Università degli Studi di Salerno
Dipartimento di Informatica

Laurea Magistrale in Informatica
Sistemi Operativi Avanzati

Twitter Sentiment Analysis with Spark

Professore
Prof. Giuseppe Cattaneo

Studenti
D'Ambrosio Giuseppe
Vigorito Paolo

Anno Accademico 2018-2019

Abstract

Le opinioni, i feedback e le informazioni che gli utenti forniscono all'interno di siti e, in generale, sul web costituiscono una risorsa molto preziosa per aziende e compagnie commerciali. Grazie al fenomeno dei social network, tali dati sono in continua crescita e risultano sempre più facilmente reperibili, ma allo stesso tempo sono difficili da analizzare. L'analisi di questo tipo di dati appartiene al dominio della Sentiment Analysis, processo che si pone l'obiettivo di conoscere l'opinione degli utenti rispetto ad un determinato prodotto, soggetto o argomento e che è diventato un'area di ricerca di grande interesse negli ultimi anni. Twitter è considerato uno dei social network più adatti all'applicazione della Sentiment Analysis, grazie al fatto che gli utenti tendono ad esprimere liberamente i propri pensieri tramite i caratteristici "tweet".

In questo lavoro si andrà ad implementare un sistema su Apache Spark, un framework open-source per la programmazione su architetture distribuite, che risultano particolarmente adatte a lavorare con i Big Data, volto a classificare i tweet in negativi, positivi o neutri in base all'opinione da essi espressa. L'analisi è effettuata attraverso un classificatore Naive Bayes utilizzando la libreria di machine-learning MLlib fornita da Apache Spark stesso. L'algoritmo è inizialmente classificato su di un dataset già etichettato per generare un modello Naive Bayes da poter utilizzare per la classificazione di "tweet" non etichettati.

Indice

1 - Introduzione

Lo scopo di questo progetto è di dimostrare come può essere elaborata e analizzata una grande quantità di dati attraverso l'utilizzo di un sistema distribuito. In particolare l'obiettivo implementativo è di creare un'applicazione per la Sentiment Analysis che sia in grado di classificare porzioni di testo provenienti dal social network Twitter, chiamati "tweet".

Dato l'elevato numero di utenti e, di conseguenza di tweet prodotti, vi è la necessità di rapportarsi con una mole di dati molto grande con un conseguente carico di lavoro molto elevato. Ciò pone il problema nell'ambito dei Big Data che, per definizione, necessitano di tecnologie e paradigmi adeguati per essere elaborati in modo efficace.

L'utilizzo di un cluster risulta quindi un'ottima soluzione per il problema in questione e Apache Spark, grazie alle sue caratteristiche e alle sue funzionalità diviene lo strumento ideale per tale progetto, grazie anche alla libreria MLlib che mette a disposizione, che fornisce diverse implementazioni di algoritmi di machine-learning.

La scelta dell'algoritmo di machine-learning da applicare per effettuare la Sentiment Analysis è ricaduta su di un classificatore Naive Bayes. Studi accademici dimostrano come approcci basati su questo tipo di classificatori permettano di ottenere un'alta accuratezza a discapito di prestazioni più basse sul tempo di esecuzione, mentre algoritmi basati sul lessico forniscono migliori risultati in termini di tempo perdendo in accuratezza[1]. Sfruttando il framework Apache Spark risulta però possibile evitare la perdita in prestazioni che si ha con i classificatori Naive Bayes riuscendo così ad ottenere un'elevata

accuratezza con tempi di esecuzione ridotti, grazie all'utilizzo del calcolo distribuito.

Il modello Naive Bayes è creato in seguito ad una fase di addestramento eseguita su un dataset di tweet già etichettato e viene in seguito applicato su di un dataset di tweet non etichettati di grandi dimensioni per effettuarne la classificazione.

Il linguaggio di programmazione utilizzato è Scala in quanto permette di ottenere le migliori prestazioni possibili in linea con lo scopo del progetto.

1.1 Big Data

Il termine Big Data indica genericamente una grande mole di dati eterogenei prodotti in modo continuo che, a causa della complessità intrinseca per essere elaborati, richiede tecnologie e metodi specifici per l'estrazione di informazioni utili e significative. Dataset così grandi e complessi rendono inefficaci le tradizionali tecniche di elaborazione dati e, di conseguenza richiedono strumenti non convenzionali per estrapolare, gestire e processare informazioni entro un tempo ragionevole, andando per questo a porsi come un campo ideale per il calcolo parallelo e distribuito.

I Big Data non comprendono solamente dati di grandi dimensioni, ma possono essere costituiti anche da piccoli pacchetti di dati prodotti in modo costante e ad alta velocità, come nel caso di flussi di dati provenienti da reti di sensori o sistemi di streaming. Per questo motivo non è solo la dimensione del dataset a collocare un problema nella categoria dei Big Data, piuttosto è la complessità della sua elaborazione.

L'analista Doug Laney, in uno studio del 2001, definisce il modello di crescita dei dati come tridimensionale attraverso il modello delle 3V: volume, velocità e varietà. Esso spiega come i dati siano prodotti in grandi quantità, volume, con

frequenza sempre maggiore, velocità, e con caratteristiche anche molto differenti tra loro, varietà. Il modello proposto da Laney risulta ancora estremamente valido ed è stato esteso con ulteriori proprietà negli anni, per indicare ulteriori caratteristiche peculiari dei Big Data, come la veridicità che ne descrive l'accuratezza e l'affidabilità, il valore che rappresenta quanto i dati possano essere trasformati in informazioni utili e, infine, la variabilità che specifica il livello di accuratezza.

I Big Data sono un argomento interessante per molte aziende che negli ultimi anni hanno investito molto su questa tecnologia, motivate dalla necessità di analizzare enormi quantità di dati per conoscere meglio la propria clientela, migliorare le strategie di marketing e il loro business. Tendenze future, consigli di acquisto, pubblicità, analisi di mercato sono solo alcune delle potenzialità per cui i Big Data possono essere sfruttati nell'ambito del marketing.

1.2 Sentiment Analysis

La Sentiment Analysis si riferisce all'utilizzo dell'elaborazione del linguaggio naturale, analisi del testo, computazione linguistica e biometrica per identificare, estrarre, quantificare e studiare opinioni e informazioni soggettive.[4]

Essa viene largamente utilizzata per conoscere le opinioni delle persone riguardo specifici argomenti come possono essere politica, prodotti commerciali, eventi e così via, andando ad analizzare recensioni, sondaggi, commenti e tutte le interazioni che gli utenti hanno sul web, soprattutto tramite i social media.

Obiettivo principale della Sentiment Analysis è la classificazione della polarità di un dato testo, ossia stabilire se tale testo esprime uno stato emozionale positivo, negativo o neutro. Ciò può essere fatto attraverso tecniche di machine-learning che siano supervisionate o non supervisionate. La

classificazione non supervisionata utilizza prevalentemente approcci basati sul lessico che sfruttano un dizionario di parole etichettate con relativo peso e polarità, per analizzare un testo ed effettuarne la valutazione. Metodi di classificazione supervisionati, invece, includono algoritmi come il Naive Bayes, le reti neurali e le macchine a vettori di supporto che necessitano di un dataset di addestramento dal quale poter imparare per acquisire la capacità di analizzare nuovo testo. Questo tipo di tecniche utilizzano quindi un dataset già etichettato da cui generano una serie di regole a cui affidarsi in seguito quando saranno forniti loro i dati da analizzare[4].

I classificatori Naive Bayes hanno dimostrato di essere un metodo di machine-learning semplice ed efficace in studi sulla classificazione del testo, risultando persino ottimali in alcuni casi[2][3]. Essi sono classificatori probabilistici, ossia per classificare un dato teso utilizzano equazioni che calcolano la probabilità che esso appartenga ad una specifica classe. La classe con la probabilità più alta è quella che verrà assegnata al testo.

1.3 Twitter

Twitter è un servizio di notizie e microblogging su cui gli utenti postano sulla propria pagina personale e interagiscono tra loro tramite messaggi chiamati "tweet", che consistono in messaggi di testo di lunghezza massima di 280 caratteri. Il servizio vanta grande popolarità in tutto il mondo, nel 2012 ha raggiunto i 500 milioni di iscritti e a partire dal 2016, contava oltre 319 milioni di utenti attivi mensilmente.[4]

Twitter è considerato uno dei social network più adatti all'applicazione della Sentiment Analysis grazie ad una delle sue caratteristiche chiave, cioè l'immediatezza nella comunicazione e nell'espressione dei propri pensieri tramite i tweet. Gli utenti di Twitter, infatti, tendono ad esprimere liberamente le

proprie opinioni, rendendolo una fonte ideale di dati utilizzabili per ottenere informazioni riguardanti una grande varietà di argomenti. Altro vantaggio che rende Twitter largamente utilizzato in ambito di ricerca sono le Twitter API che permettono l'accesso, seppur limitato, ai dati in esso contenuti, una possibilità che risulta sempre più rara in altri social network.

2 - Tecnologie utilizzate

La crescente mole di dati generati da sorgenti eterogenee ha posto l'attenzione su come essi possano essere estratti, archiviati e processati per ottenere informazioni, conoscenze e di conseguenza profitto. La difficoltà che si riscontra nell'affrontare tali processi è dovuta alla gestione dei Big Data attraverso database tradizionali, difficoltà causata dalle caratteristiche stesse di questa tipologia di dataset. Conseguentemente a ciò è nata la necessità di sviluppare modelli di elaborazione che siano in grado di gestire e processare questi dati, in modo efficace ed efficiente.

L'utilizzo di architetture distribuite, unitamente a paradigmi di calcolo parallelo, permette di ottenere le prestazioni necessarie alla gestione dei Big Data, andando a sfruttare cluster di computer, ossia un insieme di macchine connesse tra loro tramite una rete. Scopo del cluster è distribuire un'elaborazione molto complessa tra i vari computer, aumentando la potenza di calcolo del sistema e garantendo una maggiore disponibilità di servizio, con un conseguente maggior costo e complessità di gestione dell'infrastruttura.[4]

2.1 Apache Spark

Apache Spark è un framework open source per il calcolo distribuito general purpose. Esso fornisce un'interfaccia per la programmazione di cluster con parallelismo dei dati implicito e tolleranza ai guasti. Originariamente sviluppato dall'AMPlab di Berkeley dell'Università della California, il sorgente di Spark è stato successivamente donato alla Apache Software Foundation.

Apache Spark ha come fondamento della sua architettura l'RDD (Resilient Distributed Dataset), un insieme di dati di sola lettura distribuito all'interno di

un cluster di macchine e mantenuto in modo da garantire al tolleranza ai guasti, sui quali vengono effettuate una serie di trasformazioni per elaborare i dati.[4]
L'utilizzo di Apache Spark è stata una scelta quasi obbligata proprio per la difficoltà di gestione e sfruttamento del cluster a disposizione, attraverso questo strumento infatti, si è potuto utilizzare a pieno le risorse delle singole macchine, andando ad ottimizzare l'elaborazione del problema.
Attraverso l'uso di Apache Spark si ottengono una serie di vantaggi rispetto alla controparte Hadoop MapReduce. Innanzitutto per quanto riguarda la velocità risulta capace di eseguire programmi fino a 100 volte più velocemente se eseguiti in memoria principale, e fino a 10 volte più velocemente su memoria secondaria. Questo risulta possibile grazie al processamento dei nodi worker effettuato in memoria principale, prevenendo operazioni di I/O non necessarie. Oltre a ciò Apache Spark si presta molto bene a lavorare con algoritmi iterativi, rendendolo ideale per l'implementazione di tecniche di machine-learning principalmente basate proprio su job iterativi.
In aggiunta al nucleo operativo di gestione e processamento dei dati, Apache Spark fornisce una serie di librerie specifiche per lavorare su diversi domini che offrono quindi varie funzionalità utili per le diverse necessità coinvolte nell'elaborazione di Big Data. Una di queste librerie è MLlib, che viene sfruttata all'interno del progetto per l'implementazione del classificatore Naive Bayes.
Apache Spark può essere eseguito sia in modalità standalone che in modalità cluster, ciò permette di testare il codice anche utilizzando un semplice computer prima di passare a lavorare su di un cluster vero e proprio. Questa possibilità è stata sfruttata durante la fase embrionale di sviluppo dell'applicazione.
Apache Spark necessita di un cluster manager e di un sistema di memorizzazione distribuito, supportando numerose opzioni. All'interno di

questo lavoro viene utilizzato Hadoop YARN come cluster manager e l'Hadoop Distributed File System, HDFS, come storage distribuito.

2.1.1 MLlib

Spark MLlib fa parte della serie di librerie aggiuntive comprese in Apache Spark e consiste in un framework per il machine-learning distribuito. All'interno di questa libreria sono disponibili la maggior parte dei più comuni algoritmi di machine-learning, tra i quali è stato sfruttato il classificatore Naive Bayes.

2.1.2 Hadoop YARN

Acronimo per Yet Another Resource Negotiator, YARN è un framework per la gestione di risorse per il calcolo distribuito e si occupa di assegnare le risorse disponibili alle varie applicazioni nella maniera più efficace possibile.
In un'architettura distribuita YARN si pone tra l'HDFS e il framework utilizzato per eseguire le applicazioni.[4] In questo caso l'utilizzo di Apache Spark con YARN fa sì che ogni esecutore Spark venga eseguito come un container YARN. YARN può assegnare risorse dinamicamente in caso di necessità, in modo da migliorare l'utilizzo delle stesse e le performance dell'applicazione.
YARN decentralizza l'esecuzione e il monitoraggio dei processi andando a ad assegnare a diverse componenti i vari compiti necessari. Un ResourceManager globale si occupa di accettare le sottomissioni dei job, li schedula e alloca loro risorse. Per ogni nodo è presente un NodeManager che si occupa del suo monitoraggio e comunica le informazioni al ResourceManager. Infine per ogni applicazione viene creato un ApplicationMaster che le permette di ottenere risorse, esso inoltre lavora insieme al NodeManager per eseguire e monitorare i vari task.

2.1.3 HDFS

L'Hadoop Distributed File System (HDFS) è un file system distribuito, scalabile e portabile scritto in Java per il framework Hadoop, che fornisce la possibilità di immagazzinare dati in modo distribuito. Basato sul Google File System, l'HDFS è un file system con struttura a blocchi in cui i singoli file sono memorizzati come blocchi di grandezza fissata e in cui blocchi dello stesso file non sono necessariamente nella stessa macchina. [4]

Uno dei grandi vantaggi dell'HDFS è la possibilità di sfruttare la data locality, i blocchi di dati vengono infatti assegnati in modo tale che ogni nodo del cluster lavori su dati direttamente presenti nella propria memoria locale, permettendo di velocizzare tutti i processi di elaborazione.

L'architettura è basata su un sistema master-slave in cui un master node, detto NameNode, gestisce il namespace e l'accesso degli slave ai file, mentre i molteplici slave corrispondono ognuno ad un DataNode, uno per ogni nodo del cluster. La tolleranza ai guasti è fornita grazie alla replicazione del NameNode in primis, oltre alla possibilità di impostare un parametro di replicazione anche per i singoli blocchi di file in base all'affidabilità voluta direttamente all'interno di un file di configurazione.

2.2 Scala

Scala è un linguaggio di programmazione general purpose multi-paradigma studiato per integrare le caratteristiche e funzionalità dei linguaggi orientati agli oggetti e di programmazione funzionale. Il codice sorgente di Scala viene compilata in Java bytecode per essere eseguito su di una Java Virtual Machine.[4]

Il linguaggio è staticamente tipizzato, e questo, insieme al fatto che Apache Spark è direttamente scritto in Scala, permette di ottenere i migliori risultati in termini di prestazioni.

2.3 Twitter API

Twitter offre a società, sviluppatori e utenti l'accesso programmatico ai suoi dati attraverso delle specifiche API (Application Programming Interfaces). Le API sono uno dei metodi più diffusi per ottenere informazioni attraverso l'utilizzo di endpoint che andranno a specificare il tipo di informazione ricercato.

La piattaforma API di Twitter fornisce l'accesso ai dati che gli utenti stessi scelgono di condividere pubblicamente, ulteriori dati sono invece disponibili solo qualora gli sviluppatori siano stati autorizzati direttamente dagli utenti. Nonostante siano quindi disponibili solo una parte dei dati contenuti all'interno di Twitter, quest'ultimo risulta molto più accessibile e disponibile alla ricerca rispetto ad altre piattaforme social che invece non consentono l'accesso e la fruizione di alcun tipo di dato in loro possesso.[5]

Le API di Twitter includono un'ampia gamma di endpoint, che rientrano in cinque gruppi principali, account e utenti, tweet e risposte, messaggi diretti, annunci e SDK. L'unica categoria di interesse per questo progetto è quella dei tweet che mette a disposizione i tweet pubblici degli utenti.

Il processo di utilizzo delle Twitter API prevede la creazione di un account sviluppatore dall'apposito sito e la creazione di un'applicazione che consente l'effettivo sfruttamento delle API. Il consenso per il loro utilizzo avviene solo dopo aver comunicato a Twitter i motivi per i quali si richiede tale servizio, l'uso che si andrà a fare dei dati ottenuti e lo scopo generale dell'applicazione che utilizzerà tali dati. Solo dopo essersi accertati che il tutto avviene secondo le norme e gli accordi per sviluppatori di Twitter, verrà fornita ed abilitata una

chiave di accesso con relativa chiave di sicurezza che consente effettivamente di inviare richieste e ottenere dati da Twitter.

3 - **Dettagli implementativi**

La soluzione proposta al problema della Sentiment Analysis è lo sviluppo di un'applicazione basata sul calcolo distribuito, che consenta di applicare un modello predittivo Naive Bayes su un dataset di grandi dimensioni contenente porzioni di testo, ossia tweet ottenuti direttamente da Twitter. L'analisi ha il fine di classificare ogni singolo tweet in base alla sua polarità, definendola come positiva, negativa o neutra.

Due elementi caratteristici di Apache Spark sono stati fondamentali per la gestione dei dati e in particolare per la loro distribuzione su tutto il cluster, l'RDD e il Dataframe.

L'RDD è un dataset distribuito resiliente, Resilient Distributed Dataset, ossia una collezione immutabile di elementi, distribuita sui diversi nodi del cluster, su cui possono essere effettuate operazioni in parallelo. Gli RDD sono creati a partire da un file sorgente e sono accessibili in sola lettura, di conseguenza non possono essere modificati, possono però subire delle trasformazioni per diventare RDD con nuove caratteristiche. È inoltre possibile rendere un RDD persistente in memoria in modo da poter essere riutilizzato in modo efficiente da più operazioni in parallelo. Caratteristica chiave di questo oggetto è il ripristino automatico in caso di guasto dei nodi, il che lo rende particolarmente adatto all'utilizzo in ambiente distribuito.

Un Dataframe è un Dataset, ossia una collezioni di dati distribuita, organizzato in colonne nominate, proprietà che lo differenzia dagli RDD. Concettualmente risulta equivalente ad una tabella all'interno di un database relazione ma con ottimizzazioni che ne rendono l'utilizzo più efficiente. Un Dataframe può essere

costruito partendo da una grande varietà di sorgenti dati come tabelle, database esterni o RDD già esistenti, a patto che i dati siano strutturati o semistrutturati.
Mentre un RDD offre funzionalità a basso livello e maggior controllo, un Dataframe porta l'astrazione ad un livello più alto consentendo l'utilizzo di operazioni specifiche di dominio, come query o accessi a colonne.
L'applicazione sviluppata fornisce, nel suo insieme, due output principali che sono il modello predittivo e il dataset classificato, per tale motivo è necessario il salvataggio dei dati su disco che prevede che i dati vengano prima serializzati. La serializzazione gioca un ruolo molto importante nelle performance di ogni applicazione distribuita dato che esse lavorano con file di grandi dimensioni per cui è necessaria questa operazione. Al fine di ottenere buone performance e ottimizzare il processo di scrittura, si è scelto di utilizzare il KryoSerializer il quale si comporta particolarmente bene e fornisce prestazioni elevate.

3.1 Fase preliminare

Tutte le dipendenze del progetto, con librerie e versioni software utilizzate, sono contenute all'interno del file *build.sbt*. Essendo Scala il linguaggio di programmazione utilizzato, è necessaria una fase di compilazione iniziale tramite il compilatore *sbt* che utilizzerà proprio questo file.
A causa dei diversi ambienti di esecuzione in cui il progetto è stato testato si è ritenuto opportuno utilizzare uno specifico file di configurazione, chiamato *application.conf*, in cui sono presenti una serie di proprietà configurabili come i diversi percorsi dei file necessari all'esecuzione. Il progetto, durante la fase di implementazione iniziale, è stato eseguito in modalità standalone, di conseguenza, i percorsi all'interno del file di configurazione sono stati impostati facenti riferimento al disco locale. Quando il tutto è stato spostato in ambiente

distribuito, i percorsi fanno invece riferimento direttamente alle posizioni dei file sull'HDFS, per permettere ad ogni nodo del cluster di ottenere i file.

3.2 File di input

I file di input necessari al funzionamento dell'applicazione sono diversi e si dividono in quelli usati per la fase di addestramento e il dataset su cui effettuare l'analisi. In entrambi i casi i file sono in formato *csv*, formato largamente utilizzato in ambito di machine-learning e ampiamente supportato sia da Spark che dalle sue librerie.

I dataset utilizzati per la fase di addestramento si compongono di una serie di file contenenti dati già etichettati che il modello predittivo può utilizzare per imparare a classificare i nuovi input che gli verranno sottoposti. In particolare all'interno dei file è presente una colonna contenente il testo dei tweet con una colonna corrispondente contenente la sua polarity sotto forma di intero. Per ottenere un modello predittivo di maggiore accuratezza si è ritenuto necessario l'utilizzo di più di un dataset per la fase di addestramento, in particolare sono stati utilizzati tre dataset differenti integrati tra loro.[6] La fase di preparazione dei dati necessaria per uniformarli è stata gestita direttamente all'interno dell'applicazione grazie alle funzioni fornite dal Dataframe di Spark.

3.3 Fase di addestramento

Prima dell'esecuzione della fase di analisi è necessaria la creazione del modello predittivo che andrà in seguito a classificare i nuovi tweet. L'implementazione del classificatore Naive Bayes, presente all'interno della libreria MLlib, permette di ricevere come parametro il dataset di addestramento che viene accompagnato da un file di stop world. Tale file contiene una serie di parole che

risultano ininfluenti ai fini della classificazione, come congiunzioni e preposizioni, che vengono conseguentemente ignorate dal classificatore poichè potrebbero compromettere la correttezza del risultato finale.
Il primo passo è la creazione di uno SparkContext, il principale entry point per le funzionalità di Spark, che permette la connessione ad un cluster Spark e la creazione di RDD e variabili condivise su tale cluster.
I dataset su cui sarà addestrato il modello vengono caricati ognuno all'interno di un Dataframe, questo permette di effettuare le operazioni di preparazione dei dati necessarie per renderli utilizzabili in seguito, i Dataset sono poi uniti per formarne uno singolo da cui generare un RDD.
Il classificatore Naive Bayes prende come input l'RDD di LabeledPoint generato, classe rappresentante le feature e le etichette dei dati, insieme ad un parametro di rifinitura, restituendo infine un oggetto NaiveBayesModel, che rappresenta le regole che verranno applicate dal modello predittivo. In questo contesto il LabeledPoint ha come come etichetta la polarità e come feature il testo del tweet corrispondente, quest'ultimo viene trasformato in vettore attraverso la classe HashingTF.
Dopo il completamento dell'addestramento il modello viene salvato all'interno del percorso specificato nel file di configurazione.
Infine grazie ad un dataset già etichettato si valuta l'accuratezza del modello generato andandolo ad eseguire su di esso e confrontando i valori generati con quelli già presenti per ottenere la percentuale di accuratezza del modello.

3.4 Fase di analisi

La classificazione vera e propria del dataset contenente i tweet avviene durante la fase di analisi, successivamente alla fase di addestramento.

Come primo passo il modello Naive Bayes generato viene prelevato dall'HDFS e caricato in memoria. Viene quindi creato un Dataframe prendendo in input il file contente i dati da classificare e così come nella fase di addestramento vengono sfruttati i suoi metodi per effettuare la preparazione dei dati per la classificazione.

Il Dataframe appena creato viene trasformato in un RDD per permettere la distribuzione del carico di lavoro sull'intero cluster da parte di Apache Spark, anche in questo caso in seguito all'istanziazione di uno SparkContext. Infine il dataset risultante, contenente la polarità generata dal classificatore, viene salvato sull'HDFS dopo essere stato compresso per ridurne le dimensioni.

Il risultato finale sarà quindi un file in formato *csv* compresso contenente il testo del tweet e la polarità corrispondente, calcolata dal modello predittivo. Tale polarità sarà un valore intero uguale a 0 per un tweet negativo, 4 per uno positivo e 2 per uno neutro.

4 - Esecuzione

La trattazione dell'esecuzione dell'applicazione vedrà come protagonista solamente la fase di analisi, questo poiché la precedente fase di addestramento non risulta critica per la valutazione delle prestazioni, oltre che essere di poca rilevanza per gli scopi del progetto in sé, che si propone di effettuare nel modo più efficiente e accurato possibile una Sentiment Analysis di un file di grandi dimensioni. La fase di addestramento infatti, risulta essere necessaria solo al primo avvio dell'applicazione al fine di generare il modello predittivo che sarà usato per la classificazione, in seguito non sarà più necessaria la sua esecuzione a meno di voler aggiornare il modello stesso.

La fase di analisi si compone di diversi stage di cui i primi risultano di basso interesse in quanto consistono solo nel caricamento in memoria del modello predittivo che dovrà essere applicato, perciò risultano di brevissima durata. Gli stage di maggior importanza sono lo stage 4 rappresentante la fase di Load e lo stage 5 rappresentante la fase di Save. Il primo effettua il caricamento del file di input da analizzare, mentre il secondo esegue la classificazione dei tweet e salva il file di output sull'HDFS.

I dati che saranno analizzati in questo paragrafo fanno riferimento all'esecuzione dell'applicazione su di un dataset di 57 Gb, precedentemente caricato sull'HDFS, utilizzando un cluster di 24 nodi. In seguito saranno descritti ed analizzati altri ambienti di esecuzione che non risultano particolarmente rilevanti per questa analisi. Attraverso l'utilizzo della WebUI fornita da Apache Spark, in particolare attraverso l'history-server è stato possibile ottenere i dettagli dei vari stage attraverso i log delle applicazioni.

4.1 Fase di Load

La fase di Load comprende il caricamento del file di input dall'HDFS in memoria per poi distribuirlo a tutti i nodi del cluster. Questa fase non è particolarmente dispendiosa sia in termini di risorse che di tempo, avendo una durata molto inferiore rispetto alla successiva fase di Save.

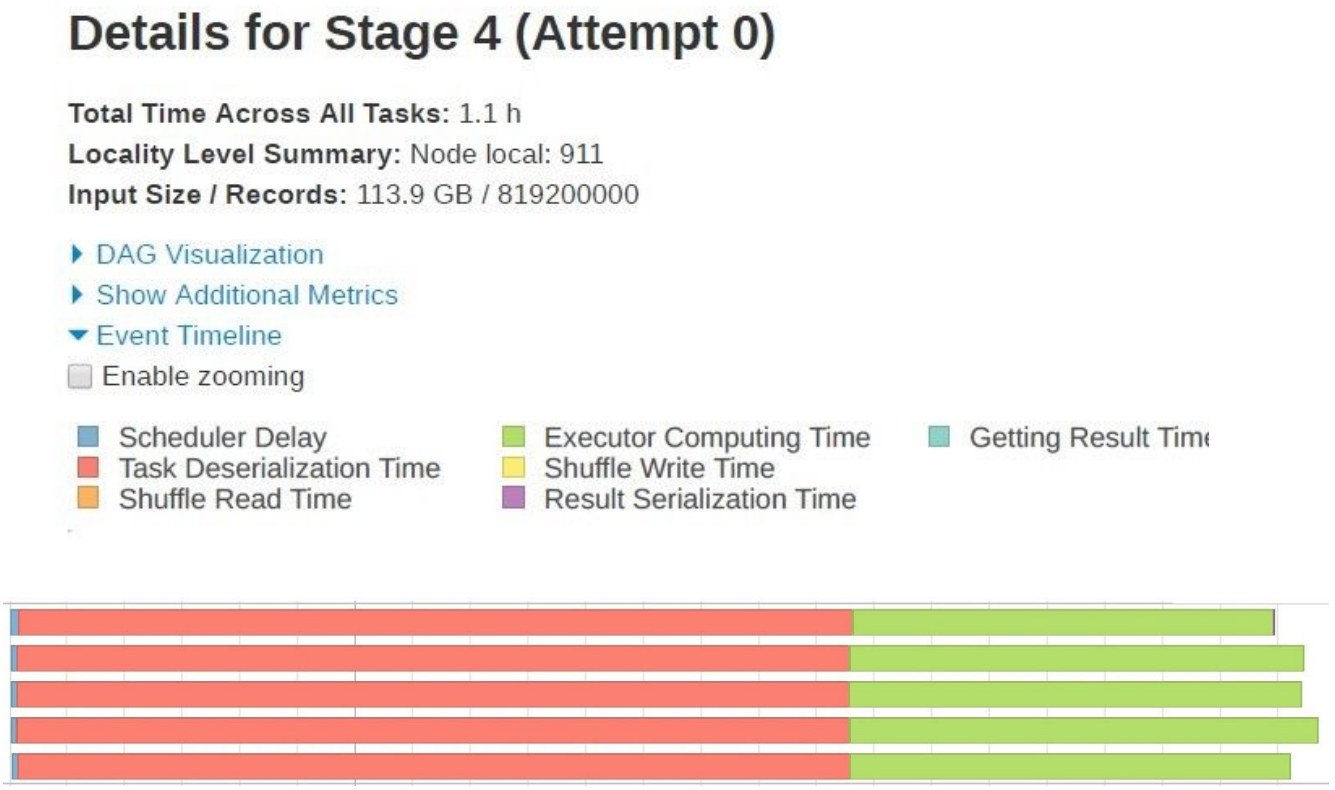


Immagine 1 - Stage 4, fase di Load

L'immagine 1 mostra come la maggior parte delle risorse siano utilizzate per la deserializzazione del file di input. Si noti come la dimensione dell'input sia due volte la grandezza del file in questione, di dimensione di 57 Gb, dovuto al fattore di replicazione impostato a 2 nel file di configurazione dell'HDFS.

4.2 Fase di Save

La fase di Save si può dividere in due sottofasi, una dedicata alla classificazione dei tweet e una alla scrittura del risultato sull'HDFS. La fase di Save risulta critica per l'intera applicazione, in quanto necessita della maggior potenza computazionale e allo stesso del maggior tempo di esecuzione.

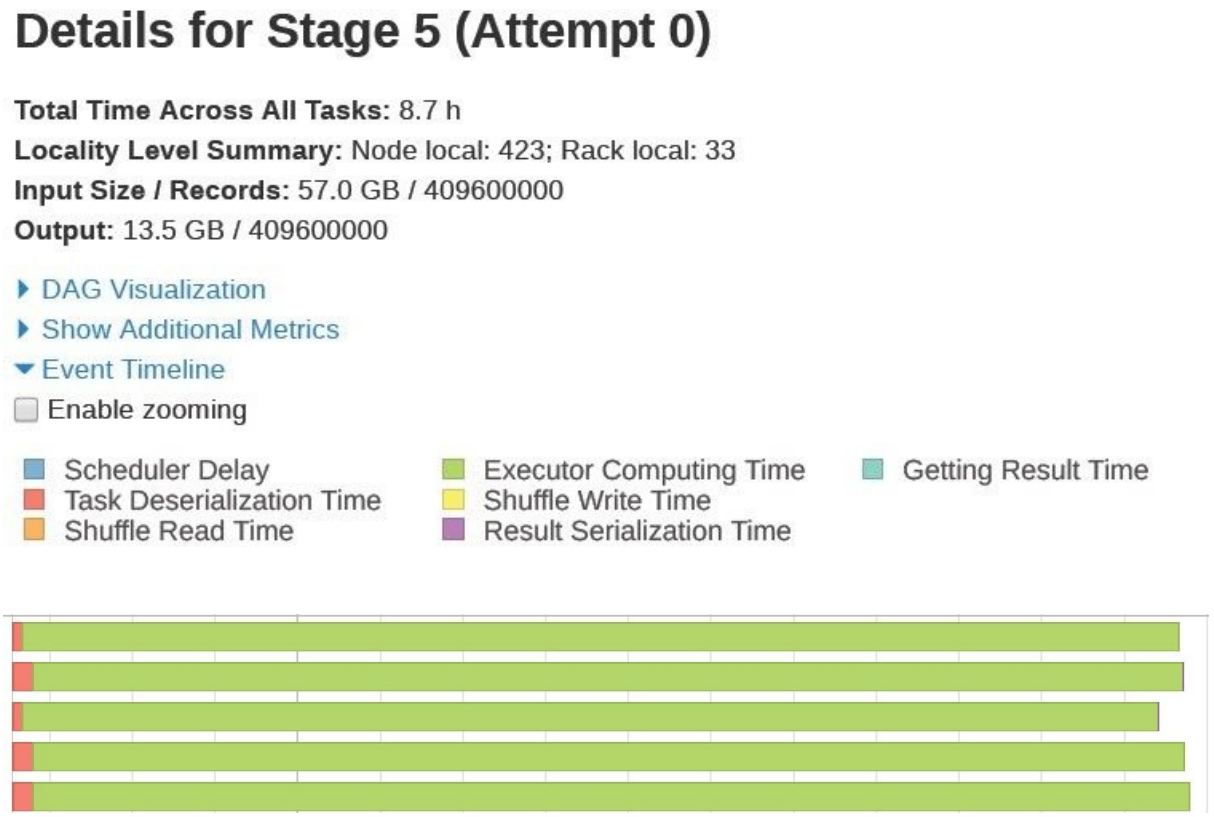


Immagine 2 - Stage 5, fase di Save

L'immagine 2 mostra come, a differenza dello stage precedente, la quasi totalità dell'esecuzione sia di tipo computazionale, proprio per l'elaborazione della polarity dei tweet. Altro dato significativo è la dimensione dell'input, che è pari alla dimensione del dataset da analizzare, e la dimensione dell'output, che risulta inferiore a quella del file di partenza. Questa riduzione di dimensioni è

dovuta alla pulizia delle colonne non utili ai fini della classificazione che viene effettuata dal Dataframe durante la fase di caricamento del dataset come descritto nei capitoli precedenti. Tutte le informazioni superflue vengono quindi eliminate e non sono presenti nel file di output che risulta di dimensione quattro volte inferiore al file di input.

4.3 Utilizzo delle risorse

Attraverso lo strumento di monitoraggio *dstat* è stato possibile ricavare l'utilizzo di CPU e memoria durante tutta l'esecuzione, ottenendo così la percentuale di utilizzo del processore e la quantità di memoria in Gigabyte per ogni secondo dell'intera durata dell'applicazione. I dati ottenuti sono stati poi elaborati per ottenere i grafici mostrati di seguito.

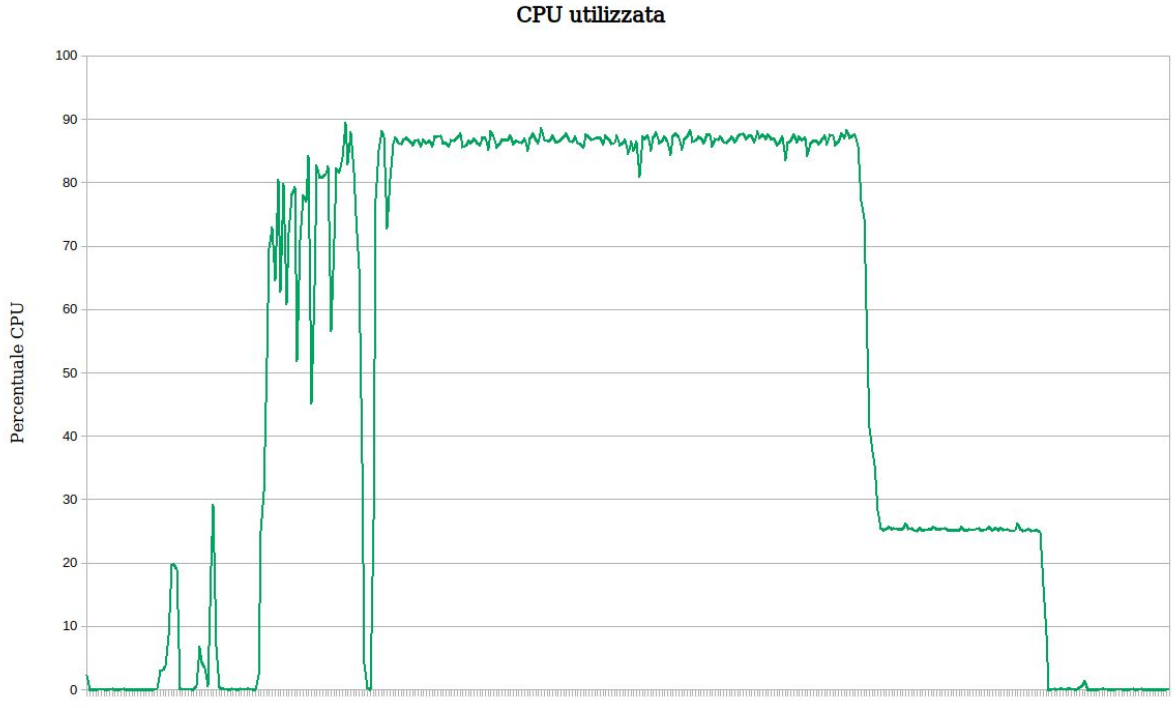


Immagine 3 - Percentuale di utilizzo della CPU durante l'esecuzione.

L’immagine 3 mostra come la CPU venga particolarmente stressata dall’applicazione soprattutto nella seconda parte dell’esecuzione. Partendo dal processore completamente in idle i primi picchi di utilizzo, oltre il 70%, iniziano al partire della fase di Load dove è richiesta una certa potenza di calcolo per la deserializzazione dell’input.

Al concludersi della fase di Load vi è un piccolo periodo in cui il processore va in idle a causa del passaggio alla fase successiva, ovvero la fase di Save. Durante quest’ultima l’utilizzo della CPU è costantemente sopra l’80% proprio per la necessità di elaborare il gran numero di tweet ed effettuarne la classificazione. In seguito la percentuale scende in modo consistente in corrispondenza del passaggio dalla fase di classificazione alla fase di scrittura sull’HDFS.

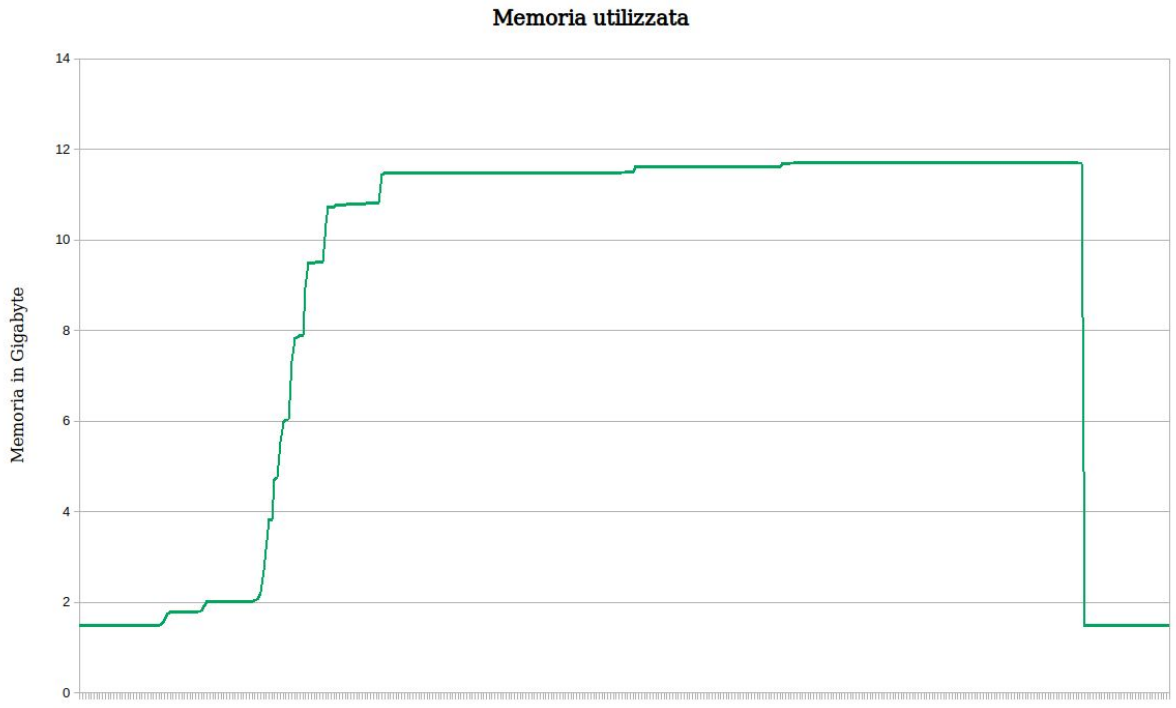


Immagine 4 - Quantità di memoria utilizzata durante l’esecuzione.

Analogamente l'immagine 4 mostra l'utilizzo della memoria durante l'intera esecuzione, in questo caso la quantità utilizzata va crescendo durante la fase di Load, raggiungendo poi il suo picco al termine di essa e mantenendosi costante fino alla conclusione dell'intera elaborazione.

5 - Test e valutazioni

I test sono stati effettuati utilizzando un cluster montato sulla rete GARR, acronimo di Gruppo per l'Armonizzazione delle Reti della Ricerca, la rete italiana a banda ultralarga dedicata alla comunità dell'istruzione, della ricerca e della cultura. L'accesso al cluster veniva effettuato attraverso il protocollo *ssh* grazie ad una chiave di sicurezza.

Sono stati necessari diversi test per valutare le prestazioni dell'applicazione e le eventuali modifiche da apportare al codice. Inoltre ulteriori test sono stati necessari per la ricerca della configurazione ottimale in termini di prestazioni.

Per ottenere risultati quanto più veritieri possibili ad ogni test è stata eseguita la procedura di formattazione dell'HDFS con conseguente riavvio dei servizi e caricamento dei file necessari all'applicazione.

5.1 Configurazione cluster

- ➔ **Numero nodi:** 24
- ➔ **Hardware:** 8 CPU - 30 Gb RAM
- ➔ **Sistema operativo:** Linux
- ➔ **Ambiente software:**
  - ◆ Apache Spark v. 2.3.3
  - ◆ Java v. 1.8
  - ◆ Scala v. 2.11

5.2 Configurazione software

- ➔ **Spark serializer:** KryoSerializer

➔ **Spark master:** YARN

➔ **DFS replication:** 2

➔ **DFS blocksize:** 128 Mb

➔ **YARN nodemanager resource**

- ◆ **memory:** 30 Gb
- ◆ **cpu-vcores:** 8

➔ **YARN scheduler**

- ◆ **minimum allocation:** 3.8 Gb
- ◆ **maximum allocation:** 30 Gb

5.3 Configurazione esecuzione distribuita

Apache Spark prevede una serie di parametri addizionali che possono essere configurati per esecuzione delle applicazioni, per tale motivo sono stati necessari una prima serie di test per trovare la configurazione ottimale. I parametri in questione sono rappresentati dal numero di esecutori da eseguire e il numero di core e la quantità di memoria assegnata ad ognuno di essi.

Partendo dal numero di core per esecutore si è proceduto a calcolare le restanti impostazioni. Tutti i calcoli sono effettuati ragionando per difetto e lasciando sempre una parte di risorse libere per l'overhead di esecuzione dei servizi di Apache Spark e dell'HDFS. Ciò è stato necessario in quanto utilizzando l'intero pool di risorse si sono riscontrati una serie problemi.

Tutti i test effettuati in questo paragrafo prevedono un file di input di 57 Gb.

5.3.1 Test configurazione Apache Spark 1

Il primo test ha come obiettivo la saturazione delle risorse del cluster andando verso una delle due configurazioni estreme, in questo caso quella con il minor numero di esecutori equipaggiati con il massimo delle risorse disponibili del singolo nodo.

Riservando 1 core di ogni nodo per il daemon di YARN il cluster ha a disposizione 7 core * 24 nodi per un totale di 168 core.

Assegnando il massimo numero di core a disposizione di un singolo nodo per executor, ossia 7, si potranno istanziare 24 executor.

Su ogni nodo sarà quindi in esecuzione 1 singolo executor con assegnate tutte le risorse disponibili di quel nodo.

Andando a riservare il 7% della memoria per l'overhead richiesto da Apache Spark si ottiene un totale di 27 Gb assegnabili ad ogni executor.

Riassumendo questa configurazione presenta 24 executor con 7 core e 27 Gb di memoria, per un tempo di esecuzione totale di 5 minuti e 42 secondi.

5.3.2 Test configurazione Apache Spark 2

Il secondo test aveva come obiettivo la saturazione delle risorse del cluster andando verso l'altro lato delle due configurazioni estreme, in questo caso quella con il maggior numero di esecutori.

Riservando 1 core di ogni nodo per il daemon di YARN il cluster ha a disposizione 7 core * 24 nodi per un totale di 168 core.

Assegnando 1 core per executor si potranno quindi istanziare 168 executor.

Su ogni nodo saranno quindi in esecuzione 168 / 24 executor , per un risultato di 7 executor.

Avendo ogni nodo 30 Gb di memoria si ottiene un valore di 30 Gb / 7 executor, ossia 4 Gb di memoria per executor. Da questo risultato bisogna escludere il 7% rappresentante l'overhead dell'heap per ottenere circa 3 Gb di memoria per executor.
Riassumendo questa configurazione presenta 168 executor con 1 core e 3 Gb di memoria, per un tempo di esecuzione totale di 13 minuti e 24 secondi.

5.3.3 Test configurazione Apache Spark 3

Il terzo test ha come obiettivo l'ottimizzazione delle risorse, andando ad utilizzare 5 core per executor, valore che risulta il range superiore di quello indicato come un buon compromesso per il throughput dell'HDFS.
Riservando 1 core di ogni nodo per il daemon di YARN il cluster ha a disposizione 7 core * 24 nodi per un totale di 168 core.
Assegnando 5 core per executor si potranno quindi istanziare 168 / 5 executor, per un totale di 33 executor.
Su ogni nodo saranno quindi in esecuzione 33 / 24 executor , per un risultato di 1 executor circa per nodo.
Avendo ogni nodo 30 Gb di memoria ogni executor potrà avere 30 Gb di memoria. Da questo valore bisogna escludere il 7% rappresentante l'overhead dell'heap per ottenere circa 27 Gb di memoria per executor.
Riassumendo questa configurazione presenta 33 executor con 5 core e 27 Gb di memoria, per un tempo di esecuzione totale di 6 minuti e 30 secondi.

5.3.4 Test configurazione Apache Spark 4

Il quarto test, anch'esso con l'obiettivo di ottimizzare le risorse utilizzate, assegna 3 core per executor, valore che risulta il range inferiore di quello indicato come un buon compromesso per il throughput dell'HDFS.

Riservando 1 core di ogni nodo per il daemon di YARN il cluster ha a disposizione 7 core * 24 nodi per un totale di 168 core.
Assegnando 3 core per executor si potranno quindi istanziare 168 / 3 executor, per un totale di 55 executor.
Su ogni nodo saranno quindi in esecuzione 55 / 24 executor , per un risultato di 2 executor per nodo.
Avendo ogni nodo 30 Gb di memoria ogni executor la metà della memoria rappresentato da 15 Gb. Da questo valore bisogna escludere il 7% rappresentante l'overhead dell'heap per ottenere circa 13 Gb di memoria per executor.
Riassumendo questa configurazione presenta 55 executor con 3 core e 13 Gb di memoria, per un tempo di esecuzione totale di 9 minuti e 18 secondi.

5.3.5 Configurazione Apache Spark ottimale

Dai test effettuati si è concluso che la configurazione che permette di ottenere il miglior risultato in termini di tempo di esecuzione risulta essere la prima, costituita da 24 executor con 7 core e 27 Gb di memoria.

5.4 Benchmark

Una serie di benchmark sono stati effettuati per valutare l'incremento o la diminuzione del tempo di esecuzione al variare sia dei nodi utilizzati sia della dimensione del file di input, ottenendo rispettivamente speedup e sizeup. Questa tipologia di benchmark ha lo scopo di valutare la scalabilità dell'applicazione sviluppata, una caratteristica fondamentale per un sistema di calcolo distribuito.
La configurazione Spark utilizzata è quella che, nella fase di test precedente, ha prodotto il risultato migliore. Per uniformare i test si è quindi utilizzato un

numero di esecutori pari al numero dei nodi, assegnando a ciascuno di essi tutte le risorse disponibili del nodo, considerando l'overhead necessari al sistema.

Configurazione	**Executor**	**Executor-core**	**Executor-memory**
24 nodi	24	7	27 Gb
16 nodi	16	7	27 Gb
8 nodi	8	7	27 Gb
4 nodi	4	7	27 Gb

Tabella 1 - Configurazione Apache Spark utilizzata per i benchmark.

5.4.1 Speedup

La speedup è un numero che misura le prestazioni relative a due sistemi che elaborano lo stesso problema. Tecnicamente, la speedup, è il miglioramento nella velocità di esecuzione di un task eseguito su due architetture quanto più simili possibili ma con risorse differenti. La nozione di speedup fu stabilita dalla legge di Amdahl, legge fondamentale del calcolo distribuito, ma essa può essere applicata più genericamente per mostrare gli effetti sulle prestazioni dopo un qualsiasi incremento delle risorse. Banalmente quindi la speedup dimostra come aggiungendo più risorse ad un sistema, che siano processori, memoria e così via, esso sia in grado di risolvere un problema di una data dimensione più velocemente.

La speedup calcolata è ottenuta al variare del numero dei nodi, mantenendo costanti le restanti variabili, in particolare si è partiti utilizzando 4 nodi andando a raddoppiare il numero fino a 16 per poi utilizzare a pieno le risorse del cluster con 24 nodi.

Test	Numero nodi	Tempo di esecuzione
#1	4 nodi	31 minuti
#2	8 nodi	14 minuti
#3	16 nodi	7 minuti 12 secondi
#4	24 nodi	5 minuti 42 secondi

Tabella 2 - Speedup su file di input di 57 Gb.

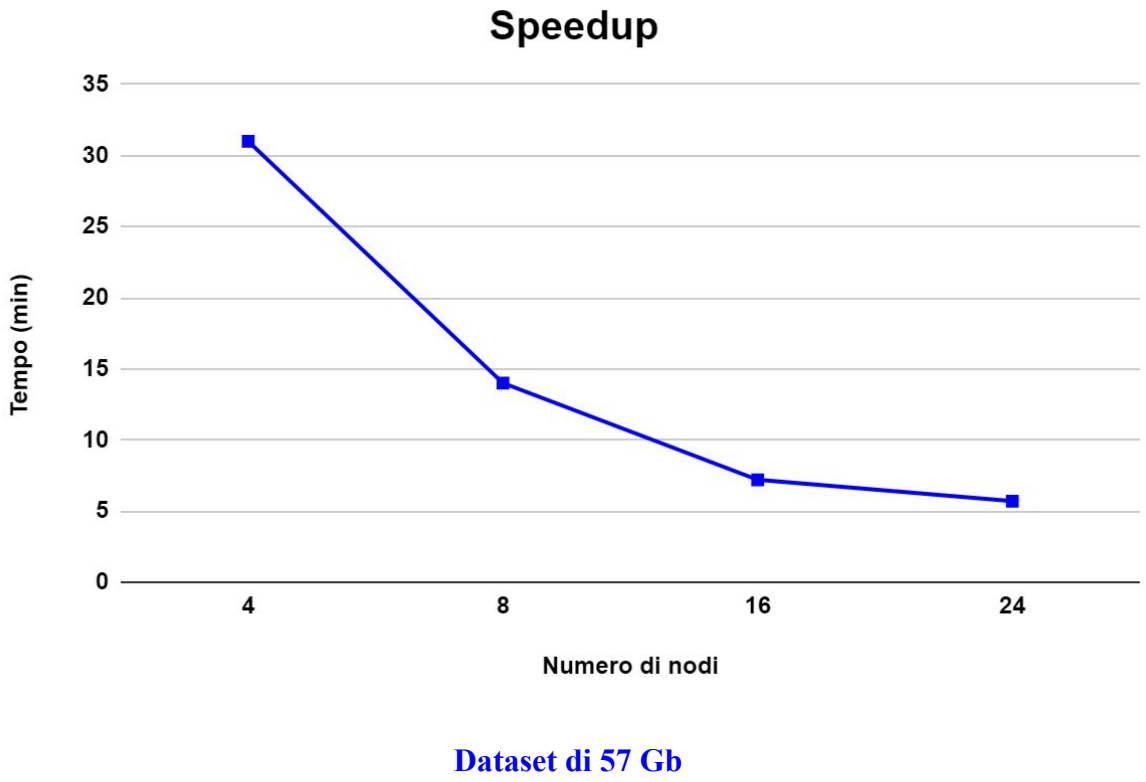


Immagine 5 - Speedup ottenuta al variare del numero dei nodi.

I risultati del benchmark sul file di input di 57 Gb mostrano una speedup che risulta lineare, ottenendo un miglioramento sul tempo di esecuzione proporzionale all'incremento del numero dei nodi. I test #1, #2, #3 mostrano infatti un dimezzamento dei tempi di esecuzione al raddoppio dei nodi utilizzati,

mentre il test #4 mostra come anche aumentando del 25% il numero dei nodi il tempo si riduca della stessa percentuale.

Test	Numero nodi	Tempo di esecuzione
#1	4 nodi	14 minuti
#2	8 nodi	7 minuti 24 secondi
#3	16 nodi	5 minuti 6 secondi
#4	24 nodi	3 minuti 42 secondi

Tabella 3 - Speedup su file di input di 28.5 Gb.

Test	Numero nodi	Tempo di esecuzione
#1	4 nodi	3 minuti 12 secondi
#2	8 nodi	2 minuti 6 secondi
#3	16 nodi	1 minuto 36 secondi
#4	24 nodi	1 minuto 18 secondi

Tabella 4 - Speedup su file di input di 3.4 Gb.

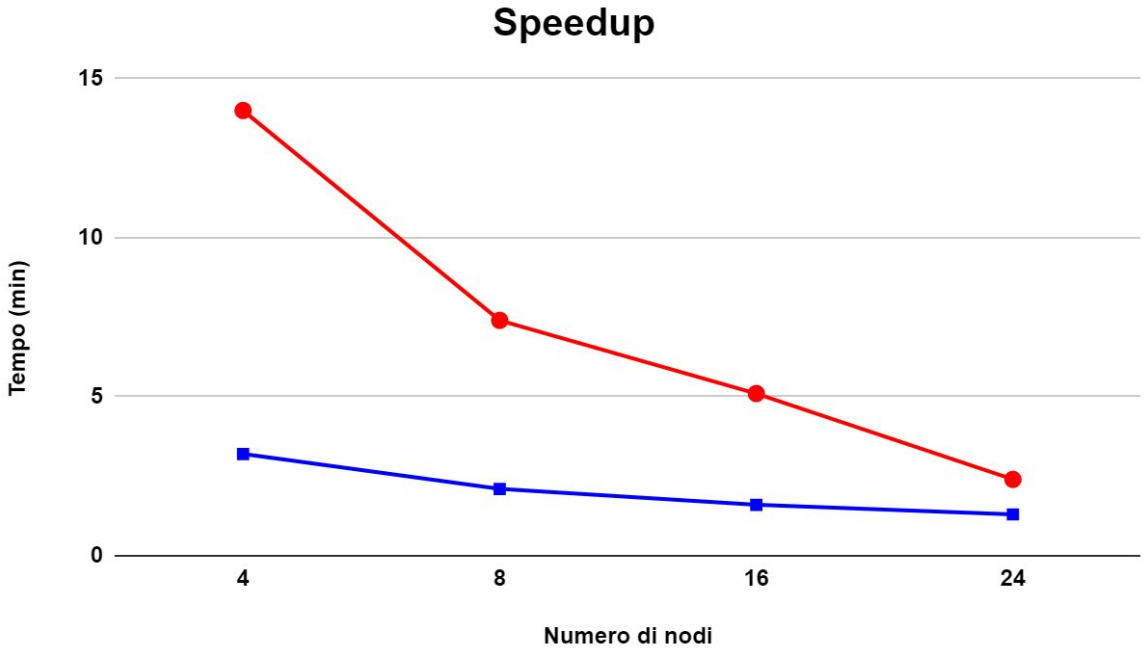


Immagine 6 - Speedup ottenuta al variare del numero dei nodi.

I benchmark per la speedup sono stati replicati anche sui dataset di minori dimensioni. Se con il dataset di 28.5 Gb si possono trarre le medesime conclusioni del precedente test, lo stesso non vale per il dataset di 3.4 Gb. In questo caso infatti, la riduzione del tempo di esecuzione non è proporzionale all'aumento delle risorse, ossia del numero dei nodi. Questo è probabilmente determinato dall'eccessivo overhead, applicato da Apache Spark per la gestione di un sistema distribuito e dall'HDFS, che non vale la pena pagare per un file di tali dimensioni, evidentemente troppo piccole.

5.4.2 Sizeup

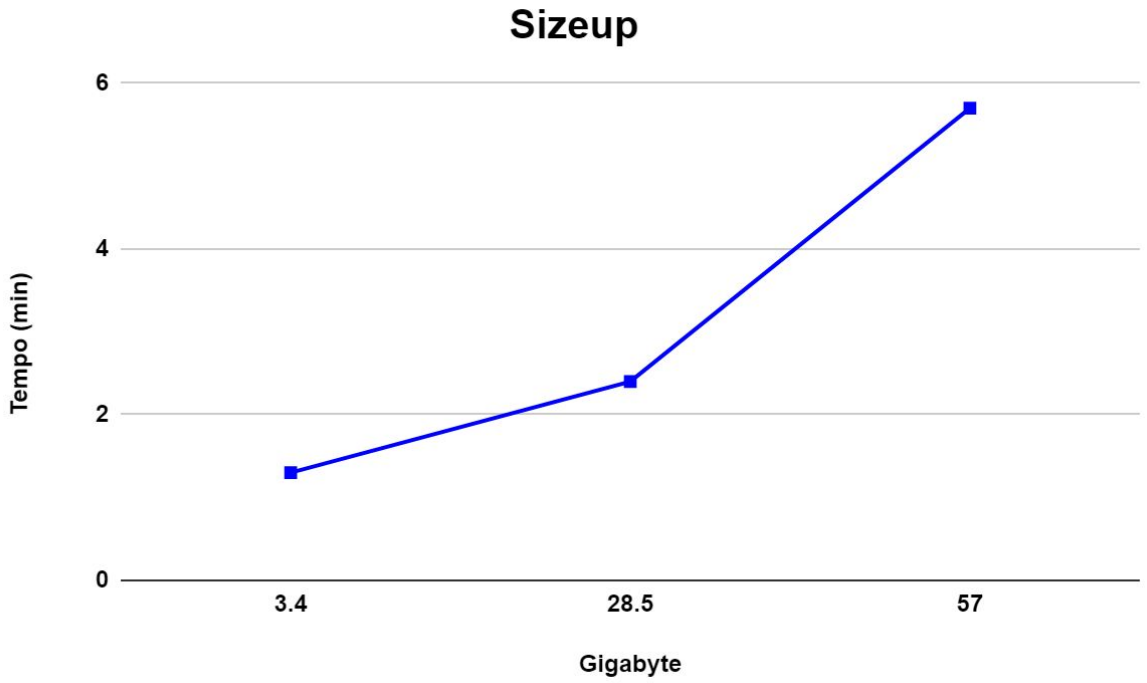
La sizeup è un numero che misura le prestazioni relative all'elaborazione di un problema di differenti dimensioni processato dallo stesso sistema. Tecnicamente, la sizeup, è la possibilità di aumentare la complessità di un task

proporzionalmente all'aumento delle risorse disponibili mantenendo un tempo di esecuzione simile. Banalmente quindi la sizeup dimostra come aggiungendo più risorse ad un sistema, che siano processori, memoria e così via, esso sia in grado di risolvere un problema di dimensione, o complessità, maggiore in un dato periodo di tempo .

La sizeup calcolata è ottenuta al variare della dimensione del file di input, mantenendo costanti le restanti variabili, in particolare il dataset di dimensioni maggiori è di 57 Gb, il dataset intermedio è 2 volte più piccolo del primo, quindi di 28.5 Gb, infine il dataset di dimensioni minori è di 3.4 Gb, 8 volte più piccolo del dataset intermedio e 17 volte più piccolo del dataset maggiore.

Test	**Dimensione dataset**	**Tempo di esecuzione**
#1	57 Gb	5 minuti 42 secondi
#2	28.5 Gb	3 minuti 42 secondi
#3	3.4 Gb	1 minuto 18 secondi

Tabella 5 - Sizeup con 24 nodi.



Esecuzione con 24 nodi

Immagine 7 - Sizeup ottenuta al variare della dimensione del dataset.

I risultati del benchmark effettuati utilizzando 24 nodi mostrano una sizeup non lineare, infatti il miglioramento ottenuto con l'aumento delle dimensioni del file di input non è proporzionale con l'incremento dei tempi di esecuzione. Avendo ottenuto tale risultato si presume che, l'utilizzo di 24 nodi, permetta la gestione di dataset di dimensioni anche molto maggiori a quelle utilizzate.

Test	**Dimensione dataset**	**Tempo di esecuzione**
#1	57 Gb	7 minuti 12 secondi
#2	28.5 Gb	5 minuti 6 secondi
#3	3.4 Gb	1 minuto 36 secondi

Tabella 6 - Sizeup con 16 nodi.

Test	Dimensione dataset	Tempo di esecuzione
#1	57 Gb	14 minuti
#2	28.5 Gb	7 minuti 24 secondi
#3	3.4 Gb	2 minuti 6 secondi

Tabella 7 - Sizeup con 8 nodi.

Test	Dimensione dataset	Tempo di esecuzione
#1	57 Gb	31 minuti
#2	28.5 Gb	14 minuti
#3	3.4 Gb	3 minuti 12 secondi

Tabella 8 - Sizeup con 4 nodi.

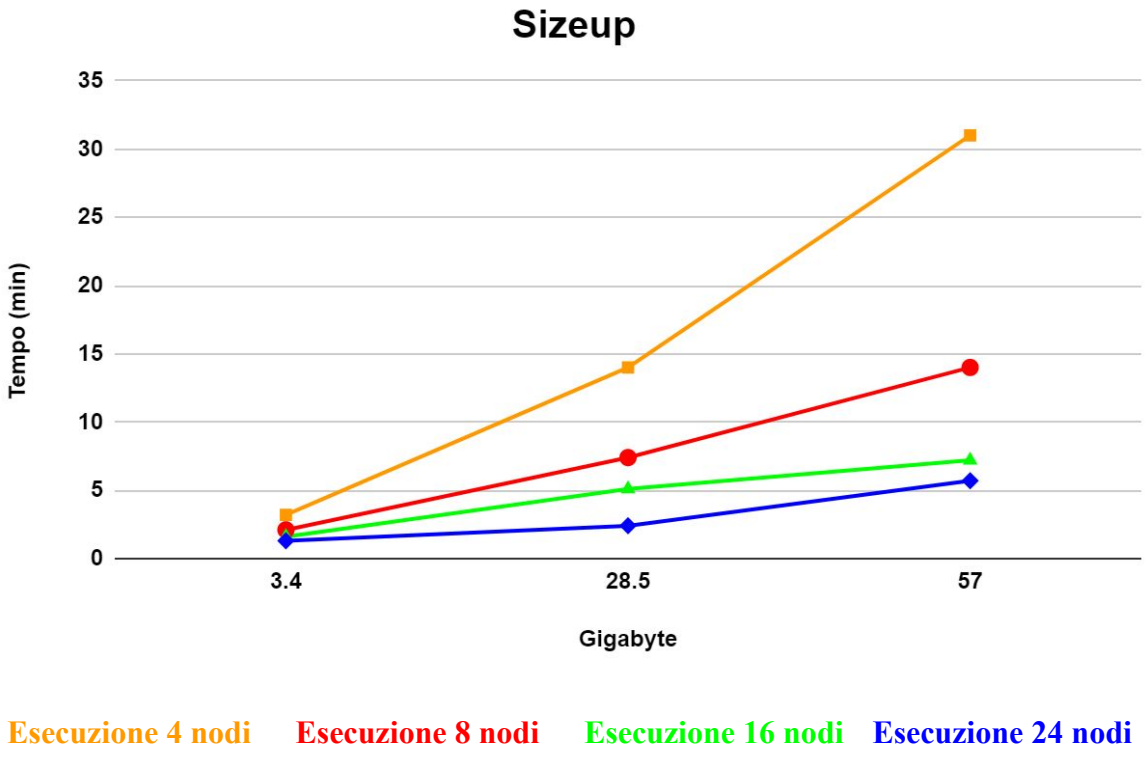


Immagine 8 - Sizeup ottenuta al variare della dimensione del dataset.

I benchmark per la sizeup sono stati replicati anche utilizzando configurazioni del cluster con un minor numero di nodi. Se utilizzando 16 nodi, o paragonando i risultati ottenuti dai test #3, che utilizzano il dataset di dimensioni minori, si possono trarre le medesime conclusioni del precedente test, lo stesso non vale per le configurazioni da 8 e 4 nodi, in particolare nei rispettivi test #1 e #2. In questo caso infatti, l'aumento del tempo di esecuzione diviene proporzionale all'aumento della dimensione del file di input, come previsto.

6 - **Conclusioni**

Il lavoro presentato ha voluto dimostrare come determinati problemi possano essere risolti in maniera più efficace ed efficiente attraverso l'utilizzo di sistemi distribuiti. Nonostante la difficoltà intrinseca nella gestione di sistemi di questo tipo, grazie alla tecnologia fornita da Apache Spark le operazioni necessarie risultano ampiamente semplificate e in alcuni casi del tutto automatizzate o gestite dal software stesso. Non si voglia negare la necessità di conoscere i paradigmi caratterizzanti le architetture distribuite, Apache Spark infatti richiede comunque una serie di configurazioni iniziali che vanno ritoccate in corso d'opera, ma si può affermare che questo strumento porti una serie di vantaggi importanti a cui difficilmente un programmatore può rinunciare.

Molti aspetti legati al problema affrontato hanno trovato una rapida soluzione grazie alle funzionalità offerte da Apache Spark e dalle sue librerie, come la gestione delle sorgenti di dati, che comprende la loro pulizia e preparazione per l'utilizzo effettivo, e l'implementazione di algoritmi di machine-learning.

Nonostante ciò non sono mancati i problemi principalmente derivati dall'utilizzo di un cluster sulla rete fornita dal GARR. A causa di problematiche di rete interna ai nodi è stata necessaria un'interruzione del lavoro per cercare soluzione ad essi, in particolare problemi di connessione a specifici nodi ha reso inutilizzabile l'HDFS.

Concludendo i test hanno dimostrato come l'approccio utilizzato risulti di valido e scalabile, rendendo soddisfacente l'intero progetto.

Appendice

A.1 Fase di addestramento

```
object SparkNaiveBayesModelCreator {

  def main(args: Array[String]) {
    val sc = createSparkContext()
    val stopWordsList = sc.broadcast(StopwordsLoader.loadStopWords(PropertiesLoader.nltkStopWords))
    createAndSaveNBModel(sc, stopWordsList)
    validateAccuracyOfNBModel(sc, stopWordsList)
  }

  def replaceNewLines(tweetText: String): String = {
    tweetText.replaceAll("\n", "")
  }

  def createSparkContext(): SparkContext = {
    val conf = new SparkConf()
      .setAppName("SentimentAnalysis Twitter")
      .set("spark.serializer", classOf[KryoSerializer].getCanonicalName)
    val sc = SparkContext.getOrCreate(conf)
    sc
  }

  def createAndSaveNBModel(sc: SparkContext, stopWordsList: Broadcast[List[String]]): Unit = {
    val tweetsDF: DataFrame = loadSentiment140File(sc, PropertiesLoader.sentiment140TrainingFilePath)
    val result = tweetsDF.filter("polarity == '0'")
    val tweetsDF2: DataFrame = loadSentiment140FileAmerica(sc, PropertiesLoader.sentiment140TrainingFilePathAmerica)
    val tweetsT4saTextRaw: DataFrame = loadT4saFileTextRaw(sc,PropertiesLoader.trainingFileT4saRawTweetsSentiment)
    val tweetsT4saTextSentiment: DataFrame = loadT4saFileTextSentiment(sc,PropertiesLoader.trainingFileT4saTextSentiment)
    val dfrawtweets = tweetsT4saTextRaw.as("dfrawtweets")
    val dftextsentiment = tweetsT4saTextSentiment.as("dftextsentiment")
    val joined_dft4sa = dftextsentiment.join(dfrawtweets,col("dftextsentiment.user_id") === col("dfrawtweets.user_id"), "inner")
    val joined_dft4sa_dropped = joined_dft4sa.drop("user_id")
    val tweetsDFUnion = tweetsDF.union(tweetsDF2).union(joined_dft4sa_dropped)

    val labeledRDD =tweetsDFUnion.select("polarity", "status").rdd.map {
      case Row(polarity: Int, tweet: String) =>
        val tweetInWords: Seq[String] = MLlibSentimentAnalyzer.getBarebonesTweetText(tweet, stopWordsList.value)
        LabeledPoint(polarity, MLlibSentimentAnalyzer.transformFeatures(tweetInWords))
    }
    labeledRDD.cache()

    val naiveBayesModel: NaiveBayesModel = NaiveBayes.train(labeledRDD, lambda = 1.0, modelType = "multinomial")
    naiveBayesModel.save(sc, PropertiesLoader.naiveBayesModelPath)
  }
```

```
def validateAccuracyOfNBModel(sc: SparkContext, stopWordsList: Broadcast[List[String]]): Unit = {
  val naiveBayesModel: NaiveBayesModel = NaiveBayesModel.load(sc, PropertiesLoader.naiveBayesModelPath)
  val tweetsDF: DataFrame = loadSentiment140File(sc, PropertiesLoader.sentiment140TestingFilePath)
  val actualVsPredictionRDD = tweetsDF.select("polarity", "status").rdd.map {
    case Row(polarity: Int, tweet: String) =>
      val tweetText = replaceNewLines(tweet)
      val tweetInWords: Seq[String] = MLlibSentimentAnalyzer.getBarebonesTweetText(tweetText, stopWordsList.v
      (polarity.toDouble,
        naiveBayesModel.predict(MLlibSentimentAnalyzer.transformFeatures(tweetInWords)),
        tweetText)
  }
  val accuracy = 100.0 * actualVsPredictionRDD.filter(x => x._1 == x._2).count() / tweetsDF.count()
  println(f"""\n\t<==******** Prediction accuracy compared to actual: $accuracy%.2f%% ********==>\n""")
  saveAccuracy(sc, actualVsPredictionRDD)
}

def loadSentiment140File(sc: SparkContext, sentiment140FilePath: String): DataFrame = {
  val sqlContext = SQLContextSingleton.getInstance(sc)
  val tweetsDF = sqlContext.read
    .format("com.databricks.spark.csv")
    .option("header", "false")
    .option("inferSchema", "true")
    .load(sentiment140FilePath)
    .toDF("polarity", "id","date","query","user","status")
  tweetsDF.drop("id").drop("date").drop("query").drop("user")
}

def loadSentiment140FileAmerica(sc: SparkContext, sentiment140FilePathAmerica: String): DataFrame = {
  val sqlContext = SQLContextSingleton.getInstance(sc)
  val tweetsDF = sqlContext.read
    .format("com.databricks.spark.csv")
    .option("header", "false")
    .option("inferSchema", "true")
    .load(sentiment140FilePathAmerica)
    .toDF("polarity", "status")
  tweetsDF
}
```

```
def loadT4saFileTextRaw(sc: SparkContext, sentiment140FileT4saTextRaw: String): DataFrame = {
   val sqlContext = SQLContextSingleton.getInstance(sc)
   val tweetsDF = sqlContext.read
     .format("com.databricks.spark.csv")
     .option("header", "true")
     .option("inferSchema", "true")
     .load(sentiment140FileT4saTextRaw)
     .toDF("user_id", "status")
   tweetsDF
 }

def loadT4saFileTextSentiment(sc: SparkContext, sentimentFileT4saText: String): DataFrame = {
   val sqlContext = SQLContextSingleton.getInstance(sc)
   val tweetsDF = sqlContext.read
     .format("com.databricks.spark.csv")
     .option("header", "true")
     .option("inferSchema", "true")
     .load(sentimentFileT4saText)
     .toDF("polarity", "user_id")
   tweetsDF
 }

 def saveAccuracy(sc: SparkContext, actualVsPredictionRDD: RDD[(Double, Double, String)]): Unit = {
   val sqlContext = SQLContextSingleton.getInstance(sc)
   import sqlContext.implicits._
   val actualVsPredictionDF = actualVsPredictionRDD.toDF("Actual", "Predicted", "Text")
   actualVsPredictionDF.coalesce(1).write
     .format("com.databricks.spark.csv")
     .option("header", "true")
     .option("delimiter", "\t")
     .option("codec", classOf[GzipCodec].getCanonicalName)
     .mode(SaveMode.Append)
     .save(PropertiesLoader.modelAccuracyPath)
 }
}
```

```scala
object MLlibSentimentAnalyzer {

  def computeSentiment(text: String, stopWordsList: Broadcast[List[String]], model: NaiveBayesModel): Int = {
    val tweetInWords: Seq[String] = getBarebonesTweetText(text, stopWordsList.value)
    val polarity = model.predict(MLlibSentimentAnalyzer.transformFeatures(tweetInWords))
    normalizeMLlibSentiment(polarity)
  }

  def normalizeMLlibSentiment(sentiment: Double) = {
    sentiment match {
      case x if x == 0 => -1 // negativo
      case x if x == 2 => 0 // neutro
      case x if x == 4 => 1 // positivo
      case _ => 0 // neutro se non può calcolare il sentiment
    }
  }

  def getBarebonesTweetText(tweetText: String, stopWordsList: List[String]): Seq[String] = {
    tweetText.toLowerCase()
      .replaceAll("\n", "")
      .replaceAll("rt\\s+", "")
      .replaceAll("\\s+@\\w+", "")
      .replaceAll("@\\w+", "")
      .replaceAll("\\s+#\\w+", "")
      .replaceAll("#\\w+", "")
      .replaceAll("(?:https?|http?)://[\\w/%.-]+", "")
      .replaceAll("(?:https?|http?)://[\\w/%.-]+\\s+", "")
      .replaceAll("(?:https?|http?)//[\\w/%.-]+\\s+", "")
      .replaceAll("(?:https?|http?)//[\\w/%.-]+", "")
      .split("\\W+")
      .filter(_.matches("^[a-zA-Z]+$"))
      .filter(!stopWordsList.contains(_))
  }

  val hashingTF = new HashingTF()

  def transformFeatures(tweetText: Seq[String]): Vector = {
    hashingTF.transform(tweetText)
  }
}
```

A.2 Fase di analisi

```
object BatchAnalyzer {

  def main(args: Array[String]) {
    val sc = createSparkContext()
    LogUtils.setLogLevels(sc)
    val naiveBayesModel: NaiveBayesModel = NaiveBayesModel.load(sc, PropertiesLoader.naiveBayesModelPath)
    Console.println("NaiveBayesModel Caricato");
    val stopWordsList = sc.broadcast(StopwordsLoader.loadStopWords(PropertiesLoader.nltkStopWords))
    batchAndSaveAnalyze(sc, stopWordsList, naiveBayesModel)
  }

  def createSparkContext(): SparkContext = {
    val conf = new SparkConf()
      .setAppName("BatchAnalyzer Twitter Sentiment")
      .set("spark.serializer", classOf[KryoSerializer].getCanonicalName)
    val sc = SparkContext.getOrCreate(conf)
    sc
  }

  def batchAndSaveAnalyze(sc: SparkContext, stopWordsList: Broadcast[List[String]], naiveBayesModel: NaiveBayesModel): Unit = {
    val tweetsDF: DataFrame = loadSentiment140File(sc, PropertiesLoader.batchTestDataAbsolutePath)
    val actualVsPredictionRDD = tweetsDF.select("polarity", "status").rdd.map {
       case Row(polarity: Int, tweet: String) =>
        val tweetText = replaceNewLines(tweet)
        val tweetInWords: Seq[String] = MLlibSentimentAnalyzer.getBarebonesTweetText(tweetText, stopWordsList.value)
        (polarity.toDouble,
          naiveBayesModel.predict(MLlibSentimentAnalyzer.transformFeatures(tweetInWords)),
          tweetText)
    }
    saveClassifiedTweets(actualVsPredictionRDD, PropertiesLoader.tweetsClassifiedPath)
    Console.println("Result prediction saved!");
  }
```

```
  def loadSentiment140File(sc: SparkContext, sentiment140FilePath: String): DataFrame = {
    val sqlContext = SQLContextSingleton.getInstance(sc)
    val tweetsDF = sqlContext.read
      .format("com.databricks.spark.csv")
      .option("header", "false")
      .option("inferSchema", "true")
      .load(sentiment140FilePath)
      .toDF("polarity", "id", "date", "query", "user", "status")
    tweetsDF.drop("id").drop("date").drop("query").drop("user")
  }

  def replaceNewLines(tweetText: String): String = {
    tweetText.replaceAll("\n", "")
  }

  def saveClassifiedTweets(rdd: RDD[(Double, Double, String)], tweetsClassifiedPath: String): Unit= {
    val dir = new String("results_batch")
    val sqlContext = SQLContextSingleton.getInstance(rdd.sparkContext)
    import sqlContext.implicits._
    val classifiedTweetsDF = rdd.toDF("Actual", "Predicted", "Text")
   classifiedTweetsDF.write
      .format("com.databricks.spark.csv")
      .option("header", "true")
      .option("delimiter", "\t")
      .option("codec", classOf[GzipCodec].getCanonicalName)
      .mode(SaveMode.Append)
      .save(PropertiesLoader.tweetsClassifiedPath + dir)
  }
}
```

Bibliografia

[1] M. Khader, A. Awajan, G. Al-Naymat, “Sentiment Analysis Based on MapReduce: A survey”, 2018.

[2] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” Communications of the ACM, vol. 51, no. 1, pp. 107–113, 2008.

[3] D. Lewis, “Naive (bayes) at forty: The independence assumption in information retrieval,” Machine Learning: ECML-98, pp. 4–15, 1998.

[4] “Wikipedia, L'enciclopedia libera”, en.wikipedia.org

[5] “About Twitter’s APIs”, help.twitter.com/en/rules-and-policies/twitter-api

[7] Dataset utilizzati:

1. Sentiment140 (www.sentiment140.com/),
2. Twitter US Airline Sentiment | Kaggle (https://www.kaggle.com/crowdflower/twitter-airline-sentiment),
3. T4SA Dataset - Twitter for Sentiment Analysis Dataset (www.t4sa.it/)