UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA

Corso di Laurea Triennale in Informatica

Rilevamento di smart contract Ponzi sulla blockchain di Ethereum tramite machine learning

Relatori:

Dott. Letterio Galletta

Prof. Pierpaolo Degano

Dott. Fabio Pinelli

Candidato:

Andrea Giusti

ANNO ACCADEMICO 2020/2021

Abstract

L'elaborato riguarda lo studio e il rilevamento tramite machine learning di particolari frodi, gli schemi Ponzi, implementati in smart contract presenti sulla blockchain di Ethereum: gli smart contract Ponzi o più semplicemente gli smart Ponzi.

Per riuscire ad individuare questi tipi di smart contract è stato costruito un insieme di dati uniforme a partire da tre dataset iniziali provenienti da tre articoli differenti ([2], [3] e [4]) con cui allenare e valutare le prestazioni di un classificatore binario che discrimina tra i contratti Ponzi e non Ponzi. Il risultato ottenuto con questo modello è stato infine confrontato coi punteggi ottenuti da quelli proposti in [3] risultando migliore nell'identificazione di questi tipi di frodi.

I programmi sviluppati e il dataset sono disponibili all'indirizzo: https://github.com/Lychett/tesi-smart-ponzi.

Indice:

1. Introduzione

La tecnologia della blockchain è descritta come una tecnologia dirompente e rivoluzionaria. Di fatto è una lista append-only, in continua crescita, che memorizza transazioni e che viene mantenuta e aggiornata da una rete di nodi P2P attraverso un meccanismo di consenso distribuito.
Nel tempo sono state create diverse piattaforme che mirano a supportare applicazioni basate su blockchain. Ethereum è la piattaforma più utilizzata al momento. Questa permette la pubblicazione e l'esecuzione decentralizzata di applicazioni chiamate contratti intelligenti, o smart contract. Questi programmi generalmente gestiscono risorse economiche e sono implementati in un linguaggio di programmazione Turing-completo. Gli smart contract rilasciati sulla piattaforma Ethereum non possono essere manomessi e vengono eseguiti in modo automatico quando ricevono delle transazioni e quando si verificano delle condizioni prestabilite.
Data la connotazione prettamente economica, la mancanza di regolamentazioni efficaci e un certo livello di anonimato garantito agli utenti, la blockchain è stata anche utilizzata come strumento per commettere delle truffe.
Una di queste truffe sono gli schemi Ponzi. In questi schemi si richiede ai partecipanti di investire del denaro, che viene usato per pagare le quote dei precedenti finanziatori, garantendo un profitto per i primi utenti che entrano nello schema a scapito degli ultimi arrivati.
Ad oggi, molti schemi Ponzi sono implementati sotto forma di smart contract e rilasciati sulla piattaforma di Ethereum. Questi contratti sono chiamati smart contract Ponzi o più semplicemente smart Ponzi.

L'obiettivo perseguito in questa tesi è la creazione, utilizzando tecniche di machine learning, di un classificatore che sia in grado di discriminare tra smart Ponzi e smart contract non Ponzi. Per fare questo inizialmente abbiamo costruito un dataset contenente circa 900 contratti Ponzi e 4700 contratti non Ponzi con cui poter allenare e valutare un classificatore binario. Il dataset contiene principalmente account features, cioè informazioni derivate dalle transazioni ricevute o effettuate dal contratto.
Dopo la costruzione del dataset abbiamo implementato un classificatore binario in Python. L'implementazione del classificatore ha seguito un processo iterativo di addestramento e valutazione, con l'obiettivo di massimizzarne l'accuratezza della classificazione. Il classificatore così costruito è stato confrontato con altri simili proposti in [3], mostrando che riesce ad ottenere risultati migliori.

1.1 Sommario

Il contenuto di questo documento è organizzato come segue: il capitolo 2 introduce il lettore agli schemi Ponzi, partendo dal loro funzionamento e ripercorrendo la loro storia, dall'invenzione, all'inizio del secolo scorso, fino all'approdo sul web. Nella prima parte del capitolo 3 si effettua una introduzione alla piattaforma Ethereum, fornendo una spiegazione generale delle sue funzionalità e delle sue caratteristiche, descrivendo le componenti principali; gli account, le transazioni, gli smart contract e il linguaggio ad alto livello con cui scrivere il codice dei contratti: Solidity. Nella seconda parte del terzo capitolo si introducono gli smart contract Ponzi. Qui si descrivono i requisiti che portano un contratto a essere considerato uno schema Ponzi, la tassonomia e i metodi di ridistribuzione dell'ether (o ETH - la criptovaluta di Ethereum). Il capitolo 4 contiene invece l'analisi dei contributi forniti dai vari articoli studiati ([1], [2], [3] e [4]), spiegando i passi compiuti per la costruzione del dataset, la scelta delle features e lo script Python implementato per recuperare le informazioni sui contratti dal web. Il capitolo 5 contiene la descrizione degli algoritmi di classificazione scelti (Decision Tree, Random Forest e XGBoost) e degli iperparametri, la costruzione del modello, l'analisi dei risultati ottenuti nei vari test svolti e l'analisi statistica svolta per mostrare che il modello da noi costruito risulta migliore rispetto a quello proposto in [3]. Nell'ultimo capitolo si riassumono i risultati della tesi presentando una breve panoramica sui possibili sviluppi futuri.

2. Gli schemi Ponzi

In questo capitolo si introducono gli schemi Ponzi, il loro funzionamento, la storia e l'evoluzione, partendo dall'invenzione di questa truffa fino all'approdo sul web.

2.1 Cos'è uno schema Ponzi

Uno **schema Ponzi** può essere definito come un modello economico o aziendale pianificato dettagliatamente, alla cui base non c'è alcuna tipologia di prodotto o servizio capace di garantire profitto e valore. L'unico obiettivo dello schema Ponzi è il guadagno che otterrà l'ideatore.
Gli schemi Ponzi sono spesso mascherati da programmi di investimento "ad alto rendimento", dove i potenziali investitori sono attratti dalla promessa di ingenti guadagni e dalla facile accessibilità**.** Gli utenti entrano nello schema investendo del denaro, le condizioni effettive che permettono di guadagnare dipendono dalle regole specifiche dello schema. Tutti gli schemi Ponzi, però, hanno in comune il fatto che, per riscattare il proprio investimento, bisogna far entrare nuovi utenti nello schema.

La definizione di schema Ponzi data dall'U.S. Securities and Exchange Commission (SEC) [9] è la seguente:
''A Ponzi scheme is an investment fraud that involves the payment of purported returns to existing investors from funds contributed by new investors. Ponzi scheme organizers often solicit new investors by promising to invest funds in opportunities claimed to generate high returns with little or no risk. With little or no legitimate earnings, Ponzi schemes require a constant flow of money from new investors to continue. Ponzi schemes inevitably collapse, most often when it becomes difficult to recruit new investors or when a large number of investors ask for their funds to be returned.''

2.2 Come funziona uno schema Ponzi

Uno schema Ponzi lo si può immaginare come una struttura piramidale, come mostrato in fig. 2.1. All'apice risiede il creatore dello schema e al livello L+1 gli utenti che compenseranno gli investimenti di coloro che sono al livello L. I partecipanti situati ai livelli più alti della piramide ottengono un profitto mentre quelli ai livelli più bassi perderanno i loro investimenti.

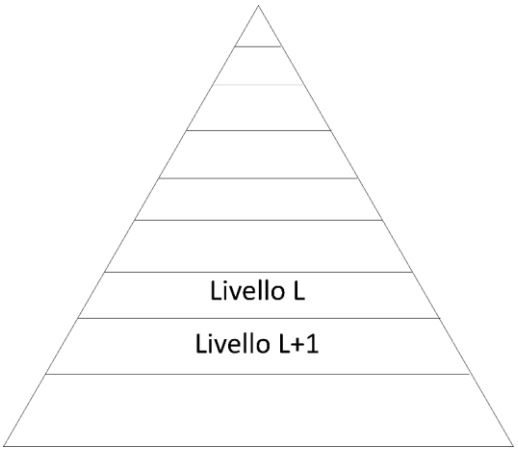


Fig. 2.1. La classica forma piramidale assunta dagli schemi Ponzi. Gli investitori presenti ai livelli più alti guadagnano a spese dei finanziatori posizionati nei livelli sottostanti.

Per fare in modo che questo modello economico funzioni, ogni persona viene invogliata a coinvolgerne altre. Il successo di questa truffa è dato dal fatto che quasi sempre i reclutatori parlano di questa opportunità ad amici e parenti. La fiducia e la conoscenza della persona che ne parla facilita la propagazione del business. Lo schema per reclutare membri è sempre lo stesso, promessa di grandi guadagni col minimo sforzo, investimento accessibile e talvolta offerta a tempo limitato.

2.3 Un po' di storia

2.3.1 Le origini

Lo schema Ponzi è una truffa ideata negli Stati Uniti, nei primi del 900, dall'italiano Carlo Ponzi. A quel tempo era comune che le lettere per l'estero includessero un "buono" per l'acquisto di un francobollo per la risposta: i buoni avevano un costo diverso in ciascun Paese, ma il loro controvalore in francobolli era lo stesso dappertutto. Con i tassi di cambio e postali fluttuanti, Ponzi capisce che era questione di tempo prima che aumentasse il valore dei coupon per francobolli. Così, attraverso una rete di contatti in Italia, inizia a rastrellare quelli che gli emigranti inviano ai loro parenti, per rivenderli sul mercato americano.

Forte dei primi guadagni ottenuti con questa forma di arbitraggio, Ponzi apre una società e incoraggia amici e colleghi a scommettere sul suo schema, promettendo tassi di rendimento sugli investimenti del 50% in 3 mesi. La voce comincia a diffondersi e nel giro di due anni la rete Ponzi ha dipendenti e clienti in tutto il paese, al punto che al suo apice, nel 1920, ha accumulato un patrimonio di centinaia di migliaia di dollari (una fortuna, per l'epoca).

Il suo business, però, non è solido. Anzi, non è nemmeno un business: Ponzi infatti paga i rendimenti promessi con il denaro dei nuovi investitori, che affluisce copioso nelle sue casse, e non con i guadagni. È il primo, perfetto schema piramidale.

A scoprire la frode è Clarence Barron, editore del Wall Street Journal. Barron calcola che Ponzi dovrebbe vendere 160 milioni di coupon per raccogliere i soldi di cui ha bisogno per garantire i guadagni promessi, ma dal momento che ci sono solo 27.000 coupon postali in circolazione nel mondo, ciò non è possibile.

All'inizio nessuno crede a Barron e Ponzi continua a fare affari, finché un'ispezione negli uffici della società mette a nudo la verità: Ponzi non possiede i buoni postali millantati. Viene così accusato di frode e di altri 85 reati, e condannato ad alcuni anni di carcere. Scontata la pena, torna prima in Italia, dove cerca di replicare il suo schema senza successo, poi si trasferisce a Rio, dove morirà nel 1949.

2.3.2 L'eredità

Nel 2008, a un secolo dalla sua invenzione, lo schema Ponzi guadagna di nuovo le prime pagine dei giornali, quando l'FBI arresta il finanziere Bernie Madoff con l'accusa di aver "volatilizzato" 65 miliardi di dollari. Madoff, come Ponzi, prometteva rendimenti stratosferici (il 10% fisso), che onorava però con i capitali dei nuovi arrivati, in mancanza dunque di rendimenti veri.
Come si può immaginare, il meccanismo funziona bene all'inizio, o fintanto che esiste un flusso continuo e sempre più numeroso di *investitori*, ma prima o poi deve interrompersi e ciò causa la perdita di denaro della maggior parte dei finanziatori.

2.3.3 Cripto-Ponzi

La truffa di Madoff ha rappresentato una sorta di versione 2.0 dello schema Ponzi: ma oggi, nel vasto universo delle criptovalute, potrebbe nascondersi la versione 3.0.
Oltre a **OneCoin**, l'etichetta di schema Ponzi ha infatti colpito anche la piattaforma **Bitconnect**, che prometteva a chi lasciava in consegna le sue criptovalute, un ingente ritorno per molti mesi. Era pure previsto un incentivo in denaro per chi portava nuovi affiliati, e questa è una caratteristica tipica (anche se non esclusiva) di ogni schema Ponzi. Bitconnect ha chiuso il 17 gennaio 2018, dopo due ingiunzioni delle autorità degli Stati Uniti.

3. Schemi Ponzi “smart” su blockchain

In questo capitolo si introducono Ethereum, gli smart contract e Solidity, il linguaggio di programmazione con cui sono implementati gli smart contract. Successivamente si descrivono le caratteristiche che deve possedere un contratto per essere considerato uno schema Ponzi e infine si espone la tassonomia degli schemi Ponzi e la ridistribuzione del denaro nei vari tipi di schemi.

3.1 Introduzione a Ethereum

Lo sviluppo di Ethereum è iniziato nel 2013 per mano del programmatore canadese Vitalik Buterin. Ethereum è una piattaforma decentralizzata per la creazione e pubblicazione peer-to-peer di contratti intelligenti, o smart contract, creati in un linguaggio di programmazione Turing-completo [5]. Ciò significa che, date le risorse e la memoria, qualsiasi programma eseguito all'interno della piattaforma Ethereum può risolvere ogni tipo di problema calcolabile e la piattaforma può funzionare direttamente come un computer di uso generale. La criptovaluta legata ad Ethereum è l’Ether (o ETH), a oggi (gennaio 2022) la seconda valuta per valore e per importanza dopo Bitcoin.

3.1.1 Componenti di Ethereum

Le principali componenti di Ethereum sono:
• **Macchina a stati**: una macchina virtuale basata su stack che esegue bytecode EVM per le transizioni di stato di Etherum. Questa macchina si chiama Ethereum Virtual Machine (EVM). I programmi che la EVM esegue sono gli smart contract, scritti in linguaggi ad alto livello come Solidity e compilati in bytecode EVM.
• **Strutture dati**: Lo stato di Ethereum è memorizzato localmente su ciascun nodo come un database e contiene le transazioni e lo stato del sistema.
• **Algoritmo di consenso**: Ethereum attualmente utilizza un algoritmo di consenso Proof of Work (PoW), anche se nel futuro prevede un passaggio al Proof of Stake (PoS).
• **Le transazioni**: le transazioni Ethereum sono messaggi di rete che includono un mittente e un destinatario (identificati con indirizzi di 20 byte), un valore espresso in wei (1 wei = 10^{-18} ETH) di fondi inviati e un payload contenente dati.
• **Rete P2P**: Ethereum ha una propria rete di funzionamento, la rete principale di Ethereum, indirizzabile sulla porta TCP 30303.
• **Client**: Ethereum ha diverse implementazioni del software client, tra le quali si annoverano Trinity, Geth e Parity.

3.1.2 La generazione dell'Ether

La generazione dei nuovi ether (di nuova criptovaluta) avviene attraverso un processo competitivo di mining che si sviluppa in una successiva validazione che dà corso alla criptovaluta. L'ether non si propone solo come una criptovaluta sostitutiva alle monete tradizionali, ma viene utilizzata anche per la gestione dei contratti presenti sulla blockchain.

3.1.3 Un po' di terminologia

Prima di proseguire, vale la pena chiarire il significato di alcuni termini chiave.

• **Account Ethereum**: permette di inviare transazioni ed è caratterizzato da un bilancio. È identificato da un indirizzo di 160 bit che può essere condiviso con chiunque voglia interagire con l'account.
In Ethereum si hanno due tipi di account: **account esterni** (o Externally owned Account – **EOA**), controllati da utenti umani, e i **contratti** (o **smart contract**), controllati dal bytecode memorizzato sulla blockchain. Indipendentemente dai tipi di account, essi sono trattati allo stesso modo dall'EVM.
Ogni account ha due campi che ne determinano lo stato:
- il parametro **nonce**: indica il numero di transazioni inviate da quell'indirizzo;
- il **bilancio:** la quantità di ether mantenuta dall'account;

I contratti hanno una memoria permanente dove conservano i dati e un insieme di funzioni che possono essere invocate sia dagli EOA che da altri contratti. Gli EOA e i contratti possono inviare/ricevere ether a/da EOA o altri contratti. Gli EOA possono inviare transazioni alla rete Ethereum per:
1. Creare nuovi contratti;
2. Invocare una funzione di un contratto;
3. Trasferire ether ai contratti o ad altri utenti.

• **Indirizzo Ethereum**: un account Ethereum possiede un indirizzo alfanumerico, che può essere utilizzato per contattarlo (inviare transazioni), mandandogli dell'ether.

• **Wallet**: permette di gestire il proprio account Ethereum. Dai wallet è possibile osservare il bilancio del proprio account, inviare delle transazioni ecc…. La maggior parte dei wallet permette anche di generare un account Ethereum; pertanto, non c'è bisogno di avere già un account prima di scaricare un wallet [7]. Per l'utilizzo del wallet sono necessari due codici:

- **Indirizzo**: che identifica il proprio account attraverso un indirizzo alfanumerico.

- **Chiave privata**: una chiave crittografica segreta usata per firmare le transazioni, necessaria per provare l'identità del mittente.

Vi sono vari tipi di wallet. I più sicuri sono i wallet su hardware, che permettono di mantenere le criptovalute offline. Abbiamo poi le mobile app che permettono di accedere al wallet da ovunque, così come i web wallet interagibili via browser ed infine applicazioni desktop.

3.1.4 Le transazioni

Le transazioni sono eventi che modificano lo stato globale di Ethereum. Una transazione può essere vista come un messaggio che viene inviato da un account a un altro. Esistono due tipi di transazioni: le transazioni esterne, generate da un EOA, ad esempio quando un account utente invia dell'ETH a un altro account, e le transazioni interne, generate dai contratti come risposta a seguito della ricezione di una transazione esterna. Supponiamo, per esempio, che un contratto B abbia una funzione che trasferisce ETH agli account C e D quando B riceve dell'ether da altri account. Quindi, se l'account A invia 5 ether a B, avverranno due transazioni di trasferimento da B a C e da B a D. In questo esempio, la transazione da A a B è una transazione esterna al contratto B, le transazioni da B a C e da B a D sono due transazioni interne del contratto B. Una volta convalidate dai miners, le transazioni, sono impacchettate in blocchi e aggiunte in un libro mastro pubblico, la blockchain: una lista append-only, in continua crescita, contenente record di transazioni e mantenuta da una rete di nodi P2P attraverso un meccanismo di consenso distribuito. Le transazioni però possono anche fallire. Per capire come mai, occorre introdurre il concetto di gas, una criptovaluta separata dall'ETH che consente ai nodi di validare ogni passaggio. Il gas è il carburante di tutta la blockchain, è una valuta virtuale autonoma con un proprio tasso di cambio. Ethereum si serve del gas per controllare la quantità di risorse che una transazione può utilizzare; in mancanza di gas, viene generata un'eccezione "out-of-gas", che causa il fallimento della transazione e che annulla tutte le azioni effettuate da essa, ripristinando lo stato della blockchain. Il gas in eccesso, una volta terminata l'esecuzione della transazione, viene rimborsato al mittente. L'utilizzo del gas è molto importante all'interno del sistema, dato che all'aumentare di esso, cresce la probabilità che una transazione venga considerata dai miners per essere successivamente validata. Inoltre, come detto in precedenza, il linguaggio bytecode di Ethereum è Turing-completo e proprio per questo il gas gioca un ruolo fondamentale in quanto, in sua assenza, un esecutore malintenzionato potrebbe facilmente perturbare la rete creando programmi infiniti e producendo danni irreparabili.
Ogni transazione è composta da:
• **Nonce**: è un valore scalare calcolato in modo dinamico ottenuto contando il numero di transazioni confermate che hanno avuto origine da un indirizzo. L'utilizzo

del nonce garantisce una maggior sicurezza nella transazione e l'impossibilità della sua duplicazione.
• **Gas Price**: rappresenta il prezzo massimo che si intende pagare per ottenere un'unità di gas necessaria per eseguire una qualsiasi transazione. Il Gas Price di solito consiste in una certa quantità di Gwei. Il wei è la più piccola unità di ether, quindi Gwei corrisponde a 1000000000 wei.
• **Gas Limit**: fornisce la quantità massima di gas che il mittente è disposto ad acquistare per una transazione.
Ricapitolando, si decide la quantità massima di carburante da utilizzare (Gas Limit) e il prezzo che si intende pagare alla rete per il carburante che effettivamente si utilizzerà (Gas Price).
• **Destinatario**: l'indirizzo Ethereum di destinazione può essere sia un indirizzo EOA che un contratto. Se il campo di destinazione corrisponde a un indirizzo errato, ovvero senza una chiave privata associata, la transazione avviene comunque, ma l'ether inviato risulterà inaccessibile.
• **Valore e Dati**: il valore rappresenta la quantità di ether da trasferire dal mittente al destinatario, espressa in wei. I dati, nel caso di deployment del contratto, contengono il bytecode e gli argomenti codificati, nel caso, invece, dell'esecuzione di una funzione del contratto, contengono la signature della funzione e gli argomenti codificati. Nel trasferimento di fondi il campo dati viene lasciato vuoto.
• **v, r, s**: questi tre componenti sono utilizzate per generare la firma che identifica il mittente della transazione.

3.1.5 Introduzione agli smart contract

Gli smart contract sono script il cui funzionamento è autonomo. Uno smart contract può essere visto come una macchina deterministica che segue la logica della programmazione tradizionale, ovvero: IF-THIS-THEN-THAT (IFTT); ciò significa che un'operazione viene eseguita soltanto se vengono soddisfatte precise condizioni. Le caratteristiche che contraddistinguono gli smart contract sono il determinismo, l'isolamento e l'immutabilità.
Un programma è deterministico se fornisce ogni volta lo stesso output a fronte dello stesso input. I partecipanti alla rete Ethereum devono ottenere lo stesso risultato dall'esecuzione di uno stesso contratto.
Per isolamento intendiamo che ogni contratto creato rimane rinchiuso in una scatola (sand-box) per evitare che comprometta l'intero sistema: in una blockchain qualsiasi utente può caricare un nuovo contratto e proprio per questo motivo gli smart contract possono contenere (volontariamente o involontariamente) bug e vulnerabilità di qualsiasi genere. Se il contratto non fosse isolato potrebbe danneggiare l'intero sistema portandolo a eseguire comportamenti imprevisti e pericolosi.
Dal momento in cui il codice di un contratto è distribuito sulla blockchain, esso è immutabile e non può essere modificato o manomesso. È possibile l'eliminazione

attraverso la funzione `Selfdestruct()`, ma solo se il creatore del contratto l'ha inserita nel codice.
In Ethereum, gli smart contracts devono essere compilati nel bytecode a basso livello ed eseguiti utilizzando l'EVM. Una volta compilati i contratti vengono distribuiti all'interno del sistema tramite una transazione speciale.
L'indirizzo Ethereum di un contratto può essere utilizzato in una transazione come destinatario, inviando fondi al contratto o chiamando una delle funzioni del contratto.
Un esempio di esecuzione di uno smart contract è riportato in fig. 3.1.

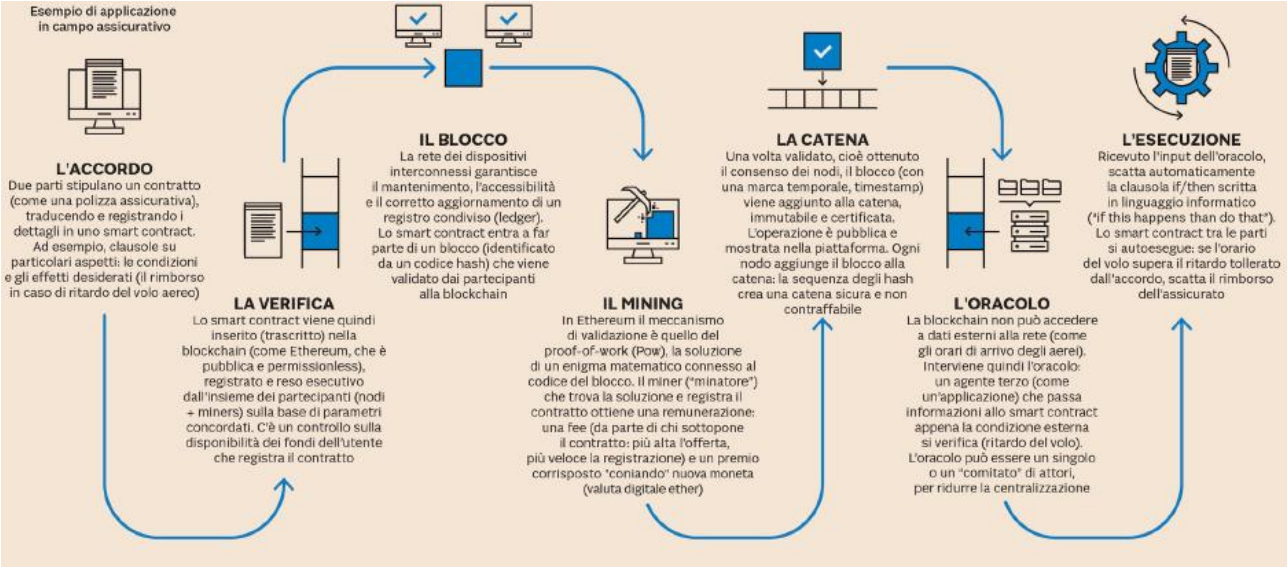


Fig. 3.1. Esecuzione di uno smart contract [8]

3.2 Introduzione a Solidity

In questa sezione si introduce il linguaggio Solidity, utile per comprendere quanto fatto nella sezione 3.4, mediante l'esempio di un contratto che rappresenta un wallet. Il wallet considerato è in fig. 3.2. Il contratto può ricevere ether da altri indirizzi, e il suo proprietario può inviare (parte dell') ETH ad altri utenti tramite la funzione `pay()`. L'hashtable `outflow` registra tutti gli indirizzi a cui il proprietario invia denaro, e associa a ciascuno di essi l'importo totale trasferito. L'hashtable `inflow` registra tutti gli indirizzi da cui ha ricevuto denaro. Tutto l'ether ricevuto è detenuto dal contratto e il suo importo è automaticamente registrato in balance: questa è una variabile speciale, che non può essere alterata dal programmatore. Quando un contratto riceve ETH, esegue anche una speciale funzione senza nome chiamata fallback.
La funzione `AWallet()` alla linea 6 è un costruttore che sarà eseguito solo una volta quando il contratto viene creato. La funzione `pay()` invia l'importo in wei

dal wallet al destinatario. Alla linea 9 il contratto lancia un'eccezione se il chiamante (`msg.sender`) non è il proprietario. Dal momento che le eccezioni ripristinano gli effetti collaterali, l'ETH viene restituito al chiamante, che però perde le fee dovute alla transazione. Alla linea 10, la chiamata termina se la quantità di ETH non è disponibile; in questo caso, non è necessario ripristinare lo stato con un'eccezione. Alla linea 11, il contratto aggiorna il registro di uscita, prima di trasferire l'ETH al destinatario. Alla linea 12 il wallet cerca di inviare l'ETH tramite la funzione `send()`.
La funzione `fallback()` alla linea 16 viene attivata alla ricezione dell'ETH, quando non viene invocata nessun'altra funzione. In questo caso, la funzione aggiorna semplicemente una entry della variabile `inflow`. In entrambi i casi, quando si riceve o si invia l'ether, la quantità totale di ETH del contratto, memorizzata nella variabile this.balance, è automaticamente aggiornata.

```
1  contract AWallet{
2      address owner;
3      mapping (address => uint) public outflow;
4      mapping (address => uint) public inflow;
5
6      function AWallet(){ owner = msg.sender; }
7
8      function pay(int amount, address recipient) returns (bool){
9          if (msg.sender != owner) throw;
10         if (amount > this.balance) return false;
11         outflow[recipient] += amount;
12         if (!recipient.send(amount)) throw;
13         return true;
14     }
15
16     function (){ inflow[msg.sender] += msg.value; }
17 }
```

Fig. 3.2. Un wallet Ethereum scritto in Solidity

3.3 Cos'è uno smart contract Ponzi

Di seguito si propongono quattro requisiti per determinare se uno smart contract è uno schema Ponzi, in tal caso si parla di smart Ponzi. Questi requisiti si basano esclusivamente sulla logica implementata all'interno del contratto.

- R1: il contratto deve spartire ETH fra gli investitori, in accordo ad una certa logica distributiva (i vari tipi di logiche distributive sono descritte nella sottosezione 3.4) implementata al suo interno. Pertanto, il codice del contratto dovrà mantenere una struttura dati che salvi le informazioni degli investitori, fra cui l'indirizzo e la quantità di ether investita;
- R2: il contratto riceve soldi solo da investitori;

- R3: ogni investitore ottiene un profitto, a condizione che i nuovi investitori continuino a inviare denaro al contratto;
- R4: più tardi un investitore entra, maggiore è il rischio di perdere soldi, in altre parole, il rischio di perdere l'investimento cresce al crescere dello schema.

Il requisito R1 richiede che il contratto distribuisca denaro agli investitori. Questo requisito non pone alcun vincolo sulla logica usata per distribuire il denaro; ciò significa che R1 da solo non è sufficiente a classificare un contratto come Ponzi. Per esempio, i giochi d'azzardo, le lotterie, le assicurazioni e obbligazioni, soddisfano R1. Tuttavia, R1 esclude i contratti che forniscono agli utenti qualche tipo di bene, ma non implementano la logica per distribuirli, piuttosto, questi beni sono scambiati attraverso mercati esterni. Questo è il caso, ad esempio, della maggior parte delle implementazioni dei token ERC-20, tra cui le Initial Coin Offerings[1].
Il requisito R2 ci garantisce che per entrare a far parte di questi contratti sia sempre necessario fare un investimento iniziale.
Il requisito R3 richiede che i nuovi investitori continuino a inviare denaro al contratto affinché ogni investitore possa ottenere un profitto. Assieme ai primi due requisiti, implica che gli utenti avranno profitti solo attraverso gli investimenti di altri utenti. Si noti che i giochi d'azzardo, le scommesse e le lotterie violano R3: in questi casi, anche se c'è un flusso costante di investimenti, un utente sfortunato non ha la garanzia di ottenere alcun profitto, ad esempio perdendo sempre. Da questo requisito deriva R4.
R4 asserisce che a un certo punto diventa impossibile fare soldi e ciò comporta il collasso dello schema. L'idea è che nel momento in cui il cerchio degli investitori smette di allargarsi, coloro che avevano già ottenuto una reward avranno guadagnato, gli ultimi entrati, invece, hanno solo investito per pagare le reward ai partecipanti precedenti, perdendo dunque i loro soldi. A questo punto nessuno riuscirà più a ottenere dei profitti e lo schema crolla.

Da R1 ed R2 si capisce che: uno smart Ponzi ha fondamentalmente due tipi di azioni:
• **azione di investimento (I)**: un partecipante invoca una transazione per investire ETH, e i contratti salvano informazioni degli utenti come l'indirizzo del destinatario (msg.sender) e la quantità investita (msg.value).
• **azione di reward (R)**: il contratto paga un bonus ai partecipanti secondo uno schema prestabilito.

[1] ICO indica un mezzo non regolamentato di crowdfunding nel settore finanziario. Le prime ICO furono lanciate per raccogliere fondi per nuove criptovalute, ma le attuali ICO vengono usate per qualsiasi scopo. Nelle ICO, generalmente, sono venduti dei token per raccogliere denaro.

3.4 Tassonomia degli schemi di Ponzi

L'articolo [2] ha analizzato il codice sorgente di vari smart Ponzi, e propone una tassonomia approssimativa che li classifica in base al modello utilizzato per ridistribuire il denaro. Sono state identificate quattro categorie: tree-shaped schemes, chain-shaped schemes, handover schemes e waterfall schemes, che sono descritte di seguito.

3.4.1 Tree-shaped schemes

Questi schemi usano una struttura dati ad albero per indurre un ordinamento tra gli utenti. Ogni volta che un utente si unisce allo schema, dovrà indicare un altro utente come invitante, che diventerà il suo nodo genitore. Se non viene indicato nessun invitante, il genitore sarà il nodo radice, cioè il proprietario dello schema. Il denaro investito dal nuovo utente verrà suddiviso tra i suoi antenati con la logica che più un antenato è vicino e maggiore è la sua quota. Non c'è limite al numero di figli che un nodo può avere; perciò, più figli il nodo avrà e più guadagnerà. In fig. 3.3 viene mostrato un archetipo di questo schema.

```
1  contract TreePonzi {
2
3    struct User {
4      address inviter;
5      address itself;
6    }
7    mapping (address=>User) tree;
8    address top;
9
10   function TreePonzi() {
11     tree[msg.sender] =
12       User({itself: msg.sender,
13             inviter:msg.sender});
14     top = msg.sender;
15   }
16   function enter(address inviter) public {
17     if ((msg.value < 1 ether) ||
18       (tree[msg.sender].inviter != 0x0) ||
19       (tree[inviter].inviter == 0x0)) throw;
20
21     tree[msg.sender] = User({itself: msg.sender,
22                              inviter: inviter});
23     address current = inviter;
24     uint amount = msg.value;
25     while (next != top) {
26       amount = amount/2;
27       current.send(amount);
28       current = tree[current].inviter;
29     }
30     current.send(amount);
31   }}
```

Fig. 3.3. Uno schema Tree-shaped

Per aderire allo schema, attraverso la funzione `enter()`, un utente deve inviare del denaro, e deve indicare un invitante (che sarà il suo nodo padre). Se l'importo è troppo basso (linea 17), o se l'utente è già presente (linea 18), o se l'invitante non esiste (linea 19), l'utente viene rifiutato; altrimenti viene inserito nell'albero (linea 21). Una volta che l'utente è iscritto, il suo investimento viene ripartito tra i suoi antenati (righe 25-29), dimezzando l'importo ad ogni livello. In questo schema, un utente non può prevedere quanto guadagnerà: questo, infatti, dipende da quanti utenti è in grado di invitare e da quanto essi investiranno. L'unico che ha la garanzia di avere profitto è il proprietario, cioè il nodo radice dell'albero.

3.4.2 Chain-shaped schemes

Sono un caso particolare di schemi ad albero in cui ogni nodo ha esattamente un figlio (l'ordinamento indotto tra gli utenti è lineare). Gli schemi in questa categoria di solito moltiplicano l'investimento per un fattore costante predefinito, che è uguale per tutti gli utenti. Lo schema inizia a pagare gli utenti, uno alla volta, in ordine di arrivo, e per intero: tutti i nuovi investimenti vengono raccolti fino a quando non si ottiene l'importo dovuto. In quel momento, il contratto rimanda il pagamento in un solo colpo e passa all'utente successivo nella catena. L'importo da investire può essere fisso, libero, o avere un limite inferiore. Di solito, il proprietario del contratto trattiene una commissione da ogni investimento.
Mostriamo in fig. 3.4 un tipico schema chain-shaped, che raddoppia l'investimento di ogni utente.
Per aderire allo schema, un utente invia `msg.amount` ETH al contratto, innescando così la funzione di `fallback()` (linea 16). Il contratto richiede una quota minima di 1 ETH: se `msg.amount` è inferiore a questo minimo, l'utente viene rifiutato (linea 17); altrimenti, il suo indirizzo viene aggiunto all'array (linea 19). Il proprietario del contratto trattiene il 10% dell'investimento (riga 22). Con i fondi rimanenti, il contratto cerca di ripagare gli utenti precedenti. Se ha l'ETH necessario a pagare l'utente individuato, il contratto paga all'utente il suo investimento moltiplicato per 2 (linea 25). Dopodiché, il contratto cerca di pagare l'utente successivo, e così via fintanto che ha ether a sufficienza. In questo schema, un utente può prevedere esattamente quanto guadagnerà, a condizione che lo schema continui a funzionare.

```
1  contract ChainPonzi {
2
3    struct User {
4      address addr;
5      uint amount;
6    }
7    User[] public users;
8    uint public paying = 0;
9    address public owner;
10   uint public totalUsers=0;
11
12   function ChainPonzi() {
13     owner = msg.sender;
14   }
15
16   function() {
17     if (msg.value < 1 ether) throw;
18
19     users[users.length] = User({addr: msg.sender,
20                                 amount: msg.value});
21     totalUsers += 1;
22     owner.send(msg.value/10);
23
24     while (this.balance > users[paying].amount * 2) {
25       users[paying].addr.send(users[paying].amount * 2);
26       paying += 1;
27     }
28   }}
```

Fig. 3.4. Uno schema chain-shaped

3.4.3 Waterfall schemes

Gli schemi waterfall sono simili ai chain-shaped per quanto riguarda l'ordinamento degli utenti, ma diversi per la logica di distribuzione del denaro. Ogni nuovo investimento viene riversato lungo la catena di investitori, in modo che ognuno possa prendere la sua parte. La logica è di tipo FIFO: "primo arrivato,

primo servito". La distribuzione parte sempre dall'inizio della catena, ed è dunque più probabile che gli utenti più avanti (quindi quelli entrati tardi) nella catena non ricevano mai denaro.
In fig. 3.5 è mostrato un archetipo di questo schema. Il contratto ha una fee di ingresso pari o superiore a 1 ETH (linea 19), il 10% della fee viene dato al proprietario (linea 24-25) e il contratto effettua un pagamento del 6% degli investimenti agli utenti ogni volta che ottiene ETH. Se il bilancio del contratto è sufficiente per pagare il primo utente nell'array (posizione 0), allora il contratto invia a quell'utente il 6% del suo investimento originale (righe 29-30). Dopodiché, il contratto prevede di pagare l'utente successivo nell'array, e così via fino all'esaurimento dell'ether da esso mantenuto. All'investimento successivo, l'array viene iterato di nuovo, sempre partendo dal primo utente. Per assicurare che tutti i partecipanti ricevano i pagamenti (coerentemente con il requisito R3), gli investimenti dei nuovi utenti devono crescere proporzionalmente rispetto al loro numero.

```
1  contract WaterfallPonzi {
2
3    struct User {
4        address addr;
5        uint amount;
6    }
7
8    User[] public users;
9
10   uint pos = 0;
11   uint public totalUsers=0;
12   address public owner;
13   uint public fees = 0;
14
15   function WaterfallPonzi() {
16     owner = msg.sender;
17   }
18   function() {
19     if (msg.value < 1 ether) throw;
20
21     users[totalUsers] = User({addr: msg.sender,
22                               amount:msg.value});
23     totalUsers += 1;
24     fees = mgs.value / 10;
25     owner.send(fees);
26
27     pos=0;
28     while (this.balance >= users[pos].amount
           *6/100 && pos<totalUsers){
29       users[pos].etherAddress.send
           (users[pos].amount * 6/100);
30       pos += 1;
31   }}}
32
```

Fig. 3.5. Uno schema Waterfall

3.4.4 Handover schemes

Gli schemi Handover sono un tipo di schema chain-shaped, dove la somma di ingresso è determinata dal contratto, ed è aumentata ogni volta che un nuovo investitore si unisce allo schema. L'investimento iniziale di un nuovo partecipante è dato per intero a quello precedente, che otterrà un profitto immediato. Non appena l'utente viene pagato, cede questo privilegio al prossimo utente.
Un esempio di questo tipo di schema è mostrato in fig. 3.6.
Per aderire allo schema un utente deve inviare ETH al contratto, per innescare la funzione di `fallback()` (linea 11). Il contratto inoltra questa somma al precedente utente, meno una tassa che viene mantenuta dal contratto (linea 13). Poi, l'indirizzo del nuovo utente viene registrato (linea 14) e la variabile price viene

raddoppiata (linea 15), ciò significa che adesso entrare nello schema costerà il doppio.

```
1  contract HandoverPonzi {
2    address owner;
3    address public user;
4    uint public
5      price = 100 finney;
6
7    function HandoverPonzi() {
8      owner = msg.sender;
9      user = msg.sender;
10   }
11   function() {
12     if (msg.value < price) throw;
13     user.send(msg.value * 9 / 10);
14     user = msg.address;
15     price = price * 2;
16   }
17
18   function sweepCommission(uint amount)  {
19     if (msg.sender == owner) owner.send(amount);
20   }}
```

Fig. 3.6. Uno schema Handover

Il proprietario del contratto può ritirare la sua quota chiamando la funzione `sweepCommission()`. Negli schemi Handover, al momento dell'investimento gli utenti sanno esattamente quanto guadagneranno. Tuttavia, poiché il pedaggio aumenta man mano che lo schema va avanti, è più probabile che gli utenti successivi perdano i loro soldi (coerentemente con il requisito R4).

3.4.5 Analisi della ridistribuzione dell'Ether

Gli schemi Ponzi hanno la particolarità che ogni investitore può realizzare un profitto, a condizione che un numero sufficiente di investitori dopo di lui immetta abbastanza denaro nel contratto (requisito R3).

3.4.5.1 Ridistribuzione nei Chain-shaped

Consideriamo uno schema chain-shaped che raddoppia il denaro ricevuto, accetta pedaggi d'ingresso di 1 ETH, e non ha spese per il proprietario tranne il primo ether inviato al contratto. Supponiamo che il proprietario, U0, invii 1 ETH così come il primo utente U1. L'ether di U1 viene dato a U0, e il bilancio del contratto torna a 0. Affinché U1 riveda il suo ether più l'altro promesso (dato che il contratto raddoppia l'investimento), egli deve aspettare che altri due utenti U2 e U3 aderiscano allo schema, inviando 1 ETH ciascuno. Così, U2 deve aspettare che U1 riscatti la sua quota, e poi deve aspettare che U3, U4 ed U5 inviino ether. L'utente U3, che è l'ultimo del suo livello, deve aspettare che il livello successivo sia pieno, il che dà un totale di 4 utenti da aspettare. In generale, un utente Uk al livello i deve aspettare che tutti gli utenti del livello precedente abbiano riscattato la loro quota, e poi deve aspettare tutti quelli del suo livello che sono arrivati prima di lui. Se Uk è il primo nodo al livello i, deve aspettare che tutti gli altri utenti del livello i-esimo si uniscano (per pagare gli utenti del livello i - 1), più i 2 necessari per riscattare la sua quota, ciò richiede 2^i - 1 + 2 utenti. Poiché la quantità totale di nodi fino al livello i – 1 è 2^i – 1 e poiché Uk è il primo al livello i, abbiamo che k

è uguale 2^i e quindi, nel caso migliore, Uk deve aspettare k + 1 utenti. Invece, se Uk è l'ultimo utente al livello i, deve aspettare tutti gli altri utenti al livello i + 1. Questo richiede 2^{i+1} utenti. Poiché $k = 2^{i+1} - 1$, in questo caso Uk deve aspettare k + 1 utenti cioè 2^{i+1}.
In un contratto che non pone limiti su quanto si possa investire, un investimento insolitamente alto potrebbe far sì che il contratto smetta di inviare i payouts per molto tempo, scoraggiando così i nuovi utenti ad unirsi. Inoltre, tasse più alte per il proprietario e fattori di moltiplicazione più alti rallenteranno il flusso, così che i nostri risultati costituiscono un limite inferiore al numero di utenti da aspettare.

3.4.5.2 Ridistribuzione nei Tree-shaped schemes

Le considerazioni sopra valgono come limite inferiore anche per gli schemi ad albero, poiché questi sono rallentati dal fatto che i nuovi utenti non possono essere tutti discendenti di un dato nodo.

3.4.5.3 Ridistribuzione nei Waterfall schemes

Supponiamo uno schema waterfall con un pedaggio fisso di 1 ETH, nessuna tassa, e che dà ad ogni utente il 10% dell'importo investito. Per ogni nuovo investitore, i vecchi hanno diritto a 0.1 ETH: quindi, sono necessari 10 nuovi utenti per ripagare l'investimento del primo utente, e altri 10 utenti per permettergli di raddoppiare il suo investimento. Si noti che per i primi dieci utenti, l'importo che stanno dando non è interamente distribuito: una parte viene lasciata all'interno del contratto. Tuttavia, dopo che il decimo investitore si unisce alla catena, il denaro che sta investendo non è sufficiente per essere diviso tra tutti gli utenti: da quel momento in poi, il contratto deve usare i propri fondi per riempire il gap. Alla fine, anche questa somma finirà: man mano che lo schema avanza non importa quanti altri investitori si uniranno, ma solo i primi 10 utenti hanno la garanzia di ricevere introiti. Quindi, per garantire che ogni utente possa raddoppiare il suo investimento, dobbiamo fare in modo che gli investimenti siano distribuiti su tutti gli utenti. Supponiamo ora che, per aderire al sistema, un utente debba dare 0.1 ETH volte il numero di utenti già iscritti. Il primo utente investirà 0, e l'undicesimo utente investirà 1 ETH. Con una tale regola, se lo schema contiene n utenti, il k-esimo utente ha dato 0.1*(k-1) ETH mentre riceve 0.1*(n-k) ETH. All'aumentare del numero di utenti, anche il denaro ricevuto cresce. Per esempio, il terzo utente che si unisce allo schema investirà 0.2 ETH, e riceverà 0.4 ETH non appena altri 4 utenti si uniscono. Invece, il cinquantesimo utente deve dare un pedaggio di 4.9 ETH, e deve aspettare 98 nuovi utenti per raddoppiare l'investimento. Tuttavia, poiché il pedaggio aumenta per ogni nuovo investitore, il sistema risulterà meno attraente man mano che avanza.

3.4.5.4 Ridistribuzione negli Handover schemes

Negli schemi Handover, affinché un investitore riceva un pagamento è sufficiente aspettare che un altro utente si unisca. Tuttavia, poiché il pedaggio continua ad aumentare, entrare nello schema è meno appetibile a mano a mano che questo cresce.

Nel complesso, abbiamo dimostrato che i requisiti R3 e R4 sono validi per tutti i tipi di schemi che abbiamo identificato, vale a dire che ogni investitore ha la garanzia di guadagnare denaro se ne viene investito in seguito, ma i finanziatori che entrano tardi nello schema hanno un rischio maggiore di perdere il loro investimento.

4. Dall'analisi della documentazione alla creazione di un dataset unificato

In questo capitolo si descrivono le pubblicazioni studiate per la stesura della tesi: [1], [2], [3] e [4], concentrandosi su come sono stati utilizzati i vari articoli e su come questi hanno influito sullo svolgimento della tesi. Man mano che si avanza con la discussione degli articoli si descrivono anche la creazione del dataset, utilizzato per la costruzione del classificatore, e le features inserite all'interno esso. In particolare, saranno presentati i passaggi svolti per passare dai tre dataset iniziali, presi dagli articoli [2], [3] e [4], a uno unico e standardizzato.

4.1 Analisi dei contributi di [1]

Nell'articolo "Data mining for detecting Bitcoin Ponzi schemes" [1], pubblicato da Massimo Bartoletti, Barbara Pes, Sergio Serusi nel 2018 per la Crypto Valley Conference on Blockchain Technology, viene fatta una introduzione sul problema degli schemi Ponzi su Bitcoin.
Benché questo articolo si focalizzi sullo studio di Bitcoin, alcune idee possono essere applicate anche per Ethereum, ad esempio certe features individuate dagli autori per effettuare machine learning.
Le features individuate per caratterizzare la differenza fra indirizzi Ponzi e indirizzi non Ponzi[2] sono le seguenti:

- Lifetime: la durata di vita dell'indirizzo, calcolata come la differenza fra la prima e l'ultima transazione effettuata o ricevuta dall'indirizzo.
- Activity days: numero di giorni in cui c'è stata almeno una transazione effettuata dall'indirizzo o ricevuta dall'indirizzo.
- Il massimo numero di transazioni giornaliere.
- Il coefficiente di Gini del valore in Bitcoin trasferito all'indirizzo (rispettivamente dall'indirizzo). I coefficienti di Gini sono una rappresentazione standard del grado di disuguaglianza della ricchezza: 0 indica la perfetta uguaglianza della distribuzione, mentre 1 è l'assoluta disuguaglianza.
- La quantità di denaro trasferita all'indirizzo e in uscita dall'indirizzo.
- Il numero di transazioni in ingresso all'indirizzo che trasferiscono denaro all'indirizzo.

[2] essendo su Bitcoin non si può parlare di contratti ma solo di indirizzi.

- Il numero di transazioni in uscita dall'indirizzo che inviano denaro ad altri indirizzi.
- Il rateo fra le transazioni in ingresso ed in uscita dall'indirizzo.
- Media e deviazione standard del denaro trasferito all'indirizzo.
- Il numero di indirizzi diversi che hanno inviato denaro all'indirizzo Ponzi e successivamente hanno ricevuto indietro soldi.
- Il divario minimo, massimo e medio in termini di tempo da quando l'indirizzo ha ricevuto dei soldi e li ha rinviati.
- La differenza massima nel bilancio dell'indirizzo in due giorni consecutivi.

In questo articolo, oltre a descrivere le features, si descrive anche il problema della costruzione di un classificatore mediante un dataset pesantemente sbilanciato (32 indirizzi Ponzi e 6400 non Ponzi). Ciò ha portato gli autori dell'articolo a sperimentare tecniche come il Random Oversampling, il Random Undersampling e ad usare matrici di costo per dare più peso alla giusta classificazione dei contratti Ponzi rispetto alla giusta classificazione di quelli non Ponzi. Citando [1]: "In fraud detection applications, as in many domains with imbalanced class distributions, a correct classification of the rare class (i.e., the 'Ponzi' class in our problem) is far more important than a correct classification of the majority class". Per ultimo parlano anche dell'importanza di usare le metriche giuste per questi tipi di problemi. Le metriche che hanno preso in considerazione sono presentate in fig. 4.1.

Accuracy ((TP + TN)/(TP + TN + FP + FN)) is the fraction of test instances whose class is predicted correctly;
Specificity (TN/(TN+FP)) is the fraction of negative instances classified correctly;
Sensitivity (TP/(TP+FN)), also called **Recall**, is the fraction of positive instances classified correctly;
Precision (TP/(TP+FP)) is the fraction of instances that actually are positive in the group the model has predicted as positive;
F-measure (2·Precision·Recall / (Precision+Recall)) is the harmonic mean between precision and recall;
G-mean $(\text{Recall}\cdot\text{Specificity})^{(0.5)}$ is the geometric mean between specificity and recall;
AUC is the area under the *Receiver Operating Characteristics* (ROC) curve, which shows the trade-off between true positive and false positive rates (the better the model, the closer the area is to 1).

Fig. 4.1. Rappresenta le metriche utilizzate per valutare la bontà del classificatore costruito da [1].

4.2 Analisi dei contributi di [2]

L'articolo "Dissecting Ponzi schemes on Ethereum: Identification, analysis, and impact" [2], scritto da Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, Roberto Saia, è una pubblicazione introduttiva al mondo degli schemi Ponzi sulla blockchain di Ethereum. Fornisce una panoramica generale di come funzionano questi schemi implementati negli smart contract e stila una misura del loro impatto economico, quantificando il valore complessivo scambiato attraverso essi ed effettuando una analisi dei guadagni e delle perdite degli utenti.
Da questo articolo è stata ricavata una prima definizione di schema Ponzi, che successivamente abbiamo ampliato e dettagliato (presentata nel capitolo 3) e la tassonomia degli smart Ponzi (anche questa esposta nel capitolo 3). Oltre a ciò, questo articolo fornisce anche un dataset costituito da soli contratti Ponzi.

4.2.1 Dataset

I passi seguiti dagli autori dell'articolo per la costruzione del dataset sono stati i seguenti:
• In primis è stato recuperato il codice Solidity dei contratti pubblicati su Ethereum tramite il sito etherscan.io[3], un blockchain explorer per la rete Ethereum, che consente di cercare transazioni, blocchi, indirizzi di wallet e di smart contract. Ispezionando a mano questi contratti sono stati individuati 138 schemi che soddisfano i quattro criteri definiti nel capitolo 3.
• Al fine di incrementare questo primo dataset è stata effettuata una seconda ricerca, questa volta direttamente sulla blockchain di Ethereum basandosi sulle similitudini del bytecode. Per far ciò è stato usato un algoritmo Monte Carlo per stimare la distanza di Levenshtein fra due contratti. La Normalized Levenshtein Distance (NLD) è la misura della similarità fra due stringhe. Il risultato che si ottiene è una metrica il cui valore è un numero reale fra 0 ed 1, dove 0 indica l'uguaglianza tra le due stringhe mentre 1 indica una disuguaglianza completa. Sono stati così comparati i bytecodes del primo dataset con quelli della blockchain. Se il valore della NLD risulta minore o uguale di 0.35 questi vengono etichettati come potenziali smart Ponzi.
• Una volta ottenuto il bytecode dei potenziali Ponzi, è stato utilizzato un decompiler[4] per passare da bytecode EVM a Solidity. Ottenuto il codice Solidity sono stati analizzati a mano i contratti trovati allo step 2 e 46 di questi sono stati classificati

[3] https://etherscan.io/
[4] https://ethervm.io/decompile

come Ponzi. In conclusione, il dataset è formato da 184 schemi Ponzi, disponibile all'indirizzo: goo.gl/CvdxBp.
Questo dataset, così come le altre considerazioni presentate in questo articolo, sono stati il punto di partenza per svariati studi, incluso il mio. Seguendo il lavoro svolto in [4], che sarà ampliamente descritto più avanti nel capitolo, questo dataset è stato ripulito da 51 smart Ponzi, in quanto duplicati a livello di bytecode o perché non rispecchiano i quattro requisiti definiti nel capitolo 3 per essere considerati degli schemi Ponzi.
Le informazioni di questi contratti sono state recuperate dagli autori di [1] tramite uno script Python che sfrutta le API del sito etherscan.io. Lo script è disponibile su GitHub: https://github.com/blockchain-unica/ethereum-ponzi/blob/master/ponzi-stats.py, ed è stato il punto di partenza del programma che è stato sviluppato e descritto nella sezione 4.6.

4.2.2 Features individuate

Le informazioni contenute in questo dataset sono le seguenti:

- *IN TX*: il numero di transazioni in ingresso nel contratto
- *OUT TX*: il numero di transazioni in uscita dal contratto
- *IN ETH*: ether in ingresso nel contratto
- *OUT ETH*: ether in uscita dal contratto
- *IN USD*: quantità di denaro espressa in dollari in ingresso nel contratto
- *OUT USD*: quantità di denaro espressa in dollari in uscita dal contratto

Queste due metriche le ho ritenute poco interessanti data la volatilità del valore dell'ether.

- *Paying*: numero di contratti che hanno pagato il contratto
- *Paid*: numero di contratti che sono stati pagati dal contratto
- *Date Firts tx*: data prima transazione, coincide con la data di creazione del contratto
- *Date Last tx*: data ultima transazione, coincide con la data dell'ultima transazione registrata su etherscan.io
- *Lifetime*: durata di vita del contratto espressa in giorni

4.3 Analisi dei contributi di [3]

Nell'articolo "Exploiting Blockchain Data to Detect Smart Ponzi Schemes on Ethereum" [3], scritto da Weili Chen, Zibin Zheng, Edith Cheuk-Han Ngai, Peilin Zheng, Yuren Zhou, si descrivono metodi basati sul machine learning per effettuare la classificazione degli smart Ponzi. Le tecniche di machine learning adottate da questo paper sono basate sia sulle account features (cioè le informazioni ottenute a partire dalle transazioni ricevute o effettuate dal contratto) che le code features (legate agli opcodes. Ogni byte dell'EVM bytecode rappresenta una operazione, la forma mnemonica di queste operazioni sono gli opcodes. L'appendice dello yellow paper di Ethereum contiene la lista completa sia dell'EVM bytecode che degli opcodes [13]).
In questo sommario si descrivono le sole features relative all'account dato che sia il dataset che, di conseguenza, il classificatore sono stati costruti utilizzando queste.

4.3.1 Dataset

Il dataset originale consiste in 200 smart Ponzi di cui la maggior parte (134) provenienti dal dataset dell'articolo [2] e 66 trovati dagli autori tramite un'analisi manuale di 3780 contratti presenti sulla blockchain di Ethereum.
Le features descritte ed usate da questo articolo non sono state inserite dagli autori nel dataset pubblicato e pertanto è stato necessario implementare uno script che le calcolasse.

Dopo aver rimosso manualmente i 134 contratti già presenti nel dataset di [1], sui 66 rimanenti sono stati effettuati maggiori controlli: è stato reperito il bytecode dal sito contract-library.com[5], ed è stato scritto un programma in C che calcola la distanza di Levenshtein per capire se fra questi ci fossero dei duplicati. Il programma è molto semplice: acquisisce da tastiera i nomi dei due file necessari: indirizzi.txt - il file contenente gli indirizzi dei 66 contratti - e bytecode.txt - il file contenente i bytecodes dei contratti. Dopo aver letto questi file, si inserisce il contenuto in due array di stringhe, un array per gli indirizzi e l'altro per i bytecodes, ottenendo due strutture dati con 66 entries e con una corrispondenza biunivoca. A questo punto si calcola la distanza di Levenshtein tra i bytecodes presenti. Una volta stilate le distanze per tutti i contratti, si verifica se siamo in presenza di doppioni o se ci sono dei contratti che possono essere considerati simili. Il codice è riportato in fig. 4.2.
La variabile `min` contiene il minor valore trovato (e quindi il bytecode più simile a quello che stiamo analizzando) nella matrice `LevDist`, che contiene i valori

[5] https://contract-library.com/

calcolati con l'algoritmo di Levenshtein. Se `min` vale 0, abbiamo trovato un contratto con lo stesso bytecode di quello sotto esame, pertanto possiamo terminare il ciclo iniziato a linea 140.
Se alla fine del ciclo interno `min` non vale 0 significa che il contratto sotto analisi non ha un doppione, ma ce ne può essere uno simile. Negli if successivi (linee 159-163), dopo aver normalizzato la distanza, che passa così da essere uno scalare a numero reale tra 0 e 1, si controlla se ci sono dei contratti che possono essere ritenuti molto simili, cioè con un valore di Levenshtein inferiore a 0.05.
A seguito dell'esecuzione di questo codice ho trovato 6 contratti duplicati (a due a due doppioni) e 21 affini tra loro, 3 dei 6 contratti duplicati sono stati rimossi e i 21 segnalati come contratti tra loro molto simili.

```
138     int pos = -1;
139
140     for(int i=0; i<NUM_SMART_PONZI; i++){
141         int min = INT_MAX; // inizializzo il minimo al massimo intero possibile
142         for(int j=0; j<NUM_SMART_PONZI; j++){
143             if(i != j){ // non siamo sulla diagonale (che ricordiamo non contiene valori)
144                 if(LevDist[i][j] < min){
145                     min = LevDist[i][j];
146                     pos = j;
147                 }
148                 if(min == 0){ // abbiamo una equivalenza completa, si tratta sicuramente di un doppione
149                     printf("lo smart contract %s e' una copia dello smart contract %s \n", addresses[i], addresses[pos]);
150                     break;
151                 }
152             }
153         } // finito il ciclo piu' interno, se min != 0, vuol dire che non abbiamo una copia perfetta, ma potremmo avere degli smart contract molto simili lo stesso
154         if(min != 0){
155             double normalizedLD;
156             if(strlen(bytecodeSC[i]) >= strlen(bytecodeSC[pos]))
157                 normalizedLD = (double)min/strlen(bytecodeSC[i]);
158             else
159                 normalizedLD = (double)min/strlen(bytecodeSC[pos]);
160
161             if(normalizedLD < 0.05)
162                 printf("lo smart contract %s potrebbe essere una copia dello smart contract %s \n", addresses[i], addresses[pos]);
163             else
164                 printf("lo smart contract %s non ha copie nel dataset \n", addresses[i]);
165         }
166     }
```

Fig. 4.2. Il ciclo for del programma che compara i bytecodes dei contratti Ponzi di [3].

4.3.2 Estrazione delle features

Gli autori di [3], basandosi su quanto fatto in [2], hanno individuato criteri simili o comunque ricavabili da quelli descritti nel capitolo 3 per identificare gli smart Ponzi:
1) i contratti Ponzi di solito inviano ether a coloro che investito su di essi;
2) alcuni indirizzi ricevono un numero ed una quantità in ether di pagamenti maggiore rispetto al numero dei suoi investimenti. Ad esempio, il creatore è pagato più frequentemente dal contratto rispetto ad altri utenti;
3) per mantenere l'immagine di contratti ad alto rendimento, questi tipi di smart contract pagano gli investitori non appena hanno un bilancio in positivo, il che li porta spesso ad avere un bilancio inferiore rispetto ai contratti non Ponzi.

Per capire il comportamento di uno schema Ponzi è stato analizzato lo storico delle transazioni di un noto smart Ponzi: Rubixi.
1) Le transazioni di pagamento di solito si verificano dopo l'investimento, il che indica che il contratto di solito paga indirizzi che sono "conosciuti".
2) Molti partecipanti non ricevono nulla dal contratto.
3) Alcuni partecipanti hanno più transazioni di pagamento che operazioni di investimento.

Sulla base di queste osservazioni, sono state estratte 13 account features. Sono stati così definiti due vettori differenza V_1 e V_2, che descrivono rispettivamente la differenza nel numero e nell'ammontare dei pagamenti e degli investimenti di tutti coloro che partecipano alla vita del contratto. Se ci fossero p partecipanti, V_1 e V_2 sarebbero dunque vettori di lunghezza p. L'i-esimo elemento di V_1 è $V_1[i]$ = n[i] – m[i] dove n[i] denota il numero di investimenti fatti (verso il contratto) dall'i-esimo partecipante mentre m[i] il numero di pagamenti (ricevuti).
Le account features introdotte da questo paper sono le seguenti:
- *Balance*: il bilancio dello smart contract, inteso la differenza tra la sommatoria dell'ether entrante e la sommatoria dell'ether uscente.
- *N_maxpay*: il numero massimo di pagamenti che il contratto ha fatto verso un partecipante (considerando tutti i partecipanti).
- *N_investment*: il numero di investimenti al contratto.
- *N_payment*: il numero di pagamenti dal contratto agli utenti.
- *Paid_one*: la proporzione di investitori che hanno ricevuto almeno un pagamento.
- *Known_rate*: la proporzione di riceventi che hanno investito prima del pagamento da parte del contratto. Negli schemi Ponzi ci aspettiamo un known rate molto alto. Riguarda un'osservazione fatta precedentemente: uno smart Ponzi tende a pagare più spesso contratti che già conosce.
- *Difference counts mean*: La media del vettore differenza V_1.
- *Difference counts skewness*: la asimmetria del vettore differenza V_1. La skewness è il grado di asimmetria osservabile in una distribuzione di probabilità. Le distribuzioni possono presentare un'asimmetria destra (positiva) o sinistra (negativa). Una distribuzione normale (curva a campana) presenta un'asimmetria pari a zero.
- *Difference counts standard deviation*: La deviazione standard di V_1, per indicare la dispersione nella distribuzione dei valori.
Queste ultime tre features vengono calcolate anche per V_2.
- *Paid rate*: N_payment/N_investment, il rapporto fra il numero di pagamenti fatti dal contratto ed il numero di investimenti ricevuti.

L'introduzione di media, deviazione standard e asimmetria derivano dal fatto che le loro caratteristiche catturano la differenza fra guadagno e spese degli

investitori che sono considerate la base per capire se siamo di fronte ad uno schema Ponzi.

4.3.3 Classificatori

In [3] sono stati testati vari classificatori, tra cui SVM, Decision Tree, Random Forest e XGBoost ecc.... Quelli che hanno restituito risultati migliori sono stati gli ultimi due. Nell'articolo vengono definiti anche i valori provati per gli iperparametri per i classificatori Random Forest e XGBoost, rappresentati in fig. 4.3.
Per la costruzione del modello, il dataset è stato diviso in 80% training e 20% testing. Le metriche per valutare i classificatori sono riportate in fig. 4.4 e risultano essere un sottoinsieme delle metriche utilizzate in [1].
Fra i vari modelli testati il migliore, per quanto riguarda le account features, risulta essere XGBoost, in quanto ottiene una recall e F1 superiore a random forest.
I risultati ottenuti dai principali modelli testati dal paper sono riportati in fig. 4.5.

```
model_params = {
    'random_forest': {
        'model': RandomForestClassifier(),
        'params': {
            'n_estimators':[20,40,60,80,100,120,140,160,180,200],
            'criterion':['gini', 'entropy'],
            'bootstrap': [True, False]
        }
    },
    'xgboost': {
        'model': xgb.XGBClassifier( eval_metric = 'aucpr',  use_label_encoder = False),
        'params': {
                'n_estimators' : [100,110,120,130,140,150,160,170,180,190,200],
                'learning_rate': [0.1,0.2,0.3],
                'max_depth': [3,6,8,9]
        }
    }
}
```

Fig. 4.3. Rappresenta i valori assegnati agli iperparametri per i classificatori Random Forest e XGBoost.

$$Precision = \frac{true\ positive}{true\ positive + false\ positive}$$

$$Recall = \frac{true\ positive}{true\ positive + false\ negative}$$

$$F - score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

Fig. 4.4. Metriche utilizzate da [3] per valutare le prestazioni del classificatore.

Algorithm	Precision	Recall	F1
SVM	0.32	0.06	0.09
RF	0.64	0.20	0.30
XGB	0.59	0.22	0.32

Fig. 4.5. Risultati ottenuti da [3] sui modelli testati.

4.4 Analisi dei contributi di [4]

L'ultimo articolo analizzato: "SADPonzi: Detecting and Characterizing Ponzi Schemes in Ethereum Smart Contracts" [4], scritto da Weimin Chen, Xinran Li, Yuting Sui, Ningyu He, Haoyu Wang, Lei Wu, Xiapu Luo, affronta un nuovo modo di identificare schemi Ponzi negli smart contract, un metodo basato sull'analisi semantica. Il sistema proposto può rilevare gli schemi Ponzi in base al loro bytecode confrontando le informazioni semantiche estratte con quelle sintetizzate dagli schemi Ponzi ground truth (cioè un insieme di dati ritenuti corretti, nel nostro caso si intende che i contratti Ponzi siano effettivamente Ponzi e i non Ponzi siano effettivamente tali).

I principali contributi che questo articolo ci ha fornito sono stati:

1- la pubblicazione di un dataset contenente 835 schemi Ponzi e oltre 1000 contratti non Ponzi. Il dataset è disponibile all'indirizzo: https://github.com/Kenun99/SADPonzi.
2- La rimozione di alcuni schemi Ponzi presenti nel dataset proposto in [1], da cui gli autori di questo articolo hanno attinto per ottenere un insieme di dati da utilizzare come ground truth. In questa parte sono stati analizzati a mano i 184 contratti individuati nel paper [1] e sono stati rimossi 51 contratti. Di seguito si mostrano un paio di esempi dei contratti rimossi dagli autori del paper con una breve analisi del codice eseguita da noi.

4.4.1 Analisi dei contratti non considerabili smart Ponzi

In fig. 4.6 e fig. 4.7, viene presentato il codice del contratto Crystal Dubler (indirizzo: 0x51170b18bca7896b49c52dcc18e66e5c921e100f).
In fig. 4.6 si ha la parte iniziale del contratto, contenete le variabili di utilità per la corretta esecuzione. A riga 4 si ha la struct `InvestorArray` che modella i partecipanti del contratto come coppie <indirizzo: ether investito>. A riga 10 è presente la variabile `depositor`: un array di `InvestorArray` che contiene tutti i

partecipanti al contratto. A riga 18, il costruttore imposta come `owner` il creatore del contratto. Infine, a riga 19 abbiamo la funzione di `fallback()` che fa partire la funzione `enter()` presentata in fig. 4.7. Se questo contratto fosse uno schema Ponzi, la funzione `enter()` dovrebbe pagare i partecipanti del contratto secondo uno schema chain-shaped, dunque partendo dal primo partecipante e pagando in base all'ordine di ingresso nello schema, fintanto che ha ether a disposizione. In realtà quello che accade, come fatto notare a riga 39, è che il valore della variabile `nr` non incrementa mai all'interno del ciclo che, pertanto, invia i pagamenti solo al primo partecipante ossia il creatore. Questo contratto è chiaramente una truffa, ma non rispecchia i requisiti definiti nel capitolo 3 per essere classificato come schema Ponzi, perciò è stato rimosso dal dataset.

```
1
2   contract CrystalDoubler {
3
4     struct InvestorArray  // mantiene informazioni sul partecipante
5       {
6           address EtherAddress;
7           uint Amount;
8       }
9
10    InvestorArray[] public depositors;
11    uint public Total_Players=0;
12    uint public Balance = 0;
13    uint public Total_Deposited=0;
14    uint public Total_Paid_Out=0;
15    string public Message="Welcome Player! Double your ETH Now!";
16    address public owner;
17
18   function CrystalDoubler() { //====================INIT
19      owner = msg.sender; // come da prassi andiamo a mettere come owner il creatore del contratto
20    }
21
22    function() { //====================TRIGGER
23      enter();
24    }
```

Fig. 4.6. Contratto Crystal Doubler parte 1.

```
26    function enter() { //====================ENTER
27      if (msg.value > 500 finney) { // se il msg.value supera questa cifra allora il giocatore e' ammesso a partecipare
28
29      uint Amount=msg.value;
30
31      // add a new participant to the system and calculate total players
32      Total_Players=depositors.length+1;
33      depositors.length += 1;
34      depositors[depositors.length-1].EtherAddress = msg.sender;
35      depositors[depositors.length-1].Amount = Amount;
36      Balance += Amount;                      // Balance update
37      Total_Deposited+=Amount;                // update deposited Amount
38      uint payout;
39      uint nr=0; // questa variabile si dovrebbe incerementare ad ogni iterazione nel ciclo sottostante per poter inviare
40                 // denaro ai vari partecipanti. Non aumentando pero', inviera' ETH sempre al solito utente.
41
42      while (Balance > depositors[nr].Amount * 200/100 && nr<Total_Players)
43       {
44        payout = depositors[nr].Amount *200/100;                          //calculate pay out
45        depositors[nr].EtherAddress.send(payout);                         //send pay out to participant
46        Balance -= depositors[nr].Amount *200/100;                        //balance update
47        Total_Paid_Out += depositors[nr].Amount *200/100;                 //update paid out amount
48        }
49
50    }
51  }
52  }
```

Fig. 4.7. Contratto Crystal Doubler parte 2.

Il secondo esempio è esposto in fig. 4.8 e 4.9, si tratta del contratto A Free Ether Day (indirizzo: 0x9d31ff892f984a83e8b342a5ece8e8911ed909e0).

```
1
2  contract A_Free_Ether_A_Day {
3
4      address the_stupid_guy;                    // thats me
5      uint256 public minimum_cash_proof_amount;  // to prove you are a true "Ether-ian"
6
7      function A_Free_Ether_A_Day()  { // create the contract
8              the_stupid_guy = msg.sender;
9              minimum_cash_proof_amount = 100 ether;
10
11     }
12
13     // This function allows you to claim your special bonus ether. Send any amount > minimum_cash_proof_amount
14     // to this function, and receive a special bonus ether back.
15     function show_me_the_money ()  payable  returns (uint256)  {
16
17         if ( msg.value < minimum_cash_proof_amount ) throw; // nope, you don't have the cash.. go get some ether first
18
19         uint256 received_amount = msg.value;    // remember what you have sent
20         uint256 bonus = 1 ether;                // the bonus ether
21         uint256 payout;                         // total payout back to you, calculated below
22
23         if (the_stupid_guy == msg.sender){    // doesnt work for the_stupid_guy (thats me), e' l'owner che manda soldi al contratto
24             bonus = 0;
25             received_amount = 0; // nothing for the_stupid_guy
26         }
27
28         // calculate payout/bonus and send back to sender
29         bool success;
30         payout = received_amount + bonus; // calculate total payout
31         if (payout > this.balance) throw; // nope, I dont have enough to give you a free ether, so roll back the lot
32         success = msg.sender.send(payout);
33         if (!success) throw;
34         return payout;
35     }
36
```

Fig. 4.8. Contratto A Free Ether Day parte 1.

```
37     function Good_Bye_World(){ // self-destrtuct
38         if ( msg.sender != the_stupid_guy ) throw;
39         selfdestruct(the_stupid_guy);
40
41     }
42
43     // I may increase the cash proof amount lateron, so make sure you check the global variable minimum_cash_proof_amount
44     //     ==> but don't worry, if you dont send enough, it just rolls back the transaction via a throw
45     function Update_Cash_Proof_amount(uint256 new_cash_limit){
46         if ( msg.sender != the_stupid_guy ) throw;
47         minimum_cash_proof_amount = new_cash_limit;
48     }
49
50     function () payable {}  // catch all. dont send to that or your ether is gonigone
51 }
```

Fig. 4.9. Contratto A Free Ether Day parte 2.

In fig. 4.8 si ha la parte iniziale del contratto, a riga 4 si definisce la variabile che conterrà l'indirizzo del proprietario: `the_stupid_guy` e, a linea 5, la variabile che contiene l'ammontare minimo di ether che il contratto accetta: `minimum_cash_proof_amount`. Dopodiché si crea il contratto, impostando come valore di `minimum_cash_proof_amount` 100 ETH. Da riga 15 a riga 35, si ha il cuore del contratto: la funzione `show_me_the_money()`. Questa riceve ETH dal partecipante che viene inserito nella variabile `recived_amount`. Lo scopo della funzione è rispedire la stessa cifra più un ETH bonus al chiamante, a meno

che non sia chiamata dall'owner del contratto, in tal caso il bonus è 0. Quindi, se il proprietario chiamasse questa funzione, otterrebbe solo l'ammontare che ha inviato. A questo punto la funzione calcola il payout, che sarà pari a `received_amount` più il bonus. Nel caso in cui il payout fosse maggiore del bilancio del contratto allora l'ether non viene inviato. In fig. 4.9 ci sono poi le funzioni di utilità del contratto per effettuare self-destruct (linea 37) e per cambiare valore a `minimum_cash_proof_amount`.
L'idea dietro questo contratto è subdola. Poniamo, ad esempio, che un partecipante A invii 200 ETH. Il contratto gli promette di ricevere 200 + 1 ETH indietro. Inizialmente il contratto avrà bilancio pari a 0 e, con il primo versamento, il cliente non otterrà nulla (linea 31, se il contratto non ha ether a sufficienza allora si lancia una eccezione). Di contro l'owner, che non avrà bonus, potrà reclamare tutti i 200 ETH in quanto l'ether contenuto nel contratto è sufficiente. L'unico modo per un cliente di recuperare i suoi X + 1 ether è che un altro partecipante abbia precedentemente versato dei soldi nel contratto e che l'owner non li abbia ancora reclamati. La struttura di questo contratto viola R3: ogni investitore ottiene un profitto solo se altri investitori hanno investito abbastanza soldi precedentemente nel contratto. Ciò non è vero poiché ogni qualvolta un partecipante Z depone soldi e l'owner li preleva, benché ci siano altri partecipanti che sottomettono ether dopo Z, se questi ether sono reclamati subito dal proprietario, Z non otterrà mai reward dal contratto.

4.4.2 Informazioni inserite nel dataset

Le informazioni inserite nel dataset di [4] sono le seguenti:

- Data di creazione del contratto;
- Numero di transazioni in ingresso;
- Numero di transazioni in uscita;
- Quantità totale di ether espressa in wei (1 ETH = 10^{18} wei) in ingresso;
- Quantità totale di ether espressa in wei (1 ETH = 10^{18} wei) in uscita;
- Quantità di criptovaluta media in ingresso, calcolata come il rapporto tra quantità di ether in ingresso e numero di transazioni in ingresso;
- Quantità di criptovaluta media in uscita, calcolata come il rapporto tra quantità di ether in uscita e numero di transazioni in uscita;
- Deviazione standard in ingresso ed in uscita, calcolata con la formula $\sqrt{\frac{\sum_{i=1}^{N}(x_i-\mu)^2}{N}}$, dove N è pari al numero di transazioni in ingresso (rispettivamente in uscita), μ è la media in ingresso (rispettivamente in uscita) e x_i sono i valori delle transazioni in ingresso (rispettivamente in uscita);

- Coefficiente di Gini in ingresso ed in uscita, utilizzato come indice di concentrazione per misurare la diseguaglianza nella distribuzione dei pagamenti ricevuti dal contratto e fatti dal contratto.

La prima cosa notabile dai valori presenti in questo dataset, è l'errore effettuato dagli autori nel calcolare il numero di transazioni in ingresso e in uscita. Questo errore si propaga anche nel calcolo delle altre informazioni. Facendo controllo coi dati presenti su etherscan.io, si nota che gli autori contano pure le transazioni fallite. Un esempio di errore è riportato nelle fig. 4.10, 4.11, 4.12.

	address	create_time	nbr_tx_in	nbr_tx_out	Tot_in	Tot_out	mean_in	mean_out	sdev_in	sdev_out	gini_in	gini_out	lifetime
2	0x0093cdd1ad5e2b8e8990213c9d3d832ec83e453d	2016-04-13	0	0	0	0	0	0	0	0	0	0	0
3	0x01680dc54cf0942bcabc1d6c955007e180ed4dd1	2020-02-17	8	0	1.40E+17	0	1.75E+16	0	1.39E+16	0	0.410714	0	9241
4	0x017d4abee6a74ac4f0ba5a6fd95527827e69fec2	2018-10-28	0	0	0	0	0	0	0	0	0	0	0
5	0xa38feb7b0aaad949a1b99ff516d4e3ea2e317bbf	2018-10-28	59	0	2.66E+19	0	4.51E+17	0	1.25E+17	0	0.094823	0	12387

Fig. 4.10. La linea evidenziata contiene le informazioni recuperate da [4] per l'indirizzo 0x01680dc54cf0942bcabc1d6c955007e180ed4dd1.

Txn Hash	Method	Block	Age	From		To	Value	Txn Fee
0xf71df103cbfe2a7dadc…	Transfer	9495956	690 days 14 hrs ago	0xd9cc93af3954f352cda…	IN	0x01680dc54cf0942bcab…	0.01 Ether	0.000378
0x8f717248e68759d409…	Transfer	9495946	690 days 14 hrs ago	0x2206c2bb68b7ee9521…	IN	0x01680dc54cf0942bcab…	0.01 Ether	0.000252
0xe9b71a7b6528d18845…	Transfer	9495929	690 days 14 hrs ago	0x2206c2bb68b7ee9521…	IN	0x01680dc54cf0942bcab…	0.01 Ether	0.0003696
0x37678289e34ceeaefa…	Transfer	9495895	690 days 15 hrs ago	0xbc761cc84fdaef965ca…	IN	0x01680dc54cf0942bcab…	0.04 Ether	0.00824179125
0xaedf3baf90f0af78feaf6…	Transfer	9495895	690 days 15 hrs ago	0xbc761cc84fdaef965ca…	IN	0x01680dc54cf0942bcab…	0.04 Ether	0.00500044
0x8b8a95bd99618f3bd5…	Transfer	9495847	690 days 15 hrs ago	0x1a1f736e3e156a7c07…	IN	0x01680dc54cf0942bcab…	0.02 Ether	0.00508547
0x94d916ad92eb27c7d9…	Transfer	9495697	690 days 15 hrs ago	0x1a1f736e3e156a7c07…	IN	0x01680dc54cf0942bcab…	0.01 Ether	0.00458402
0xdc2a1801ba6466b512…	0x60806040	9495253	690 days 17 hrs ago	0x4f65a0e0abb5dfe4cef…	IN	Contract Creation	0 Ether	0.005368584

Fig. 4.11. Rappresenta le transazioni in ingresso nel contratto 0x01680dc54cf0942bcabc1d6c955007e180ed4dd1.

Parent Txn Hash	Block	Age	From		To	Value
0x37678289e34ceeaefa…	9495895	690 days 15 hrs ago	0x01680dc54cf0942bcab…	→	0xbc761cc84fdaef965ca…	0.012 Ether
0x37678289e34ceeaefa…	9495895	690 days 15 hrs ago	0x01680dc54cf0942bcab…	→	0xbc761cc84fdaef965ca…	0.012 Ether
0x37678289e34ceeaefa…	9495895	690 days 15 hrs ago	0x01680dc54cf0942bcab…	→	0xbc761cc84fdaef965ca…	0.012 Ether
0x37678289e34ceeaefa…	9495895	690 days 15 hrs ago	0x01680dc54cf0942bcab…	→	0x4f65a0e0abb5dfe4cef…	0.006 Ether
0xaedf3baf90f0af78feaf6…	9495895	690 days 15 hrs ago	0x01680dc54cf0942bcab…	→	0xbc761cc84fdaef965ca…	0.012 Ether
0xaedf3baf90f0af78feaf6…	9495895	690 days 15 hrs ago	0x01680dc54cf0942bcab…	→	0x1a1f736e3e156a7c07…	0.012 Ether
0xaedf3baf90f0af78feaf6…	9495895	690 days 15 hrs ago	0x01680dc54cf0942bcab…	→	0x4f65a0e0abb5dfe4cef…	0.006 Ether
0x8b8a95bd99618f3bd5…	9495847	690 days 15 hrs ago	0x01680dc54cf0942bcab…	→	0x1a1f736e3e156a7c07…	0.012 Ether
0x8b8a95bd99618f3bd5…	9495847	690 days 15 hrs ago	0x01680dc54cf0942bcab…	→	0x1a1f736e3e156a7c07…	0.012 Ether
0x8b8a95bd99618f3bd5…	9495847	690 days 15 hrs ago	0x01680dc54cf0942bcab…	→	0x4f65a0e0abb5dfe4cef…	0.003 Ether
0x94d916ad92eb27c7d9…	9495697	690 days 15 hrs ago	0x01680dc54cf0942bcab…	→	0x4f65a0e0abb5dfe4cef…	0.0015 Ether

Fig. 4.12. Rappresenta le transazioni interne nel contratto 0x01680dc54cf0942bcabc1d6c955007e180ed4dd1.

Osserviamo il contratto 0x01680dc54cf0942bcabc1d6c955007e180ed4dd1. In fig. 4.10, nella linea evidenziata, sono riportati i valori calcolati dagli autori. Come si può osservare abbiamo 8 transazioni in ingresso (nbr_tx_in) e 0 in uscita (nbr_tx_out). Andando a controllare il contratto su etherscan.io, otteniamo dei valori diversi. La fig. 4.11 considera le transazioni in ingresso nel contratto. Delle 8 effettuate, solo 5 sono andate a buon fine, mentre 3 sono fallite. Le transazioni fallite non apportano un cambiamento allo stato del contratto e pertanto non devono essere considerate nel conteggio. La fig. 4.12 considera le transazioni interne, in questo caso tutte in uscita dal contratto sotto esame. Queste (come è intuibile dalla freccia verde) non sono fallite, ma gli autori non le hanno considerate; infatti, il valore del campo nbr_tx_out è 0. Questi errori si ripercuotono anche sugli altri valori calcolati dato che si basano tutti sul numero di transazioni in ingresso o in uscita. Nello script implementato questi problemi sono stati risolti, verificando se una transazione fosse fallita o meno.
Infine, anche la parte di questo dataset riguardante gli schemi Ponzi, ha necessitato di una pulizia: essendo costruito a partire dal dataset di [2], molti contratti presenti sono doppioni, nel senso che identificano lo stesso contratto (stesso indirizzo), pertanto li ho rimossi effettuando un controllo a mano.

Al termine di questi vari processi di epurazione, dall'unione dei tre dataset descritti fino ad ora, ho ottenuto un insieme di dati contenente circa un migliaio smart Ponzi.

4.5 Features selezionate

Una volta creato il dataset è stato necessario uniformarlo. Fino ad ora avevamo una raccolta di contratti con informazioni eterogenee: contratti provenienti da dataset diversi con features diverse. Per standardizzarlo è stato realizzato uno script in Python che raccoglie per tutti i contratti le stesse informazioni.
Di seguito la lista di features calcolate con lo script e inserite nel dataset:

1. *Balance*: contiene il bilancio del contratto, ottenuto sottraendo l'ETH in ingresso nel contratto con l'ETH in uscita;
2. *Lifetime*: tempo di vita del contratto, calcolato come la differenza tra la prima transazione (data di creazione del contratto) e l'ultima transazione fatta o ricevuta dal contratto;
3. *Tx_in*: numero di transazioni in ingresso nel contratto;
4. *Tx_out*: numero di transazioni in uscita dal contratto;
5. *Investment_in*: numero di transazioni che hanno versato ETH nel contratto;
6. *Payment_out*: numero di transazioni fatte dal contratto che contengono ETH;

7 *Investment_in/Tx_in*: rapporto tra il numero di transazioni che hanno pagato il contratto (investment_in) e il numero di transazioni in ingresso nel contratto (tx_in);
8 *Payment_out/Tx_out*: rapporto tra il numero di transazioni fatte dal contratto che contengono ETH (payment_out) e il numero di transazioni in uscita dal contratto (tx_out);
9 *#addresses_paying_contract*: numero di indirizzi diversi che hanno pagato il contratto;
10 *#addresses_paid_by_contract*: numero di indirizzi diversi pagati dal contratto;
11 *Mean_v1*: media della differenza tra il dizionario che contiene le coppie <indirizzo, #tx_in> e il dizionario che contiene le coppie <indirizzo, #tx_out>. L'i-esima entry del vettore V1 è $V1[i] = n_i - m_i$, dove n_i è il numero di transazioni fatte dal contratto verso l'i-esimo indirizzo e m_i è il numero di transazioni fatte dall'i-esimo indirizzo al contratto;
12 *Mean_v2*: media della differenza tra il dizionario che contiene le coppie <indirizzo, #amount_in> e il dizionario che contiene le coppie <indirizzo, #amount_out>. L'i-esima entry del vettore è $V2[i] = p_i - q_i$ dove p_i è il valore in ETH delle transazioni che l'i-esimo indirizzo ha ricevuto dal contratto e q_i è il valore delle transazioni fatte dall'i-esimo indirizzo al contratto, cioè quello che ha versato nel contratto;
13 *Sdev_v1*: Deviazione standard del vettore differenza V1;
14 *Sdev_v2*: Deviazione standard del vettore differenza V2;
15 *Paid_rate*: rapporto tra (#tx_out)/(#tx_in);
16 *Paid_one*: rapporto fra il numero di investitori che sono stati ripagati e il numero di investitori totali (con investitori si intendono i soli indirizzi che hanno versato ETH nel contratto: se, ad esempio, il creatore del contratto non investe ma ottiene ETH, non sarà contato come investitore);
17 *Percentage_some_tx_in*: numero reale tra 0 ed 1 che esprime i giorni di attività di un contratto (in cui c'è stata almeno una transazione in ingresso, non necessariamente contenente ETH) rispetto alla sua lifetime;
18 *Sdev_tx_in*: deviazione standard delle transazioni in ingresso, ci dice se ci sono stati dei giorni con picchi di transazioni rispetto ad altri. Più è alto questo valore e più è eterogeneo il numero delle transazioni in ingresso;
19 *Percentage_some_tx_out*: numero reale tra 0 ed 1 che esprime i giorni di attività di un contratto (in cui c'è stata almeno una transazione in uscita, non necessariamente contenente ETH) rispetto alla sua lifetime;
20 *Sdev_tx_out*: deviazione standard delle transazioni in uscita, ci dice se ci sono stati dei giorni con picchi di transazioni rispetto ad altri. Più è alto questo valore e più è eterogeneo il numero delle transazioni in uscita;
21 *Creator_get_eth_wo_investing*: vale 1 se il creatore ha ottenuto ETH senza investire (inviare soldi al contratto), 0 altrimenti;

22 *Creator_get_eth_investing*: vale 1 se il creatore ha ottenuto ETH investendo, 0 altrimenti;
23 *Creatore_get_no_eth*: vale1 se il creatore del contratto non ha ottenuto ETH, 0 altrimenti.

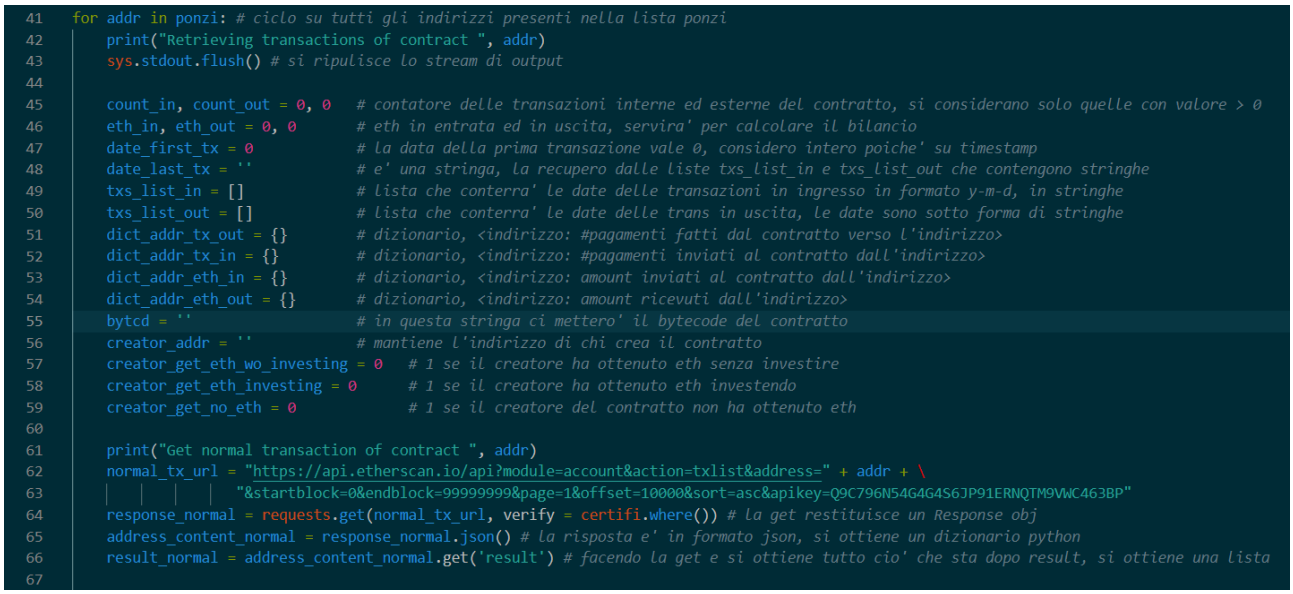
Circa il 60% di queste features proviene dagli articoli analizzati: [1], [2], [3] e [4]. Le features aggiunte da noi aggiunte sono la settima, l'ottava e le ultime sette. La 7 e la 8 sono state inserite con l'idea di cogliere la percentuale delle transazioni che coinvolgono l'invio di ETH fra indirizzi Ethereum. Quello che ci si aspetta è che i contratti non Ponzi abbiano percentuali più basse rispetto ai contratti Ponzi, dato che quest'ultimi si basano unicamente sull'invio di ether.
Le features 17, 18, 19, 20 sono state introdotte per osservare se ci sono stati picchi di transazioni ricevute o effettuate da un contratto. Un valore basso della feature 17 (rispettivamente della 19) indica che durante la vita del contratto ci sono stati pochi giorni in cui sono avvenute delle transazioni in ingresso (rispettivamente in uscita). Viceversa, un valore alto di questa features si traduce nel dire che il contratto durante la sua vita è stato contattato in maniera regolare. Se avessimo inserito solo questa informazione nel dataset, non avremmo catturato lo sbilanciamento che si può avere nel numero di transazioni; che un contratto in un giorno X abbia ricevuto 1 transazione o 1000, ai fini del valore di queste features non cambia nulla. Pertanto, è stato introdotto anche lo scarto quadratico medio. La feature 18 (rispettivamente la 20) calcola la deviazione standard delle transazioni in ingresso (rispettivamente in uscita). Un valore alto di questa features indica che le transazioni in ingresso (rispettivamente in uscita) non sono regolari, ossia che ci sono giorni con un numero superiore alla media di transazioni e giorni in cui ce ne sono poche o nessuna. Poniamo, ad esempio, che un contratto abbia una lifetime di 4 giorni, e che le transazioni in ingresso ricevute in questi giorni sono 54, 3, 8, 1. Calcolando la deviazione standard otterremmo 23. Per contro, se avesse ricevuto delle transazioni in ingresso più bilanciate ad esempio 15, 16, 14, 13, avremmo avuto uno scarto quadratico medio di 1.118.
Le ultime 3 features sono state inserite per verificare se il creatore ha ricevuto dell'ether dal contratto. Tra i contratti Ponzi capita che il proprietario riesca ad ottenere ETH anche senza investire. Ciò che ci si aspetta è che la maggior parte dei contratti Ponzi abbiano le features 21 o 22 pari a 1 e che difficilmente per un Ponzi il creatore non riceva dell'ether, cioè che la features 23 valga 1.

4.6 Script Python per Etherscan

Per calcolare le features descritte nella sezione precedente è stato realizzato uno script Python che sfrutta le API di etherscan[6] per recuperare le informazioni dei contratti.
Lo script per recuperare le informazioni degli smart Ponzi è pubblicato su GitHub: https://github.com/Lychett/tesi-smart-ponzi/blob/main/smart_ponzi.
Nelle altre due cartelle della pagina GitHub (smart_contract_non_ponzi_p1 e smart_contract_non_ponzi_p2) si troveranno script affini per calcolare le informazioni dei contratti non Ponzi. I principali cambiamenti tra questi risiedono nei nomi delle variabili, modificati per rendere più chiaro cosa analizza lo script.
Di seguito si fornisce un'illustrazione del codice utilizzato per raccogliere le informazioni sugli smart Ponzi, concentrandosi sui punti più significativi.

Nelle prime righe di codice, dalla 1 alla 25, si importano le librerie necessarie per la corretta esecuzione dello script e si definiscono le funzioni che calcolano media, varianza e deviazione standard.
Nelle linee successive, dalla 27 alla 39, si acquisisce la lista di indirizzi da analizzare dal file ponzi_addresses.csv (file anch'esso presente nella cartella GitHub), leggendo gli indirizzi riga per riga (linee 35-39) e inserendoli nella lista `ponzi`, infine si aprono due file in scrittura: bytecodes.txt e ponzi_result.csv associati rispettivamente alle variabili `bytecodes_result` e `ponzi_result` e nella prima riga del file ponzi_result.csv si scrivono i nomi delle features.
Dalla linea 41 in poi inizia il ciclo principale dello script, che sarà eseguito per ogni contratto presente nella lista `ponzi` (fig. 4.13).
Dalla linea 45 alla linea 59 inizializzo le variabili che saranno utilizzate per calcolare le informazioni.
Alle righe 62-66 preparo ed invio la richiesta al server di etherscan per ottenere la lista delle normal transactions, cioè la lista delle transazioni ricevute dal contratto esaminato.
L'URL a linea 62 ci permette di specificare:
• l'indirizzo (campo address) del contratto sotto osservazione;
• lo startblock e l'endblock, due valori interi che indicano rispettivamente il numero del blocco da cui iniziare a cercare transazioni e il numero di blocco oltre il quale smettere;
• sort: indica se avere un ordinamento crescente o decrescente delle transazioni rispetto alla data, con asc si ottiene un ordinamento dalla data più vecchia a quella più recente, viceversa per desc.

[6] https://docs.etherscan.io/



```
41  for addr in ponzi: # ciclo su tutti gli indirizzi presenti nella lista ponzi
42      print("Retrieving transactions of contract ", addr)
43      sys.stdout.flush() # si ripulisce lo stream di output
44
45      count_in, count_out = 0, 0   # contatore delle transazioni interne ed esterne del contratto, si considerano solo quelle con valore > 0
46      eth_in, eth_out = 0, 0       # eth in entrata ed in uscita, servira' per calcolare il bilancio
47      date_first_tx = 0            # la data della prima transazione vale 0, considero intero poiche' su timestamp
48      date_last_tx = ''            # e' una stringa, la recupero dalle liste txs_list_in e txs_list_out che contengono stringhe
49      txs_list_in = []             # lista che conterra' le date delle transazioni in ingresso in formato y-m-d, in stringhe
50      txs_list_out = []            # lista che conterra' le date delle trans in uscita, le date sono sotto forma di stringhe
51      dict_addr_tx_out = {}        # dizionario, <indirizzo: #pagamenti fatti dal contratto verso l'indirizzo>
52      dict_addr_tx_in = {}         # dizionario, <indirizzo: #pagamenti inviati al contratto dall'indirizzo>
53      dict_addr_eth_in = {}        # dizionario, <indirizzo: amount inviati al contratto dall'indirizzo>
54      dict_addr_eth_out = {}       # dizionario, <indirizzo: amount ricevuti dall'indirizzo>
55      bytcd = ''                   # in questa stringa ci mettero' il bytecode del contratto
56      creator_addr = ''            # mantiene l'indirizzo di chi crea il contratto
57      creator_get_eth_wo_investing = 0   # 1 se il creatore ha ottenuto eth senza investire
58      creator_get_eth_investing = 0      # 1 se il creatore ha ottenuto eth investendo
59      creator_get_no_eth = 0             # 1 se il creatore del contratto non ha ottenuto eth
60
61      print("Get normal transaction of contract ", addr)
62      normal_tx_url = "https://api.etherscan.io/api?module=account&action=txlist&address=" + addr + \
63                      "&startblock=0&endblock=99999999&page=1&offset=10000&sort=asc&apikey=Q9C796N54G4G4S6JP91ERNQTM9VWC463BP"
64      response_normal = requests.get(normal_tx_url, verify = certifi.where()) # la get restituisce un Response obj
65      address_content_normal = response_normal.json() # la risposta e' in formato json, si ottiene un dizionario python
66      result_normal = address_content_normal.get('result') # facendo la get e si ottiene tutto cio' che sta dopo result, si ottiene una lista
67
```

Fig. 4.13. Inizio del ciclo principale dello script. Il ciclo viene eseguito per ogni contratto presente nella lista 'ponzi'. Il ciclo inizia alla linea 41 e termina a linea 194.

Se si vogliono inviare più richieste al secondo, sarà necessario specificare pure una API key, altrimenti sarà imposto un limite temporale al numero di richieste inviabili. La API key deve essere generata sul sito di etherscan dopo essersi registrati. La versione gratuita permette di effettuare fino a cinque API calls al secondo. Alla riga successiva si invia la richiesta get stabilendo una connessione https con il server. Essendo una connessione criptata sarà necessario un certificato e, per fornirglielo, si usa il modulo Python `certifi`. A seguito della richiesta get ottengo un oggetto Response. Per comodità si converte l'oggetto Response in JSON tenendo solo la parte di informazioni sotto la voce 'result'.
In fig. 4.14 si presenta il codice contenente il ciclo per il calcolo delle informazioni riguardanti le normal transactions.
A linea 68 inizia il ciclo sulle normal transactions. Il primo controllo effettuato (linea 69) è la verifica che la transazione non sia fallita, cioè che il campo 'isError' della transazione valga 0, altrimenti non la terrò in considerazione. Se la transazione è andata a buon fine e se si tratta della prima transazione del contratto (la variabile `date_first_tx` vale ancora 0 – linea 70), assegno il timestamp (data della transazione) a tale variabile e recupero anche il creatore del contratto. Successivamente ottengo il bytecode del contratto, inviato assieme alla prima transazione e lo vado a stampare nel file bytecode_result.csv.

```
68  for t in result_normal: # transazioni in ingresso nel contratto
69      if (t['isError'] == '0'): # assenza di errori nella transazione
70          if (date_first_tx == 0):
71              date_first_tx = int(t['timeStamp']) # assegno la prima che trovo
72              creator_addr = t['from'] # assegno anche il creatore del contratto
73          if not bytcd: # disponendo le transazione in ordine asc, la prima che si incontra manterra' il bytecode dello smart contract
74              bytcd = t['input']
75              to_print_in_file = bytcd[2:]
76              to_print_in_file += "\n" # inserico il new line altrimenti non va accapo
77              bytecodes_result.write(to_print_in_file)
78          # si trasforma il timestamp in una stringa e lo inserisco in txs_list_in
79          obj_datetime = datetime.fromtimestamp(int(t['timeStamp']))
80          txs_list_in.append(obj_datetime.strftime('%Y-%m-%d')) # inserisco in coda la data
81          eth_val = int(t['value'])
82          if (eth_val > 0): # se il valore della transazione e' > 0 allora...
83              if t['from'] in dict_addr_tx_in: # se l'indirizzo e' gia' presente, cioe' il mittente della transazione aveva gia' pagato il contratto
84                  dict_addr_tx_in[t['from']] += 1 # incremento il valore associato alla chiave, che indica il numero di transazioni
85                  dict_addr_eth_in[t['from']] += round(Decimal(eth_val)/Decimal('1000000000000000000'), 6)
86              else:
87                  dict_addr_tx_in[t['from']] = 1 # se non era ancora presente allora e' la prima volta che invia soldi al contratto
88                  dict_addr_eth_in[t['from']] = round(Decimal(eth_val)/Decimal('1000000000000000000'), 6)
89              eth_in += eth_val # incremento la variabile che contiene l'ETH ricevuto
90              count_in += 1
```

Fig. 4.14 Ciclo for per il calcolo delle informazioni relative alle normal transactions.

Nelle righe 79 e 80 si trasforma il timestamp relativo alla data della transazione, in una stringa formato 'Y-m-d' che è inserita nella lista `txs_list_in`.
Nelle linee successive si ottiene il valore in wei inviato con la transazione. Se il valore è positivo allora si modifica il valore associato alla chiave 'indirizzo mittente' del dizionario `dict_addr_tx_in` e il valore (questa volta espresso in ETH) che il mittente ha inviato al contratto (il valore associato all'indirizzo mittente del dizionario `dict_addr_eth_in`). Infine, si incrementa il numero di `count_in`: il numero di transazioni in ingresso contenete wei e la variabile `eth_in`: il valore totale espresso in wei che è entrato nel contratto.
Una volta analizzate le normal transactions, vado ad esaminare le transazioni interne. Il codice è riportato in fig. 4.15, e risulta essere molto simile a quello già presentato in fig. 4.14.
Tra riga 92 e 97 si invia una richiesta al server di etherscan e si ottiene una risposta che sarà poi convertita in formato JSON. Successivamente da riga 99 a 124 si svolge il ciclo sulle transazioni interne. A differenza di quanto fatto in fig. 4.14, dove l'ETH poteva solo entrare nel contratto, adesso si deve verificare se il flusso di ether sia entrante o uscente. La divisione in questo secondo caso si rivela necessaria in quanto a differenza delle normal transactions, contenenti le sole transazioni in ingresso nel contratto e generate quando un EOA invia ether al contratto; le internal transactions si hanno quando un intermediario nel flusso di ether è uno smart contract [6] e possono dunque essere dirette al contratto analizzato o uscire da esso. Dato che nelle features si distinguono le transazioni entranti da quelle uscenti, è stato necessario effettuare un controllo.
Se l'indirizzo del contratto è pari al campo t['from'] allora l'ETH sarà uscente dal contratto, altrimenti vuol dire che l'ether è entrante. In base al flusso dell'ether si aggiungono/modificano le informazioni nelle strutture dati riguardanti le transazioni in uscita dal contratto o quelle riguardanti le transazioni in ingresso.
Finito questo secondo ciclo interno comincia la terza ed ultima fase: il calcolo delle features utilizzando i dati raccolti.
Dalla linea da 126 alla 148 (fig. 4.15) si inizia il calcolo della parte relativa alle

date, necessarie per calcolare features come la lifetime del contratto e le percentuali di transazioni in ingresso ed in uscita, inoltre, si ottengono anche le features riguardanti il numero di transazioni in ingresso e in uscita dal contratto (mentre `count_in` e `count_out` contano le transazioni che contengono ETH, `tx_in` e `tx_out` considerano tutte le transazioni, linee 129-130). Infine, da riga 150 a 191, si effettua il calcolo delle restanti features. A riga 193 si scrivono i risultati ottenuti sul file ponzi_result.csv e a riga 194 per evitare errori, avendo ho un limite di 5 API calls al secondo, si mette il thread dello script in pausa per 0.2 secondi.

```
92  print("Get internal transaction of contract ", addr)
93  internal_tx_url = "https://api.etherscan.io/api?module=account&action=txlistinternal&address=" + addr + \
94                    "&startblock=0&endblock=99999999&page=1&offset=10000&sort=asc&apikey=Q9C796N54G4G4S6JP91ERNQTM9VWC463BP"
95  response_internal = requests.get(internal_tx_url, verify = certifi.where())
96  address_content_internal = response_internal.json()
97  result_internal = address_content_internal.get("result")
98
99  for t in result_internal:
100     if (t['isError'] != '0'): continue # siamo in presenza di errore
101     obj_datetime = datetime.fromtimestamp(int(t['timeStamp']))
102     eth_val = int(t['value'])
103     if (t['from'].lower() == addr.lower()):  # internal transaction from contract to t['to'] aka addr, flusso di ETH uscente
104         txs_list_out.append(obj_datetime.strftime('%Y-%m-%d')) # inserisco in coda la data
105         if (eth_val > 0):
106             if t['to'] in dict_addr_tx_out:
107                 dict_addr_tx_out[t['to']] += 1
108                 dict_addr_eth_out[t['to']] += round(Decimal(eth_val)/Decimal('1000000000000000000'), 6)
109             else:
110                 dict_addr_tx_out[t['to']] = 1 # se non era ancora presente allora e' il primo pagamento che ha ricevuto
111                 dict_addr_eth_out[t['to']] = round(Decimal(eth_val)/Decimal('1000000000000000000'), 6)
112             eth_out += eth_val
113             count_out +=1
114     else:  # internal transaction from t['from'] to contract, flusso di ETH entrante
115         txs_list_in.append(obj_datetime.strftime('%Y-%m-%d')) # inserisco in coda la data
116         if (eth_val > 0): # se il valore della transazione e' >0 allora...
117             if t['from'] in dict_addr_tx_in: # se l'indirizzo e' gia' presente, cioe' il mittente della transazione aveva gia' pagato il contratto
118                 dict_addr_tx_in[t['from']] += 1 # incremento il valore associato alla chiave, che indica il numero di transazioni
119                 dict_addr_eth_in[t['from']] += round(Decimal(eth_val)/Decimal('1000000000000000000'), 6)
120             else:
121                 dict_addr_tx_in[t['from']] = 1 # se non era ancora presente allora e' la prima volta che invia soldi al contratto
122                 dict_addr_eth_in[t['from']] = round(Decimal(eth_val)/Decimal('1000000000000000000'), 6)
123             eth_in += eth_val # incremento la variabile che contiene l'ETH ricevuto
124             count_in += 1
125
```

Fig. 4.15 richiesta get e ciclo sulle transazioni interne del contratto analizzato.

4.7 Costruzione del dataset con cui fare training

Una volta calcolate le features e ottenuto il bytecode sia per gli smart Ponzi che per i contratti non Ponzi provenienti dai dataset di [2], [3] e [4], si ha un insieme di dati uniforme. L'ultimo passo consiste nel rimuovere eventuali contratti doppioni tra i Ponzi. Per fare ciò è stato implementato lo script mostrato in fig. 4.16.

```python
import sys
import csv
import hashlib

ponzi_addr = "ponzi_addresses.csv"  # indirizzi smart ponzi
bytecodes_result = "bytecodes.csv"  # bytecode smart ponzi
ponzi_addr_hash = {} # dizionario con chiavi gli indirizzi e valori hash(bytecode) cioe' l'hash calcolato sul bytecode
ponzi_hash_bytc = []
print("Get addresses")
with open(ponzi_addr, 'r', encoding = 'utf-8-sig') as fi:  # fi: file indirizzi
    reader1 = csv.reader(fi, delimiter = ',', quotechar = '|') # reader: oggetto che itera le linee del file csv
    for row in reader1:
        addr = row[0] # si legge per righe, il primo elemento di ogni riga viene inserito in addr
        ponzi_addr_hash[addr] = None

print("Get bytecode")
with open(bytecodes_result, 'r', encoding = 'utf-8-sig') as fb: # fb: file bytecode
    reader2 = csv.reader(fb, delimiter = ',', quotechar = '|') # reader: oggetto che itera le linee del file csv
    for row in reader2:
        bytc = row[0] # si legge per righe, il primo elemento di ogni riga viene inserito in addr
        hash_bytc = hashlib.sha256(bytc.encode('utf-8')).hexdigest()
        ponzi_hash_bytc.append(hash_bytc) # metto hash(bytecode) nella lista

# chiavi del dizionario: ponzi_addr_hash, valori del dizionario: ponzi_hash_bytc
dictionary = dict(zip(ponzi_addr_hash, ponzi_hash_bytc))
flipped = {} # finding duplicate values from dictionary using flip

for key, value in dictionary.items():
    if value not in flipped:
        flipped[value] = [key]
    else:
        flipped[value].append(key)

for key, value in flipped.items():
    print(key, ':', value)
```

Fig. 4.16. Script Python per l'identificazione di contratti Ponzi con stesso bytecode.

Dopo aver importato le librerie necessarie, sono state istanziate due variabili per leggere gli indirizzi e il bytecode degli smart Ponzi da file.
Nelle linee dalla 10 alla 14 si leggono gli indirizzi degli smart Ponzi individuati e si inseriscono come chiavi del dizionario `ponzi_addr_hash`.
Nelle linee dalla 17 alla 22 si ottengono i bytecodes dei contratti analizzati e se ne calcola l'hash, in modo da ridurre la dimensione degli oggetti che da confrontare (i bytecodes appunto), si inserisce poi l'hash nella lista `ponzi_hash_bytc`. Successivamente si crea un dizionario avente come chiavi gli indirizzi e come valore l'hash del bytecode. Infine, per individuare i duplicati si esegue un 'flip' in modo da rendere le chiavi i valori e i valori le chiavi ottenendo così l'hash come chiave e gli indirizzi come valore. Indirizzi con lo stesso hash sono contratti doppioni. Al termine dell'esecuzione di questo script ho trovato poco più di un centinaio di contratti duplicati. Rimuovendoli siamo passati a circa 900 smart Ponzi presenti nel datset. Lo script e i risultati sono pubblicati su GitHub all'indirizzo: https://github.com/Lychett/tesi-smart-ponzi/tree/main/analisi%20bytecodes%20duplicati%20negli%20smart%20Ponzi.

5. Creazione di un classificatore binario per il rilevamento di smart Ponzi sulla blockchain di Ethereum

Questo capitolo descrive la creazione di un classificatore binario per il riconoscimento dei contratti Ponzi. Si parte da una descrizione degli algoritmi di classificazione utilizzati: Decision Tree, Random Forest e XGBoost e dei loro principali iperparametri definiti nell'implementazione di Scikit-learn. Successivamente si presentano le tecniche adottate per lo sviluppo del modello e il relativo codice. Dopodiché si descrive la fase di sperimentazione e valutazione dei modelli creati, concludendo con la scelta del migliore tra questi. Infine, sul modello scelto, si analizzano i falsi positivi ossia contratti non Ponzi che vengono classificati come Ponzi e si esegue un'analisi statistica per provare che il modello costruito risulti effettivamente migliore rispetto a quello presentato in [3].

5.1 Descrizione dei classificatori

Di seguito si effettua una rassegna degli algoritmi di classificazione adottati per la creazione del modello, descrivendo anche gli iperparametri selezionati per ognuno di essi e di cui si è eseguito il tuning in fase di training.

5.1.1 Decison Tree

L'albero decisionale o decision tree è un algoritmo di tipo supervisionato. Si tratta di un modello predittivo che può essere usato sia per **classificazione** che per **regressione**. L'output dell'albero decisionale sarà dunque o una variabile categorica (per esempio una binaria: 0/1) o una quantità continua, come il prezzo di una casa. Un esempio di albero decisionale è riportato in fig. 5.1.
Due concetti chiave per capire il funzionamento dei decision tree sono i nodi e le foglie. I **nodi** o **nodi di decisione** contengono una condizione per dividere i dati. Le **foglie**, invece, permettono di decidere la classe di un nuovo oggetto che vogliamo classificare.

Un albero decisionale è a tutti gli effetti un albero binario che ricorsivamente divide il dataset finché non si ottengono delle foglie pure, cioè contenenti dati che appartengono alla stessa categoria. In caso di dati complessi non è detto però che si riescano ad ottenere foglie pure. Per risolvere questo problema, in caso di classificazione ci si basa sul voto a maggioranza; si decide cioè la categoria del nuovo punto in base alla classe in maggioranza nella foglia dove esso è stato classificato. L'obiettivo del modello è quello di effettuare i migliori split possibili per i dati, scegliendo le migliori features e le relative soglie per i loro valori, in modo tale da generare gruppi di dati il più omogenei possibili. Ci sono varie metriche adottabili per calcolare questi split che saranno successivamente spiegate.

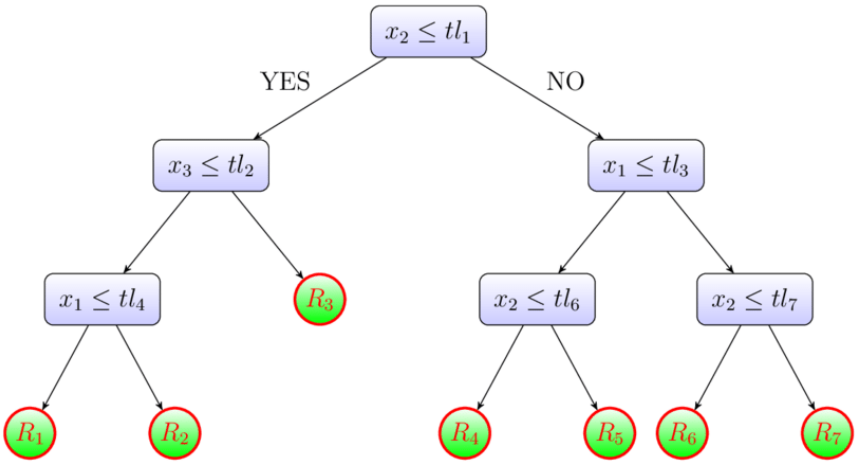


Fig. 5.1. Struttura tipica assunta da un albero decisionale.

5.1.1.1 Principali parametri ed iperparametri

Tra i parametri da tenere in considerazione c'è la metrica di splitting, che indica come saranno separati i dati. Questo parametro si chiama **criterion** e dipende dal task da svolgere. Per la classificazione può avere come valore associato "gini" o "entropy". "gini" fa riferimento all'impurità di Gini, mentre "entropy" all'information gain, basato sul concetto di entropia nella teoria dell'informazione. Per la regressione, invece, può essere usato: "mse", "friedman_mse", "mae", e "poisson", il valore tipicamente utilizzato è "*mse*", ovvero il mean squared error (l'errore quadratico medio).
Un altro iperparametro da gestire è **max_depth**. Questo indica la profondità massima che potrà avere l'albero di decisione. Più l'albero sarà profondo più tenderà ad essere preciso sui dati di training. Essere più preciso sui dati di training non implica che il modello sarà in grado di generalizzare bene. Tipicamente, infatti, più un decision tree è profondo meno probabilmente sarà in grado di generalizzare su dati nuovi. Una maggiore profondità rischia, dunque, di portare al

fenomeno dell'overfitting, ovvero un ottimo comportamento sul training set ma pessimo sul test set, che si traduce in una incapacità di generalizzare su nuovi dati (ciò che interessa).
Un altro parametro è **min_samples_split**, ovvero il numero minimo di campioni necessari per effettuare lo split di un nodo di decisione. Un valore basso associato a questo iperparametro si traduce nell'avere una maggior precisione sul training set.
Altro iperparametro importante è **max_features**: questo parametro ci dà la possibilità di scegliere il numero di features da utilizzare per trovare il migliore split. Di default è inizializzato a "None", quindi verrà cercato su tutte le features. Altri valori tipici sono "sqrt" che terrà in considerazione la radice quadrate della quantità di features e "log2" che ne userà il logaritmo in base due.
L'ultimo iperparametro considerato è **class_weight**. Questo è utile da definire in caso di dataset sbilanciati. Questo valore è implementato come un dizionario Python che definisce per ogni categoria un peso da applicare nel calcolo della purezza degli split durante il fitting dell'albero decisionale. Se non venisse definito, tutte le classi (sia quella in maggioranza che quella in minoranza) avrebbero stesso peso.

5.1.1.2 Pro e contro dei Decison Tree

PRO
• Molto semplice da capire e interpretare;
• Può lavorare su dati numerici e categorici contemporaneamente;
• Non richiede moltissimi dati;
• Riesce a performare bene sia su pochi dati sia su grandi dataset.

CONTRO
• Non è un modello robusto: un piccolo cambiamento nel training può comportare un grande cambiamento nella predizione;
• Rischio di overfitting;
• È un algoritmo greedy che seleziona la divisione che massimizza la metrica definita in criterion, non effettua il backtracking per cambiare lo split precedente. Pertanto, tutti gli split successivi dipenderanno da quello corrente.

5.1.2 Random Forest

Random forest è un algoritmo di machine learning supervisionato, costruito a partire da una collezione di alberi decisionali. Può essere utilizzato sia per risolvere problemi di classificazione che di regressione.
Questo algoritmo risulta essere più accurato dei decison tree e, generalmente,

tende a produrre previsioni più ragionevoli anche senza regolazione degli iperparametri.
Per poter utilizzare random forest, si devono generare dei nuovi sotto-dataset a partire dal dataset originale. Per crearli si scelgono dei campioni in modo randomico dall'insieme iniziale di dati. Ogni nuovo sotto-dataset avrà la stessa grandezza dell'originale. Questo processo è detto bootstrapping, ed è mostrato in fig. 5.2. Una volta costruiti questi nuovi insiemi di dati si allenano i decision tree indipendentemente, ognuno su un nuovo dataset. Inoltre, non si usano tutte le features per allenare i singoli decision tree ma se ne selezionano un sottoinsieme in maniera casuale.

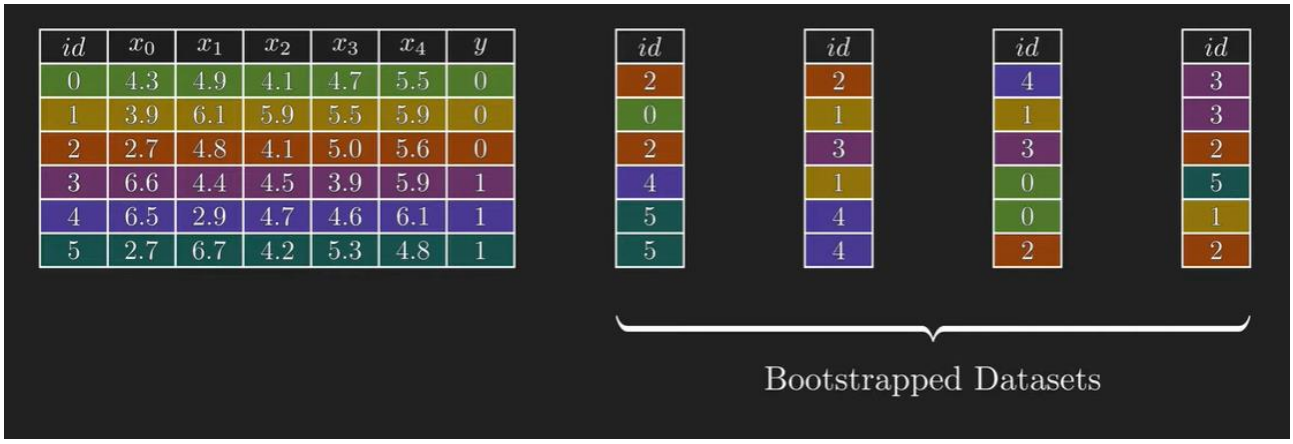


Fig. 5.2 A sinistra abbiamo il dataset originale, contenete 5 features e 6 campioni, a destra invece abbiamo i dataset ottenuti tramite bootstrap, abbiamo ancora 6 campioni ma alcuni risultano duplicati.

Mentre nei decision tree per effettuare una predizione si dava in input il sample all'unico albero presente, adesso il dato da classificare viene fornito in input ad ogni decision tree della foresta. Una volta ottenute le predizioni da tutti i decision tree si combinano. Nel caso di classificazione, per scegliere la classe di appartenenza, si usa la tecnica del voto a maggioranza, mentre nel caso di regressione si usa la media dell'output degli alberi. Questa tecnica di combinare il risultato a partire da più modelli è detta aggreggation. Nel random forest effettuiamo prima il bootstrapping e poi aggreggation, in gergo si parla in di bagging.
La casualità di questo algoritmo proviene da due fattori: la scelta dei campioni da inserire nei nuovi dataset e la scelta delle features utilizzate da ogni decision tree. Tutto ciò permette al nostro modello di essere meno sensibile ai dati di training rispetto a quanto lo fossero i decision tree. La scelta casuale delle features, invece, decrementa la correlazione tra gli alberi decisionali: se si usassero tutte le features, i nodi decisionali si comporterebbero in maniera molto simile, facendo crescere la varianza[7].

[7] Varianza: le prestazioni del modello allenato sono influenzate da piccoli cambiamenti nel set di dati. Avere una varianza alta si traduce nel rischio di overfit.

5.1.2.1 Principali parametri ed iperparametri

Gli iperparametri utilizzati per random forest sono: **criterion**, **min_samples_split**, **class_weigth**, **max_depth**, **n_estimators** e **bootstrap**. I primi quattro parametri, definiti anche per gli alberi decisionali, mantengono le stesse funzioni anche in random forest.
Il parametro **n_estiamtors** indica il numero di estimatori, cioè di decision tree, che si creano. Un numero alto di estimatori tende a dare delle performance migliori a costo di un rallentamento dell'esecuzione nella fase di training. In ogni caso, far crescere questo valore oltre una certa soglia non apporta più benefici al classificatore, come mostrato in fig. 5.3.

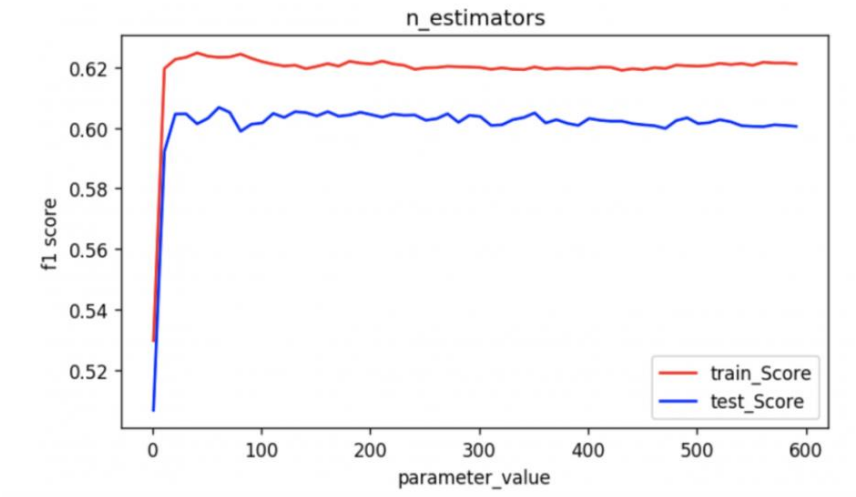


Fig. 5.3. Da un certo punto in poi, aumentare il numero di estimatori non comporta un miglioramento delle performance del modello.

Il parametro **bootstrap** assume valori booleani, nel caso in cui sia impostato a True si effettua il già descritto bootstrapping, se, invece, vale False ogni estimatore creato, sarà allenato sul dataset originale.

5.1.3 XGBoost

XGBoost sta per eXtreme Gradient Boosted trees. Si tratta di un algoritmo di machine learning supervisionato che può essere usato sia per classificazione che per regressione. XGBoost si basa sui concetti di decision tree e di gradient boosting.
Il **boosting** è un metodo di ensemble: diversi modelli sono combinati insieme per formare quello finale. Invece di addestrare tutti i modelli in modo isolato l'uno dall'altro, il boosting addestra i modelli in successione, ogni nuovo modello viene addestrato per correggere gli errori fatti dai precedenti. I modelli vengono aggiunti in modo sequenziale fino a quando non è possibile apportare ulteriori

migliorie (o si raggiunge il numero di massimo di estimatori che siamo disposti a creare, definiti dall'iperparametro n_estimators). Il vantaggio di questo approccio iterativo è che i nuovi modelli aggiunti si concentrano sulla correzione degli errori che sono stati causati dai modelli precedenti.
Le caratteristiche principali di XGBoost sono le seguenti:
• boosting regolarizzato: permette di prevenire l'overfitting, assicurandoci che il modello che si costruisce sia generalizzato;
• gestione automatica dei valori mancanti nel dataset (qualora ce ne fossero);
• scalabilità: si comporta bene sia con dataset di piccole dimensioni che con dataset più grandi;
• tree pruning: mentre gli alberi decisionali terminano la loro crescita (quindi la generazione di nuovi nodi) quando non c'è più margine di miglioramento o quando abbiamo raggiunto il valore di campioni dopo i quali non si permette più il branching (espresso in min_samples_weight), con XGBoost possiamo produrre un albero molto profondo e poi effettuare il pruning, ottenendo alberi ottimizzati.

5.1.3.1 Principali parametri e iperparametri

Gli iperparametri usati con XGBoost sono i seguenti:
• **learning_rate**: il gradient boosting comporta la creazione e l'aggiunta di alberi al modello in maniera sequenziale. Nuovi alberi vengono creati per correggere gli errori nelle previsioni della sequenza di alberi precedente. L'effetto è che il modello tende a adattarsi rapidamente rischiando l'overfitting. Una tecnica per rallentare l'apprendimento dei modelli gradient boosting è quello di applicare un peso (weight). Il learning rate indica quanto velocemente un modello apprende. Ogni albero aggiunto modifica il modello generale e la magnitudo della modifica è controllata da questo iperparametro, più basso è e più lentamente il modello apprenderà. Il vantaggio di avere un modello che apprende più lentamente è che questo risulterà essere più robusto ed efficiente in fase di predizione. Valori tipici di questo iperparametro sono tra 0.1 e 0.3, ma possono essere anche più piccoli.
• **max_depth**: indica il numero massimo di livelli che gli alberi possono avere. Vale ancora quanto detto per gli alberi di decisione; aumentare troppo questo valore conduce all'overfitting.
• **subsample**: il numero di istanze di training che XGBoost utilizzerà per creare gli alberi. Il valore di default è 1, perciò XGBoost userà il 100% delle informazioni contenute nel dataset per costruire il modello. I valori tipici sono tra 0.6 e 1.
• **colsample_bytree**: indica il numero di features che vengono utilizzate per la creazione degli alberi. Anche in questo caso il valore di default è 1, ciò significa

che XGBoost userà il 100% delle features presenti nel dataset. I valori assegnabili sono tra lo 0 e 1, ma solitamente sono numeri tra 0.6 ed 1.
• **reg_lambda** e **gamma**: questi iperparametri proprio come i due precedenti sono utilizzati per la regolarizzazione[8].
• **n_estimators**: sono il numero di alberi (o estimatori) che si possono costruire. Anche in questo caso continuare ad aumentare il numero di alberi non porta più a dei miglioramenti tangibili, anzi, causa il rallentamento dell'esecuzione.
• **scale_pos_weight:** viene utilizzato in caso di dataset sbilanciati permette di assegnare un peso maggiore alla categoria in minoranza, i valori tipici stanno attorno al rapporto tra la classe in maggioranza e quella in minoranza.

5.2 Metodologia

In questa sezione si illustrano i passi svolti per lo sviluppo del classificatore binario.
Per prima cosa è necessario acquisire i dati con cui allenare e valutare il classificatore, separandoli in set di training e test set, e il file contenete i target dei campioni, che contiene 1 per i contratti Ponzi e 0 altrimenti.
Adesso si possono iniziare ad allenare i classificatori. È durante questa fase che si effettua il tuning degli iperparametri, ottenendo i valori migliori con cui costruire il modello. Per far ciò si usa la grid search, che calcola tutte le combinazioni tra i valori che abbiamo assegnato agli iperparametri e determina le prestazioni per ogni combinazione, selezionando la migliore. Su Scikit-learn la grid search è implementata nella funzione `GridSearchCV()` che esegue anche la cross validation. Come già detto, prima del training del modello, si dividono i dati in due parti: training set e test set. Nella cross validation si dividono ulteriormente i dati di training in dati di addestramento e in dati di convalida. La cross validation più utilizzata è chiamata K-fold. Si tratta di un processo iterativo che divide i dati di training in k partizioni. Ogni iterazione mantiene una partizione per la validazione e le restanti k-1 partizioni per l'addestramento del modello. L'iterazione successiva imposterà la prossima partizione come dati di validazione (o di test) e le restanti k-1 come dati di addestramento e così via. Ad ogni iterazione si registrano le prestazioni del modello e alla fine si fornisce in output la loro media.
Una volta allenati tutti e tre gli algoritmi di classificazione che abbiamo preso in considerazione (Decision Tree, Random Forest, XGBoost), si seleziona quello che

[8] Un modello non regolarizzato effettua un buon fitting ma generalizzerà male sui nuovi dati. Per evitare ciò si aggiunge una penalizzazione alla complessità del modello tramite i termini di regolarizzazione.

restituisce il miglior punteggio e che si utilizzerà per fare le predizioni sul test set. I risultati della predizione del classificatore sul test set sono riportati sotto forma di matrice di confusione (fig. 5.4), uno strumento usato per analizzare gli errori compiuti da un modello di machine learning. La matrice di confusione è costituita da quattro celle che contengono rispettivamente i veri negativi (TN – true negative), i falsi positivi (FP – false positive), i falsi negativi (FN – false negative) e i veri positivi (TP – true positive).

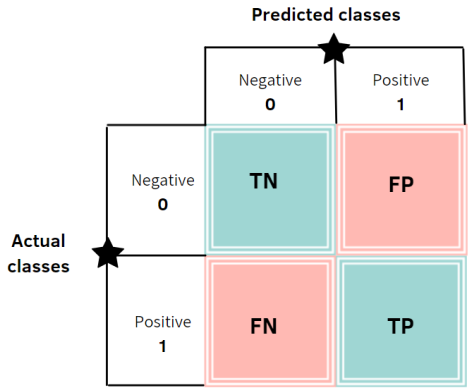


Fig. 5.4. Una matrice di confusione.

Infine, per tentare di migliorare ancora le prestazioni del classificatore è stata analizzata la curva di precision-recall [8]. Molti algoritmi di machine learning sono in grado di prevedere una probabilità di appartenenza ad una classe. Per classificare i samples nelle relative categorie, si osserva la "**soglia di decisione**" detta comunemente "**soglia**". Di default il valore della soglia è 0.5. Tutti i valori uguali o superiori alla soglia sono mappati in una classe e tutti gli altri sono mappati nell'altra.
Per problemi di classificazione su dataset sbilanciati (come nel nostro caso), la soglia predefinita può portare a prestazioni scadenti. Un semplice approccio per migliorare le prestazioni del classificatore consiste nell'effettuare il tuning della soglia in modo tale da cambiare le mappature nelle categorie. Quando si usa la curva di precision-recall, la soglia ottimale per il classificatore può essere calcolata direttamente.
La procedura per modificare il valore della soglia è il seguente:
1- allenare il modello sul training set;
2- predire le probabilità sul test set;
3- per ogni possibile valore S della soglia:
- si convertono le probabilità in categorie utilizzando la soglia S;
- si valutano le etichette;
- se lo score che si ottiene è migliore del precedente;
  - si adotta tale soglia;

4- si usa la soglia migliore per effettuare le predizioni di nuovi punti.

La curva di precision-recall si concentra sulle prestazioni di un classificatore rispetto alla classe in minoranza. Questa curva viene calcolata creando etichette di classe a partire dalle probabilità calcolate sul test set attraverso una serie di soglie, calcolando precision e recall per ogni soglia.

5.3 Codice del classificatore implementato

In questa sezione si descrive il codice del classificatore realizzato. Per svilupparlo è stato utilizzato Anaconda[9], una distribuzione per il calcolo scientifico che utilizza Python ed R, e Scikit-learn[10], una libreria open source per il machine learning. Da questa spiegazione sono omesse le parti di codice dove si ottengono ed analizzano i falsi negativi e i falsi positivi. Queste porzioni saranno descritte nella sezione 5.5.
Per prima cosa sono state importate tutte le librerie necessarie al funzionamento del codice, Fig. 5.5. Le prime sei sono librerie di utilità, la sesta, `matplotlib`, permette la creazione di grafici e la settima, `sklearn.metrics`, è la libreria che contiene metriche di valutazione utilizzate per calcolare le prestazioni dei classificatori. Le restanti sono necessarie per la parte di machine learning.

```
import sys
import numpy as np
import pandas as pd
import seaborn as sns # libreria utile per matrice di confusione
import plotly.express as px
from collections import Counter
from matplotlib import pyplot as plt
from sklearn import tree
from sklearn.metrics import accuracy_score, precision_score, recall_score, \
    f1_score, fbeta_score, make_scorer, confusion_matrix, precision_recall_curve
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
import xgboost as xgb
```

Fig. 5.5. Rappresenta tutte le librerie importate per la corretta esecuzione del codice.

Una volta importate le librerie, si legge il dataset e il file dei target (fig. 5.6). Il file dei target è un file in formato csv che contiene una sola colonna i cui valori sono 0 o 1, dove 0 indica contratto non Ponzi e 1 contratto Ponzi. Il file dataset.csv e target.csv sono in corrispondenza biunivoca. Oltre a ciò, si recupera il nome delle features, che saranno utili per quando si traccerà il grafico della loro importanza.

[9] https://www.anaconda.com/
[10] https://scikit-learn.org/stable/

```
# leggo il dataset ed il target

dataset = pd.read_csv(r'C:\Users\andre\Desktop\materiale tesi\settimana 16\dataset\cleaned+\dataset+.csv', header = 0)
features_names = list(dataset.columns) # ottengo i nomi delle features
features_names.pop(0) # rimuovo il primo addresses, in quanto non e' una feature
pairs_number_features = list() # creo una lista che contiene coppie <#_feature, nome_feature>
idx = 0

while idx < len(features_names):
    pair = (idx, features_names[idx])
    pairs_number_features.append(pair)
    idx+=1

print(pairs_number_features)

target = pd.read_csv(r'C:\Users\andre\Desktop\materiale tesi\settimana 16\dataset\cleaned+\target.csv', header = 0)
```

Fig. 5.6. Codice che mostra l'acquisizione del dataset e del file contenete i target. Questi sono recuperati fornendo il path assoluto della locazione del file.

Ottenuto il dataset, lo si divide in set di training e test set, sfruttando la funzione `train_test_split()` a cui sono forniti come argomenti il dataset e i target (fig. 5.7).

```
X = dataset.values[:, 1:] # rimuovo la prima colonna che contiene gli indirizzi
y = target.values # acquisico i target sulle y

# divido il dataset in train set (TR) e test set (TS)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42, stratify = y)
```

Fig. 5.7. Divisione del dataset in training set e test set.

Oltre a questi due parametri, se ne possono fornire altri:
• **test_size**: indica la dimensione del test set rispetto al train set, può assumere valori tra 0 ed 1, impostata a 0.2 crea un test set che è il 20% del dataset.
• **random_state**: controlla il rimescolamento applicato ai dati del dataset prima che questi vengano divisi in train set e test set. Mantenendo lo stesso valore di random_state nei test svolti si garantisce la stessa divisione dei dati; dunque, avremmo sempre il medesimo training set e test set.
• **stratify**: se impostato a True, permette di avere lo stesso rapporto tra le classi (Ponzi e non Ponzi) sia nel training set che nel test set.

Adesso ci prepariamo per il training dei classificatori. Per prima cosa, in fig. 5.8, si definisce la metrica utilizzata da grid search per valutare le fold della cross validation. Se questa metrica non è definita allora grid search utilizzerà la metrica di default cioè l'accuracy. Successivamente si crea un dizionario che contiene gli iperparametri e i relativi valori da testare.

```
fbeta = make_scorer(fbeta_score, beta = 1.5)


param_dist_DT = {
                'criterion' : ['entropy', 'gini'],
                'max_depth' : [6, 7, 8, None],
                'min_samples_split' : [20, 25, 30, 35], # il valore tipico sta fra 1 e 40
                'class_weight': [{0:1, 1:3.5}, {0:1, 1:3.75}]
}
```

Fig. 5.8. Creazione della metrica fbeta e del dizionario che sarà utilizzato nella grid search.

Una volta definita la configurazione degli iperparametri su cui si esegue la grid search, ha inizio la fase di training del classificatore, che viene svolta con le prime due righe di fig. 5.9.

```
# effettuo la GridSearchCV() per scegliere i migliori iperparametri
grid_dt = GridSearchCV(DecisionTreeClassifier(), param_grid = param_dist_DT, scoring = fbeta, cv = 3, n_jobs = 8)
grid_dt.fit(X_train, y_train) # alleno su train
y_pred_train = grid_dt.predict(X_train) # effettuo la predizione su train set
print('best hyperparameters:: ', grid_dt.best_params_) # stampo i migliori iperparametri
print('best score:: ', grid_dt.best_score_) # media degli score (basata sulla metrica di scoring) date sulle cross validation
print('recall score on TR:: ', recall_score(y_train, y_pred_train))
print('precision score on TR:: ', precision_score(y_train, y_pred_train))
```

```
best hyperparameters::  {'class_weight': {0: 1, 1: 5}, 'criterion': 'entropy', 'max_depth': 7, 'min_samples_split': 15}
best score::  0.6100482018386231
recall score on TR::  0.7301293900184843
precision score on TR::  0.5170157068062827
```

Fig. 5.9. Codice contenente la funzione di grid search utilizzata per effettuare il tuning degli iperparametri sull'algoritmo di classificazione decision tree.

L'obiettivo è quello di trovare i valori degli iperparametri con cui ottenere le migliori predizioni possibili.
Nelle righe successive, dopo aver ottenuto le predizioni sul set di training, si stampano i risultati ottenuti dal classificatore sulle varie metriche considerate e, in base a questi, si modificano i valori associati agli iperparametri.
Una volta ottenuto l'insieme dei migliori iperparametri, si controlla se siamo in rischio di overfitting (fig. 5.10), stampando l'accuracy che il modello ottiene su training set e su test set. Nel caso di overfit si esegue un nuovo tuning degli iperparametri, cambiandone il valore per cercare di regolarizzare il modello.

```
clf_dt = DecisionTreeClassifier(**grid_dt.best_params_)
clf_dt.fit(X_train, y_train)
print('accuracy score on TR:: ', clf_dt.score(X_train, y_train))
print('accuracy score on TS:: ', clf_dt.score(X_test, y_test))
```

```
accuracy score on TR::  0.8447187928669411
accuracy score on TS::  0.8135964912280702
```

Fig. 5.10. Codice per controllare il rischio di overfitting. In questo caso la differenza in punteggio tra train e test è piccola e non si rischia l'overfitting.

Tutto quello descritto, dalla creazione del dizionario contenente gli iperpametri fino all'analisi dell'overfit, è ripetuto anche per gli altri algoritmi di classificazione utilizzati: random forest e XGBoost.
Una volta ottenuti i migliori iperparametri con cui allenare i tre classificatori è stato necessario confrontarli per capire quale fra questi risultasse il migliore.
Per poterli comparare è stato istanziato un nuovo dizionario (fig. 5.11) con tre entries. Ogni entry ha a sua volta due campi: 'model' che contiene il nome dell'algoritmo di machine learning e 'params' che mantiene gli iperparametri.

```
model_params = {
    'decision_tree': {
        'model': DecisionTreeClassifier(),
        'params': {
            'criterion' : ['entropy', 'gini'],
            'max_depth' : [6, 7, 8, None],
            'min_samples_split' : [20, 25, 30, 35], # il valore tipico sta fra 1 e 40
            'class_weight': [{0:1, 1:3.5}, {0:1, 1:3.75}]
        }
    },
    'random_forest': {
        'model': RandomForestClassifier(),
        'params': {
            'n_estimators':[180, 190, 200, 210],
            'min_samples_split' : [35, 40],
            'max_depth' : [6,7,8],
            'criterion':['gini', 'entropy'],
            'class_weight' : [{0:1, 1:5.5}, {0:1, 1:6}],
            'bootstrap': [True, False]
        }
    },
    'xgboost': {
        'model': xgb.XGBClassifier(eval_metric = 'aucpr', use_label_encoder = False),
        'params': {
                'learning_rate' : [0.05],
                'n_estimators' : [70, 80, 90, 100, 110, 120],
                'max_depth': [6, 7, None],
                'colsample_bytree' : [0.8],
                'subsample' : [0.8],
                'scale_pos_weight' : [5, 5.5, 6],
                'reg_lambda' : [10, 100]
        }
    }
}
```

Fig. 5.11. Dizionario contenente gli algoritmi di classificazione considerati con i range dei migliori iperparametri.

Si esegue perciò una nuova grid search, fig. 5.12. Alla prima riga si dichiara la lista `scores`, che conterrà, per ognuno dei tre classificatori utilizzati, il nome (Decision Tree, Random Forest o XGBoost), il miglior score ottenuto e la miglior combinazione degli iperparametri. Il ciclo for alla riga successiva viene svolto tre volte, ad ogni ciclo si allena un classificatore con il relativo set di iperparametri. Al termine del ciclo si stampano i risultati ottenuti.
A questo punto, si ordinano in base al punteggio i classificatori e rialleno il migliore (fig. 5.13).

```
scores = []

for model_name, mp in model_params.items():
    clf =  GridSearchCV(mp['model'], mp['params'], scoring = fbeta, cv = 3, return_train_score = False, n_jobs = 8)
    clf.fit(X_train, y_train.ravel())
    scores.append({
        'model': model_name,
        'best_score': clf.best_score_,
        'best_params': clf.best_params_,
        'best_estimator': clf.best_estimator_ # necessario per riallenare
    })

pd.set_option("display.max_colwidth", None)
df = pd.DataFrame(scores,columns=['model','best_score','best_params'])
df
```

Fig. 5.12. Codice per ottenere la classifica dei tre classificatori.

```
models_list = sorted(scores, key=lambda d: d['best_score']) # ordino in base al best_score
model = models_list[-1] # prendo l'ultimo modello (quello con score piu' alto)
clf = model.get('best_estimator') # ottengo i valori con cui allenare
clf.fit(X_train, y_train.ravel()) # rialleno
```

Fig. 5.13. Selezione del miglior classificatore.

Ora si esegue la predizione sul test set, fig. 5.14. Con la prima riga di codice si ottengono le predizioni sui contratti del test set con il classificatore che abbiamo adottato. Nelle righe successive stampiamo, invece, i punteggi ottenuti sulle metriche di precision, recall, accuracy e F-beta. Infine, si produce la matrice di confusione.

La matrice di confusione presentata in fig. 5.14 contiene 697 veri negativi, cioè contratti non Ponzi classificati come tali, 91 veri positivi, cioè contratti Ponzi classificati come tali, 44 falsi negativi, cioè contratti Ponzi classificati come non Ponzi e 80 falsi positivi, contratti non Ponzi classificati come Ponzi.

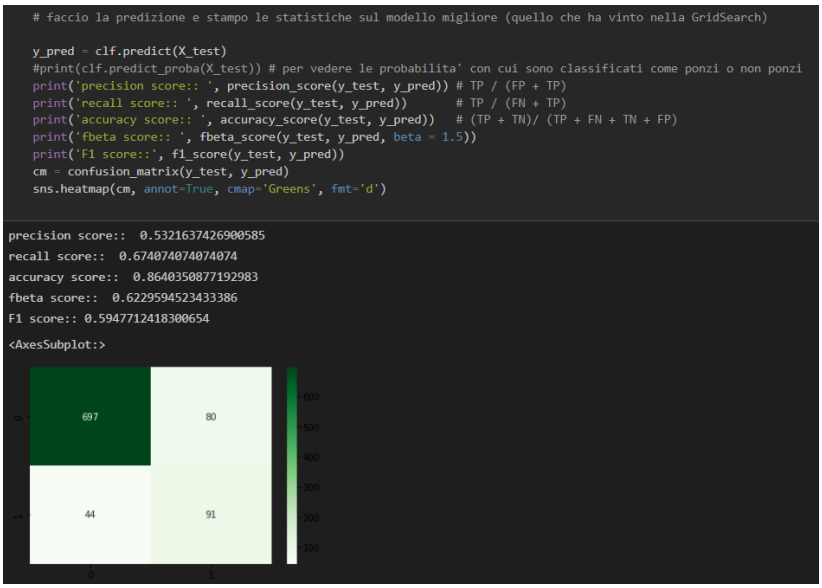


Fig. 5.14. Predizione sul test set fatta col classificatore che restituisce lo score migliore.

Successivamente si stampa la classifica delle features in ordine di importanza per il modello vincitore, rappresentandole sia in forma numerica che tramite istogramma. Il codice è presentato in fig. 5.15. Con la prima riga di codice: `clf.feature_importances_` otteniamo una lista coi soli punteggi delle features ma senza il loro nome. Per poterle riconoscere, con le righe successive, associamo i punteggi ai nomi delle features e se ne stampa sia la classifica in ordine di importanza che un grafico a istogramma.

```
# vado a stampare l'importanza delle features del modello vincitore
features_importances = clf.feature_importances_
res = dict(zip(pairs_number_features, features_importances))
sorted_dict = dict(sorted(res.items(), key=lambda item: item[1], reverse = True))
print(pd.DataFrame(list(sorted_dict.items()),columns=['(Number, Feature)', 'Importance']))
plt.bar(range(len(features_importances)), features_importances) # stampo istogramma delle features
plt.show()
```

Fig. 5.15. Codice per ottenere la classifica, in ordine di importanza, delle features.

Tutto ciò che è stato fatto per il modello vincitore, viene svolto anche per il secondo, sia per confrontarne le prestazioni, sia perché può capitare che il secondo modello dia dei risultati migliori sul test set rispetto al primo classificato.
Nell'ultimo step eseguito abbiamo modificato la soglia di decisione (sezione 5.3) per vedere se si riesce a migliorare il comportamento del modello sui dati di test. A seguito del codice proposto in fig. 5.16, dove si calcola precision e recall per ogni soglia, viene creato il grafico della curva di precision-recall con recall sull'asse delle x e precision sull'asse delle y (fig. 5.17).

```
# predizione delle probabilita' del primo modello
y_pred_proba_winner = clf.predict_proba(X_test)
# si tengono solo le probabilita' che il contratto sia Ponzi
y_pred_proba_w = y_pred_proba_winner[:, 1]
# calcola pr-curve
p, r, thrs = precision_recall_curve(y_test, y_pred_proba_w)

# predizione delle probabilita' del secondo modello
y_pred_proba_second = clf2.predict_proba(X_test)
# si tengono solo le probabilita' che il contratto sia Ponzi
y_pred_proba_s = y_pred_proba_second[:, 1]
# calcola pr-curve
p2, r2, thrs2 = precision_recall_curve(y_test, y_pred_proba_s)

# si fanno i plot della curva di pr per i due modelli
no_skill = len(y_test[y_test == 1]) / len(y_test)
plt.plot([0,1], [no_skill, no_skill], linestyle = '--', label = 'No Skill') # stampo il no-skill
plt.plot(r, p, marker = '.', label = 'XGB') # stampo il vincitore (in questo caso XGB)
plt.plot(r2, p2, marker = '.', label = 'RF') # stampo il secondo migliore (RF)
# axis labels
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.legend()
# si mostra il grafico
plt.show()
```

Fig. 5.16. Codice per disegnare la precision-recall curve.

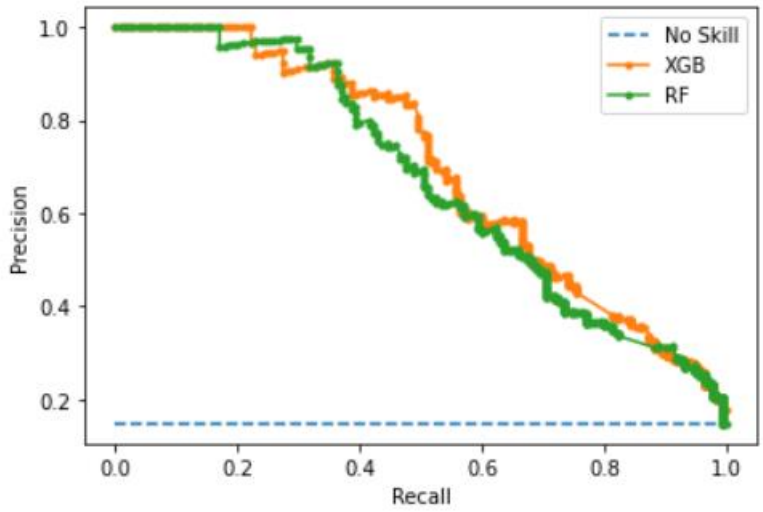


Fig. 5.17. Le curve di precision-recall dei due modelli migliori. In verde la curva di random forest, in arancione la curva di XGBoost. In azzurro tratteggiato, la curva di un modello senza skill.

Per ottenere la miglior soglia per il problema che stiamo affrontando è necessario definire una metrica adeguata, nel nostro caso è stata utilizzata F-beta (spiegata nella sottosezione 5.4.4) con valore di beta pari a 1.75. Successivamente si individua l'indice dove il valore soglia è maggiore e si recupera sia il valore della soglia che il valore di F-beta. Tutto ciò è stato eseguito con il codice mostrato in fig. 5.18.

```
# convert to f score
beta = 1.5 # mettere valori di beta <= 1 equivale a favorire la giusta classificazione dei non Ponzi. Viceversa se beta >= 1
# definisco fbeta per il primo modello
fbetaw = ((1 + beta**2) * p * r) / (beta**2 * p + r)
# locate the index of the largest fbeta score
ixw = np.argmax(fbetaw)

# definisco fbeta del secondo modello
fbetas = ((1 + beta**2) * p2 * r2) / (beta**2 * p2 + r2)
# locate the index of the largest fbeta score
ixs = np.argmax(fbetas)
print('Best Threshold of the winner model = %f, Fbeta-Score = %.3f' % (thrs[ixw], fbetaw[ixw]))
print('Best Threshold of the second model = %f, Fbeta-Score = %.3f' % (thrs2[ixs], fbetas[ixs]))
```

Fig. 5.18. Codice per ottenere la soglia migliore ed il punteggio maggiore di fbeta.

Individuato il miglior valore della soglia, è possibile determinare una nuova matrice di confusione utilizzando quanto appena calcolato. Il codice e la matrice sono riportati in fig. 5.19.
Alla prima riga calcolo le etichette utilizzando la soglia ottenuta con il codice di fig. 5.18. Rispetto a quanto fatto in fig. 5.14, adesso utilizziamo la funzione `predict_proba()` che, a differenza della `predict()` che dà in output delle etichette (0 o 1), restituisce le probabilità di un contratto del test set di essere Ponzi o non Ponzi. Se la probabilità di essere Ponzi supera la soglia decisionale allora sarà classificato come contratto Ponzi altrimenti come contratto non Ponzi. Nelle righe successive si stampano le varie statistiche (come già fatto in fig. 5.14).

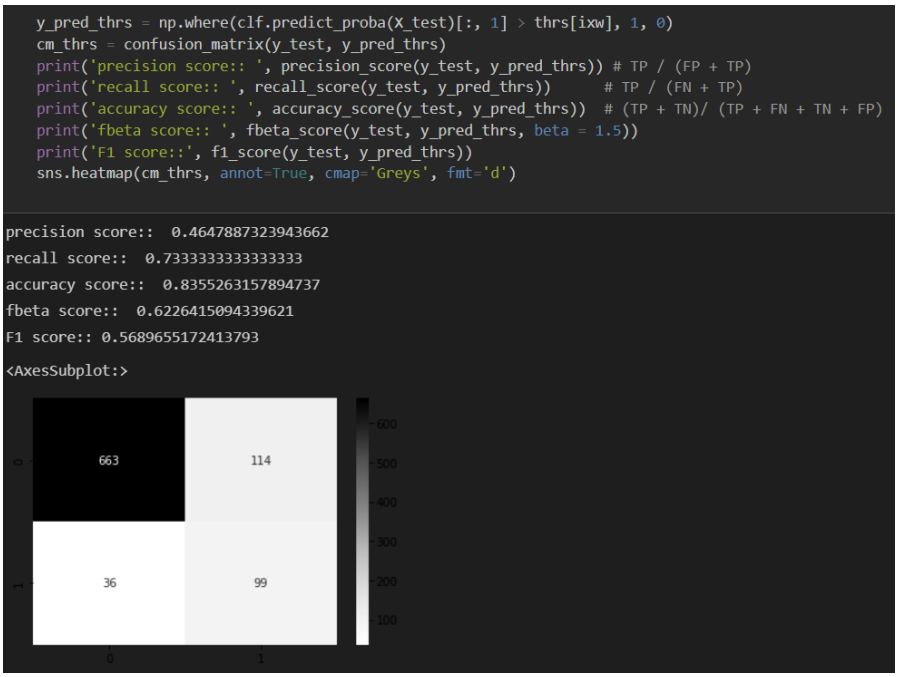


Fig. 5.19. Codice per stampare la nuova matrice di confusione utilizzando la nuova threshold e matrice di confusione

5.4 Evoluzione dei modelli

Questa sezione descrive le sperimentazioni e le valutazioni dei modelli costruiti, dai primi test eseguiti fino alla scelta del modello finale, concentrandosi in particolare sulle modifiche e i ragionamenti fatti di volta in volta per cercare di migliorare le prestazioni.

Dataset utilizzato	Nome del test
A1	5.4.1 #1
B1	5.4.1 #2
C1	5.4.2 #1
C1	5.4.2 #2
B1	5.4.4 #1
B2	5.4.4 #2
B3	5.4.4 #3
B3	5.4.5 #1
B3	5.4.5 #2

Tab. 1. Mostra i dataset utilizzati e i relativi nomi dei test, saranno utili in fase di confronto.

Prima di tutto conviene assegnare dei nomi ai vari dataset utilizzati. Il dataset originale è chiamato A1. Contiene circa 5600 contratti di cui poco più di un migliaio con valori delle features nulli, i così detti contratti “vuoti”, privi cioè di

informazioni utili. Questi sono i contratti che dopo essere stati pubblicati sulla blockchain non sono mai stati contattati. Il secondo dataset, chiamato B1, è stato ricavato da A1 dopo la rimozione dei contratti "vuoti". Da B1 deriva un altro dataset, B2. Questo contiene una feature in più, mentre B3 ne ha sette in meno di B2. Infine, l'ultimo dataset è C1 e contiene le features di [3]. In tabella 1 sono riportati i nomi dei dataset associati ai test eseguiti.

5.4.1 I primi test eseguiti

Il primo modello (test 5.4.1 #1) è stato costruito col dataset A1, utilizzando come metrica di scoring nella `GridSearchCV()` l'accuracy, ciò significa che per valutare i punteggi delle fold create dalla cross validation si tengono in considerazione il numero dei contratti correttamente classificati rispetto al loro totale. Il codice del primo classificatore è disponibile su GitHub all'indirizzo: https://github.com/Lychett/tesi-smart-ponzi/blob/main/classifiers/original%20dataset/original_dataset.ipynb. Tutti i classificatori costruiti hanno di base la stessa struttura, descritta nella sezione 5.2.

Fig. 5.20. Matrice di confusione ottenuta con il primo test.

Il risultato migliore si ottiene con l'algoritmo decision tree. In fig. 5.20, è riportata la matrice di confusione. Questo primo modello produce sia troppi falsi positivi che troppi falsi negativi.
Questo scarso risultato deriva dal fatto che il dataset A1 contiene dei contratti "vuoti" sia lato Ponzi che lato non Ponzi. Tutto ciò manda in confusione il classificatore che, da questi smart contract, non riesce ad apprendere informazioni con cui discriminare tra le due classi.

Il passo successivo è stato dunque quello di ripulire l'insieme di dati ottenendo il dataset B1. In fig. 5.21 sono riportati i migliori punteggi calcolati con il dataset A1 mentre in fig. 5.22 sono presenti quelli ottenuti con B1 (sul test 5.4.1 #2).

	model	best_score	best_params
0	decision_tree	0.788122	{'class_weight': {0: 1, 1: 4.25}, 'criterion': 'gini', 'max_depth': 6, 'min_samples_split': 10}
1	random_forest	0.759991	{'bootstrap': True, 'class_weight': {0: 1, 1: 4.25}, 'criterion': 'entropy', 'min_samples_split': 15, 'n_estimators': 125}
2	xgboost	0.760438	{'colsample_bytree': 0.8, 'gamma': 0, 'n_estimators': 140, 'reg_lambda': 5}

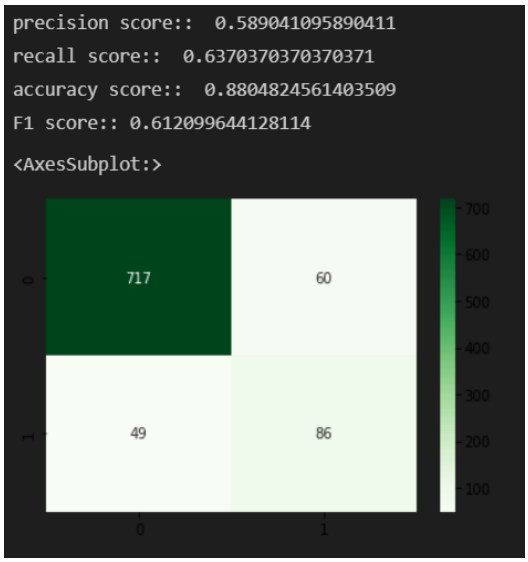
Fig. 5.21. best_score e best_params dei classificatori basati su A1.

	model	best_score	best_params
0	decision_tree	0.854321	{'class_weight': {0: 1, 1: 4}, 'criterion': 'entropy', 'max_depth': 9, 'max_features': 18, 'min_samples_split': 5}
1	random_forest	0.858711	{'bootstrap': False, 'class_weight': {0: 1, 1: 5.5}, 'criterion': 'entropy', 'n_estimators': 225}
2	xgboost	0.877366	{'colsample_bytree': 0.8, 'gamma': 0, 'n_estimators': 120, 'reg_lambda': 15, 'scale_pos_weight': 5.5}

Fig. 5.22. best_score e best_params dei classificatori basati su B1.

Confrontando i risultati di fig. 5.22 con quelli del primo test, si apprezza, oltre ad un cambio del miglior classificatore, che passa da essere un decision tree ad un XGBoost, anche un incremento del miglior punteggio che passa da 0.788 a 0.877. Ripulire il dataset dai contratti vuoti ha effettivamente contribuito al miglioramento dei risultati.

I risultati sul test set ottenuti con questo modello sono riportati in fig. 5.23.



Si ha un discreto aumento della precision rispetto al primo test effettuato, che passa da 0.404 a 0.589, così come di recall che va da 0.561 a 0.637, con l'accuracy che guadagna circa 0.1. L'obiettivo dei prossimi test sarà quello di provare ad abbassare il numero di falsi negativi anche a scapito di una diminuzione di precision. Come già detto nei capitoli precedenti, vorremmo un classificatore che riesca a individuare abbastanza contratti Ponzi anche al prezzo di un aumento (in misura ragionevole) dei falsi positivi.

Fig. 5.23. Matrice di confusione del secondo test effettuato.

5.4.2 Test con le features adottate in [3]

Nei seguenti due test i classificatori sono stati costruiti basandosi esclusivamente sulle features descritte nella sottosezione 4.3.2. Per poter allenare i classificatori

con queste features è stato necessario implementare un nuovo script Python (con una struttura simile a quello già descritto nel capitolo 4.6), dato che gli autori dell'articolo non hanno messo a disposizione un programma da eseguire con cui poterle recuperare. Il dataset costituito con queste feature è detto C1.
La prima prova (5.4.2 #1) è stata eseguita usando l'insieme di iperparametri proposti in [3] e riportati in fig. 5.24.

```
model_params = {
    'random_forest': {
        'model': RandomForestClassifier(),
        'params': {
            'n_estimators':[20,40,60,80,100,120,140,160,180,200],
            'criterion':['gini', 'entropy'],
            'bootstrap': [True, False]
        }
    },
    'xgboost': {
        'model': xgb.XGBClassifier( eval_metric = 'aucpr',  use_label_encoder = False),
        'params': {
                'n_estimators' : [100,110,120,130,140,150,160,170,180,190,200],
                'learning_rate': [0.1,0.2,0.3],
                'max_depth': [3,6,8,9]
        }
    }
}
```

Fig. 5.24. Iperparametri coi relativi valori usati da [3].

Il comportamento del classificatore vincitore (XGBoost) sul test set lascia molto a desiderare (fig. 5.25): l'accuracy rimane sullo stesso livello del test precedente, ma il numero di falsi negativi è troppo elevato. Ciò si traduce nell'avere un punteggio in recall estremamente basso. Se vogliamo un classificatore che sia in grado di identificare i contratti Ponzi non possiamo di certo adottare questo.

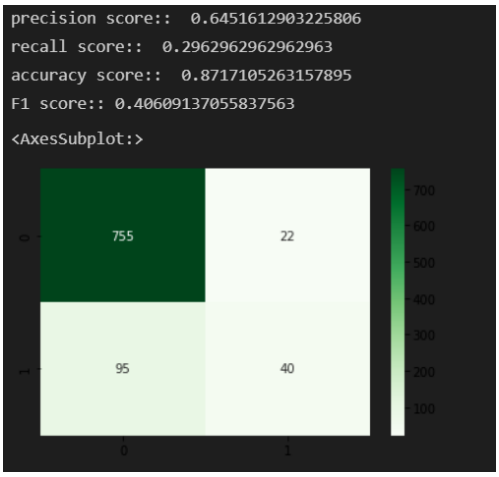


Fig. 5.25. Matrice di confusione ottenuta con il modello proposto da [3].

Gli scarsi risultati di questo test sono dovuti principalmente a due fattori:
• non vengono utilizzati iperparametri importanti, ad esempio random forest non usa 'min_samples_split' o 'class_weight' mentre per XGBoost non sono presi in considerazione 'scale_pos_weight' e altri iperparametri necessari per la regolarizzazione del modello. Tutti questi iperparametri risultano utili quando gli

algoritmi di classificazione sono allenati su dataset sbilanciati come in questo caso.
• La seconda causa è che, basandoci sulle sole 13 features individuate dall'articolo, una grande quantità di contratti (per entrambe le classi) assume dei valori nulli. Come già detto, questo non fa bene al classificatore che non ha modo di apprendere da questi campioni.
Nella Tab. 2 sono riportati i risultati di questo test confrontati con quelli ottenuti dagli autori dall'articolo basandosi sulle account features. Dalla tabella si nota un miglioramento su tutte le metriche dato dal fatto che la qualità del dataset, arricchito sia con contratti Ponzi che con non Ponzi, è stata migliorata.

	Punteggi ottenuti col test 5.4.2 #1 su XGB	Punteggi ricavati dall'articolo su XGB - account features
precision	0.645	0.59
recall	0.296	0.22
F1	0.406	0.32

Tab. 2. Mette a confronto i risultati ottenuti da [3] e quelli ottenuti col nostro classificatore.

Il secondo test (5.4.2 #2) eseguito su C1 è stato fatto modificando gli iperparametri; aggiungendo quelli descritti nella sezione 5.2 ma non usati dagli autori dell'articolo [3]. Questa prova è stata costruita con l'intento di capire se, effettuando il tuning degli iperparametri, si potessero raggiungere lo stesso dei buoni risultati. In questo test il classificatore che produce risultati migliori è Random Forest. I risultati sul test set e i punteggi su precision, recall, accuracy e F1 sono riportati in fig. 5.26.

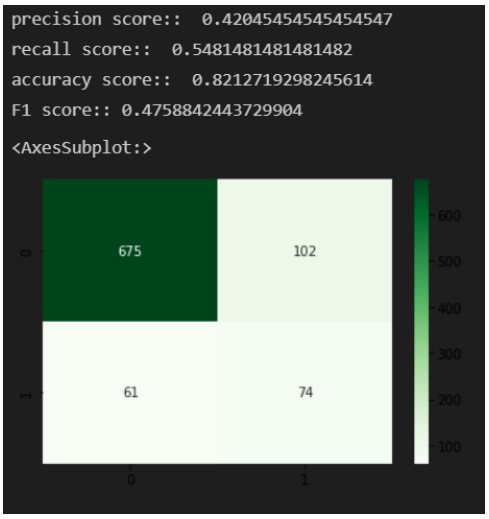


Fig. 5.26. Matrice di confusione ottenuta utilizzando le features di [3] con nuovi iperparametri.

Questi valori sono ancora mediocri, ma comunque migliori rispetto a quelli del test precedente, segno che selezionando i giusti iperparametri è possibile

incrementare le prestazioni del modello, anche se questi da soli non bastano se non abbiamo un dataset contenente features rilevanti.

A seguito dei risultati ottenuti fino ad ora ci potremmo già ritenere soddisfatti. Infatti, con il test 5.4.1 #2, abbiamo ottenuto dei punteggi che superano quelli dei modelli basati su C1. Ad ogni modo sono state eseguite altre prove per verificare se fosse possibile migliorare ulteriormente il modello, concentrandosi in particolare sul diminuire il numero di falsi negativi.

5.4.3 Sulle metriche della grid search:

Per provare a migliorare le prestazioni, sono stati letti vari articoli ([10], [14], [15]) che trattano l'importanza di utilizzare una metrica adeguata in base al tipo di problema. Nel nostro caso vorremmo cambiare metrica con cui valutare i punteggi sulle fold della cross validation eseguita dalla `GridSearchCV()`. In particolare, per classificatori allenati su dataset sbilanciati, si consiglia di non usare l'accuracy, cioè il rapporto tra il numero di predizioni corrette e il numero totale di predizioni. Il problema di utilizzare questa metrica risiede nell'avere un numero di samples di una classe maggiore rispetto a quelli dell'altra. Quindi un modello che classifica bene solo i samples in maggioranza può raggiungere accuracy attorno al 90% (ad esempio se il rapporto dei sample fosse di 9 ad 1). Per allenare un classificatore su un dataset sbilanciato risultano più interessanti metriche come recall, nel caso in cui si voglia andare a minimizzare il numero di falsi negativi, o F1, quando precision e recall hanno la medesima importanza.
L'articolo [10] fornisce dei consigli pratici su quali metriche usare in base al problema affrontato, che sono riassunte nel seguente schema.

"Are you predicting probabilities?
    Do you need class labels?
        Is the positive class more important?
            Use Precision-Recall AUC
        Are both classes important?
            Use ROC AUC
    Do you need probabilities?
        Use Brier Score and Brier Skill Score
Are you predicting class labels?
    Is the positive class more important?
        Are False Negatives and False Positives Equally Important?
            Use F1-Measure
        Are False Negatives More Important?
            Use F-beta, with beta > 1
        Are False Positives More Important?

Use F-beta, with beta < 1
Are both classes important?
Do you have < 80%-90% Examples for the Majority Class?
Use Accuracy
Do you have > 80%-90% Examples for the Majority Class?
Use G-Mean"

5.4.4 Una nuova metrica

Abbiamo eseguito così altri tre test adottando una nuova metrica con cui valutare i risultati della cross validation: F-beta con valore di beta pari a 1.5. Il valore che assume beta determina se dare più importanza a precision o a recall. Per valori di beta maggiori di 1, si conferisce più importanza a recall, altrimenti a precision.
Perché usare F-beta anzi che F1 o recall?
Usando una metrica come F1 si favorisce la creazione di un modello dove recall e precision hanno stessa importanza; dunque, dove falsi positivi e falsi negativi hanno medesimo rilievo.
Usando, invece, recall costruiremmo un modello che classifica correttamente la maggior parte dei contratti Ponzi, ma troppi non Ponzi diventerebbero falsi positivi. Come consiglia [10], è meglio basarsi sugli F-score (F1, F-beta ecc…).

Il primo classificatore costruito (5.4.4 #1) è stato allenato sul dataset B1.
I risultati della grid search ottenuti da questo test sono presentati in fig. 5.27:

	model	best_score	best_params
0	decision_tree	0.589375	{'class_weight': {0: 1, 1: 4.5}, 'criterion': 'gini', 'max_depth': 8, 'max_features': 15, 'min_samples_split': 15}
1	random_forest	0.615627	{'bootstrap': False, 'class_weight': {0: 1, 1: 6}, 'criterion': 'entropy', 'max_depth': 10, 'min_samples_split': 30, 'n_estimators': 200}
2	xgboost	0.641344	{'gamma': 0.2, 'learning_rate': 0.1, 'max_depth': 6, 'n_estimators': 90, 'reg_lambda': 50, 'scale_pos_weight': 6}

Fig. 5.27. Risultati della grid search sul dataset B1 con metrica F-beta.

I punteggi risultano essere minori rispetto a quelli restituiti dai test precedenti. Questo è dovuto al fatto che le fold della cross validation sono adesso valutate con F-beta.
Il classificatore che restituisce score migliore sui dati di training è ancora l'XGBoost, i risultati che ottiene sul test set sono riportati in fig. 5.28.
Se paragonati ai risultati da 5.4.1 #2, si nota una riduzione dei falsi negativi, al costo di un aumento nel numero di falsi positivi. Di conseguenza si alza il punteggio in recall mentre si abbassa il valore di precision. Questo è ciò che effettivamente ci si aspetta passando a F-beta.

Negli altri due test svolti utilizzando come metrica F-beta sono stati modificati i dataset con cui allenare i classificatori. Nel primo (5.4.4 #2) è stata aggiunta

all'insieme di dati una feature utilizzata da [3], costruendo così il dataset B2. La feature in questione è 'known_rate' che nelle classifica stilate nel test 5.4.2 #2 risulta abbastanza importante trovandosi in seconda posizione.

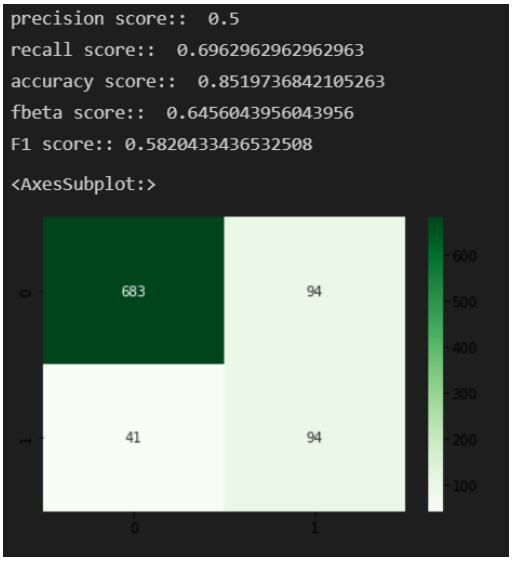


Fig. 5.28. Matrice di confusione ottenuta con la metrica F-beta sul dataset B.

Nel secondo test (5.4.4 #3), invece, sono state rimosse sette features dal dataset B2, ottenendo il dataset B3. Queste features sono quelle che nelle prove precedenti tendevano a ricoprire le ultime posizioni nella classifica di importanza e che dunque aiutavano meno il classificatore a discriminare tra i due tipi di contratti. Le features rimosse sono: 'tx_out', 'sdev_tx_out', 'investment_in', 'payment_out', 'mean_v2', 'owner_no_eth' e 'owner_gets_eth_investing'.

I risultati ottenuti con questi due test non variano troppo rispetto al primo proposto in questa sottosezione, apportando comunque delle piccole migliorie in precision.

5.4.4 Gli ultimi test eseguiti - oversampling

Negli ultimi due test eseguiti (5.4.5 #1 e 5.4.5 #2) si utilizza la tecnica dell'oversampling. L'oversampling permette di creare dei nuovi campioni della classe in minoranza nel training set, in modo da ottenere un insieme di dati di allenamento bilanciato.
In Scikit-learn i principali due metodi implementati per effettuare l'oversampling sono la tecnica SMOTE, acronimo di Synthetic Minority Oversampling Technique e il random oversampling.
SMOTE prende in considerazione le osservazioni più vicine, usando la distanza euclidea, tra quelle della classe minoritaria, effettua la differenza tra i due vettori di features e moltiplica questo valore per un numero casuale tra 0 e 1. In altre

parole, applica una perturbazione alla distanza tra due punti della classe minoritaria, creando dei campioni artificiali.
Random oversampling, invece, duplica le istanze della classe minoritaria presenti set di allenamento fino a che non si raggiunge il bilanciamento tra le due classi.

Il dataset usato in questi test è il B3 dato che la rimozione delle sette features non causa un peggioramento delle prestazioni, mentre lo scoring utilizzato nella grid search torna a essere l'accuracy, infatti, avendo un training set bilanciato possiamo tranquillamente basarci su questa metrica.

Il risultato migliore tra i due test ci viene fornito dal random oversampling (5.4.5 #2) che riesce ad ottenere un numero minore di falsi negativi rispetto allo SMOTE. I risultati sono riportati in fig. 5.29. Ad ogni modo entrambe le tecniche di oversampling non ci permettono di migliorare il classificatore, risultando leggermente peggiori rispetto ai test svolti nella sottosezione 5.4.3.

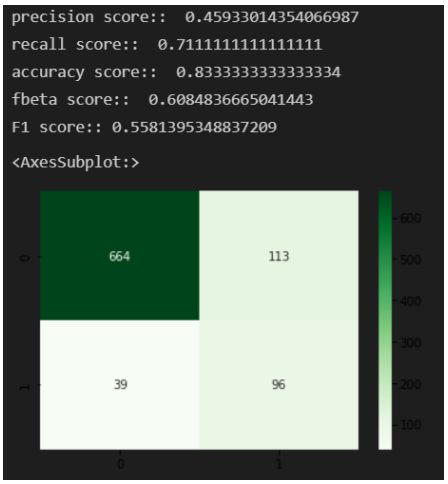


Fig. 5.29. Matrice di confusione ottenuta con tecnica di random oversampling con classificatore XGBoost.

5.4.5 La soglia di decisione e la scelta del miglior modello

L'ultimo metodo utilizzato per cercare di migliorare il comportamento dei modelli costruiti è stato effettuare il tuning della soglia di decisione (introdotta nella sezione 5.2) tramite la curva di precision-recall. Una volta allenato il modello, anzi che calcolare subito le etichette della classe dei samples presenti nel test set, se ne ottengono le probabilità di appartenenza ad una delle due classi. Modificando la soglia con cui decidere se un contratto è considerabile uno schema Ponzi o meno, abbiamo ottenuto un effettivo miglioramento trovando così il migliore modello. Si tratta del test 5.4.4 #2 il cui codice è disponibile all'indirizzo: https://github.com/Lychett/tesi-smart-ponzi/tree/main/classifiers/dataset%20B2.

I risultati ottenuti col nuovo valore di soglia (0.422) sul test set sono riportati in fig. 5.30.

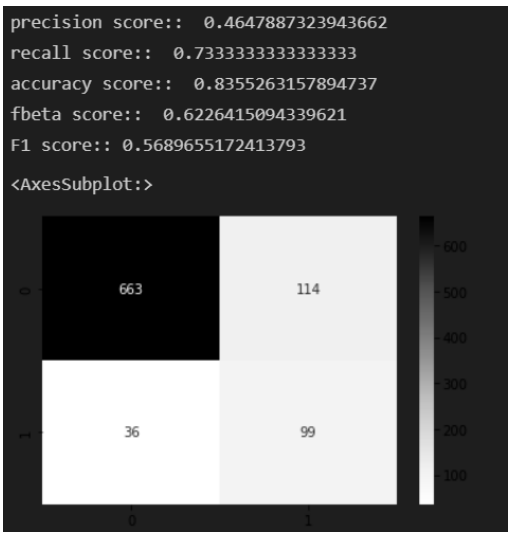


Fig. 5.30. Matrice di confusione sul test set del modello scelto.

In Tab. 3 sono, infine, riportati i punteggi ottenuti col nostro modello vincitore, col modello costruito utilizzando il dataset C1 e iperparametri usati in [3] (5.4.2 #1) e con il modello allenato e valutato con C1 ma su cui abbiamo modificato gli iperparametri (5.4.2 #2). Col nostro modello, al netto di una perdita in precision (c'è sempre un trade-off fra precision e recall) di 0.18 rispetto al test 5.4.2 #1, riusciamo a incrementare sia recall che F1. Il modello addestrato sul dataset B2 sembra essere migliore nel riconoscere i contratti Ponzi rispetto ai due modelli basati su dataset C1. Per essere certi di ciò, nella sezione 5.6 effettuiamo una analisi statistica per capire se la differenza riscontrata nei risultati ha una qualche rilevanza statistica.

	Valori ottenuti col nostro modello vincitore	Valori ottenuti con test sottosezione 5.4.2 #2	Valori ottenuti con test sottosezione 5.4.2 #1
precision	0.464	0.420	0.645
recall	0.733	0.548	0.296
F1	0.568	0.475	0.406

Tab. 3. Confronto tra i risultati ottenuti.

5.5 L'analisi dei falsi positivi

Una volta scelto il modello abbiamo cercato di capire come mai ci fossero 114 contratti "safe" classificati come Ponzi nel test set. Per far ciò abbiamo eseguito

un'analisi delle caratteristiche di questi contratti scrivendo il codice che recuperasse i falsi positivi e che successivamente li esaminasse.
Nella prima fase abbiamo raccolto le informazioni relative ai contratti mal classificati (sia FP che FN) e abbiamo disegnato i grafici e istogrammi per visualizzare la situazione. Nella seconda parte, invece, ci siamo concentrati sull'analisi dei valori assunti dalle features per tentare di capire come mai 114 contratti non Ponzi fossero classificati come Ponzi.

Per prima cosa abbiamo ottenuto gli indici dei falsi positivi e dei falsi negativi con il codice illustrato in fig. 5.31. La lista `fp_list` conterrà gli indici dei falsi positivi presenti nel test set mentre `fn_list` gli indici dei falsi negativi.

```
# acquisico il numero dei contratti nel test set che sono falsi positivi o falsi negativi
list_pred = list(y_pred_thrs)
list_test = [item for sublist in y_test for item in sublist]

i = 0
fn_list, fp_list = [], []
while i < len(list_pred):
    if list_test[i] != list_pred[i]:
        if (list_test[i] == True) and (list_pred[i] == False):
            fn_list.append(i)
        elif (list_test[i] == False) and (list_pred[i] == True):
            fp_list.append(i)
    i+=1

print(fp_list)
print('false positive:: ', len(fp_list))
print()
print(fn_list)
print('false negative:: ', len(fn_list))
```

Fig. 5.31. Acquisizione degli indici dei contratti FP e FN.

Successivamente si calcolano le probabilità con cui i contratti del test set sono classificati come Ponzi o non Ponzi tramite la funzione `predict_proba()`. Questa restituisce una coppia di reali che rappresentano rispettivamente le probabilità di un contratto di essere non Ponzi o Ponzi (la somma di questi due valori farà 1). Adesso, prendendo in considerazione solo i FP e i FN, si inseriscono i valori calcolati con `predict_proba()` in due liste: `fp_coordinates` e `fn_coordinates`. Queste due liste avranno un numero di entry pari, rispettivamente, al numero di FP e di FN. Di seguito divideremo le coordinate dell'asse x (date dalla probabilità di un contratto di non essere Ponzi) da quelle dell'asse y (date dalla probabilità di un contratto di essere Ponzi) per poter tracciare più agevolmente il grafico bidimensionale che rappresenta i contratti mal classificati come punti. Tutto ciò è riportato in fig. 5.32.

```
coordinates = clf.predict_proba(X_test)
fp_coordinates = []
i = 0

while i < len(fp_list):
    fp_coordinates.append(coordinates[fp_list[i]])
    i+=1


fn_coordinates = []
i = 0

while i < len(fn_list):
    fn_coordinates.append(coordinates[fn_list[i]])
    i+=1


fp_coo_list = np.array([l.tolist() for l in fp_coordinates])
fn_coo_list = np.array([l.tolist() for l in fn_coordinates])

x_fp, y_fp = fp_coo_list.T
x_fn, y_fn = fn_coo_list.T
```

Fig. 5.32. Acquisizione delle probabilità per la classificazione.

Nel grafico in fig. 5.33, come ci si aspetta, i falsi negativi (identificati dai punti rossi) sono al di sotto del valore della soglia (0.422) calcolato con la curva di precison-recall, mentre i falsi positivi risultano al di sopra di questo valore.

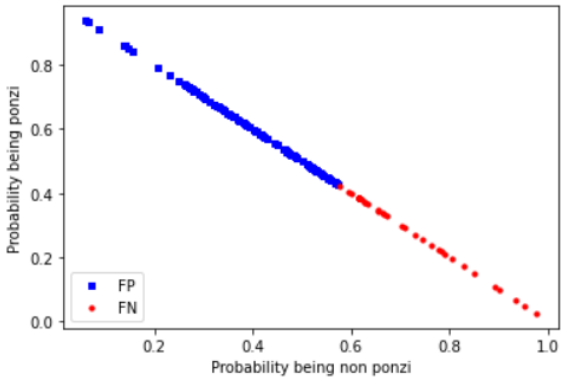


Fig. 5.33. Grafico della distribuzione dei contratti mal classificati.

Adesso si raccolgono e inseriscono i valori delle features dei falsi positivi e dei falsi negativi, utilizzando gli indici calcolati in fig. 5.31, in due dataframe `df_FP` e `df_FN` che sono successivamente scritti su due file csv: false_positive.csv e false_negative.csv fig. 5.34.

```
# Creo un dataframe contenente i Falsi positivi e i Falsi negativi, saranno successivamente scritti su di un file csv.
X_test_wrong_false_positive = []

for i in fp_list:
    X_test_wrong_false_positive.append(X_test_address[i, :])

df_FP = pd.DataFrame(data = X_test_wrong_false_positive, columns = features_names.insert(0, 'addresses'))

X_test_wrong_false_negative = []

for i in fn_list:
    X_test_wrong_false_negative.append(X_test_address[i, :])

df_FN = pd.DataFrame(data = X_test_wrong_false_negative, columns = features_names.insert(0, 'addresses'))

df_FN.to_csv(r'C:\Users\andre\Desktop\materiale tesi\settimana 16\dataset\cleaned+\fbeta\false negative.csv')
df_FP.to_csv(r'C:\Users\andre\Desktop\materiale tesi\settimana 16\dataset\cleaned+\fbeta\false positive.csv')
```

Fig. 5.34. Inserimento in dataframe dei falsi negativi e dei falsi positivi. Nelle ultime due righe si scrivono su di un file in formato csv.

Infine, è stato disegnato un istogramma per vedere dove ricadono i contratti in base alle probabilità calcolate con `predict_proba()`. Sull'asse delle ascisse sono disposte le probabilità dei contratti di essere uno schema Ponzi. Sull'asse delle ordinate è indicato il numero di contratti. Il grafico è riportato in fig. 5.35. Nella parte superiore del grafico, in blu, abbiamo i contratti che sappiamo non essere Ponzi. La maggior parte, infatti, ricade nei bins di sinistra, dove è più alta la probabilità che il contratto sia "safe".
Nella parte inferiore del grafico, in arancione, sono riportati i contratti che sappiamo essere Ponzi; anche qui, come ci si aspetta, la maggior parte sono contenuti nei bins con probabilità più alte di essere Ponzi.

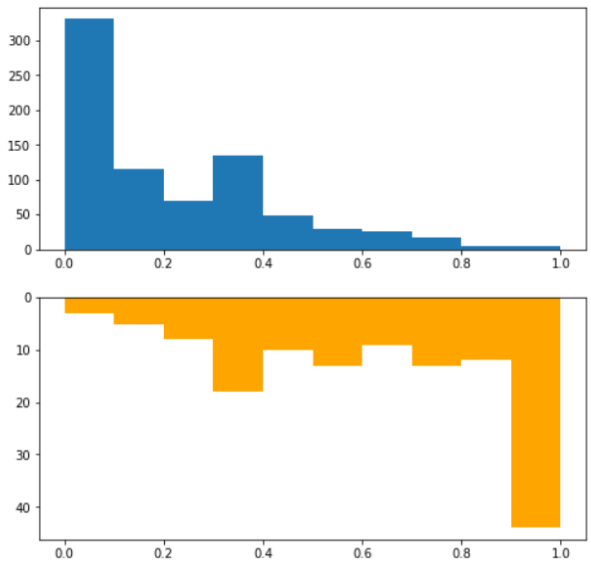


Fig. 5.35. Istogramma della classificazione dei contratti del test set.

Il codice per disegnare gli istogrammi di fig. 5.35 è riportato in fig. 5.36.

```
# acquisico le probabilita' dei contratti di essere Ponzi
prob_of_being_ponzi = coordinates[:, 1]
prob_of_being_ponzi
y_test_list = [item for sublist in y_test for item in sublist]
paired_hist_data = list(zip(y_test_list, prob_of_being_ponzi))


# preparo i dati per la creazione dell'istogramma
ponzi_hist = []
non_ponzi_hist = []

for t in paired_hist_data:
    if t[0] == 1:
        ponzi_hist.append(t[1])
    else:
        non_ponzi_hist.append(t[1])


# costruisco l'istogramma che mantiene le probabilita' delle classificazioni
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(211)
ax2 = fig.add_subplot(212)
bins = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]

ax.hist(non_ponzi_hist, bins = bins, align = 'mid')
ax2.hist(ponzi_hist, color = 'orange', bins = bins, align = 'mid')
ax2.invert_yaxis()
```

Fig. 5.36. Codice per la creazione degli istogrammi.

Per prima cosa si recuperano le probabilità che un contratto ha di essere classificato come Ponzi e si crea una lista di coppie, `paired_hist_data`, che, come primo valore, ha 0 o 1, a seconda che il contratto sia non Ponzi o Ponzi, e come secondo valore la probabilità di essere Ponzi. Adesso si preparano i dati per la creazione dell'istogramma separando `paired_hist_data` in due liste: una che contiene i soli contratti Ponzi e l'altra esclusivamente non Ponzi. Infine, si definiscono i bins. Ogni bins contiene un range di probabilità. Il primo avrà un range di probabilità che va da 0 a 0.1, il secondo da 0.1 a 0.2 e così via fino all'ultimo bins che conterrà le probabilità tra 0.9 e 1. Infine, sfruttiamo la libreria `matplotlip` per disegnare gli istogrammi.

In un altro script, pubblicato all'indirizzo: https://github.com/Lychett/tesi-smart-ponzi/tree/main/utility/script%20false%20positive, si analizzano più approfonditamente le cause per cui 114 contratti non Ponzi vengono classificati erroneamente.
Dopo aver importato le librerie necessarie, si caricano le informazioni dei contratti Ponzi, dei contratti non Ponzi e dei falsi positivi, inserendoli rispettivamente nei dataframe: `ponzi`, `non_ponzi`, `false_positive` (fig. 5.37).

```
# leggo le stats dei ponzi del mio dataset
ponzi = pd.read_csv(r'C:\Users\andre\Desktop\materiale tesi\settimana 16\nuovi script etherscan\risultati\ponzi_result.csv', header = 0)
features_names_ponzi = list(ponzi.columns) # ottengo i nomi delle features
fn_without_address_ponzi = features_names_ponzi[:] # copio il nome delle features
fn_without_address_ponzi.pop(0) # rimuovo il primo campo addresses, in quanto non e' una feature

# leggo le stats dei non ponzi del mio dataset
non_ponzi = pd.read_csv(r'C:\Users\andre\Desktop\materiale tesi\settimana 16\nuovi script etherscan\risultati\non_ponzi_result.csv', header = 0)
features_names_non_ponzi = list(non_ponzi.columns) # ottengo i nomi delle features
fn_without_address_non_ponzi = features_names_non_ponzi[:] # copio il nome delle features
fn_without_address_non_ponzi.pop(0) # rimuovo il primo campo addresses, in quanto non e' una feature

# leggo la lista dei falsi positivi: i non ponzi classificati come ponzi
false_positive = pd.read_csv(r'C:\Users\andre\Desktop\materiale tesi\settimana 16\dataset\cleaned+\fbeta\false positive.csv', header = 0)
features_names_false_positive = list(false_positive.columns)
pd.set_option('max_columns', None) # per poter stampare tutte le colonne
pd.set_option('max_rows', None)
```

Fig. 5.37. Codice per l'acquisizione dei dati dei contratti Ponzi, non Ponzi e falsi positivi.

Dopo aver rimosso dai dataframe le colonne e le righe che non contengono dati utili (ad esempio la colonna degli indirizzi), si esegue sui dataframe la funzione `discribe()` che restituisce le principali statistiche dalle features. Per velocizzare il confronto manteniamo solo le statistiche delle tre features più importanti: 'paid_one', 'known_rate' e 'tx_in'. Come si osserva da fig. 5.38, i falsi positivi hanno le statistiche delle features principali molto più simili ai contratti Ponzi rispetto ai non Ponzi.

	paid_one	known_rate	tx_in
count	114.000000	114.000000	114.000000
mean	0.499453	0.696098	180.508772
std	0.426872	0.441832	870.829604
min	0.000000	0.000000	2.000000
25%	0.000000	0.000000	4.000000
50%	0.500000	1.000000	11.000000
75%	1.000000	1.000000	100.750000
max	1.000000	1.000000	8996.000000

	paid_one	known_rate	tx_in
count	673.000000	673.000000	673.000000
mean	0.443041	0.645629	198.156018
std	0.404492	0.450212	716.665369
min	0.000000	0.000000	2.000000
25%	0.000000	0.000000	5.000000
50%	0.482800	1.000000	20.000000
75%	0.848500	1.000000	197.000000
max	1.000000	1.000000	9691.000000

	paid_one	known_rate	tx_in
count	3884.000000	3884.000000	3884.000000
mean	0.143661	0.248151	2135.774202
std	0.314830	0.414673	3677.714855
min	0.000000	0.000000	1.000000
25%	0.000000	0.000000	5.000000
50%	0.000000	0.000000	43.500000
75%	0.000000	0.500000	2005.250000
max	1.000000	1.000000	19485.000000

Fig. 5.38. Partendo da destra, il primo contiene i valori statistici dei FP, il secondo dei contratti Ponzi e il terzo dei contratti non Ponzi.

Infine, contiamo il numero di valori uguali che hanno le features dei falsi positivi rispetto alle features dei contratti Ponzi: per ogni FP e per ogni features, si confronta il valore della feature X coi valori assunti dalla feature X nei contratti Ponzi, contando quanti hanno lo stesso valore. Una volta terminato il ciclo si può notare che i falsi positivi tendono ad avere i valori delle features più importanti uguali alla maggior parte dei valori mantenuti dai contratti Ponzi.
In conclusione, possiamo affermare che la maggior parte dei contratti falsi positivi sono classificati erroneamente in quanto somigliano molto di più (a livello di valori delle features) a contratti Ponzi rispetto che a non Ponzi.

5.6 Analisi statistica

Quando si confrontano modelli di machine learning vorremmo capire se le differenze osservate sono statisticamente significative; quindi, che sia improbabile che queste differenze siano dovute al caso o al rumore nei dati. Per capirlo si utilizzano i test statistici.

Per definire un test statistico abbiamo bisogno di determinare un'ipotesi nulla.
L'ipotesi nulla indica che le performance di due modelli sono uguali, e che qualsiasi piccola perdita, o guadagno, osservato non è statisticamente significativo.
Oltre all'ipotesi nulla dobbiamo definire il p-value. Questo ci permette di capire se possiamo rifiutare o meno l'ipotesi nulla. Il p-value sarà calcolato dal test statistico effettuato. Se il p-value è inferiore a una determinata soglia, solitamente fissata a 0.05, possiamo rifiutare l'ipotesi nulla e dedurre che la differenza tra i modelli è statisticamente significativa, altrimenti, qualsiasi effetto osservato o differenza non è statisticamente rilevante.

In caso di modelli di classificazione binaria, il test statistico più appropriato è McNemar [11]. McNemar permette, infatti, di confrontare due modelli valutati sullo stesso test set. Nel nostro caso, il test set è costituito dai medesimi contratti in cui però cambiano le features.
Il test di McNemar è un test statistico per dati appaiati (paired). Nel contesto dei modelli di machine learning, possiamo usare il test di McNemar per confrontare la precisione sul test set di due modelli. Il test di McNemar è basato su una tabella di contingenza delle previsioni dei due modelli.
La tabella di contingenza è riportata in fig. 5.39. A differenza delle classiche matrici di confusione che si ottengono dai classificatori, dove abbiamo in ogni cella il numero di TN, FP, FN e TP, le tabelle di contingenza contengono il confronto tra due classificatori:

A: numero di predizioni che sia il modello1 che il modello2 effettuano correttamente;
B: numero di predizioni eseguite correttamente dal modello1 ma non dal modello2;
C: numero di predizioni eseguite correttamente dal modello2 ma non dal modello1;
D: numero di predizioni sbagliate da entrambi i modelli.

Il test controlla se c'è una differenza significativa tra le celle B e C. Se le celle hanno valori simili allora entrambi i modelli fanno errori in proporzione simili, ma su istanze diverse del test set. In questo caso, non si riesce a rifiutare l'ipotesi nulla,

dunque la differenza tra i due modelli non risulterà significativa.
Se queste celle hanno valori non simili allora i modelli non solo fanno errori diversi, ma, hanno anche una diversa proporzione relativa agli errori sul test set. In questo caso possiamo rifiutare l'ipotesi nulla.

	Model2 Correct	Model2 Incorrect
Model1 Correct	A	B
Model1 Incorrect	C	D

Fig. 5.39. Tabella di contingenza.

Nel test di McNemar, l'ipotesi nulla è che le probabilità p(B) e p(C) siano uguali, in pratica nessuno dei due modelli si comporta meglio dell'altro. Quindi, l'ipotesi alternativa è che le prestazioni dei due modelli siano diverse.
Tramite il test di McNemar si calcolano il p-value e il chi-squared (o chi-quadro), se il p-value è minore della soglia alfa (solitamente fissata a 0.05), allora si può rifiutare l'ipotesi nulla e concludere che le performance dei due modelli sono diverse, altrimenti l'ipotesi nulla non può essere rigettata.
Il chi-squared o χ^2 può essere calcolato in due modi:
• nella forma classica, proposta dall'ideatore del test di McNemar (Quinn McNemar) nel 1947, è ottenuto dal seguente rapporto: $\chi^2 = \frac{(B-C)^2}{(B+C)}$;

• nella forma con correzione continua, proposta da Edwards l'anno seguente: $\chi^2 = \frac{(|B-C|-1)^2}{(B+C)}$;

dove B e C indicano i valori delle celle nella tabella di contingenza in fig. 5.39.

Il p-value invece viene calcolato dalla seguente formula: $p = 2\sum_{i=b}^{n} \binom{n}{i} 0.5^i (1-0.5)^{n-i}$,

A livello di implementazione del test di McNemar, sono state usate le API sviluppate da rasbt per Python, consultabili all'indirizzo: https://rasbt.github.io/mlxtend/user_guide/evaluate/mcnemar/ ([12]). Di seguito si illustra il codice.
In fig. 5.40 si importano le librerie e si leggono i dati necessari alla creazione della matrice di contingenza. La libreria più interessante è `mlxtend.evaluate` che contiene vari test statistici sviluppati per Python tra cui McNemar che è appunto quello che importiamo.
Successivamente si caricano i dati da confrontare. Per prima cosa sono reperiti i dati ground truth cioè il file contenente i target del test set, chiamato y_test_list.txt, poi si ottengono le predizioni dei due modelli da confrontare: il nostro modello

vincitore (5.4.4 #2) ed il modello 5.4.2. #2. Per rendere il confronto più equo, infatti, abbiamo paragonato il nostro modello a quello in cui sono stati adottati gli stessi iperparametri, utilizzando le predizioni ottenute a seguito del tuning della soglia di decisione.
Un paragone con il modello 5.4.2 #1 avrebbe avuto poco senso dato che la differenza è chiaramente apprezzabile già dalla sua matrice di confusione.

```
import numpy as np
import pandas as pd
from mlxtend.evaluate import mcnemar, mcnemar_table


# acquisico le classificazioni corrette, le ho inserite nella cartella contenente il dataset di exploiting blockchain
with open(r'C:\Users\andre\Desktop\materiale tesi\settimana 16\dataset\exploiting blockchain - cleaned\y_test_list.txt') as f:
    test = f.read().splitlines()

integer_map = map(int, test)
y_test_list = list(integer_map)

# acquisico le predizioni del mio modello migliore fatte sul test set
with open(r'C:\Users\andre\Desktop\materiale tesi\settimana 16\dataset\cleaned+\fbeta\y_pred_list.txt') as f:
    fbeta_pred7 = f.read().splitlines()

integer_map = map(int, fbeta_pred7)
y_fbeta_pred_list = list(integer_map)

# acquisico le predizioni del modello della sottosezione 5.4.2 #2 fatte sul test set
with open(r'C:\Users\andre\Desktop\materiale tesi\settimana 16\dataset\exploiting blockchain - cleaned\y_pred_list.txt') as f:
    pred3 = f.read().splitlines()

integer_map = map(int, pred3)
y_pred3_list = list(integer_map)
```

Fig. 5.40. Codice per l'acquisizione dei dati da confrontare.

In fig. 5.41, si ha il codice per creare la tabella di McNemar, per calcolare il χ^2 e il p-value. Per costruire la tabella di McNemar si utilizza la funzione `mcnemar_table()` a cui si passano come parametri i dati recuperati in fig. 5.40. Leggendo i risultati della tabella abbiamo 665 contratti che entrambi i modelli classificano correttamente, 97 che il nostro modello classifica bene e il 5.4.2 #2 classifica male (viceversa per 30) e infine 120 contratti che vengono classificati in maniera errata da entrambi.

```
    # creo la tabella di McNemar
    tb = mcnemar_table(y_target = np.array(y_test_list),
                       y_model1  = np.array(y_fbeta_pred_list),
                       y_model2 = np.array(y_pred3_list))

    print(tb)
```

```
[[665  97]
 [ 30 120]]
```

```
    chi2, p = mcnemar(ary=tb, corrected=True)
    print('chi-squared:', chi2)
    print('p-value:', p)
```

```
chi-squared: 34.2992125984252
p-value: 4.72572930207707e-09
```

Fig. 5.41 Codice e risultati ottenuti dal confronto dei due modelli.

Nella seconda parte si calcolano i valori del χ^2 e il p-value, ottenendo come valore di quest'ultimo 4.725×10^{-9} che è nettamente inferiore alla soglia alfa fissata a 0.05, perciò possiamo rifiutare l'ipotesi nulla, asserendo che le prestazioni dei due classificatori sul test set sono diverse, in particolare che il nostro modello è migliore.

6. Conclusioni e sviluppi futuri

Con lo sviluppo della blockchain, gli schemi Ponzi sono approdati anche su questa nuova piattaforma. In questa tesi sono state utilizzate tecniche di machine learning per il rilevamento di questi tipi di contratti. Il primo contributo del lavoro è stato la costruzione di un dataset uniforme contenente circa 900 contratti Ponzi e 4700 contratti non Ponzi. Questo dataset è stato costruito ampliando e completando tre dataset pubblicati dagli articoli [2], [3] e [4]. A seguito della costruzione di questo insieme di dati abbiamo implementato un classificatore binario in Python per il riconoscimento dei contratti Ponzi basato sulle account features. Il classificatore è stato ottenuto a seguito di un processo iterativo di addestramento e valutazione usando differenti algoritmi di machine learning, il cui obiettivo finale era di massimizzarne l'accuratezza. Il modello ottenuto, grazie all'introduzione di nuove features, all'uso di iperparametri adeguati durante il training e alla creazione di un dataset più grande con cui allenarlo, è risultato migliore rispetto a quelli implementati dall'articolo [3].

In futuro sarà possibile estendere il dataset con altri contratti, migliorando così il modello. Il numero degli smart contract, infatti, continua ad aumentare giorno dopo giorno, è possibile dunque aumentare il nostro insieme di dati controllando manualmente i codici sorgente. Con più contratti Ponzi presenti nel dataset sarà possibile sviluppare un modello più accurato e preciso per il loro rilevamento.
Un altro sviluppo di questo lavoro potrà essere l'implementazione di un classificatore basato anche su features estratte dal bytecode piuttosto che solo sull'account features. È proprio per questo che oltre a quest'ultime, durante la costruzione del dataset, sono stati reperiti pure i bytecodes degli smart contract considerati, rimuovendo eventuali duplicati. Il limite principale di un classificatore basato sulle account features è il non riuscire a predire se un contratto sia Ponzi o non Ponzi se questo non ha ricevuto o effettuato transazioni. Di contro, un classificatore basato anche su features estratte dal bytecode, non ha bisogno che un contratto sia stato contattato per poterlo individuare come schema Ponzi.

7. Bibliografia e linkografia

[1] Massimo Bartoletti, Barbara Pes, Sergio Serusi: Data mining for detecting Bitcoin Ponzi schemes. CoRR abs/1803.00646 (2018)

[2] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, Roberto Saia: Dissecting Ponzi schemes on Ethereum: Identification, analysis, and impact. Future Gener. Comput. Syst. 102: 259-277 (2020)

[3] Weili Chen, Zibin Zheng, Edith Cheuk-Han Ngai, Peilin Zheng, Yuren Zhou: Exploiting Blockchain Data to Detect Smart Ponzi Schemes on Ethereum. IEEE Access 7: 37575-37586 (2019)

[4] Weimin Chen, Xinran Li, Yuting Sui, Ningyu He, Haoyu Wang, Lei Wu, Xiapu Luo: SADPonzi: Detecting and Characterizing Ponzi Schemes in Ethereum Smart Contracts. Proc. ACM Meas. Anal. Comput. Syst. 5(2): 26:1-26:30 (2021)

[5] https://it.wikipedia.org/wiki/Ethereum

[6] https://info.etherscan.com/understanding-an-ethereum-transaction/

[7] https://ethereum.org/en/wallets/

[8] https://machinelearningmastery.com/threshold-moving-for-imbalanced-classification/

[9] https://www.sec.gov/spotlight/enf-actions-ponzi.shtml

[10] https://machinelearningmastery.com/tour-of-evaluation-metrics-for-imbalanced-classification/

[11] McNemar, Quinn, 1947. "Note on the sampling error of the difference between correlated proportions or percentages". Psychometrika. 12 (2): 153–157.

[12] https://rasbt.github.io/mlxtend/user_guide/evaluate/mcnemar/

[13] https://ethereum.github.io/yellowpaper/paper.pdf

[14] https://towardsdatascience.com/the-5-classification-evaluation-metrics-you-must-know-aa97784ff226

[15] https://www.analyticsvidhya.com/blog/2021/07/metrics-to-evaluate-your-classification-model-to-take-the-right-decisions/

8. Ringraziamenti

Un sentito ringraziamento al Professor Pierpaolo Degano, al Dottor Letterio Galletta e al Dottor Fabio Pinelli, relatori di questa tesi di Laurea, per avermi guidato e supportato nella fase più importante del mio percorso accademico.
Un grande ringraziamento ai miei familiari che con il loro sostegno mi hanno aiutato ad arrivare alla conclusione di questo mio ciclo di studi.
Grazie, infine, agli amici e ai compagni di corso con cui ho condiviso questi tre anni rendendoli più leggeri e spensierati.