POLITECNICO DI TORINO

DIPARTIMENTO DI AUTOMATICA E INFORMATICA

Master in Ingegneria Informatica

Tesi di Laurea Magistrale

Architetture di elaborazione dati in streaming per analisi e previsione di serie temporali

Relatore
Prof. Luca Ardito

Candidato
Alberto Conti

Supervisore aziendale
Blue Reply
Dott. Andrea Bardone

Anno Accademico 2019-2020

Sommario

La quantità totale di dati creati, acquisiti, copiati e consumati nel mondo è in continua crescita, una crescita esponenziale che ha raggiunto nel 2020 i 59 zettabyte [1]. La semplicità di raggiungere e creare nuovi dati permette anche ai meno esperti di accrescere questo calderone, si pensi a quanto l'utilizzo dei social media sia integrato nella vita quotidiana, senza dimenticare che ogni dispositivo elettronico dotato della possibilità di connettersi alla rete genera e scambia dati: il famoso Internet of Things, ormai diventato un termine ricorrente. Cercare di stare al passo di una tale disponibilità di dati ha fatto si che si evolvessero sempre più le tecnologie in grado di elaborarli in modo efficiente, ma oltre ad essere in grado di processare queste grandi moli di dati, bisogna riuscire a farlo nel minor tempo possibile. Ad ogni secondo che passa nuovi dati vengono generati, se si vuole essere aggiornati e agire in tempo reale bisogna sempre avere a disposizione le informazioni più recenti aggiungendo un ulteriore difficoltà al nostro processo di elaborazione. Se si fosse in grado di processare in tempo reale la totalità delle informazioni pubbliche riguardanti uno specifico argomento, riuscendo a rielaborare allo scopo di trovare una certa logica di evoluzione, sarebbe possibile avere una visione tale da pensare di poter prevedere con una certa accuratezza l'evoluzione futura. Come nel caso della previsione di code di traffico, o nell'individuazione di transazioni fraudolente, dove la tempestività è un elemento chiave, noi vogliamo poter sfruttare grandi quantità di informazioni aggiornate all'ultimo secondo per effettuare delle analisi che possano influenzare decisioni. Come ogni cosa automatizzata bisogna fare attenzione ed essere in grado di interpretare i risultati proposti, sapendo di poter incorrere in casi come "beer and diapers" [2] o "chocolate and nobel prizes" [3] dove può essere individuata una relazione di causa-effetto laddove i fattori sono realisticamente indipendenti.

Lo scopo di questa tesi è volto ad affrontare questo tipo di problematiche, prima è necessario realizzare un infrastruttura Big Data che sia in grado di supportare un flusso continuo di dati in tempo reale, che possa essere scalabile per adattarsi a qualsiasi carico e successivamente che riesca a elaborare questi dati reagendo in un breve lasso di tempo, mettendo in pratica algoritmi di machine learning e deep learning, e che infine possa scaturire delle decisioni.
L'intero lavoro è stato svolto per la compagnia Blue Reply del gruppo Reply con sede a Torino. Anche se l'intero progetto è stato svolto da remoto in smart working per via del lock down causato dal COVID-19, ci è stato messo a disposizione un server per poter mantenere operativa l'infrastruttura senza interruzioni 24/24. L'obiettivo è satato quello di analizzare nel mercato finanziario una particolare valuta, nello specifico il Bitcoin. Si è studiato quindi l'andamento del Bitcoin al fine di cercare di effettuare una previsione sulle fluttuazioni future, sfruttando algoritmi di machine learning classici e moderni. Questo lavoro è stato svolto sul Bitcoin, ma può essere esteso a qualsiasi altra moneta virtuale, e vale anche per azioni quotate.

La tesi è divisa in cinque capitoli. Il primo avrà principalmente uno scopo introduttivo: si spiegheranno i due concetti portanti di Big Data e di Continuous Intelligence che rappresentano il fondamento su cui si basa l'intero lavoro, e sarà analizzato lo stato dell'arte, sottolineando prima alcuni dei requisiti fondamentali di un architettura streaming, poi alcune delle più recenti infrastrutture realizzate per il real-time processing. Nel secondo capitolo progetteremo l'infrastruttura che vorremmo utilizzare, andando a scoprire tutti i layer necessari e per ognuno di essi effettuare alcuni confronti tra i possibili componenti e scegliendo tra i diversi framework quelli che sembrano i più idonei. Nel terzo capitolo si affronterà l'aspetto pratico, in cui l'architettura pensata sarà realizzata. Per ogni componente è spiegato come viene trasformato il flusso di dati, mentre compie il percorso da sorgente a rielaborazione a analisi finale, specificando le configurazioni utilizzate e mostrando i frammenti di codice più significativi. Nel capitolo quarto si analizzano i dati finali ottenuti da un lungo flusso di trasformazioni, con l'idea di non poter costruire un modello se non si conoscono i dati. Dopo questo infatti si è cercato di realizzare un modello che fosse in grado di prevedere l'andamento del Bitcoin. Per primo si è messo in pratica il modello ARIMA, un approccio classico nella

predizione di serie storiche; come secondo si è voluta testare l'efficacia di un modello che sfruttasse il deep learning: l'LSTM, un particolare caso di RNN. Infine si trarranno le conclusioni dell'intero lavoro.

Indice

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Introduzione

1.1 Continuous Intelligence e Big Data

In questo 2020 ma soprattutto negli anni futuri, la Continuous Intelligence (CI) vedrà uno smisurato aumento della sua applicazione, che diventerà uno standard nell'ambito aziendale [4]. Si tratta di un analisi che elabora oltre a dati storici anche quelli in tempo reale, per prescrivere azioni tempestive o semplicemente per migliorare le decisioni. É un approccio che sfrutta tecniche di machine learning per l'apprendimento ed è in grado di basarsi su tutti i dati di cui ha bisogno, indipendentemente dalla loro grande quantità e dalla presenza di pattern conosciuti: è un modello di progettazione che automatizza l'analisi in modo che sia continua e senza interruzioni. La sua definizione si potrebbe confondere con il concetto di Business Intelligence (BI), la differenza sta nel fatto che gli strumenti di BI non coinvolgono il machine learning o l'AI. La BI si basa fortemente sulle competenze e ci si aspetta che gli esperti di dati guidino tale strumento in ogni fase del flusso di lavoro, direttamente dall'estrazione dei dati e dall'integrazione attraverso l'interpretazione. Inoltre, a differenza della CI, BI non è stato ideato per velocizzare l'accesso a dati e informazioni.

Nell'applicazione della CI dovremo sfruttare le proprietà dei Big Data, ma vediamo nel dettaglio che cosa sono. Non esiste una definizione univoca, ognuno ha coniato la sua interpretazione. Doug Laney, l'analista di Gartner, definì le caratteristiche dei Big Data attraverso il modello delle 3V [5].

- **Volume**, indica la spropositata mole di dati utilizzabili in input.

- **Variety**, ci fa capire quanto queste sorgenti possano essere diverse, tra audio, immagini, video, dati catturati da sensori...

- **Velocity**, mette in luce il fattore tempo, essendo moltissimi dati e sempre in continuo aumento, è fondamentale riuscire a processarli nel minor tempo possibile.

Un'altra possibile definizione è quella data da Andrea De Mauro, Marco Greco e Michele Grimaldi nel loro articolo "A Formal definition of Big Data based on its essential features" [6], in cui vedono i Big Data come l'insieme delle tecnologie e dei metodi di analisi, che indipendentemente dal campo di applicazione, sono in grado di trasformare grandi quantità di dati, anche eterogenei, allo scopo di estrarre delle dei legami nascosti utili alle società e alle imprese. Ricordiamo anche le tre D2D del Dr. Kirk Borne.

- **Data to Decisions**, possiamo dire che le decisioni siano l'obiettivo finale. Le scelte sono governate da algoritmi di machine learning, allenati e validati da data scientist. Anche semplici statistiche possono essere sufficienti per giustificare una decisione importante.

- **Data to Discovery**, corrisponde all'obiettivo conoscenza. Scoprire relazioni tra dati per scoprire nuove opportunità, strategie, segnali che precedono un evento, categorie di interessi e altro.

- **Data to Dollars**. Dobbiamo l'istanza di questo concetto a Jaime Fitzgerald di Fitzgerald Analytics [7]. Il significato non lascia ambiguità: i big data sono il "nuovo petrolio", la nuova fonte di entrate, il carburante della nuova economia dell'innovazione, il motore della creazione di ricchezza nell'era dell'informazione.

1.2 Architetture streaming

Molte applicazioni hanno la necessità di ottenere risultati nel minor tempo possibile allo scopo di prendere decisioni migliori, per questo lo streaming in tempo-reale è un fattore importante. Attualmente molti sistemi che gestiscono big data sono basati sul framework Hadoop, altamente scalabile e in grado di gestire i guasti in modo efficiente. Per questo non hanno nulla

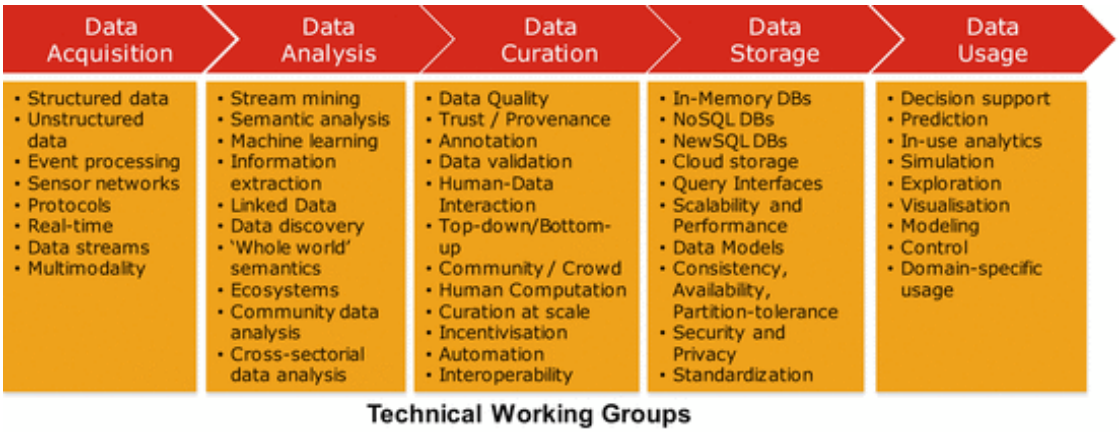


Figura 1.1: Catena per estrarre informazioni dai Big Data. [8]

da invidiare ad altri sistemi, se non che non sia appropriato per lo stream processing. É necessaria quindi un'architettura alternativa in grado di elaborare flussi di dati in tempo-reale, mantenendo una bassa latenza. Tra le varie possibilità esistono due principali architetture che soddisfano le nostre necessità: Lambda e Kappa.
Lambda è un framework per gestire grandi quantità di dati, progettato da Nathan Marz con l'idea di creare un sistema che usi contemporaneamente il batch processing e il real-time processing. La figura 1.2 mostra la rappresentazione di questo sistema suddividendolo in tre diversi layer.

1. **Batch layer**. Ha il compito di archiviare in memoria l'intero dataset, tenedolo aggiornato con l'arrivo dei nuovi dati in real-time e correggendo eventuali errori nello streaming. Rappresenta il livello affidabile e si serve di data warehouse tradizionale.

2. **Serving layer**. É in grado di ridurre la latenza del sistema, comunicando con il batch layer tramite opportune query ad-hoc eseguibili in tempo reale. Questo livello può essere realizzato con database NOSQL come Cassandra o HBase.

3. **Speed layer**. Si occupa di gestire gli ultimi dati generati, mettendo la bassa latenza sopra alla precisione. Si serve di algoritmi veloci per la generazione di viste incrementali. Dei possibili framework applicabili a questo layer possono essere Apache Storm e Spark Streaming.

Passiamo ora ad analizzare Kappa, l'architettura software ideata da Jay Kreps. Riesce ad elaborare dati sia in tempo reale che in batch, ma la sua

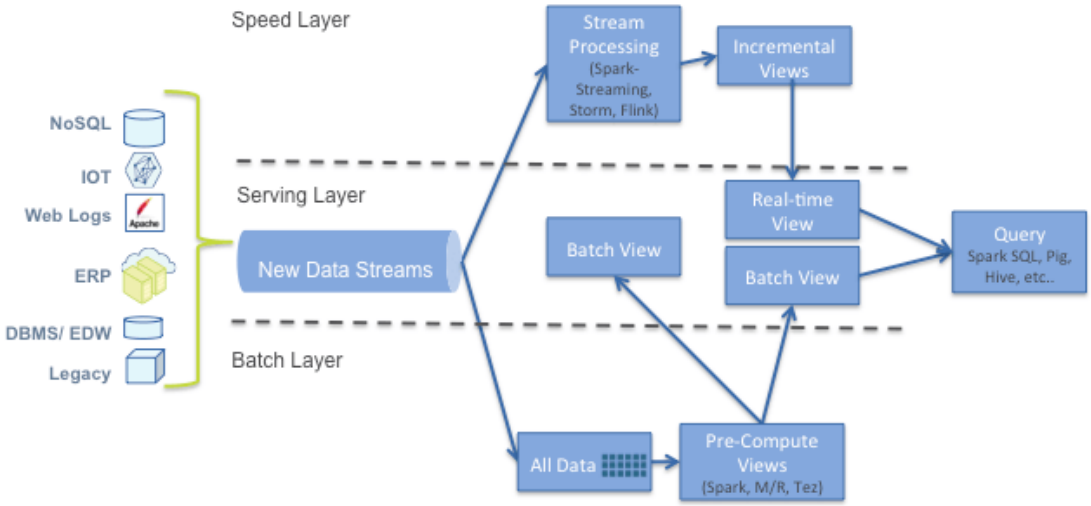


Figura 1.2: Diagramma architettura Lambda. [9]

peculiarità sta nell'utilizzare lo stesso stack tecnologico per gestire l'elaborazione del flusso in tempo reale e l'elaborazione batch storica. Questa sua caratteristica la fa considerare un'alternativa più semplice della Lambda. In generale si basa su un'architettura di streaming in cui una serie di dati in input viene prima archiviata in un motore di messaggistica come Apache Kafka, per poi essere letta da un motore di elaborazione del flusso che, se necessario li trasformerà in un formato analizzabile e li memorizzerà in un database di analisi affinché gli utenti finali possano interrogarli. Dopo che i dati sono inseriti nel motore di messaggistica, possono essere analizzati in tempo reale. Ma è anche possibile l'analisi storica, leggendo in un secondo momento i dati di streaming archiviati in modo batch. Si hanno quindi due layer principali: lo stream-processing layer e il serving layer, mentre il batch layer dell'architettura Lambda non è più necessario. Per fare un paragone, entrambe le architetture comportano l'archiviazione di dati storici per consentire analisi su larga scala, ed entrambe sono utili dal punto di vista della fault tolerance, in cui i problemi con il codice di elaborazione (bug o limitazioni note) possono essere superati aggiornando il codice ed eseguendolo nuovamente sui dati storici. La differenza principale è che l'architettura Kappa non utilizza uno storage persistente e tutti dati vengono trattati come se fossero un flusso e per questo tipo di dati è più performante. Mentre in Lambda i dati non sono immediatamente consistenti e la logica di business è implementata sia nello speed layer

che nel batch layer, causando un aumento del lavoro di mantenimento da parte degli sviluppatori.

Cercando di mettere insieme i vantaggi di queste due architetture è nata un nuovo sistema [10], composto da 5 diversi layer.

1. **Integration layer**. Cattura i dati provenienti dall'esterno, indipendentemente dal loro formato.

2. **Filtering layer**. É uno strato intermedio che effettua delle operazioni di pre-processing per snellire i dati, elimando ad esempio campi inconsistenti.

3. **Real-Time processing layer**. Da un lato è composto da Storm per essere in grado di processare grandi quantità di dati a bassa latenza, dall'altro usa il machine learning per apprendere continuamente conoscenza sui nuovi dati in ingresso.

4. **Storage layer**. Utilizza HBase per memorizzare in modo persistente i risultati ottenuti dal layer precedente.

5. **Presentation layer**. Ultimo strato che mostra all'utente i risultati finali.

Passiamo ad analizzare un'altra applicazione di architettura Big Data per l'analisi in streaming. Considerando che nel nostro caso pratico interagiremo con Twitter, vediamo una possibile architettura che ne effettua un'integrazione. Tutti i dettagli sono presentati nell'articolo "Developing a Real-time Data Analytics Framework For Twitter Streaming Data" [11]. Si cerca di creare un'infrastruttura capace di analizzare in tempo reale tutti i tweet generati secondo per secondo. Il framework, come nell'immagine 1.3, è divisibile in tre moduli: data ingestion, data processing e data visualization. Kafca tramite diversi producer si occupa dell'ingestione dei dati sorgente, attraverso chiamate all'API di Twitter memorizzano su appositi topic le informazioni interessate. Da lì vengono attinte dai consumer che passano la palla a Spark. Il cluster di Kafka è composto da diversi broker che gestiscono la lettura/scrittura dai/sui topic. Zookeeper entra in gioco per garantire il corretto funzionamento a seguito di un malfunzionamento di un broker. Il secondo modulo è realizzato con Skark. Al suo interno avvengono diversi processi, Spark Streaming riesce

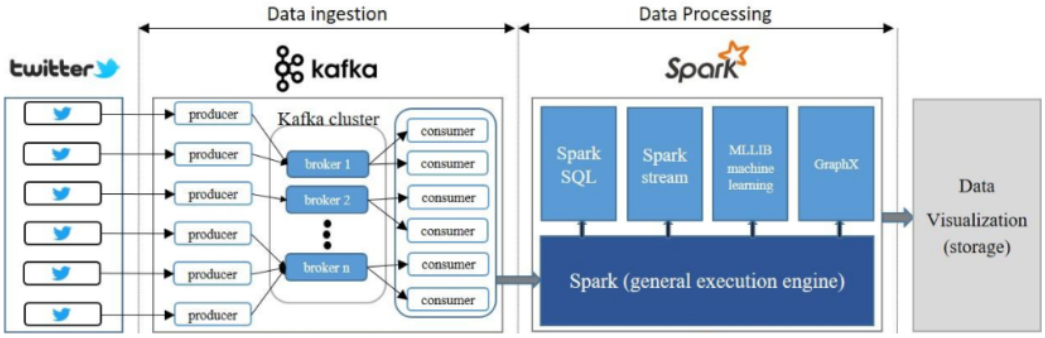


Figura 1.3: Framework per analisi in tempo reale di uno streaming di tweet. [11]

ad analizzare i dati real-time ricevuti tramite i consumer di Kafka e li divide in mini-batches. Spark Engine si serve dei risultati dei mini-batches e li trasforma in modo da permettere una memorizzazione persistente, ad esempio con Spark SQL si può interagire con un database. Per effettuare previsioni più complesse possono essere sfruttate librerie come MLlib che permettono l'uso del machine learning, sfruttando le sue capacità vengono generate soluzioni più accurate. L'ultima parte è il data visualization, dove si possono osservare i dati tramite un database noSQL, per poter visualizzare oltre al testo, anche immagini e altri formati.

L'ultima architettura che proponiamo è pensata per un applicazione di IoT con le smart city come caso d'uso [12]. Questo genere di applicazione tipicamente necessitano di reagire ad eventi appena si verificano, e di basare le proprie scelte in funzione degli eventi passati. Per questa ragione può essere utile pre-analizzare lo storico prima di passare all'analisi in streaming. L'architettura è divisibile in due flussi separati: batch e stream (figura 1.4). Per prima cosa è necessaria l'ingestion, a questo scopo è utilizzato il software Note Red che è in grado di attingere da dati appartenenti a diversi dispositivi/sorgenti esterne, in qualsiasi formato desiderato, che sia XML piuttosto che JSON. Può essere utile un operazione di pre-procesing, poiché avendo più fonti è possibile che alcuni dati siano ridondanti. Kafka ospita i risultati di Node Red e dal message broker sono trasferiti tramite Secor, un tool open source, verso OpenStack Swift. Qui si fa largo uso di metadati per facilitare la ricerca tra tutte le informazioni attraverso Elastic Search. Infine entra in gioco Spark che riesce ad

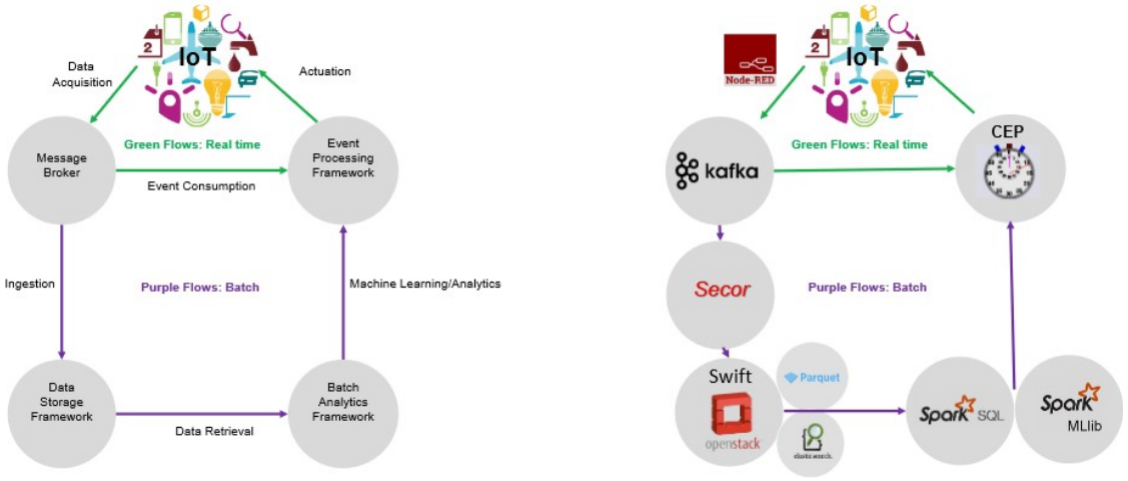


Figura 1.4: Architettura per acquisizione e analisi dati in applicazioni IoT. [12]

accedere ai dati posti in ogni tipo di database e mette a disposizione librerie per applicare machine learning. Per creare eventi complessi in fase di streaming viene usato il framework Complex Event Processing che riesce a correlare più eventi indipendenti. Inoltre per migliorare l'acquisizione di dati, CEP ha a disposizione degli appositi pugin che semplificano la logica di business definendo alcune regole da seguire fino al raggiungimento del risultato finale.

Capitolo 2

Progettazione dell'infrastruttura

Innanzitutto individuiamo quali sono i componenti fondamentali di un architettura in streaming che porta dalla cattura dati in tempo reale all'analisi finale. La figura 2.1 mostra alcuni degli elementi chiave necessari per l'analisi che ci interessa. Primo tra tutti è il blocco che rappresenta la fonte dei dati, come può essere un sito o un database da cui attingeremo tutte le informazioni necessarie in tempo reale. Successivamente, sapendo da dove attingere i dati, sarà necessario un data ingestion, seguito da un data storage, da cui potrà usufruire un data processing, che per il nostro scopo avrà bisogno di un integrazione per il machine learning. Andiamo ad analizzare per ognuno di questi blocchi, quali potrebbero essere dei buoni componenti.

.

2.1 Data ingestion

Le informazioni devono essere ingerite prima di poter essere digerite, per questo il primo ingranaggio della nostra architettura deve svolgere una funzione di data ingestion: il processo di acquisizione e importazione dati per l'uso immediato o l'archiviazione in un database. I dati così ottenuti possono essere trasmessi in real-time streaming o inseriti in batch. Nel caso i dati vengano importati in tempo reale, ogni elemento viene importato man mano che viene emesso dall'origine, alternativamente se i dati

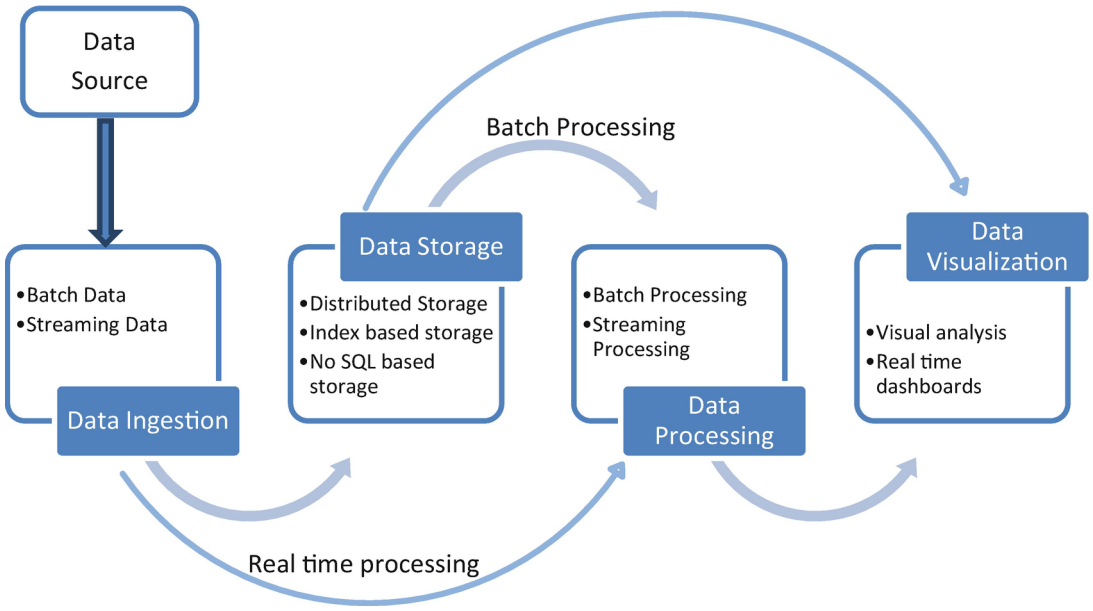


Figura 2.1: Struttura per un'analisi in tempo reale. [13]

vengono inseriti in batch, ogni elemento è importato in blocchi discreti a intervalli di tempo periodici. Un efficace processo di acquisizione inizia dando la priorità alla sorgente dei dati, convalidando i singoli file e instradando i vari elementi verso la destinazione corretta. Quando l'immissione dei dati è automatizzata, il software utilizzato per eseguire il processo può anche includere funzionalità di preparazione dei dati per strutturare e organizzare i dati in modo che possano essere analizzati al volo o in un secondo momento da programmi di Business Intelligence (BI) e Business Analytics (BA). In sostanza questo componente ci permetterà di correlare dati provenienti da più origini, anche con formati diversi, gestendo solidamente il flusso di dati per non avere a disposizione un quadro incompleto dei dati disponibili che può comportare relazioni fuorvianti, conclusioni analitiche spurie e un processo decisionale compromesso. Di seguito spiegheremo alcune caratteristiche fondamentali che deve possedere un data ingestion seguendo i principi di Pat Research [14]:

- **Estrazione dati e preprocessamento**. Come già detto corrisponde al principale obiettivo. Gli strumenti di data ingestion possono utilizzare protocolli di trasporto dati diversi per raccogliere, integrare, elaborare e consegnare i dati alle destinazioni appropriate.

- **Visualizzazione del flusso di dati**. Questa funzione permette agli utenti di poter visualizzare graficamente il flusso di dati con una semplice interfaccia drag-and-drop che consente di visualizzare dati anche complessi e poterli gestire in modo intuitivo.

- **Scalabilità**. Gli strumenti di inserimento dati più efficaci sono in grado di adattarsi a qualsiasi flusso di dati, sia per quanto riguarda la dimensione che per l'intensità, soddisfacendo le esigenze di elaborazione desiderate. Questo è realizzabile grazie ad una configurazione appropriata dei nodi, volta ad aumentare il numero di transazioni e/o il parallelismo.

- **Supporto e integrazione multi-piattaforma**. Un'altra caratteristica importante è la capacità di estrarre diverse tipologie di dati, sia che provengano da cloud che da locale. Gli strumenti di data ingestion possono accedere regolarmente ai dati da diversi tipi di database e sistemi operativi senza influire sulle prestazioni.

- **Caratteristiche di sicurezza avanzate**. I migliori strumenti di data ingestion utilizzano vari meccanismi di crittografia dati e protocolli di sicurezza come SSL, HTTPS e SSH per proteggere i dati aziendali.

Proviamo adesso ad entrare nel dettaglio ed analizzare dei software specifici. Il mercato offre un ampio ventaglio di possibilità, non è una scelta semplice scegliere gli strumenti più appropriati, è anche possibile usare diversi tipi di strumenti per captare dati da più sorgenti in modo efficiente ed efficace. Noi ci limiteremo ad analizzare due delle possibilità, Apache NiFi e Wavefront. Al fondo del capitolo tireremo le somme.

2.1.1 Apache NiFi

Apache NiFi è una piattaforma per la logistica dei dati, pensata per automatizzare lo spostamento dei dati tra sorgenti e destinazioni di qualsiasi tipo. Si basa sulla tecnologia inizialmente chiamata "Niagara Files" sviluppata e dalla NSA [15] per poi essere usata dalla Apache Foundation. É una tecnologia flessibile ed estensibile che può essere utilizzata su ogni tipo di dispositivo, da un raspberry al colud. Inoltre è in grado di adattarsi dinamicamente alla disponibilità di rete che potrebbe influire sulle

comunicazioni e quindi sul flusso di dati.

NiFi cerca di realizzare i concetti correlati con l'idea di FBP(flow based programming), di seguito la tabella 2.1 tratta dalla panoramica del sito ufficiale di NiFi [16] correlerà i concetti di NiFi facendo riferimento alle proprietà di una FBP. Alcune delle caratteristiche di alto livello di Apache NiFi che potrebbero permettere di scegliere NiFi tra altri software di ingestion sono la predilezione per la loss tolerant contro la garanzia di invio, la bassa latenza contro un alto throughput, avere una gestione delle priorità dinamica, la possibilità di modificare il flusso in runtime e la back pressure. Senza contare la comoda interfaccia grafica per gestire il flusso più intuitivamente con la possibilità di aggiungere processori ad hoc potendosi servire di un rapido sviluppo e di un test efficace. In aggiunta è la possibilità di usare SSL, SSH, HTTPS, contenuti criptati, autorizzazioni multi-tenant [17] e gestire autorizzazioni/policy interne.

2.1.2 Wavefront

Wavefront è una piattaforma ospitata per l'importazione, l'archiviazione, la visualizzazione e l'avviso su dati metrici. Si basa su un approccio di elaborazione del flusso inventato da Google che consente agli ingegneri di manipolare i dati metrici con una potenza senza pari. Wavefront rende l'analisi semplice ma potente; Il nostro linguaggio di query consente di manipolare i dati delle serie storiche in modi mai visti prima di nbsp; Il linguaggio è di facile comprensione, ma abbastanza potente da gestire dati ad alta dimensione. Wavefront può ingerire milioni di punti dati al secondo. Sfruttando un linguaggio di query intuitivo, è possibile manipolare i dati in tempo reale e fornire informazioni fruibili. Questo aiuta ad affrontare la consegna delle applicazioni: ridurre i tempi di inattività delle applicazioni, rilevare e riparare i problemi molto rapidamente. Cloud Scale: monitora le prestazioni SaaS e software attraverso i servizi cloud e on-premise. Analitica operativa: abbatti i silos di dati e migliora. Collaborazione DevOps. Ottieni una vista a 360 ° su IT nbsp; infrastruttura, allineare l'IT con il business e ottenere efficienze. Grande gestione dei dati: ricerca, esplorazione, navigazione, navigazione, analisi, correlazione e visualizzazione di milioni di punti dati al secondo da ogni parte dell'azienda.

2.2 Message Broker

Il message broker è un elemento di transizione, fondamentale per mettere in comunicazione diverse applicazione per costruire un meccanismo di integrazione comune a supporto dell'intera infrastruttura. La capacità di tradurre messaggi per renderli compatibili con protocolli diversi rende possibile mettere in comunicazione servizi che possono rimanere indipendenti coi propri linguaggi e le proprie piattaforme, mantenendo tutti i vantaggi di un'architettura a micro-servizi. Gli sviluppatori si trovano così uno strumento standardizzato per gestire il flusso di dati tra diversi componenti in modo da potersi concentrare maggiormente sulla logica principale. In aggiunta, al fine di fornire un'archiviazione affidabile dei messaggi e una consegna garantita, i broker di messaggi spesso fanno affidamento su una sotto-struttura o un componente chiamato message queue che archivia e ordina i messaggi fino a quando le applicazioni che li utilizzano possono elaborarli. In questa coda di messaggi, i messaggi vengono archiviati nell'ordine esatto in cui sono stati trasmessi e rimangono nella coda fino alla conferma della ricezione. La comunicazione asincrona che viene a generarsi tra le applicazioni previene la perdita di dati preziosi e consente ai sistemi di continuare a funzionare anche di fronte a una connettività intermittente o latenza, problemi comuni sulle reti pubbliche. I message broker possono comprendere dei gestori di code, utili per gestire le interazioni tra più code di messaggi, nonché servizi che forniscono funzionalità di routing dei dati, traduzione dei messaggi, persistenza e gestione dello stato del client.
I message broker possono possono seguire due diversi stili di messaggistica:

- Point to point
- Publish and subscribe

Nel primo caso c'è una relazione uno a uno tra il mittente e il destinatario del messaggio. Ogni messaggio nella coda viene inviato a un solo destinatario e viene consumato una singola volta. La messaggistica point to point viene sfruttata quando un messaggio deve essere usato una sola volta.
Nella seconda modalità viene sfruttato il concetto di topic. Un produttore pubblicherà tutti i messaggi che genera in un determinato topic, e ogni consumatore iscritto a quello stesso topic potrà ricevere quei messaggi.

Questo è un metodo di distribuzione in stile broadcast, in cui esiste una relazione uno-a-molti tra l'editore del messaggio e i suoi consumatori.
Proviamo ora a considerare due possibili candidati a svolgere le funzioni di un message broker: Apache RabbitMQ e Apache Kafka. Al fondo del capitolo tireremo le somme.

2.2.1 Apache RabbitMQ

RabbitMQ 'è un broker di messaggi generico in grado di supportare diversi protocolli, tra i quali AMQP (Advaced Message Queieng Protocol), MQTT (Message Queue Telemetry Transport) e STOMP (Streaming Text Oriented Messagging Protocol). Può essere usato in applicazioni ad alto rendimento, come l'elaborazione dei pagamenti online, e fungere da broker di messaggi tra microservizi. Tra le sue caratteristiche abbiamo bassa latenza e scalabilità. Può elaborare anche milioni di messaggi al secondo ma richiede più risorse rispetto a Kafka. L'availability è fornita tramite replicazione dei messaggi. Sfrutta le dinamiche di un modello push, ovvero, tramite un meccanisco di ack esplicito, il consumatore decide la corretta ricezione dei messaggi impedendo un eventuale sovraccarico tramite un limite impostogli. Con questo tipo di modello i messaggi sono distribuiti rapidamente ed individualmente, garantendo che il lavoro sia parallelizzato in modo uniforme e che i messaggi siano elaborati rispettando l'ordine di coda [18]. Altre funzionalità che mette a disposizione sono l'interfaccia grafica per il monitoraggio delle code, la possibilità di assegnare una priorità diversa ad ogni messaggio, e supporto alle transazioni, fornendo commit e rollback.

2.2.2 Apache Kafka

Apache Kafka è una piattaforma di streaming distribuita, usata per la costruzione di comunicazioni tra dati real-rime e altre applicazioni streaming. Scalabile orizzontalmente e fault-tolerant

- Emette e scrive a stream di record (come una message queue).
- Memorizza stream di record garantendo fault-torelarance.
- Processa stram di record nel momento in cui si presentano.

Kafka è pensato per gestire e manipolare streaming real-time tra sistema e applicazioni, o di reagire a questi generando delle proprie applicazioni di streaming. Kafka è lanciata come cluster in uno o più server, ed archivia record di stream in topics. Ogni record è formato da key, value e timestamp. Ci sono quattro core APIs:

- **Producer**: crea stream di record su uno o più topic.
- **Consumer**: riceve stream da uno o più topic.
- **Streams**: permette all'app di agire come stream processor: consuma uno stream in input, anche da più topic, e lo trasforma in uno stream in output, se necessario in più topic.
- **Connector**: permette di lanciare consumer e producer su app esistenti o data system (es. su DB relazionale, posso catturare tutte le modifiche di una tabella).

I messaggi in kafka sono categorizzati in topic, questi possono essere visti come tabelle di un database o cartelle in un filesystem. I topic possono essere composti da più partizioni per favorirne la scalabilità e possono essere replicati n volte, dove n è il fattore di replica dell'argomento che può essere specificato nel file di configurazione. Ciò permette che i messaggi rimangano disponibili anche in presenza di errori o nel caso che un server nel cluster si guastasse, andando a recuperare i dati dalle repliche. Tra più repliche una è designata come leader mentre le altre sono follower. Come suggerisce il nome, il leader riceve i messaggi dal produttore mentre i follower semplicemente copiano il registro del leader mantenendo l'ordinamento. La garanzia fondamentale che deve fornire un algoritmo di replica dei registri è che se comunica al client che è stato eseguito il commit di un messaggio e il leader fallisce, anche il nuovo leader eletto dovrà avere quel messaggio. Kafka offre questa garanzia richiedendo che il leader sia eletto da un sottoinsieme di repliche che sono "sincronizzate" con il leader precedente o, in altre parole, recuperate nel registro del leader. Il leader di ogni partizione tiene traccia di questo elenco di repliche sincronizzate calcolando il ritardo di ogni replica da se stesso. Quando un produttore invia un messaggio al broker, questo viene scritto dal leader e replicato in tutte le repliche della partizione. Su un messaggio viene eseguito il commit solo dopo essere stato copiato correttamente in tutte

le repliche sincronizzate. Poiché la latenza di replica dei messaggi è limitata dalla replica in-sync più lenta, è importante rilevare rapidamente repliche lente e rimuoverle dall'elenco di repliche in-sync. Leggere dati da kafka è diverso dal leggerli da altri sistemi di messaggistica. Bisogna sfruttare un consumer che legga da un kafka topic e si serva di un sistema di validazione dati per poi poterli utilizzare, ed i risultati potrebbero successivamente essere messi in un altro punto di stoccaggio. Kafka, da buon sistema distribuito, permette che i dati in entrata possano superare in numero la capacità di analizi di un singolo producer, evitando all'applicazione di rimanere sempre più indietro in presenza di un flusso ingente di dati. Questo è possibile perché si possono utilizzare gruppi di consumer per un singolo topic, ognuno dei quali potrà attingere alla porzione dei dati provenienti da un sottoinsieme delle partizione in cui è diviso il topic. Attenzione che avendo a disposizione un numero di consumer superiore al totale delle partizioni, i consumer in eccesso rimarranno in idle senza poter gestire nessun messaggio [19]. Ogni partizione è una sequenza immutabile di record, ovvero offset che identificano univocamente ogni record. L'offset corrisponde all'ultima posizione tra i messaggi che il consumer ha utilizzato, se un consumer smette di funzionare possiamo sostituirlo o ripararlo facendolo ripartire continuando dal punto in cui l'altro si era interrotto. In questo modo siamo sicuri di non dimenticare nessun dato senza il rischio di duplicati. I record pubblicati hanno una durata in base alla retention policy e in quel tempo possono essere consumati quanti volte si vuole, e una volta raggiunto il tempo limite o la capacità specificata, vengono scartati lasciando spazio ai successivi.

2.3 Data Processing

Una volta ottenuti i dati di partenza si passa alla fase di trasformazione e manipolazione per poter generare delle nuove informazioni significative, pronte dopo essere interpretate, per supportare e fornire decisioni. Come accennato nella sezione precedente, i big data di solito sono archiviati in migliaia di commodity server, quindi i modelli di programmazione tradizionali come l'MPI (Message Passing Interface) non possono gestirli in modo efficace. Per questo motivo vengono utilizzati nuovi modelli di programmazione parallela per migliorare le prestazioni dei database NoSQL nei data center. MapReduce è uno dei modelli di programmazione più

popolari per l'elaborazione dei big data utilizzando cluster su larga scala. MapReduce è un framework brevettato da Google e sviluppato da Yahoo. Le funzioni Map e Reduce sono programmate per elaborare i big data distribuiti su più nodi eterogenei. Il vantaggio principale di questo modello di programmazione è la semplicità, quindi gli utenti possono facilmente utilizzarlo per l'elaborazione dei big data [20]. Tra i possibili framework di processing, analizziamo Spark streaming e Flink.

2.3.1 Spark streaming

Apache Spark Streaming è un sistema di elaborazione di streaming scalabile con tolleranza agli errori che supporta nativamente carichi di lavoro sia in batch che in streaming. Spark Streaming è un'estensione dell'API Spark principale che consente ai data engineer e ai data scientist di elaborare dati in tempo reale da varie fonti, inclusi (ma non limitati a) Kafka, Flume e Amazon Kinesis. Questi dati elaborati possono essere inviati a file system, database e dashboard live. La sua astrazione chiave è un Discretized Stream o, in breve, un DStream, che rappresenta un flusso di dati suddiviso in piccoli batch. I DStream sono basati su RDD, l'astrazione di dati di base di Spark. Ciò consente a Spark Streaming di integrarsi perfettamente con qualsiasi altro componente Spark come MLlib e Spark SQL. Spark Streaming è diverso dagli altri sistemi che dispongono di un motore di elaborazione progettato solo per lo streaming o hanno API di batch e streaming simili ma compilano internamente a motori diversi. Il motore di esecuzione unico di Spark e il modello di programmazione unificato per batch e streaming portano ad alcuni vantaggi unici rispetto ad altri sistemi di streaming tradizionali. [21]

2.3.2 Flink

Flink è un framework e motore di elaborazione distribuito. I dati da lui utilizzati sono prodotti come stream di eventi, ad esempio transazioni di carte di credito, misure di sensori, log, interazione utenti... Esistono due tipologie di stream che è in grado di gestire:

- **Unbounded streams**, hanno un inizio ma non una fine, devono essere continuamente processati. Alle volte si richiede che gli eventi siano gestiti in un preciso ordine (come l'ordine di arrivo).

- **Bounded streams** (batch processing), hanno un inizio e una fine, è possibile digerire tutti i dati per processarli in una volta sola.

Flink è compatibile con gestori di risorse cluster come Hadoop YARN e Apache Mesos, ma può anche agire in stand-alone. Appena è messo in campo identifica le risorse di cui ha bisogno e le richiede. Tutte le comunicazioni avvengono per chiamate REST. Riesce a gestire eventi fuori ordine grazie Le applicazioni sono parallelizzate in task distribuiti ed eseguiti concorrentemente in cluster. Quindi un app può potenzialmente sfruttare risorse infinite. Mantiene un check-point asincrono incrementato di volta in volta che garantisce un impatto minimo sulle latenze senza intaccarne la coerenza dello stato. C'è un ottimizzazione per task locali (accesso locale di risorse), risorse mantenute in memoria e se troppo grandi sono usati accessi efficienti su strutture on-disk per bassa latenza.

2.4 Data Storage

Tutti i dati ottenuti da fonti esterne o generati e trasformati internamente devono essere messi a disposizione in un magazzino centrale. Questo componente è indipendente dai processi di ingestion e rielaborazione, ed è costituito da un database in cui i dati vengono raccolti ed archiviati, per poi essere disponibili per eventuali operazioni di Business Intelligence (BI).
Abbiamo deciso di adottare un database non relazionale, che rispetto alla loro controparte hanno diversi vantaggi:

- Hanno la possibilità di memorizzare diverse tipologie di dati come JSON, coppie chiave/valore o grafi.
- Ogni record non deve avere una struttura fissa, lo schema non deve essere definito prima dell'inserimento e l'aggiunta di nuove proprietà al volo è ammissibile, potendo lavorare così con schemi flessibili e dinamici.
- I dati non devono essere normalizzati per far si che non ci siano incongruenze o ridondanze.
- Aggiungere nuovi nodi al pool di risorse è molto semplice, quindi la scalabilità orizzontale non è un problema e lo sharding (capacità di

dividere i dati tra più server in modo trasparente alle applicazioni) è un supporto intrinseco.

- Vantano elevate prestazione in operazioni di lettura/scrittura sull'intero data-set.

Di contro non abbiamo i benefici di possedere completamente le proprietà ACID (Atomicità Consistenza, Isolamento, Durabilità), ma ci si deve accontentare delle BASE (Basic Available, Soft state, Eventully consistent). Tra i diversi database non relazionali abbiamo scelto due tra i più utilizzati: Mongo DB e Cassandra.

2.4.1 MongoDB

MongoDB è un database non relazionale incentrato sui documenti, file simili a JSON che possono vantare molti tipi di strutture che possono essere dinamiche. Risulta molto veloce, non avendo uno schema può immagazzinare un documento senza prima definirvi una struttura. I documenti sono come i record di un RDBMS e possono essere modificati a piacimento. Usa un suo proprio linguaggio di query, e per rendere più efficienti le ricerche è bene sfruttare le chiavi, altrimenti controllare l'intero documento potrebbe essere dispendioso. MongoDB possiede un set di repliche integrate, quando il database primario non è disponibile inverte il suo ruolo con quello secondario, questo richiede del tempo (tra i 10 e i 40 secondi) nel quale non è possibile scrivere sulle repliche. Se ci sono più repliche una assume un ruolo principale e le operazioni di letture e scritture vengono effettuate prima rispetto a tutte le altre repliche che sono considerate secondarie. Per le scritture in parallelo, Mongo ha a disposizione un singolo nodo principale che funge da master, per cui le capacità di quel singolo nodo determinano i limiti di velocità del database. [22]

2.4.2 Cassandra

Apache Cassandra è un database NoSQL distribuito, la sua forza risiede nel saper gestire grandi quantità di dati non strutturati, avendo la capacità di poter essere espanso orizzontalmente senza grandi sforzi. Ha uno schema simile a database relazionali con righe e colonne. Ogni riga ha una chiave univoca e il numero di colonne può essere variabile da dato

a dato, con possibilità di aggiornamenti al volo tramite linguaggio CQL (Cassandra Query Language). Questo linguaggio è molto simile all'SQL e le query per essere eseguite si basano sulla chiave primaria. Cassandra è in grado di eseguire immediatamente le repliche e consente a più master di scrivere nel cluster, anche in caso di perdita di singoli nodi. Questa fault tolerance senza necessità di fermarsi, riesce a risparmiare quei 10/40 secondi richiesti da MongoDB. Per quanto riguarda le scritture in parallelo, ogni nodo master può lavorare indipendentemente da altri master, quindi ci sono nodi di questo tipo, maggiori sono le possibili scritture in parallelo. [22]

2.5 Resoconto Architettura

Il punto di partenza, chi si occupa di interagire direttamente con la sorgente dei dati, per la nostra architettura abbiamo designato il compito a NiFi. Il message broker che andremo ad utilizzare sarà Kafka, non abbiamo a che fare con protocolli legacy e permette il disaccoppiamento tra produttore e consumatore, cancellando il problema delle diverse velocità tra consumatore e produttore, e di conseguenza una differente frequenza di messaggi non influisce sulle prestazioni. La scelta tra Stark e Flink è combattuta, entrambi sono strumenti all'avanguardia che si inseriscono perfettamente nel panorama dei Big Data. Forniscono nativamente la connessione con database Hadoop e NoSQL e soddisfano entranbi le nostre necessità. Flink però, grazie alla sua architettura, risulta essere più veloce. Apache Spark è un componente più attivo nel repository Apache, mentre Spark oltre ad avere un forte supporto da parte della comunità, per quanto riguarda la capacità di streaming, è di gran lunga migliore di Spark (poiché Spark gestisce lo streaming sotto forma di micro-batch) e ha il supporto nativo per lo streaming. Per questo abbiamo preferito scegliere Flink. Come database MongoDB ha un maggiore grado di flessibilità, ma noi ci limiteremo a memorizzarci record con una serie di proprietà, per cui avere a disposizione qualcosa di più simile a MySQL che offre una grande scalabilità sembra essere una scelta migliore. Ora che abbiamo scelto i componenti della nostra architettura possiamo mostrare la figura 2.3 ne stilizza la forma.

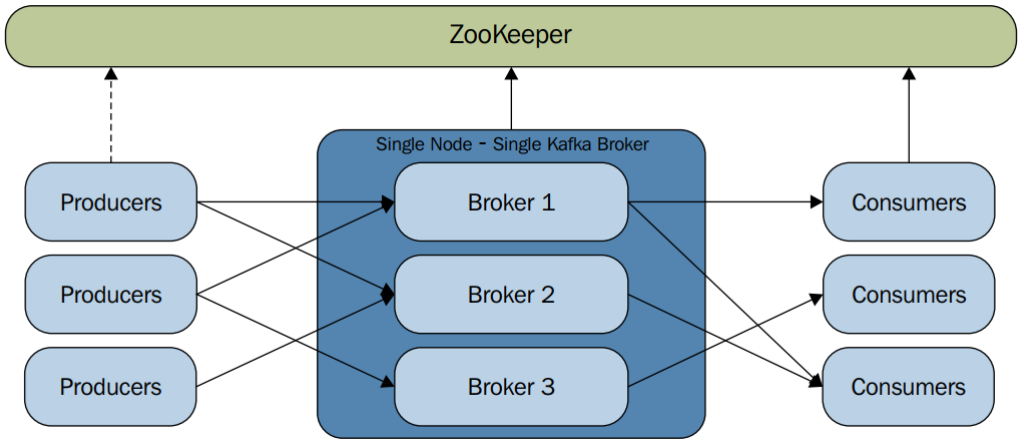


Figura 2.2: Schema che mostra un cluster con più broker, con un singolo nodo. [23]

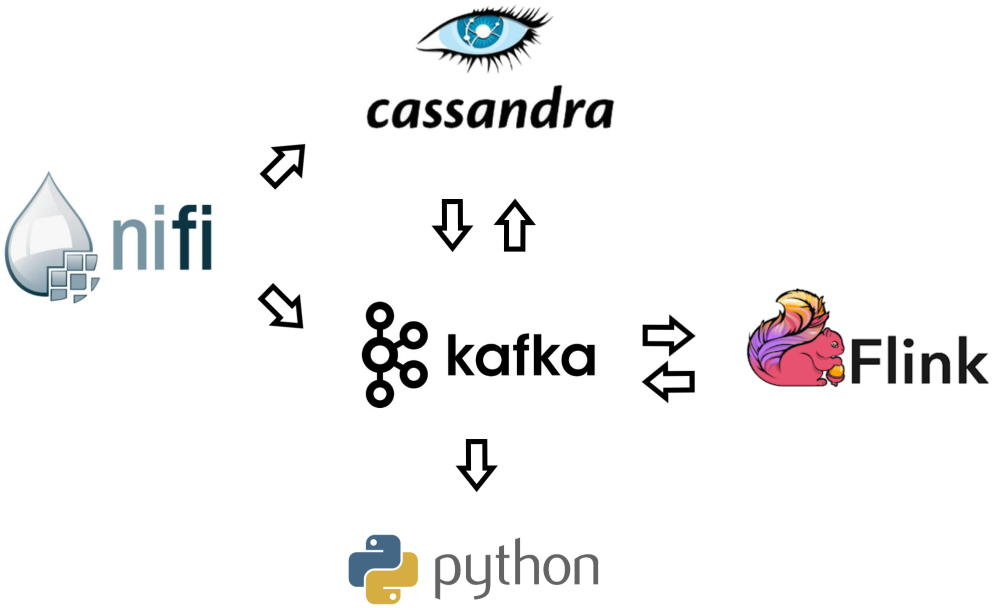


Figura 2.3: Componenti principali adottati.

NiFi	**FBP**	**Descrizione**
FlowFile	Information Packet	Un FlowFile rappresenta ogni oggetto che si muove attraverso il sistema e per ciascuno di essi NiFi tiene traccia di una mappa di stringhe degli attributi chiave-valore e del contenuto associato di zero o più byte.
Processore	Black Box	Il processore è colui che svolge effettivamente il lavoro e può eseguire una combinazione tra routing, trasformazione e mediazione dei dati tra sistemi. L'input corrisponde agli attributi di un determinato FlowFile entrante e al suo flusso di contenuti. I processori possono lavorare su zero o più FlowFile e durante la loro esecuzione possono effettuare commit o rollback di tale lavoro.
Connessione	Buonded Buffer	Le connessioni forniscono il collegamento tra i processori. Questi fungono da code e consentono a vari processi di interagire a velocità diverse. Queste code possono avere una priorità dinamica e possono avere limiti superiori sul carico.
Controllore del flusso	Scheduler	Il controllore del flusso mantiene la conoscenza di come i processi si collegano e gestisce i thread e le allocazioni che utilizzano tutti i processi. Ha quindi la funzione di broker, che facilita lo scambio di FlowFile tra processori.
Gruppo di processi	Subnet	Un gruppo di processi è un insieme specifico di processi e relative connessioni, che possono ricevere dati tramite porte di input e inviare dati tramite porte di output. In questo modo, i gruppi di processi consentono la creazione di componenti completamente nuovi semplicemente per composizione di altri componenti.

Tabella 2.1: Concetti su cui si basa NiFi che rispecchiano l'idea di una FBP

Capitolo 3

Realizzazione dell'infrastruttura

Per la realizzazione di questo tipo di compito è fondamentale che la ricezione dei dati avvenga 24/24. La strategia adottata non richiede di accedere allo storico del trend del Bitcoin per analizzarlo e scoprire i pattern che l'hanno governato nel periodo di tempo analizzato, ma ciò che ci serve viene direttamente preso in tempo reale con tutte le successive analisi che verranno effettuate e rimodellate nel tempo. A questo scopo ci servirà un server sempre operativo su cui installare la nostra architettura. In questo capitolo andremo ad implementare l'architettura pensata al capitolo precedente, in modo molto generico ci si può riferire alla figura 2.3. Implementeremo passo passo tutte le funzionalità necessarie per essere in grado di predire il futuro andamento del bitcoin, interlacciando i vari componenti e continuando a garantire il corretto funzionamento anche in presenza di guasti. Proprio per questo motivo, in aggiunta ai framework scelti, si è preferito lasciarsi appoggiare da Apache Zookeeper. Andiamo con ordine.

3.1 Data source

Partiamo dal principio; uno dei fattori su cui abbiamo deciso di costruire la nostra analisi sono le fluttuazioni in tempo reale del bitcoin, che nel tempo diventeranno lo storico. L'idea di conoscere il trend della criptovaluta potrebbe permetterci di scoprire che è soggetta a pattern specifici.

Un evento ricorrente o le conseguenza di una certa mentalità potrebbero generare un simile e riconoscibile frammento di andamento, che se riconosciuto fornirebbe una certa sicurezza nella previsione futura. Non c'è dubbio che come prima cosa sia necessario conoscere i valori in tempo reale, che oltretutto serviranno come risposta al set di allenamento con cui andremo a creare il modello per l'analisi in machine learning (Capitolo 4). Come fattore aggiuntivo abbiamo deciso di considerare le informazioni ricavate da Twitter. Se con il riconoscimento dei pattern si ha una discreta possibilità di prevedere i valori successivi, allora se si riuscisse a conoscere direttamente le cause scatenanti quei comportamenti e non solo, sarebbe un lavoro ancora più facile. Per ora ci limiteremo ad assimilare tweet inerenti al bitcoin, per creare una possibile linea guida, successivamente sarà possibile estendere le fonti a qualsiasi social o blog, puntando a conoscere idealmente il pensiero della collettività.

3.1.1 Alpha Vantage

La sorgente scelta per acquisire la serie temporale del bitcoin è Alpha Vantage [24], grazie alle API messe a disposizione è possibile ottenere dati in formato json o csv. In ogni caso si sarebbe potuto utilizzare qualsiasi altra applicazione con API free come coinmarketcap o yahoo finance. L'ipotesi di web scraping invece è stata esclusa a favore di una più semplice creazione di un client per chiamate REST. Per informazione il web scraping è una tecnica di estrazione dati per mezzo di programmi software che simulano la navigazione umana, utile soprattutto per quanto riguarda dati non strutturati [25].

Ritornando ad Alpha Vantage, sono disponibili una vasta serie di azioni quotate e valute sia fisiche che digitali, tra cui il bitcoin ovviamente. Ciò che ci serve è effettuare una richiesta che ci dia come risposta un informazione come quella in figura 3.1, possiamo trovare la data con il valore attuale (exchange rate) della valuta in questo caso in euro, il prezzo di vendita e di acquisto sono riferiti all'exchange in questione e non ci interessano. Per ottenere questo tipo di risposta è necessaria una richiesta intraday, ovvero che vari all'interno della stessa giornata. In particolare si può ottenere un valore aggiornato ad ogni minuto, ma bisogna far attenzione che il numero di richieste per un API key gratuita è limitata. Alpha

```
{
    "Realtime Currency Exchange Rate": {
        "1. From_Currency Code": "BTC",
        "2. From_Currency Name": "Bitcoin",
        "3. To_Currency Code": "EUR",
        "4. To_Currency Name": "Euro",
        "5. Exchange Rate": "6292.15215600",
        "6. Last Refreshed": "2020-04-21 12:38:03",
        "7. Time Zone": "UTC",
        "8. Bid Price": "6291.96771600",
        "9. Ask Price": "6292.65014400"
    }
}
```


Figura 3.1: Risposta di Alpha Vantage a una chiamata API intraday.

Vantage ha le condizioni meno restrittive tra le varie possibilità analizzate e non permette più di 5 API request al minuto e non più di 500 al giorno [26], per il limite al minuto non ci sono problemi, per quello giornaliero invece limita a 1 richiesta ogni 4 minuti, che potrebbe comunque essere accettabile, ma per avere qualche dato in più si è optato per richiedere due diverse chiavi, arrivando così a poter effettuare una richiesta ogni 2 minuti.
Spostiamoci su NiFi Per effettuare questo tipo di chiamata ci affideremo al comodo processore di NiFi: InvokeHTTP, ogni processore non è altro che un programma java prefabbricato, in questo caso basterà configurare l'endpoint HTTP, specificando l'URL di destinazione e il metodo HTTP.

3.1.2 Twitter

Per riuscire a captare tutti i possibili tweet inerenti al bitcoin possiamo sfruttare l'API di twitter per developers, utile per poter sfruttare le tecnologie che hanno a che fare coi tweet. Tutti i dettagli possono essere consultati nella documentazione messa a disposizione da Twitter [27]. A noi interessa riuscire ad ottenere un flusso continuo di tweet servendosi di un client che faccia delle richieste sfruttando il protocollo HTTP. Esistono 3 modi per accedere ai dati di twitter:

- Search API
- Streaming API

- Firehose

Shearch API permette di accedere all'interro archivio dei tweet già esistenti, con la possibilità inserire dei filtri attraverso un qualche criterio di ricerca che permette di specificare parole chiave, posizioni geografiche e specifici username. Il numero di richieste da poter rivolgere all'API nell'arco di una finestra di tempo (15 minuti) è limitato, a seconda del tipo di richiesta, inoltre esistono tre livelli di API di cui solo il primo è free. La limitazione consiste di quanto indietro nel tempo si possono cercare tweet.

Streaming API è in grado di servirsi di tweet che sono appena stati generati. Anziché fornire dati in batch tramite richieste ripetute, come ci si aspetterebbe da un'API REST, viene aperta una singola connessione e ogni volta che un nuovo tweet combacia coi criteri di ricerca, viene inviata la nuova risposta sempre sfruttando la connessione già creata. Ciò si traduce in un meccanismo di consegna a bassa latenza che può supportare una velocità effettiva molto elevata. Potenzialmente con questa strategia si possono ricevere uno spropositato numero di tweet, considerando che vengono scritti oltre 500 milioni di tweet al giorno, con una cadenza media di 6000 tweet al secondo [28]. Per questo motivo in realtà è presente una limitazione dovuta ai limiti dell'infrastruttura Twitter, che riducono le risposte fornite solamente a un campione dei tweet, che corrisponde ad un sottoinsieme del reale numero che ha fatto match. Alcuni studi [31] hanno stimato che l'ammontare di tweet ricevuti si aggiri intorno all'1% con leggere variazioni in base ai criteri di ricerca e al traffico corrente.

L'inconveniente della limitazione del numero di tweet considerati è rimediabile con l'utilizzo di Twitter Firehose. Come lo Streaming API permette di fornire dati real-time ma allo stesso tempo è in grado di garantire la consegna del 100% dei tweet, grazie alla gestione di due fornitori di dati: GNIP e DataSift. Il prezzo da pagare per questo tipo di servizio è il non essere free, per questo motivo ci accontenteremo di usare la streaming API

tweet che farà push.

3.2 Sentiment Analysis

Capire che cos'è il bitcoin per poter capire quali sono le regole che lo muovono [32]. Il Bitcoin a differenza di azioni e obbligazioni, ha una fluttuazione molto più veloce e imprevedibile, l'alta volatilità fa si che il suo

valore possa oscillare molto più rapidamente rispetto al comportamento generale delle quote finanziarie. Si pensi alla sua ascesa fulminea nel 2017 che da meno di 1000$ è salito oltre i 15000$ per poi continuare ad avere pesanti oscillazioni che continuano tuttora. Questo tipo di oscillazioni rendono più difficile l'anticipazione dei sui prossimi movimenti rispetto ad un azione che assume un comportamento molto più stazionario. Allo stesso tempo però, grandi oscillazioni permettono grandi guadagni, questa caratteristica ha fatto gola a tutti gli appassionati del trading, spingendo sempre più gente a fare investimenti e renderlo sempre più famoso, contribuendo ad accrescere le sue impennate e le sue cadute. E difficile capire quale sia il valore effettivo del Bitcoin, il suo prezzo è altamente influenzato dalla speculazione e sta assumendo il valore che la gente gli attribuisce. Quando il Bitcoin scaturisce una sensazione positiva, ci sarà gente che lo comprerà facendo così aumentare il suo valore ed auto-avverando le proprie convinzioni, al contrario in presenza di una sensazione negativa, convinti di una bolla momentanea pronta a scoppiare, il suo valore sarà condizionato in tal senso.

Puntando su questo fattore si può approfittare della miriade di dati non strutturati forniti volontariamente e di dominio pubblico che ogni giorno vengono generati sui social network, è interessante pensare di poterli sfruttare ai fini della nostra analisi, anche solo per verificare se e quanto le diverse opinioni influiscano macroscopicamente sul trend del bitcoin. Considerando anche che la propria idea può essere causa del condizionamento di una porzione di popolazione ma è anche il frutto di un elaborazione soggettiva che potrebbe avere un'importanza.

L'elaborazione di questo tipo di dati è molto costosa, un programma non è in grado di cogliere in modo accurato le intenzioni di un testo, magari arricchito di sarcasmo, slang ed errori di battitura. Le soluzione che abbiamo deciso di adottare è la sentiment analysis. L'analisi del sentimento, anche conosciuta come opinon mining è una tecnica utile ad ottenere informazioni aggiuntive, attualmente utilizzata in molti settori, in particolare in quello del marketing digitale e dell'analisi finanziaria. Lo scopo è analizzare un testo in linguaggio naturale al fine di identificare l'informazione presente per attribuirla a una classe di appartenenza, come suggerisce il nome, può essere usata per identificare, estrarre e quantificare i sentimenti che traspaiono da un testo scritto dall'opinion holder. I

vantaggi di questo tipo di approccio sono la scalabilità, la possibilità di un analisi in tempo reale e l'uso di criteri coerenti. Gli umani infatti tendono ad essere fortemente influenzati dai propri pensieri ed esperienze personali che li porta a concordare sul sentimento di uno specifico testo solo nel 60-65% dei casi. Affidandosi invece alla sentiment analysis, ad ogni dato sono applicati gli stessi criteri, gli errori sono ridotti e la coerenza dei dati è migliorata. Nel dettaglio attribuiremo ad ogni tweet captato un tipo di polarità che può variare tra positiva, negativa o ininfluente. Il valore ininfluente potrà essere assegnato sia se l'informazione non è classificabile in uno degli altri casi, sia se il testo in questione non sia inerente all'ambito del bitcoin nonostante la presenza della parola in questione.

3.2.1 VADER

Per effettuare quest'operazione ci avvaleremo della VADER (Valence Aware Dictionary and sEntiment Reasoner) Sentimant Analysis, è uno strumento di analisi del sentiment basato su regole e lessico che è specificamente in sintonia con i sentimenti espressi nei social media. É completamente open-source sotto la licenza MIT [33]. L'algoritmo che viene sfruttato tiene conto di regole grammaticali e sintattiche incorporando valori derivati empiricamente dall'impatto di ciascuna regola sull'intensità percepita del sentimento. Nel file "vader_lexicon" sono presenti parole e simboli (come emoticon) di riferimento 3.2. Non è presente qualsiasi parola, e sono pensate per attribuire un sentimento positivo o negativo che varia da -4 a 4. Per cercare di valutare più accuratamente il sentiment per quanto riguarda contesti finanziari abbiano aggiunto qualche termine cardine che altrimenti verrebbe considerato come ininfluente, alcuni esempi positivi sono "resurgent", "heated", "speedy", altri negativi possono essere "falsified", "bottleneck", "depressing". Per la selezione delle parole ci siamo affidati a delle ricerche sul "market sentiment dictionary" [34] e sulla stima di parole positive/negative per prezzi di azioni [35]

3.3 Processori NiFi

Come anticipato sfrutteremo il framework di NiFi come data ingestor, creeremo due flussi paralleli che attingeranno da Alpha Vantage e Twitter per salvare i dati su due topic kafka.

```
2505  elegant 2.1     0.83066 [2, 2, 2, 1, 4, 1, 2, 3, 2, 2]
2506  elegantly       1.9     0.83066 [2, 1, 1, 3, 2, 2, 1, 3, 3, 1]
2507  embarrass       -1.2    1.66132 [-2, -2, -3, -1, -2, -2, 2, 2, -2, -2]
2508  embarrassable   -1.6    0.8     [-3, -2, -1, -3, -1, -1, -1, -2, -1, -1]
2509  embarrassed     -1.5    0.67082 [-2, -2, -1, -2, -1, -3, -1, -1, -1, -1]
2510  embarrassedly   -1.1    1.44568 [-2, -1, -2, -3, 1, -1, -1, -2, -2, 2]
2511  embarrasses     -1.7    0.78102 [-2, -3, -1, -2, -1, -3, -2, -1, -1, -1]
2512  embarrassing    -1.6    0.8     [-3, -1, -1, -1, -1, -2, -1, -2, -3, -1]
2513  embarrassingly  -1.7    0.64031 [-2, -1, -1, -2, -1, -2, -1, -3, -2, -2]
2514  embarrassment   -1.9    0.53852 [-2, -2, -1, -2, -2, -2, -2, -1, -3, -2]
2515  embarrassments  -1.7    0.64031 [-2, -1, -2, -1, -1, -2, -2, -1, -2, -3]
2516  embittered      -0.4    1.35647 [1, -2, -1, 1, -2, 2, 0, -1, 0, -2]
2517  embrace 1.3     1.34536 [3, 2, 1, 3, 2, -1, 2, 1, -1, 1]
2518  emergency       -1.6    2.05913 [-3, -3, -3, -3, -4, 2, 1, -1, 1, -3]
2519  emotional       0.6     1.0198  [1, -1, 0, 0, 0, 2, 0, 2, 2, 0]
2520  empathetic      1.7     1.1     [-1, 3, 2, 2, 2, 1, 3, 1, 2, 2]
2521  emptied -0.7    0.64031 [-1, 0, 0, 0, -1, -1, -1, -2, 0, -1]
2522  emptier -0.7    0.64031 [-1, 0, 0, 0, -1, -1, -1, -2, 0, -1]
```

Figura 3.2: Alcune parole con il relativo valore di sentiment. Prima è indicata la media e la varianza, poi 10 valori che sono stati attribuiti alla parola corrispondente in contesti differenti.

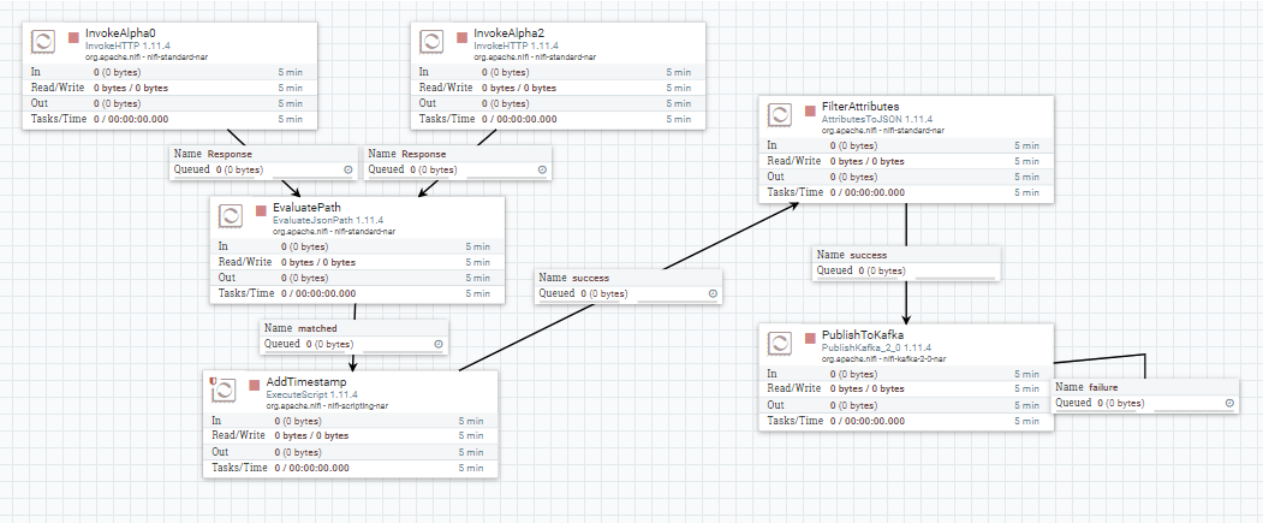


Figura 3.3: Flusso completo di processi NiFi utilizzati per trasformare i dati ottenuti da Alpha Vantage e consegnarli a Kafka.

Il primo flusso di dati considerati sarà quello con partenza da Alpha

Vantage, la figura 3.4 mostra la batteria di processi utilizzati. Per poter fare richieste al server di Alpha Vantage abbiamo creato un procesore InvokeHTTP, impostato in modo da fare una richiesta GET. Tutto quello che ci serve è ottenere l'API key per avere accesso alle richieste gratuite che si possono inviare al server, abbiamo una limitazione giornaliera di 500 richieste che si traduce in un interrogazione ogni 4 minuti. I valori dell'exchange rate che ci interessano sono aggiornati ogni minuto, eventuali richieste consecutive produrranno lo stesso risultato. Per avere una sensibilità più elevata e un maggior numero di dati preferiamo captare i movimenti ogni 2 minuti, a questo fine ci siamo procurati due differenti chiavi che utilizzeremo in parallelo una a due minuti dall'altra. I risultati che ci interessano possono essere ottenuti con la seguente richiesta: https://www.alphavantage.co/query?function=CURRENCY_EXCHANGE_RATE &from_currency= BTC&to_currency=EUR&apikey=MYKEY dove al posto di "mykey" è inserita la chiave effettiva. Per far si che i due processori si alternino ogni due minuti si è sfruttata la strategia CRON driven, in questo modo possono essere specificati sei numeri per indicare come impostando un timer, i momenti in cui far partire il processore. Il primo numero indica i secondi, poi i minuti, le ore, giorni del mese, mesi, giorni della settimana, anni. Nel caso del primo processore del nostro flusso: InvokeAlpha0 (fig. 3.4), abbiamo impostato la sua esecuzione al secondo uno di tutti i minuti multipli di 4 a partire dal minuto zero. Per InvokeAlpha1 i valori saranno "1 2/4 * * * ?". In tutte le connessioni tra un processore e l'altro si è fatto in modo che eventuali flowfile più vecchi di 500 secondi presenti nel flusso siano eliminati, in questo modo flowfile vecchi che per un qualsiasi motivo non hanno potuto proseguire non rimangono ad intasare la coda.
Il secondo processore NiFi che unisce input dei blocchi precedenti, ha lo scopo di formattare il flowfile in formato json generato da Alpha Vantage, attribuendogli ad ogni campo un nominativo, utile per una manipolazione successiva (fig. 3.5). Un campo necessario che non è presente nei nostri dati è il timestamp, avere un numero intero per la rappresentazione della data è una grande comodità per un computer. Per aggiungerlo abbiamo voluto provare ad inserire uno script scritto in Python che trasformi la data presente nelle risposte REST in timestamp, tramite il processore ExecuteScript. Abbiamo fatto in modo che ogni timestamp generato sia un multiplo di 120, in questo modo indipendentemente dalla data associata all'exchange rate, riusciamo ad orientarci suddividendo la linea temporale

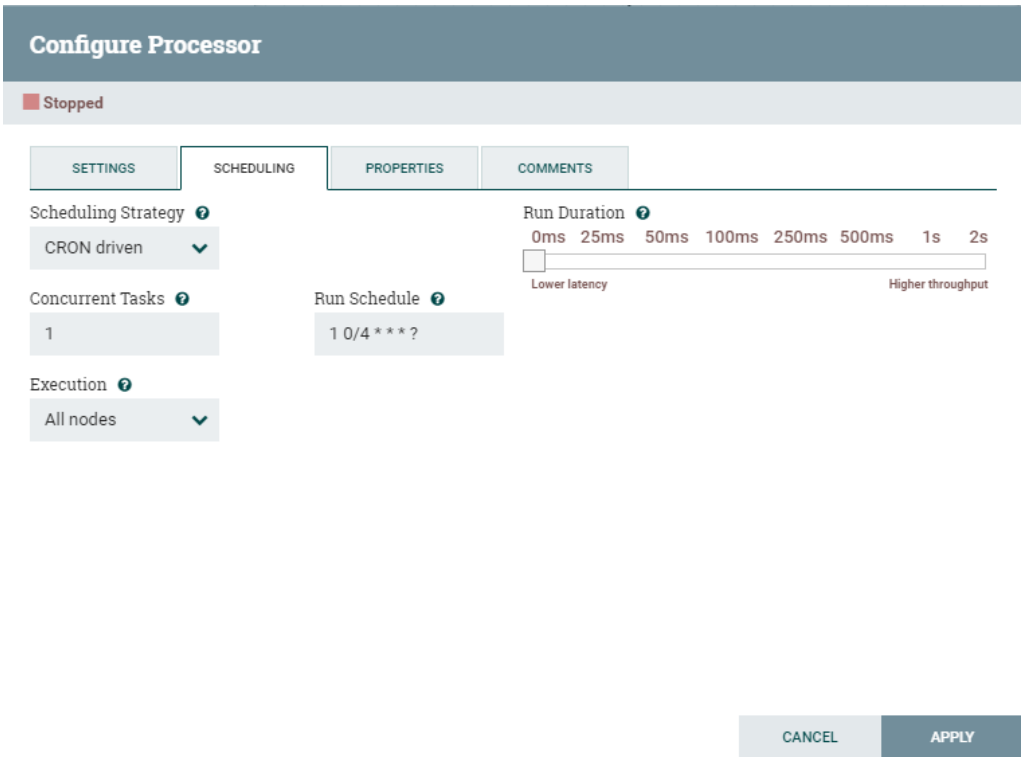


Figura 3.4: Impostazioni di scheduling per il processore InvokeAlpha0.

in frammenti da due minuti, cosa che ci sarà molto utile quando avremo a che fare con Flink. Al quarto strato di processori facciamo in modo di filtrare solo gli attributi che ci interessano, usando i nominativi assegnati dal processore EvaluatePath, per poi infine connettersi a Kafka al localhost:9092 e pubblicare sul topic "exchange" il flowfile risultante. Il topic può essere già esistente, ma se così non fosse verrà creato seguendo la configurazione di Kafka in server.propreties che vedremo al capitolo successivo. Questa volta per la connessione tra gli ultimi due processori non abbiamo impostato nessun tempo di scadenza, perché nel caso Kafka smettesse di funzionare, tutti i flowfile si cumulerebbero nella coda dell'ultima connessione e in un secondo momento quando Kafka sarà di nuovo disponibile tutti i file sospesi saranno recuperati evitando possibili perdite.

Guardiamo adesso il gruppo di processori che costituiscono il flusso che porta le informazioni da Twitter a Kafka, la figura 3.6 mostra l'interfaccia Nifi con ognuno dei blocchi di cui ci siamo serviti. Twitter usa il protocollo Oauth per autorizzare le richieste ricevute, verificando così l'identità del

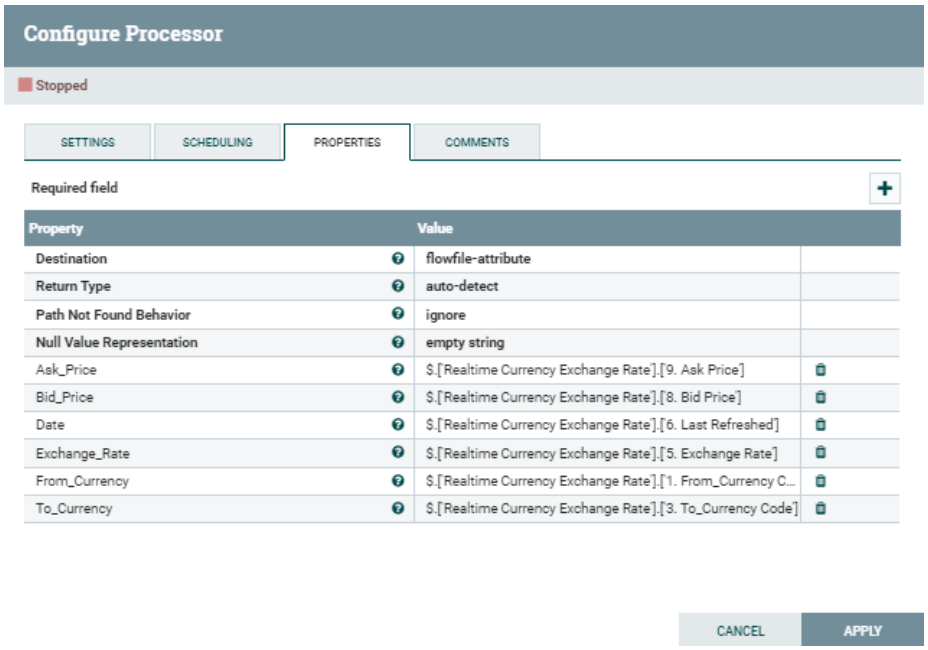


Figura 3.5: Dettagli processore EvaluateJsonPath.

mittente. Per l'autenticazione delle richieste è stato usato il bearer token, utile per chi ha bisogno solo informazioni read-only, in questo modo per l'autenticazione non è necessario specificare nessun informazione del proprio account Twitter developer, ma è sufficiente possedere un token valido, ottenibile registrandosi al sito ufficiale di Twitter developer. Servendosi di questo e creando un processore InvokeHTTP similmente a quanto fatto precedentemente, potremmo riuscire ad ottenere i nostri dati. C'è però un problema, nel 2017 Twitter ha raddoppiato la lunghezza dei messaggi, passando da 140 a 280 caratteri. Questa modifica non è prevista nel comodo processore preconfezionato, e tutti i messaggi con una lunghezza maggiore vengono troncati. Approfittiamo di questa limitazione per creare un processore ad hoc che soddisfi le nostre necessità. Sulla falsa riga dei processori esistenti [36], e avvalendoci della libreria twitter4J [37] abbiamo creato le funzionalità base di un processore che appare come in figura 3.7. É possibile impostare la lingua dei tweet a cui si è interessati, e si possono definire quali parole chiave devono essere presenti. Noi abbiamo scelto di rilevare tweet che contenessero "Bitcoin" e che siano esclusivamente in inglese, quest'ultima condizione è vincolata dalla sentiment analysis che non ha nel suo database altre lingue. In aggiunta bisogna specificare la

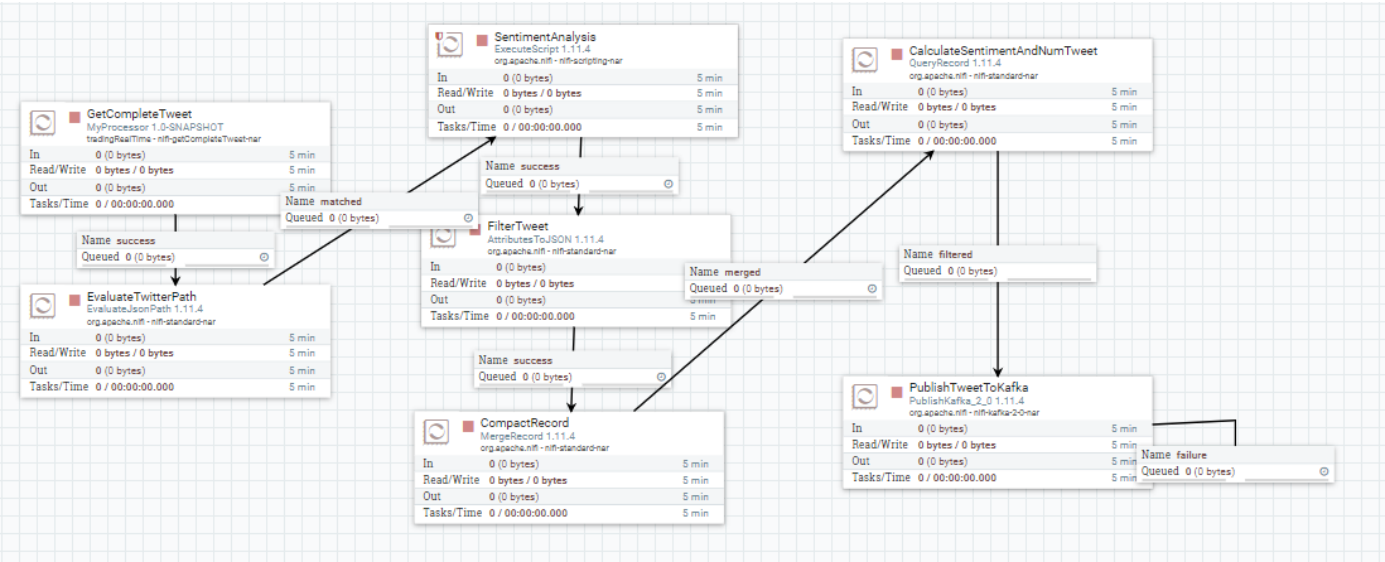


Figura 3.6: Flusso di processori NiFi che portano le informazioni da Twitter a Kafka.

key e il token che abbiamo generato dall'account Twitter.
Proseguendo nel flusso di processori troviamo EvaluateJsonPath, che come prima assegna dei nominativi ai vari campi del json restituito da Twitter. Con il processore successivo aggiungiamo il campo del sentiment, calcolato attraverso uno script grazie alla VADER sentiment spiegata al capitolo precedente. Per i dettagli abbiamo il codice in figura 3.8, il testo del tweet può trovarsi all'interno del campo full_text se è un testo abbastanza lungo, oppure nel campo full_text_retweeted se si tratta di un re-tweet, altrimenti se entrambi sono vuoti vuol dire che era un tweet corto che non ha avuto bisogno di una troncatura e quindi il processore GetCompleteTweet non ha avuto bisogno di spostarlo dal suo campo originario "text". Capito questo viene effettuata l'analisi del sentimento e trasformato il timestamp presente in ogni tweet in un multiplo di 120. Infine vengono assegnati come attributi i vari campi generati. Il processore successivo è un AttributeToJson, usato per filtrare gli attributi utili. Ci limiteremo a tenere timestamp, sentiment e numero di follower. I tweet captati possono essere abbastanza sparpagliati nel tempo e non costanti, inoltre ogni tweet ha poco significato se preso singolarmente. Tenendo in considerazione che il trend del bitcoin viene tracciato ogni due minuti, possiamo memorizzare nello stesso lasso di tempo il numero totale di

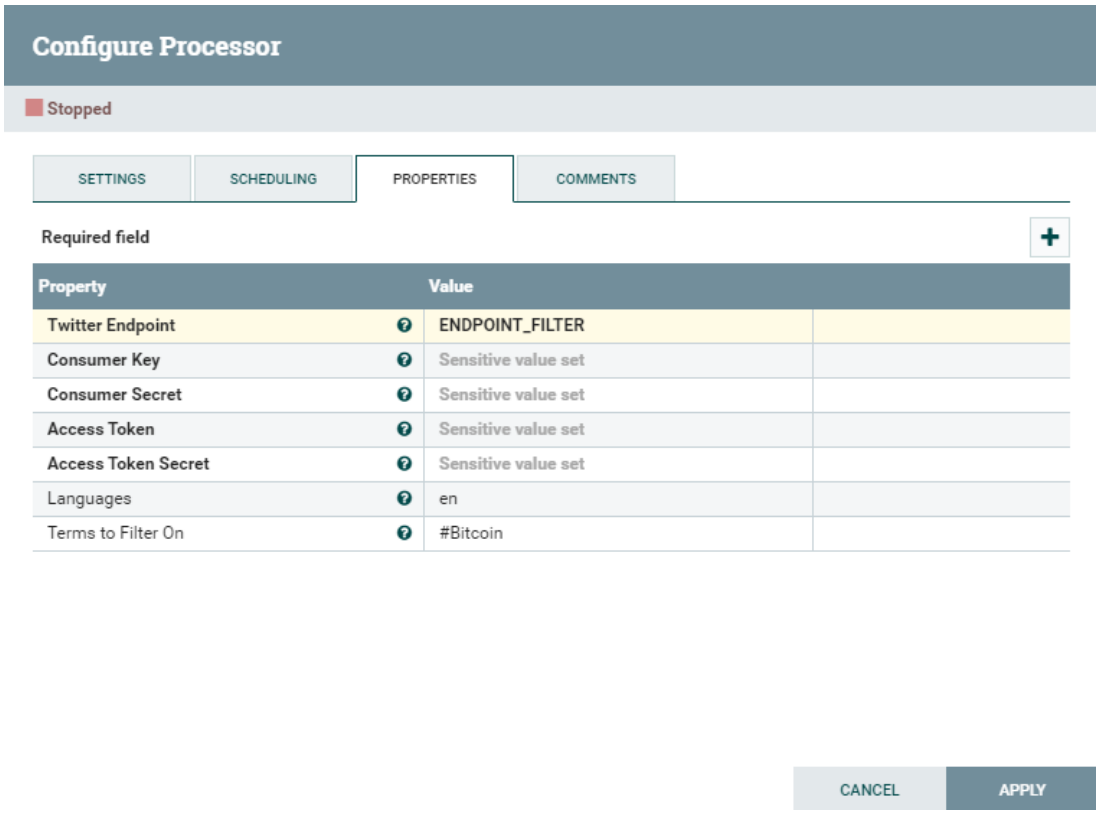


Figura 3.7: Proprietà processore personalizzato per l'ingestion da Twitter.

tweet rilevati con il loro relativo sentiment. Il processore successivo infatti è un MergeRecord, fa in modo che si cumulino due minuti di flowfile in un unico record per poi mandarlo al processore successivo che effettua la query: *SELECT SUM(sentiment*followers)/SUM(followers) AS sentiment, COUNT(*) AS numTweet, MAX(timestampSec) AS timestampSec FROM FLOWFILE*. Così abbiamo ottenuto il numero di tweet nei due minuti e il sentiment calcolato tramite la media dei sentiment dei vari tweet ponderata dall'influenza del singlo tweet corrispondente al numero di follower. Finalmente abbiamo le informazioni che che ci interessano e possiamo salvarle nel topic Kafka denominato "Twitter" e passare la palla a Flink.

```
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer

flowFile = session.get()

if flowFile != None:
    labels = ['neg', 'neu', 'pos', 'compound']
    sentiment = {}
    retweet = False
    if flowFile.getAttribute("id_retweeted") != '':
        retweet = True
    # case not truncated
    if flowFile.getAttribute("full_text") == flowFile.getAttribute("full_text_retweeted"): #both empty
        tweet = flowFile.getAttribute("text").strip()
    # case truncated
    else:
        tweet =  flowFile.getAttribute("full_text_retweeted").strip() if retweet else
flowFile.getAttribute("full_text").strip()

    for label in labels:
        sentiment[label] = analyzer.polarity_scores(unicodedata.normalize('NFKD', tweet).encode('ascii','ignore'))
[label]

    # escaping caracters if you want to save on Cassandra
    tweet = "$$" + tweet + "$$"
    # timestamp from millsecs to secs
    timestamp = int(flowFile.getAttribute("timestamp")) / 1000
    # timestamp multiple of 2 minutes
    timestamp -= timestamp % 120

    if sentiment['compound'] == 0: # case neutral
        followers = '0'
    else:
        followers = flowFile.getAttribute("followers_count_retweeted") if retweet else
flowFile.getAttribute("followers_count")
    # generate attribute for NiFi
    flowFile = session.putAttribute(flowFile, "full_text", tweet) if retweet == False else
session.putAttribute(flowFile, "full_text_retweeted", tweet)
    flowFile = session.putAttribute(flowFile, "completeSentiment", str(sentiment))
    flowFile = session.putAttribute(flowFile, "sentiment", str(sentiment['compound']))
    flowFile = session.putAttribute(flowFile, "timestampSec", str(timestamp))
    flowFile = session.putAttribute(flowFile, "followers", followers)
    flowFile = session.putAttribute(flowFile, "retweet", str(retweet))

    session.transfer(flowFile, REL_SUCCESS)
else:
    log.error('flowfile is empty')
```

Figura 3.8: Codice per effettuare una sentiment analysis in python.

3.4 MiNiFi

Blue Reply, l'azienda per cui sto svolgendo questa tesi, ha messo a disposizione i suoi server. É sorta però la necessità di lavorare in remoto per via della recente quarantena dovuta al COVID-19. Abbiamo quindi utilizzato PuTTY per emulare un terminale e gestire da VPN il server, ma per l'uso di NiFi è necessaria l'interfaccia utente disponibile dalla porta 8080 che ne rende semplice l'utilizzo. Per aggirare questa problematica ci siamo affidati a MiNiFi, anche questo è un progetto Apache, usato per avere una

gestione centralizzata col vantaggio di essere più leggero e meno impattante. Ci sono alcune differenze da tenere in considerazione passando Nifi che in MiNiFi non sono supportate [38]

- Gli imbuti
- Le relazione tra più sorgenti per una singola connessione
- I gruppi di processi
- Ogni processo richiede un nome univoco

inoltre la versione di MiNiFi utilizzata potrebbe non essere compatibile con alcune proprietà di processori con versione di NiFi superiore a 1.7.0, che sono sconosciute per MiNiFi. Per quanto riguarda le limitazioni, nel nostro caso non abbiamo creato un flusso che utilizzi simili caratteristiche ma potrebbe essere problematico il fatto che sia realizzato interamente con la versione 1.11 di NiFi. Per cui alcuni dettagli dovranno essere riadattati per la versione precedente. I passi che seguiremo per poter far funzionare NiFi sono i seguenti

1. Mantenere il flusso di processori creati con l'interfaccia NiFi
2. Salvare il flusso come template e scaricare il suo file XML
3. Usare Converter Toolkit [39] per convertire il file XML in un equivalente file config.yml
4. Spostare config.yml in $MINIFI_HOME/conf e avviare MiNiFi

Alcuni processori incompatibili sono stati ricreati in versione 1.7, convertiti in xml tramite salvataggio del template e aggiunto all'xml originale. Per farlo bisogna ricordarsi di modificare le variabili d'ambiente perché la vecchia versione di NiFi non è compatibile con jdk 11. Per rendere funzionante ogni processore bisogna controllare quali librerie utilizza e verificare se sono presenti nella cartella lib di minifi. Un modo veloce per individuare quali librerie sono utilizzate è analizzare il file .xml del template e cercare quali artifact sono utilizzati ed eventualmente aggiungere i file nar indicati. Nel nostro caso sono stati aggiunti i file nar in figura 3.9.

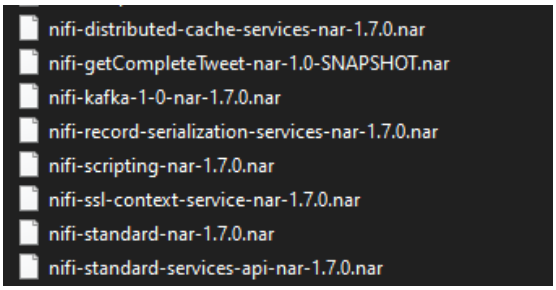


Figura 3.9: File .nar aggiunti nella libreria di MiNiFi.

3.5 Kafka

Kafka ha bisogno dell'aiuto di ZooKeeper per gestire il coordinamento tra daversi broker. ZooKeeper è un progetto open source della Apache Software Foundation in grado di fornire un servizio centralizzato che sia in grado di fornire servizi di livello superiore per la sincronizzazione, mantenimento della configurazione e denominazione in sistemi distribuiti, anche su cluster. É progettato per essere facile da programmare, è scritto in Java ed è stato pensato per sollevare le applicazioni distribuite dalla responsabilità di implementare i servizi di coordinamento da zero, ponendo attenzione a rendere affidabile l'adattamento di tutti i sistemi alle modifiche, evitando possibili race conditions e deadlock.

3.6 Flink

All'interno dell'applicazione Flink ci sono due consumer, il primo attinge i dati dal topic exchange di Kafka, mentre il secondo dal topic twitter. La politica di fruizione dei dati è "latest", in questo modo nel caso il programma venisse fermato, al riavvio ripartirebbe a prendere dati dal topic dalla posizione dell'ultimo offset. Infatti ogni consumer appartiene ad uno specifico consumer-group che ha un offset personale memorizzato. Il connettore usato verso Kafka è impostato con il meccanismo "exactly-once", assicurando che ogni messaggio sia recapitato una e una sola volta, gestendo in un caso di perdita non ci siano record duplicati e distinguendosi così dalla politica "at least once" in cui un messaggio può venire ritrasmesso ma ignorando la creazione di possibili duplicati.

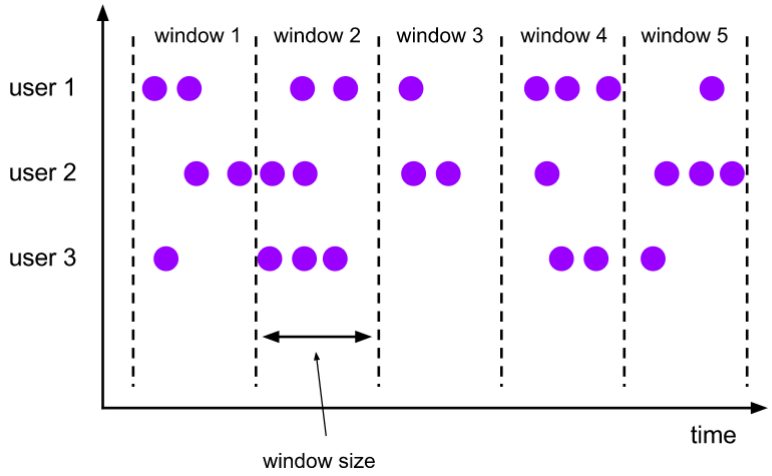


Figura 3.10: Visualizzazione grafica del comportamento tumbling-windows di flink. Lo stream è semplicemente diviso in finestre di una data ampiezza, è anche possibile che in determinate finestre non ci siano dati. [40]

Nella prima trasformazione del flusso, ad ogni record viene associato esplicitamente un timestamp, lo stesso già presente all'interno del record stesso preso da Kafka. Questo permetterà a Flink di conoscere l'esatta posizione temporale che ci servirà per effettuare le operazioni che andremo ad analizzare. Prima tra tutte: il join. L'unione tra i due flussi altro non è che un semi-join sul timestamp, infatti un istante di tempo che non ha corrispondenze nel flusso complementare, viene scartato. Non verranno così a crearsi record con delle informazioni parziali, anche se in generale non dovrebbero comunque venirsi a creare. Bisogna fare attenzione che potenzialmente i due flussi potrebbero essere sfalsati, e non dimentichiamoci che in quanto streaming, non hanno una fine. Se la chiave di join fosse il numero di tweet, e volessi unire tutti i record con lo stesso ammontare di tweet, dovrei considerare che in futuro, tra dati che ancora non mi sono arrivati, ci possano essere record che faranno match con dati già esistente. La lunghezza non limitata dello stream rende questa casistica inaccettabile, per questo è necessario specificare una finestra di tempo nel quale effettuare il join, come in figura 3.10, andando a limitare la quantità di tempo per la quale si osservano dati con lo scopo di effettuare il join.

Ora che abbiamo ottenuto unico flusso contenete tutti i dati ricavati dal-

le fonti, abbiamo come obiettivo il mettere a disposizione due nuovi topic con dati già pronti per effettuare l'analisi; partiamo dal più semplice.
Nel primo topic manterremo come informazioni oltre al timestemp solo l'exchange rate, il numero di tweet e il valore di sentiment. Questi valori che ricordiamo hanno una frequenza di 2 minuti e corrispondono ai 2 minuti precedenti, dovranno essere inglobati in record che terranno in considerazione 48 ore. Una prima soluzione potrebbe essere generare un messaggio ogni 48 ore, che comprenda al suo interno $48 * 60/2 = 1440$ record. Però è bene cercare di avere più dati possibile se si vuole eseguire un algoritmo di machine learning, anche ricorrendo alla data augmentation. Per questo considereremo una sovrapposizione tra un messaggio e l'altro con l'ulteriore vantaggio di considerare la variazione di trend presente tra i gruppi di due giornate. Il caso limite corrisponderebbe a uno slittamento di 2 minuti, ma riteniamo eccessiva una tale sovrapposizione, che genererebbe dati molto simili tra loro, per questo abbiamo scelto un offset di un ora. Per riuscire a raggruppare i dati in questo modo è sufficiente una sliding windows (figura 3.11): consideriamo i dati presi in una finestra di 2 giorni e ognuna di queste finestre è considerata ad intervalli di un ora. Questo ci permette di popolare il primo topic.
Prima di passare alla seconda parte facciamo notare alcune caratteristiche del nostro flusso. I dati all'interno di Kafka sono inseriti da NiFi man mano che il tempo passa e quindi sono ordinati per timestamp. Questa proprietà non è però necessaria a Flink, infatti avendo associato ad ogni record un timestamp, è in grado di ottenere in modo efficiente le finestre necessarie senza bisogno di un ordinamento esplicito. Inoltre anche in fase di popolamento dei topic i dati generati non sono necessariamente ordinati, mentre lo devono essere quelli all'interno di un singolo messaggio: requisito necessario per l'analisi con LSTM model, per rispettare l'ordinamento infatti si è ricorso ad un sort esplicito.

Per il secondo topic invece faremo in modo di inserire dei dati per un analisi leggermente diversa, che abbia a disposizione anche molte più informazioni. Se la nostra prima strategia era considerare ogni valore captato ad ogni 2 minuti, ora continueremo a mantenere la stessa frequenza ma all'interno di ogni record saranno compresse le informazioni di un intera ora. Questa strategia punta ad essere più precisi nella predizione ad intervalli orari, lasciando da parte le oscillazioni di periodi brevi come 2 minuti che teoricamente nascondono al loro interno più rumore e quindi un maggior

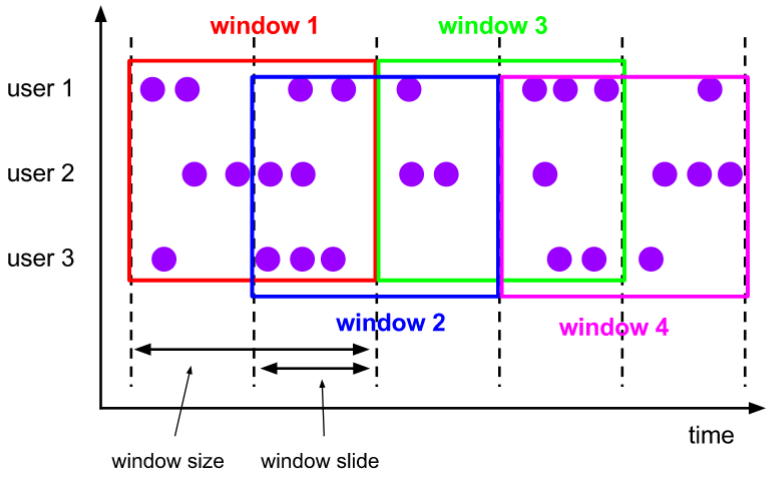


Figura 3.11: Visualizzazione grafica del comportamento sliding-windows di flink. Ogni finestra ha una durata temporale fissa e ad ogni slittamento possono generarsi sovrapposizioni se windows size è inferiore a window size. [40]

numero di falsi segnali. Come detto al capitolo precedente, calcoleremo le informazioni che ci permetterebbero di creare un diagramma a candele (candlestick), quindi i valori di open, close, high e low. In aggiunta a questi calcoleremo anche la media mobile, la varianza, l'RSI e alcuni indicatori basati sulla media mobile su più periodi: LWMA, SMMA e l'EMA (nella sezione 4.1.2 si studieranno i dati da qui ottenuti). L'exchange rate non sarà più necessario in quanto si riferirebbe ad un unico istante di tempo, e il numero di tweet e sentiment sarà esteso al contesto di un ora.
Per andare a generare ognuno dei nostri predictors abbiamo effettuato uno sliding-windows del tutto simile a quello usato per raggruppare i due giorni precedenti. Qui però considereremo intervalli di un ora con uno slittamento di due minuti, e con i timestamp che rispettano questa condizione, calcoliamo tutte le informazioni sopra descritte generando un unico record che le contenga tutte. Per andare più nel dettaglio, flink gestisce tutti i dati in parallelo, ognuno è considerato a se stante e ci sono delle variabili "accumulatore" che noi inizializziamo e con il massimo grado di parallelismo aggiorniamo l'accumulatore con le informazioni del dato originale oppure l'accumulatore con un altro accumulatore, per arrivare ad averne solamente uno finale. Per ottenere l'open come risultato alla fine

di ogni unione ci è bastato mantenere l'exchange rate con il timestamp minore, con strategie simili ricaviamo close, high e low, la media non può essere calcolata in questo modo perché necessita di tutti i dati contemporaneamente, per questo viene calcolato il count e la sum, per poi nel record finale combinarle per ottenere la media. Per la varianza è stato più dispendioso, prima per ogni finestra di un ora è stata calcolata la media e successivamente con una sliding-windows si è calcolata la sommatoria della differenza al quadrato tra ogni valore e la media di quella finestra, per poi al risultato finale dividere per il numero di elementi nell'ora (30). Discorso diverso si applica agli indici basati su più periodi, ogni periodo corrisponde ad un ora e per semplicità abbiamo considerato lo stesso range temporale per ognuno degli indici: 16 ore. Come caso esemplificativo vediamo il calcolo dell'RSI. Nello sliding-windows consideriamo 16 ore come ampiezza delle finestre ma eseguiamo un raggruppamento tra chiavi. Al posto di considerare tutti gli elementi come valori concorrenti a fornire un unico risultato, li dividiamo invece in gruppi che hanno in comune il fatto di avere un timestamp multiplo di un ora con punto di partenza allo stesso offset, attraverso un KeyedStream. Quindi i valori ottenuti ad un ora in punto saranno considerati un gruppo, così come tutti i valori ottenuti alle x e 2 minuti, ecc. Per ognuno di questi gruppi che contengono valori distanziati di un ora possiamo calcolare i nostri indici, per l'RSI ci serve semplicemente per ogni elemento del gruppo, se corrisponde ad un caso di rialzo (close maggiore dell'open) o di ribasso, e quanti casi di rialzo e di ribasso ci sono. Tutte operazioni effettuabili con "aggregate" e l'uso dell'accumulatore.

3.7 Cassandra

Per il salvataggio dei dai in modo persistente ci siamo affidati a Cassandra. In un primo momento, data la semplicità d'uso NiFi abbiamo preso il contenuto dei due topic di partenza, contenenti le informazioni di Alpha Vantage e Twitter, e le abbiamo salvate tramite due flussi in Apache Cassandra. La figura 3.12 mostra la panoramica dei processori utilizzati nell'interfaccia utente di NiFI. Il primo processore è un KafkaConsumer, prende i dati dai topic corrispondenti creando con consumer group con politica latest. Il FlowFile generato può venire modificato per avere una formattazione tale da renderlo un json per poter identificare gli attributi

con EvaluateJsonPath. Infine l'intero flowfile viene trasformato in una query come: "INSERT INTO tweet (Timestamp, Sentiment, NumTweet) VALUES ($'timestamp', $'sentiment', $'numTweet');". L'ultimo processore, il PutCassandraQL, è creato per ricevere in input un FlowFile il cui contenuto dovrebbe essere un comando CQL, in questo modo riusciamo ad eseguire nel keyspace di Cassandra specificato, le istruzioni per il popolamento del database.
L'alternativa all'uso di NiFi per veicolare le informazioni su Cassandra poteva sfruttare il servizio di Schema Registry, disponibile in maniera open dalla piattaforma Confluent, ideata dagli stessi sviluppatori di Kafka. Per sfruttare questo servizio abbiamo bisogno che i dati su Kafka siano in formato Avro, quindi salviamo lo schema desiderato attraverso l'API REST dello Schema Registry. Il processo AvroConverter serializzerà i dati in entrata su Kafka con l'uso di un producer. Una volta su Kafka basterà l'uso di un apposito tool molto comodo, in grado di pubblicare automaticamente ogni messaggio su Cassandra non appena ricevuto. Si tratta di Kafka Connect, in particolare Cassandra Sink, per esportare verso Cassandra.

3.8 Conseguenze shut down

L'intera infrastruttura per come è stata pensata dovrebbe essere eseguita su un server che rimanga attivo 24 ore su 24 e che sia tollerante ad ogni genere di errore senza che nessun componente termini la sua esecuzione. Poniamoci invece nel caso che questo possa accadere, così da poter analizzare se tutto il lavoro svolto possa essere continuato come se nulla fosse, o se senza una continuità il lavoro debba essere ripreso da zero. Un modo in cui questo possa verificarsi può essere ad esempio nel caso in cui saltasse la corrente e non si avesse a disposizione un gruppo di continuità, o più semplicemente un crash dovuto alla presenza di bug o sviste che spesso possono essere presenti in applicazioni "giovani" che non hanno avuto esperienza con l'imprevedibilità dei casi reali.

- Per quanto riguarda NiFi è facile intuire che nel frammento di tempo in cui rimarrà disattivo, non sarà possibile captare i dati dalle sorgenti e causerà all'interno dei topic di Kafka un buco di dati, che bisogna gestire in modo da non creare problemi. Nel caso fossero gli altri componenti a riscontrare qualche malfunzionamento, tratteremo solamente di Kafka, perché unico componente direttamente connesso.

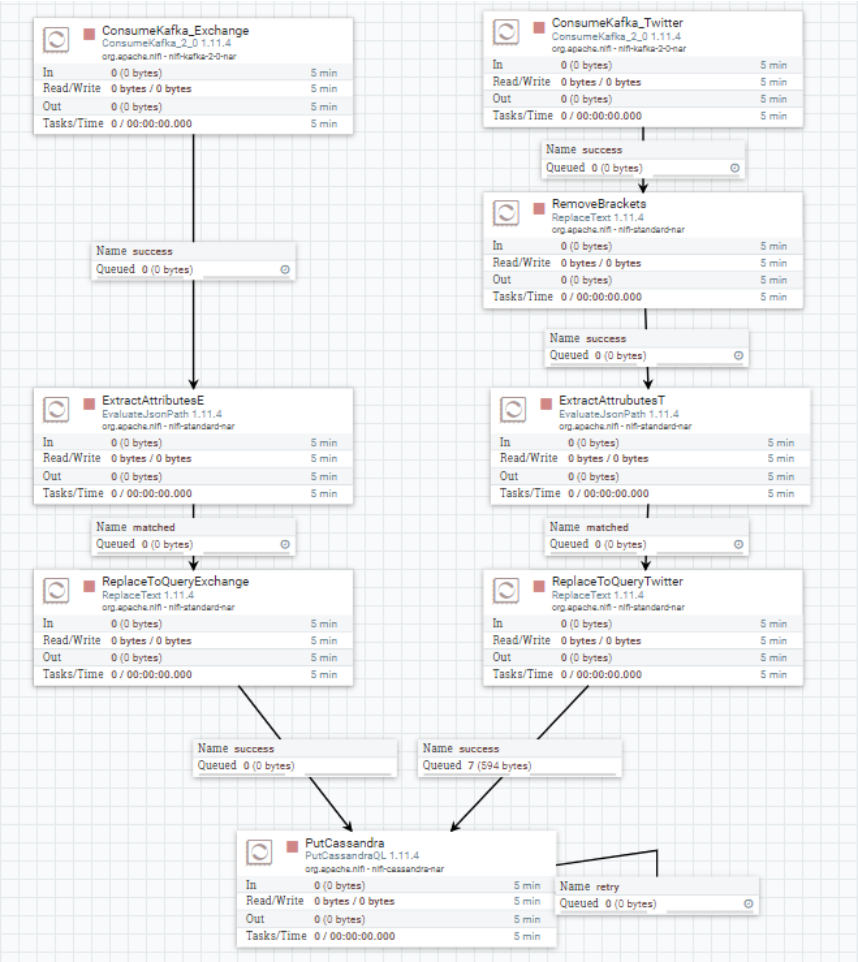


Figura 3.12: Schema di processori NiFi che attingono dati da Kafka e li inseriscono in Cassandra.

In quel caso tutti i flowfile si cumulerebbero nella coda della connessione degli ultimi due processori della catena, ovvero quelle che contengono il flow-file finale da pubblicare su Kafka, come la connessione che porta a PuplishTweetToKafka in 3.6. Una volta che Kafka sarà di nuovo disponibile, tutti i dati cumulatisi in quella coda si sbloccheranno e saranno serviti senza ulteriori attese e senza quindi risultare in nessuna perdita. Bisogna tenere presente che però, il limite di flowfile in una connessione è 10000 (valore incrementabile), quindi raggiunto quel numero i dati successivi andranno persi. Nel nostro caso per arrivare ad un tale numero di flow-file devono passare quasi 14 giorni.

- Kafka di per se non presenta alcun problema, grazie ai suoi log e al suo fattore di replica è in grado di ritornare alla situazione in cui si trovava prima dello shut down, compresi gli offset dei vari group che attingono dai suoi dati. Non ha perdita di dati e è pronta a recuperare eventuali messaggi arretrati da NiFi. Inoltre su Cassandra vengono memorizzati tutti i dati man mano che popolano Kafka.

- Per Flink che ha anche la funzione di mettere insieme i dati di due giorni per usarli come dati di training si ha qualche problema. Se si vuole generare un dato utile c'è bisogno che alle spalle ci siano almeno due giorni di dati, quindi dopo uno shut down bisognerà attendere questi due giorni di accumulo prima di ritornare a regime, avendo spezzato blocco di dati precedente. Per questo basterà usare una politica earliest che permetta di accedere ai dati precedenti con conseguente creazione di possibili duplicati. Se invece il blocco di dati mancanti è perso (NiFi ha avuto un periodo di fermo) allora l'attesa è inevitabile. Analizziamo anche cosa succederebbe in presenza di buchi nei dati di input, come abbiamo accennato prima parlando di NiFi. É importante per l'analisi in machine learning che ogni dato abbia il proprio storico, non ammettendo i dati incompleti, ma per come è strutturato Flink questa possibilità non può verificarsi. Infatti effettuando delle tumbling windows (fig. 3.10) con ampiezza di due giorni; fintanto che all'interno di questa finestra è presente almeno un informazione, il valore in uscita viene emesso. Questo valore viene calcolato in funzione di tutto quello presente all'interno della finestra, quindi considerando il valore di close della candela, potrebbe essere assegnata la stessa quantità della finestra precedente: se ci trovassimo in presenza di un buco di ampiezza superiore allo slide della finestra. Così anche tutti gli le altre componenti sono calcolate considerando i valori disponibili.

- Se terminasse lo script per l'analisi di machine learning, come per Kafka, il danno non sarebbe sostanziale. Sarebbe necessario avere a disposizione tutti i dati ottenuti fino al momento dell'analisi per poter eseguire un training con tutte le informazioni disponibili, per questo si potrebbe impostare una politica "earliest" per l'offset del consumer incaricato di catturare i dati. Ma periodicamente lo stato del modello viene salvato, con i pesi aggiornati da tutti i dati di training che si

è utilizzati. Quindi recuperato il modello lo si può utilizzare ancora sia per effettuare previsioni che per aggiornarlo con ulteriori dati d itraining.

Capitolo 4

Previsione serie temporale tramite machine learning

In questa sezione ci focalizzeremo sull'analisi dei dati a nostra disposizione, individuandone le caratteristiche per essere in grado di predire l'andamento del Bitcoin tramite machine learning. L'apprendimento automatico (ML) è una parte dell'informatica in cui l'efficienza di un sistema si migliora da sola eseguendo ripetutamente le attività basandosi sui dati anziché essere programmata esplicitamente dai programmatori. In generale esistono tre tipi di tecniche che si possono adottare, il supervised learning in cui l'algoritmo si allena su un set di features-label per imparare a predirre le label, l'unsupervised learning dove si cerca di comprendere la forma dei dati per attribuire ad ognuno la sua label senza una conoscenza pregressa, e il reinforcement learning in cui l'algoritmo interagisce con l'ambiente ed impara a reagire nel migliore dei modi sotto ricompense e punizioni. Per risolvere il nostro problema dobbiamo servirci della prima tecnica, quella del supervised lerning; prevedere la label corrispondente al valore del Bitcoin basandosi su caratteristiche come il numero di tweet e sentiment, avendo la possibilità di allenarsi su dati in cui quest'accoppiamento è certo. La particolarità di questo tipo di problema è che la risposta che vogliamo prevedere è anche una caratteristica usata per prevederla. Nella mia carriera universitaria mi sono cimentato nell'applicazione di algoritmi di classificazione e regressione per predire valori futuri (quindi supervised

learning), ma con le time series l'approccio da adottare è diverso.
Il problema che distingue una time series da un semplice elenco predictors/response è come suggerisce anche il termine, la dipendenza dal tempo. Nel primo caso infatti i dati hanno un certo ordine: il valore del Bitcoin all'istante t2 è successivo a quello dell'istante t1 ed è importante che sia così. Quando si allena un modello supervised non è importante l'ordine, ogni dato è di per se indipendente, infatti sono possibili operazioni come la cross-validation, che richiedono di poter mescolare liberamente i dati senza preoccuparsi di una gerarchia temporale. Per risolvere questo tipo di problemi è possibile raggruppare in un unico record i valori di un arco di tempo scelto, in modo da segmentare l'intero trend in tanti segnali separati e sfruttare questi segnali in modo indipendente tra di loro, riconducendoci in questo modo alla tipologia dei dati non dipendenti dal tempo. Analizziamo ora una differenza intrinseca rispetto a un problema supervised, come detto sopra un modello supervisionato si allena su coppie predictors-response, per poi essere applicato a predictors di cui non si conosce la response, per prevederla. Invece in una time series quando si va a sfruttare il modello che è stato allenato, volendo predire in istanti di tempo non immediatamente successivi, non è possibile usufruire dei predictors come appoggio poiché anch'essi sono delle incognite al pari del dato che ci interessa. Quindi si deve predire una serie di dati consecutivi sapendo solamente che sono la continuazione di un precedente flusso di dati, diminuendo la precisione rispetto a un problema classico.
Comprese le problematiche nel lavorare con serie temporali, prima di adottare qualsiasi metodo, passiamo alla profilazione dei dati. Conoscere i propri dati è il primo passo fondamentale per la realizzazione di un modello funzionante.

4.1 I Dati

I dati a nostra disposizione hanno subito una serie di trasformazioni prima di essere collocati da Flink in due differenti topic di Kafka per affrontare due casistiche che considerino record a diverse cadenze temporali.

2020-08-12 09:01:03 1597222800 0.6632977 109 9739.789, 2020-08-12 09:03:02 1597222920 0.25191477 119 9738.661,
2020-08-12 09:05:02 1597223040 0.35303968 160 9749.014, 2020-08-12 09:07:02 1597223160 0.46794593 103 9755.109,
2020-08-12 09:09:02 1597223280 0.51174974 98 9753.821, 2020-08-12 09:11:02 1597223400 0.49255568 99 9755.749,
2020-08-12 09:13:02 1597223520 0.63793796 92 9735.699, 2020-08-12 09:15:06 1597223640 0.56639254 85 9746.632,

Figura 4.1: Frammento di un singolo dato dal topic joinedDataSimple. I valori corrispondo a: data, timestamp, sentiment, numero tweet, exchange rate.

4.1.1 Dati ogni due minuti

Il primo topic, che chiameremo "simple", mantiene l'originale cadenza di due minuti. Ogni dato però contiene al suo interno i valori di due giornate, per un totale di 1440 elementi che si presentano come in figura 4.1. Ciò che abbiamo a disposizione sono la data e l'ora, anche in formato timestamp per essere usata più facilmente, poi abbiamo il valore dell'exchange rate, il numero di tweet rilevati nei due minuti di riferimento e la media del sentiment di quei tweet, calcolata come già spegato nello scorso capitolo attraverso una ponderazione in base al numero di follower.
Passiamo ora ad analizzare la correlazione fra le variabili. Alcuni algoritmi richiedono che i predictors siano scorrelati fra loro, ma quest'informazione potrà già anticiparci se qualche variabile è fortemente legata all'exchange rate, dandoci una sicurezza in più sulla buona riuscita dal nostro modello. La figura 4.2 nega ogni tipo di relazione tra sentiment e exchange rate, mentre per il numero di tweet c'è almeno un minimo di legame. Nel lasso di tempo considerato il Bitcoin in generale assume un trend positivo, per questo motivo ha una relazione molto forte con il timestamp, che a scanso di equivoci non sarà un predictors. Questo ci fa anche riflettere sulla validità del modello che andremo a creare, in generale quando si usano algoritmi di machine learning maggiori sono i dati a disposizione, migliore sarà la nostra conoscenza e di conseguenza anche il modello. Con le serie temporali c'è l'ulteriore complicazione del tempo, anche avendo la totale conoscenza delle fluttuazioni dell'ultimo mese, se in quel periodo ha sempre avuto un valore crescente, per quanto si abbiano a disposizione i migliori predictors non si sarà in grado di riconoscere i pattern di discesa perché questi eventi non si sono ancora mai verificati. Il nostro obiettivo comunque è essere in grado vedere attraverso le variazioni intraday che potrebbero esulare dal trand di fondo essendo soggette a una grande varianza che permette delle oscillazioni continue anche importanti. Per

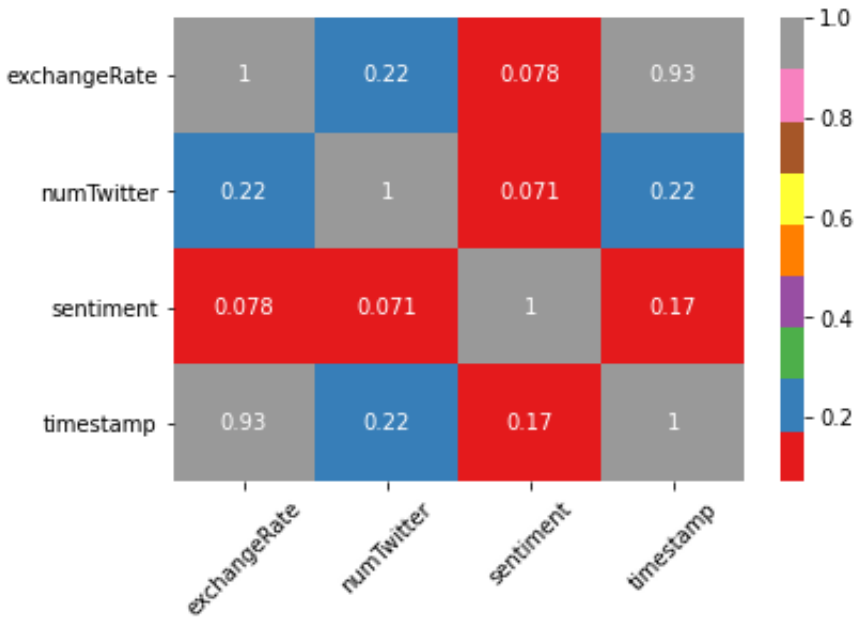
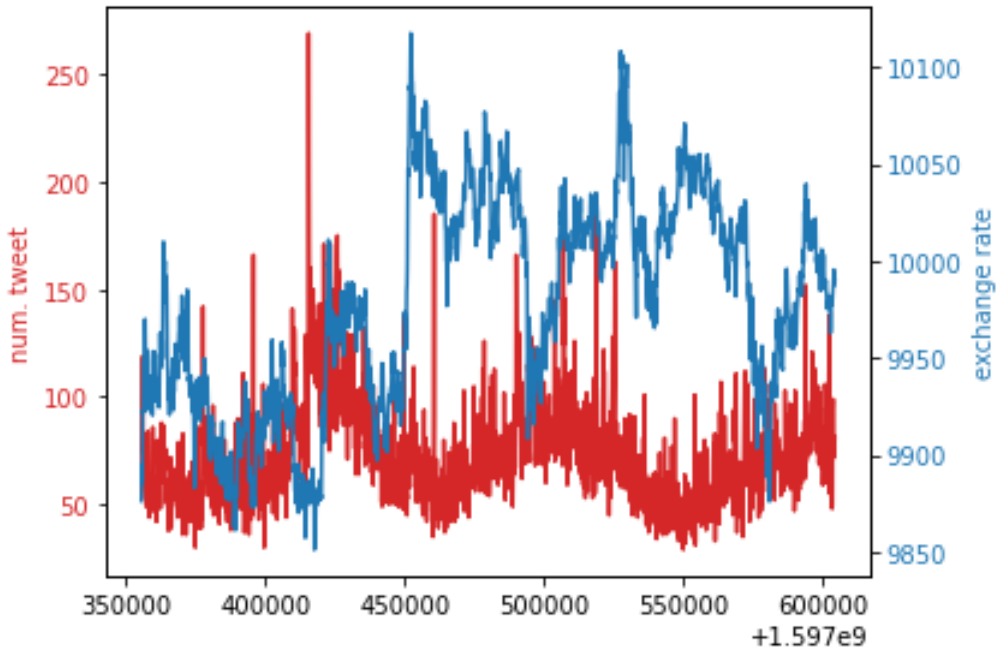


Figura 4.2: Heatmap della correlazione delle variabili caso simple.

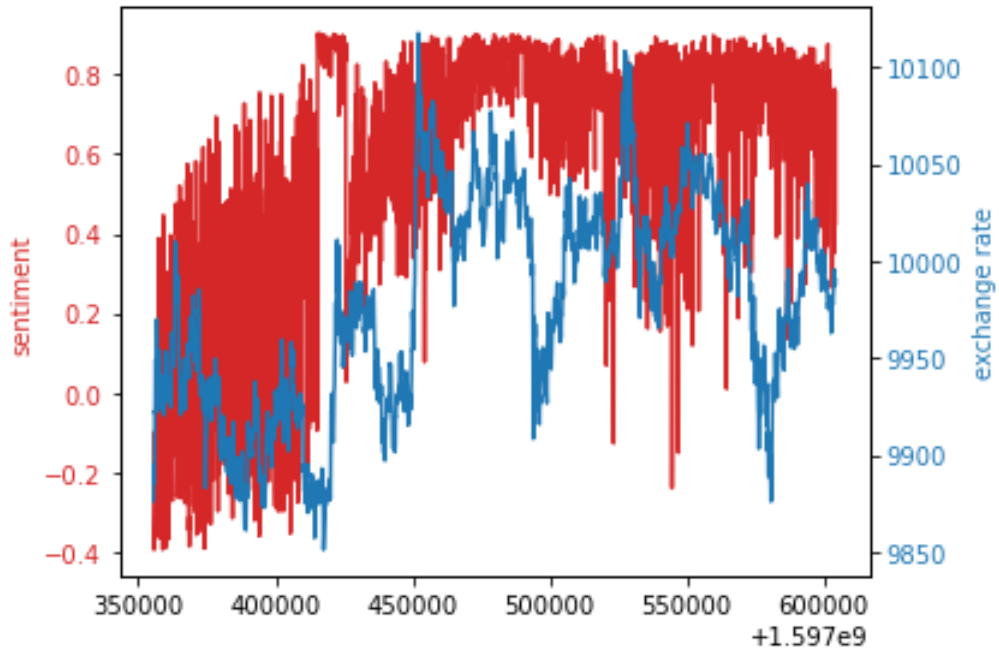
avere una visione più chiara dei dati a nostra disposizione, la tabella4.1 mostra qualche informazione statistica sulle nostre variabili, e la figura 4.3 rappresenta un confronto tra il valore del Bitcoin e le varie features in un range di tempo limitato (circa 7 ore). Come già anticipato dall'heat map non c'è un evidente correlazione, inoltre i grafici dei predictors hanno oscillazioni ancora più pronunciate.

	Exchange rate	**Num. tweet**	**Sentiment**
count	18773	18773	18773
mean	9245.85	63.28	0.43
std	587704.83	554.63	0.09
min	7966.87	14	-0.78
max	10269.66	363	0.93

Tabella 4.1: Statistiche per i predictors, simple data



(a) Exchange rate e numero di tweet



(b) Exchange rate e sentiment

Figura 4.3: Confronto dei valori delle features con il trend del Bitcoin.

4.1.2 Dati a frequenza oraria

Con i dati del secondo topic invece si cercherà di cogliere variazioni in un tempo più elevato ma comunque breve, per poter reagire all'interno della giornata. I dati infatti sono distanziati di un ora gli uni dagli altri e come prima sono raggruppati in due giorni, la figura 4.4 ne mostra un estratto. In questo caso oltre ai semplici dati diretti come il numero di tweet nell'ora corrispondente e la media del sentiment nello stesso periodo, sono stati calcolati anche valori "complessi" che derivano de elaborazioni sul recente andamento del Bitcoin. Se coi dati del topic precedente avevamo tutti i record a disposizione possedendo la conoscenza totale dell'andamento, in questo caso abbiamo compensato quella mancanza di informazioni con l'aggiunta di variabili calcolate da quel gruppo di valori che sono andati persi. Con lo sperato vantaggio che variabili dipendenti da periodi di tempo maggiori siano meno soggetti a fluttuazioni, attutendo i vari rumori ad esempio con il meccanismo della media. Per avere a disposizione un informazione più precisa sull'andamento del Bitcoin, sono stati calcolati i valori che potrebbero essere usati per costruire un diagramma a candele, ovvero: open, close, min, max, al posto del semplice exchange rate che ora corrisponderebbe al close. A questi sono aggiunti alcuni indicatori che spesso vengono utilizzati dai traders per riconoscere in quale direzione si muove il mercato. Uno di questi indicatori è la media mobile ponderata (WMA), presa in considerazione l'ora come periodo, si calcola in modo simile a una media mobile ma con l'aggiunta di pesi. Questi pesi fanno si che valori più remoti abbiano un effetto minore rispetto a quelli più recenti, tentando di diminuire il ritardo seguendo maggiormente le tendenza a breve termine. Nella nostra formula i pesi scalano in maniera lineare per questo la media prende il nome di LWMA (linear weighted moving average), il calcolo è il seguente

$$LWMA_i = \frac{\sum_{i=0}^{N-1}(P_{-i+N} * (i+1))}{\frac{N*(N+1)}{2}}$$

dove P è il prezzo di close e N è il numero di periodi considerati. Più piccolo è N, meno rumori di mercato vengono filtrati dall'indicatore e più velocemente reagisce alle variazioni del Bitcoin.
Il secondo degli indicatori utilizzati è la media mobile esponenziale (EMA). L'idea è simile alla precedente, il peso aggiunto rispetto a una semplice media mobile da più importanza agli ultimi valori e riduce in modo

esponenziale i valori sempre più remoti.

$$EMA_i = EMA_{i-1} + \frac{2 * (P_i - EMA_{i-1})}{(N+1)}$$

L'ultimo indicatore usato basato sulla media è SMMA (smoothed moving average), i suoi valori sono leggermente diversi perché nel calcolo si utilizzano tutti i dati storici disponibili di quel periodo. Quindi come suggerisce il nome i valori sono livellati maggiormente ma dimostrano anche un maggiore ritardo rispetto al valore di mercato del Bitcoin. Segue la formula

$$SMMA_i = \frac{\sum_{i=0}^{N}(P_{N-i}) - SMMA_{i-1}}{N}$$

Il numero di periodi N preso in considerazione è 16, spesso questi indici basati sulla media vengono sfruttati effettuando dei paragoni con gli stessi ma calcolati su periodi diversi. In questo caso invece ci baseremo sulle differenze di calcolo dei vari indici considerando sempre gli stessi periodi, la figura 4.5 mostra come riescono ad essere sfruttati indici sugli stessi periodi per individuare trand positivi e negativi.
L'ultimo dei nostri indici sfruttato è l'RSI (Relative Strength Index), è efficace per rilevare gli eccessi di mercato. Viene calcolato con la formula

$$RSI = 100 - \frac{100}{1 + \frac{U}{D}}$$

A sua volta per calcolare U e D, bisogna considerare N periodi precedenti (assumeremo sempre 16) e per ognuno di questi bisogna capire se sono state sedute al rialzo, quindi differenza tra apertura e chiusura negativa, o al ribasso. U corrisponde alla media delle chiusure al rialzo, mentre D è la media delle chiusure al ribasso. In questo modo l'RSI può variare da un minimo di 0 se in presenza solo chiusure a ribasso e un massimo di 100 quando tutte le chiusure sono al rialzo. Un metodo veloce per ricevere informazioni è controllare se il suo valore scende sotto il 30 o sopra il 70. Nel primo caso ci troviamo in una situazione di iper-venduto, preludio di un inversione rialzista; nel secondo caso può annunciare una situazione di iper-comprato, preludio di un inversione ribassista.

Analogamente a quanto visto precedentemente analizziamo la correlazione tra le variabili tramite la heat map di figura 4.6. Vediamo un blocco

```
2020-08-12 09:03:02 1597222920 0.54184365 2853 9738.8545 541.3746 9748.857 9738.661 9721.2 9763.638 57.25347,
2020-08-12 09:05:02 1597223040 0.5301019 2929 9738.859 457.67615 9750.739 9749.014 9721.2 9763.638 63.646996,
2020-08-12 09:07:02 1597223160 0.5258336 2939 9739.006 383.485 9755.391 9755.109 9721.2 9763.638 59.79735,
2020-08-12 09:09:02 1597223280 0.5349533 2964 9738.954 304.18454 9736.1 9753.821 9721.2 9763.638 55.92302,
2020-08-12 09:11:02 1597223400 0.5268713 2972 9739.608 286.13727 9727.974 9755.749 9721.2 9763.638 49.828377,
```

Figura 4.4: Frammento di un singolo dato dal topic joinedDataHourly. I valori corrispondo a: data, timestamp, sentiment, numero tweet, media, varianza, open, close, low, high, RSI.

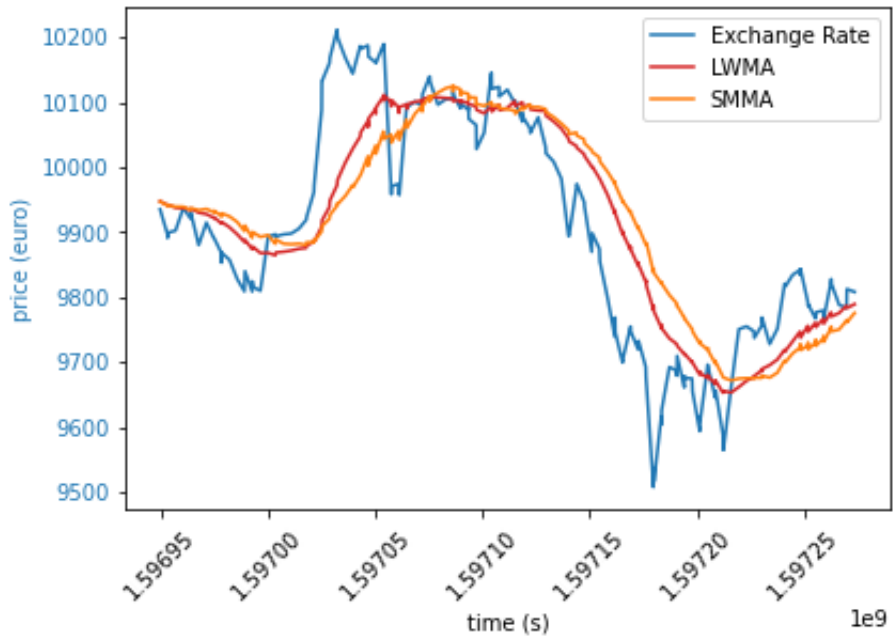


Figura 4.5: Confronto tra gli indici LWMA e SMMA che tra i punti di intersezione individuano gruppi di trend crescente e decrescente.

direttamente dipendente all'exchange rate, ma non ci stupisce perché sono i valori della candela open high, low e gli indici della media mobile, per questo hanno correlazione massima. Numero tweet, sentiment e le restanti variabili sono scorrelate, esattamente come capitava nel topic precedente. Per avere una visione più completa dei dati a nostra disposizione, la tabella 4.2 mostra alcune informazioni statistiche sui vari predictors, e la figura 4.7 rappresenta un confronto tra il valore del Bitcoin e le varie features in un range di tempo limitato (circa 7 ore). Il distanziamento dei valori di un ora fa si che la densità di dati sia nettamente inferiore al caso precedente.

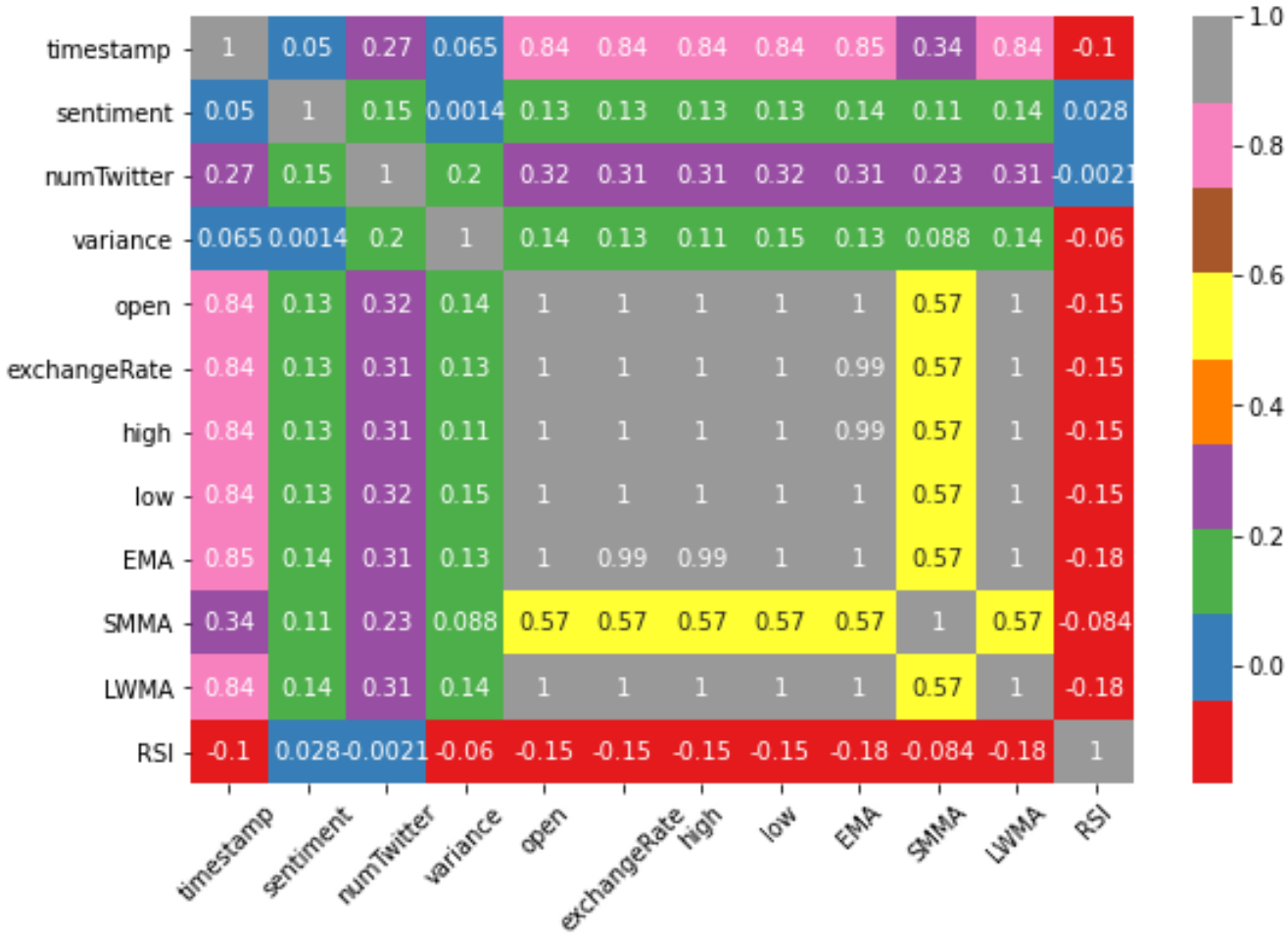


Figura 4.6: Matrice di correlazione tra le variabili del topic dataHourly.

	Close	**Tweet**	**Sent.**	**Var.**	**EMA**	**SMMA**	**LWMA**	**RSI**
count	1077	1077	1077	1077	1077	1077	1077	1077
mean	8954.85	1882.34	0.43	612.81	9362.81	9250.43	9366.76	51.01
std	418954	253331	0.04	2089832	565705	1617776	566118	124
min	7975.49	503	-0.50	2.73	7997.34	7987.45	7993.32	18.22
max	10159.31	4071	0.87	18491.89	10236.81	10363.52	10369.56	82.80

Tabella 4.2: Statistiche per i predictors, data hourly

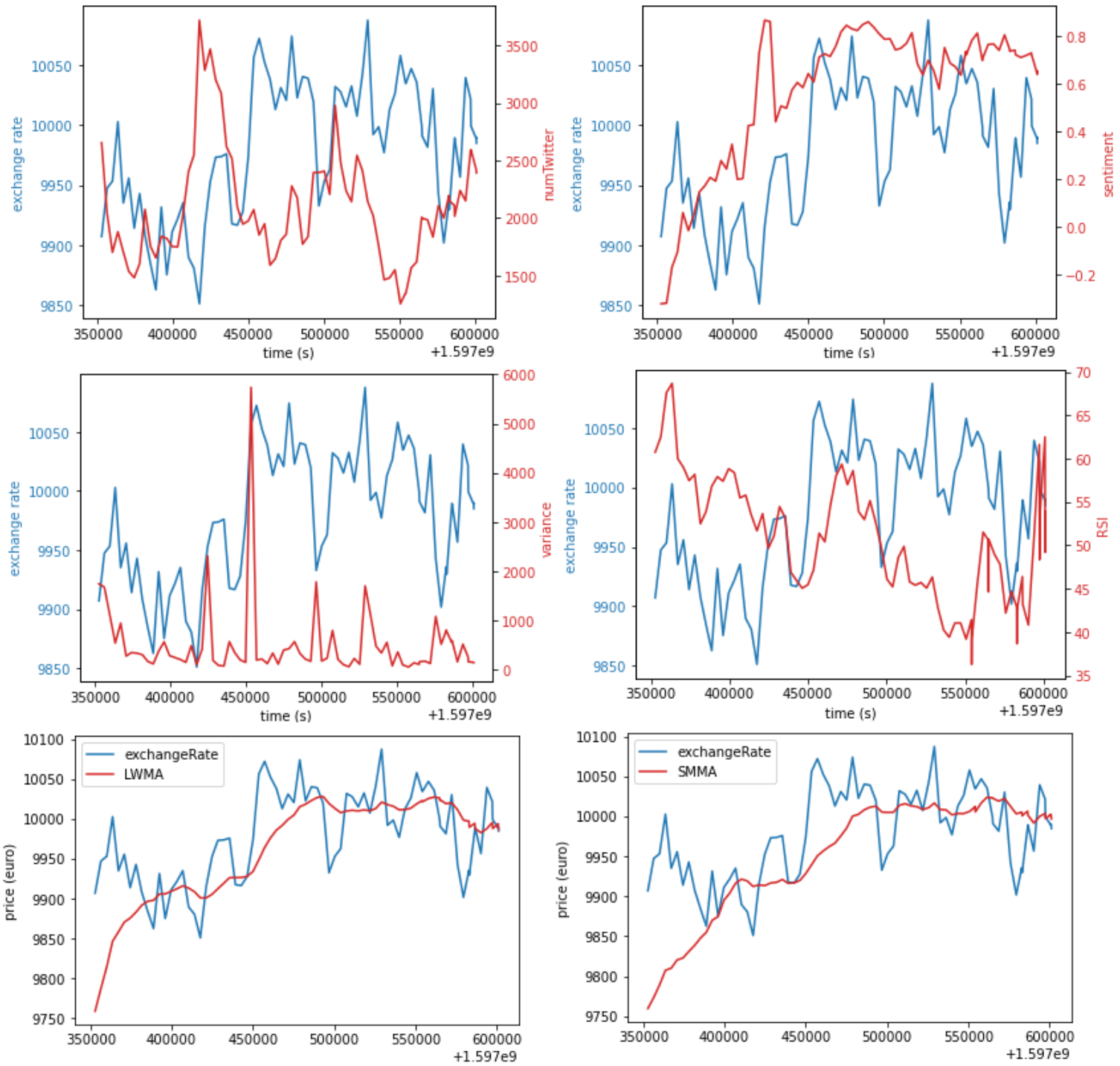


Figura 4.7: Confronto dei valori delle features con dati ogni ora.

Andiamo adesso ad affrontare il problema della realizzazione di un modello, ricordando che attualmente non esistono ancora metodi veramente efficaci per prevedere i prezzi delle azioni e così delle cripto-valute, anche con le più recenti tecnologie/algoritmi. Le due principali soluzioni di machine lerning prese in considerazione sono ARIMA e LSTM.

4.2 ARIMA

Un metodo statistico per la previsione delle serie temporali molto popolare e largamente utilizzato è il modello ARIMA. È una generalizzazione della più semplice media mobile auto-regressiva con l'aggiunta della nozione di integrazione. I metodi ARIMA sono basati sull'assunzione che i dati delle serie temporali possano essere riprodotti da un modello di probabilità, assumendo a priori che i valori futuri delle serie storiche siano correlati ai valori ed errori passati. Un prerequisito necessario per poter utilizzare questo tipo di modello è che la serie temporale analizzata deve essere stazionaria, ovvero deve avere media, varianza e funzione di autocorrelazione costanti. Nel caso questa proprietà non sia soddisfatta è possibile segmentare la serie in porzioni che possono essere utilizzate come serie stazionarie su cui applicare il modello ARIMA. Proviamo a capire meglio questo modello partendo dal significato dell'acronimo che gli da il nome, questo riesce a racchiuderne gli aspetti chiave:

- AR: AutoRegressive. Un modello che utilizza la relazione dipendente tra un'osservazione e un certo numero di osservazioni ritardate.
- I: Integrated. L'uso della differenziazione delle osservazioni grezze (ad esempio sottraendo un'osservazione da un'osservazione nella fase temporale precedente) al fine di rendere stazionarie le serie temporali.
- MA: Moving Average. Un modello che utilizza la dipendenza tra un'osservazione e un errore residuo a partire da un modello a media mobile applicato alle osservazioni ritardate.

Ognuno di questi componenti è esplicitamente specificato nel modello come parametro. Viene utilizzata la notazione standard di ARIMA (p, d, q) in cui i parametri vengono sostituiti con valori interi per indicare rapidamente lo specifico modello ARIMA utilizzato. I parametri del modello ARIMA sono definiti come segue:

- p: il numero di osservazioni di lag (osservate ad una quantità fissa di tempo) incluse nel modello, chiamato anche lag order.
- d: il numero di volte in cui le osservazioni grezze sono differenziate, chiamato anche grado di differenziazione.
- q: la dimensione della finestra della media mobile, detta anche ordine della media mobile.

Partiamo dalla costruzione di un modello di regressione lineare comprendente il numero e il tipo di termini specificati, i dati vengono preparati con un grado di differenziazione adeguato al renderli stazionari, vale a dire rimuovere trend e strutture stagionali che influenzino negativamente il modello di regressione. È possibile utilizzare un valore 0 per un parametro per indicare di non utilizzare quell'elemento del modello. In questo modo, il modello ARIMA può essere configurato per eseguire la funzione di un modello ARMA e persino un semplice modello AR, I o MA. L'adozione di un modello ARIMA per una serie storica presuppone che il processo sottostante che ha generato le osservazioni sia un processo ARIMA. Ciò può sembrare ovvio, ma aiuta a motivare la necessità di confermare le ipotesi del modello nelle osservazioni grezze e negli errori residui delle previsioni del modello.
La prima cosa da fare prima di iniziare la parametrizzazione del modello è assicurarci che la serie temporale sia stazionaria. Questo è un presupposto su cui si basano molti metodi statistici, che altrimenti non potrebbero essere applicati. Le serie temporali non stazionarie sono irregolari e imprevedibili mentre quelle stazionarie sono mean-reverting, ovvero fluttuano attorno a una media costante con varianza costante. Inoltre, la stazionarietà e l'indipendenza delle variabili casuali sono strettamente correlate perché molte teorie che valgono per le variabili casuali indipendenti valgono anche per le serie temporali stazionarie in cui l'indipendenza è una condizione richiesta. In parole povere, le serie temporali stazionarie non mostrano una tendenza a lungo termine ed hanno media e varianza costanti. Più specificamente, ci sono due definizioni di stazionarietà: stazionarietà debole e stazionarietà forte (anche detta stretta). Nel nostro caso per stazionarietà intenderemo stazionarietà debole, in cui le serie temporali soddisfano tre condizioni: media costante, varianza costante e funzione di autocovarianza dipendente solo da (t-s) (non t o s). D'altra parte, la stazionarietà forte implicherebbe che la distribuzione di probabilità delle

serie temporali non cambi nel tempo.
Per convertire serie non stazionarie in stazionarie, è possibile utilizzare il metodo di differenziazione, dove ogni valore corrente è sottratto al valore precedente originario (ad esempio $Y_t = Y_{t-1} + e_t \Rightarrow e_t = Y_t - Y_{t-1}$).
Guardiamo la figura 4.8:

- Il grafico in alto a sinistra è la serie temporale originale del prezzo del Bitcoin, che mostra una crescita sostanziosa.

- Il grafico in basso a sinistra mostra il residuo, calcolato sottraendo il valore del Bitcoin nell'istante precedente. Si può vedere che la serie non ha una grande dipendenza dal prezzo, ma nel primo periodo si possono notare delle oscillazioni meno ampie. In generale la la varianza delle serie aumenta all'aumentare del livello delle serie originali, e quindi non è stazionaria.

- L'angolo in alto a destra mostra il grafico del prezzo logaritmico del Bitcoin. La serie dovrebbe accentuare le oscillazioni per i valori più bassi e ammortizzarle in quelli alti, la differenza di prezzo da 8000 a 10000 non è così elevata da causare un effetto evidente a livello di grafico.

- L'angolo in basso a destra mostra il residuo del prezzo logaritmico del Bitcoin. L'effetto del logaritmo ha l'effetto di omogenizzare l'ampiezza delle oscillazioni, in questo caso però non si ha un miglioramento evidente.

Per il nostro modello useremo comunque il residuo del logaritmo, anche se ha un effetto minimo. Le differenze di ampiezze si verificano solo per valori molto diversi fra loro, quindi la soluzione è spezzare la nostra serie in frammenti che siano mean-reverting. Non c'è bisogno di creare molti frammenti, a parte nel primo periodo di monitoraggio le oscillazioni sono contenute in un range abbastanza ristretto.
Nell'adattare il modello ARIMA, la parsimonia è un concetto importante, nel senso che il modello dovrebbe avere il minor numero di parametri possibile ma essere comunque capace di spiegare le serie. Ad esempio p e q dovrebbero essere 2 o meno, oppure il numero totale di parametri dovrebbe essere inferiore a 3. Più alti sono i parametri, maggiore è il rumore che può essere introdotto nel modello che ne incrementa la deviazione standard.

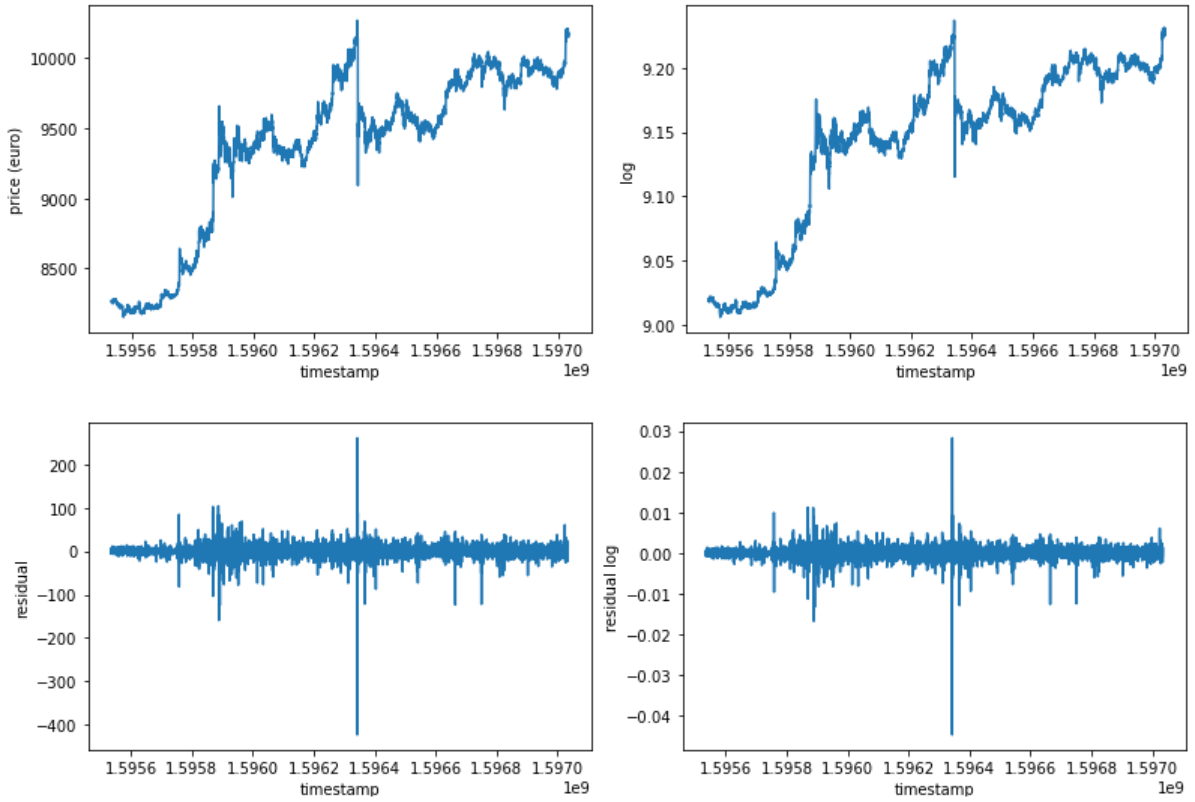


Figura 4.8: Sopra, l'andamento del Bitcoin e il logaritmo dello stesso. Sotto, il grafico dei residui della controparte superiore.

L'Akaike Information Critera (AIC) è una misura ampiamente usata di un modello statistico. In pratica quantifica 1) la bontà di adattamento e 2) la semplicità/parsimonia del modello in un'unica statistica. Quando si confrontano due modelli, quello con l'AIC inferiore è generalmente "migliore". Useremo questo criterio per scegliere i parametri migliori, non essendo esperti abbiamo adottato la soluzione più semplice: provare tutte le possibili combinazioni possibili per capire i migliori accoppiamenti, e una volta compresi impostare direttamente quei parametri nel modello, così da non sovraccaricarlo inutilmente.
L'obiettivo sarà di sfruttare il modello ARIMA per effettuare ad ogni istante 3 previsioni nel futuro, la prima a 2 minuti, la seconda a 1 ora e la terza a un giorno di distanza. Per ognuna di queste, abbiamo modellato i nostri dati in modo che fossero distanziati tra loro della stessa quantità di tempo in cui si intende prevedere. La figura 4.10 mostra un frammento

```
shiftIndex = [1, 1, 30, 720] #il valore predetto normalmente è il successivo (1), 30 indica
che predico 30 istanti dopo: 60min (intervallo ogni 2min)
def forecastWithBestModel(data, index, columnIndex):
    # ogni colonna è per una predizione diversa
    global table
    bestAic = np.inf
    bestOrder = None
    bestModel = None
    p_rng = q_rng = range(5)
    d_rng = range (2)
    for p in p_rng:
        for d in d_rng:
            for q in q_rng:
                try:
                    tmpModel = ARIMA(data, order=(p,d,q)).fit(method='mle', trend='nc')
                    tmpAic = tmpModel.aic
                    if tmpAic < bestAic:
                        bestAic = tmpAic
                        bestOrder = "(%d,%d,%d)" %(p, d, q)
                        bestModel = tmpModel
                except: continue
    if bestOrder == None:
        return table
    output = bestModel.forecast()[0]
    if columnIndex != 0: #residualCase
        output = np.exp(output)* table["exchangeValue"][index]
    # aggiorno tabella
    table.loc[index + shiftIndex[columnIndex], column[columnIndex][0]] = output
    table.loc[index + shiftIndex[columnIndex], column[columnIndex][1]] = bestOrder
    return table
```

Figura 4.9: Codice per applicare il modello ARIMA testando le varie combiazioni di parametri.

di soluzione ottenuta dal codice in figura 4.9, dove per ogni timestamp è fornita la previsione effettuata due minuti, un ora e un giorno prima, con accanto segnati i parametri del modello utilizzati. Notiamo che i modelli definitivi vedono, per le predizioni da 2 e 60 minuti, dei modelli ARMA in cui il grado di integrazione non è necessario (uguale a zero). Questo perché non è necessario considerare la differenza tra i dati, essendo questi già stazionari. Inoltre, escludendo il primo modello, l'ordine dell'auto-regressione e della media mobile varia tra 0 e 1, riducendo l'entità del rumore introdotto nel modello.

I risultati grafici sono mostrati in figura 4.11, mentre l'errore in figura 4.12. La previsione a due minuti genera valori molto vicini all'ultimo elemento della serie, sovrapponendosi al valore dell'exchange rate con un

```
2020-08-17 14:20:03,1597674000.0,10241.3388,10357.649039288286,"(2,0,4)",10024.063390354955,"(0,0,1)",10081.398101149965,"(0,1,1)"
2020-08-17 14:22:03,1597674120.0,10244.217756,10301.096100123368,"(2,0,4)",10056.034591495581,"(0,0,1)",10064.994223384861,"(0,1,1)"
2020-08-17 14:24:02,1597674240.0,10225.605558,10186.804502798217,"(2,0,4)",10100.384338205997,"(0,0,1)",10099.5640371793,"(0,1,1)"
2020-08-17 14:26:02,1597674360.0,10185.831075,10190.317340475714,"(3,0,4)",10099.99717946192,"(0,0,1)",10104.041296077463,"(0,1,1)"
2020-08-17 14:28:02,1597674480.0,10201.24258,10199.387430089835,"(2,0,4)",9998.27164565933,"(0,0,1)",10097.216410640758,"(0,1,1)"
2020-08-17 14:30:02,1597674600.0,10207.28814,10266.240468276103,"(2,0,4)",10033.27370982395,"(0,0,1)",10087.101561320802,"(0,1,1)"
2020-08-17 14:32:04,1597674720.0,10238.24006,10254.522878018324,"(2,0,4)",10050.168312155327,"(0,0,1)",10096.615461548734,"(0,1,1)"
```

Figura 4.10: In ogni riga è indicata la data, il timestamp, l'exchange rate, la previsione effettuata due minuti prima per quell'istante, i parametri del modello nell'effettuare la previsione precedente, il previsione fatta un ora prima con i relativi parametri e il valore predetto il giorno precedente con relativi parametri.

ritardo minimo. La previsione calcolata a un'ora di distanza riesce a produrre valori abbastanza realistici, i problemi si verificano nelle grandi variazioni. Nel grafico è mostrato un incremento repentino di 400€ nel trend reale, la previsione non è in grado di aspettarsi un tale comportamento basandosi semplicemente sullo storico passato, infatti riesce a correggere il suo valore solo dopo che il picco si è verificato. Lo stesso vale per i dati generati con un giorno di anticipo che in più aggiungono delle oscillazioni molto più marcate.

4.3 LSTM

A differenza del modello ARIMA precedentemente analizzato, ora cercheremo di tenere in considerazione, oltre al valore di exchange rate, anche tutte le caratteristiche che abbiamo analizzato nella sezione dati. In questo modo si elimina la pesante assunzione che tutti i predictors prima non considerati influenzino il prezzo sempre nella stessa maniera. Questo tipo di previsione prende il nome di multivariate time series. Un altro cambiamento che vogliamo apportare è la possibilità di lavorare meglio in previsioni a lungo termine sfruttando previsioni multi-step che possano predire un range temporale senza la creazione di più modelli separati. Le neural network sono in grado di soddisfare le nostre necessità.
Una delle migliori sfide del machine learning è far sì che il modello parli da solo. Non è importante solamente sviluppare una soluzione accurata, con un grande potere previsionale, ma anche sapere come il modello fornisce questi risultati: quali variabili sono maggiormente coinvolte, la presenza

di correlazioni, le possibili relazioni di causalità e così via. Con Neural Network questo tipo di informazioni sono off limits. Le reti neurali sono spesso viste come scatole nere, dalle quali è molto difficile estrarre informazioni utili per un altro scopo come le spiegazioni delle funzionalità. Si basano sull'uso di diversi layer, maggiore è il loro numero, più il modello diventa complesso e in grado di comprendere più a fondo ed essere più sensibile ai dati di ingresso. In ognuno di questi layer ci sono dei neuroni che possono essere connessi in vari modi e con varie funzionalità agli strati successivi, questi modificano i pesi dei vettori in input generando soluzioni che non possono essere prevedibili. L'apprendimento avviene in modo simile ad algoritmi supervised: usciti dall'ultimo strato il modello ha una soluzione che viene messa a confronto con ciò che ci si aspettava e se diversa avviene una fase di back-propagation, ovvero partendo dalla soluzione si ritorna a monte cercando portare a zero l'errore, applicando l'algoritmo di ottimizzazione del gradient descent. Più nello specifico per predire una time series andremo a realizzare un particolare tipo di recurrent nural network (RNN). In questo caso il flusso da input ad output non è feed foreward, in cui da ogni nodo si deve passare ad uno strato più profondo, ma è una rete con memoria: ogni uscita dipende sia dagli input che da valori generati da passaggi precedenti. Ogni valore intermedio può memorizzare informazioni su input passati per un tempo indeterminato, quindi è possibile un lavoro in parallelo che generi più output. La rete ha una sorta di memoria dovuta al collegamento che connette un nodo con un nodo dello strato precedente o dello stesso strato, ciò interrompe la semplice sequenza dall'input all'output, provocando l'attivazione di neuroni in diversi momenti e anche più volte nel tempo, essendo possibile la presenza di cicli. La back-propagation viene propagata nel tempo, oltre che attraverso la rete, questo amplifica maggiormente il problema del vanishing/exploding gradient, dove la rete nel tentativo di correggersi tenta di azzerare o far tendere all'infinito i vari pesi, invalidando la risposta del modello. Per questo si parla di long short term memory (LSTM), reti i cui nodi sono come delle celle di memoria in grado di mantenere lo stato per lungo tempo attraverso il CEC (constant error carousel) consentendo a LSTM di apprendere relazioni a lungo termine mitigando i rischi di test prolungati.
Per la realizzazione del modello si è scelto di utilizzare la libreria TensorFlow, sfruttando L'API di depp learning Keras.

4.3.1 Preparazione dei dati

Prima di passare alle creazione del modello vero e proprio, diamo uno sguardo all'allacciamento ai topic popolati da Flink. Il codice in figura 4.13 mostra i vari moduli che servono e l'inizializzazione delle variabili che saranno usate nel caso simple. Nella parte finale è realizzato il connettore a Kafka, con una politica "earliest" per accedere a tutto lo storico già cumulato dal framework. Dato che l'ordinamento non è garantito bisogna avere l'intero storico, la variabile "allDataStored" indica proprio se i dati già stoccati sono stati esauriti, e fintanto rimane falsa si aspetta di riceverli tutti. Lo sblocco avviene quando tra un dato e l'altro passano almeno 50 minuti, con questa condizione si assicura di star ricevendo in tempo reale e quindi con cadenza oraria, per il caso simple. Una volta ricevuti e trasformati in data frame si ordinano per timestamp. Da qui i dati ricevuti in tempo reale sono logicamente ordinati, nel caso Kafka si stoppasse e al riavvio venisse rifornito di tutti i dati del periodo in cui è rimasto inattivo, questi sarebbero disordinati, quindi ogni volta che un dato non mi arriva a distanza di un ora bisogna rifare l'ordinamento (figura 4.14). Nella stessa figura è anche mostrato il codice per assegnare la risposta y ad ogni dato di training, ovvero ciò che si vuole prevedere. Questa è l'unica parte che Flink non ha potuto svolgere, in quanto il vettore di 720 elementi (1 giorno) corrisponde a dati che non sono ancora arrivati, quindi si genererebbero dati ritardati ma noi vogliamo avere il valore più aggiornato per effettuare la previsione, perciò le y sono calcolate in un secondo momento. Nel calcolo si è fatto attenzione nel non avere buchi, e le informazioni incomplete sono state eliminate. Una volta posseduti un numero minimo di dati viene allenato il modello, quest'operazione è la più pesante computazionalmente e viene effettuata in pochi momenti molto distanziati (anche più di quanto riportato nel codice). Con questo modello ad ogni nuovo dato è possibile stimare l'andamento.

Per i dati a cadenza oraria l'impostazione non cambia, basta modificare le variabili iniziali come in figura 4.15. Qui riceviamo un dato ogni due minuti che contiene le informazione dei due giorni precedenti a cadenza oraria, quindi nei dati di training avremo come x 48 valori (uno ogni ora) e come y 24.

4.3.2 Il modello

Per prima cosa è necessaria una fase di normalizzazione che aiuterà il modello a raggiungere la convergenza e ne velocizzerà l'apprendimento. Anche i valori predetti sono normalizzati: il Bitcoin ha una grande varianza e tocca valori anche intorno ai 10000€ nel periodo monitorato, questo può causare valori anomali in caso di previsioni errate che con una normalizzazione sono mitigate. La normalizzazione non è calcolata considerando l'intero storico, ma considerando batch di due giorni. In questo modo ogni dato sarà normalizzato per conto suo e sarà così calibrato solamente sugli ultimi valori senza subire influenze dal passato. I dati a nostra disposizione coprono un mese e mezzo di osservazioni tra il luglio e l'agosto 2020, l'ultimo tratto di questo periodo è usato come test del modello ed è abbastanza adeguato a mostrare le debolezze nel predire una caduta del prezzo, mostrando un oscillazione che da 10000€ scende sotto i 9500€ nell'arco di poche ore. Partiamo da un modello semplice, un vanilla LSTM, composto da un singolo hidden layer, usiamo tanh come funzione di attivazione di questo layer e come strato finale ci deve essere un fully connected layer con un numero di unità pari al numero di elementi di output, quindi 720. Tra le funzioni di ottimizzazione scegliamo adam che ha una curva di apprendimento più lenta, e permette di raggiungere la convergenza con più sicurezza. Il risultato dopo 20 epoche è mostrato in figura 4.16, si notano delle predizioni abbastanza realistiche per il modello, che non sempre riescono ad intuire il corretto trand del Bitcoin. Per valutare l'accuratezza del modello non possiamo usare i metodi tradizionali che ci danno una percentuale di quanto è preciso perché non siamo in un problema di classificazione. Per le serie temporali possiamo usare l'RMSE (Root Mean Squared Error)

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \hat{x}_i)^2}{n}}$$

ci indica con una misura della stessa grandezza del valore interessato, di quanto la previsione si scosta dal valore reale. Nella figura 4.17 sono mostrati due grafici raffiguranti l'RMSE. Quello a destra indica l'errore per ogni singola predizione, ricordiamo che nel caso "simple" una predizione corrisponde a 720 valori, a cui è stato calcolato l'RMSE, i valori oscillano intorno ai 50/70 quando il modello si comporta bene, ma in corrispondenza di grandi oscillazioni la discrepanza tra predizione e realtà si fa sentire.

Nel grafico a sinistra invece è indicato l'errore tra tutte le predizioni ad ogni range di previsione, quindi il primo valore indica quanto il modello è preciso a predire nei due minuti successivi, il secondo nei 4 minuti seguenti e così via. Da questo secondo grafico si nota come, più si prevede lontano nel tempo, maggiore è l'oscillazione a cui varia l'errore; questo è l'effetto nell'usare tanh come funzione di attivazione, mentre con un relu l'errore tende ad aumentare progressivamente senza incrementare significativamente le oscillazioni.
Solamente per arrivare a questo risultato abbiamo dovuto testare diversi modelli e configurazioni, alcune delle scelte effettuate riguardano la normalizzazione. La figura 4.18 mostra due predizioni ottenute precedentemente. Inizialmente per effettuare la standardizzazione ci basavamo sull'intero database, calcolando per ogni variabile la media e la varianza, ma nell'intera serie molte caratteristiche sono molto variabili, coprendo dei range di valori abbastanza ampi. Per questo motivo penso che il modello, nel momento in cui deve predire basandosi solamente sui due giorni precedenti, non riesca ad essere preciso nell'individuare dei valori realistici e oscilli fortemente influenzato da una normalizzazione globale. Il grafico di destra invece mostra una previsione, standardizzando la variabile del sentiment come fatto con le altre, e quindi non binarizzandola. Questo ha causato un aumento dell'imprecisione, incrementando le oscillazioni per tutte le previsioni.
Proviamo ora ad aggiungere un ulteriore layer al nostro modello, oltre ai 50 neuroni che compongono il primo strato LSTM aggiungiamo un fully connected layer, anch'esso composto da 50 neuroni oltre all'obbligatorio strato sempre fully connected che dia in uscita i 720 finali. Per i primi due layer abbiamo usato la relu come funzione di attivazione e adam come algoritmo di ottimizzazione. In più essendo un modello più complesso con molti più parametri e più facile cadere in overfitting, per questo all'uscita di ogni layer abbiamo sfoltito il numero di connessioni tra neuroni con l'aggiunta di un dropout: 20% di sfoltimento dopo l'LSTM layer e 50% dopo il fully connected. Alcune previsioni effettuate usando questo modello sono mostrate in figura 4.19, mentre gli errori nella figura 4.20. Notiamo come la precisione complessiva sia simile al modello precedente, presentando sempre un grande picco di errore in presenza della rapida caduta del prezzo. Nonostante la precisione simile notiamo che però tutte le previsioni hanno la stessa forma, oscillando semplicemente tra valori

diversi. Questo comportamento nasce dalla presenza del doppio strato, che riesce ad apprendere maggiormente dai dati di training e ci fa pensare che, piuttosto di capire in base ai dati precedenti con quale possibile trand si muoveranno, ha individuato una forma definita nei 720 valori di previsione, da traslare per tutte le possibili situazioni, che si comporta meglio di qualsiasi altro tipo di soluzione.

Passiamo ad analizzare il caso dataHourly, qui abbiamo molti più predictors, quindi potrebbe essere necessario un modello più complicato per poter comprendere meglio i dati. D'altro canto i valori hanno le informazioni di un ora al posto dei 2 minuti, per questo sono più armoniosi, e il modello potrebbe apprendere più facilmente. Come prima proviamo ad inserire un unico layer LSTM con 50 neuroni, usando la relu come funzione di attivazione e adam come algoritmo di ottimizzazione. In 20 epoche abbiamo il risultato di fig 4.21, sono state effettuate oltre 3000 previsioni in uno degli ultimi range temporali che non sono stati utilizzati per il training, esattamente come per il caso "simple". L'accuratezza cambia da previsione a previsione, la figura 4.22 mostra prima l'accuratezza de ogni singola predizione, a sinistra è calcolata con la media del MSE in ogni istante, e a destra l'accuratezza ad ogni step. Quest'ultima ci dice quanto il modello è preciso a fare il primo step di previsione, il secondo, e così via. Ci sono infatti 24 valori, che seppur con qualche oscillazione, diminuiscono di precisione man mano che si prevede nel futuro. Come accaduto nel caso precedente anche qui abbiamo un picco di errore nel periodo in cui si ha la rapida caduta all'interno dei due giorni di storico dei dati di test, in altri tratti riesce ad anticipare ribassi o rialzi così come li prevede dove non ci sono. Nel complesso riesce a predire meglio in alcuni tratti ma nei casi in cui sbaglia fa degli errori molto più marcati.
Aggiungiamo un secondo livello di complessità al nostro modello, dopo un primo layer LSTM aggiungiamo un fully connected layer con dropout 50% sul FC e 20% sull'LSTM. Abbiamo anche aggiunto qualche neurone in più e acambiato la funzione di attivazione da relu a tanh. La figura 4.23 e 4.24 mostrano i risultati delle nostre modifiche. L'accuratezza complessiva è più elevata, nei primi dati di test abbiamo un peggioramento nella previsione, ma i picchi di errore massimi sono attenuati nonostante ancora presenti e la precisione negli ultimi dati è rimasta simile.

Possiamo concludere da quanto osservato che la struttura dati "simple"

che tiene in considerazioni tutti i record a cadenza di 2 minuti, senza raggruppamenti orari e senza considerare variabili derivate aggiuntive volte ad individuare pattern con più sicurezza, abbiamo un RMSE minore e quindi una previsione più vicina ai valori reali.

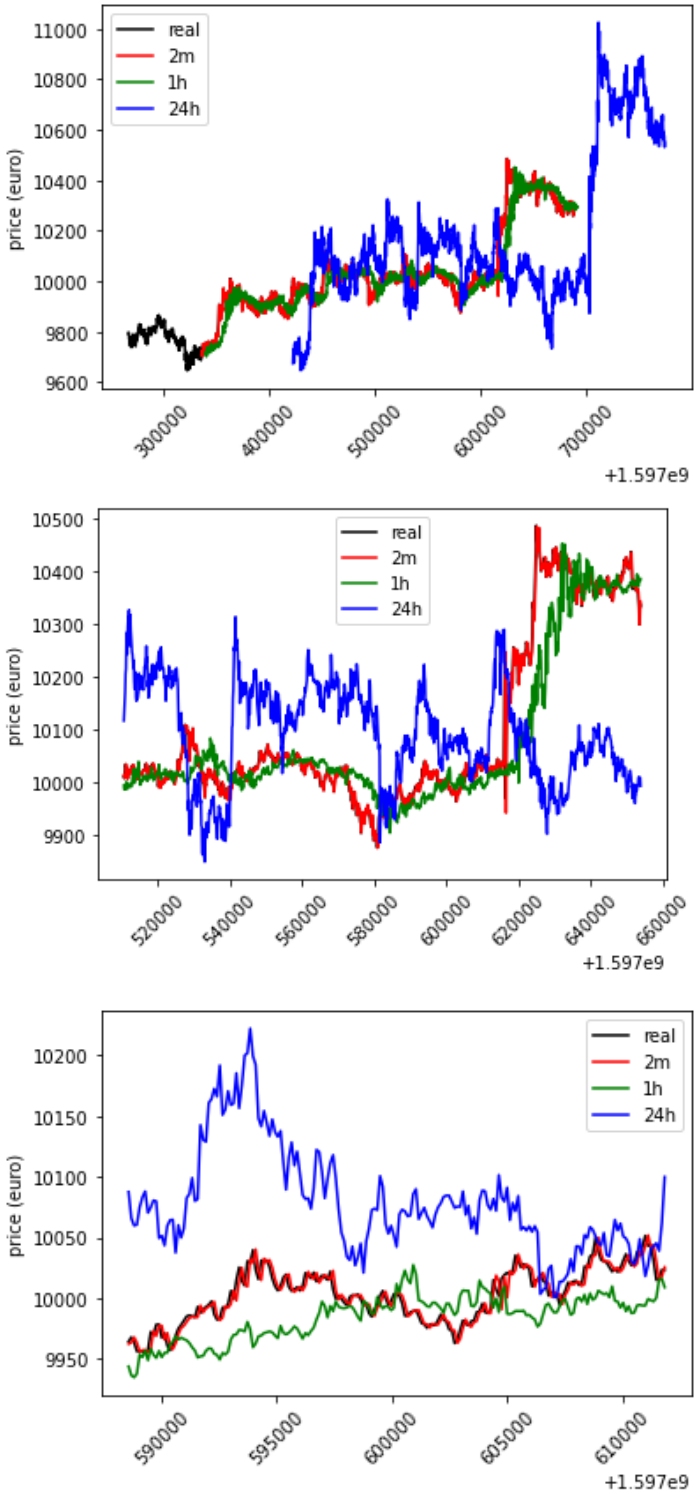


Figura 4.11: Risultato grafico per la predizione usando il modello ARIMA per diverse sezioni temporali.

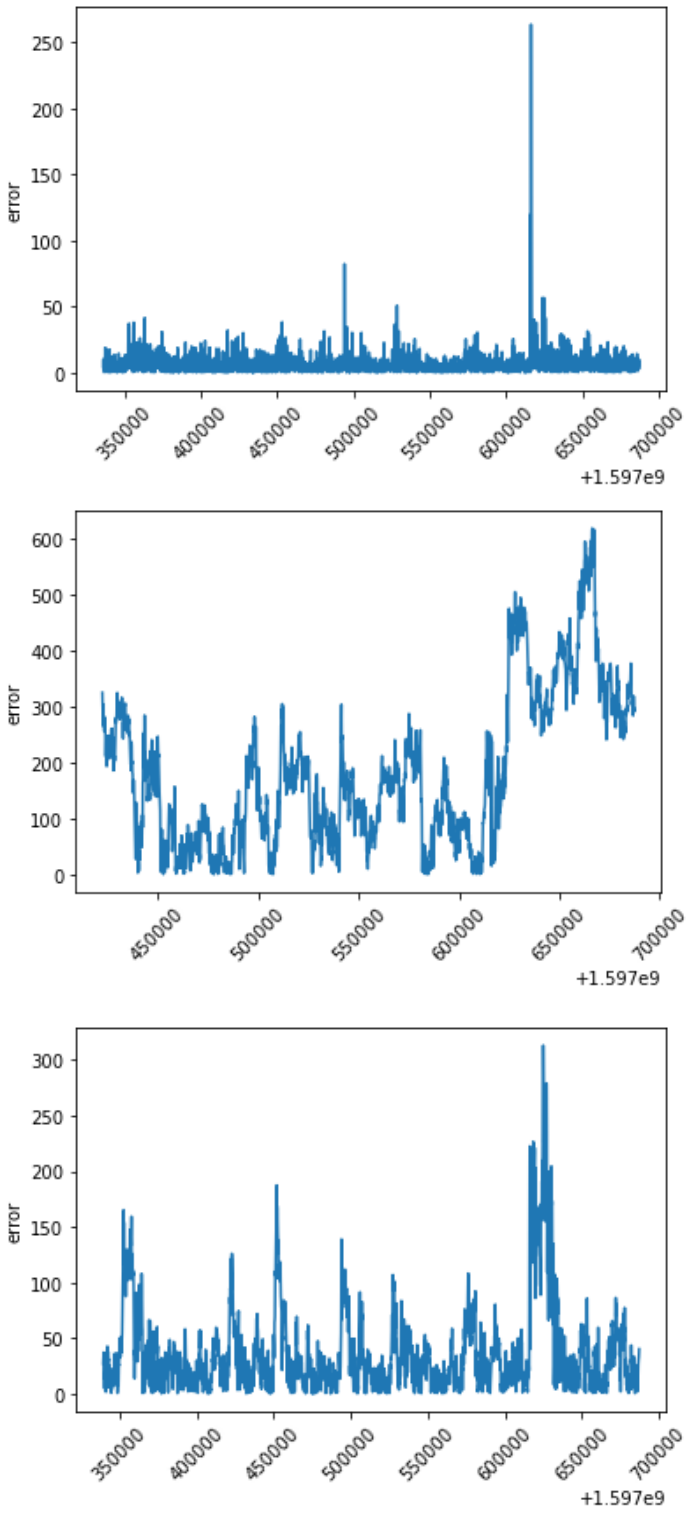


Figura 4.12: Accuratezza misurata tramite RMSE in ordine dall'alto verso il basso, della predizione a 2 minuti, a 1 ora e 1 giorno.

```
import numpy as np
import pandas as pd
from copy import deepcopy
from kafka import KafkaConsumer
from sklearn.metrics import mean_squared_error
from matplotlib import pyplot as plt
from keras.initializers import glorot_normal as GlorotNormal
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import LSTM
from keras.layers import RepeatVector
from keras.layers import TimeDistributed
from keras.layers import Concatenate
from keras.layers import Dropout
import time

# record inviati da flink in un unico messaggio
n_input = 1440 #2min ciasuno -> 48h
# record futuri che si vuole predire
n_out = 720 #24h
# numero di messaggi ricevuti nel periodo di tempo che voglio predire
oneDayStored = 24 #n_out/juttingElements
# numero di record di cui è shiftato un messaggio dall'altro (1h = 30record)
juttingElements = 30

featuresToSelect = ["timestamp", "sentiment", "tweets", "exchangeRate"]
trainY, trainX = list(), list()
predictions = list()
indexResponse = featuresToSelect.index("exchangeRate")
topicName = 'joinedDataSimple'
i = 0
yStart = 0
startTime = time.perf_counter()
notSorted = True
firstBuilding = True
allPreviusDataStored = False

consumer = KafkaConsumer(
     topicName,
     bootstrap_servers=['localhost:9092'],
     auto_offset_reset='earliest', #Lastest
     enable_auto_commit=True,
     group_id='pythonjoinedDataSimple-group')
```

Figura 4.13: Codice per applicare un modello LSTM (prima parte).

```
# ricevo un dato ogni ora
for messageH in consumer:
    endTime = time.perf_counter()
    # se ho atteso almeno 50min (3000s) vuol dire che sto captando i dati in tempo reale
    allPreviusDataStored = (True if endTime-startTime > 3000 else False)
    startTime = time.perf_counter()

    # decodifico il messaggio
    messageH = messageH.value.decode("utf-8")
    # formatto il messaggio in una struttura facile da usare
    twoDaysBlock = storeData(messageH)
    # se i dati sono incompleti ritorna un oggetto vuoto
    if twoDaysBlock.empty == True:
        continue
    trainX.append(twoDaysBlock[featuresToSelect].values)

    if allPreviusDataStored:
        if notSorted:
            # ordino per timestamp
            trainX.sort(key= lambda x: x[0][0])
            notSorted = False

        # calcolo le y per il training
        # per ognuna devo avere a disposizione i dati dell'intero giorno subito successivo
        if len(trainX) > oneDayStored:
            # un solo giro a meno di dati non real time
            yDim = yStart # yDim indica l'indice dell'ultimo elemento di Ytrain
            for elementToAssingY in range(i+1-oneDayStored-yDim):
                # la y corrisponde ai primi n_out valori presi subito dopo oneDayStored
                currentResponse = trainX[yStart+oneDayStored][n_out:,indexResponse]
                # se i timestamp differiscono più di un giorno vuol dire che ci sino dati
                #mancanti, quindi non vengono usati per il train
                if trainX[yStart+oneDayStored][-1,featuresToSelect.index("timestamp")] <
                        trainX[yStart][-1,featuresToSelect.index("timestamp")]+86600:
                    del trainX[yStart]
                    i -= 1
                    continue
                trainY.append(currentResponse)
                yStart += 1

        # effettuo la previsione
        # ho cumulato almeno 2d di dati di yTrain (48)
        if i >= oneDayStored + oneDayStored*2:
            # riaggiorno il modello ogni 6h
            if  i%(oneDayStored/4) == 0 or firstBuilding:
                model, meanVar = buildModel(deepcopy(
                    trainX[:-oneDayStored]), deepcopy(trainY))
                firstBuilding = False
            prediction = forecast(model, deepcopy(trainX), meanVar)
            predictions.append(prediction)
    i+=1
```

Figura 4.14: Codice per applicare un modello LSTM (seconda parte).

```
featuresToSelect = [ "timestamp", "sentiment", "tweets", "movingAverage", "variance",
                     "open", "close", "low", "high", "EMA", "SMMA", "LWMA", "RSI"]
indexResponse = featuresToSelect.index("close")
topicName = 'joinedDataHourly'
# record inviati da flink in un unico messaggio
n_input = 48 #1h ciasuno -> 48h
# record futuri che si vuole predire
n_out = 24 #24h
# numero di messaggi ricevuti nel periodo di tempo che voglio predire
oneDayStored = 720
# numero di record di cui è shiftato un messaggio dall'altro (1h = 30record)
juttingElements = 1
```

Figura 4.15: Codice per applicare un modello LSTM, caso dataHourly.

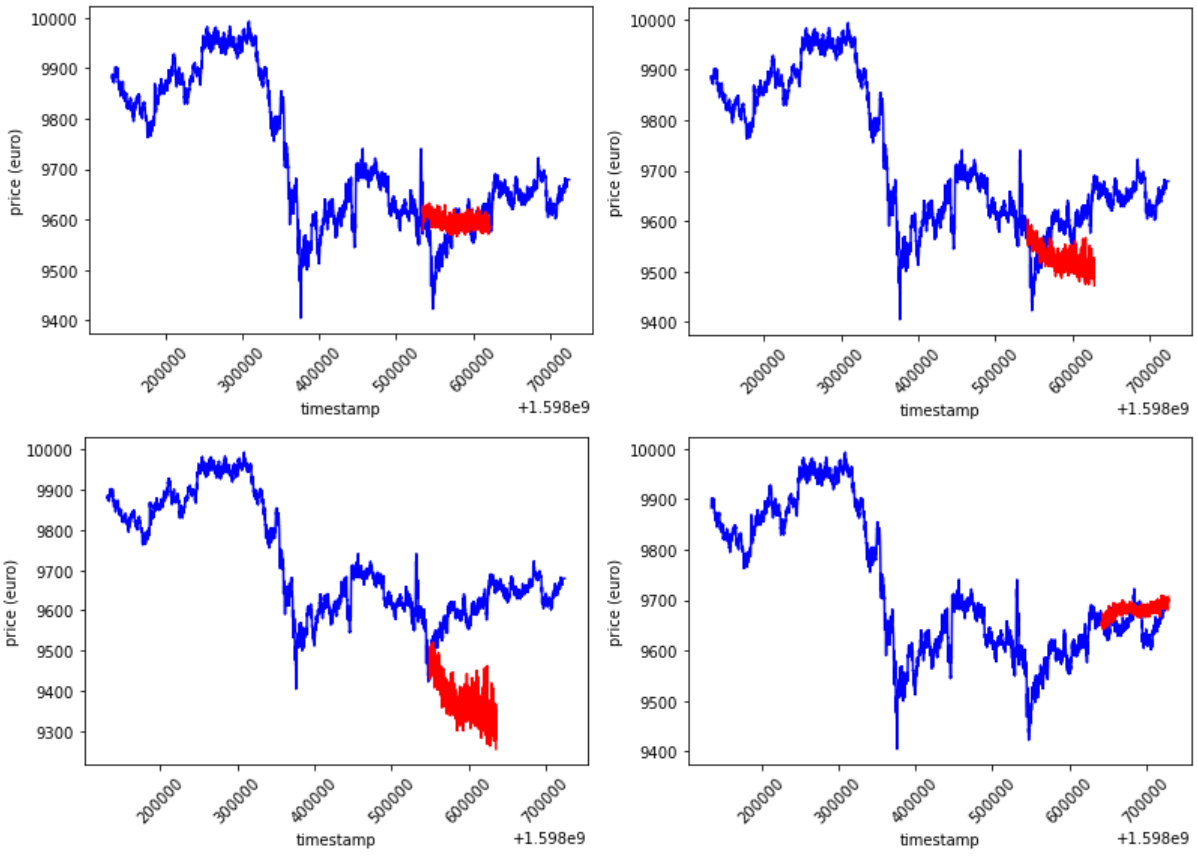


Figura 4.16: Predizione con modello Vanilla, caso simple.

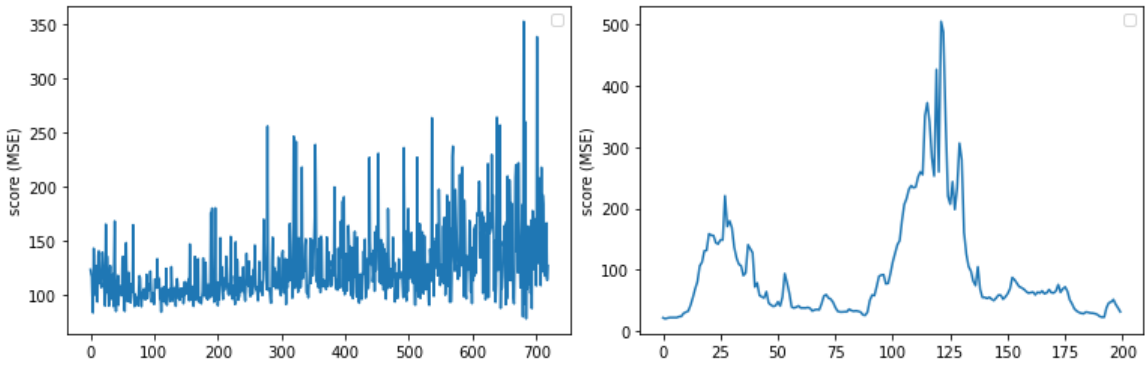


Figura 4.17: Accuratezza modello Vanilla. A sinistra l'RMSE per profondità di previsione, a destra il valore dell'RMSE step by step. Caso simple.

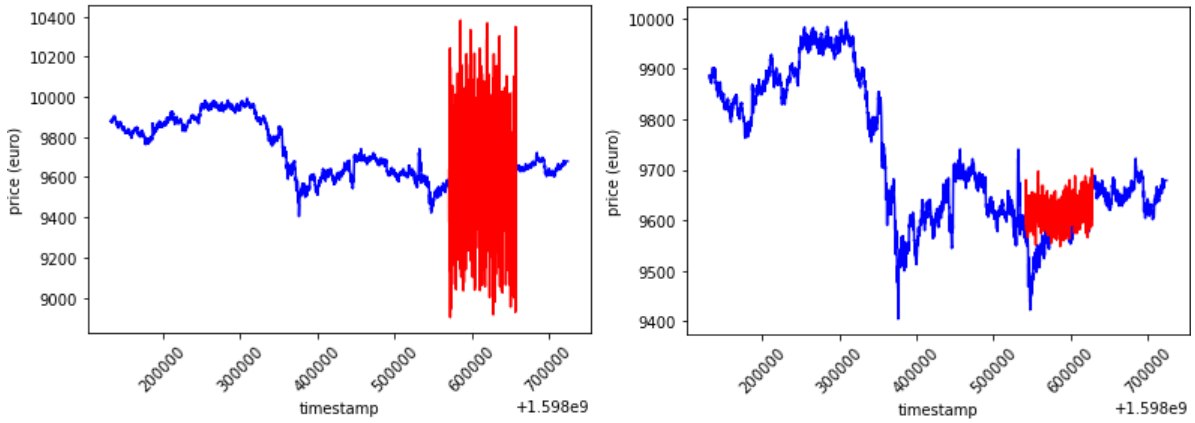


Figura 4.18: A sinistra una predizione con modello vanilla standardizzando considerando la totalità dei dati. A destra una predizione sempre con modello vanilla standardizzando il sentiment come le altre variabili.

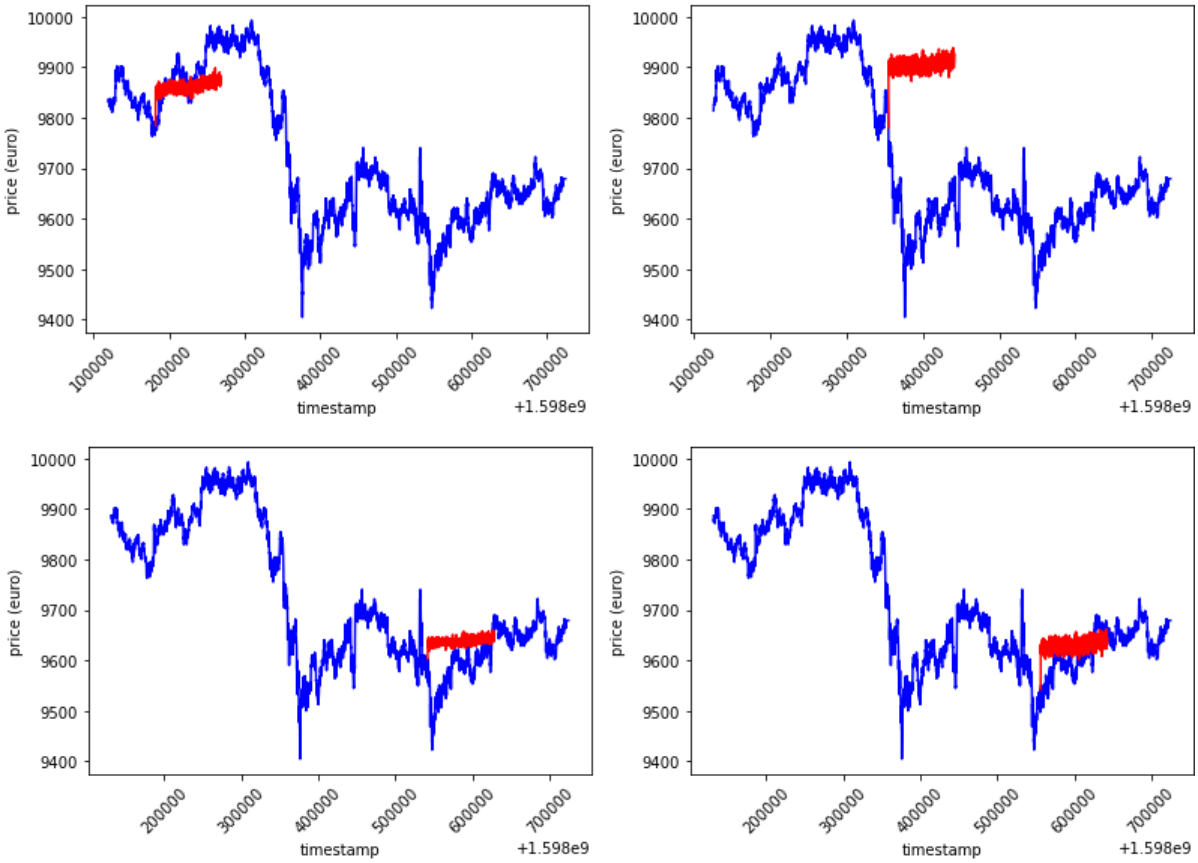


Figura 4.19: Alcune predizioni usando un modello a più strati: LSTM(50) + dropout(0.2) + fully connected(50) + dropout(0.5) + fully connected(720).

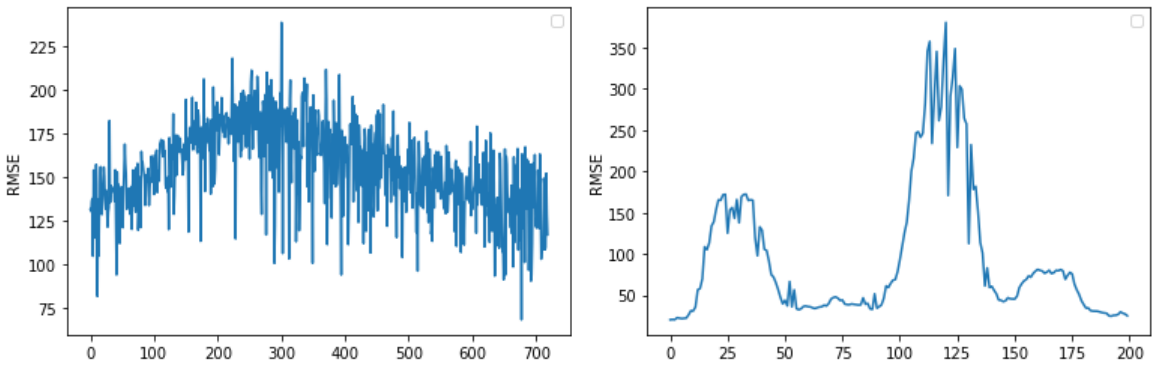


Figura 4.20: Accuratezza modello a più strati. A sinistra l'RMSE per profondità di previsione, a destra il valore dell'RMSE step by step. Caso simple.

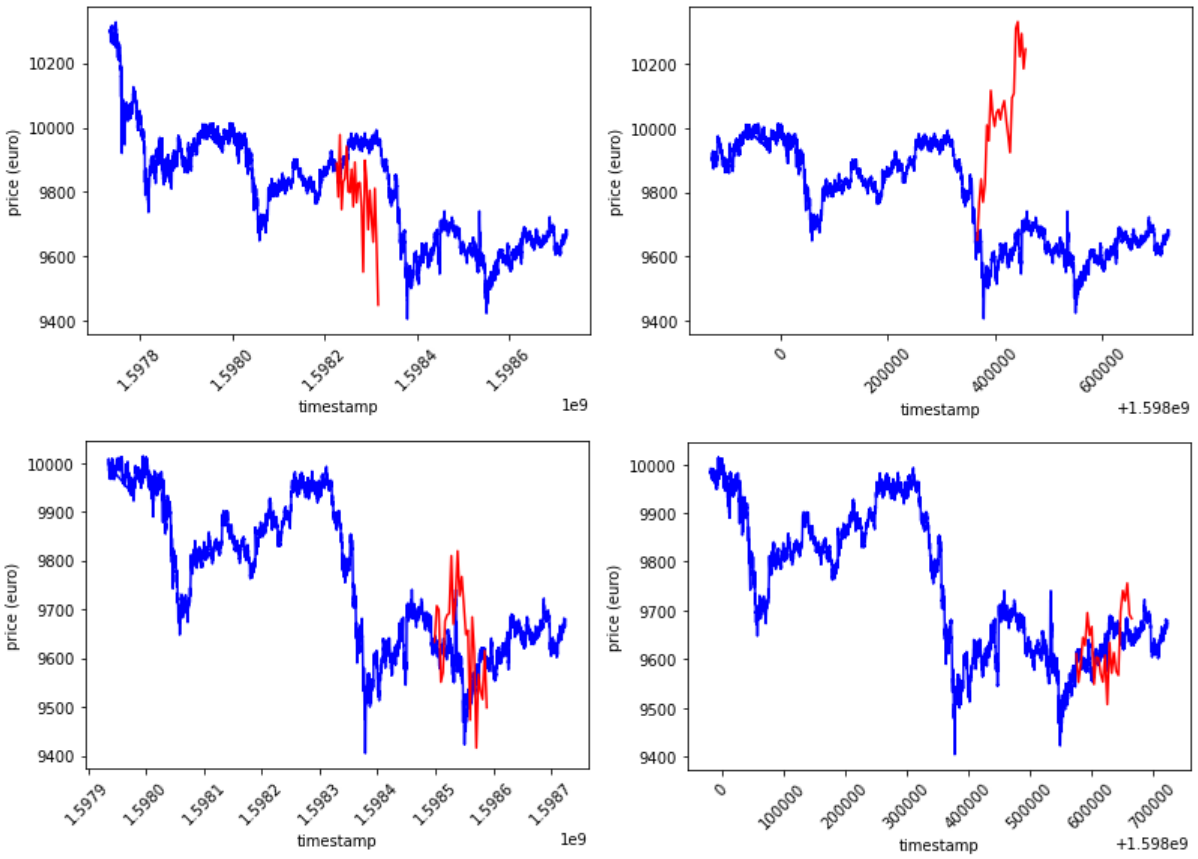


Figura 4.21: Predizione con modello Vanilla, caso data hourly.

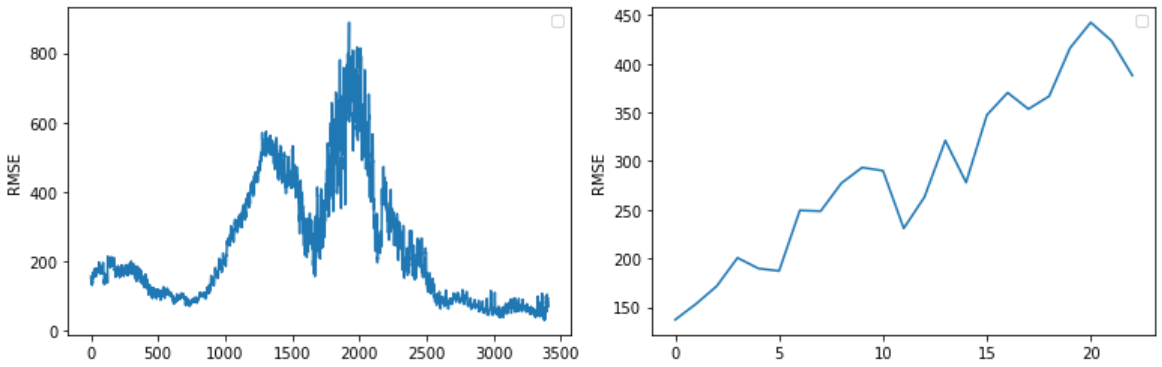


Figura 4.22: Accuratezza modello Vanilla. A sinistra il valore dell'RMSE step by step, a destra l'RMSE per profondità di previsione. Caso data hourly.

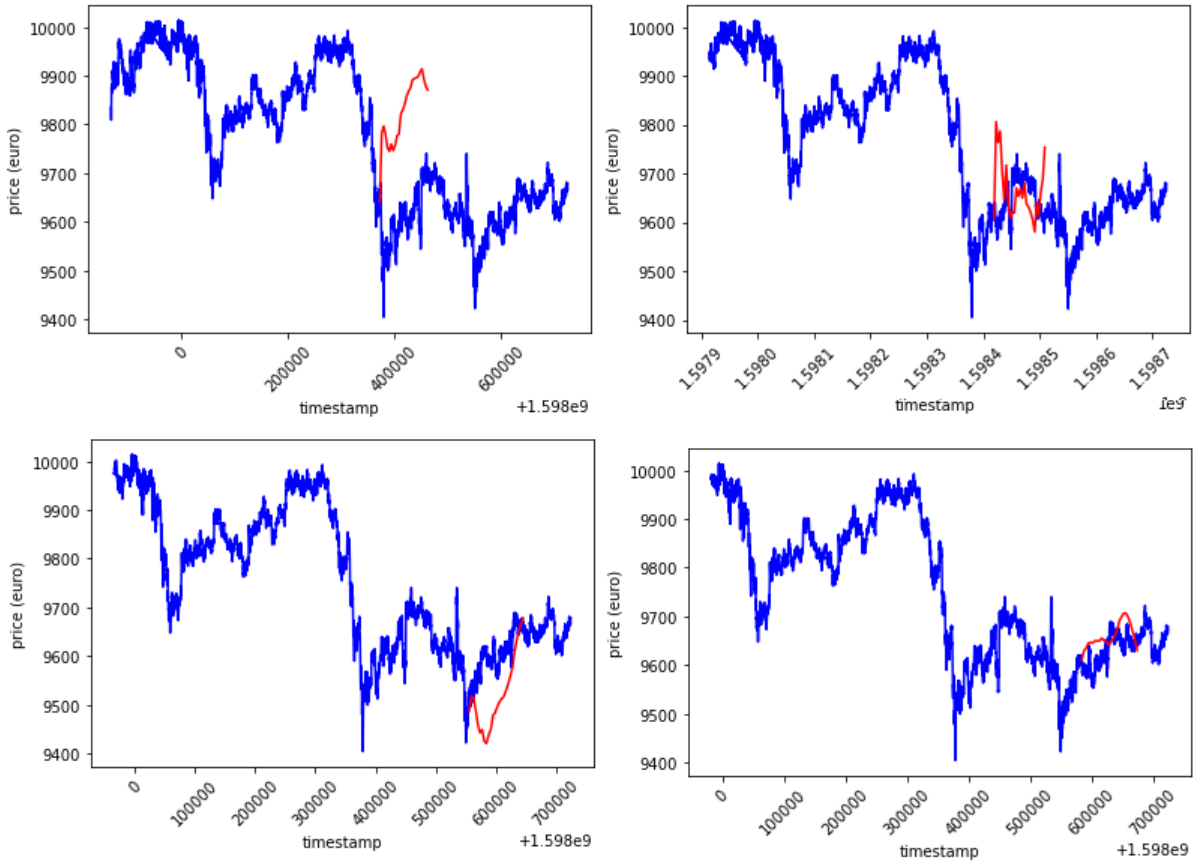


Figura 4.23: Predizione con modello a più strati: LSTM(100) + dropout(0.2) + fully connected(50) + dropout(0.5) + fully connected(24). Caso data hourly.

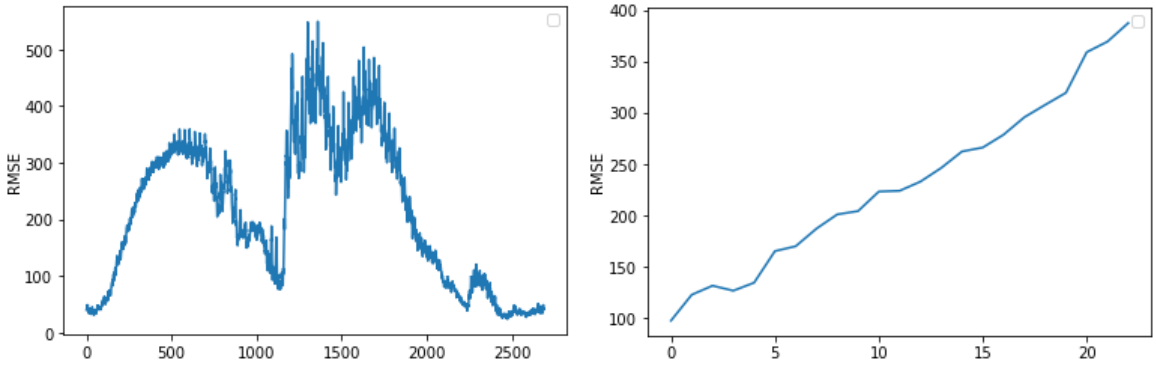


Figura 4.24: Accuratezza modello a più strati. A sinistra il valore dell'RMSE step by step, a destra l'RMSE per profondità di previsione. Caso data hourly.

Capitolo 5

Conclusioni

Le applicazioni di continuous intelligence sono infinite, e il loro utilizzo sta continuando a riempire spazio nel mercato di tutte le aziende moderne. L'analisi in streaming è la naturale svolta delle applicazioni Big Data che hanno una disponibilità di dati continui nel tempo, e l'automazione anche solo parziale delle strategie di business, permette soluzioni tempestive e affidabili al verificarsi di determinate condizioni, ottimizzando i processi di produzione ed incrementando i profitti ottenuti.
Nella nostra applicazione ci siamo limitati a catturare informazioni in streaming riguardanti il Bitcoin, considerando direttamente il valore di exchange, o attraverso indici largamente utilizzati dai traders, ed anche da informazioni ricavate da Twitter, come la quantità di persone che discutono dell'argomento e il loro sentimento riguardo lo stesso. Potenzialmente le fonti su cui plasmare i nostri dati possono essere davvero vaste, ma bisogna anche saper riconoscere i dati ininfluenti. Il sentiment ad esempio si è rivelato essere molto meno utile di quanto ci si aspettasse: il numero di tweet sul Bitcoin è sorprendentemente alto, considerando che dall'API di Twitter si riesce ad attingere solo ad una percentuale del totale, 70 tweet di media ogni due minuti è un buon dato. Ognuno di questi però aveva sentimenti molto variabili e generalmente positivi, che, effettuando una media, non erano in grado di fornire un informazione abbastanza forte su cui poter basare una previsione. Parlando sempre di sorgenti dei dati, in questa tesi è stato deciso, partendo da zero, di acquisire informazioni in streaming. In generale però si possono considerare anche gli storici, dati stazionari presi da database da affiancare a informazioni real-time per effettuare analisi più consistenti.

Per la realizzazione dell'architettura sono stati effettuati alcuni paragoni con altri framework per gestire un'acquisizione, un'elaborazione e un'archiviazione real-time. Nel caso trattato molte soluzione potevano essere valide, dovendo gestire pochi flussi di dato in parallelo con cadenza nell'ordine dei minuti. Il nostro unico server infatti è stato in grado di gestirli senza troppi problemi. In ogni caso la soluzione proposta, adottata per acquisire familiarità con i framework più utilizzati, può essere tranquillamente scalata e distribuita su più server, garantendo affidabilità, bassa latenza e tolleranza ai guasti. Un elemento mancante che avrebbe fornito del valore aggiunto è un framework di visualizzazione dati in tempo reale, dai grafici dei capitoli precedenti si nota che i risultati sono stati estrapolati e visualizzati in modo statico con la libreria Matplotlib di Python.
Il confronto avvenuto tra il metodo classico ARIMA per la predizione di serie temporali in streaming e il modello moderno LSTM ha rivelato alcuni aspetti che ignoravamo. ARIMA non richiede quasi nessuna pre-elaborazione, l'importante è lavorare con dati stazionari, altrimenti in modello non è in grado di effettuare una predizione. Oltre questo hanno costo computazionale veramente esiguo e il tempo di predizione si adatta perfettamente per analisi in tempo reale. I risultati ottenuti sono persino superiori ai modelli più moderni qui rappresentati dal LSTM, persino la previsione ad un giorno di distanza in questo confronto ha avuto risultati rispettabili. Passando ai metodi moderni di deep learning, la normalizzazione è una procedura fondamentale in quanto migliora notevolmente i tempi di addestramento e risolve problemi di convergenza. Oltre a questo una corretta scelta dei parametri e il loro tuning risulta un operazione molto lunga, valutando i risultati di una grid search tra numero di neuroni da utilizzare, funzione di attivazione, funzione di ottimizzazione, numero di epoche, scelta dei layer e dell'aggiunta del dropout. Il costo computazionale è nettamente superiore, ma solo in fase di allenamento: per ogni epoca nel caso che abbiamo chiamato "simple" impiega intorno ai 30 secondi, mentre per il casso "data hourly" ne bastano 6. Fortunatamente si possono salvare i modelli appresi per poi allenare solamente i nuovi dati di training, permettendo così l'applicazione in real-time. In conclusione ci si aspetterebbe che LSTM, che è uno dei layer più avanzati di neural network, sia molto più accurato di altri metodi statistici come ARIMA. Invece è in grado di adattarsi molto bene ai dati di training cadendo facilmente in overfitting, ma è molto meno efficace nell'effettuare una previsione.

Bibliografia

[1] Statista, *Volume of data/information created worldwide from 2010 to 2024*. Disponibilità: https://www.statista.com/statistics/871513/worldwide-data-created/

[2] Steve Swoyer, *Beer and Diapers: The Impossible Correlation*, TDWI. Disponibilità: https://tdwi.org/articles/2016/11/15/beer-and-diapers-impossible-correlation.aspx

[3] Franz H. Messerli, M.D., *Chocolate Consumption, Cognitive Function, and Nobel Laureates*, The new england journal of medicine. Disponibilità: http://www.biostat.jhsph.edu/courses/bio621/misc/Chocolate%20consumption%20cognitive%20function%20and%20nobel%20laurates%20(NEJM).pdf

[4] ABIresearch, *54 Technology Trends To Watch In 2020*. Disponibilità: https://go.abiresearch.com/lp-54-technology-trends-to-watch-in-2020?utm_source=media&utm_medium=email

[5] Douglas Laney. *BControlling Data Volume, Velocity and Variety*. Disponibilità: https://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf

[6] Andrea De Mauro, Marco Greco, Michele Grimaldi, (2016) *A formal definition of Big Data based on its essential features*, Library Review, Vol. 65 Issue: 3, pp.122-135

[7] Jaime G. Fitzgerald, Gniewko Lubecki, Cynthia Jaggi, *The Data to Dollars (TM) Value Chain : A Practical Guide to Business Analytics*

[8] Cavanillas J., Curry E., Wahlster W. (eds) New Horizons for a Data-Driven Economy. *Big Data Usage*, Becker T. (2016). Disponibilità: https://link.springer.com/chapter/10.1007/978-3-319-21569-3_8

[9] James Serra. *What is the Lambda Architecture*, Big Data and Data Warehousing. Disponibilità: https://www.jamesserra.com/archive/2016/08/what-is-the-lambda-architecture/hadoop_summit_2015_takeaway_the_lambda_architecture-picture_1/

[10] Soumaya Ounacer, Mohamed Amine Talhaoui, Soufiane Ardchir, Abderrahmane Daif, Mohamed Azouazi, *A New Architecture for Real Time Data Stream Processing*. International Journal of Advanced Computer Science and Applications, Vol. 8, No. 11, 2017. Disponibilità: https://www.researchgate.net/publication/321494828_A_New_Architecture_for_Real_Time_Data_Stream_Processing

[11] Babak Yadranjiaghdam, Seyedfaraz Yasrobi, Nasseh Tabrizi, *Developing a Real-Time Data Analytics Framework for Twitter Streaming Data*. 2017 IEEE 6th International Congress on Big Data. Disponibilità: https://ieeexplore.ieee.org/document/8029342

[12] Paula Ta-Shma, Adnan Akbar, Guy Gerson-Golan, Guy Hadash, Francois Carrez, and Klaus Moessner, *An Ingestion and Analytics Architecture for IoT Applied to Smart City Use Cases*, IEEE Internet of Things Journal Vol:5 2018. Disponibilità: https://ieeexplore.ieee.org/abstract/document/7964673

[13] Subhashini Chellappan, Dharanitharan Ganesan, *Spark Real-Time Use Case*, Practical Apache Spark pp 261-273. Disponibilità: https://link.springer.com/chapter/10.1007/978-1-4842-3652-9_10

[14] Pat Research *Top 18 data ingestion tools*. Disponibilità: https://www.predictiveanalyticstoday.com/data-ingestion-tools/

[15] NSA Releases First in Series of Software Products to Open Source Community. Disponibilità: www.nsa.gov

[16] Apache NiFi Overview. Disponibilità: https://nifi.apache.org/docs/nifi-docs/html/overview.html

[17] CLOUDERA DOCS. Multi-Tenant Authorization. Disponibilità: https://docs.cloudera.com/HDPDocuments/HDF3/HDF-3.0.1.1/bk_administration/content/multi-tenant-authorization.html

[18] Eran Levy, *Kafka vs. RabbitMQ: Architecture, Performance Use Cases*. Disponibilità: https://www.upsolver.com/blog/kafka-versus-rabbitmq-architecture-performance-use-case

[19] Neha Narkhede, Gwen Shapira, Todd Palino, *Kafka: The Definitive Guide*. Disponibilità: https://www.oreilly.com/library/view/

kafka-the-definitive/9781491936153/ch04.html

[20] Farhad Mehdipour, Hamid Noori, Bahman Javadi, *Advances in Computers*, Chapter Two - Energy-Efficient Big Data Analytics in Datacenters. Disponibilità: https://www.sciencedirect.com/topics/computer-science/big-data-processing

[21] Farhad Mehdipour, Hamid Noori, Bahman Javadi, *Spark Streaming*, Chapter Two - Energy-Efficient Big Data Analytics in Datacenters. Disponibilità: https://databricks.com/glossary/what-is-spark-streaming

[22] Alfonso Puttini, *Cassandra vs MongoDB*. Disponibilità: https://www.alfonsoputtini.it/programmazione/nosql/cassandra-vs-mongodb/

[23] Nishant Garg, *Apache Kafka*, Packt Publishing, 2013.

[24] Alpha Vantage. Disponibilità: https://www.alphavantage.co/

[25] Wikipedia, *Web scraping* Disponibilità: https://it.wikipedia.org/wiki/Web_scraping

[26] Alpha Vantage Support, *API Key* Disponibilità: https://www.alphavantage.co/support/#api-key

[27] Twitter, Docs. Disponibilità: https://developer.twitter.com/en/docs

[28] InternetLiveStats, *Twitter Usage Statistics*. Disponibilità: https://www.internetlivestats.com/twitter-statistics/

[29] Georg Dorffner, *Neural Networks for Time Series Processing*. Dept. of Medical Cybernetics and Artifcial Intelligence, University of Vienna and Austrian Research Institute for Artifcial Intel ligence. Disponibilità: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.45.5697

[30] John Cristian, Borges Gamboa, *Deep Learning for Time-Series Analysis*. University of Kaiserslautern Kaiserslautern, Germany. Disponibilità: https://arxiv.org/abs/1701.01887

[31] Twitter developer, *Potential adjustments to Streaming API sample volumes*. Disponibilità: https://twittercommunity.com/t/potential-adjustments-to-streaming-api-sample-volumes/31628

[32] Massimo Amato, Luca Fantacci, *Per un pugno di bitcoin: Rischi e opportunità delle monete virtuali*.

[33] cjhutto, *VADER-Sentiment-Analysis*. Disponibilità: https://github.com/cjhutto/vaderSentiment

[34] Keiichi Goshima, Hirohi Takahashi, Takao Terano, *Estimating financial words' negative-positive from stock prices*. Disponibilità: https://editorialexpress.com/cgi-bin/conference/download.cgi?db_name=CEF2015&paper_id=477

[35] Chung-Chi Chen, Hen-Hsen Huang, Hsin-Hsi Chen, *NTUSD-Fin: A Market Sentiment Dictionary for Financial Social Media Data Applications*. Disponibilità: https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/windows.html

[36] Apache NiFi, github. Disponibilità: https://github.com/apache/nifi/tree/master/nifi-nar-bundles/nifi-standard-bundle/nifi-standard-processors/src/main/java/org/apache/nifi/processors/standard

[37] twitter4j. Disponibilità: http://twitter4j.org/en/index.html

[38] Hortonworks *MiNiFi Quick Start*. Disponibilità: https://docs.cloudera.com/HDPDocuments/HDF3/HDF-3.4.0/minifi-quick-start/hdf-minifi-quick-start.pdf

[39] Apache NiFi Team *Minifi Java Agent Quick Start*. Disponibilità: https://nifi.apache.org/minifi/minifi-java-agent-quick-start.html

[40] Apache Flink Documentation. Disponibilità: https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/windows.html