Big Data

Lezione 1 - Introduzione

I dati al giorno d'oggi vengono prodotti velocemente, sono eterogenei e ricchi di informazioni. Esiste la regola delle 3 V:

- Volume: vanno da pochi MB a PB
- Velocità: da batch, a periodici fino a real time
- Varietà: tabelle, database, log, foto, audio, social e non strutturati

In generale si definisce come big data qualsiasi dato la cui dimensione è un problema per operazioni di conservazione, trasmissione ed elaborazione. L'idea è quella di scalare orizzontalmente. Questo comporta la richiesta di più spazio fisico, costi energetici e rete di comunicazione, ma migliora i costi, fault tolerance e scalabilità.

Distinguiamo la High Performance Computing e la Big Data Computing:

- HPC funziona come un cluster di macchine indipendenti dette nodi, che comunicano in una rete ad alta velocità con accesso ad uno storage centralizzato. Dato che ogni macchina ha una sua RAM, si dice architettura a memoria distribuita. Avendo diversi nodi questi devono coordinarsi correttamente, ciò viene fatto dal nodo master che si occupa dell'orchestrazione, mentre i nodi worker si occupano del calcolo effettivo. Possono comunicare con Message Passing Interface. Il problema è che in questo modo dato che la memoria di massa è centralizzata questa non scala facilmente.
- Nei sistemi a Big Data ogni nodo ha il suo storage e l'unica cosa che li lega è la rete di comunicazione. Ovviamente, devono agire come cluster, pertanto dobbiamo adottare dei File System distribuiti come Hadoop Distribuited File System, e sistemi di calcolo distribuito come Spark.

Link utile: https://colab.research.google.com/

Esame: Scritto 25%, Progetto 75% da consegnare entro 60 giorni dal superamento dello scritto, laboratorio in aula che dà 3 punti extra.

Lezione 2 - Map Reduce

Serve come piattaforma di storage ed analisi di big data, permette a programmatori senza esperienza di sistemi distribuiti di utilizzare le risorse di un data center per elaborare grandi moli di dati.

Il sistema Hadoop è composto da:

- **MapReduce Runtime (coordinatore):** paradigma per programmazione distribuita
- **Hadoop Distribuited FIle System (HDFS):** per permettere alle macchine di agire come cluster, implementando ridondanza dei dati e fault tolerance

Ovviamente nasce dall'esigenza di dover scalare orizzontalmente piuttosto che verticalmente. Vogliamo processare molti dati, in maniera distribuita e semplice. Questa soluzione è proprio quella di MapReduce. Si utilizza un file system distribuito, dividendo i file in chunk, ognuno di essi verrà distribuito sul cluster e replicato. Si ragionerà in termini di nodi master, in HDFS è Name Node che appunto memorizza metadati di dove si trovano i chunks dei file, e i Data Node, che immagazinano e reuperano i blocchi se richiesto.

Le sfide sono molte, come per esempio

- **Map**
  - Come itero su molti dati in parallelo?
  - Come estraggo informazioni per ogni iterazione?
  - Come ordino i risultati?
- **Reduce**
  - Come aggrego i dati intermedi?
  - Come genero l'output finale?

Il paradigma MapReduce fonda le radici nella programmazione funzionale, dove Map prende una funzione e la applica ad ogni elemento, mentre Fold applica iterativamente una funzione per aggregare i risultati. Tra l'altro processando molti dati è complicato farli navigare in rete, pertanto è meglio avvicinare le unità di calcolo ai dati effettivi.

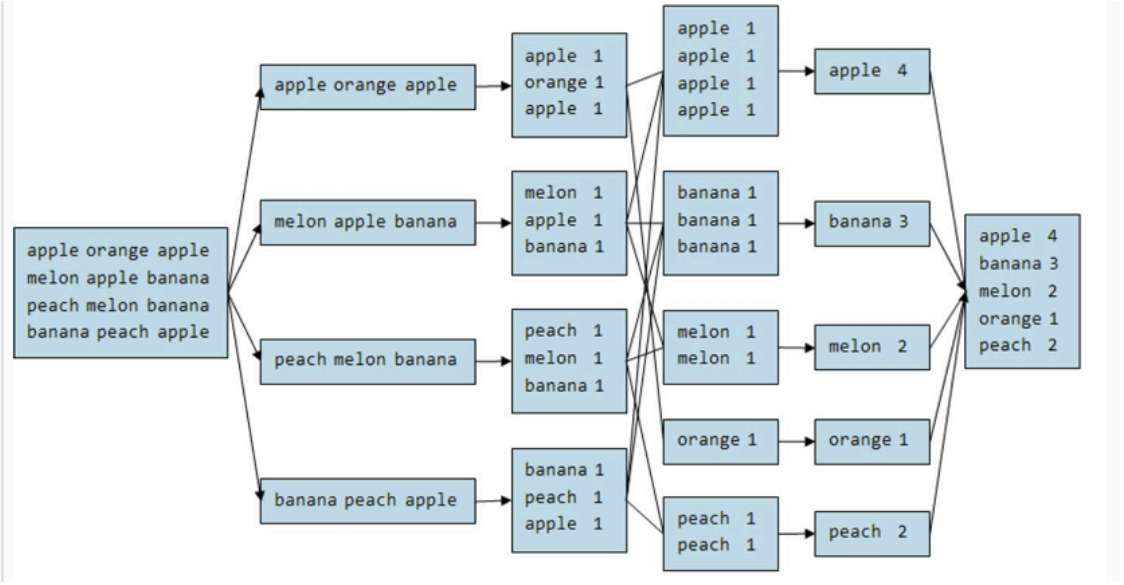
In MapReduce si ha:

- **Map:** prende in input un oggetto key-value (k, v) e restituisce $(k_1, v_1), \ldots, (k_n, v_n)$
- **Group by key:** colleziona le coppie con stessa chiave k e vi associa tutti i valori con una lista.
- **Reduce:** prende in input l'elemento $(k, [v_1, \ldots, v_n])$ e li combina in un certo modo.

La pipeline è sempre la stessa, variano le funzioni applicate da Map e Reduce.

Più nello specifico, la fase Map prende n elementi key-value, e ne restituisce m intermedi. Questi vengono raggruppati per chiave, creando elementi con chiave e una lista di valori associato, e la Reduce riduce la lista rimanente restituendo elementi key-value.

Un tipico problema risolvibile con MapReduce è il word counting, altamente parallelizzabile.



Vediamo il codice:

```
map(key, value):
    # key: document name, value: text
    for w in value:
        emit(w,1)

reduce(key, values):
    # key: a word, value: an iterator over counts
    result = 0
    for v in values:
        result += v
    emit(key, result)
```

L'ambiente MapReduce si occupa di:

- Partizionare i dati in input
- Schedula l'esecuzione del codice sulle varie macchine
- Effettua la group by key
- Gestisce i crash delle macchine
- Gestisce la comunicazione tra macchine

Un altro esempio di problema risolvibile con MapReduce è contare quante parole di una certa lunghezza esistono in una collezione di documenti, l'input è proprio l'insieme dei documenti.

```
map(key, value):
    # key: document name; value: text of the document
    for w in value:
       emit(length(w), 1)

reduce(key, values):
   # key: a word; value: an iterator over counts
   result = 0
   for w in values:
       result += 1
   emit(key, result)
```

Ovviamente input e output finale verranno memorizzate nel file system distribuito, mentre i risultati intermedi vengono memorizzati sul file system locale dei worker.

Il nodo che coordina i worker è il Master. Ogni task può essere:

- **Idle:** pronto ad essere eseguito ma non assegnato
- **In-progress:** in esecuzione su un worker
- **Completato:** terminato con successo e risultati pronti

Quando un worker diventa disponibile il Master gli assegna immediatamente un **task idle** da eseguire.

Dopo che un **Map Task** ha completato la sua esecuzione, deve generare dei file intermedi che saranno utilizzati dai **Reduce Task**.

- Ogni **Map Task** scrive più file intermedi, chiamati **R file**, uno per ogni Reduce Task.
- Quando il Map Task termina, comunica al Master:
    - **La posizione** dei file intermedi (es. su HDFS o su un nodo specifico).
    - **La dimensione** dei file intermedi.

  Così il Master può pusharli ai reducer.

Periodicamente il Master invia ping periodici ai Worker per verificare che siano attivi, se uno di essi non risponde allora è crashato e il Master può riassegnare i task ad altri Worker attivi.

Si sceglie il numero di Map task M molto maggiore del numero di nodi del cluster così che ogni nodo possa eseguire più Map Task in parallelo. Il numero di Map Task viene determinato dal numero di chunk Distribuited File System (DFS) quindi per esempio se ci sono per un file 100 chunks allora 100 Map Tasks.

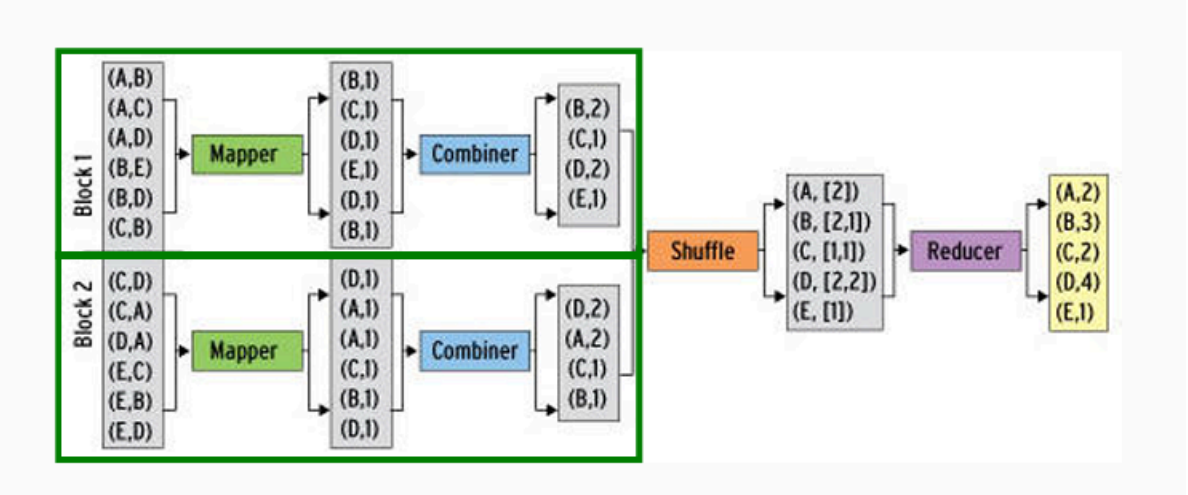
In genere il numero di Reduce Task R è più piccolo di M perchè l'output del Reduce verrà scritto su R file e se ce ne sono troppi è troppo frammentato, inoltre i Reduce Task raccolgono più dati per produrre un risultato più aggregato quindi non c'è bisogno di grande parallelismo.

Potrebbe succedere di avere worker lenti e questo è un problema detto **straggler problem**, e dato che il job MapReduce si può dire terminato solo se tutti i worker hanno finito, se anche un solo worker rallenta allora l'intero processo rallenta. La soluzione è la speculative execution, dove verso la fine dell'elaborazione vengono identificati i task che stanno impiegando troppo tempo e questi sono gli stragglers, per velocizzare viene creata una copia dello stesso task su un altro nodo, chi finisce prima vince e possiamo così migliorare i tempi di esecuzione in caso di nodi problematici.

Un'altra ottimizzazione è l'uso di **combiners**, che permettono di ridurre il traffico di rete, e ciò viene fatto riducendo i dati inviati ai Reducer, ogni map task genera coppie chiave-valore che dopo la fase di shuffle vengono inviati ai Reducer. Se una chiave k appare molte volte si genera un numero elevato di coppie con stessa chiave aumentando il traffico di rete, pertanto possiamo far fare al map parte del task reduce prima di inviarli ai Reducer effettivi. Questa operazione si può fare *solo se la map è associativa e commutativa*. In questo contesto, possiamo avere un caso del genere:

```
Map Task -> (A,1), (A,2), (B,1), (C,1), (C,1), (C,1)
Preprocessing -> (A, 3), (B,1), (C,3)
```

Tali dati verranno mandati alla fase di shuffle che li combinerà eventualmente con altri risultati di altri worker!



Ancora, è possibile definire una **Partition Function** che determina quale Reducer riceverà una determinata chiave. Infatti, dopo la fase di map i dati devono essere mandati ai reducer in maniera bilanciata, se le chiavi non sono distribuite alcuni reducer lavoreranno troppo. A volte quindi è utile personalizzarla, per esempio Hadoop usa:

$$Partition = hash(k) \mod R$$

e a volte potremmo volere qualcosa di più complesso come

$$Partition = hash(hostname(URL)) \mod R$$

garantendo che URL di un host finiscano nella stessa partizione.

Es. 1 Supponiamo di avere un grande corpus web, guardando i metadati vediamo che le linee hanno il formato (URL, dimensioni, data, ...). Vogliamo trovare la somma delle dimensioni delle pagine web di ogni host in byte.

```
map(key, value):
    # key: URL; value: {size,data,...}
    emit(hostname(URL), size)

reduce(key, values):
    # key: hostname; values: iterator over sizes
    result = 0
    for size in values:
        result+=size
    emit(key,result)
```

La funzione reduce prende in input il risultato della fase di shuffle group by key, quindi key sarà l'host e values sarà la lista dei valori calcolata, su cui andremo ad iterare.

Chiarimento su iterator da ChatGPT:

> Quando si dice che values è un **iteratore** sui valori associati a una chiave, significa che il framework MapReduce (es. Hadoop o Spark) non fornisce direttamente una **lista** con tutti i valori raccolti, ma piuttosto un **oggetto che permette di scorrere i valori uno alla volta**.
>
> In pratica non riceviamo una lista completa in memoria, ma un iteratore che ci permette di leggere questi valori in sequenza evitando di dover caricare tutta la lista in memoria, il che è fondamentale quando si lavora con **grandi volumi di dati**.

Es. 2 Dato un grafo diretto descritto con lista di adiacenza costruiamo il grafo con gli archi invertiti.

```
map(key, value):
    # key: filename; value: lista di adiacenza
    for r in value:
        for v in adj(r)
            emit(v,r)

reduce() -> Identity function
```

Es. 3 *(No soluz.)* Immaginiamo di avere un grande corpus web, ogni documento ha un ID. Per ogni parola restituire l'elenco di ID dove la parola appare.

Es. 4 *(No soluz.)* Filtriamo le linee di un insieme di documenti in cui appare una parola X.

Es. 5 *(No soluz.)* Dato un elenco di chunk di un documento ordinali secondo un certo criterio.

Es. Matrix-Vector Multiply *pass*

Lezione 3 - Spark

MapReduce ha rivoluzionato l'elaborazione distribuita, ma scrivere codice direttamente con MapReduce è complesso, soprattutto per problemi che non si adattano facilmente al paradigma di mappatura e riduzione visto che molte applicazioni reali richiedono più passaggi. Inoltre MapReduce basa la sua esecuzione sulla lettura e scrittura su disco tra le fasi di mappatura e riduzione, il che introduce latenze notevoli rispetto all'elaborazione in memoria centrale.

Questo approccio è inefficiente per applicazioni che necessitano di analisi iterative o interattive, come quelle tipiche del machine learning e dell'analisi grafica. Spark è nato proprio per superare queste limitazioni, introducendo un modello di elaborazione in memoria molto più veloce e un'API più intuitiva rispetto a quella di MapReduce, che si basa su un modello di elaborazione a due fasi. I dati passano sequenzialmente dalla prima fase alla seconda, rendendo il modello rigido e limitato nella sua flessibilità.

I sistemi basati su Data-Flow rappresentano un'evoluzione di questo approccio, generalizzando MapReduce:

- Consentono di avere un **numero arbitrario di fasi di elaborazione**, superando la rigida divisione in Map e Reduce
- Permettono l'uso di **operazioni più complesse e personalizzabili**, rendendo possibile la modellazione di workflow più sofisticati
- Finché il flusso di dati rimane unidirezionale, è possibile **isolare i guasti a livello di singoli task**, evitando di dover rilanciare interi job migliorando l'affidabilità del sistema e riducendo il tempo necessario per completare le operazioni di analisi su grandi volumi di dati

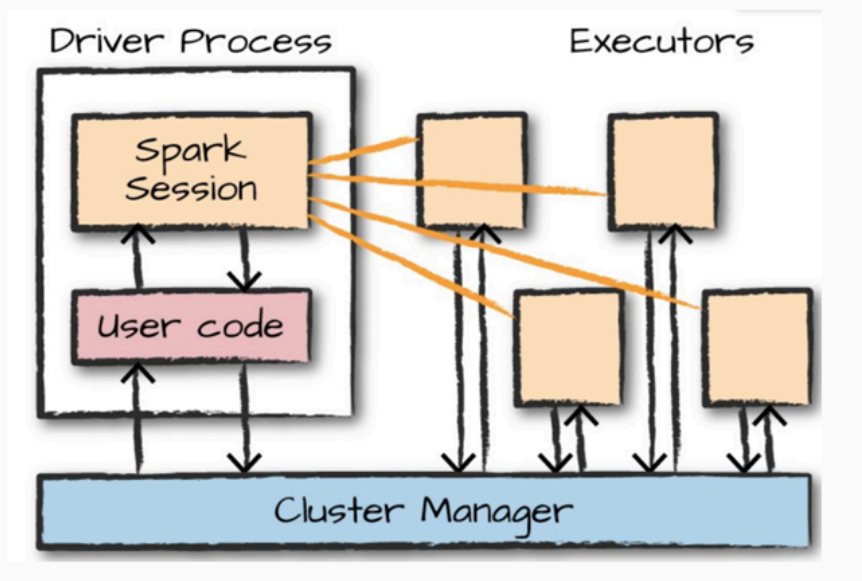
Spark è un sistema di calcolo distribuito che a differenza di MapReduce non si basa esclusivamente su due fasi di elaborazione, ma permette l'esecuzione di operazioni più avanzate. Uno dei principali vantaggi di Spark è la **condivisione veloce dei dati** tra le operazioni, evitando di salvare i risultati intermedi su disco, e l'introduzione del **caching dei dati**, una funzionalità fondamentale per machine learning.

Utilizza un modello basato su **grafi aciclici diretti (DAG)** per definire le dipendenze tra le operazioni, rendendo possibile l'ottimizzazione automatica dell'esecuzione. Inoltre è compatibile con Hadoop, che permette di integrarlo facilmente con ecosistemi esistenti basati su HDFS e YARN, sfruttando le infrastrutture già in uso senza necessità di migrazioni complesse.

Consiste in un framework **open source**, si integra bene con linguaggi come **Java, Scala, Python e R**, rendendolo accessibile a un'ampia comunità di sviluppatori e data scientist. Il concetto chiave su cui si basa Spark è il **Resilient Distributed Dataset (RDD)**, una struttura dati immutabile e distribuita che consente l'elaborazione parallela in modo fault-tolerant. Gli RDD hanno rappresentato la base dell'architettura di Spark sin dalle prime versioni, ma con le evoluzioni del framework come **DataFrame e Dataset**, che forniscono un'interfaccia più ottimizzata e dichiarativa rispetto agli RDD. I **DataFrame** sono tabelle distribuite simili a quelle di un database SQL o di Pandas in Python, mentre i **Dataset** hanno i vantaggi degli RDD. Spark inoltre dispone di un forte supporto a **SQL**, questo permette di eseguire query sui dati in modo familiare, utilizzando operazioni simili a quelle di un database relazionale.

Un **cluster** è una configurazione che consente di risolvere problemi computazionalmente impossibili per una singola macchina, suddividendo il carico di lavoro tra più computer detti **nodi**. Tuttavia, avere semplicemente più macchine non è sufficiente, infatti per gestire l'allocazione dei compiti e coordinare il lavoro tra i nodi è necessario un **framework** che ottimizza l'esecuzione delle operazioni distribuite. Spark gestisce anche questo assegnando i compiti ai nodi in modo bilanciato sfruttando al massimo la capacità del cluster.

Sono tre i punti chiave di Spark, il primo è la **gestione del cluster** responsabile di tracciare e gestire le risorse, potrebbe essere per esempio **YARN**, **Mesos** o il gestore autonomo di Spark. Poi c'è l'**applicazione Spark**, che rappresenta il programma che Spark eseguirà, e si divide in **Driver Process**, che assegna i compiti, e gli **Esecutori**, che sono i nodi del cluster che eseguono i calcoli veri e propri. Ogni esecutore è responsabile di una parte del lavoro e segnala periodicamente il progresso al driver. Infine, Spark offre diverse **modalità di esecuzione**. In **modalità cluster**, le attività vengono distribuite su più macchine, permettendo l'elaborazione parallela di grandi volumi di dati. Invece la **modalità locale** consente di eseguire Spark su una singola macchina, ideale per test, prototipi oppure operazioni piccole.



Spark è principalmente scritto in **Scala**, che offre una sintassi concisa e potente, particolarmente adatta per gestire l'elaborazione distribuita, perfetta per sfruttare al meglio le caratteristiche del framework. Offre supporto anche per **Java**, anche se le performance non sono sempre ottimali come nel caso di Scala. Un altro linguaggio ampiamente utilizzato con Spark è **Python** che offre un supporto completo per tutti i costrutti di Scala, particolarmente popolare tra i data scientist grazie alla sua sintassi semplice e all'ampio ecosistema di librerie di analisi dei dati. Supporta anche **SQL** e questo permette agli utenti di eseguire query direttamente sui dati distribuiti, integrando così la potenza di Spark con la familiarità del linguaggio SQL. Supporta anche **R**.

Il costrutto chiave di Spark è il **Resilient Distributed Dataset (RDD)**, una collezione partizionata di record che possono essere distribuiti su più nodi di un cluster. Forniscono una struttura dati distribuita e immutabile su cui eseguire operazioni di trasformazione e azione. Gli RDD possono generalizzare le **coppie key-value**, vengono comunemente utilizzati per aggregazioni e operazioni su dataset complessi. Gli RDD una volta creati non possono essere modificati per migliorare l'efficienza, tuttavia è possibile eseguire su di essi operazioni di trasformazione che restituiscono nuovi RDD.

Gli **RDD** di Spark offrono diverse tecniche di **partizionamento** per distribuire i dati su più nodi del cluster, ottimizzando così le operazioni di calcolo parallelo. Le tecniche di partizionamento più comuni sono:

1. **HashPartitioner**: le chiavi vengono partizionate in base al loro **hashcode**

2. **RangePartitioner**: ordina i dati sulla base della chiave e suddivide i record nelle partizioni specificate in base a intervalli predefiniti
3. **CustomPartitioner**: consente agli sviluppatori di definire un **partizionamento personalizzato**, che può essere utile per casi d'uso specifici. In questo caso, l'utente ha il controllo completo sulla logica di partizionamento.

Gli **RDD** possono essere creati in due modi: **direttamente da Spark** o tramite **trasformazioni** su altri RDD, permettendo anche di impilare più RDD. Inoltre sono particolarmente adatti per le **applicazioni che applicano la stessa operazione a tutti gli elementi di un set di dati**. Esempi comuni includono operazioni come `map`, `filter` e `flatMap`, dove ogni elemento del dataset viene trattato in modo indipendente dalle altre operazioni, sfruttando appieno il parallelismo.

La **persistenza** e il **caching** sono tecniche utilizzate per memorizzare i dati intermedi in modo da poterli riutilizzare in fasi successive senza doverli ricalcolare.

- **cache()**: memorizza i dati intermedi in **memoria**, è un'opzione veloce ma limitata, quindi è importante valutare attentamente la quantità di dati da memorizzare.
- **persistent()**: offre un controllo più dettagliato rispetto a `cache()`, consentendo di specificare dove memorizzare i dati intermedi. Permette di scegliere tra diverse opzioni di storage, come:
  - **MEMORY_ONLY**: Salva i dati solo in memoria.
  - **MEMORY_AND_DISK**: Memorizza i dati in memoria, ma se la memoria non è sufficiente, li salva anche su disco.
  - **DISK_ONLY**: Memorizza i dati esclusivamente su disco.
  - **MEMORY_ONLY_SER**: Simile a `MEMORY_ONLY`, ma i dati sono memorizzati in formato serializzato, riducendo l'uso della memoria. (Solo Java e Scala)
  - **MEMORY_AND_DISK_SER**: Combinazione di `MEMORY_AND_DISK` con dati serializzati. (Solo Java e Scala)

Le operazioni sugli RDD sono suddivise in:

1. **Trasformazioni**: sono operazioni che producono un nuovo RDD a partire da un RDD esistente. Non modificano direttamente gli RDD originali, ma ne creano di nuovi.
   - **map**: Applica una funzione a ciascun elemento
   - **filter**: Seleziona gli elementi che soddisfano una condizione
   - **join**: Unisce due RDD in base a una chiave comune
   - **union**: Combina due RDD, mantenendo tutti gli elementi da entrambi
   - **intersection**: Restituisce gli elementi comuni tra due RDD
   - **distinct**: Rimuove gli elementi duplicati da un RDD.

   Le trasformazioni sono **lazy**, quindi il calcolo non viene eseguito finché non è necessario. I dati vengono trasformati solo quando viene chiamata un'**azione**, che avvia effettivamente il processo di calcolo.
2. **Azioni**: sono operazioni che restituiscono un valore, forzando il calcolo effettivo degli RDD.
   - **count**: Conta il numero di elementi in un RDD
   - **collect**: Recupera tutti gli elementi di un RDD e li restituisce come una lista
   - **reduce**: Combina gli elementi di un RDD utilizzando una funzione di riduzione

- **save**: Salva i dati in un formato specificato

In Spark, è possibile utilizzare variabili condivise per facilitare la gestione dei dati distribuiti tra i nodi del cluster. Possono essere:

- **Broadcast**: Consentono di distribuire una variabile in sola lettura a tutti i nodi del cluster, memorizzandola in cache su ciascuna macchina.
- **Accumulators**: Variabili utilizzate per tenere traccia di un valore che può essere aggiornato da più nodi, come un contatore condiviso.

Il **Task Scheduler** di Spark supporta DAG, cioè una rappresentazione di come le operazioni vengono eseguite. Ogni nodo nel DAG rappresenta un'operazione di calcolo, mentre le dipendenze tra queste operazioni sono rappresentate dagli archi. Se ci sono operazioni che possono essere eseguite senza dipendere da altre, Spark cerca di eseguire queste operazioni in parallelo, sfruttando il modello di calcolo distribuito tramite pipeline, che ottimizzano l'esecuzione dei task riducendo al minimo le operazioni di shuffle e migliorando il throughput complessivo.

Il task scheduler è **cache-aware**, quindi cerca di riutilizzare i dati memorizzati in cache per evitare di ricalcolarli ogni volta che sono necessari, ed è consapevole della **località dei dati** cercando di eseguire le operazioni il più vicino possibile ai dati stessi riducendo il traffico di rete. Spark è anche **partitioning-aware**, quindi tiene conto della struttura delle partizioni dei dati quando assegna i task. Questo approccio aiuta a evitare **shuffle** migliorando le prestazioni dell'esecuzione.

Esercitazione1 Spark

Funzioni varie: map, flatmap, mapValues, flatMapValues, filter, groupByKey, reduceByKey, join, subtract, sortByKey

Un **DataFrame** in Spark è una collezione distribuita di dati organizzati in colonne con nome, simile a una tabella di un database relazionale. A differenza degli RDD, che non impongono una struttura sui dati, un DataFrame fornisce una rappresentazione più organizzata dei dati, consentendo operazioni come selezioni, filtri e aggregazioni su colonne specifiche.

Consentono una **maggiore astrazione** rispetto agli RDD, semplificando l'analisi e manipolazione dei dati. Ad esempio, un DataFrame potrebbe contenere dati provenienti da una sorgente strutturata come JSON, CSV, o database esterni. Spark SQL, attraverso il **Catalyst optimizer**, ottimizza le operazioni sui DataFrame, riducendo significativamente i tempi di esecuzione delle query. Grazie a queste ottimizzazioni, operazioni come **join** o **groupBy** sui DataFrame sono molto più veloci rispetto a quando vengono eseguite sugli RDD.

1. **Esecuzione Ottimizzata**: Grazie al Catalyst optimizer, Spark è in grado di applicare ottimizzazioni automatiche per migliorare le prestazioni delle operazioni, riducendo il tempo di esecuzione rispetto agli RDD.
2. **Interfacce più ricche**: Offrono una **API di più alto livello** rispetto agli RDD, che consente operazioni come SQL-like query, aggregazioni e manipolazioni di dati con sintassi simile a quella SQL.
3. **Compatibilità con varie sorgenti di dati**: I DataFrame possono essere facilmente creati da una varietà di formati di dati, inclusi Parquet, JSON, CSV e altre sorgenti esterne come i database.

Il **Dataset** in Spark è un'estensione dell'API DataFrame che fornisce una **programmazione orientata agli oggetti** e **type-safe**, ovvero consente di eseguire operazioni sui dati con una maggiore garanzia di sicurezza a livello di tipo, permettendo la rilevazione degli errori al momento della compilazione anziché a runtime. Combinano i punti di forza sia degli RDD che dei DataFrame infatti come gli RDD sono immutabili, ereditano la tipizzazione forte e le operazioni lambda, ma dispongono delle operazioni SQL-like come i Dataframe. Entrambi i tipi di dati (Dataset e DataFrame) sono basati sul motore **Spark SQL**, che ottimizza

automaticamente le operazioni di query. I Dataset sono attualmente disponibili per **Java** e **Scala**, ma non per **Python**.

Librerie utili: SparkSQL, Spark Streaming, Mlib, GraphX

Confrontiamo adesso Spark, Hadoop MapReduce. Spark è generalmente **più veloce** rispetto a Hadoop MapReduce. La principale ragione è che Spark può elaborare i dati in **memoria**, evitando il costo di scrittura e lettura dai dischi durante il processo di map-reduce. Tuttavia, Spark ha bisogno di una **grande quantità di memoria** per funzionare efficientemente. Se non dispone di risorse di memoria sufficienti o se ci sono altri servizi che competono per le risorse, le prestazioni di Spark possono **degradare** significativamente. In questi casi, potrebbe diventare meno performante di MapReduce, che non dipende tanto dalla memoria, ma piuttosto dall'I/O su disco. MapReduce è più adatto a lavori che richiedono **operazioni a passaggio singolo**, per cui è stato originariamente progettato. Quando le operazioni richiedono più passaggi o necessitano di elaborazioni in tempo reale, Spark risulta più vantaggioso. Spark è generalmente **più facile da programmare** rispetto a Hadoop MapReduce. Spark è molto più **generale** nelle sue capacità di elaborazione rispetto a Hadoop MapReduce, che è stato progettato principalmente per operazioni batch di **analisi dei dati**.

Lezione 4 - Similarity Search

Nel contesto dei **Big Data**, trovare elementi simili è un problema cruciale in molte applicazioni, come il rilevamento di documenti duplicati, il clustering di utenti con preferenze simili e la raccomandazione di contenuti. L'obiettivo è identificare elementi simili in modo efficiente, senza dover confrontare ogni coppia possibile, il che sarebbe proibitivo su dataset di grandi dimensioni. L'idea di base è ridurre il numero di confronti effettuando una pre-selezione intelligente delle coppie da analizzare più in dettaglio.

Lo **Shingling** è una tecnica che trasforma i documenti in insiemi di sottostringhe (shingles o k-grammi).

- Un **shingle** di lunghezza k è una sequenza di k caratteri consecutivi estratti da un documento.
- Questa tecnica converte un documento testuale in un insieme, permettendo di calcolare la **similarità di Jaccard** tra due documenti.
- Più alto è il valore della similarità di Jaccard tra due insiemi di shingles, maggiore è la probabilità che i documenti siano simili.

Esempio: Per la frase *"hello world"*, con **shingles di lunghezza 3**, otteniamo:
"hel", "ell", "llo", "lo ", "o w", " wo", "wor", "orl", "rld"

Il **MinHashing** è una tecnica di compressione degli insiemi di shingles che permette di ridurre la loro dimensione senza perdere troppa informazione sulla similarità. Converte un insieme in una firma più compatta (minhash signature), utilizzando funzioni di hashing che garantiscono una distribuzione uniforme. La probabilità che due insiemi abbiano la stessa firma MinHash è approssimativamente uguale alla loro **similarità di Jaccard**. Questo permette di confrontare documenti in modo più efficiente rispetto a un confronto diretto degli shingles.

Il **Locality-Sensitive Hashing (LSH)** è una tecnica che utilizza funzioni di hashing speciali per raggruppare elementi simili in bucket comuni, aumentando la probabilità che due elementi simili vengano confrontati direttamente. LSH riduce drasticamente il numero di confronti richiesti. È particolarmente utile in scenari con grandi dataset, come la rilevazione di spam, il plagiarism detection e la ricerca di immagini simili.

Molti problemi nel mondo reale possono essere riformulati come la **ricerca di insiemi simili**. Le pagine web possono essere confrontate in base alle parole che contengono, permettendo di identificare articoli simili o di rilevare copie di uno stesso documento. Un altro scenario riguarda il comportamento degli utenti nell'e-commerce, infatti se due clienti hanno acquistato prodotti simili è possibile ipotizzare che abbiano preferenze affini e quindi suggerire loro articoli potenzialmente interessanti. La stessa logica si applica alle immagini, dove il confronto avviene attraverso caratteristiche visive piuttosto che tramite testo. Questo approccio è alla base di strumenti di ricerca per immagini simili o per il riconoscimento automatico di oggetti e volti.

Possiamo quindi generalizzare definendo tale problema come individuare gruppi di oggetti simili all'interno di un dataset di grandi dimensioni. Ogni oggetto è descritto da un insieme di caratteristiche (features), e l'obiettivo è identificare somiglianze tra di essi in modo efficiente.

Gli oggetti analizzati possono essere di varia natura: utenti in una piattaforma digitale, pagine web, tweet, prodotti acquistati. Ciò che accomuna questi elementi è il fatto di essere descritti da un insieme di **feature misurabili**, che possono essere di tipo binario (es. "ha comprato un prodotto: sì/no"), categoriale (es. "genere musicale preferito") o numerico (es. "tempo medio trascorso su un sito web").

Per confrontare due oggetti, viene utilizzata una funzione di **similarità** che, prendendo in ingresso i loro insiemi di features, restituisce un valore compreso tra 0 e 1. Un valore vicino a 1 indica una forte somiglianza, mentre un valore vicino a 0 suggerisce una scarsa relazione tra i due oggetti. La scelta della funzione dipende dal tipo di dati trattati: per dati testuali si utilizza spesso la similarità di Jaccard, mentre per dati numerici metriche come la distanza Euclidea o il coseno di similarità risultano più appropriate.

Es. 1 Dati degli utenti U, abbiamo le features che sono dati personali, preferenze, navigazioni ... Vogliamo trovare gli utenti simili a Dave, e individuare i cluster di utenti simili. Qui possiamo applicare la distanza di Hamming.

	Brahma Bull	Spaghetti House	Mango	Il Fornaio	Zao	Ming's	Ramona's	Straits	Homma's
Alice		Yes	No	Yes				No	
Bob		Yes				No		No	
Cindy				Yes	No			No	
Dave	No			No	Yes	Yes			Yes
Estie				No	Yes	Yes		Yes	
Fred	No						No		

La **distanza di Hamming** è una metrica utilizzata per misurare la differenza tra due sequenze di uguale lunghezza. Si calcola contando il numero di posizioni in cui i simboli corrispondenti nelle due sequenze sono diversi. Questa misura è particolarmente utile in ambiti come il riconoscimento di errori nei codici, la biologia computazionale e il confronto di dati binari.

Un esempio tipico si ha con stringhe binarie. Supponiamo di avere due sequenze:

- **A = 1011101**
- **B = 1001001**

Poiché le differenze sono in due posizioni, la distanza di Hamming tra queste stringhe è **2**. Tale approccio può essere usato nel contesto delle features caratterizzate da valori booleani come l'esempio appena visto.

Es. 2 Dato un oggetto grafo $G(V, E)$ abbiamo le features che sono le liste di adiacenza sparse. L'obiettivo è trovare nodi simili a un nodo specifico, e individuare cluster di nodi simili. Qui possiamo usare il coefficiente di Jaccard, ovvero:

$$JaccardCoeff = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|}$$

- $N(u)$ è l'insieme dei nodi adiacenti a u (i suoi vicini).
- $N(v)$ è l'insieme dei nodi adiacenti a v (i suoi vicini).
- $|N(u) \cap N(v)|$ è il numero di vicini **comuni** tra u e v.
- $|N(u) \cup N(v)|$ è il numero totale di vicini **unici** considerando entrambi i nodi.

Il coefficiente di Jaccard misura **quanto i due nodi condividono gli stessi vicini** rispetto al totale dei loro vicini distinti.

- Se $JaccardCoeff = 1$, significa che u e v hanno **gli stessi vicini** e quindi sono molto simili.
- Se $JaccardCoeff = 0$, significa che u e v **non condividono alcun vicino** e quindi sono completamente diversi.
- Valori intermedi indicano un certo grado di somiglianza.

Es. 3 Immaginiamo di avere dei documenti, utilizziamo la rappresentazione in spazio vettoriale per la rappresentazione dei documenti. Il peso di ciascun vettore viene rappresentato dal punteggio TF-IDF che misura l'importanza di un termine nel documento in relazione alla sua frequenza in tutti i documenti. Per calcolare i cluster di documenti simili possiamo utilizzare la Cosine Similarity, facendo il prodotto vettoriale tra due documenti.

LSH utilizza un **approccio di hashing** per ridurre il numero di confronti necessari. Abbiamo delle funziomappano i vettori ad alta dimensione in codici di hash a bassa dimensione, o bucket. Ogni funzione di hash è progettata per fare in modo che gli **oggetti simili** abbiano una probabilità maggiore di essere mappati nello stesso bucket. Questo riduce drasticamente il numero di confronti da fare ed esamina solo le coppie che finiscono nello **stesso bucket** per almeno una delle funzioni di hash. Non è una singola funzione, ma una **famiglia di tecniche correlate** che possono essere adattate a diversi tipi di misure di similarità (ad esempio, similarità del coseno, distanza di Hamming, similarità di Jaccard, ecc.).

Un aspetto importante da notare è che LSH offre una ricerca di **vicini più prossimi approssimativa**. Ciò significa che l'algoritmo potrebbe non trovare sempre i vicini più prossimi **esatti**, ma troverà quelli "abbastanza vicini", il che è spesso sufficiente nelle applicazioni pratiche. Per esempio, supponiamo di avere un dataset di documenti di testo rappresentati come vettori in uno spazio ad alta dimensione (ad esempio, vettori TF-IDF). Si desidera trovare documenti simili a una query data.

Servizi come Pinecone offrono database vettoriali gestiti che sfruttano tecniche come LSH per la ricerca della similarità. Questi database semplificano l'interazione con i dati vettoriali ad alta dimensione, permettendo ricerche rapide ed efficienti sulla similarità e sul clustering.

Nel problema della ricerca di punti simili in spazi ad alta dimensionalità l'obiettivo è trovare coppie di punti dati $x_1, x_2, \ldots$ che siano **sufficientemente vicini** secondo una data funzione di distanza $d(x_i, x_j)$. Questo problema è molto comune in applicazioni come il riconoscimento delle immagini, la ricerca di documenti simili e i sistemi di raccomandazione.

Un'immagine può essere rappresentata come un vettore lungo, in cui ogni valore corrisponde al colore di un pixel. Ad esempio, prendiamo questa immagine 3x3:

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

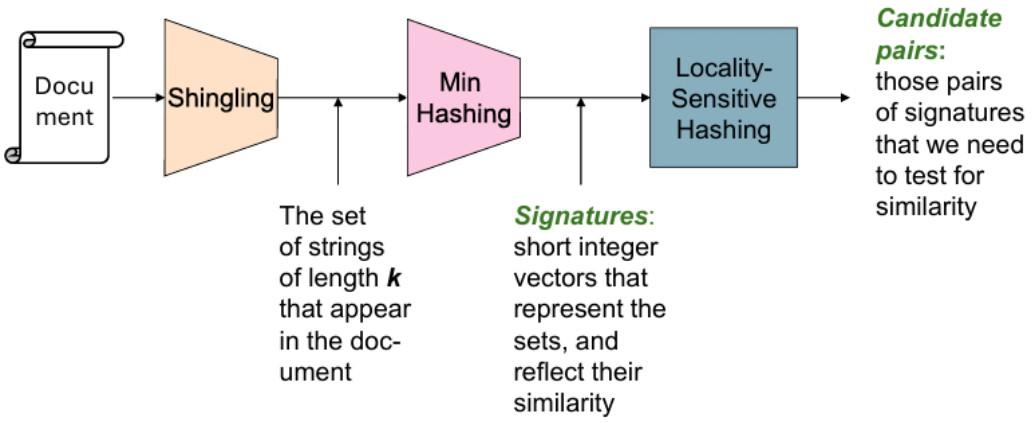
Possiamo appiattirla in un vettore:

$$[1, 2, 1, 0, 2, 1, 0, 1, 0]$$

Questa trasformazione è comune quando si lavora con reti neurali o tecniche di clustering basate su distanze. Un approccio ingenuo confronterebbe **tutte le coppie di punti**, con una complessità di $O(N^2)$, dove N è il numero di punti. Questo è proibitivo per dataset di grandi dimensioni. La magia sta nell'usare **Locality-Sensitive Hashing (LSH)** o altre tecniche di indicizzazione avanzate.

Ci sono tre passi essenziali per documenti simili:

- **Shingling:** Il primo passo consiste nel trasformare ogni documento in un **insieme di sottostringhe**, chiamate **shingles**. Un **shingle** è una sequenza di k parole consecutive o caratteri estratti dal testo. Questo passaggio permette di rappresentare il documento come un **insieme di elementi confrontabili**, piuttosto che come un'unica stringa.
- **Min-Hashing:** permette di ridurre la dimensione dei dati preservando la similarità. Il problema con gli shingles è che un documento diventa un insieme molto grande, difficile da confrontare direttamente. Si applicano **funzioni hash** agli shingles per ottenere una **firma compatta** del documento. Questo riduce drasticamente lo spazio necessario per archiviare i documenti e rende il confronto molto più veloce.
- **Locality-Sensitive Hashing:** Anche con firme compatte, confrontare tutte le coppie sarebbe ancora inefficiente. **LSH** serve per concentrarsi solo sulle coppie **potenzialmente simili**. Si suddivide la firma MinHash in **bande**, ognuna delle quali viene **hashata in un bucket**. Se due documenti finiscono nello stesso bucket in almeno una banda, vengono considerati **candidati per il confronto**. Questo riduce enormemente il numero di confronti, passando da un approccio $O(N^2)$ a una ricerca molto più veloce.



Uno shingle viene detto k-shingle ovvero una sequenza di k tokens che appaiono nel documento, possono essere caratteri o parole. Per esempio se $D_1 = abcab$ allora 2-shingle sono $S(D_1) = \{ab, bc, ca\}$ dove stiamo andando a considerare solo il set e non i pezzi ripetuti. Eventualmente possiamo estenderla considerando anche quelli, a seconda dei casi d'uso. Sostanzialmente documenti che hanno molti shingles in comune sono simili anche se appaiono in ordini diversi. Dobbiamo ovviamente scegliere un k sufficientemente grande altrimenti molti documenti saranno classificati come simili anche se non lo sono. Solitamente per documenti piccoli $k = 5$, mentre per documenti lunghi $k = 10$. Gli shingle vanno compressi e possiamo usare hash a 4 bytes. Possiamo quindi rappresentare il documento con il set degli hash values dei suoi k-shingles.

Per quantificare la somiglianza tra due documenti D_1 e D_2, si utilizza la **similarità di Jaccard**:

$$sim(D_1, D_2) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|}$$

Dove:

- $|C_1 \cap C_2|$ è il numero di shingles comuni tra i due documenti.
- $|C_1 \cup C_2|$ è il numero totale di shingles distinti presenti in entrambi.

Perchè usare Minhash/LSH? Su un milione di documenti dovremmo effettuare il confronto di similarità di Jaccard per ogni coppia e ci metteremmo anni. Quindi convertiamo con MinHashing in un set di firme corte preservando la similarità.

Molti problemi di similarità possono essere visti come il confronto tra **sottoinsiemi** di un insieme più grande, misurando quanto si sovrappongono. Un modo efficiente per rappresentare e confrontare questi insiemi è usare **vettori binari (0/1)**. Se un insieme contiene un determinato elemento, il valore in quella posizione è **1**, altrimenti è **0**.

Esempio: $C_1 = (1, 0, 1, 1, 1), \quad C_2 = (1, 0, 0, 1, 1)$

La **similarità di Jaccard** si calcola come:

$$J(C_1, C_2) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|}$$

Interpretando operazioni sugli insiemi come operazioni bitwise:

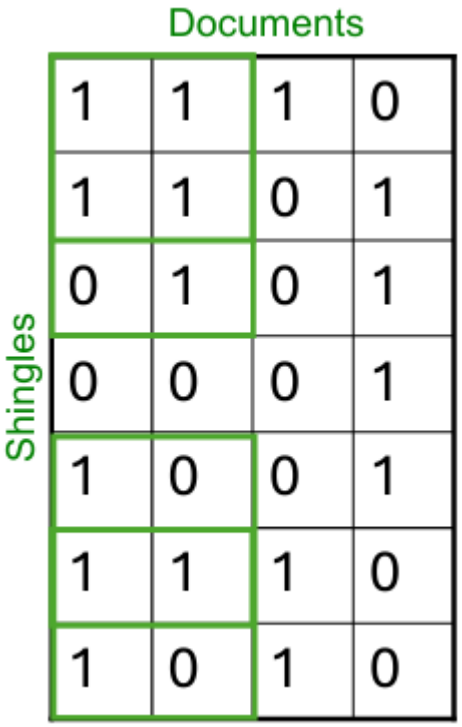
- **Intersezione** $C_1 \cap C_2 \rightarrow$ Bitwise AND (&):
  $(1, 0, 1, 1, 1) \& (1, 0, 0, 1, 1) = (1, 0, 0, 1, 1) \quad (3 \text{ elementi a } 1)$
- **Unione** $C_1 \cup C_2 \rightarrow$ Bitwise OR (|): $(1, 0, 1, 1, 1) | (1, 0, 0, 1, 1) = (1, 0, 1, 1, 1) \quad (4 \text{ elementi a } 1)$
- **Calcolo della Similarità**: $J(C_1, C_2) = \frac{3}{4} = 0.75$

Ne segue che la distanza è $d(C_1, C_2) = 1 - 0.75 = 0.25$.

Per confrontare documenti sulla base dei loro **shingles**, possiamo rappresentarli in una **matrice binaria sparsa**, dove:

- Le righe rappresentano gli elementi (shingles).
- Le colonne rappresentano gli insiemi (documenti).
- L'elemento della matrice $M_{e,s} = 1$ se e solo se lo shingle e è presente nel documento s.

Questa struttura è utile perché permette di applicare efficientemente la **similarità di Jaccard** tra documenti.



Ci chiediamo per esempio $sim(C_1, C_2)$, troviamo che l'intersezione è 3, l'unione è 6, quindi Jaccard Similarity è 3/6 e la distanza è 3/6. Quindi finora abbiamo capito come trovare la similarità tra documenti, il prossimo passo è farlo confrontando signature più piccole in quanto può essere vista come la similarity delle signatures. Confrontare tutte le coppie potrebbe essere un approccio non ottimale, quindi meglio usare LSH.

L'idea è quella di hashare ogni colonna in una signature piccola $h(C)$ così che possa entrare in RAM, calcolare la similarity sulle colonne o sulle signature hashate è la stessa cosa. Quindi l'obiettivo è trovre una funzione hash così che se $sim(C_1, C_2)$ è alto, allora c'è alta probabilità che $h(C_1) = h(C_2)$. Viceversa se sim è bassa allora le due hash probabilmente saranno diverse, cioè finiscono in bucket diversi.

L'idea è di applicare una **permutazione casuale** delle righe della **matrice booleana** e poi assegnare a ogni colonna (documento) una firma compatta, detta **MinHash signature**.

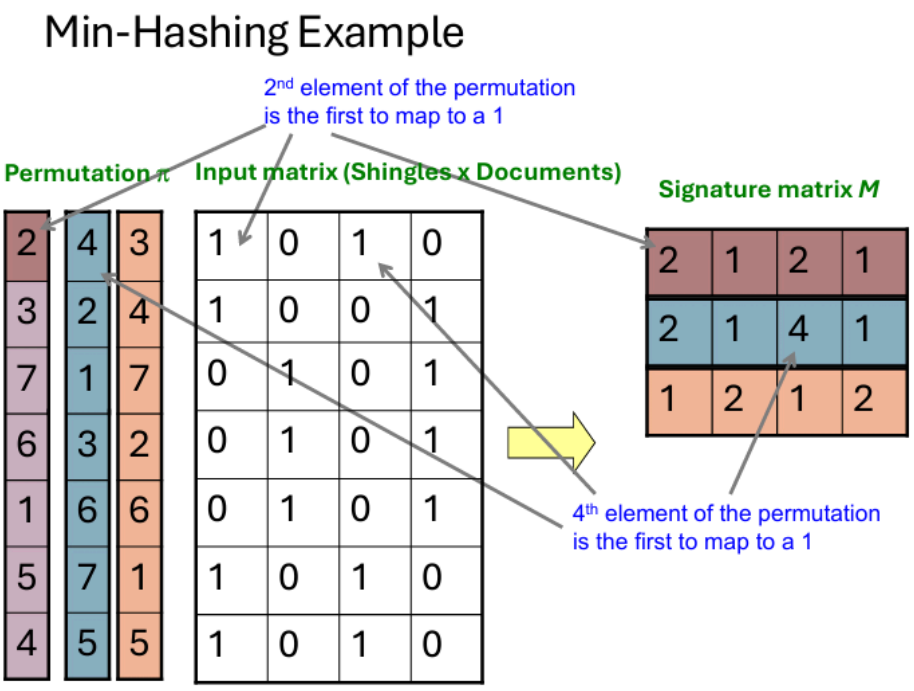
Sia π una permutazione casuale delle righe della matrice. Definiamo la funzione di hash associata alla permutazione come:

$$h_\pi(C) = \min_\pi \pi(C)$$

Dove $h_\pi(C)$ restituisce **l'indice della prima riga (secondo la permutazione π) in cui la colonna C ha valore 1**. Per ogni documento, ripetiamo il processo più volte con **diverse permutazioni casuali** e otteniamo una **firma MinHash** fatta di più valori di $h_\pi(C)$. La **probabilità che due documenti abbiano lo stesso valore di MinHash** per una permutazione è approssimativamente uguale alla loro **similarità di Jaccard**.

$$P(h_\pi(C_1) = h_\pi(C_2)) \approx J(C_1, C_2)$$

Quindi invece di calcolare la Jaccard Similarity possiamo stimarla confrontando quante volte le firme MinHash coincidono.



> PARTI DA 1 NELLA PERMUTAZIONE E LEGGI LA RIGA CORRISPONDENTE, NON DALL'ALTO

Parliamo adesso della proprietà del MinHashing, quindi fissata una permutazione allora

$$P[h_\pi(C_1) = h_\pi(C_2)] = sim(C_1, C_2)$$

Infatti, sia X un documento (quindi un set di shingles) e sia $y \in X$ una shingle del documento. Allora:

$$P[\pi(y) = min(\pi(X))] = \frac{1}{|X|}$$

Sia y tale che $\pi(y) = min(\pi(C_1 \cup C_2))$ allora sono due le possibilità

1. $\pi(y) = min(\pi(C_1))$ se $y \in C_1$
2. $\pi(y) = min(\pi(C_2))$ se $y \in C_1$

Quindi la probabilità che entrambe siano vere cioè che $y \in C_2 \cap C_2$ è:

$$P[min(\pi(C_1)) = min(\pi(C_2))] = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|} = sim(C_1, C_2)$$

Quindi sostanzialmente la probabilità che $h_\pi(C_1) = h_\pi(C_2)$ è righe con entrambe le colonne 1 fratto le righe con almeno una delle colonne 1, cioè la similarity. L'intersezione e l'unione della formula sopra operano rispettivamente come AND e OR Bitwise.

Computazionalmente parlando calcolare la similarità sugli hashing è meglio, infatti la similarità tra $Sign(C_1)$ e $Sign(C_2)$ è:

$$Sim(Sign(C_1), Sign(C_2)) = \frac{righe\ con\ h_\pi(C_1) = h_\pi(C_2)}{numero\ tot.\ righe} = P[h_\pi(C_1) = h_\pi(C_2)] = Sim(C_1, C_2)$$

Correttezza del Min-Hashing - Skip

Sappiamo che la probabilità che due colonne abbiano la stessa funzione hash corrisponde alla loro similarità. Questo concetto può essere esteso utilizzando più funzioni di hash: in questo caso, la similarità tra due firme viene calcolata come la frazione di funzioni di hash per cui esse coincidono.

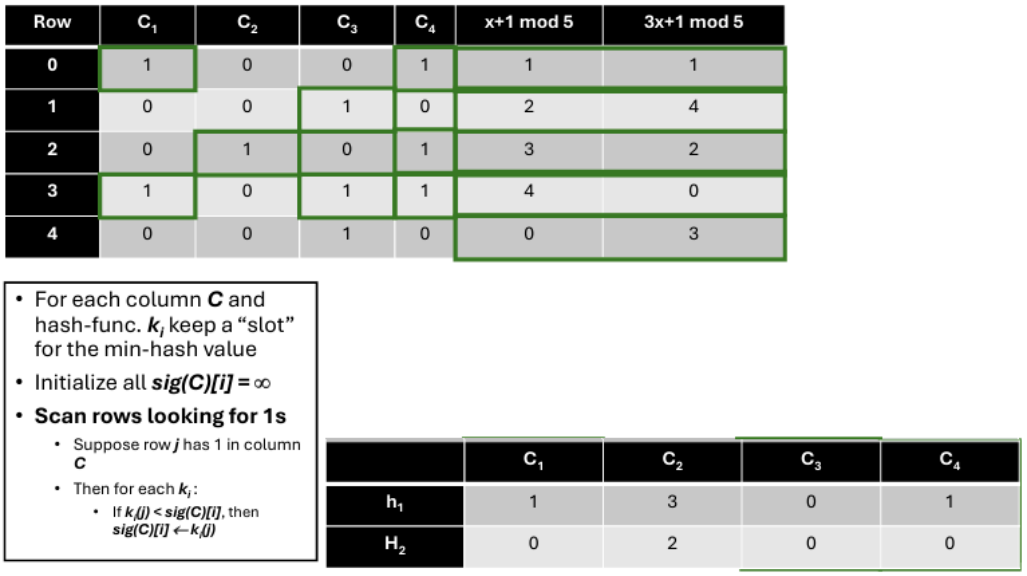
Tornando all'esempio dell'immagine precedente possiamo vedere la differenza tra la similarity tra colonne e tra signatures e noteremo che seppur leggermente diversi comunque si avvicinano l'un l'altro. Questo effetto credo che sia mitigato se abbiamo a che fare con molto testo.

	1 e 3	2 e 4	1 e 2	3 e 4
Col/Col	0.75	0.75	0	0
Sig/Sig	0.67	1.00	0	0

Ricorda che la prima si calcola con AND fratto OR, mentre la seconda l'abbiamo appena vista.

Prendiamo per esempio $K = 100$ permutazioni random delle righe per avere stime più esatte. Applichiamo l'hash function ad ogni permutazione e la firma del documento sarà un vettore di K valori. Formalmente $Sig(C)[i] = min(\pi_i(C))$. Nota bene che la signature del documento è solo 100 bytes se abbiamo hash a 4 byte quindi piccola, e abbiamo compresso un lungo vettore di bit in una signature più gestibile!

Trick: permutare le righe in realtà non è così immediato, possiamo usare anche qui l'hashing.



Row	C_1	C_2	C_3	C_4	x+1 mod 5	3x+1 mod 5
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

	C_1	C_2	C_3	C_4
h_1	1	3	0	1
H_2	0	2	0	0

Lezione 5 - Locality Sensitive Hashing

La Locality Sensitive Hashing (LSH) si basa sull'idea fondamentale di generare un *sketch* per ogni oggetto soddisfi due proprietà principali:

1. Deve essere molto più corto rispetto al numero totale di caratteristiche (d) dell'oggetto originale
2. La similarità tra due vettori di caratteristiche viene trasformata in una relazione di uguaglianza tra i loro *sketch*. In altre parole, se due oggetti sono simili nello spazio originale, avranno una probabilità elevata di essere rappresentati dallo stesso *sketch* dopo l'hashing

Si basa sulla randomizzazione, il che significa che non garantisce risultati esatti, ma è corretto con alta probabilità. Questo approccio probabilistico è spesso l'unica soluzione praticabile quando si lavora con dataset di grandi dimensioni, dove un confronto esatto sarebbe troppo costoso in termini di tempo e risorse.

Un altro vantaggio chiave è la capacità di garantire un accesso locale ai dati, rendendolo particolarmente efficiente in ambienti distribuiti o quando i dati vengono letti da disco. Questo aspetto è cruciale per migliorare le prestazioni in sistemi Big Data, dove la latenza dell'accesso ai dati può rappresentare un collo di bottiglia significativo.

Nel caso della *Hamming distance*, consideriamo vettori p e q composti da features binarie. La distanza di Hamming tra questi due vettori, indicata come $D(p, q)$, è definita come il numero di posizioni in cui i due vettori differiscono.

Per applicare la *Locality Sensitive Hashing*, definiamo una funzione di hash h che seleziona un sottoinsieme I con r coordinate casuali dal vettore originale. L'output della funzione di hash è la proiezione del vettore originale sulle coordinate scelte. Ad esempio, se scegliamo $r = 2$ e il sottoinsieme di indici è $I = \{1, 4\}$, allora per un vettore $p = 01011$, la sua versione compressa sarà data solo dai valori nelle posizioni 1 e 4, ovvero $h(p) = 01$. Questo approccio consente di ridurre la dimensionalità del problema poiché quelli con una distanza di Hamming ridotta avranno una maggiore probabilità di condividere lo stesso valore di hash.

Una proprietà chiave della Locality Sensitive Hashing nel caso della distanza di Hamming è la probabilità che due vettori p e q abbiano lo stesso valore di hash in base alla selezione casuale delle coordinate.

Questa probabilità è data da:

$$Pr[\text{picking } x : p[x] = q[x]] = \frac{d - D(p, q)}{d}$$

Dove d è la dimensione totale dei vettori binari, $D(p, q)$ è la distanza di Hamming.

Questa formula indica che la probabilità di selezionare una coordinata in cui p e q coincidono dipende direttamente dalla loro similarità: maggiore è il numero di bit uguali tra i due vettori, maggiore sarà la probabilità che la funzione di hash li assegni allo stesso *bucket*.

Possiamo quindi definire:

$$P[h(p) = h(q)] = \left(1 - \frac{D(p, q)}{d}\right)^r = S^r$$

Dove S^r è la similarity tra i due vettori. Possiamo controllare questa probabilità variando il numero di coordinate selezionate, cioè r. Se r è piccolo, la probabilità che due vettori simili vengano mappati allo stesso valore di hash è più alta. Se r è grande, la discriminazione aumenta e i vettori con anche piccole differenze saranno più facilmente separati e possiamo ridurre i falsi positivi.

Per ridurre i **falsi negativi** nella LSH possiamo ripetere il processo di hashing più volte utilizzando il concetto di *banding*. L'idea è di eseguire il hashing b volte, ognuna con una proiezione indipendente del vettore.

Si ripetono b volte le proiezioni di hash $h_i(p)$, ognuna scegliendo r coordinate casuali.

1. Si costruisce una funzione aggregata $g(p) = \langle h_1(p), h_2(p), \ldots, h_b(p) \rangle$.
2. Due vettori p e q vengono considerati *matching* se almeno una delle funzioni di hash h_i produce lo stesso valore per entrambi.

Per esempio, impostiamo $r = 2$ e $b = 3$, e consideriamo due vettori:

- $p = 01001$
- $q = 01101$

Definiamo tre gruppi di coordinate:

- $I_1 = \{3, 4\} \rightarrow h_1(p) = 00, h_1(q) = 10$
- $I_2 = \{1, 3\} \rightarrow h_2(p) = 00, h_2(q) = 01$
- $I_3 = \{1, 5\} \rightarrow h_3(p) = 01, h_3(q) = 01$

Dal risultato si nota che $h_3(p) = h_3(q)$, quindi almeno una funzione di hash ha prodotto lo stesso valore. Ciò significa che p e q vengono considerati simili.

Se aumentassimo b riduciamo la probabilità di falsi negativi. Questo avviene perché, anche se una funzione di hash fallisce nel riconoscere la similarità, altre potrebbero riuscirci.

La funzione aggregata $g(p)$ è costituita da b funzioni di hash indipendenti h_i. Per calcolare la probabilità che due vettori p e q **non siano considerati simili** (ossia che nessuna delle funzioni di hash produca lo stesso valore), si procede come segue:

$$Pr[p \text{ non-simile } q] = Pr[h_i(p) \neq h_i(q), \forall i = 1, \ldots, b]$$

Dato che gli hash sono indipendenti, la probabilità di fallire in tutti i b tentativi è data dal prodotto delle singole probabilità di fallimento:

$$Pr[p \text{ non-simile } q] = (Pr[h_i(p) \neq h_i(q)])^b = (1 - Pr[h_i(p) = h_i(q)])^b$$

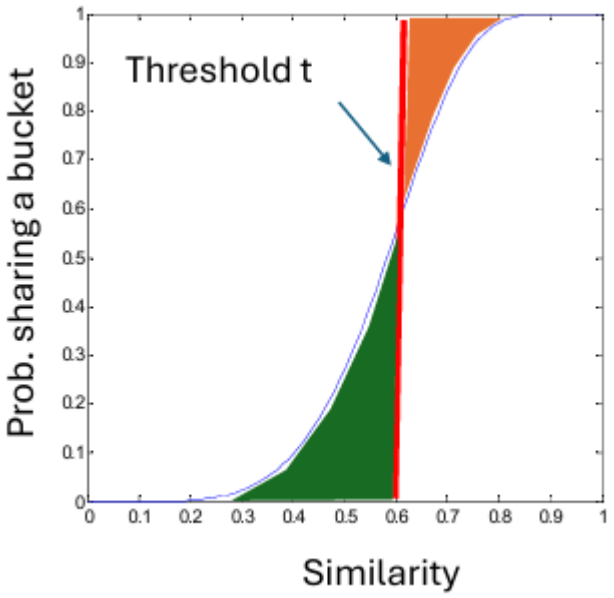
La funzione probabilistica che descrive la probabilità che due vettori p e q siano considerati simili in LSH segue una **S-curve**, data dalla formula:

$$Pr[\text{p simile a q}] = 1 - (1 - s^r)^b$$

Dove:

- $(1 - s^r)$ è la probabilità che una singola band **fallisca** nel riconoscere p e q come simili.
- $(1 - s^r)^b$ è la probabilità che **tutte** le b bands falliscano nel riconoscere la similarità.
- Sottraendo il risultato da 1, otteniamo la probabilità complessiva che **almeno una** delle bands riconosca p e q come simili.

Quando s è molto alto (cioè i vettori sono molto simili), la probabilità di riconoscerli come tali è molto alta. Viceversa, quando s è molto basso la probabilità di considerarli simili si riduce drasticamente. L'andamento della funzione crea una curva a "S", mostrando una transizione netta tra vettori considerati simili e vettori considerati diversi. Nella situazione ideale, vorremmo che una volta raggiunta la threshold si consideri automaticamente la similarità. In casi reali, invece, abbiamo anche falsi negativi e falsi positivi, identificati nel grafico dalle aree arancioni e verdi.



L'area arancione sono i falsi negativi, mentre quella verde sono i falsi positivi. La linea rossa è la threshold.

Demo: https://www.desmos.com/calculator/lzzvfjiujn

Es.1 Immaginiamo di avere:

- **100.000 documenti**, rappresentati come **100.000 colonne** di una matrice M.
- **Firma MinHash** di **lunghezza 100** (100 righe per ogni colonna).
- **Soglia di similarità desiderata**: $s = 0.8$.
- Parametri LSH: $b = 20$, $r = 5$

Immaginiamo che le colonne C_1, C_2 abbiano similarità 0.8, significa che vogliamo che vengano classificate come simili, quindi almeno una band dovrebbe essere uguale.

$$Pr[match\ in\ una\ band\ particolare] = s^r = 0.8^5 = 0.328$$

Quindi:

$$Pr[nessun\ match\ in\ 20\ band] = (1 - 0.328)^{20} = 0.00035$$

Elementi con similarità almeno 0.8 saranno classificati come simili nel 99.965% dei casi.

Es.2 Prendiamo lo stesso esempio ma considerando che le due colonne abbiano in effetti similarità 0.3, pertanto non le vorremmo considerare coppie candidate. In questo caso si ha:

$$Pr[match\ in\ una\ band\ particolare] = s^r = 0.3^5 = 0.00243$$

Ne segue che

$$Pr[nessun\ match\ in\ almeno\ una\ band] = 1 - (1 - 0.00243)^{20} = 0.0474$$

Quindi avremo comunque il 4,75% di possibilità di classificarli lo stesso come coppie candidate, quindi falsi positivi.

Vediamo adesso l'algoritmo generale per la LHS.

1. **Calcolo delle firme hash**: per ogni vettore di features p, si calcola la sua **firma di hashing** basata sulle b *bands*:

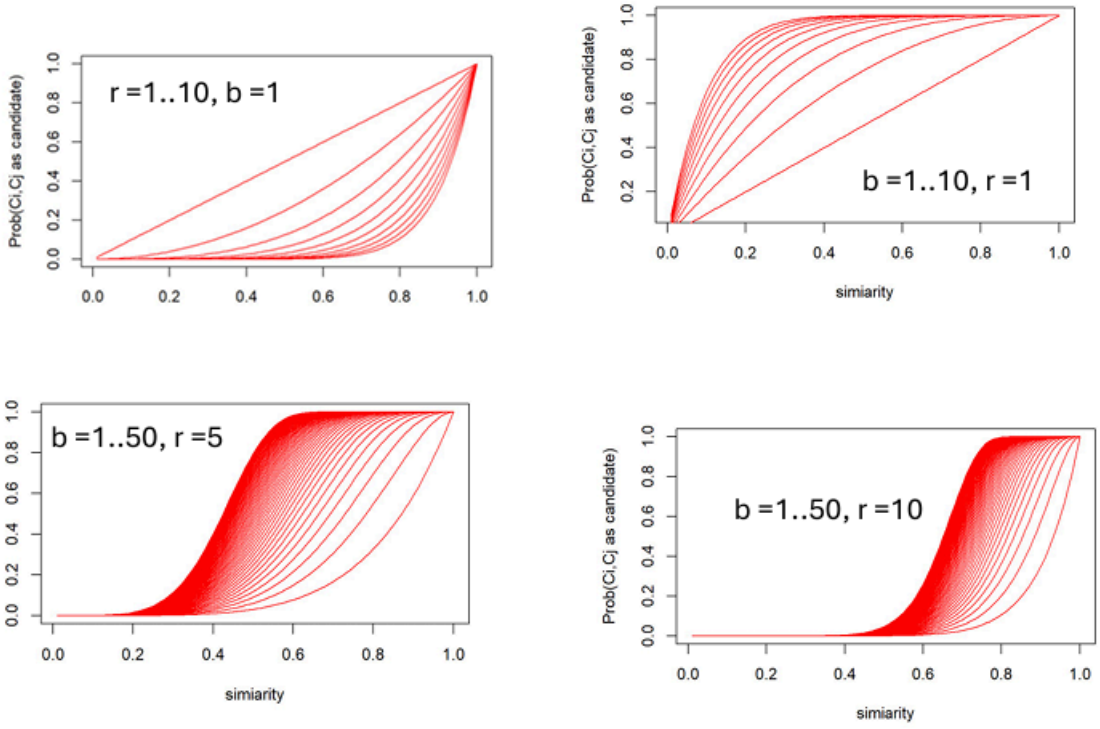
$$g(p) = \langle h_1(p), h_2(p), \ldots, h_b(p) \rangle$$

dove ogni $h_i(p)$ rappresenta l'hash del vettore sulla i-esima band. Si dice $Sketch(p)$.

2. **Creazione dei clustering** C_i: per ogni **band** i ($i = 1, 2, \ldots, b$) si creano **cluster** raggruppando insieme i vettori p e q che hanno almeno uno stesso valore hash $h_i(p) = h_i(q)$. Si dice $Sort$.
3. **Costruzione del grafo dei candidati**: si crea un **grafo non orientato**, in cui ogni vettore p è un nodo. Si crea un collegamento tra due nodi p e q **se almeno una band li ha raggruppati insieme** in un cluster C_i .
4. **Individuazione delle componenti connesse**: il grafo viene analizzato per trovare **componenti connesse**, che rappresentano gruppi di vettori considerati simili.

Lezione 6 - LSH Tuning

Il prossimo step è ottimizzare LHS, quindi il passaggio di LHS Tuning. Vogliamo impostare M, b, r così da considerare tutte le coppie con firme simili (riducendo i falsi negativi) ed eliminare la maggior parte delle coppie che non sono simili (riducendo i falsi positivi). Ricordiamo che b è il numero di funzioni hash applicate (ridurre falsi negativi), r il numero di features incluse nella proiezione (ridurre falsi positivi), M la matrice.



Le funzioni locality-sensitive (LSH) sono un insieme di funzioni progettate per distinguere coppie di elementi vicini da quelle lontane in uno spazio di similarità, con l'obiettivo di ridurre il costo computazionale della ricerca di elementi simili.

Oltre alla MinHash vista in precedenza, esistono altre famiglie di funzioni LSH che soddisfano tre condizioni fondamentali:

1. **Priorità ai vicini**: le coppie di elementi con una distanza ridotta devono avere una probabilità più alta di essere considerate simili rispetto a coppie con distanza elevata.

2. **Indipendenza statistica**: è necessario stimare la probabilità che due o più funzioni locality-sensitive restituiscano la stessa risposta per una coppia di elementi, garantendo che i risultati non siano casuali ma statisticamente significativi.
3. **Efficienza computazionale**: l'identificazione delle coppie candidate deve avvenire in un tempo molto inferiore rispetto a quello richiesto per calcolare tutte le distanze tra le coppie possibili. Le funzioni devono essere combinabili riducendo sia i falsi positivi che i falsi negativi.

Questi principi permettono di progettare algoritmi efficienti per la ricerca approssimata di elementi simili in grandi dataset.

Una funzione locality-sensitive è una funzione che prende in input due elementi e decide se questi devono essere considerati una **coppia candidata**

- Se $f(x) = f(y) \rightarrow x$ e y sono una **coppia candidata**.
- Se $f(x) \neq f(y) \rightarrow x$ e y **non** sono una coppia candidata.

Un insieme di queste funzioni viene chiamato **famiglia di funzioni locality-sensitive**.

Si dice che una famiglia di funzioni F è (d_1, d_2, p_1, p_2)-sensibile se, per ogni funzione f appartenente a F:

- Se $d(x, y) \leq d_1$, allora la probabilità che $f(x) = f(y)$ è almeno p_1 (alta probabilità candidati)
- Se $d(x, y) \geq d_2$, allora la probabilità che $f(x) = f(y)$ è al massimo p_2 (bassa probabilità candidati)

Nel caso della LHS con MinHash abbiamo:

- **S** = insieme di tutti i possibili insiemi.
- **d** = distanza di Jaccard, definita come $d(x, y) = 1 - J(x, y)$, dove $J(x, y)$ è la similarità di Jaccard.
- **H** = famiglia di funzioni di Min-Hash generate da tutte le possibili permutazioni delle righe

Per ogni funzione di hash h appartenente alla famiglia H:

$$Pr[h(x) = h(y)] = 1 - d(x, y)$$

Pertanto meno sono distanti più è probabile che siano considerati candidati.

Quando si utilizza la **distanza di Jaccard**, la famiglia di funzioni di MinHash è $(d_1, d_2, 1 - d_1, 1 - d_2)-$ sensibile, il che significa che la probabilità di collisione tra due elementi dipende direttamente dalla loro similarità di Jaccard, e questo con $0 < d_1 < d_2 < 1$. Per altre distanze non c'è garanzia che sia locality-sensitive.

La famiglia MinHash **H** è una famiglia $(\frac{1}{3}, \frac{2}{3}, \frac{2}{3}, \frac{1}{3})-$sensibile per l'insieme **S** e la distanza di Jaccard **d**.
Ciò significa che se la distanza di Jaccard tra due insiemi è $< \frac{1}{3}$, la probabilità che abbiano lo stesso valore di MinHash è $> \frac{2}{3}$, altrimenti se la distanza di Jaccard tra due insiemi è$> \frac{2}{3}$, la probabilità che abbiano lo stesso valore di MinHash è $< \frac{1}{3}$.

L'effetto **"S-Curve"**, osservato nel caso di **MinHashing con la tecnica delle bande**, può essere generalizzato a qualsiasi famiglia di funzioni **Locality-Sensitive**. La tecnica delle bande utilizzata per le **matrici delle firme** si applica anche in contesti più generali. Infatti, è possibile eseguire **Locality-Sensitive Hashing (LSH)** con qualsiasi famiglia $(d_1, d_2, 1 - d_1, 1 - d_2)-$sensibile.

In effetti possiamo applicare due metodologie:

- **AND-Construction:** costruiamo una nuova famiglia di funzioni F' dove ogni funzione f è costruita da r membri di F per qualche r fissato. Si ha che $f(x) = f(y)$ se e solo se le funzioni in famiglia coincidono tutte! Questo abbassa la probabilità dei falsi positivi ma aumenta i falsi negativi, pertanto stiamo in generale abbassando tutte le probabilità. La nuova famiglia F' sarà $(d_1, d_2, p_1^r, p_2^r)-$sensibile.

  La probabilità che due funzioni hash restituiscano lo stesso valore per una coppia di elementi è approssimativamente il **prodotto** delle probabilità individuali. Tuttavia, due funzioni hash specifiche potrebbero essere **altamente correlate**. Nonostante questa possibile correlazione le probabilità di successo sono calcolate considerando **tutte le possibili funzioni hash nella famiglia** H quindi nel caso medio funziona.
- **OR-Construction:** costruiamo una nuova famiglia di funzioni F' dove ogni funzione f è costituita da b membri di F per un certo b fissato. Si ha che $f(x) = f(y)$ se anche solo una funzione coincide! La nuova famiglia sarà $(d_1, d_2, 1-(1-p_1)^r, 1-(1-p_2)^r)-$sensibile. Questo rende l'algoritmo più permissivo, stiamo riducendo i falsi negativi ma rischiamo di aumentare i falsi positivi (ovviamente, stiamo in qualche modo aumentando tutte le probabilità)

Le costruzioni **AND** e **OR** in LSH possono essere combinate in sequenza per ottenere un miglioramento significativo nel controllo delle probabilità di selezione. Questo approccio consente di avvicinarsi a una distinzione perfetta tra coppie simili e non simili. Possiamo ridurre drasticamente la probabilità di selezionare coppie non simili (falsi positivi) rendendo le probabilità basse praticamente nulle, ed aumentare la probabilità di selezionare coppie simili (veri positivi), facendo in modo che le probabilità siano molto vicine a 1.

Es. $(0.2, 0.8, 0.8, 0.2)-$sensitive, con $b = 4$ e $r = 4$.

Questa è una configurazione iniziale della famiglia LSH, la probabilità che due elementi simili vengano selezionati come candidati è **0.8**, mentre la probabilità che due elementi non simili vengano selezionati è **0.2**.

- **AND-OR**: la probabilità per le **coppie simili** dopo l'operazione **AND** si riduce.

  $$P_{\text{simili AND}} = 0.8 \times 0.8 = 0.64$$

  La probabilità per le **coppie non simili** dopo l'operazione **AND** si riduce anch'essa:

  $$P_{\text{non simili AND}} = 0.2 \times 0.2 = 0.04 **OR**$$

  Adesso applichiamo la costruzione **OR**, la probabilità per le coppie simili aumenta.

  $$P_{\text{simili OR}} = 1 - (1 - 0.64)^4 \approx 0.878$$

  La probabilità per le coppie non simili diminuisce:

  $$P_{\text{non simili OR}} = 1 - (1 - 0.04)^4 \approx 0.0064$$

  Quindi è diventato $(0.2, 0.8, 0.9936, 0.1215)-$sensitive.
  Analogamente possiamo calcolare gli altri casi in sequenza dopo il risultato appena raggiunto.
- **OR-AND**: $(0.2, 0.8, 0.9936, 0.1215)-$sensitive
- **OR-AND**: $(0.2, 0.8, 0.9999996, 0.0008715)-$sensitive

Demo: [Locality Sensitive Hashing — Threshold + Banding Exploration | Desmos]

Finora abbiamo utilizzato le distanze di Jaccard e Hamming, che misurano quanto due insiemi siano vicini. Ma abbiamo anche altre alternative. La distanza di Hamming consiste nel contare il numero dei componenti in cui due vettori differiscono.

Immaginiamo di avere due vettori x e y di dimensione d, e denotiamo la loro distanza di Hamming come $h(x, y)$. Definiamo la funzione $f_i(x)$ come il valore della i-esima posizione nel vettore, quindi se la i-esima posizione nei vettori è uguale avremo $f_i(x) = f_i(y)$. La probabilità che si verifichi questo evento può essere calcolata come:

$$1 - \frac{f(x, y)}{d}$$

E la famiglia delle funzioni f_i, ovvero F, è $(d_i, d_2, 1 - \frac{d_1}{d}, 1 - \frac{d_2}{d})$-sensitive con $d_1 < d_2$.

> ChatGPT: Perchè $d_1 < d_2$?
>
> La famiglia delle funzioni LSH è definita come (d_1, d_2, p_1, p_2)**-sensibile**, il che significa che:
>
> - Se due vettori hanno distanza **al massimo** d_1, la probabilità che vengano mappati alla stessa posizione (cioè $f(x) = f(y)$) è **alta** (p_1).
> - Se due vettori hanno distanza **almeno** d_2, la probabilità che vengano mappati alla stessa posizione è **bassa** (p_2).
>
> Affinché questa proprietà sia significativa, è necessario che $d_1 < d_2$, perché:
>
> 1. **Se** $d_1 = d_2$, non avremmo un criterio efficace per distinguere coppie vicine da coppie lontane.
> 2. **Se** $d_1 > d_2$, avremmo il problema opposto: le coppie più distanti avrebbero una maggiore probabilità di collisione rispetto a quelle più vicine.

Esiste anche la distanza coseno. Consiste in una misura nello spazio Euclideo, si usa il coseno dell'angolo tra i vettori associati ai punti considerati.

$$d(A, B) = \theta = \arccos\left(\frac{A \cdot B}{||A|| \cdot ||B||}\right)$$

Che ha ovviamente valori in $[0, \pi]$, cioè l'angolo ha sempre valore tra 0 e 180 gradi.

Pertanto possiamo considerare:

$$d([x_1, \ldots, x_n], [y_1, \ldots, y_n]) = 1 - \frac{A \cdot B}{||A|| \cdot ||B||}$$

dove $A \cdot B = ||A|| \cdot ||B|| \cos\theta$.

Per la distanza coseno si usa una tecnica chiamata Iperpiani Random: scegliamone uno e consideriamo il suo vettore normale v, calcoliamo i prodotti $v \cdot x$ e $v \cdot y$. Possono esserci due casi, $sign(v \cdot x) = sign(v \cdot y)$ oppure $sign(v_f \cdot x) \neq sign(v_f \cdot y)$.

Definiamo una funzione $f(x)$ scegliendo a caso un vettore v_f. Allora $f(x) = f(y)$ se e solo se $sign(v_f \cdot x) = sign(v_f \cdot y)$. La famiglia F, è $(d_i, d_2, \frac{180 - d_1}{180}, \frac{180 - d_2}{180})$-sensitive con $d_1 < d_2$.

Ogni vettore v determina una funzione hash h_v con due bucket:

- $h_v(x) = +1$ se $v \cdot x \geq 0$

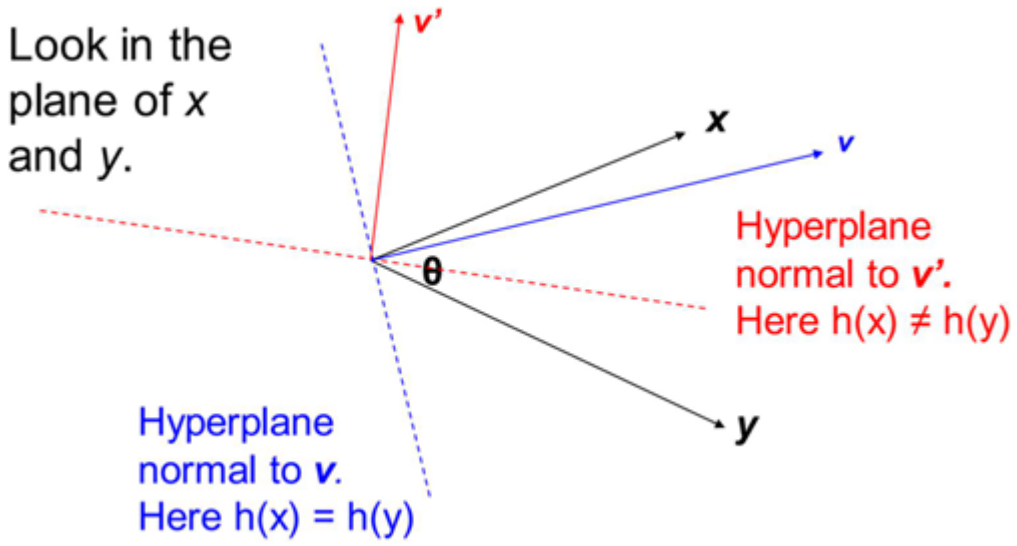
- $h_v(x) = -1$ se $v \cdot x < 0$

L'insieme di tutte le funzioni Local Sensitive derivate da un vettore si chiama H.

Dati x e y allora:

$$Pr[h(x) = h(y)] = 1 - \frac{d(x,y)}{\pi}$$

Per esempio, analizziamo due punti x e y e vediamo come vengono classificati sulla base di due diversi iperpiani.



Notiamo che i due iperpiani v e v' sono abbastanza diversi, e infatti in uno vengono i due punti classificati come simili e nell'altro come diversi.

Vediamo come costruire la signature:

1. Prendiamo tot. vettori random e calcoliamo l'hash per ogni vettore
2. Il risultato è la signature, ovvero uno sketch composto da +1 e -1 per ogni dato
3. Possiamo usarlo per LSH

Questa tecnica preserva la randomicità dato che ogni vettore normale ha la stessa probabilità di mappare i dati in una direzione o l'altra.

Uno spazio euclideo (n)-dimensionale è un insieme di punti rappresentati come vettori di (n) numeri reali. La distanza tra due punti $p = (p_1, p_2, \ldots, p_n)$ e $q = (q_1, q_2, \ldots, q_n)$ viene calcolata usando una norma, che dipende dal parametro (r) nella definizione della norma L_r.

Concentriamoci sulla distanza euclidea, ovvero con $r = 2$. Si ha:

$$L_2 = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

Consiste nalla distanza in linea retta tra punti p_i e q_i con dimensionalità n.

Costruire una funzione LSH per la distanza euclidea è più complesso perché dipende dalla geometria "in linea retta" nello spazio (n)-dimensionale. LSH cerca di raggruppare punti vicini in "bucket" (contenitori) con alta probabilità, preservando questa proprietà, ma deve farlo in modo efficiente e approssimativo.

Un **query point** è semplicemente il punto per cui vogliamo trovare vicini simili. Supponiamo di avere un database di punti nello spazio e vogliamo cercare quali sono i punti più vicini a un nuovo punto x. In questo contesto, x è il query point.

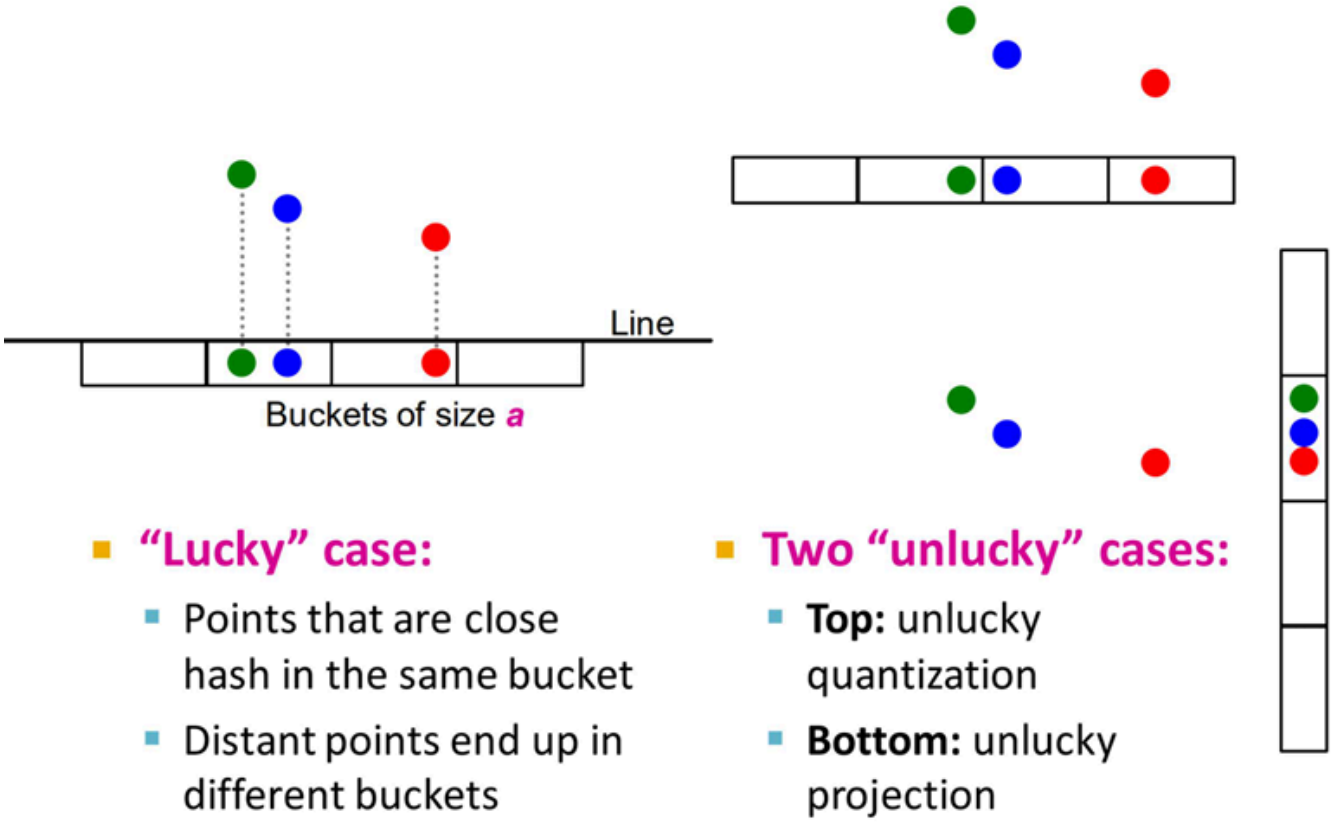
La funzione $h(v)$ è una funzione hash che associa ogni punto dello spazio a un valore scalare. Questo valore viene poi usato per assegnare il punto a un **bucket**, ovvero un intervallo che raggruppa punti vicini.Per organizzare i punti in gruppi (bucket), la proiezione viene **quantizzata** in intervalli di larghezza w:

$$h(v) = \frac{x \cdot v + b}{w}$$

- w è la **larghezza del bucket**
- b è una **variabile casuale** distribuita uniformemente in $[0, w]$

Questa operazione serve per **ridurre la dimensionalità** e permettere confronti più efficienti.

In base a come si dispone il vettore casuale potremo effettivamente mappare correttamente i vettori vicini, o avere un caso sfortunato dove mappiamo in maniera poco significiativa.



Riassumendo:

1. **Shingling:** Trasformi i documenti in insiemi di shingle e assegni ID con hashing.
2. **Min-Hashing:** Crei firme brevi che mantengono la similarità tra insiemi, usando hashing per evitare permutazioni.
3. **LSH:** Usi hashing per trovare velocemente coppie di firme (e quindi documenti) probabilmente simili.

https://colab.research.google.com/drive/1FTVN0dvm-eCGSEIF0d7x3i-PH7BM8yJb?usp=sharing

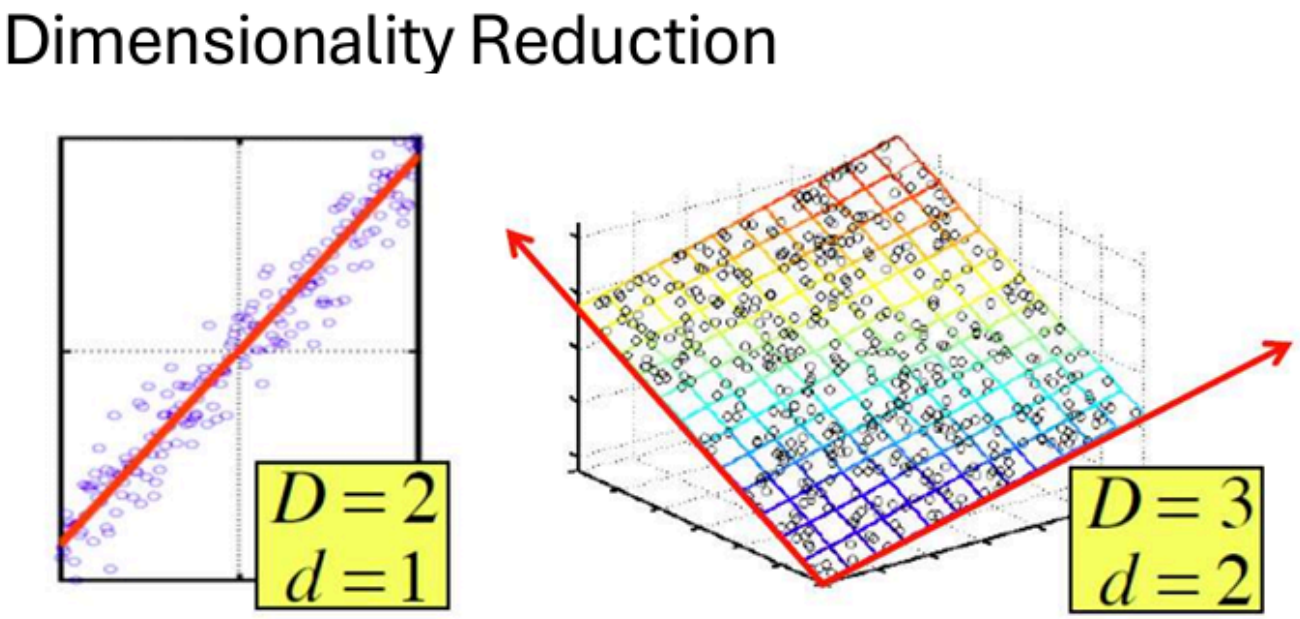
Lezione 7 - Dimensionality Reduction

Potremmo avere fonti di dati con matrici di grandi dimensioni molto sparse, in realtà potrebbero essere riassunte in matrici ben più compatte. Possiamo avere matrici strette quindi poche righe o poche colonne, il che è molto più efficiente. Le troviamo con la dimensionality reduction. Alcune feature potrebbero essere irrilevanti, ed è meglio liberarcene.

Esiste la Unsupervised feature selection:

- Non seleziona un sottoinsieme di feature, ma ne crea di nuove definite come funzione di tutte le feature.
- Sono punti in uno spazio multidimensionale, non etichette in uno spazio multidimensionale

Per esempio se abbiamo dati su 3 dimensioni possiamo immaginare che i dati possano distribuirsi su un singolo piano a 2 dimensioni. Possiamo quindi approssimare la rappresentazione dei dati!



Per esempio se avessimo:

$$A = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 2 & 2 & 2 & 0 & 0 \\ 3 & 3 & 3 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 2 & 2 \end{bmatrix}$$

Possiamo notare che tutte le righe possono essere ricostruite dalla base:

$$[1, 1, 1, 0, 0][0, 0, 0, 1, 1]$$

Ricordiamo che il rango di una matrice è il numero di colonne linearmente indipendenti. Per esempio:

$$A = \begin{bmatrix} 1 & 2 & 1 \\ -2 & -3 & 1 \\ 3 & 5 & 0 \end{bmatrix}$$

Le prime due righe sono linearmente indipendenti quindi rango almeno 2, ma notiamo che la prima riga è la somma delle altre due quindi il rango non è 3. Quindi rango 2. Se il rango è inferiore al numero totale di righe o colonne, significa che la matrice (A) può essere rappresentata usando meno informazioni. Invece di memorizzare tutte e 3 le righe, puoi usare solo le 2 righe della base e le loro coordinate.

Prima riga $(1, 2, 1)$: È esattamente $\mathbf{v}_1$, quindi ha coordinate $(1, 0)$ nella base B.

Seconda riga $(-2, -3, 1)$: È esattamente $\mathbf{v}_2$, quindi ha coordinate $(0, 1)$ nella base B.

Terza riga $(3, 5, 0)$: Notiamo che si può scrivere come $\mathbf{v}_1 - \mathbf{v}_2$, quindi le sue coordinate sono $(1, -1)$.

Ora possiamo dire che la matrice A, rispetto alla base B, è rappresentata dalle seguenti coordinate:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & -1 \end{bmatrix}$$

Quindi ogni riga della matrice è un punto nello spazio tridimensionale, ma una volta individuate le nuove basi allora cambiano le coordinate e abbiamo effettivamente ridotto in maniera efficiente! Potremmo avere anche uno spazio 2-dimensionale, a quel punto la dimensionality reduction si riduce ad identificare un punto in una retta di regressione, ovviamente introduciamo dell'errore ma è accettabile.

Vantaggi:

- Scoprire correlazioni nascoste
- Rimuovere feature ridondanti
- Visualizzazione
- Memorizzazione ed elaborazione migliori

Unsupervised feature selection

Dato un insieme di punti in uno spazio d-dimensionale, proiettiamo i dati in uno spazio con meno dimensioni preservando le informazioni principali, quindi minimizziamo il quadrato dell'errore quando ricostruiamo i dati originali.

Data una matrice quadrata M, sia λ una costante e sia e un vettore colonna non zero con lo stesso numero di righe d M. Allora λ è un autovalore di M se e è il suo corrispondente autovettore di M se $Me = \lambda e$.
Se e è un autovettore di M e c è una costante allora anche ce è un autovettore di M con stesso autovalore.

Moltiplicare il vettore per la costante cambia il modulo ma non la direzione. Per evitare ambiguità consideriamo che ogni autovettore sia vettore unitario quindi modulo 1.

Tecniche come **PCA (Analisi delle Componenti Principali)** sfruttano gli autovalori per ridurre la dimensione dei dati e trovare le direzioni più importanti in uno spazio.

Autocoppia: un autovalore e il corrispondente autovettore.

Tramite un algoritmo possiamo calcolare tutte le autocoppie in una matrice simmetrica.

Riscriviamo l'equazione $Me = \lambda e$ come $(M - \lambda I)e = 0$ dove I è la matrice identità, mentre 0 è il vettore con tutte le entry pari a 0. Per essere vera questa equazione, ponendo e diverso da zero, allora il determinante della matrice $(M - \lambda I)$ deve essere zero.

Tale matrice è simile alla matrice M ma se M ha valore c in un elemento diagonale, allora $(M - \lambda I)$ avrà $c - \lambda$ nella stessa posizione. Il determinante della matrice può essere calcolato con il metodo della pivotal condensation (skip)

Principal Component Analysis

La PCA è una tecnica che prende un dataset relativo ad un insieme di tuple in uno spazio ad alta dimensione e trova le direzioni lungo le quali le tuple si allineano meglio. L'insieme di tuple viene trattato come una matrice M e troviamo gli autovettori di MM^T o M^TM.

Esempio. Abbiamo i punti $(1, 2), (2, 1), (3, 4), (4, 3)$. Troviamo la matrice:

$$M = \begin{bmatrix} 1 & 2 \\ 2 & 1 \\ 3 & 4 \\ 4 & 3 \end{bmatrix}$$

Calcoliamo la matrice di covarianza (metodo diverso rispetto quello visto prima):

$$M^TM = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 3 & 1 & 4 & 3 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 2 & 1 \\ 3 & 4 \\ 4 & 3 \end{bmatrix} = \begin{bmatrix} 30 & 28 \\ 28 & 30 \end{bmatrix}$$

Trova gli autovalori:

$$(30 - \lambda)(30 - \lambda) - 28 \cdot 28 = 0$$
$$\dots$$
$$\lambda = 58 \quad \lambda = 2$$

E i corrispondenti autovettori:

$$\begin{bmatrix} 30 & 28 \\ 28 & 30 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = 58 \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{cases} 30x + 28y = 58x \\ 28x + 30y = 58y \end{cases} \rightarrow \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

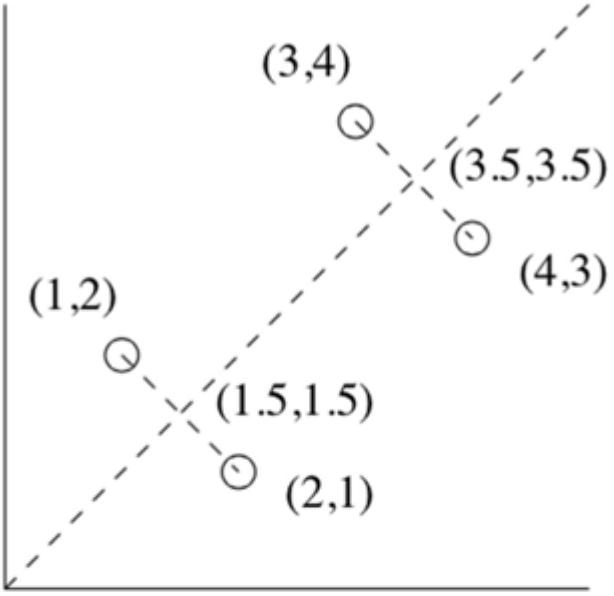
$$\begin{cases} 30x + 28y = 2x \\ 28x + 30y = 2y \end{cases} \rightarrow \begin{bmatrix} \frac{-1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

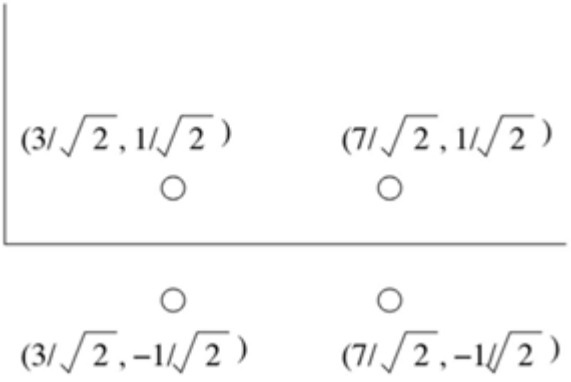
Quindi abbiamo la matrice degli autovettori di M^TM ovvero:

$$E = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

La magia avviene qui: moltiplicando la matrice degli autovettori con la matrice inziale M possiamo ottenere una rotazione rigida degli assi che meglio rappresenta i nostri dati, ottenendo le coordinate che tali dati avranno sui nuovi assi:

$$ME = \begin{bmatrix} 1 & 2 \\ 2 & 1 \\ 3 & 4 \\ 4 & 3 \end{bmatrix} \begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix} = \begin{bmatrix} 3\sqrt{2} & 1\sqrt{2} \\ 3\sqrt{2} & -1\sqrt{2} \\ 7\sqrt{2} & 1\sqrt{2} \\ 7\sqrt{2} & -1\sqrt{2} \end{bmatrix}$$





Notiamo quindi che possiamo rappresentare i dati dando priorità all'asse con maggior varianza, e cancelliamo gli autovettori degli assi che hanno meno varianza. Il numero delle dimensioni dipende da noi, inoltre qui stiamo passando semplicemente da due dimensioni a una, ma immaginando di averne 50 e decidere di passare a 10 è un bel miglioramento.

Notebook: https://colab.research.google.com/drive/1lykoVbdYyHVvdJ7dUme5yN2wNGbQOrQV?usp=sharing

Singular Value Decomposition

Un altro metodo per liberarci delle dimensioni che meno rappresentano i nostri dati è la SVD, applicabile a qualsiasi matrice. Sia $M(m, n)$ una matrice dei dati di input e sia r il suo rango, ricordiamo che il rango è il più grande numero di righe o colonne linearmente indipendenti. Allora possiamo definire le seguenti matrici:

- $U(m, r)$ è una matrice colonna ortonormale di sinistra, ognuna delle sue colonne è un vettore unitario e ortogonale alle altre colonne. Si considerano m documenti ed r concetti.
- $V(n, r)$ è una matrice colonna ortonormale di destra, si considerano n termini ed r concetti.
- Σ è una matrice diagonale di dimensiione r dove i suoi elementi sono chiamati singoli valori di M. Ogni valore rappresenta la forza di ogni concetto.

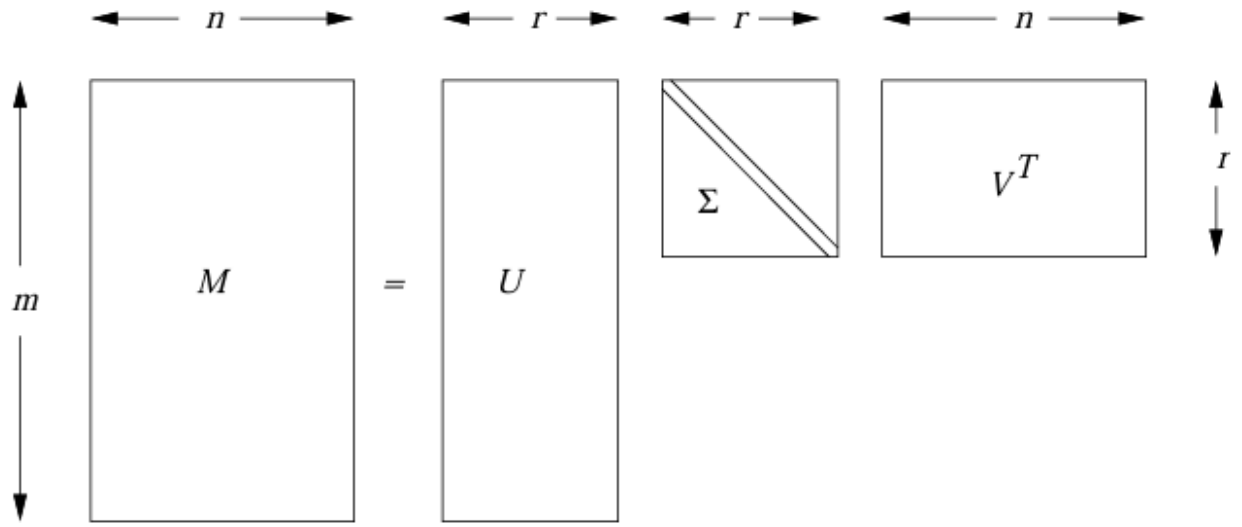


Figure 11.5: The form of a singular-value decomposition

Pertanto la matrice iniziale viene riassunta con queste tre matrici. Per ogni matrice è sempre possibile decomporre una matrice reale in questo modo. Per esempio:

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 2 & 2 & 2 & 0 & 0 \\ 3 & 3 & 3 & 0 & 0 \\ 4 & 4 & 4 & 0 & 0 \\ 0 & 0 & 0 & 4 & 4 \\ 0 & 0 & 0 & 5 & 5 \\ 0 & 0 & 0 & 2 & 2 \end{bmatrix} = \begin{bmatrix} .14 & 0 \\ .42 & 0 \\ .56 & 0 \\ .70 & 0 \\ 0 & .60 \\ 0 & .75 \\ 0 & .30 \end{bmatrix} \begin{bmatrix} 12.4 & 0 \\ 0 & 9.5 \end{bmatrix} \begin{bmatrix} .58 & .58 & .58 & 0 & 0 \\ 0 & 0 & 0 & .71 & .71 \end{bmatrix}$$

In questa matrice il rango è 2 quindi 2 sono i concetti.

La cosa importante da capire è che nella SVD si può osservare ogni colonna di ogni sottomatrice, che rappresentano i concetti nascosti nella matrice originale. Per esempio, quello che abbiamo visto poco fa è una matrice dove ogni riga è un utente che esprime valutazioni sui film, le prime 4 righe sono utenti a cui piaccioni film Sci-Fi, le ultime 3 sono relative a film romantici.

In questo senso la scomposizione sta meglio rappresentando l'associazione di ogni utente ai concetti, Immaginiamo la matrice iniziale come il voto che ogni utente da ai vari film. La matrice U sta connettendo gli utenti ai concetti, per esempio il primo utente ha connessione solo con i film Sci-Fi, ma comunque non così tanto. Il quarto utente invece è molto più legato dato che da valutazioni alte.

La matrice V collega i film ai concetti. Notiamo che ovviamente i primi 3 sono legati al primo concetto, che è Sci-Fi, gli ultimi due film sono legati al concetto Romance. Infine, la matrice Σ presenta la forza di ogni concetto, notiamo che Sci-Fi è lievemente più forte di Romance perchè nei dati è più rappresentato. Nel caso generico i dati non saranno così ben ripartiti, qui abbiamo riportato un esempio evidente per capire meglio.

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 3 & 3 & 3 & 0 & 0 \\ 4 & 4 & 4 & 0 & 0 \\ 5 & 5 & 5 & 0 & 0 \\ 0 & 2 & 0 & 4 & 4 \\ 0 & 0 & 0 & 5 & 5 \\ 0 & 1 & 0 & 2 & 2 \end{bmatrix} =$$

$$M'$$

$$\begin{bmatrix} .13 & .02 & -.01 \\ .41 & .07 & -.03 \\ .55 & .09 & -.04 \\ .68 & .11 & -.05 \\ .15 & -.59 & .65 \\ .07 & -.73 & -.67 \\ .07 & -.29 & .32 \end{bmatrix} \begin{bmatrix} 12.4 & 0 & 0 \\ 0 & 9.5 & 0 \\ 0 & 0 & 1.3 \end{bmatrix} \begin{bmatrix} .56 & .59 & .56 & .09 & .09 \\ .12 & -.02 & .12 & -.69 & -.69 \\ .40 & -.80 & .40 & .09 & .09 \end{bmatrix}$$

$$U \qquad \Sigma \qquad V^{\mathrm{T}}$$

Facendo una modifica all'esempio precedente, per esempio cambiando da 0 a 1 uno dei film Sci-Fi delle tre persone a cui piacciono film Romance il rango diventa 3, quindi i concetti sono 3: i primi due sono ancora Sci-Fi e Romance, il terzo è un po' più fumoso, ma in effetti non ha molta importanza. Immaginiamo che potrebbe essere eliminato, facendo quindi riduzione della dimensionalità con la SVD.

Immaginiamo di voler rappresentare una matrice molto grande tramite la sua SVD. Le sottomatrici sono anch'esse troppo grandi da essere memorizzate, la miglior cosa da fare è ridurre la loro dimensionatà. Per esempio, poco fa avevamo tre valori singoli, vogliamo ridurre la dimensionalità quindi l'ultimo concetto per importanza lo impostiamo a zero. Ne segue, che a questo concetto stiamo dando importanza zero, e si perderanno le relative informazioni. La matrice che otterremo sarà quella che meglio rappresenta i dati di partenza ma con dimensionalità ridotta, la perdita di informazioni è accettabile.

La scelta di rimuovere il valore singolare più piccolo riflette il voler minimizzare l'errore quadratico medio tra la matrice iniziale e la sua approssimazione. La norma di Frobenius consente di quantificare l'errore tra tali matrici.

$$||M||_F = \sqrt{\Sigma_{ij} M_{ij}^2}$$

Pertanto per considerare la differenza di matrici avremo proprio:

$$||A - B||_F = \sqrt{\Sigma_{ij}(a_{ij} - b_{ij})^2} \quad (*)$$

è possibile dimostrare che data una matrice A scomposta con SVD, allora la nuova matrice B ottenuta riducendo le dimensioni a $k = \text{rank}(B)$ è la miglior rappresentazione possibile della matrice A. Significa che minimizza la formula $(*)$. Questo perchè stiamo introducendo meno errore possibile con le dimensioni scelte.

Ci chiediamo, come decidiamo quante dimensioni tenere? Diciamo che è bene tenere quelle dimensioni che ci consentono di mantenere il 90% delle informazioni di partenza. Quindi la somma dei quadrati dei valori singolari dovrebbe essere almeno il 90% della somma dei quadrati di tutti i valori singolari

Rapporto tra la SVD e decomposizione mediante autovalori ...

La SVD ha complessità $O(nm^2)$, è implementata in diversi pacchetti in Matlab o R. Comunque sia la SVD è molto utile, ma è difficile da interpretare dato che ci stiamo privando di matrici sparse allora i dati sono molto densi.

Decomposizione CUR

La SVD ha complessità cubica ed è eccessiva. Si ovvia con la decomposizione CUR. Magari avremo più errore ma almeno sarà un'operazione rapida. Si compone di tre matrici, C, U ed R. C sta per Column e consiste in un set di colonne della matrice iniziale, mentre R sta per rows ed è un set di righe della matrice iniziale, per sceglierle si utilizza un algoritmo random. U invece è la pseudo inversa dell'intersezione di C ed R. Ha complessità quadratica.
Perchè questo funziona? C'è un teorema. Seleziona circa 4k colonne/righe e sei a posto. Il sampling avviene randomicamente, sia per colonne che righe, questo potrebbe far nascere duplicati. Questo metodo è facile da interpretare, i vettori della base sono esattamente righe e colonne della matrice.

Nonnegative Matrix Factorisation

Consente una rappresentazione dei dati lineare con $A \approx W(m,k) * H(k,n)$ dove $k(m+n) < mn$.
Consiste in una tecnica di riduzione della dimensionalità non supervisionata

- A è la matrice originale, a valori non negativi. Le righe sono le features e le colonne sono le osservazioni.
- W è la matrice dei vettori base, detta matrice dizionario, una combinazione lineare di questi consentono di approssimare ogni valore di A. Ogni colonna è un vettore base.
- H è la matrice di attivazione, ogni vettore esprime una combinazione lineare delle basi per rappresentare una entry di A. Ogni colonna rappresenta i pesi di un vettore base.

La scelta del rango è importante perchè così garantiamo una compressione dei dati significativa. La nonnegatività riflette meglio la realtà di molti dataset, inducendo più sparsità e rendendo i dati più interpretabili. Tale decomposizione inoltre consente di avere una sorta di decomposizione in parti o concetti sottostanti, delineati nella matrice dizionario. NMF è una tecnica non supervisionata quindi non richiede

etichette per scoprire strutture latenti nei dati il che è un vantaggio. Ci sono librerie in R e Python che consentono la NMF.

Lezione 8 - Sistemi di raccomandazione

Un sistema di raccomandazione è una classe di applicazioni che predicono le risposte degli utenti sulla base delle loro preferenze. Si distinguono due gruppi di tecnologie principali:

- **Content-based systems:** esamina le proprietà degli elementi per raccomandarne di nuovi
- **Collaborative filtering systems:** usa misure di similarità tra utenti/prodotti per raccomandare nuovi prodotti (quindi oggetti simili o proprietà di utenti simili)

Abbiamo quindi un insieme di oggetti ed utenti rappresentate dentro una utility matrix $U(n, m)$ dove nelle righe abbiamo gli utenti e nelle colonne gli oggetti, che è una matrice sparsa dove per ogni coppia utente-oggetto calcola il grado di preferenza dell'utente. Vogliamo predire le entry vuote della matrice per inferire le preferenze dell'utente.

I sistemi di raccomandazione sono molto utili per gli utenti visto che online la quantità di prodotti è pressochè illimitata, quindi consistono in un modo per aiutarli. Costruire tale matrice comunque non è semplice, possiamo adottare due approcci:

- Esplicito: chiedere di valutare un item
- Implicito: apprendere dalle azioni dell'utente

Vediamo adesso qualoi algoritmi possiamo adottare:

- **Content-Based:** gli oggetti raccomandati ad un utente siano simili agli oggetti valutati positivamente dallo stesso. Tale sistema si concentra sulle proprietà degli oggetti, la similarità viene determinata misurando le proprietà degli item. Per ogni utente si crea un profilo, cioè un insieme di features rappresentanti caratteristiche salienti.

  Pro: non ci servono dati di altri utenti, possiamo consigliare agli utenti con gusti unici, possiamo raccomandare articoli nuovi e non popolari ed è facile da interpretare.

  Contro: trovare feature giuste è difficile e soprattutto abbiamo difficoltà a raccomandare elementi ad utenti nuovi.

- **Collaborative Filtering:** consideriamo un utente x e troviamo altri N altri utenti i cui rating sono simili a quelli di x, stimiamo allora i rating di x sulla base degli altri rating.

  Come misurare la similarità? Possiamo farla con Jaccard Similarity:

  $$Sim(x, y) = \frac{|r_x \cap r_y|}{|r_x \cup r_y|}$$

  Oppure con la similarità del coseno $sim(x, y) = cos(r_x, r_y)$ o il coefficiente di correlazione di Pearson.