Streaming Data Analytics

LaTeX by Min Soo Jeon
mjeon@uc.cl
Professor Emanuele della Valle
Politecnico di Milano

Semester 2024-2

Disclaimer:

This material is an unofficial resource created for educational purposes only. The authors and contributors do not assume any liability for errors, inaccuracies, or misinterpretations that may arise from its use. Readers should verify information from official sources.

Version 1.0.0

Introduction

These notes are based on the lectures delivered by Professor Emanuele della Valle during the Streaming Data Analytics course at Politecnico di Milano during Semester 2024-2. Official course information can be found on the official website (della Valle, 2024b). Some sections, particularly the practical components, were taught by Teaching Assistants Alessio Bernardo, Federico Giannini, and Giacomo Ziffer. Further details on the practical part can be found in the course repository (della Valle, 2024a).
This material may be outdated, the latest version is available in the repository. To report an error, please contact mjeon@uc.cl.

Contents

1 Introduction

Traditional vs Continuous analysis of data The classical approach we use to analyze data is to get it and then take a whole batch and analyze it.
In the continuous approach, it is possible to analyze the data online, while new data is being generated.

Key ingredients

1. Sensor and actuators
2. Connectivity
3. Streaming Data Engineering
4. Streaming Data Science

Online models Changes cause Machine Learning Models to lose accuracy. Then, we have to be able to deploy models that could adapt to the new data

2 Four Streams

2.1 Stream take Many forms

There are myriads of tiny flows that we can collect. For example in physical or software sensors.
There are massive flows that we cannot stop. For example, Telco's monitoring or physical and cyber alarms. Anyway, we can transform these flows into torrents (data that we need, in a convenient form) that can trigger actions, for example to physical or software actuators

2.2 Time models

In Streaming Data Engineering, we model our torrent based on our needs. We have to design the models according to the type of queries we want to do. We don't create our models based on the data, but the needs.
For example, if we want to know which is the first event that occurred, it is not necessary to continue reading data if we already know it. On the other hand, if we want to know which is the last event that has occurred, there it is important to keep the streaming continuously. For this reason, we have different time models. Choosing them well implies having a good model without wasting resources.

2.2.1 Stream-only Time Model

We want to perform the queries, just with the information of the order in which the events occurred. In this type of model, the answer can change over time, but once the events are processed, they can be forgotten.

2.2.2 Absolute Time Model

If we need a query where we ask about the specific time something occurred, we have to use a more expressive model, the absolute time model. In this type of model, a time feature is assigned to each event, then we can compose queries taking into account the time. For example, ask about "the last 5 minutes", "the first hour", "within 5 minutes", etc.

2.2.3 Interval-based Time Model

If we need to store not only punctual events but also events with duration, we have to use Interval-base Time model. In this case, we can check things like overlapping, how much an event lasts, how much time 2 events occurred at the same time, etc.

2.2.4 Trade off

There is a trade-off between latency and expressiveness of the model (how complex queries it supports). More expressive models very often imply more response latency, since they require checking whether an event occurred or not for a longer time. In other words, the window opened must be longer in order to execute the query correctly.
Stream-only Time model has low latency and low expressiveness. Absolute time model has medium latency and expressiveness. Finally, Interval-based Time model requires high latency and expressiveness.
Queries with high latency and low expressiveness don't need to be performed in streaming, while queries with low latency and high expressiveness are not viable.

2.3 There's no solution for all streams

An Event-based system is a software architecture paradigm in which data flows and operations are driven by events rather than following a predefined control flow.

1. Events:
   An immutable and append-only stream of "business facts". Data that capture the state of something at some point.

2. Decoupling:
   The components are decentralized in the freedom to act, adapt, and change

There are different types of EBS (Event-Based Systems)

1. Content-based system:
   There's a unique publisher that filters and routes messages based on the actual content within the messages

2. Topic-based system:
   The messages are organized by topic. There are publishers that can send messages to specific topics. Subscribers received all messages sent to the topics they have subscribed. These types are the most used, with some examples as MQTT or Kafka.

2.4 Data Stream Management Systems (DSMS)

Data streams are unbounded sequences of time-varying data elements representing an almost continuous flow of information. This requires a change in the paradigm. From persistent data (one-time semantics) to transient data (continuous semantics). The continuous queries are performed over windows.

2.5 Incrementally calculus

Incremental calculus is necessary for streaming data analytics since it doesn't have the capacity to store every event to calculate something over and over. However, not every operation can be performed with incremental calculus.

1. It is possible
   Count, sum, max, min, and mean have incremental operators. Specifically, Variance and Standard deviation are not straightforward. The Welford's algorithm for standard deviation is the following:

Algorithm 1 Welford's Algorithm

Require: n=0: n° points already processed
mean=0: mean of points already processed
M2=0: sum of squares of differences from the current mean
Ensure: Standard Deviation
$n \leftarrow n + 1$
$delta \leftarrow x - mean$
$mean \leftarrow mean + delta/n$
$M2 \leftarrow M2 + delta(x - mean)$
$variance \leftarrow M2/(n - 1)$
$std\_dev \leftarrow sqrt(variance)$

2. It is not, because in the worst case it is necessary to revise the whole data
   - Median: It could be done in an almost approximated way (Jain & Chlamtac, 1985)
   - Percentile: It could be done in an almost approximated way (Greenwald & Khanna, 2001)
3. It is not because it is necessary to pass twice over the data
   - Mode and Entropy
   - Skew and Kurt
   - Covariance and Pearson Corr.

   All of them could be done in an almost approximated way (Cormode & Muthukrishnan, 2005)
4. It is not, because in the worst case it is necessary to revise the whole data, and there are no approximations for the calculus, but it is possible to calculate it in a window (this is true for every operator)

- Total order
- Distinct

2.6 Windows

A window is a space in the time where the DSMS focuses on performing the query. It could exist physical and logical windows.

- **Physical windows** are windows that are defined by event counting.
- **Logical windows** are windows that are defined by time counting.

We have three kinds of windows based on how they are placed into the timeline.

1. **Tumbling:** A size is defined, and the windows are along each other but not overlapping. The examples with each type could be:
   Physical window: every 10 events
   Logical window: every 10 minutes
2. **Sliding:** A size is defined, and the window covers from $\langle size \rangle$ before the event arrives. The examples with each type could be:
   Physical window: Last 10 events until the triggered event
   Logical window: Last 10 minutes until the triggered event
3. **Hopping:** A size and a stride are defined. The windows can overlap if strides are smaller than the sizes. There could also be gaps between windows if strides are bigger than sizes. The examples with each type could be:
   Physical window: the last 10 events every 5 events
   Logical window: the last 10 minutes every 5 minutes
4. **Session:** It defines a gap where the window is active. If another event arises before the gap finishes, the window extends for $\langle gap \rangle$ again. The examples with each type could be:
   Physical window: are not possible
   Logical window: events closer than 5 minutes (or inactivity gap larger than 5 minutes)

2.7 CEP semantics

CEP (Complex Event Processing) systems add the ability to deploy rules that describe how composite events can be generated from primitive (or composite) ones

Relation	Illustration	Interpretation
$X < Y$ $Y > X$	X Y	X precedes Y Y is preceded by X
$X \mathbf{m} Y$ $Y \mathbf{mi} X$	X Y	X meets Y Y is met by X (*i* stands for *inverse*)
$X \mathbf{o} Y$ $Y \mathbf{oi} X$	X Y	X overlaps with Y Y is overlapped by X
$X \mathbf{s} Y$ $Y \mathbf{si} X$	X Y	X starts Y Y is started by X
$X \mathbf{d} Y$ $Y \mathbf{di} X$	X Y	X during Y Y contains X
$X \mathbf{f} Y$ $Y \mathbf{fi} X$	X Y	X finishes Y Y is finished by X
$X = Y$	X Y	X is equal to Y

Figure 1: Allen's interval algebra

CEP has the ability to model the most expressive queries

	Stream only	Absolute time	Interval-based
EBS	x		
DSMS	x	x	
CEP	x	x	x

Figure 2: Service types and model tools

2.8 From tight to loose temporal coupling

There are two types of architectures that a system could implement.

1. Service oriented architecture:
   Must answer each request timely.

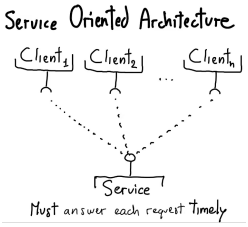

Figure 3: Service Oriented Architecture

2. Event-driven architecture:

Writes once. Clients read many times asynchronously. The service is decoupled from the clients with an event store. The clients speak through writing events. Then, it is put in the queue and the service can reply upon availability.

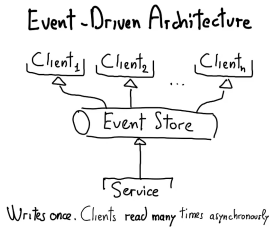

Figure 4: Event-driven Architecture

2.9 Tools

There are different tools and languages we can use to implement this theory. The scalability can be split into 2 groups. Vertical and horizontal scalability.
Vertical Scalability means that when we need more computational power, we need to acquire a larger system.
Conversely, in horizontal scalability, when we need more computational power, we need to acquire more systems, instead of larger ones.

3 Esper and EPL

3.1 What's Esper

It is a Complex Event Processing and Streaming Analytics EsperTech's software. It turns large volumes of disparate event series or streams into actionable intelligence.
Streaming Analytics is another way to say DSMS
It is written in C++ and Java. Plus, it has an open source part, but to be used for commercial purposes a license is needed.

3.2 What's EPL

It is a rich language to express rules or statements. As in the DSMS approach, it offers

- stream-to-relation operator as windowing
- relation-to-relation operators as selection, projection, join, aggregate, etc
- relation-to-stream, operators to produce output. For example output clauses, dstream, estream, etc.

As in the CEP approach, it offers comprehensive pattern detection.
Queries combined can be thought as a DAG (Directed Acyclic Graph), since we can create tables that can provide information for a post query. Then, the outputs can come from a successive concatenation of queries.

3.3 EPL syntax

3.3.1 Event declarations

To declare an event type, the following syntax is used

```
create schema <schema_name>(
<property_name1> <property_type1>,
<property_name1> <property_type1>,
...,
<property_name1> <property_type1>
);
```

3.3.2 Basic queries

To make a query, the following syntax is used

```
select <sch1>.<feat1>, <sch1>.<feat2>, ..., <sch2>.<feat3> -- (Or *)
from <schema1_name>.<windowing1> as <sch1>,
    <schema2_name>.<windowing2> as <sch2>, -- windowing and nicknames are
    optional
    ...,
    <scheman_name>.<windowingm> as <schm>
where <condition_1>, ..., <condition_n>
group by <feati>, ..., <featj>
having <grouping search conditions>
output <output_specification>
order by <featk>, ..., <featl> DESC
limit <num_rows>
;
```

Doing the 'group by' aggregation clause will change the output mode to one that gives you an output when something goes out as well. Note that this remark has only sense in Logical types, since in physical windows, when an event goes in, another goes out necessarily
Another style instead of the from-where is

```
from <variable_name>=<schema_name>(<condition_1>,..., <condition_n>),
```

The difference, in terms of results, between these two, is the query velocity.
When we use the clause WHERE, data points are entering into the query that filters them. When we use ($\langle condiction \rangle$) syntax, data points are filtered before flowing into the query, and the query execution doesn't get triggered, making it much more efficient.
Summarizing, the second one is better.

3.3.3 Windowing

For windowing syntax, the following syntax is used

- **Logical Sliding:**

```
        win:time(time_period) -- or
        #time(time_period)
```

- **Logical Tumbling:**

```
        win:time_batch(time_period) -- or
        #time_batch(time_period)
```

- **Physical Sliding:**

```
win:length(size) -- or
#length(size)
```

- **Physical Tumbling:**

```
win:length_batch(size) -- or
#length_batch(size)
```

When no window is indicated, the system assumes the Landmark Window Function. This function starts from the beginning and each time an event arrives, it uses a window from the beginning up to that event
The time window does not consider the end of the interval.
Note that every time, # can replace to win:

3.3.4 Output-control

For output-control syntax, the following syntax is used

```
output <all | first | last | snapshot>
every <output_rate> <seconds | events>
```

The meanings of the output clauses are the following

- **first** Ensures that the first result generated by the query is output immediately in the specified interval
- **last** output the most recent result at the time of the query's execution, discarding any intermediate results in the specified interval
- **all** outputs every result generated by the query, including both current and previous calculations, in the specified interval. If the previous events don't update, it will write null.
- **snapshot** outputs only the current state of the query at the specified interval. Unlike all, it doesn't include previous results, i.e. results of previous intervals

It makes no sense to use output control sizes smaller than the size in tumbling windows, since Esper implements it by accumulating batches in a buffer until the size is completed. The output clauses change the sliding windows behavior, since it goes from an event-to-event output to a fixed period output.

3.3.5 Derivated tables

Just writing

```
insert into <name_new_event>
```

before a query, we are wrapping it down in new events instead of printing it up. Later, the new event can be called as any event in the queries

3.4 Pattern matching

An event pattern is emitted when one or more event occurrences match the pattern definition, which can include constraints on the content or time occurrence of events, and the inner conditions for the pattern creation/termination.
A pattern should be placed after the from clause. It must be initialized with the clause "pattern", followed by [ ] brackets wrapping the pattern itself.
The pattern always, although we say otherwise, catches only the first appearance or match.
Some of the operators are the following:

- **and:** Logical and
- **or:** Logical or
- **not:** Logical not, but it only revises the no-occurrence if the match has not occurred before.
- **− >:** The left one comes before the right one
- **every:** Do the pattern for every element wrapped by the clause that matches
- **timer:within(x min):** It is used after where clause. Indicates that the previous expression has to occur all in x minutes
- **every-distinct:** PENDING
- **[num]:** PENDING
- **until:** PENDING
- **timer:withinmax(x min):** PENDING
- **while-expression:** PENDING

Inside the pattern, the events could be assigned with a name with this notation:

```
    <variable> = <event>(conditions,..., parameter=<another_variable>.<
    feature>)
```

Note that we can't use WHERE notation here, because it would filter the results a posteriori, causing that, a priori, the pattern misses our desired events.
The time saved is the time when the pattern closes.

3.5 Example

Let be the following the schema definitions

```
create schema TemperatureSensorEvent (
sensor string,
temperature double
);

create schema SmokeSensorEvent (
sensor string,
smoke boolean
);
```

```
create schema FireEvent (
sensor string,
smoke boolean,
temperature double
);
```

We can create the following queries

```
@Name('Q0bis')
select *
from TemperatureSensorEvent(temperature > 50) ;
```

```
@Name('Q1')
select sensor, avg(temperature)
from TemperatureSensorEvent
group by sensor;
```

Example hopping window, but the windows start from the first event

```
@Name('Q.agg.snapshot')
select sensor, avg(temperature) as avgTemp
from TemperatureSensorEvent.win:time(10 seconds)
group by sensor
output snapshot every 5 seconds;
```

```
@Name('Q12.every.projection')
insert into FireEvent
select s.sensor as sensor, s.smoke as smoke, t.temperature as temperature
from pattern [
   every (
    s=SmokeSensorEvent(smoke=true)
      -> (
    t=TemperatureSensorEvent(temperature>50,sensor=s.sensor)
    where timer:within(2 seconds)
    )
  )
]
;
@Name('Q13')
select count(*)
from FireEvent.win:time(10 seconds);
```

3.6 Examples 2 and 3

```
create schema A (
n int
);
create schema B (
n int
);
create schema C (
n int
);
```

Every could wrap different parts of pattern and that implies different queries:

```
@Name('Q.11')
select x.n, y.n
from pattern [ every (x=A -> y=B)];
```

The following is another query

```
@Name('Q.12')
select x.n, y.n
from pattern [ (every x=A ) -> y=B];
```

This query would be the same without parenthesis

```
@Name('Q.15')
select x.n, y.n
from pattern [ every x=A -> y=B where timer:within(2 sec)];
```

Let's revise another example

```
create schema StockTick( symbol string, price double );
```

Here we are reporting when IBM's stock prices go from above 100 to below 100 in less than 5 seconds

```
@Name('Q.Lab2.2.every-last-price-above-followed-by-below')
select x.symbol AS stock,
          x.price AS before,
          y.price AS after
from pattern [
every
x = StockTick(symbol='IBM', price > 100)
-> ( y =  StockTick(symbol='IBM', price < 100)
      where timer:within(5 seconds)
)
]
;
```

3.7 Joins

In classical SQL, joins are performed on tables, which are static data structures. Here, queries will join the results only when they find the match (i.e. the second appearance) There are 3 most known types of joins:

3.7.1 Inner Join

Classically, it returns only rows with matches in both tables.
Inner Join requires that at least one window is specified for each stream.
We see the following example

```
select *
from View#time(9 sec) as v
inner join
Click#time(9 sec) as c
on v.id=c.id;
```

We see the results in figure 5

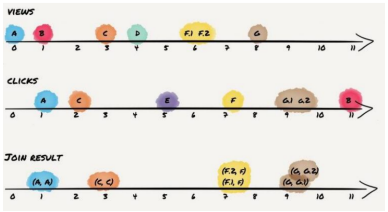

Figure 5: Inner Join Example

3.7.2 Left/Right Outer Join

Classically, it returns all rows from one table with matches found in another table, filling with NULLS for rows with no matches.
Left Outer Join starts a computation each time an event arrives for either the left or right input stream.
For input records of the left stream, an output event is generated every time an event arrives. Then, if an event with the same key has previously arrived in the right stream, it is joined with the one in the primary stream, otherwise, it is set to null.
For the right side, it is only joined if an event with the same key previously arrived in the primary stream.
We see the following example

```
select *
from View#time(9 sec) as v
left outer join
Click#time(9 sec) as c
on v.id=c.id;
```

We see the results in figure 6

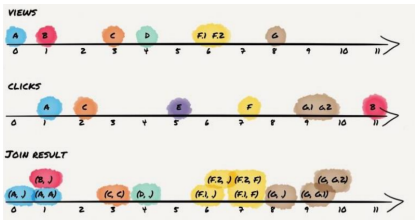

Figure 6: Left Outer Join Example

The results follow the same logic in Right Outer Join

3.7.3 Full Outer Join

Classically, it returns all rows when a match occurs in one of the tables.
An outer join will emit an output each time an event is processed in either stream.
The join method will be applied to both elements if the window state already contains an element with the same key in the other stream. If not, it will only apply to the incoming element.

```
select *
from View#time(9 sec) as v
```

```
full outer join
Click#time(9 sec) as c
on v.id=c.id;
```

We see the results in figure 7

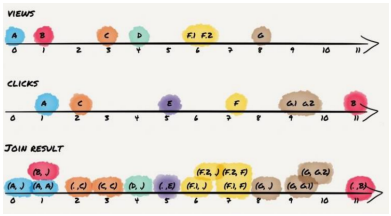

Figure 7: Full Outer Join Example

3.7.4 Table to Table

EPLS offers ways to build tables from unbounded streams (without time windows). It should be done with "create table" statement, with the corresponding "insert into" aggregation query.

- **keep all** window is a data window that retains all arriving events. However, it is very dangerous, care should be taken to remove events from the keep-all window on time
- **unique** window is a window that includes only the most recent among events having the same value(s) for the result of the specified expression or list of expressions

We will only dive into the unique aggregation state. To perform the query, it only has to be replaced

```
from <table>#time(x sec) as <nickname>
```

for

```
from <table>#unique(<column>) as <nickname>
```

Now, we see an inner join example with unique. It is remarkable that it is a non-symmetric operation in terms of the repetition of events

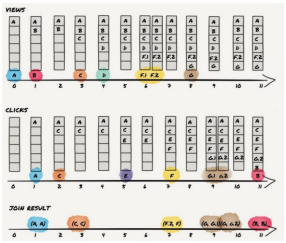

Figure 8: Inner Join Example with unique

Now, we see a left outer join example with unique

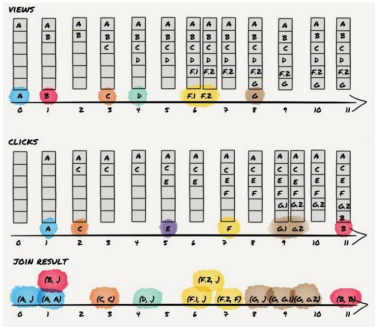

Figure 9: Left Outer Example with unique

Now, we see a full outer join example with unique

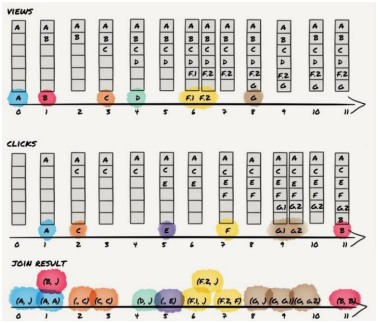

Figure 10: Full Outer Example with unique

3.7.5 Stream to Table

The 'unidirectional' keyword can be used in the from clause to identify streams that provide the events to execute the join.
If the keyword is present for a stream, all other ones in the *from clause* become passive streams.
Therefore, the unidirectional keyword makes the stream-table join asymmetric: Only the left (as an example) stream input triggers a join computation. The rest of the joined elements must be tables (that can be done with commands seen in section 3.7.4)
This semantics is usually used to enrich a data stream (the one with unidirectional) with auxiliary information from other tables.
In the case of inner join:

```
select *
from View as v
unidirectional inner join
Click#unique(id) as c
on v.id=c.id;
```

These are the results:

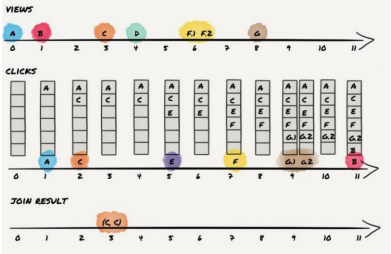

Figure 11: Inner example with unidirectional

This happened because click C is the only click that arrives before the corresponding view event
In the case of the left outer join, these are the results:

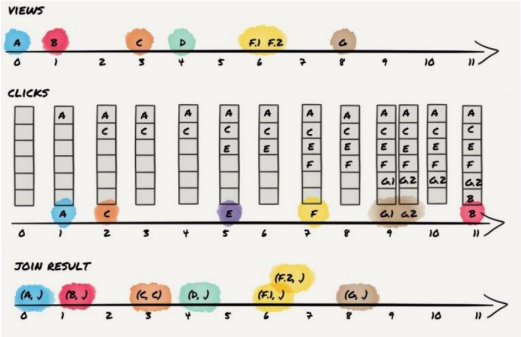

Figure 12: Left Outer example with unidirectional

3.8 Contexts

Context enables more flexible and stateful processing in scenarios requiring grouping or conditions that go beyond simple event windows. There are different types of them
The way to use context is:

```
create context <ContextName>
partition by <feature> from <Table>
```

The types of contexts that exist are:

- **Context by Key:** Partitions events based on a key, e.g., userID as follows:

```
    create context UserSessionContext
    partition by userId from UserActionEvent;
```

- **Context by Start/End Conditions:** Defined by specific events of triggers. Let's revise the following case of use:

```
    create context OrderContext
    partition by userId from UserActionEvent
    initiated by UserActionEvent(action='start_order')
    terminated by UserActionEvent(action='complete_order');
```

- **Context by time:** Defined temporal windows (generalizes logical windows, e.g., it allows to declare session windows). Let's revise the following case of use:

```
    create context TimeBatchContext
    partition by userId from UserActionEvent
    initiated by UserActionEvent
    terminated after 10 minutes;
```

Then, when created, context could be used to perform queries as usual:

```
context UserSessionContext
select avg(spentAmount) from UserActionEvent(action='purchase');
```

3.9 further example

Let be the next model for a robotic arm (with some assumptions that we have taken)

```
create schema RoboticArm( id string, status string, stressLevel int );
```

The following query emits the maximum stress for each arm. Every time something goes up, it sends the maximum

```
@Name("Q2")
SELECT id, max(stressLevel)
FROM RoboticArm
GROUP BY id;
```

The following query finds the average stress level between a pick (status==goodGrasped) and a place (status==placingGood).

```
@Name("E3")
SELECT a.id, (a.stressLevel + b.stressLevel + c.stressLevel) / 3
FROM pattern [
  every   a=RoboticArm(status="goodGrasped") ->
        b=RoboticArm(id = a.id, status="movingGood") ->
        c=RoboticArm(id = a.id, status="placingGood")  ];
```

Note that there's no column where we can do avg directly. With context, it is capable of doing that with avg

It is said that with stress levels between 0 and 6 a robotic arm is operating safely. Between 7 and 8 should raise a warning. Above 9 the robot should stop. The following query returns the robotics arms that, in less than 10 seconds:

- picked a good while safely operating,
- moved it while the controller was raising a warning
- placed it while safely operating again

```
@Name("E4")
insert into warning
SELECT a.id
FROM pattern [
  every a=RoboticArm(status="goodGrasped", stressLevel < 7) ->
      (
        b=RoboticArm(id = a.id, status="movingGood", stressLevel > 6 and
    stressLevel < 9) ->
        c=RoboticArm(id = a.id, status="placingGood", stressLevel < 7)
      )
      where timer:within(10 seconds)
    ] ;
```

Note how the 'and' is used here. Put this 'and' outside, would create two events with a logical 'and', therefore it could lead to some unexpected results

Note that now, we can use warning as an event to perform queries over it

```
@Name("Q5")
select arm, count(*)
from warning.win:time(10 sec)
group by arm
output last every 2 sec;
```

4 Kafka

4.1 Some important concepts

Definition 4.1: Latency

The amount of time needed for an operation to complete

Example.

> A packet takes 20ms to be sent from my computer to Google

Definition 4.2: Throughput

The rate at which operations are performed

Example.

> Memory can provide data to the processor at 25GB/sec

In this sense, we would like to reduce our latency and increase our throughput.

These parameters could change in different ways.
The first way is to fix the size of a message. This impacts on how often we can send messages, because when all the canals are saturated, it is necessary to wait until a message is completely sent to send the next one.
The second way is changing the characteristics of the canal. This could imply that a message could be sent slower or faster.
Compression and binary encoding are the typical ways to increase data per message.

4.2 A conceptual View of Kafka

Kafka is a distributed system. It runs in clusters, where there are actors (clusters) that maintain the system.

4.2.1 Topics

are streams of messages

4.2.2 Clusters

manage topics. Each can handle hundreds of thousands, or millions, of messages per second

4.2.3 Brokers

are the main storage and messaging components of the Kafka cluster. A production Kafka cluster has 3 or more brokers.

4.2.4 Partition

Across all brokers, topics are partitioned. Typically, hash partitioning, based on the message's key, determines the partition a message is assigned to. If the key is null, then round-robin partitioning is adopted.
Each partition is stored on the broker's disk as one or more log files.
Each message on the log is identified by its offset (id inside the partition) number.
Each partition is controlled by a unique thread, to guarantee consistency, since order is only guaranteed inside of a partition

4.2.5 Producers

Send messages on topics (publishers in pub-sub terms). Write them in the log files of the partitions.

4.2.6 Consumers

read messages from topics (subscribers in pub-sub terms). They can subscribe to the topic they are interested in. They can access to log files to find the message of their interest

4.2.7 Messages

is a key-value pair + metadata. The key contains the information to know where the message should be processed. The value is the information we want to send to the consumer

4.2.8 Consumer group

Multiple consumers can be logically combined into consumer groups. It can provide scaling capabilities. Each consumer is assigned to a specific partition for consumption, since each consumer group in Kafka manages one offset for each partition.

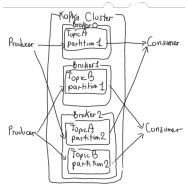

Figure 13: Kafka Diagram

4.3 Kafka Behavior

4.3.1 Log retention

Messages are retained for seven days by default. Anyway, duration is configurable per broker by setting a period or a size limit. The topic configuration can override a broker's retention policy as well.
When a log must be cleaned, the default policy is to delete it. An alternative policy is to compact it.
Compaction consists of retaining at least the last known message value for each key within the partition

4.3.2 Fault Tolerance via a Replicated Log

Kafka maintains replicas of each partition on other brokers in the clusters. The number of replicas is configurable
One broker is the leader. This means that all writes and reads go to and from the leader. Other brokers are called followers. This allows fault tolerance in case a broker goes down.
A broker can contain both, replicas and leader partitions.

4.3.3 Producers and durability

Producers can control durability by requiring the leader to make a number of acknowledgments before considering the request complete.

- **acks=0** The producer will not wait for any acknowledgment from the broker
- **acks=1** The producer will wait until the leader has written the record to its local log
- **acks=all** The producer will wait until all in-sync replicas have acknowledged receipt of the record

5 Spark

5.1 Key ideas of spark

- Use RAM instead of disk, this allows it to run in way less time than competitors
- Ease of use: A high-level language. Simple, it requires few words
- Generality: It could be run in a lot of environments. Plus, it could be used for a lot of tasks

Spark is a unified analytics engine for big data processing, with built-in modules for streaming, SQL, machine learning, and graph processing.
A spark cluster includes one driver running the app and multiple executors (typically one instance per node) with numerous slots (normally, one per core/CPU)

RDD

- Resilient: Fault-tolerant
- Distributed: Computed across multiple nodes
- Dataset: Collection of partitioned data

Characteristics: Immutable once constructed and track lineage (transformations the objects pass for) information.
An object could be logically seen as a unique object, but actually, Spark could work with it in several partitions.
As RDD are immutable, transformations create new RDD's

5.2 Transformations

Transformations are operations that do not perform immediately when defined, because they wait for an action (see section 5.3) to optimize all operations at once.
There are two types of transformations:

- **Narrow Transformations** are where the data required to compute the records in a single partition reside in at most one partition of the parent RDD. E.g. map(), filter()
- **Wide Transformations (Shuffles)** are where the data required to compute the records in a single partition reside in many partitions of the parent RDD. E.g. groupBy(...).sum(), distinct()

5.3 Actions

Actions either return a result or write to disc. Let's see the next example:

```
1 lines = spark.textFile("hdfs://...") # Base RDD
2 errors = lines.filter(lambda s: s.startswith("ERROR")) # Transformed RDD
3 messages = errors.map(lambda s: s.split("\t")[2]) # Transformed RDD
4 messages.cache() # Send messages to cache
5 messages.filter(lambda s: "mysql" in s).count() # Action
6 messages.filter(lambda s: "php" in s).count() # Action
```

The action splits the task to every worker, each one doing it by itself and then the driver incrementally groups the results (see figure 14).

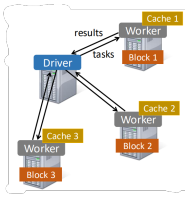

Figure 14: Action task perform example

Plus, if we execute line 4, the next time we perform queries over messages will be much faster than the first one. This is because it has to take out the data from the disc to the cache the first time anyway.

In practice, for example, the system is capable of loading the whole Wikipedia on 20 EC2 machines and doing full-text searches. It takes 0.5 seconds vs 20 seconds with an on-disk approach.

Laziness by design The fundamental notion of Apache Spark is that transformations are lazy, but actions are eager.

5.4 Finding Optimal solutions

Spark uses the next steps to generate optimal RDD code from the user request, through catalyst.

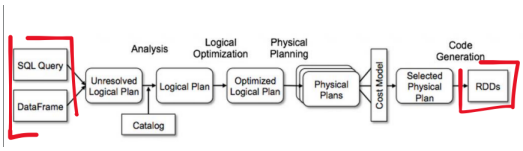

Figure 15: Catalyst way of optimizing user requests

On the other hand, it storages data efficiently through Tungsten

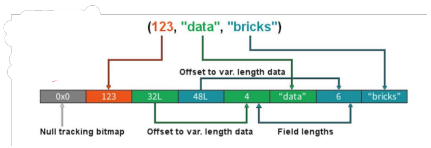

Figure 16: Tungsten way of optimizing in-memory storage row format

5.5 Latencies and guarantees

Spark Structured Streaming provides fast, scalable, fault-tolerant, end-to-end exactly-once stream processing without the user having to reason about streaming
By default, spark processes micro-batches, with end-to-end latency as low as 100 milliseconds, with exactly-once fault-tolerance guarantee (each operation is performed exactly once). Since spark 2.3, it is available end-to-end latencies as low as 1 millisecond, but with an at-least-once guarantee.

5.6 Programming model

Spark treats a live data stream as an unbounded table that is being continuously appended. Therefore, users can logically express streaming computation as a DAG of standard batch-like queries on static tables. Spark physically runs the DAG incrementally.
Spark Structured Streaming does not materialize the entire table. It reads the latest available data from the stream. Then processes it incrementally to update the result. Finally, it discards the source data.
It only keeps around the minimal intermediate state data.

5.7 Streaming Dataframes

Dataframes in Spark are pieces of immutable data with named columns, built on RDDs. Its principal advantages are the ease of use. Spark has a user-friendly API, having uniform APIs across languages like Scala, Java, Python, R, and SQL. Plus, it has an improved performance via optimizations (Tungsten and Catalyst).
We see the following example:

```
# Read a CSV
userDF = spark.read.csv(" ... /userData.csv")
# ... or Use DataFrame APIs and register a temp view
middleageSmokers = userDF.filter(col("smoker")=="Y").filter(col("age")>40)
middleageSmokers.createOrReplaceTempView("middleageSmokers") # Representation
     to be used in sql queries
# ... read a CSV and register a temp view
spark.read.json(" ... /part-00000.json.gz").createOrReplaceTempView("
    iot_stream") # Representation to be used in sql queries
# ... execute SQL query or ...
spark.sql("SELECT avg(calories_burnt) FROM iot_stream JOIN middleageSmokers
    ON ...")
```

Streaming DataFrames can be created through the DataStreamReader interface returned by SparkSession.readStream().
It allows several input sources. Files like CSV, text, JSON, or Parquet. It can use Kafka and socket source as well. This last one, the socket source, has the problem that does not provide end-to-end fault-tolerance guarantee. Let's see the next example over Kafka.

```
raw_sdf = (spark
.readStream
.format("kafka")
.option("kafka.bootstrap.servers", servers)
.option("startingOffsets", "earliest")
.option("subscribe", temperature_topic)
.load())
```

Then, we can use CAST to format it into string:

```
casted_sdf = raw_sdf.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)
    ")
```

Otherwise, to format it into a DataFrame:

```
temp_schema = StructType([
StructField("sensor", StringType(), True),
StructField("temperature", DoubleType(), True),
StructField("ts", TimestampType(), True)])
parsed_sdf = raw_sdf.select(from_json(col("value").cast("string"),temp_schema
    ))
temperature_sdf = parsed_sdf.select("value.*")
```

Then, we can use two different syntaxes to perform queries, both equivalent.

```
filtered_sds = temperature_sdf.select("sensor").where("temperature > 50")
avg_temp = temperature_sdf.groupBy("sensor").avg()
```

or

```
temperature_sdf.createTempView("TemperatureSensorEvent")
query_string = """
```

```
SELECT SENSOR, AVG(temperature)
FROM TemperatureSensorEvent
GROUP BY SENSOR
"""
query_string """
select *
from TemperatureSensorEvent where temperature > 50
"""
filtered_sds = spark.sql(query_string)
```

5.8 Writing resuts

Even streaming queries do not start to run if you do not connect a sink and start writing on it.
To do that, use the DataStreamWriter returned through Dataset.writeStream()
It must be specified:

- **Output Sink:** Where to write.
  - File sink, or a directory
  - Kafka sink. One or more topics in Kafka
  - Memory sink, for debugging
  - Console sink, for debugging
- **Output Mode:** When and what to write
  - **Append Mode (default):** Only the new rows added to the result since the last trigger, only supported for queries whose output rows will never change. Each row is output only once (assuming fault-tolerant sink)
  - **Complete Mode:** It is not recommended, it is very dangerous. The whole result will be outputted after every trigger. Supported only for aggregation queries
  - **Update Mode:** Only the rows in the result that were updated since the last trigger. Supported for aggregation queries and a few others, not for joins.
- **Query Name:** Optionally, a unique name for the query
- **Trigger Interval:** Optionally, the trigger interval
- **Checkpoint Location:** For the end-to-end fault-tolerance

5.9 Window Operations

Spark Structured Streaming supports only logical windows, on event time. It allows for declaring tumbling and hopping windows, not sliding and session ones. Treats windows as particular grouping criteria. We see an example:

```
temperature_sdf.groupBy(window("TS", "1 minutes", "30 seconds"),"SENSOR")
```

5.10 Stream to Stream Join Operations

- **Inner:** Supported, optionally specify watermark* on both sides + time constraints for state cleanup
- **Left Outer:** Conditionally supported, must specify watermark* on right + time constraints for correct results, optionally specify watermark* on left for all state cleanup
- **Right Outer:** Conditionally supported, must specify watermark* on left + time constraints for correct results, optionally specify watermark* on right for all state cleanup
- **Full Outer:** Conditionally supported, must specify watermark* on one side + time constraints for correct results, optionally specify watermark* on the other side for all state cleanup
- **Left Semi:** Conditionally supported, must specify watermark* on right + time constraints for correct results, optionally specify watermark* on left for all state cleanup

**A watermark is part the management of the late-arrived elements (see section 5.12)*

Spark Does not support the EPL's operator "$->$" (or "followed by"). Then, we need to use a stream-to-stream join with temporal constraints on the events timestamps as we see in the next example:

```
join_sdf = (smoke_events.join(
high_temperature_events, expr("""
(sensorTemp == sensorSmoke) AND
(tsTemp > tsSmoke ) AND
(tsTemp < tsSmoke + interval 2 minute )
""")
)))
```

5.11 Stream to Table Join Operations

Inner and Left-outer stream-table joins are supported and not stateful. Full Outer and Right-outer, however, are not supported

```
staticDf = spark.read. ...
streamingDf = spark.readStream. ...
streamingDf.join(staticDf, "type") # inner join
streamingDf.join(staticDf, "type", "left_outer") # left outer join
```

5.12 Late arrival

Late arrivals refer to events that arrive after the expected time window in which they were supposed to be processed. This happens because in real-time streaming, data may experience delays due to temporary disconnections, network latency, processing issues, etc. The event-time captures the time when the data was generated. With the late-threshold T, called watermarking, the system knows how long to wait for late arrivals. If the difference between the processing time and the event time of a later arrival is smaller than T, the event is considered, otherwise, it is discarded for the query. We can see the behavior of this feature with Update Mode in the next example:

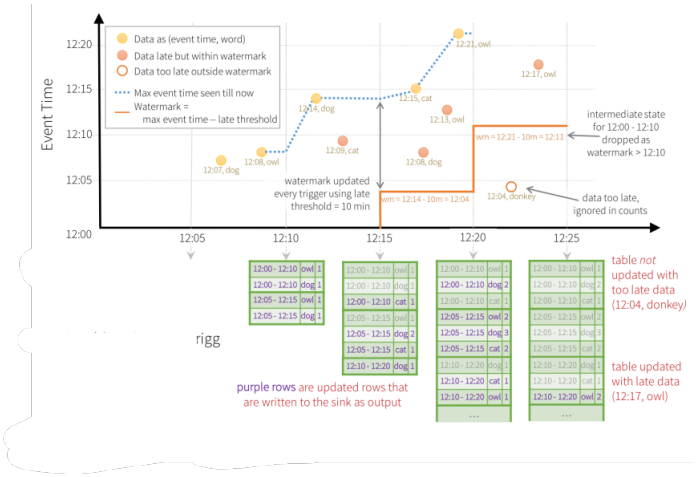

Figure 17: Species counting in a window of 10 seconds, every 5 seconds. Late arrival example with update mode tables downside

5.13 Unsuported Operations

- Aggregation without grouping (e.g., counting rows of a table)
- OrderBy operations unless after aggregation and in Complete Output Mode
- Limit and take the first N rows
- Distinct operations
- Chaining multiple stateful operations in Update and Complete mode
- Few types of outer joins (see previous discussions)

6 Time Series

6.1 Introduction

The previous models do not consider temporal dependence between the events over time. Time series is the prediction tool used to analyze events that are identically distributed but not independent.
Time Series is a sequence of observations on one or more quantitative variables that is regularly collected over time.
The modeling of a time series is based on a combination of different components:

- Trend
- Seasonality
- Residual

The sum of each of these gives the original time series (see figure 18).

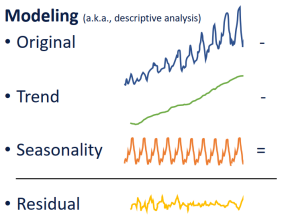

Figure 18: Decomposition of Time Series

6.2 Stationary Time Series

Definition 6.1: Stationary Time Series

A stationary time series is one whose properties do not depend on the time at which the series is observed

A stationary time series:

1. Does not have a trend different than 0: Mean is constant over time
2. Has a variance that is constant over time
3. Has no seasonality: It does not have a deterministic function that predict repetitive pattern

We are interested in stationary time series because they are predictable.
The perfect time series is the white noise, a sequence of random numbers with zero mean and finite variance.
There is also a formal definition of stationarity.

Definition 6.2: Strictly Stationarity

Let $\{X_t\}$ be a stochastic process and let $F_X(t_{1+\tau}, \ldots, t_{k+\tau})$ represent the cumulative distribution function of the unconditional (with any particular starting value) joint distribution of $\{X_t\}$ at times $t_{1+\tau}, \ldots, t_{k+\tau}$.
Then, $\{X_t\}$ is said to be strictly stationary or strongly stationary if

$$F_X(t_1, \ldots, t_k) = F_X(t_{1+\tau}, \ldots, t_{k+\tau}) \ \forall \tau \geq 0 \ \forall k \geq 0$$

6.3 Decomposition and detrend

Just a few real time series are stationary. Normally, they include long-term trends and seasonal patterns.
That's why we could want to remove these characteristics manually. There are two different models commonly used for the decomposition, the additive and multiplicative model.

Additive Model

$$X_t = m_t + s_t + Y_t$$

Multiplicative Model

$$X_t = m_t s_t Y_t$$

where

- m_t is the trend component
- s_t is the seasonal component
- Y_t is the residual component

The multiplicative decomposition is better since the residual does have constant variance. Even that, each one has its case use.
The additive model is useful when the seasonal pattern is more independent of the trend.
The multiplicative model is useful when the seasonal pattern is correlated with the trend.
There also exists the non-seasonal decomposition, that applies the same two models but without s_t
There are methods to decompose the models manually

6.3.1 Non-seasonal decomposition models with trend

There are three basic methods for estimating and therefore removing trend:

1. **Differencing:** Differencing $\{X_t\}$ one or more times removing the trend. It transforms the series in a new one $\{D_t\}$, where:

   $$d_t = x_t - x_{t-1}$$

   If a trend is linear, it is enough to differencing it once. Moreover, if it is quadratic, it needs to be differenced twice. In general, a polynomial of order n could be differentiated n times

2. **Model Fitting:** Fitting a model and then removing it. This consists of fitting a polynomial function (chosen n degree) to the data

3. **'Centered' Moving Averages:** The centered moving averages is calculated depending on whether d is even or odd

   - If d is even:

     $$\hat{m_t} = (0.5x_{t-q} + x_{t-q+1} + \cdots + x_{t+q-1} + 0.5x_{t+q})/d$$

   - If d is odd:

     $$\hat{m_t} = (x_{t-q} + x_{t-q+1} + \cdots + x_{t+q-1} + x_{t+q})/d$$

6.3.2 Decomposition with seasonality

There are three methods for estimating and removing the trend and seasonality components

1. **Differencing:** It starts with differencing as trend decomposition, and then a seasonal differencing is performed, as follows:

With the new time series created, choose a period d and differentiate

$$s_t = x_t - x_{t-d}$$

To apply differentiation to a multiplicative time series, a log function can be applied to 'become' the multiplication into a sum

2. **Filtering:** It starts with a Centered Moving Averages as trend decomposition, and then a 'Periodic Averages' is performed as follows:
   (a) Divide the detrend value into seasons of length d
   (b) Compute the seasonal component values w_k by averaging each of the d points of the season
   (c) Compute adjusted the seasonal component values s_k to ensure that they add to zero
   $$\hat{s_k} = w_k - \frac{1}{d}\sum_{j=1}^{d} w_j, k = 1, \ldots, d$$
   (d) String together the adjusted seasonal component values in a sequence
   (e) Replace the sequence for each season
3. **Joint-fit:** Fitting a combined polynomial and dynamic harmonic regression. It is done as follows:

$$\begin{aligned} X_t &= m_t + s_t + Y_t \\ &= (\beta_0 + \beta_1 t + \beta_2 t^2) + \left[\sum_{j=1}^{k}(\alpha_j \cos(\lambda_j t))\right] + Y_t \end{aligned}$$

6.4 How far we can predict

One step ahead (called short-term) is a forecast for the next observation only. On the other hand, multi-step-ahead forecast is for 2,3,. . . ,n steps ahead
We are able to predict

- **Trend:** Long-term, the easiest to predict
- **Seasonal repetition:** Medium-term
- **Residual:** Short-term, the hardest to predict

6.5 Forecasting heuristical Methods

- **Naive Approach:** The only important value to make predictions is the last one.

$$\hat{y}_{t+1} = y_t$$

- **Average Approach:** All the previous values are equally important to make predictions

$$\hat{y}_{t+1} = \frac{1}{N}\sum y_i$$

- **Last-k average approach:** Only the last k previous values are important to make predictions.

$$\hat{y}_{t+1} = \frac{1}{k}\sum_{i=0}^{k-1} y_{t-i}$$

The problem is how do we choose the k.

- **Exponential Smoothing:** A middle way between naive and average approach, but without setting any k.

$$\hat{y}_{t+1} = \alpha y_t + (1-\alpha)\hat{y}_t \qquad (= \hat{y}_t + \alpha(e_t))$$

Where $0 < \alpha < 1$ is the smoothing level. Closer to 1 forget faster and closer to 0 forget slower.
α is often set as the number of points with a considerable impact (span): $\alpha = \frac{2}{span+1}$.
If we substitute y_i iteratively, we obtain that every data point influences the decision, with more weight in the newest ones:

$$\hat{y}_{t+1} = \alpha y_t + \alpha(1-\alpha)y_{t-1} + \alpha(1-\alpha)^2 y_{t-2} + \cdots + \alpha(1-\alpha)^{t-1} y_1 + (1-\alpha)^t l_0$$

Where l_0 is the first value fitted at time 1.

- **Holt's linear method:** It is an extension to exponential smoothing that adds support for trends using an additional smoothing factor β:
  - Adding when the trend is linear $\quad \hat{y}_{y+h} = l_t + hb_t$
  - Multiplying when the trend is exponential $\quad \hat{y}_{y+h} = l_t \cdot (b_t)^h$

  where

$$l_t = \alpha y_t + (1-\alpha)(l_{t-1} + b_{t-1})$$
$$b_t = \beta(l_t - l_{t-1}) + (1-\beta)b_{t-1}$$

  ℓ_t denotes a level estimation of the time series. On the other hand, b_t denotes a trend estimation of the time series. The trend can vary adaptively over time, and β parameter controls trend-adjusting speed.

- **Holt-Winters method:** It is an extension of Holt's method, that adds support for seasonality using a new parameter γ that controls the influence on the seasonal component. As well as Holt's method, it could be modeled as an additive or multiplicative process.
  - Additive: $\quad \hat{y}_{t+h} = l_t + hb_t + s_{t+h-d}$
  - Multiplicative: $\quad$ Missing

  where

$$l_t = \alpha(y_t - s_{t-d}) + (1-\alpha)(l_{t-1} + b_{t-1})$$
$$b_t = \beta(l_t - l_{t-1}) + (1-\beta)b_{t-1}$$
$$s_t = \gamma(y_t - l_{t-1} - b_{t-1}) + (1-\gamma)s_{t-d}$$

In this model, both trend and seasonality can vary adaptively over time. The smoothing parameters β and γ adjust the speed of trend and seasonality respectively

6.6 Measure accuracy

- Mean Absolute Error (MAE) $MAE = \sum_i \frac{|Y_i - \hat{Y}_i|}{n}$
  Benefits:
  - Easy to calculate
  - Easy to understand and interpret

  Limitations:
  - It does not penalize large errors sufficiently
- Mean Absolute Percent Error (MAPE) $MAPE = \frac{100}{n} \sum_{i=1}^{n} \frac{|Y_i - \hat{Y}_i|}{Y_i}$
  Benefits:
  - Provides a percentage error representation
  - Can be used to make comparisons across different data

  Limitations:
  - Susceptible to division by 0, it shouldn't be used for data close to 0.
  - Biased towards actual values
  - Asymmetric: Negative with positive values can cancel themselves
- Mean Square Error (MSE) $MSE = \sum_{i=1}^{n} \frac{(Y_i - \hat{Y}_i)^2}{n}$
  Benefits:
  - Penalizes larger errors more significantly while rewarding small errors (less than 1)

  Limitations:
  - Units are squared, then units might not be intuitive
  - Sensitive to outliers, not robust against extreme values
- Root Mean Square Error (RMSE) $RMSE = \sqrt{MSE}$
  Benefits:
  - Provides an interpretable scale for MSE like the original data

  Limitations:
  - Sensitive to outliers, not robust against extreme values

Where

- $\hat{Y}_i$ is the predicted value of the data
- Y_i is the actual value in the data

We will almost always prefer RMSE over MSE, since it is an interpretable scale of the Mean Square Error. MSE is used in Signal Processing and other few examples

6.7 Temporal Dependence

6.7.1 Correlation

Correlation describes the degree to which two variables X_1 and X_2 move in coordination with one another. It goes from -1 to 1

6.7.2 Autocorrelation

Autocorrelation at lag k describes the degree of similarity between a given time series and a version of itself lagged k times (shift(k)). It is a correlation between the variable and the variable shifted.
The presence of autocorrelation is an indicator that the time series is more predictable, but it is not conclusive.

6.7.3 Autocorrelation Function Plot (ACF)

It is a graphic that shows the autocorrelation from lag 1 to k^* (see figure 19)

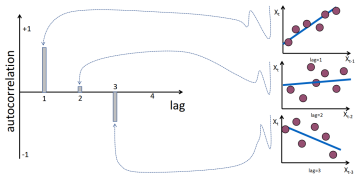

Figure 19: Autocorrelation Function Plot example

6.7.4 Partial Autocorrelation

Partial Autocorrelation at lag k describes the autocorrelation at lag k, but not accounting the dependence of the lags 1 to k-1. Intuitively, it measures a more isolated correlation, taking into account direct correlation only (see figure 20).

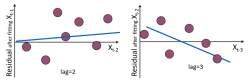

Figure 20: Decomposition of Time Series

6.7.5 Partial Autocorrelation Function Plot (PACF)

It is a graphic that shows the autocorrelation from lag 1 to k^* (see figure 21)

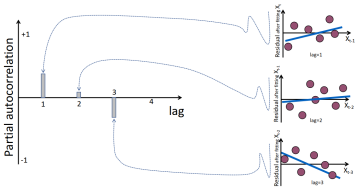

Figure 21: Partial Autocorrelation Function Plot example

6.8 Forecasting Statistical Methods

- **AR(p):** An autoregressive model of order p, denoted as AR(p), models a time series so that the current value of a time series is a linear combination of the past p values of the time series and white noise ϵ_t

$$y_t = c + \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + \cdots + \alpha_p y_{t-p} + \epsilon_t = c + \sum_{n=1}^{p} \alpha_n y_{t-n} + \epsilon_t$$

  It is similar to exponential smoothing, but it uses only p values and the weights α_i, that do not decay exponentially.

  - If $c = \alpha_1 = 0$, AR(1) is equivalent to white noise
  - If $c = 0$ and $\alpha_1 = 1$, AR(1) is equivalent to random walk
  - If $c \neq 0$ and $\alpha_1 = 1$, AR(1) is equivalent to random walk with drift
  - If $c = 0$ and $\alpha_1 < 1$, AR(1) tends to oscillate between positive and negative values

- **MA(q):** A Moving Average model of order q, denotes as MA(q), models the residual of a time series using regression of past q estimation errors

$$y_t = c + \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \cdots + \theta_p \epsilon_{t-q} = c + \sum_{n=1}^{q} \theta_n \epsilon_{t-n} + \epsilon_t$$

- **ARMA(p, q):** An ARMA model of order p, q, is the sum of an Auto Regressive (AR) model of p previous values and a moving average (MA) model of q previous estimation errors

$$\begin{gathered} y_t = c + \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + \cdots + \alpha_p y_{t-p} \,\} AR(p) \\ + \\ \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \cdots + \theta_p \epsilon_{t-q} \,\} MA(q) \end{gathered}$$

  We can identify the ARMA parameters based on the ACF and PACF:

Model	ACF	PACF
ARMA(p,0)	Tails off after some lag	Cut off after p lag
ARMA(0,q)	Cut off after q lags	Tails off after some lag
ARMA(p,q)	Tails off	Tails off

Table 1: Results of applying different parameters in ARMA

  where:

  - Tail off: Progressive decline of the values as lag increase
  - Cut off: Sudden fall in the values as lag increase

- **ARIMA:** An ARIMA model is ARMA + differencing.
  An ARIMA model of order (p, d, q) is the sum of

$$\begin{aligned} y_t^{(d)} = c + \alpha_1 y_{t-1}^{(d)} + \alpha_2 y_{t-2}^{(d)} + \cdots + \alpha_p y_{t-p}^{(d)} \} \, AR(p) \\ + \\ \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \cdots + \theta_p \epsilon_{t-q} \} \, MA(q) \end{aligned}$$

  Where $y^{(d)} = y_t^{(d-1)} - y_{t-1}^{(d-1)}$

- **Seasonal ARIMA (SARIMA):** A SARIMA model is an ARIMA model with an additional set of autoregressive and moving average components (the last two sum) for the seasonality

$$y_t = c + \sum_{n=1}^{p} \alpha_n y_{t-n} + \sum_{n=1}^{q} \theta_n \epsilon_{t-n} + \sum_{n=1}^{P} \phi_n y_{t-sn} + \sum_{n=1}^{Q} \eta_n \epsilon_{t-sn} + \epsilon_t$$

  where s is a period length.

- **SARIMAX:** A SARIMAX model is an SARIMA model but incorporating r exogenous variables. These are external factors that can impact the time series but are not influenced by it.

$$y_t = c + \sum_{n=1}^{p} \alpha_n y_{t-n} + \sum_{n=1}^{q} \theta_n \epsilon_{t-n} + \sum_{n=1}^{r} \beta_n x_{n_t} + \sum_{n=1}^{P} \phi_n y_{t-sn} + \sum_{n=1}^{Q} \eta_n \epsilon_{t-sn} + \epsilon_t$$

Note: These models should be applied with stationary time series only

It is possible to write any stationary AR(p) model as an MA(∞) model. For example, an AR(1) model can be written as:

$$\begin{aligned} y_t &= \alpha_1 y_{t-1} + \epsilon_t = \alpha_1(\alpha_1 y_{t-2} + \epsilon_{t-1} + \epsilon_t) \\ &= \alpha_1^2 y_{t-2} + \alpha_1 \epsilon_{t-1} + \epsilon_t) \\ &= \alpha_1^3 y_{t-3} + \alpha_1^2 \epsilon_{t-2} + \alpha_1 \epsilon_{t-1} + \epsilon_t) \\ &\dots \end{aligned}$$

6.9 Measure indicators

There are indicators commonly used to measure statistical models. They measure the quality of the model in terms of its goodness-of-fit to the data, its simplicity, and how much it relies on the tuning parameters.

- **Akaike Information Criteria (AIC):**

$$AIC = 2k - 2l$$

- **Bayesian Information Criteria (BIC):**

$$BIC = k \log n - 2l$$

where

- l is the log-likelihood
- k is the number of parameters
- n is the number of samples used for fitting

There are others as well like log-likelihood and HQIC.

There are also a lot of types of residuals. The two most used are Ljung-Box and Jarque-Bera.

The box-Jenkins methodology consists of starting with a model that uses minimum p and q. Then it diagnoses the residuals, verifying they have a normal distribution. Also, it checks out the ACF and PACF of the residuals to see if further temporal dependencies are present. Finally, it decides if increase p and/or q if there are large autocorrelations or partial autocorrelations

There are also tests for diagnostic models:

Test	**Good fit**
Standardized residual	There are no obvious patterns in the residuals
Histogram Plus kde estimate	The KDE curve should be very similar to the normal distribution
Normal Q-Q	Most of the data points should lie on the straight line
Correlogram	95% of the correlations for lag greater than zero should not be significant

Table 2: Tests for diagnostic statistical models

6.10 Multiple parameters

Both trend and seasonality can change over time multiple times throughout the dataset updates. To cope with this problem a viable option is forecasting each component separately and then combining forecasts. The split is done with MSTL (Multiple Seasonal-Trend) decomposition.
There also exist some more advanced methods:

- VARIMA - Vector AutoRegressive Integrated Moving Average
- ARCH - AutoRegressive Conditionally Heteroscedastic
- GARCH - Generalized AutoRegressive Conditionally Heteroscedatic
- BATS - Box-Cox transformation, ARMA errors, Trend and Seasonal components
- TBATS - Trigonometric seasonality, Box-Cox transformation, ARMA errors, Trend, and Seasonal components

6.11 Prophet

Prophet is a forecasting tool available in Python and R, open-sourced by Meta. It essentially frames the forecasting problem as a curve-fitting exercise rather than looking explicitly at each observation's time-based dependence.
Prophet is an additive regression model with 3 components:

$$y(t) = g(t) + s(t) + h(t) + \varepsilon_t$$

- $g(t)$**:** A piecewise linear or logistic growth curve trend, modeling non-periodic changes. It automatically detects changes in trends by selecting changepoints $a(t)^T$ from the data. These changepoints could also be set manually. The functions are as follows:
  Linear:
  $$g(t) = (k + a(t)^T\delta)t + (m + a(t)^T\gamma)$$
  Logistic: It allows us to set a maximum and a minimum c in the prediction
  $$g(t) = \frac{C(t)}{1 + e^{-(k+a(t)^T\delta)(t-m+a(t)^T\gamma)}}$$

- $s(t)$**:** A sub-daily, daily, weekly, and yearly seasonal component modeled using Fourier Series. i.e. it implements 3 different types of seasonalities
  $$s(t) = \sum_{n=1}^{N}(a_n \cos(\frac{2\pi nt}{P}) + b_n \sin(\frac{2\pi nt}{P}))$$

- $h(t)$**:** A user-provided list of important holidays, or exogenous variables. It adds peaks and drops that can be explained with a prior $\kappa \sim Normal(0, \sigma^2)$
  $$h(t) = Z(t)\kappa$$
  $$Z(t) = [\mathbb{I}(t \in D_1), \mathbb{I}(t \in D_2), \ldots, \mathbb{I}(t \in D_L)]$$

- ε_t**:** Noise that cannot be captured by the model

- **Extra:** There is also possible to add additional regressors, that can add exogenous variables that are not a binary indicators necessarily

6.12 Deep Learning for Time Series Forecasting

Most traditional methods are designed to forecast individual series or small groups. Anyway, some new problems have emerged:

- Forecasting many individual or grouped time series
- Learning a more general model that faces the scale of different time series
- Accounting for exogenous or multivariate inputs (although some traditional methods included this)
- Adding new items to the forecast

Therefore, the novel models needed to learn to generalize from similar series. Plus, estimating the probability distribution of a time series based on its past, this is called Probabilistic Forecasting.
The cons of using DL models are the bigger amount of computational resources needed.

6.12.1 DeepAR

One of the first DL models for this purpose was DeepAR. It is a forecasting model based on Autoregressive Recurrent Neural Networks or RNNs (see (Jeon, 2024)), which learns a global model from historical data of all time series in loads of datasets.

The main advantages of DeepAR are:

- Multiple time-series support: It supports multiple types of time series
- Extra covariates: It allows to add extra features (Covariates)
- Probabilistic output: It leverages Monte Carlo samples to output prediction intervals
- Cold forecasting: As it is trained with thousands of time series, it could succeed in forecasting time series with little or no history at all.
- No extra preprocessing: It does not require making the time series stationary first

DeepAR outperforms traditional statistical methods such as ARIMA.

Plus, it is prevalent in production, since it is part of Amazon's GluonTS toolkit for time series forecasting, an open-source Python library. Further, it can be trained on Amazon SageMaker a fully managed machine learning service provided by AWS.

6.12.2 Architecture

DeepAR uses LSTM networks to create probabilistic outputs, but differently to previous models.

Instead of using LSTMs to calculate predictions directly, DeepAR leverages LSTMs to predict all parameters of the probability distribution for the next time point. For instance, for a Gaussian Distribution, it would estimate μ and σ^2.

In each step of LSTM, DeepAR tries to find the parameters θ that predict as close as possible to the target variable. Using the recurrent connections, it uses z_i^{t-1} and x_i^t to find the parameters for z_i^t at each step.

After the LSTM layers, which are at the beginning of the network. DeepAR uses 2 dense layers to derive single μ and σ global parameters.

6.12.3 Newer DL models

Some of the newer Deep Learning Models are D-linear, N-linear, N-BEATS, N-HiTS, Temporal Convolutional Network, Temporal Fusion Transformer, TiDE - Time Series Dense Encoder, TSMixer, Space Time Former, PatchTST, TimesNet.

A newer approach is foundational modeling, which tries to create one model for absolutely every kind of Time Series. These require even much bigger resources. Examples of these models are TimeGPT and Lag-Llama.

7 Time Series Implementation

7.1 Stationarity

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Create white noise

```
# create series
mean = 0
std = 1
num_samples = 1000
samples = np.random.normal(mean, std, size=num_samples)
ts = range(0,num_samples)

# build dataframe
df = pd.DataFrame(columns = ['ts', 'values'])
df['values'] = pd.Series(samples)
df['ts'] = pd.Series(ts)
```

Summary statistics over time

```
X = data_set.values.flatten()
split = round(len(X) / 2)
X1, X2 = X[0:split], X[split:]
mean1, mean2 = X1.mean(), X2.mean()
var1, var2 = X1.var(), X2.var()
# Comparing histograms
pd.DataFrame(X1).hist()
pd.DataFrame(X2).hist()
```

A log function can be applied to the datasets so an exponential growing could be removed. The same could be done with any function

7.2 Statistic Tests

There are some already built statistical tests to test stationarity
The ADF test tries to reject the Null Hypothesis (H_0): The time series is not stationary. We try to evaluate this using a p-value with a threshold of 5% or 1%. This means that if p-value$\leq$ threshold, the hypothesis null is rejected

```
result = adfuller(X)
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
print('Critical Values:')
for key, value in result[4].items():
    print('\t%s: %.3f' % (key, value))
```

There exists the Kwiatkowski–Phillips–Schmidt–Shin (KPSS) test as well. On the contrary, its null hypothesis is that the data is stationary

```
result = kpss(X)
print('KPSS Statistic: %f' % result[0])
print('p-value: %f' % result[1])
print('Critical Values:')
for key, value in result[3].items():
```

```
    print('\t%s: %.3f' % (key, value))
```

We can compare both tests, that usually give the same results. If the two tests contradict each other, some form of differentiation or de-trending is likely needed

- If ADF finds stationary but KPSS doesn't. Probably the series is "difference-stationary": It still requires differentiation (see section 7.4).
- If KPSS finds stationary but ADF doesn't. Probably the series is "trend-stationary": It still requires differentiation or other de-trending transformations to remove the trend (see section 7.5).

7.3 Decomposition

Import the libraries

```
import pandas as pd
import matplotlib.pyplot as plt
# Decomposition
import statsmodels.api as sm
from statsmodels.tsa.strattols import adfuller
```

Additive decomposition

```
# decompose with additive model (seasonal, trend and resid)
add_decomposition = sm.tsa.seasonal_decompose(data_set, model = 'additive')
x = add_decomposition.plot()
```

Multiplicative decomposition

```
mul_decomposition = sm.tsa.seasonal_decompose(data_set, model = '
    multiplicative')
x = mul_decomposition.plot()
```

The elements of the decomposition could be extracted as follows

```
mul_decomposition.trend
mul_decomposition.seasonal
add_decomposition.resid
# They could be subscripted by [ ] notation, just using as index of the data
```

Then ADF test could be applied over the residual, then we can see if the stationarity was achieved after separate trend and seasonality from the original serie

```
df = add_decomposition.resid.dropna()

result = adfuller(df, autolag='AIC')
```

7.4 Detrend Non-Seasonal Data

In **differencing**, we can apply the following methods n times in order to detrend the data

```
df_diff = (df - df.shift(1))
# Or
df_diff = df.diff()
```

In **model fitting**, we can apply different functions to each polynomial degree desired. In the case of a linear function,

```
model = LinearRegression()
model.fit(X, y)
model.coef_, model.intercept_
trend = model.predict(X)
detrended = [y[i]-trend[i] for i in range(df.shape[0])]
```

In the case of a polynomial degree of n

```
model = np.poly1d(np.polyfit(X.flatten(), y, 2))
line = np.linspace(0, 168, len(X.flatten()))
detrended = [y[i]-model(line)[i] for i in range(df2.shape[0])]
```

7.5 Removing seasonality

In **trending differencing**, we can apply the following method

```
def difference(dataset, interval=1):
    diff = list()
    for i in range(interval, len(dataset)):
        value = dataset[i] - dataset[i - interval]
        diff.append(value)
    return diff
diff = difference(data, d)
######### or
residuals = data.diff(d)
```

In **Filtering**, we can apply the following method

```
data_set['centred-MA'] = data_set['original'].rolling(window=d, center=True).
    mean()
data_set['MA'] = data_set['original'].rolling(window=12, center=False).mean()
######### additive
data_set['add_detrended'] = data_set['original']-data_set['centred-MA']
######## multiplicative
data_set['mul_detrended'] = data_set['original']/data_set['centred-MA']
########

subsequence = data_set[["add_detrended","moy"]].groupby(["moy"]).mean()
subsequence.rename(columns={"add_detrended": "add_season"},inplace=True)
mean = subsequence.mean()

######## additive
subsequence["add_season"] = subsequence["add_season"]-float(mean)
####### multiplicative
subsequence["mul_season"] = subsequence["mul_season"]/float(mean)

#########
seasonal_component = subsequence
for i in range(0,11):
    seasonal_component = pd.concat([seasonal_component,subsequence])
seasonal_component = seasonal_component.reset_index()

###### additive
data_set['add_deseasoned'] = data_set['original'].values - seasonal_component
    ["add_season"].values
data_set['add_deseasoned & add_detrended'] = data_set['add_detrended'].values
     - seasonal_component["add_season"].values
####### multiplicative
```

```
data_set['mul_deseasoned'] = data_set['Thousands of Passengers'].values /
    seasonal_component["mul_season"].values
data_set['mul_deseasoned & mul_detrended'] = data_set['mul_detrended'].values
     / seasonal_component["mul_season"].values
```

7.6 Forecasting methods

- **Naive**

```
def naive(data):
    return data[-1]
```

- **Mean**

```
def simple_mean(data):
    return data.mean()[language=Python]
```

- **Last-k mean**

```
def lastk_mean(data, k):
    return data[-k:].mean()
```

- **Exponential Smoothing**

```
from statsmodels.tsa.api import SimpleExpSmoothing

train.index = pd.DatetimeIndex(train.index.values, freq=None)
fit02 = SimpleExpSmoothing(train, initialization_method="heuristic").fit(
    smoothing_level=alpha, optimized=False
)
SES02_predictions = fit02.forecast(len(test))
```

- **Holt**

```
from statsmodels.tsa.api import Holt
fitHolt = Holt(train['train']).fit(smoothing_level = alpha,
    smoothing_trend = beta)
test['Holt'] = fitHolt.forecast(len(test))
```

- **Holt-winters**

```
from statsmodels.tsa.api import ExponentialSmoothing

fitTES = ExponentialSmoothing(train['train'],
                              seasonal_periods=period_seasonality,
                              trend='add',
                              seasonal='add',
                              use_boxcox=True,
                              initialization_method="estimated").fit()

test['Holt-Winters'] = fitTES.forecast(len(test))
```

We can access the parameters of the models as follows

```
model.params
```

7.7 Forecasting metrics

MAE

```
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import median_absolute_error # Obtain the median
```

MSE

```
from sklearn.metrics import mean_squared_error
mse = mean_squared_error(y_true, y_pred)
from sklearn.metrics import mean_squared_log_error # Obtain the log of y_i+1
```

MAPE

```
import numpy as np
def mean_absolute_percentage_error(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
mean_absolute_percentage_error(y_true, y_pred)
```

7.8 Generate synthetic data

```
num_samples =  512

####### parameter of the AR(1) process
alpha_1 = 0.5
ar1 = np.r_[1, -np.array([alpha_1])] # add zero-lag and negate
ma1 = np.r_[1] # add zero-lag

####### parameter of the MA(1) process
beta_1 = 0.5
ar2 = np.r_[1] # add zero-lag and negate
ma2 = np.r_[1, np.array([beta_1])] # add zero-lag

AR1 = pd.DataFrame({'timestamp' : pd.date_range('2021-01-01', periods=
    num_samples, freq='MS'),
                        't' : sm.tsa.arima_process.arma_generate_sample(ar1,
    ma1, num_samples)
                       })

MA1 = pd.DataFrame({'timestamp' : pd.date_range('2021-01-01', periods=
    num_samples, freq='MS'),
                        't' : sm.tsa.arima_process.arma_generate_sample(ar2,
    ma2, num_samples)
                       })
```

7.9 Plotting ACF and PACF graphics

```
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf

f, ax = plt.subplots(nrows=2, ncols=1, figsize=(width, 2*height))
plot_acf(sample['t'],lags=lag_acf, ax=ax[0])
plot_pacf(sample['t'],lags=lag_pacf, ax=ax[1], method='ols')

```

```
plt.tight_layout()
plt.show()
```

7.10 Result's analysis

```
order = (1,0,0) # AR(1)
order = (0,0,1) # MA(1)
order_s = (2,1,1) # SARIMA(2,1,1)x seasonal_order

train_len = int(0.8* num_samples)
train = sample['t'][:train_len]
model = SARIMAX(train, order=order).fit()
print(model.summary())

### SARIMA and SARIMAX (We add the additional data in the prediction stage)
model_s = sm.tsa.statespace.SARIMAX(train,order = order_s, seasonal_order =
    (1,1,0,12)) # order_s x (1,1,0,12)
results = model.fit()
```

See the next result's example

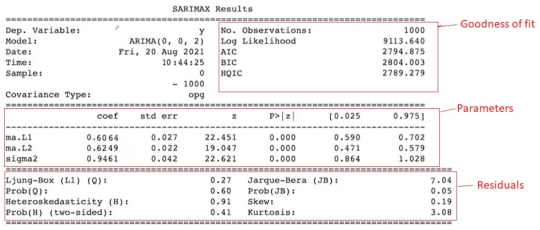

Figure 22: ARIMA Result's example

7.11 Predict Graphic

```
pred = model.predict(start=train_len, end=num_samples, dynamic=False)

#######
pred = model.predict(start=train_len, end=num_samples, dynamic=False, exog=
    test[['extradata']])
########

f, ax = plt.subplots(nrows=1, ncols=1, figsize=(12, 4))
sns.lineplot(x=sample.timestamp[train_len:num_samples], y=sample.t[train_len:
    num_samples], marker='o', label='test', color='grey')
sns.lineplot(x=sample.timestamp[:train_len], y=train, marker='o', label='
    train')
sns.lineplot(x=sample.timestamp[train_len:num_samples], y=pred, marker='o',
    label='pred')
ax.set_xlim([sample.timestamp.iloc[0], sample.timestamp.iloc[-1]])
ax.set_title('Sample Time Series')
plt.tight_layout()
plt.show()
```

See the next graph's prediction example

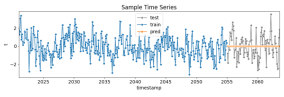

Figure 23: Prediction AR(1) example

7.12 Diagnostics

```
model.plot_diagnostics(figsize=(15,8))
plt.tight_layout()
plt.show()
```

See the next diagnostic example

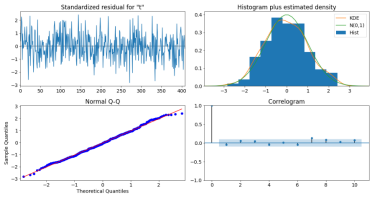

Figure 24: Diagnostic model for AR(1) example

7.13 Finding best p and q

```
order_aic_bic =[]
# Loop over AR order
for p in range(m):
    # Loop over MA order
    for q in range(n):
        # Fit model
        results = SARIMAX(train, order=(p,0,q)).fit()
        # print the model order and the AIC/BIC values
        print(p, q, results.aic, results.bic)
        # Add order and scores to list
        order_aic_bic.append((p, q, results.aic, results.bic))
order_df = pd.DataFrame(order_aic_bic, columns=['p','q', 'aic', 'bic'])

print(order_df.sort_values('aic')) # or 'bic'
```

This can be done automatically with

```
print(order_df.sort_values('aic'))
```

7.14 Prophet

```
from prophet import Prophet

# Initialize the Prophet model instance.
m = Prophet(weekly_seasonality=False, daily_seasonality=False)
# Fit the model on the time series.
m.fit(time_series_train)

# Forecast
```

```
forecast = m.predict(future)
# Plot the forecasts. Uncertainity intervals of width 80% are show by default as the shaded blue region.
fig1 = m.plot(forecast)


######### With holidays
# lower_window = -1 specifies that we should expect the week before the holiday to experience an increase in the data.
# if lower_window = 0, then we would expect a spike in only the week that contains the holiday.

holidays = pd.concat((hol1, hol2))
m = Prophet(holidays=holidays, weekly_seasonality=False,  daily_seasonality=False)
m.fit(avocado_prophet);
future = m.make_future_dataframe(periods=52, freq = 'W-Sun')
forecast = m.predict(future)
fig1 = m.plot(forecast)
fig2 = m.plot_components(forecast)
```

8 Stream Machine Learning (SML)

8.1 Introduction

SML deals with data that is non-stationary, unbounded, and available once at a time. The models cope with the time and memory problem by updating the model incrementally with the just-arrived sample and then discarding it

8.2 Prequential evaluation

To estimate prequential error (PE), the error calculated by looking at each data point only once, we can use:

- Sliding window of size w
$$PE_i = \frac{1}{w} \sum_{k=i-w+1}^{w} e_k$$
- Fading factor
$$PE_i = \frac{\sum_{k=1}^{i} a^{i-k} \cdot e_k}{\sum_{k=1}^{i} a^{i-k}}, \qquad \text{with } 0 \ll \alpha \leq 1$$

Cross Validation cannot be used as usual, there are adapted methods:

- **k-fold distributed cross-validation:** Each sample is used for testing in one classifier selected randomly, and used for training and testing all the others
- **k-fold distributed bootstrap-validation:** Each sample is used for training in one classifier selected randomly, and for testing in all the classifiers
- **k-fold distributed cross-validation:** Each sample is used for training in approximately 2/3 of the classifiers, with a separate weight in each classifier, and for testing in all the classifiers

8.3 Evaluation Metrics

The kappa statistic is the most used evaluation metric for SML. It is calculated as follows:

$$k = \frac{p - p_{rand}}{1 - p_{rand}}$$

Where p is the accuracy of the classifier and p_{rand} is the accuracy of the random classifier *We can also use p_{per}, that is the accuracy of the persistent classifier, the classifier that always predicts the same label (often the more present in the dataset)*

8.4 Concept Drift

Concept Drift consists of a change in the data distribution over time, breaking the other-models-suppositions of identically distributed data, and therefore worsening the model.
This kind of situation occurs. Actually, it was what happened during the pandemic, that broke lots of non-online recommendation models.
Formally, given an input sequence $X_1, X_2, \ldots, X_t$, we want to output at the instant an alarm signal if there is a distribution change and give a prediction $\hat{X}_{t+1}$ minimizing the prediction error:

$$|\hat{X}_{t+1} - X_{t+1}|$$

Given an input sequence $X_1, X_2, \ldots, X_t$, we want to classify X_t. A way to do it is by knowing the prior probability of observing each class, p(y), and the conditional probability of observing X_t given each class, $p(X_t|y)$. Then, using the Bayes's theorem:

$$p(y|X_t) = \frac{p(y) \cdot p(X_t|y)}{p(X_t)}$$

It is possible to compute the probability that X_t is an instance of class y, where $p(X_t)$ is the probability of observing X_t. Since the latter is constant for all the classes y, it can be ignored.

- **Virtual/Data drift:** There are cases in which only the input distribution changes, without the decision boundary, in other words. $p(y)$ and $p(X_t)$ can change. Therefore it only changes the input distribution, updating the model is unnecessary.
- **Real/Concept drift:** On the contrary, there are cases in which input distribution changes. If $p(y|X_t)$ changes, the model could change.

There are also types of concept drifts based on how fast the distribution changes

- Abrupt/Sudden Drift: In a specific time, the distribution changes
- Gradual Drift: The distribution starts to alternate between the old and new one
- Incremental Drift: The number of data coming from the old distribution starts to decrease progressively
- Recurring Drift: Each certain time, the distribution changes

At last, there are some characteristics in the concept drifts:

- **Feature Drift:** It occurs when a subset of features are not relevant anymore to learn the concept. Consequently, if the model strongly relies on that subset to predict, the prediction could be wrong.
- **Feature Evolution:** It occurs when new features appear or disappear over time or the set of all possible values of a feature changes. If a new feature becomes available and deemed relevant, the model should be able to incorporate it into its learning process. Similarly, if a feature becomes unavailable, the learning model should ignore its existence
- **Concept Evolution:** It occurs when new class labels appear/disappear. This is natural in some domains, such as intrusion detection systems, where new threats continuously appear as attackers always introduce new strategies

There is a trade-off between taking a new data point as a concept drift or an anomalous point.

8.5 Concept Drift Detectors

There are ways to identify concept drifts.

8.5.1 Monitoring input distribution

We can monitor the input distribution. It does not require supervised samples, but it is difficult to design sequential detection tools for this purpose, and it does not detect changes that do not change the input distribution. Some models that implement this are:

- **Cumulative SUM Test (CUSUM):** It gives an alarm when the mean of the input data is significantly different from 0.

Algorithm 2 CUSUM

$g_0 = 0$
$g_t = max(0, g_{t-1} + (x_t - \hat{x}) - v)$
if $g_t > h$ then Alarm

It is memoryless, and its accuracy depends on the choice of v and h. Plus, it is a one-sided test that only detects increases in the data but not decreases.

- **Page Hinkley Test:** It is designed to detect a change in the average of a Gaussian signal and monitor the difference between g_t and G_t.

Algorithm 3 Page Hinkley Test

$g_0 = 0$
$g_t = g_{t-1} + (x_t - \hat{x}) - v$
if signal increasing **then**
    $G_t = min(g_t, G_{t-1})$
    if $g_t - G_t > h$ then Alarm
else
    $G_t = max(g_t, G_{t-1})$
    if $G_t - g_t > h$ then Alarm
end if

Its accuracy depends on the choice of v and h as well.

Where $\hat{x}$ is the reference value. Usually chosen as the media or some predetermined known figure.

8.5.2 Monitoring Classification Error

We can monitor classification errors. It is the most straightforward figure of merit to monitor. Plus, changes in p_t (classification error mean) prompt adaption only when performance is affected. The con is that it can detect drift only with supervised samples. These models can have warning and drift levels, with a window between those two levels, where the model compares the performances obtained with diverse options and arrives at the drift level if a new option outperforms the old one
Some models that implement this are:

- **Drift Detection Method (DDM):** It detects concept drift as an outlier in the classification error. In stationary conditions, error decreases, so look for outliers in the tails.

Algorithm 4 Drift Detection Method

Compute the classification error mean p_t and $\sigma_t = \sqrt{\frac{p_t(1-p_t)}{t}}$
Let p_{min} and σ_{min} the minimum p_t and σ_t values seen until now
If $p_t + \sigma_t > p_{min} + 2 \cdot \sigma_{min}$ then Warning
If $p_t + \sigma_t > p_{min} + 3 \cdot \sigma_{min}$ then Change

- **Early Drift Detection Method (EDDM):** It considers the distance between two consecutive error classifications instead of considering only each error rate. While the learning method is learning, it will improve the predictions and the distance between two errors will increase. When a drift occurs, the distance between those two errors will decrease. Compute the average distance between 2 errors and its std, and look for outliers in the tails

- **Adaptative Sliding WINdow (ADWIN):** An adaptive sliding window whose size is recomputed online according to the rate of change observed. It does not need parameters.
  From a window W of a fixed minimum size, it splits it in windows W_0 and W_1 with

also fixed minimum sizes. It will try to expand the first window changing the split (keeping the minimum size) until it finds a significant difference between the means of both windows:

$$|\hat{\mu}_{W_0} - \hat{\mu}_{W_1}| \geq \epsilon_c$$

When a significant difference is found, the W_0 window, in other words, the older data, is discarded, becoming W_1 in the new W to train the model

8.6 SML Classification Models

- **Naive Bayes:** It is based on bayes theorem, where c is the class and d is the instance to classify.

  $$P(c|d) = \frac{P(c) \cdot P(d|c)}{P(d)}$$

  Estimate the probability of observing attribute a and the prior probability P(c):

  $$P(c|d) = \frac{P(c) \cdot \prod_{a \in d} P(a|c)}{P(d)}$$

  Finally, the prediction will be the statistic that maximizes $\log P(c|d)$. The value and its calculation are well-known in some distributions, such as the Gaussian one.

- **Online K-Nearest Neighbours (KNN):** The most common label of the k instances closer to a new instance determines the label. The distance between instances is calculated (commonly) using the Euclidean distance.

  To save the instances, a fixed sliding window size is used, but ADWIN can be used to automatically set the size of the sliding window.

- **Hoeffding Trees (VFDT):** It creates an online decision tree that grows incrementally. The final tree must be identical to a tree built using a batch decision tree algorithm. There exist statistical theoretical guarantees on the error rate.
  Let X_1 and X_2 be the two most informative attributes, i.e., the two best split candidates based on the current data. This is done according to the function G
  Split if

  $$G(X_1) - G(X_2) > \epsilon = \sqrt{\frac{R^2 \cdot \log(1/\delta)}{2N}}$$

  Where

  - δ is the confidence bound
  - N is the number of instances seen by that node
  - G is the splitting criteria. 'Information Gain', 'Gini Index', or 'Variance Reduction can be used'
  - R is the range of values that G can take: $\max(G(X)) - \min(G(X))$

- **Hoeffding Adaptive Tree (HAT):** Replace Frequency statistics counters with estimators based on the new data. It does not need a window to store examples, but it maintains the statistical data with estimators. It changes the way of checking the substitution of alternate subtrees using a change detector with theoretical guarantees

(ADWIN). The main advantages are the theoretical guarantees and that it has no parameters

8.7 CASH Problem and AutoML

CASH Problem: Combined Algorithm Selection and Hyperparameter.
AutoML aims to automate the data mining pipeline:

- Data cleaning
- Feature engineering
- Algorithm selection
- Hyperparameters tuning

There are different implementations with diverse search spaces and hyperparameter optimization:

- Auto Weka 2.0
- Autosklearn
- TPOT
- GAMA
- H2O

The problem is that these algorithms don't consider the adaptation of parameters in an evolving data stream. The models could be trained and retrained from scratch after a concept drift, but this is computationally expensive.

EvoAutoML is a State-of-the-art model that naturally adapts the population of algorithms and configuration. It addresses the online CASH problem by finding the joint algorithm combination (with bootstrapping) and hyperparameters setting that minimizes a predefined loss over a stream of data. It makes predictions by majority voting.

8.8 Bias-Variance trade off

When a model is less complex, it ignores relevant information, and error due to bias is high. As the model becomes more complex, errors due to bias decrease. On the other hand, when a model is less complex, error due to variance is lower. Errors due to variance increase as complexity increases, i.e. overfitting.
Overall model error is a function error of bias and variance. The ideal model minimizes errors from each of them

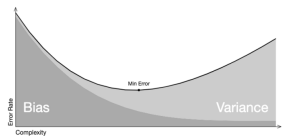

Figure 25: Bias variance trade-off and optimal point

8.9 Ensemble Classifiers

A weak learner is loosely defined as a learner that performs slightly better than random guessing. An ensemble is a composition of multiple weak learners to form one with expected higher predictive performance (strong learner). These models have a high predictive performance and flexibility, but use a lot of resources.

8.9.1 Bagging

The bootstrapping technique consists of creating a subdataset by random samples with replacement from a binomial distribution in the original dataset. This ensemble technique fits M independent models and combine their predictions in order to obtain a model with a lower variance. This technique creates M bootstrap samples (one for each model) from the original dataset of size N (in step N).

- **Random Forest:** It uses M binary trees, fitted on bootstrap samples, that are combined to produce an output with lower variance. To make the M trees less correlated, random forest also samples over features and keeps only a random subset of them to build each tree

Problem: How do we state independence if we use the same dataset for every model?

8.9.2 Boosting

Sequential method that combines weak models no longer independent from each others.
It fits the model in the current step based on the models fitted at the previous steps. It produces an ensemble model that is in general less biased than the weak learners that compose it.

- **Adaptive Boosting (AdaBoost):** It puts more weight on difficult instances and less on those already well-handled.
  It consists of updating the observation weights in the dataset and training a new weak learner with a special focus given to the observation misclassified by the current ensemble model.
  Second, it adds the weak learner to the weighted sum according to an update coefficient that expresses the performance of the model. The better a weak learner performs, the more it contributes to the strong learner

- **Gradient boosting:** Instead of fitting a weak learner on the data at each iteration, it fits a new weak learner to the residual errors made by the previous one.

8.9.3 Stacking

It considers heterogeneous weak learners (with different learning algorithms). It learns to combine the base models using a meta-model. It produces an ensemble model that is in general less biased than the weak learners that compose it

8.10 Diversity of Ensembles

The ensembles have to be diverse among learners, this could be done with

- **Horizontal Partitioning:** Build a set of M base models, with a bootstrap sample from the original dataset of size N, created by drawing random samples with replacement. Each bootstrap contains each original sample K times, where P(K=k) follows a binomial distribution.
- **Vertical Partitioning:** Train learners on different subsets of features. It could be done by local or global randomization. The first one consists of the model focusing on diverse parts of it for diverse features. The second one consists of assigning one model to one specific subset of features
- **Base Learner Manipulation:** Varying parameters of the same base learner
- **Heterogeneous Base Learners (Stacking):** Use heterogeneous base learners and obtain ensemble members with different biases

8.11 Combination of Ensembles

Ensembles need to combine the predictions, this could be done by different architectures that consider all the results:

- **Flat:** Aggregating each prediction by some operation
- **Meta-Learner:** The prediction of each model is used for another model that combines them
- **Hierarchical:** In some way, a hierarchical priority is built between models
- **Network:** Use an ANN to combine the results

We can also choose to arrange some voting system to choose the better model:

- **Majority:** The final prediction is the most voted
- **Weighted Majority:** assign weights to the prediction, giving diverse weights to each vote
- **Rank:** Each model could give not only the prediction but a rank of predictions. The final result is the average of the ranks
- **Abstaining:** Only some models are allowed to vote. For example, those that comply certain conditions after a concept drift
- **Relational:** External knowledge is inserted in order to state some relation between model predictions

8.12 Adaptation of Ensembles

Ensembles need to adapt to evolving data. This could be done by

- **Cardinality:** Fixed numbers of base learners
- **Dynamic:** Adding and removing classifiers during the run is allowed.

8.13 Ensemblers Models for SML

- **Online bagging:** Since data streams are supposed to be unbounded, the binomial distribution tends to a Poisson(1). For each learner, we use a weight k=Poisson($\lambda = 1$), and we use a subset of instances to train the model with this weight
- **Leveraging Bagging:** Add an ADWIN drift detector per base learner. Plus, it uses instead a Poisson(6) as a weight assigner.
- **Adaptive Random Forest (ARF):** It uses Hoeffding Trees as base learners. It uses leveraging bagging plus local random subspaces (different features and data instances). It uses a flat architecture. For the voting, it can use majority voting, otherwise, the Naive Bayes approach over the prediction of each tree, or an adaptive mix between both. Finally, it uses an adaptive window + a warning period (train background learners)
- **Streaming Random Patches (SRP):** It uses any base learner, using leverage bagging plus global random subspaces. Plus, it uses a flat architecture, using just the base learner's voting strategy. It has an adaptive window plus a warning period

The Hoeffding Bound is used, to build a decision tree for deciding when a split is necessary, to check any statistical evidence that the number of samples seen by a node is enough

8.14 Regression Models

We can also use regression models to make predictions over the data. Time Series Analysis models learn how the past influences future behaviors without knowing the respective future time series value. Instead, SML regression models learn how to predict data coming from a continuous flow. To predict a new label, they must see the respective features.
In the case of Regression Models, the model ensemble can be done by some operations like mean or median
Some of the models are the following

- **Online k-nearest neighbors regressor:** It uses a fixed-size sliding window to save the instances. Then, find the k nearest neighbors to the new sample in input and predict the mean/weighted mean or median of that neighbor's target features
- **Adaptive Model Rules (AMRules):** The *antecedent* of a rule is a set of literal conditions based on multiple attribute values. The *consequent* of a rule is a function that minimizes the MSE of the target attribute computed from the set of samples covered by rule.
  Each rule uses a Page-Hinkley test to detect and react to changes by pruning the rule set. Each rule is also equipped with outliers detection mechanisms to avoid model adaptation using anomalous examples. Multiple rules form a set of rules, similar to a tree. Hoeffding bound is used to grow the set. Then, the prediction strategy between the rules can be the mean, a regression model or a mix of them.
- **HT Regressor:** It uses an incremental decision tree, with a Hoeffding bound to split over nodes. The voting is done with mean, regression model or adaptively.

- **HAT Regression:** HT Regressor, but with an adaptive window plus a warning period
- **ARF Regressor:** It is the same than ARF, buses Hoeffding Tree Regressors and the voting system changes to the regression way
- **SRP Regressor:** The same as SRP, but with regressor models.

9 SML implementation

9.1 Train a model

9.1.1 The traditional approach

```
from sklearn.naive_bayes import GaussianNB as GaussianNBSKL   # Renamed to
    avoid name conflicts

model = GaussianNBSKL()
model.fit(X=X_train, y=y_train)
y_pred = model.predict(X=X_test)
```

9.1.2 The Stream Learning approach

River is a Python library for streaming machine learning.

```
from river.naive_bayes import GaussianNB

model = GaussianNB()
for xi, yi in zip(X_train, y_train):
    x = dict(zip(feature_names, xi)) # In this case we format the data
    model.learn_one(x=x, y=yi)              # Train the model

y_pred = []
for xi in X_test:
    x = dict(zip(feature_names, xi)) # In this case we format the data
    y_pred.append(model.predict_one(x=x))   # Predict class-label
```

There are a small number of incremental methods available in 'scikit-learn' as well. They use the method partial_fit() so it can train using mini-batches

9.2 Online Metrics

We can calculate metrics online adding elements to the model

```
from river.stream import iter_sklearn_dataset

y_pred = []
y_true = []

model = GaussianNB()

for x, y in iter_sklearn_dataset(iris, shuffle=True, seed=42):
    y_p = model.predict_one(x)   # Predict class-label
    model.learn_one(x, y)        # Train the model
    if y_p is None:              # Process next sample if the model is empty
        continue
```

```
    y_pred.append(y_p)
    y_true.append(y)

print(f'Model accuracy: {accuracy_score(y_true, y_pred):.4f}')
```

But we can use river tools to do it

```
from river.metrics import Accuracy

model = GaussianNB()
metric = Accuracy()

for x, y in iter_sklearn_dataset(iris, shuffle=True, seed=42):
    y_p = model.predict_one(x)   # Predict class
    if y_p is not None:
        metric.update(y_true=y, y_pred=y_p)
    model.learn_one(x, y)        # Train the model

print(f'{len(y_pred)} samples analyzed.')
print(metric)
```

Finally, we can use the progressive_val_score method

```
from river.evaluate import progressive_val_score

# Setup stream and estimators
stream = iter_sklearn_dataset(iris, shuffle=True, seed=42)
nb = GaussianNB()
metric = Accuracy()

# Setup evaluator
progressive_val_score(dataset=stream, model=nb, metric=metric, print_every
    =10)
```

9.3 Concept Drift Detectors

ADWIN

```
from river.drift import ADWIN
drift_detector = ADWIN()
```

Page Hinkley

```
from river.drift import PageHinkley
drift_detector = PageHinkley()
```

CUSUM

```
class CUSUM():
    def __init__(self,delta,lamb,min_obs):
        # Initialization
        self._n = 1
        self._x_mean = 0.0
        self._sum = 0.0
        self._delta = delta
        self._lambda = lamb
        self._min_obs = min_obs
        self.warning_detected = False
        self.drift_detected = False
```

```
12
13      def update(self,value):
14          self._x_mean += (value - self._x_mean) / self._n
15          self._sum = max(0,self._sum + value - self._x_mean - self._delta)
16          self._n += 1
17
18          if self._n >= self._min_obs and self._sum > self._lambda:
19              self.drift_detected = True
20
21      def reset(self):
22          self._n = 1
23          self._x_mean = 0.0
24          self._sum = 0.0
25          self.drift_detected = False
26  drift_detector = CUSUM(delta=0.005,lamb=50,min_obs=30)
```

For these three methods, the detection should be performed as

```
drifts = []
for i, val in enumerate(data_stream):
    drift_detector.update(val)           # Data is processed one sample at a
    time
    if drift_detector.drift_detected:
        print(f'Change detected at index {i}')
        drifts.append(i)
```

DDM

```
from river.drift.binary import DDM
drift_detector = DDM()
```

EDDM

```
from river.drift.binary import EDDM
drift_detector = EDDM()
```

For these two methods, the detection should be performed as

```
drifts = []
warnings = []
warning = -1
for i, val in enumerate(data_stream):
    drift_detector.update(val)           # Data is processed one sample at a
    time
    if drift_detector.warning_detected:
        warning = i
    if drift_detector.drift_detected:
        if warning != -1:
            print(f'Warning detected at index {warning} and Change detected
    at index {i}')
            warnings.append(warning)
            warning = -1
        else:
            print(f'Change detected at index {i}')
        drifts.append(i)     [language=Python]
```

9.4 SML Classification

Naive Bayes

```
from river.naive_bayes import GaussianNB

model = GaussianNB()
```

K-Nearest Neighbors

```
from river.neighbors import KNNClassifier,SWINN
import functools
from river import utils

l1_dist = functools.partial(utils.math.minkowski_distance, p=1)
model = KNNClassifier(n_neighbors=5, engine=SWINN(dist_func=l1_dist,seed=42,
    maxlen=1000))
```

Hoeffding Tree

```
from river.tree import HoeffdingTreeClassifier

model = HoeffdingTreeClassifier()
```

Hoeffding Adaptive Tree

```
from river.tree import HoeffdingAdaptiveTreeClassifier

model = HoeffdingAdaptiveTreeClassifier(seed=42)
```

These models could be visually evaluated with the analysis of metrics

```
metrics = Rolling(Metrics(metrics=[Accuracy(),BalancedAccuracy(),
    GeometricMean(),CohenKappa()]),window_size=500)
stream = iter_pandas(X=data[features], y=data['class'])

steps = iter_progressive_val_score(dataset=stream,model=model,metric=metrics,
    step=1)
plot_results(steps,'GaussianNB',rolling=True)
```

9.5 Ensembles

First, we create a function to visualize the results of the different models

```
# Function to plot the results
def plot_results(steps, plot_title, rolling):
    res = []
    for step in steps:
        if rolling:
            res.append([step['Rolling'].get()[0], step['Rolling'].get()[1],
    step['Rolling'].get()[2], step['Rolling'].get()[3]])
        else:
            res.append([step['Accuracy'].get(), step['BalancedAccuracy'].get
    (), step['GeometricMean'].get(), step['CohenKappa'].get()])
    fig = plt.figure()
    plt.plot(res, label=['Accuracy','BalancedAccuracy','GeometricMean','
    CohenKappa'])
    plt.xlabel('Time', fontsize=13)
    plt.ylabel('Metrics', fontsize=13)
    plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
    plt.title(plot_title)
    plt.show()
```

Then, we have to run the models and follow this pipeline to evaluate the results

```
model = # We choose the model

metrics = Metrics(metrics=[Accuracy(),BalancedAccuracy(),GeometricMean(),
    CohenKappa()])
stream = iter_pandas(X=data[features], y=data['objective'])

steps = iter_progressive_val_score(dataset=stream,model=model,metric=metrics,
    step=1)
plot_results(steps,'Name',rolling=False)
```

In case to cope with concept drift, we change the metric line to

```
metrics = Rolling(Metrics(metrics=[Accuracy(),BalancedAccuracy(),
    GeometricMean(),CohenKappa()]),window_size=500)
## And Plot results with rolling=True
plot_results(steps, f'ADWIN {model_name}',rolling=True)
```

The models are the following:

Online Bagging

```
from river.ensemble import BaggingClassifier
from river.tree import HoeffdingTreeClassifier
model = BaggingClassifier(model=HoeffdingTreeClassifier(),
                          n_models=10,
                          seed=42)  # We can choose the model
```

Leverage Bagging

```
from river.ensemble import LeveragingBaggingClassifier
from river.tree import HoeffdingTreeClassifier
model = LeveragingBaggingClassifier(model=HoeffdingTreeClassifier(
    nominal_attributes=list_nominal_attr),n_models=10,seed=42)
```

Adaptive Random Forest

```
from river.forest import ARFClassifier
model = ARFClassifier(n_models=10, nominal_attributes=list_nominal_attr)
```

Streaming Random Patches

```
from river.ensemble import SRPClassifier
from river.tree import HoeffdingTreeClassifier

model = SRPClassifier(model=HoeffdingTreeClassifier(nominal_attributes=
    list_nominal_attr),
                      n_models=10,
                      seed=42,
                      drift_detector=ADWIN(delta=0.001), # In case of drift
    det
                      warning_detector=ADWIN(delta=0.01)) # In case of drift
    det
```

ADWIN Online Bagging: Online Bagging with ADWIN detector

```
from river.ensemble import ADWINBaggingClassifier
from river.tree import HoeffdingTreeClassifier

model = ADWINBaggingClassifier(model=HoeffdingTreeClassifier(
    nominal_attributes=list_nominal_attr),
                          n_models=10,
                          seed=42)
```

9.6 Regression

The standard procedure to analyze regression models is

```
from river.stream import iter_pandas
from river.metrics.base import Metrics
from river.metrics import MAE,MAPE,MSE,RMSE,base
from river.evaluate import progressive_val_score
from river.preprocessing import StandardScaler
from river.drift import ADWIN

model = # Choose the model
metrics = Metrics(metrics=[MAE(),MSE(),RMSE(),MAPE()])
stream = iter_pandas(X=data[numerical_features], y=data['objective'])

progressive_val_score(dataset=stream,
                      model=model,
                      metric=metrics,
                      print_every=1000)
```

The linear regression

```
from river.linear_model import LinearRegression

model = (StandardScaler() |
        LinearRegression(intercept_lr=.1))
```

KNNRegressor

```
from river.neighbors import KNNRegressor,SWINN
import functools

l1_dist = functools.partial(utils.math.minkowski_distance, p=1)
model = (StandardScaler() |
        KNNRegressor(n_neighbors=5, engine=SWINN(dist_func=l1_dist,seed=42,
    maxlen=1000)))
```

AMRules

```
from river.rules import AMRules

model = (StandardScaler() |
        AMRules(
            delta=0.01,
            n_min=100,
            drift_detector=ADWIN(),
            pred_type='adaptive'
        ))
```

Hoeffding Tree Regressor

```
from river.tree import HoeffdingTreeRegressor

model = (StandardScaler() |
        HoeffdingTreeRegressor(
            grace_period=100,
            leaf_prediction='adaptive',
            model_selector_decay=0.9,
            nominal_attributes=nominal_attr_list
        ))
```

Hoeffding Adaptive Tree Regressor

```
from river.tree import HoeffdingAdaptiveTreeRegressor

model = (StandardScaler() |
        HoeffdingAdaptiveTreeRegressor(
            grace_period=100,
            leaf_prediction='adaptive',
            model_selector_decay=0.9,
            seed=1,
            nominal_attributes=nominal_attr_list
        ))
```

Adaptive Random Forest Regressor

```
from river.forest import ARFRegressor

model = (StandardScaler() |
        ARFRegressor(
            n_models=10,
            seed=1,
            model_selector_decay=0.9,
            nominal_attributes=nominal_attr_list,
            leaf_prediction='adaptive'
        ))
```

SRP Regressor:

```
from river.ensemble import SRPRegressor
from river.tree import HoeffdingTreeRegressor

model = (StandardScaler() |
        SRPRegressor(
            n_models=10,
            seed=1,
            drift_detector=ADWIN(delta=0.001),
            warning_detector=ADWIN(delta=0.01),
            model = HoeffdingTreeRegressor(
                grace_period=100,
                leaf_prediction='adaptive',
                model_selector_decay=0.9,
                nominal_attributes=nominal_attr_list
            )
        ))
```

This one achieves the best performance with less time to execute to ARF Regressor since using Global Random Subspaces

10 Continual Learning

10.1 Introduction

Similar to SML, CL goal is to improve the model based on experience and keep it updated with changes. It focuses on remembering past knowledge when learning a new one. In this case, concept drifts do not have the same meaning as SML, as it does not consider non-virtual concept drifts.

Definition 10.1: Experience

An experience e_i consists of a large batch of samples D_i. Each experience is available at once and the model can be trained on it for several epochs

D_i is usually split on D_i^{train} and D_i^{test}, that is a key difference between SML and CL
Continual Learning consists of a datastream S divided into experiences. Each experience is supposed to be i.i.d., and, in most of the cases, the distribution of the inputs changes between them
The CL algorithm consists of

$$A^{CL} : (f_{i-1}^{CL}, M_{i-1}, D_i^{train}, t_i) \rightarrow (f_i^{CL}, M_i)$$

where:

- f_i^{CL} : Model learned after the i-th experience
- M_i : Knowledge until the i-th experience
- t_i : The task, optional. It could be seen as a time interval in which the data distribution and the objective function may be fixed

Here are different scenarios we can face when we receive a data stream.

- **Incremental Task Learning:** Task labels during train and test. Each experience has a specific task identifier that allows us to select a specific distribution at inference time to do it more accurate
- **Incremental Domain Learning:** No task labels, each experience has its own class labels, that are shared through experiences
- **Incremental Class Learning:** No task labels. Different class labels during train between experiences

Avoiding forgetting is meaningful for CL, we don't want to lose previous knowledge when incorporating new ones. Regarding forgetting it assumes:

- A new task introduces a new input data distribution
- Drifts are virtual
- A new task introduces a new subproblem that should not contradict the previous ones since it specifies a new feature subspace

Hyperparameter tuning and model selection are still open issues in CL. After changing the hyperparameters, the past experiences are forgotten, and the training starts from the following experience.

10.2 Stability-plasticity dilemma

Definition 10.2: Plasticity

Ability to acquire new knowledge and keep learning over time.

Definition 10.3: Stability

Ability to remember the past acquired knowledge over time

There is a trade-off between plasticity and stability. Too much plasticity causes forgetting past knowledge, and too much stability causes difficulties in learning new knowledge In this sense, we want to avoid:

Definition 10.4: Catastrophic forgetting

Training on new experiences reduces performance on previously learned ones

Definition 10.5: Model with no plasticity

The model does not keep learning as new experiences are observed (or they do not learn enough)

On the contrary, we want to achieve both:

Definition 10.6: Forward transfer

Learning a new experience improve future experiences performance

Definition 10.7: Backward transfer

Learning a new experience improves past experiences performance

Backward transfer is usually close to 0 when the model is good, it is hard to have a positive one

10.3 Evaluation

After each experience training, we compute the metrics on all the experience test sets (that are supposed to be all available). The next metrics are calculated over $R_{i,j}$, the accuracy of the model over the test experience j after being trained with the train experience i

- **Average Accuracy:** Mean of final performance on the experiences tests

$$ACC = \frac{1}{T}\sum_{i=1}^{T} R_{T,i}$$

- **A Metric:** For each experience training, we test over only test experiences that came before it

$$A = \frac{\sum_{i=1}^{N} \sum_{j=1}^{i} R_{i,j}}{\frac{N(N+1)}{2}}$$

- **Forward Transfer Metric:**

$$\frac{1}{T-1} \sum_{i=2}^{T} R_{i-1,i} - \bar{b}_i$$

where $\bar{b}_i$ indicates the accuracy of the random model

- **Backward Transfer Metric:**

$$\frac{1}{T-1} \sum_{i=i}^{T-1} R_{T,i} - R_{i,i}$$

A negative value indicates forgetting

Performance is not enough. Excess memory occupation and computational overhead can appear.

11 Continual Learning Strategies

11.1 Strategies

The strategies in CL aim to learn new tasks without losing the prediction ability on old ones, i.e., they are ways to achieve the objectives of CL

- **Replay:** During each experience, some old examples are appended to make the model not forgetting old tasks. The training set sizes increase with the number of tasks
- **Regularization:** Changing the loss function by adding regularization terms to the new task problem to avoid losing the old ones.
- **Architectural:** Introducing new parameters to the network. Sometimes the model has parameters that depend on the task number. For some strategies, the number of parameters could increase with the number of tasks

11.2 Baselines

Some baselines of the continual learning strategies are the following

- **Naive Strategy:** It simply continues the training on the new experience.
- **Cumulative Strategy:** Each experience's training concatenates the current experience's training set with all the previous ones
- **Joint Strategy (Offline):** It trains the model on all the experience's training set together

A possible design choice for incremental learning tasks is to use a multi-head model. It adds a new head for each task. There are parameters associated with old and others with new tasks. The network has also shared parameters. It requires the task label to select the correct head, although some approaches try to recognize the tasks using an autoencoder.

11.3 Replay Strategy

It uses a fixed-size random memory (RM) to store a subset of random previous experiences' data points. During the training of the ith experience, it trains the model on the ith training set shuffled with RM. RM contains a random subset of the data points of the previous experience's training sets. After the training, it randomly substitutes some data points with a random subset of the current experience's training set in the RM.

Algorithm 5 Reply Strategy

Require: RM_{size} = number of patterns to be stored in RM
Require: $\forall i\ B_i$
  $RM = \emptyset$
  for each training batch B_i **do**
    Train the model on shuffled $B_i \cup RM$
    $h = \frac{RM_{size}}{i}$
    R_{add} = random sampling h patterns from B_i
    $R_{replace} = \begin{cases} \emptyset & i == 1 \\ \text{random sample h patterns from RM} & \text{otherwise} \end{cases}$
    $RM = (RM - R_{replace}) \cup R_{add}$
  end for

11.4 Regularization

11.4.1 Elastic Weight Consolidation (EWC)

It adds a regularization term to the loss to penalize changes in important parameters for previous tasks, minimizing the change in important parameters. The idea is to search for the optimal configuration lying in the overlapping of different tasks solution space.
The expression for the loss is the following:

$$\mathcal{L}(\theta) = \mathcal{L}_B(\theta) + \sum_i \frac{\lambda}{2} F_i(\theta_i - \theta^*_{A,i})^2$$

where

- $\mathcal{L}_B(\theta)$ is the loss on task B
- F_i is the importance index of parameter i for task A
- $(\theta_i - \theta^*_{A,i})^2$ is the distance between the current value of parameter i and the optimal one for task A

11.4.2 Learning without Forgetting (LwF)

It is usually applied to multi-head models. When the model is learning a new task, it uses knowledge distillation (KD) to reproduce the outputs of the previous tasks on the latest data.
The new training objective is:

$$\min_{\theta_s\theta_o,\theta_n} (\lambda\mathcal{L}_{old}(Y_o,\hat{Y}_o) + (1-\lambda)\mathcal{L}_{new}(Y_n,\hat{Y}_n) + \mathcal{R}(\theta_s\theta_o,\theta_n))$$

where

- $\mathcal{L}_{old}$ is the KD loss for the old task
- λ is the weighted sum coefficient, that indicates the relevance of the KD loss
- $\mathcal{L}_{new}$ is the current task loss
- $\mathcal{R}$ is the regularization term

This model does not require storing the previous data, it only needs the previous model's predicted labels on the current dataset before starting the training.

11.5 Reply+Regularization

The Average Gradient Episodic Memory (AGEM) applies reply and regularization strategies at the same time.
At every training step, mini-batch, it ensures that the average loss on the old tasks does not decrease when learning the current one. It ensures it by randomly choosing a mini-batch from the buffer and computing the gradient g_{ref} on it. Then, it computes the gradient g on the current mini-batch. If $g^T g_{ref} \geq 0$, it uses g for the gradient update. Otherwise, it projects g to have $g^T g_{ref} = 0$

11.6 Architectural

11.6.1 Progressive Neural Networks (PNN)

It starts with a single neural network. Whenever a new task appears, it freezes the parameters of the old columns and adds a new column.
The hidden layer i ($i > 1$) of the column k receives the output of the hidden layer i-1 of k, and the outputs of the hidden layers i-1 of all previous columns j ($j < k$)
It uses transfer learning to combine the previous knowledge with the current one. It freezes old columns to avoid forgetting. Plus, as the multi-head models, it needs the task label.

11.6.2 Copy weights with Re-init+ (CWR+)

It is meant to deal with class-increasing scenarios.
The network has shared weights Θ that are trained only on the first experiences and frozen. It uses temporary weights tw during training, and consolidated weights cw during inference. The algorithm is indicated in algorithm 6.

Algorithm 6 CWR+ Strategy

Require:
cw = 0
init $\bar{\Theta}$ random or from a pre-trained model as ImageNet
for each training batch B_i **do**
  Expand output layer with s_i neurons for the new classes in B_i
  $tw = 0$ for every neuron in the output layer
  Train the model with SGD on the s_i classes of B_i
  if $B_i = B_1$ **then**
    Learn Both $\bar{\Theta}$
  else
    Learn tw while keeping $\bar{\Theta}$ fixed
  end if
  for each class j among the s_i classes in B_i **do**
    cw[j] = tw[j] - avg(tw)
  end for
  Test the model by using $\bar{\Theta}$ and cw
end for

12 Continual Learning Implementation

Avalanche is an end-to-end python open source library that provides diverse modules to implement continual learning

12.1 Imports

```
from avalanche.benchmarks.classic import SplitCIFAR10
from avalanche.models import SimpleMLP, MTSimpleMLP
from avalanche.training.plugins import EvaluationPlugin
from avalanche.evaluation.metrics import accuracy_metrics,\
    forward_transfer_metrics,bwt_metrics,\
    ram_usage_metrics, timing_metrics, EpochAccuracy
from avalanche.training.plugins import EarlyStoppingPlugin
from torch.nn import CrossEntropyLoss
from torch.optim import SGD
import torch

import numpy as np
import pickle
import os
import matplotlib.pyplot as plt
import numpy as np
```

12.2 Build functions

```
# function that creates the base model (a simple MLP)
def build_model():
  return SimpleMLP(num_classes=2,input_size=32*32*3, hidden_size=512)

# Function that creates the base model with multiple heads.
# The architecture contains the feature extractor that is shared by the tasks
    .
```

```
# Whenever a new task arises, a new head (a task-specific output layer) is
# added with task-specific weights.
# During the inference phase a Multi-Head model requires the task label to
# select the associated head.
def build_model_mt():
  return MTSimpleMLP(input_size=32*32*3, hidden_size=512)

# function that creates the evaluation plugin
# it is used to define the metrics to compute
def build_eval_plugin():
  return EvaluationPlugin(
      accuracy_metrics(experience=True, stream=True),
      # compute the accuracy on each experience's test set and on on the
    entire
      # test stream
      bwt_metrics(experience=True,stream=True),
      # the same for the backward transfer metric
      forward_transfer_metrics(experience=True,stream=True),
      # the same for the forward transfer metric
      timing_metrics(epoch=True),
      # compute the time metrics during the training phase
      ram_usage_metrics(epoch=True, every=0.001),
      # compute the max ram usage metrics during the training phase,
      # update it every 0.001 seconds
      loggers=[]
      # we do not use loggers, we save the results using the dictionary
    returned
      # by the train method of the strategy
  )

# function that builds the optimizer given the model
def build_optimizer(model):
  return SGD(model.parameters(), lr=0.001, momentum=0.9)

# function that creates the loss function
def build_criterion():
  return CrossEntropyLoss()
```

```
# utility functions to take the correct keys in the metrics dictionary
def acc_exp_str(exp):
  exp = "{:03d}".format(exp)
  return f'Top1_Acc_Exp/eval_phase/test_stream/Task{exp}/Exp{exp}'

def avg_acc_str(n_exp):
  exp = "{:03d}".format(n_exp-1)
  return f'Top1_Acc_Stream/eval_phase/test_stream/Task{exp}'

def fwt_exp_str(exp):
  exp = "{:03d}".format(exp)
  return f'ExperienceForwardTransfer/eval_phase/test_stream/Task{exp}/Exp{exp
    }'

def fwt_final_str():
  return "StreamForwardTransfer/eval_phase/test_stream"

def bwt_exp_str(exp):
  exp = "{:03d}".format(exp)
```

```
  return f'ExperienceBWT/eval_phase/test_stream/Task{exp}/Exp{exp}'

def bwt_final_str():
  return 'StreamBWT/eval_phase/test_stream'

def memory_final_str(exp):
  exp = "{:03d}".format(exp)
  return f'MaxRAMUsage_Epoch/train_phase/train_stream/Task{exp}'

def time_final_str(exp):
  exp = "{:03d}".format(exp)
  return f'Time_Epoch/train_phase/train_stream/Task{exp}'

def build_perf(res):
  perf = {}
  perf["accuracies"] = [[r[acc_exp_str(exp)] for exp in range(N_EXP)] for r
    in res]
  perf["average_accuracy"] = res[-1][avg_acc_str(N_EXP)]
  perf["a_metric"] = np.sum([perf["accuracies"][i][j] for i in range(N_EXP)
    for j in range(i+1)]) / ((N_EXP)*(N_EXP+1)/2)
  perf["fwt_exp"] = [res[-1][fwt_exp_str(exp)] for exp in range(1,N_EXP)]
  perf["fwt"] = res[-1][fwt_final_str()]
  perf["bwt_exp"] = [[None]*(N_EXP-1)] + [[r[bwt_exp_str(exp)] for exp in
    range(i+1)] + [None]*(N_EXP-i-2) for i, r in enumerate(res[1:])]
  perf["bwt"] = [r[bwt_final_str()] for r in res]
  perf["time"] = [res[-1][time_final_str(exp)]*TRAIN_EPOCHS for exp in range(
    N_EXP)]
  perf["time_cum"] = np.cumsum(perf["time"])
  perf["ram"] = [res[-1][memory_final_str(exp)] for exp in range(N_EXP)]
  return perf
```

12.3 Benchmark

```
benchmark = SplitCIFAR10(
    n_experiences=N_EXP,
    return_task_id=True,
    class_ids_from_zero_in_each_exp=True,
    seed=42
)
```

12.4 Model Training

```
res = []
for experience in benchmark.train_stream:
    res.append(model.train(experience, eval_streams=[benchmark.test_stream]))

metric_matrices = build_perf(res)
# We can access to
metric_matrices["accuracy"]
metric_matrices["bwt_exp"] # Backward transfer
metric_matrices["fwt_exp"] # Forward transfer
metric_matrices["time"] # Training Time
```

12.5 Naive Model

```
from avalanche.training.supervised import Naive
strategy_name = "naive"

# create the base learner
model = build_model()

# create the strategy
cl_strategy = Naive(
    model, # the base model
    build_optimizer(model), # the optimizer
    build_criterion(), # the loss function
    train_mb_size=TRAIN_MB_SIZE, # mini-batch size during training
    train_epochs=TRAIN_EPOCHS, # number of training epochs
    eval_mb_size=TRAIN_MB_SIZE, # mini-batch size during the evaluation
    evaluator=build_eval_plugin(), # evaluator plugin
    eval_every=1, # eval every epoch
    plugins=[EarlyStoppingPlugin(patience=2, val_stream_name='train')],
    # we should have used a validation data set
    device=torch.device("cuda")
)

# Train the model as past section
```

12.6 Cumulative Strategy

```
from avalanche.training.supervised import Cumulative
strategy_name = "cumulative"

model = build_model()
cl_strategy = Cumulative(
    model=model,
    optimizer=build_optimizer(model),
    criterion=build_criterion(),
    train_mb_size=TRAIN_MB_SIZE,
    train_epochs=TRAIN_EPOCHS,
    eval_mb_size=TRAIN_MB_SIZE,
    evaluator=build_eval_plugin(),
    eval_every=1,
    plugins=[EarlyStoppingPlugin(patience=2, val_stream_name='train')],
    device=torch.device("cuda")
```

12.7 Replay Strategy

```
from avalanche.training.supervised import Replay
strategy_name = "replay"

model = build_model()
cl_strategy = Replay(
    model=model,
    optimizer=build_optimizer(model),
    criterion=build_criterion(),
    train_mb_size=TRAIN_MB_SIZE,
```

```
    train_epochs=TRAIN_EPOCHS,
    eval_mb_size=TRAIN_MB_SIZE,
    evaluator=build_eval_plugin(),
    eval_every=1,
    plugins=[EarlyStoppingPlugin(patience=2, val_stream_name='train')],
    device=torch.device("cuda")
)
```

12.8 EWC Strategy

```
from avalanche.training.supervised import EWC
strategy_name = "ewc"

model = build_model()
cl_strategy = EWC(
    model=model,
    ewc_lambda=0.4,
    optimizer=build_optimizer(model),
    criterion=build_criterion(),
    train_mb_size=TRAIN_MB_SIZE,
    train_epochs=TRAIN_EPOCHS,
    eval_mb_size=TRAIN_MB_SIZE,
    evaluator=build_eval_plugin(),
    eval_every=1,
    plugins=[EarlyStoppingPlugin(patience=2, val_stream_name='train')],
    device=torch.device("cuda")
)
```

12.9 LWF Strategy

```
from avalanche.training.supervised import LwF
strategy_name = "lwf"

model = build_model()
cl_strategy = LwF(
    model=model,
    optimizer=build_optimizer(model),
    criterion=build_criterion(),
    train_mb_size=TRAIN_MB_SIZE,
    train_epochs=TRAIN_EPOCHS,
    eval_mb_size=TRAIN_MB_SIZE,
    evaluator=build_eval_plugin(),
    eval_every=1,
    alpha = 0.3, temperature= 0.5,
    plugins=[EarlyStoppingPlugin(patience=2, val_stream_name='train')],
    device=torch.device("cuda")
)
```

12.10 PNN Strategy

```
from avalanche.training.supervised import PNNStrategy
from avalanche.models import PNN

```

```
strategy_name = "pnn"

model = PNN(in_features=32*32*3, hidden_features_per_column=512)

cl_strategy = PNNStrategy(
    model=model,
    optimizer=build_optimizer(model),
    criterion=build_criterion(),
    train_mb_size=TRAIN_MB_SIZE,
    train_epochs=TRAIN_EPOCHS,
    eval_mb_size=TRAIN_MB_SIZE,
    evaluator=build_eval_plugin(),
    eval_every=1,
    plugins=[EarlyStoppingPlugin(patience=2, val_stream_name='train')],
    device=torch.device("cuda")
)
```

12.11 GEM (Replay+Regularyzation) Strategy

```
from avalanche.training.supervised import GEM

strategy_name = "gem"

model = build_model()

cl_strategy = GEM(
    model=model,
    optimizer=build_optimizer(model),
    criterion=build_criterion(),
    train_mb_size=TRAIN_MB_SIZE,
    train_epochs=TRAIN_EPOCHS,
    eval_mb_size=TRAIN_MB_SIZE,
    evaluator=build_eval_plugin(),
    eval_every=1,
    patterns_per_exp = 100,
    plugins=[EarlyStoppingPlugin(patience=2, val_stream_name='train')],
    device=torch.device("cuda")
)
```

12.12 Multi-Head Model

In each model, we can do the multi-head strategy by simply changing our model to multi-head

```
strategy_name = "{strategy}_mt"
model = build_model_mt()
```

12.13 Evaluation

We can evaluate the model accuracy over the different experiences after the training on specific ones

```
def plot_strategy_accs(strategy, perf, ax):
  train_exp = [f"Train exp {i}" for i in range(N_EXP)]
```

```
  accs = np.array(perf[strategy]["accuracies"])
  values = {f'Test exp. {i}': accs[:,i] for i in range(N_EXP)}

  cmap = plt.cm.Blues
  x = np.arange(len(train_exp))  # the label locations
  width = 0.15  # the width of the bars
  multiplier = 0

  i = 0
  for attribute, measurement in values.items():
      color = cmap((i+1)*(1/(len(values)+1)))
      offset = width * multiplier
      rects = ax.bar(x + offset, measurement, width, label=attribute, color=
    color)
      multiplier += 1
      i += 1

  # Add some text for labels, title and custom x-axis tick labels, etc.
  ax.set_ylabel('Accuracy')
  ax.set_xticks(x + 1.5*width, train_exp)
  ax.legend(bbox_to_anchor=(1,1), loc="upper left")
  ax.set_title(strategy.capitalize())
```

```
fontsize=10
plt.rcParams.update({'font.size': fontsize})
fig, ax = plt.subplots(figsize=(2.5*len(perf.keys()),20), nrows=len(
    strategies), ncols=1, sharey=True)
for i, strategy in enumerate(strategies):
  plot_strategy_accs(strategy, perf, ax[i])
plt.tight_layout()
```

But we can also calculate mean trends

```
fig, ax = plt.subplots(figsize=(20,5), nrows=1, ncols=1)
for strategy in perf:
  accs = [np.mean(perf[strategy]["accuracies"][i][:i+1]) for i in range(len(
    perf[strategy]["accuracies"]))]
  ax.plot(accs, label=strategy, marker="o")
ax.set_xticks(ticks=np.arange(len(accs)), labels=[f"Train exp. {i}" for i in
    range(N_EXP)])
ax.set_ylabel("Accuracy on the experiences seen so far")
ax.legend()
plt.tight_layout()
```

We can remove some strategy plot that is performing far from the others so we can zoom in on the difference between the closer ones

Then we can summarize the results with pandas or numpy

12.14 Backward Transfer Learning

```
def plot_strategy_bwts(strategy, perf, ax):
  train_exp = [f"Train exp {i}" for i in range(1, N_EXP)]
  bwt = np.array(perf[strategy]["bwt_exp"])[1:,:]
  bwt[bwt == None] = 0
  values = {f'Test exp. {i}': bwt[:,i] for i in range(N_EXP-1)}

  cmap = plt.cm.Blues
```

```
  x = np.arange(len(train_exp))  # the label locations
  width = 0.15  # the width of the bars
  multiplier = 0

  i = 0
  for attribute, measurement in values.items():
      color = cmap((i+1)*(1/(len(values)+1)))
      offset = width * multiplier
      rects = ax.bar(x + offset, measurement, width, label=attribute, color=
    color)
      multiplier += 1
      i += 1
  ax.plot([ax.get_xlim()[0], ax.get_xlim()[1]], [0, 0], "k--", color="grey",
    linewidth=1)

  # Add some text for labels, title and custom x-axis tick labels, etc.
  ax.set_ylabel('Backward Transfer')
  ax.set_xticks(x + 1.5*width, train_exp)
  ax.set_yticks([0], minor=True)
  ax.legend(bbox_to_anchor=(1,1), loc="upper left")
  ax.set_title(strategy.capitalize())
```

```
fig, ax = plt.subplots(figsize=(2.5*len(perf.keys()),20), nrows=len(
    strategies), ncols=1, sharey=True)
for i, strategy in enumerate(strategies):
  plot_strategy_bwts(strategy, perf, ax[i])
plt.tight_layout()
```

12.15 Forward Transfer Learning

```
def plot_strategy_ftws(strategy, perf, ax):

  x = [f"Train exp {i}" for i in range(N_EXP-1)]
  values = perf[strategy]["fwt_exp"]

  cmap = plt.cm.Blues

  ax.bar(x, values)
  ax.plot([ax.get_xlim()[0], ax.get_xlim()[1]], [0, 0], "k--", color="grey",
    linewidth=1)

  # Add some text for labels, title and custom x-axis tick labels, etc.
  ax.set_ylabel('Forward Transfer')
  ax.set_title(strategy.capitalize())
```

```
fig, ax = plt.subplots(figsize=(2.5*len(perf.keys()),20), nrows=len(
    strategies), ncols=1, sharey=True)
for i, strategy in enumerate(strategies):
  plot_strategy_ftws(strategy, perf, ax[i])
plt.tight_layout()
```

12.16 Resources usage

RAM

```
df = pd.DataFrame([[strategy, perf[strategy]["ram"][-1]] for strategy in
    strategies], columns=["Strategy", "Max RAM Usage"])
df.sort_values("Max RAM Usage", ascending=True)
```

Time

```
df = pd.DataFrame([[strategy, perf[strategy]["time_cum"][-1]] for strategy in
    strategies], columns=["Strategy", "Total Training Time"])
df.sort_values("Total Training Time", ascending=True)
```

13 Continual Learning vs Streaming Machine Learning

13.1 Concept Drifts

A concept is the unobservable random process that produces the data points
A concept drift is an unforeseeable change to the data-generating process that implies a more significant change in the data's statistical properties than chance fluctuations.
A real concept drift makes a change in $p(y|X)$, while a virtual drift changes $p(X)$, but not $p(y|X)$, i.e., does not change the boundaries of the data, but the internal distribution in each class.

13.2 Continual Learning

Since the real concept drifts produce a contradiction between the current and past data, CL with regularization and replay is not able to predict correctly, and the model won't converge. On the contrary, architectural strategies can isolate a subpart of the network for those specific tasks. The challenge is to identify the task during inference.

13.3 Streaming Machine Learning

SML usually ignores the problem of forgetting. It only focuses on the current task

References

Cormode, G., & Muthukrishnan, S. (2005). An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, *55*(1), 58-75. Retrieved from https://www.sciencedirect.com/science/article/pii/S0196677403001913 doi: https://doi.org/10.1016/j.jalgor.2003.12.001

della Valle, E. (2024a). *Streaming data analytics.* Retrieved from https://github.com/Streaming-Data-Analytics (Accessed: 2024-02-15)

della Valle, E. (2024b). *Streaming data analytics 2024-25.* Retrieved from https://emanueledellavalle.org/teaching/streaming-data-analytics-2024-25/ (Accessed: 2024-02-15)

Greenwald, M., & Khanna, S. (2001, May). Space-efficient online computation of quantile summaries. *SIGMOD Rec.*, *30*(2), 58–66. Retrieved from https://doi.org/10.1145/376284.375670 doi: 10.1145/376284.375670

Jain, R., & Chlamtac, I. (1985, oct). The p2 algorithm for dynamic calculation of quantiles and histograms without storing observations. *Communications of the ACM*, *28*(10), 1076–1085. Retrieved from https://doi.org/10.1145/4372.4378 doi: 10.1145/4372.4378

Jeon, M. S. (2024). *Artificial neural networks and deep learning.* Retrieved from https://github.com/minsoos/class_notes/blob/main/ANN_notes.pdf (Accessed: 2025-02-25)