFs,
- Clustering,
- Classification, and many more…

Question: Compute the average of Uber driver tips, but drivers don't wan to disclose their tips.
Question: Compute how frequent the use of credit cards was

Privacy-preserving computation

Multi-party computation

We have multiple parties that do not trust each other. They want to compute some function of their collective inputs, without revealing the inputs to others.

- Single central entity in which parties trust

→ Send the value to central entity & it announces the result
Issue: The central entity knows the values and can leak them

- Multiple central entities that do not collude
  → Each party splits its value and send a one part to each central entity. Final result is the sum of partial sums of each central entity
  Issue: What if the central entities collude?
- No central entity
  → First node send its value + large random value; other nodes add their value. Upon receiving the value, the first node subtract the random and announces the result
  Issue: If x1 and x3 collude, they can know about x2.
- Combination of 2nd and 3rd bullet:
  - Each party splits its value into shares and distributes them to the others.
  - Party 1 adds a random value to its sum and sends it to Party 2.
  - Each subsequent party adds its sum of received shares to the value received and passes it on.
  - The final value from Party N is sent back to Party 1.
  - Party A subtracts the random value and announces the total sum.

Note: General purpose multi-party computation is slow.

Homomorphic encryption

Motivation: Consider you want to store data in the cloud, but data must remain private. Note, you also want to be able to make changes.
→ Encrypt data – upload data – *changes* – download data – decrypt data – make changes – encrypt data – upload data is too slow!
→ What we need: Be able to execute computations over encrypted data

Homomorphic encryption is a form of encryption that allows computations to be performed on encrypted data without first having to decrypt it.

Partially homomorphic encryption: Work only with a single operation
E.g. (RSA)
$E(m) = m^e mod\ n \rightarrow E(m_1) * E(m_2) = m_1^e m_2^e\ mod\ n = (m_1 * m_2)^e\ mod\ n = E(m_1 * m_2)$
→ Multiplying the encrypted value of m_1 and m_2 is equal to encrypting the value of the product of m_1 and m_2.

There are multiplicative (ElGammal) and additive (Paillier) partial homomorphic encryptions.

A full homomorphic encryption was discovered in 2008 by Craig Gentry.

Partial homomorphic encryption: Not too slow.
Full homomorphic encryption: Very slow.