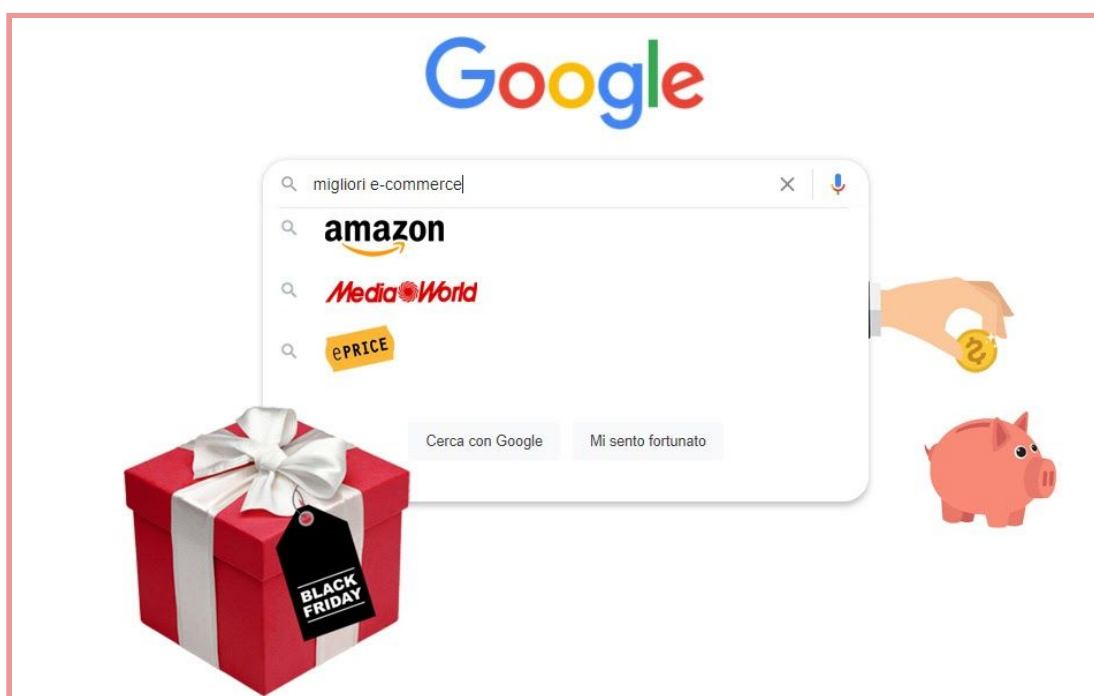


UNIVERSITÀ DEGLI STUDI DI SALERNO
Corso di Laurea Magistrale in Informatica



**Analisi dell'andamento dei prezzi sui siti di e-commerce durante i periodi
promozionali**



Docente

Prof. Delfina Malandrino

Partecipanti

Offertucci Mario

Donnarumma Antonio

Anno accademico 2020/2021

Introduzione	3
E-commerce scelti	3
Fasi del progetto	3
Strumenti ed architettura	4
Raccolta dei prezzi	4
Aggiunta e modifica dei prodotti	4
Analisi dei dati	5
Visualizzazione grafica	5
Il database	6
Schema architetturale	7
Sviluppo	8
Analisi dei siti	8
Prelievo dei dati	9
Generazione grafici	17
Bot per la gestione dei prodotti da analizzare	19
L'interfaccia grafica	24
Pannello iniziale	25
Pannello visualizzazione grafici	26
Generazione del report	27
Pannello generazione grafici	28
Pannello status generazione grafici	29
Conclusioni	30
Amazon	30
Grafici dei prodotti selezionati	31
Eprice	34
Grafici dei prodotti selezionati	35
Mediaworld	39
Grafici dei prodotti selezionati	39
Conclusioni finali	43

Introduzione

Il progetto **e-commerce Analysis** si occupa del verificare il reale sconto offerto durante i periodi di promozione. L'obiettivo di questa ricerca è stato quello di monitorare l'andamento del prezzo di vari prodotti, appartenenti a diverse categorie (elettronica, elettrodomestici, ecc), nell'arco di un mese e mezzo, e successivamente sono stati messi a confronto per verificare se il prezzo offerto durante il black friday fosse effettivamente uno sconto o no.

E-commerce scelti

Considerando l'ingente numero di e-commerce oggi presenti sul mercato è stato necessario prenderne in esame soltanto un piccolo sottoinsieme tra quelli ritenuti più importanti ed influenti nel mercato, al momento. Gli e-commerce analizzati sono:

- **Amazon:** E' probabilmente l'e-commerce più importante al momento, in grado di gestire enormi quantità di ordini offrendo prezzi quasi sempre più convenienti rispetto alla concorrenza. E' il principale promotore della campagna "Black Friday".
- **E-price:** E' un e-commerce di modeste dimensioni italiano che offre spesso prezzi competitivi ed è attualmente tra i primi venditori elettronici in Italia.
- **Mediaworld:** Anche questo è un e-commerce di discrete dimensioni in Italia, ed è l'unico tra quelli presi in esame ad avere anche dei negozi fisici attraverso cui effettuare vendite a prezzi, alcune volte, più convenienti rispetto al mercato.

Fasi del progetto

Le principali fasi in cui è stato suddiviso il progetto sono le seguenti:

1. **Analisi dei siti in questione:** durante questa fase è stata eseguita un'analisi dei vari siti di e-commerce presi di mira con lo scopo di identificare i valori di interesse, quali per esempio il prezzo, eventuali sconti o offerte, la disponibilità del prodotto e così via;
2. **Prelievo dei dati:** La fase successiva è stata incentrata sulla realizzazione di script automatizzati che consentissero il prelevamento di tutti i dati identificati nella fase precedente e la loro memorizzazione in una base di dati;
3. **Generazione grafici:** Una volta che i dati sono stati ottenuti, l'ultimo passo è stato quello di organizzarli in modo tale da avere una visione d'insieme più rapida e che consentisse di determinare in maniera più veloce lo scopo della ricerca per ogni prodotto.

Strumenti ed architettura

Vediamo adesso per ognuna delle componenti dell'architettura i relativi strumenti che ne hanno permesso la realizzazione. E' possibile scomporre l'architettura in quattro parti, una relativa alla raccolta prezzi dei prodotti, una relativa all'aggiunta e modifica dei prodotti, una relativa alle analisi dei dati ottenuti ed infine una per la visualizzazione grafica.

Raccolta dei prezzi

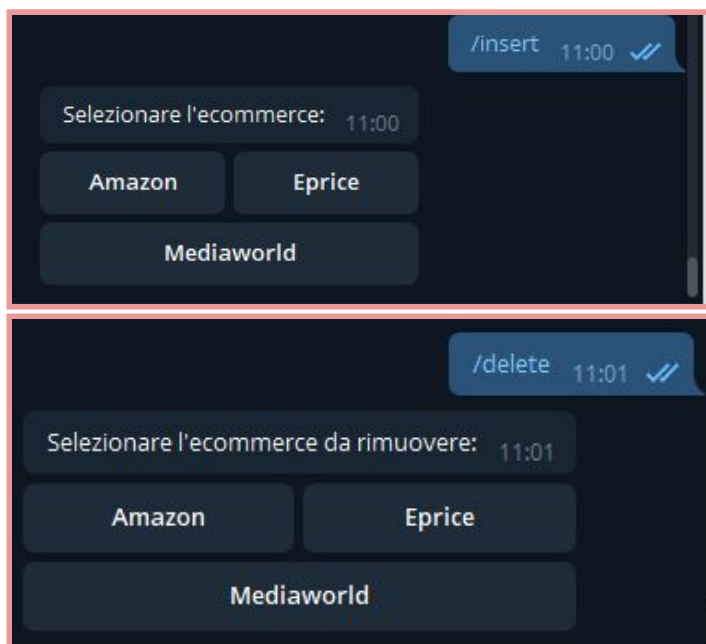
Questa operazione, eseguita due volte al giorno (a mezzogiorno e mezzanotte) ha permesso l'ottenimento dei prezzi dei prodotti selezionati in maniera del tutto automatica attraverso l'utilizzo di un algoritmo di scraping realizzato mediante **Python** (analizzato in seguito). L'architettura prevede quindi, a livello fisico, un **Raspberry** che, facendo uso di crontab (che permette di pianificare l'esecuzione di appositi comandi in UNIX), esegue il software realizzato appositamente per ottenere i dati, per poi memorizzarli all'interno di una base di dati relazionale **PostgreSQL**.

```
0 */12 * * * script.py
```

La tabella mostra il comando per il crontab che effettua l'esecuzione dello script due volte al giorno, ogni 12 ore tutti i giorni.

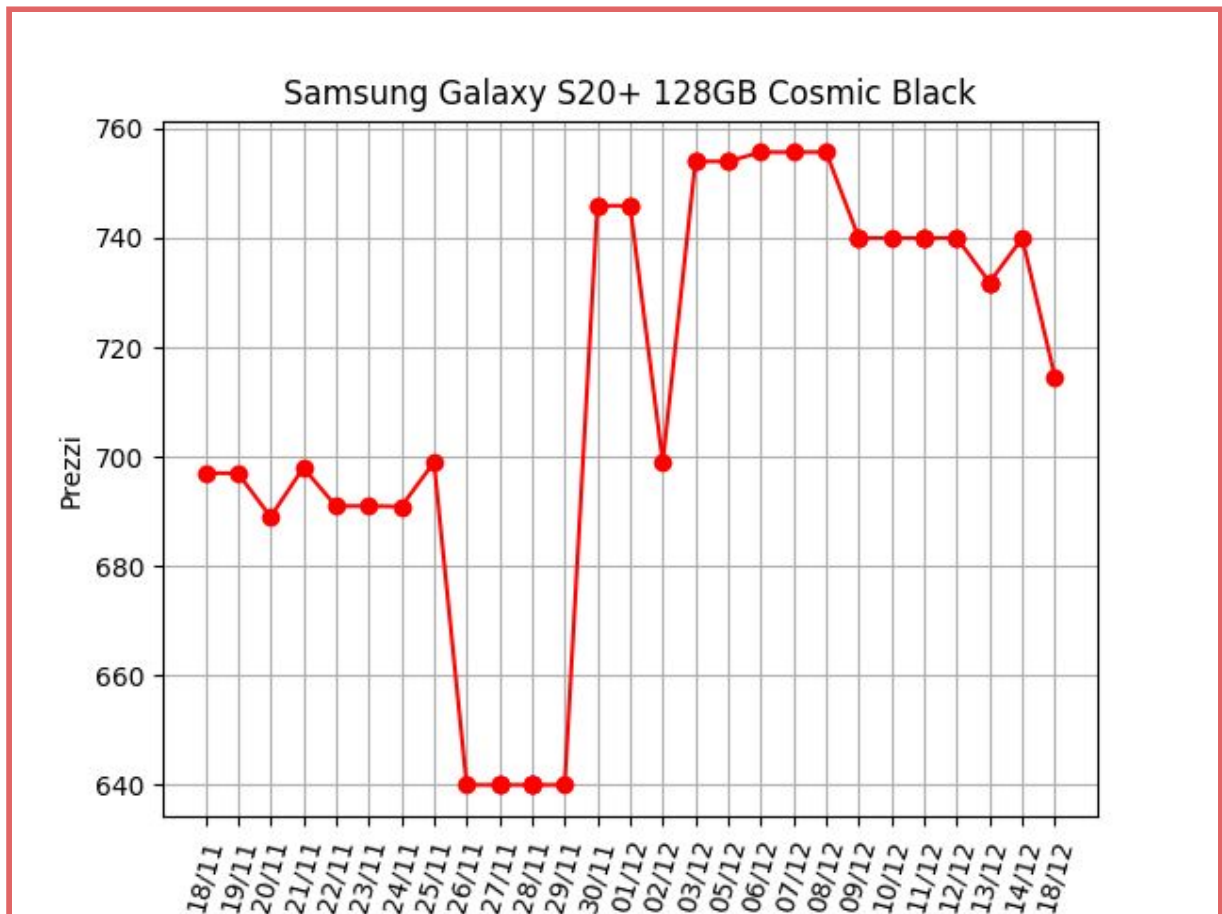
Aggiunta e modifica dei prodotti

I prodotti da analizzare sono mantenuti all'interno di un file (il nome e il relativo URL) ognuno associato al rispettivo e-commerce, presente all'interno del Raspberry. Al fine di evitare, per ogni aggiunta di prodotto l'accesso al Raspberry, è stato realizzato un Telegram bot (mediante l'omonima libreria) che ci ha permesso sia l'aggiunta che l'eliminazione (ma anche il controllo di tutti i prodotti attualmente sotto 'analisi') senza dover effettuare ogni volta operazioni SSH con il dispositivo.



Analisi dei dati

Per poter raccogliere i dati, attraverso la libreria **selectorlib** è stato possibile definire dei files in formato *yaml* che hanno permesso l'estrazione dalla pagina del prodotto (attraverso la definizione di tag *css*) delle informazioni necessarie, per poi procedere alla memorizzazione. Una volta raccolti i dati, prima della realizzazione dei report (utili per l'interfaccia utente), sono stati generati, mediante la libreria **matplotlib**, i grafici relativi ai prodotti di ciascun e-commerce.



(esempio di grafico)

Visualizzazione grafica

La visualizzazione grafica dei risultati è stata realizzata mediante la libreria **PySimpleGui**, che permette la realizzazione di interfacce in maniera davvero semplice e veloce permettendo la definizione di ognuno dei pannelli attraverso delle normali liste contenenti oggetti grafici che saranno poi posizionati seguendo una griglia definibile in base all'ordine degli oggetti grafici posizionati all'interno delle liste (in seguito maggiori dettagli).

(esempio interfaccia utente)

Il database

Il database su cui si poggia l'applicazione è realizzato mediante il DBMS **PostgreSQL**, accessibile in rete locale che ha permesso la memorizzazione attraverso tre tabelle, dei prezzi ottenuti giorno per giorno, di ogni prodotto associato a ciascun e-commerce preso in analisi. E' composto da tre tabelle, *prodottiamazon*, *prodottieprice* e *prodottimediaworld* che, nonostante presentino la stessa struttura, sono state separate fisicamente al fine di semplificare la lettura, in previsione dell'ingente mole di dati che si stava andando a memorizzare.

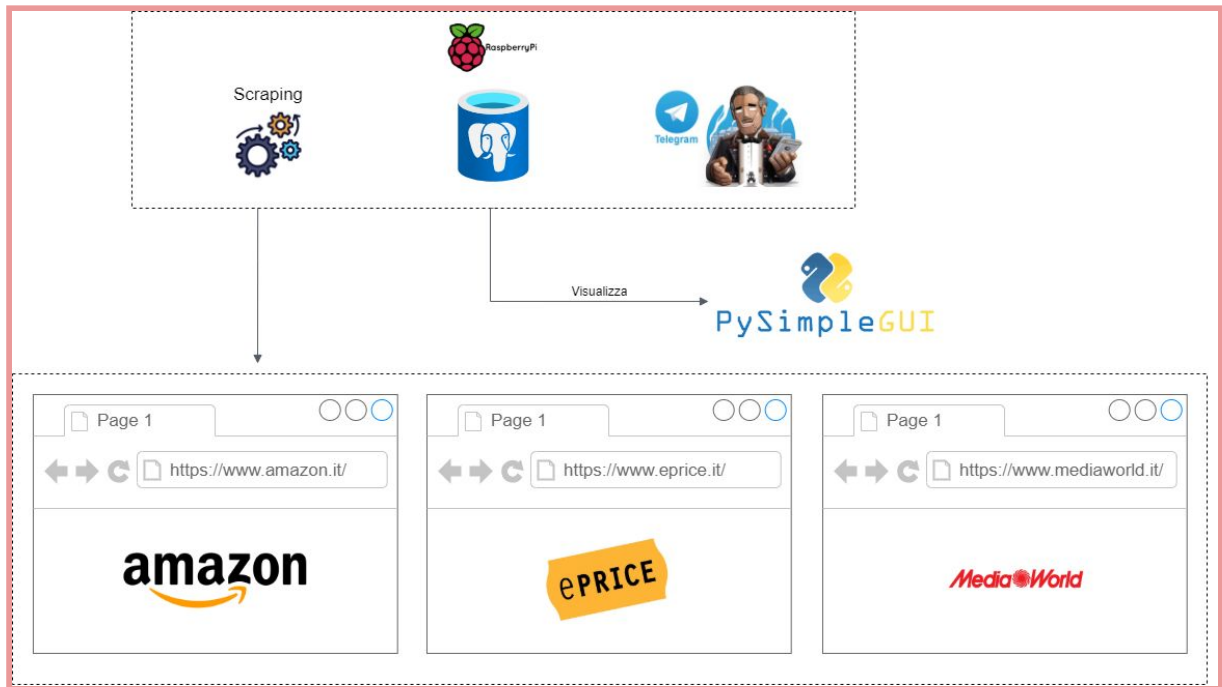
Nome del campo	Tipo
ID	integer
Nome	varchar(150)
URL	varchar(2000)
Prezzo	double
Data	date

Descrizione dei campi:

- **ID:** Identificativo univoco progressivo utilizzato come chiave primaria di ogni record del database.
- **Nome:** Nome del prodotto che si intende analizzare.
- **URL:** Indirizzo web del prodotto (di lunghezza 2000 poiché è circa la grandezza massima di una URL)

- **Prezzo:** Costo del prodotto, utilizza al più due grandezze per i decimali
- **Data:** Orario del rilevamento secondo il formato AAAA/MM/GG

Schema architetturale



In conclusione, il centro dell'architettura è rappresentato dal Raspberry che effettua lo scraping ogni 12 ore ed esegue il Telegram Bot che permette la modifica del file contenente i nomi dei prodotti. Dal lato dell'elaborazione (effettuabile su un qualsiasi computer con l'applicativo installato), si utilizza la base di dati messa in rete dal Raspberry per generare report e grafici che sono poi utilizzati dall'interfaccia utente per presentare i risultati.

Sviluppo

Analisi dei siti

La prima fase è stata quella di analizzare i siti scelti per il monitoraggio, con lo scopo di determinare i valori da far leggere allo script. In particolare i valori che sono stati letti sono relativi innanzitutto al prezzo, ma anche ad eventuali sconti già presenti e alla disponibilità del prodotto.

Utilizzando lo strumento "ispeziona elemento", fornito da tutti i browser, si è risalito alle classi o agli id che contenevano tali informazioni.

The screenshot shows the Amazon.it product page for a D-Link DGS-108 Switch. The product title is "D-Link DGS-108 Switch 8 Porte Gigabit, Struttura in Metallo, Nero/Antracite". The price is 32,31 € (19% discount from 39,99 €). The "ispeziona elemento" tool is open, showing the HTML structure of the product title. The tool indicates that the product title is contained within a `` element. The tool also shows the dimensions of the element (342.05 x 57) and the font (19px "Amazon Ember", Arial, sans-serif).

```
<span id="productTitle" class="a-size-large product-title-word-break"> == $0
"

D-Link DGS-108 Switch 8 Porte Gigabit, Struttura in Metallo, Nero/Antracite

"
</span>
```


Lo stesso procedimento è stato eseguito per tutti i valori di interesse e per tutti gli e-commerce scelti.

Prelievo dei dati

Una volta preso nota degli id e delle classi da utilizzare durante lo scraping è stato realizzato lo script che automatizza la loro lettura e il loro salvataggio. Per l'adempimento di tale obiettivo è stato utilizzato il linguaggio di programmazione Python.

Durante questa fase sono emerse particolari informazioni che permettono anche di poter classificare i tre siti campione in termini di efficienza e scalabilità.

Nello specifico, Amazon si impegna molto a impedire lo scraping dei dati dal sito. Per poterli leggere senza problemi è stato necessario simulare quanto più possibile il comportamento umano, e quindi aspettare una determinata quantità di tempo (10 secondi) prima di eseguire la successiva richiesta. Amazon, più degli altri, è particolarmente attenta al bloccaggio di queste richieste, per cui è stato necessario ricorrere a diversi user agents. Questo perché, anche se le richieste vengono eseguite a distanza di 10 secondi l'una dall'altra, ma tutte con lo stesso user agent, Amazon riesce a capire che l'utente non è un umano. Durante lo scraping delle pagine Amazon non è mai capitato di ricevere un errore dovuto alle troppe richieste che il sito stava ricevendo, nemmeno nel periodo del black friday quando l'utenza era più concentrata negli acquisti.

Eprice invece non attua nessuna politica per impedire questi comportamenti, e quindi è stato possibile eseguire le richieste una dopo l'altra senza aspettare neanche un secondo. In questo caso però nei periodi di picco sono stati ricevuti, diverse volte, errori 404. Questi due fattori sottolineano un e-commerce evidentemente meno organizzato del colosso Amazon, e che ha bisogno di fare ancora molta strada se vuole arrivare a competere con esso.

Mediaworld è risultato sicuramente il peggiore di tutti da questo punto di vista. Nonostante anche lui provi a bloccare le richieste che arrivano da "robot", il suo meccanismo è facilmente aggirabile attendendo, anche in questo caso, una decina di secondi. A differenza di Amazon però non è stato necessario utilizzare diversi user agents. Anche in questo caso è stata ricevuta una quantità inquantificabile di pagine di errore, dovute al sovraccarico del sito, il quale quindi non è per niente scalabile (come Eprice). Inoltre, molto spesso, durante il periodo di monitoraggio il sito è stato posto svariato volte in manutenzione. Tuttavia, in situazioni critiche, nelle quali c'era una grossa affluenza, si è comportato meglio di E-price perché non ha restituito errori 404, ma sono state utilizzate delle code di attesa (10 minuti).

Da queste caratteristiche si può facilmente notare la natura per niente scalabile anche di questo sito.

Lo scraping viene eseguito tramite dei file di configurazione in formato *yaml*. Di seguito un esempio per Amazon:

```
name:
  css: '#productTitle'
  type: Text
price:
  css: '#priceblock_ourprice'
  type: Text
price_deal:
  css: '#priceblock_dealprice'
  type: Text
price_promo:
  css: '#priceblock_pospromoprice'
  type: Text
price_not_available:
  css: '.availability'
  type: Text
short_description:
  css: '#featurebullets_feature_div'
  type: Text
images:
  css: '.imgTagWrapper img'
  type: Attribute
  attribute: data-a-dynamic-image
rating:
  css: span.arp-rating-out-of-text
  type: Text
number_of_reviews:
  css: 'a.a-link-normal h2'
  type: Text
```

```
variants:
  css: 'form.a-section li'
  multiple: true
  type: Text
  children:
    name:
      css: ""
      type: Attribute
      attribute: title
    asin:
      css: ""
      type: Attribute
      attribute: data-defaultasin
product_description:
  css: '#productDescription'
  type: Text
sales_rank:
  css: 'li#SalesRank'
  type: Text
link_to_all_reviews:
  css: 'div.card-padding a.a-link-emphasis'
  type: Link
```

La lista di prodotti di cui fare lo scraping invece viene letta da semplici file di testo. Esempio per Amazon:

```
https://www.amazon.it/dp/B08DR5JV8|%|SAMSUNG Galaxy Z Flip 5G Nero
https://www.amazon.it/dp/B079QM5GL9|%|iRobot Roomba 671 aspirapolvere
https://www.amazon.it/dp/B015JIAGE2|%|POLAROID Fotocamera Istantanea Snap Touch Azzurro
https://www.amazon.it/dp/B08BPK5QR4|%|OnePlus 8 NORD Onyx Grey 128GB
https://www.amazon.it/dp/B08D6NPCH7|%|HUAWEI MatePad T 10 Blue
https://www.amazon.it/dp/B08L5PP2FX|%|Apple iPhone 12 Pro blu Pacifico(128GB)
https://www.amazon.it/dp/B08L5Q2H5R|%|Apple iPhone 12 bianco (64GB)
https://www.amazon.it/dp/B08L5SMB8S|%|Apple iPhone 12 mini RED (64GB)
https://www.amazon.it/dp/B07XS2ND7L|%|Apple iPhone 11 Pro Grigio Siderale (64GB)
https://www.amazon.it/dp/B07XS52RNN|%|Apple iPhone 11 Pro Max Verde Notte (64GB)
https://www.amazon.it/dp/B07XS4J1XS|%|Apple iPhone 11 Viola (64GB)
https://www.amazon.it/dp/B0863SD8G3|%|Apple iPad Pro Argento (11", Wi-Fi, 128GB)
https://www.amazon.it/dp/B0863RPSHB|%|Apple iPad Pro Grigio siderale (12,9", Wi-Fi, 128GB)
https://www.amazon.it/dp/B08J6KQKCZ|%|Apple iPad Air Celeste (10,9", Wi-Fi, 64GB)
```

Il carattere `|%|` è un carattere speciale di separazione che serve per separare l'url del prodotto dal suo nome.

Lo script Python che si occupa dello scraping degli e-commerce fa utilizzo di una classe astratta *GenericScraper*. È stata scelta una classe astratta perché c'era la necessità di inserire alcune caratteristiche comuni a tutti gli scraper, che potessero eventualmente essere cambiate.

```
class GenericScraper:
    __metaclass__ = ABCMeta
    headers = {}
    maximum_request = 3
    extractor_file = ''
    input_file = ''
    delay_time = 10
    richieste_effettuate = 0
    request = None

    user_agents = [
        'Mozilla/5.0 (Linux; Android 8.0.0; SM-G960F Build/R16NW) AppleWebKit/537.36
        'Mozilla/5.0 (Linux; Android 7.0; SM-G892A Build/NRD90M; wv) AppleWebKit/537
        'Mozilla/5.0 (Linux; Android 7.0; SM-G930VC Build/NRD90M; wv) AppleWebKit/53
        'Mozilla/5.0 (Linux; Android 6.0.1; SM-G935S Build/MMB29K; wv) AppleWebKit/5
        'Mozilla/5.0 (Linux; Android 6.0.1; SM-G920V Build/MMB29K) AppleWebKit/537.3
        'Mozilla/5.0 (Linux; Android 5.1.1; SM-G928X Build/LMY47X) AppleWebKit/537.3
        'Mozilla/5.0 (Linux; Android 6.0.1; Nexus 6P Build/MMB29P) AppleWebKit/537.3
        'Mozilla/5.0 (Linux; Android 7.1.1; G8231 Build/41.2.A.0.219; wv) AppleWebKi
        'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_4) AppleWebKit/537.36 (KHTML,
        'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like G
    ]
```

Tutti gli scraper devono avere degli header, necessari per poter eseguire la richiesta. La variabile *maximum_request* indica il numero massimo di richieste da eseguire in caso di errori: se viene eseguita una richiesta, ma questa fallisce, si riprova al massimo per *maximum_request* volte.

extractor_file e *input_file* indicano rispettivamente il path all'interno del quale sono contenuti il file *yaml*, che serve per fare lo scraping, e il file che contiene gli url dei prodotti da analizzare.

delay_time indica il tempo che deve intercorrere tra una richiesta e un'altra e *richieste_effettuate* indica il numero di richieste che sono state effettuate.

user_agents è la lista di user agents dalla quale vengono scelti casualmente prima di eseguire la richiesta.

```
self.headers = {
    'dnt': '1',
    'upgrade-insecure-requests': '1',
    'user-agent': self.user_agents[randrange(self.user_agents.__len__())],
    'accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9',
    'sec-fetch-site': 'same-origin',
    'sec-fetch-mode': 'navigate',
    'sec-fetch-user': '?1',
    'sec-fetch-dest': 'document',
    'referer': 'https://www.amazon.com/',
    'accept-language': 'it-IT,en-GB,en-US;q=0.9,en;q=0.8',
}
```

Questa classe espone i metodi astratti *get_offers* e *getScraperName* i quali restituiscono rispettivamente tutte le offerte disponibili e il nome dello scraper utilizzato, e devono quindi essere riscritti da ogni classe che la estende.

```
@abstractmethod
def get_offers(self) -> dict: raise Exception("NotImplementedException")
```

```
@abstractmethod
def getScraperName(self):
    return "generic scraper"
```

Sono inoltre disponibili vari metodi (non astratti) che si occupano di fare la richiesta ad un url letto dal file e lo scraping, indipendentemente dal tipo di ecommerce.

```

def scrape(self, prodotti: list) -> list:
    i = 0
    result = []

    for prodotto in prodotti:
        # Eseguo la richiesta per prelevare i dati
        # request contiene la risposta
        self.request = self.makeRequest(prodotto.url)

        # Se viene eseguito qualche redirect strano chiama la funzione onRedirect
        if self.request is not None and prodotto.url != self.request.url:
            # Se onRedirect restituisce True si continua la richiesta
            if self.onRedirect():
                result = self.continuaRichiesta(prodotto, i, prodotti, result)
            else:
                # Altrimenti si imposta come valore del prodotto -1 e si continua con un nuovo prodotto
                prodotto.prezzo = -1
                result.append(prodotto)
                print(f"[{self.getScraperName().upper()} - REDIRECT ERROR]: {prodotto.nome}, {prodotto.url}, {prodotto.prezzo}", sep='')
        else:
            result = self.continuaRichiesta(prodotto, i, prodotti, result)

    return result

```

Il metodo *scrape* esegue la richiesta all'url del prodotto e controlla solamente se viene eseguito un redirect. Se questo viene eseguito la richiesta continua (chiamando il metodo *continuaRichiesta*) solamente se il metodo *onRedirect* restituisce *True*.

Tale meccanismo è necessario nel caso di Mediaworld, il quale esegue un redirect nel caso in cui il prodotto che si vuole analizzare non è disponibile. Se venisse eseguito verrebbero letti valori sbagliati di altri prodotti e quindi la richiesta si deve fermare. La classe *MediaworldScraper* riscrive il metodo *onRedirect* che semplicemente restituisce *False*.

```

'''Restituendo False la richiesta non continua e viene assegnato come valore al prodotto -1'''
def onRedirect(self):
    return False

```

Il metodo *continuaRichiesta* si occupa di rieseguire la richiesta, fino ad un massimo di *maximum_request* volte se questa fallisce. Se la richiesta va a buon fine viene utilizzato il file *yaml* per convertire i dati in oggetto, il quale verrà poi aggiunto ad una lista che verrà restituita al chiamante.


```

def continuaRichiesta(self, prodotto, i, prodotti, result):
    # print('status: ', self.request.status_code, " - ", prodotto.url)
    # Controllo errori
    while self.request.status_code != 200 and self.richieste_effettuate < self.maximum_request:
        self.richieste_effettuate += 1
        self.waitRequest(self.richieste_effettuate)
        self.request = self.makeRequest(prodotto.url)
    if self.request is not None and self.request.status_code == 200:
        # La richiesta è andata bene
        self.requestOk()
        # Crea l'estrattore per fare webscraps
        extractor = Extractor.from_yaml_file(self.extractor_file)
        # val rappresenta i prodotti letti dalla pagina (per i prodotti multipli è un dizionario)
        # print('REQUEST: ', request.text)
        val = extractor.extract(self.request.text)
        # Controllo se il prodotto è disponibile
        available = self.isAvailable(val)
        # Se il prodotto è disponibile
        if available:
            # Leggo il prezzo letto
            price = self.getPrice(val)
            # Rimuovo eventuali caratteri diversi dai numeri che possono esserci all'interno, compreso il simbolo €
            price = self.fixPrice(price)
            try:
                prodotto.prezzo = float(price)
            except:
                print("Eccezione")
        else:
            # Se il prodotto non è disponibile
            prodotto.prezzo = -1
        print(f"[{self.getScraperName().upper()}]", end="")
        if available:
            print(" - DISPONIBILE] ", end="")
        else:
            print(" - NON DISPONIBILE] ", end="")
        print(f"{prodotto.nome} - {prodotto.url} = € {prodotto.prezzo} ")
        if i < prodotti.__len__() - 1:
            time.sleep(self.delay_time)
        i += 1
        result.append(prodotto)
    return result

```

Il metodo *waitRequest* esegue solamente il log, ma alcune classi possono riscriverlo per eseguire altre operazioni.

```

def waitRequest(self, numeroRichiesta: int):
    time.sleep(3)
    if numeroRichiesta is not None:
        print("TENTATIVO ", numeroRichiesta)

```

Per esempio la classe *MediaworldScraper* riscrive questo metodo per evitare gli errori 404. Se una richiesta a Mediaworld fallisce per 404 vuol dire che le richieste

sono state temporaneamente bloccate perché sono state rilevate richieste da parte di un “robot”.

```
'''Se Mediaworld non risponde aspetto fino a 2 volte, prima 60 e poi 120 secondi e riprovo'''
def waitRequest(self, numeroRichiesta: int):
    # Dopo 3 errori 404 aspetta un minuto e riprova
    # Aspetta fino a un minuto per massimo 3 volte
    # 403/404 try parte da 1
    print("MEDIOWORLD WAIT REQUEST: TENTATIVO ", numeroRichiesta, " - 404 ERROR: ", self.__try_404)
    # se si possono fare ancora richieste
    if numeroRichiesta < self.maximum_request:
        time.sleep(self.deelay_time)
    # Se si ottengono 3 risposte 403/404 consecutive si aspetta un po' e si riprova fino __maximum_404_try volte
    elif numeroRichiesta == self.maximum_request and self.__try_404 < self.__maximum_404_try:
        # Le richieste effettuate vengono azzerate, altrimenti il metodo di GenericScraper termina dopo 3 errori 403/404
        self.richieste_effettuate = 0
        print("MEDIOWORLD: ASPETTO {} SECONDI".format(self.__404_sleep_time * self.__try_404))
        time.sleep(self.__404_sleep_time * self.__try_404)
        self.__try_404 += 1
    else:
        # Se dopo __maximum_try_404 volte non si riesce, si fallisce
        self.__try_404 = 1
```

Quindi la classe *MediaworldScraper* riprova per un massimo di 3 volte, attendendo rispettivamente 60 secondi, 120 secondi e 180 secondi. Se tutte e tre falliscono allora si considera la richiesta come fallita.

Il metodo *isAvailable* restituisce *True* se il prodotto è disponibile, mentre il metodo *getPrice* restituisce il prezzo del prodotto. Il metodo *fixPrice* invece restituisce il prezzo corretto del prodotto, rimuovendo eventuali caratteri presenti in esso.

```
def fixPrice(self, price: str):
    if price is None:
        return -1

    # Prendo tutti i numeri che sono nella stringa, facendo lo split con la ','
    # Restituisce un array contenente tutti i numeri
    # Esempio:
    # Input: 1.497,59, Output: ['1', '497', '59']
    # Input: 1189, Output: ['1189', '00']
    # Eventuali caratteri che non sono numeri vengono scartati
    temp = re.findall(r"[+]?\d*\.\d+|\d+", price)

    # print("TEMP: ", temp)

    # Se contiene solo due elementi restituisce il primo e il secondo divisi dal punto
    if temp.__len__() == 2:
        return temp[0] + "." + temp[1]
    elif temp.__len__() == 3:
        # Altrimenti restituisce il primo e il secondo concatenati e aggiunge un punto tra la
        # concatenazione di questi due e il terzo
        return temp[0] + temp[1] + "." + temp[2]
```

AmazonScraper riscrive il metodo *get_offers* in questo modo

```
def get_offers(self) -> list:
    prodotti = readFromFile(self.input_file)
    product_list = self.scrape(prodotti)
    return product_list
```

readFromFile è un metodo che legge i prodotti (url e nome prodotto) dal file.

```
def readFromFile(filepath: str):
    lista_prodotti = []
    with open(filepath, 'r') as file:
        lines = file.readlines()
        for line in lines:
            # Rimuovo lo '\n' alla fine di ogni riga
            line = line.strip("\n")
            params = line.split(SEPARATOR)
            lista_prodotti.append(Prodotto(nome=params[1][0: params[1].__len__()], url=params[0]))
    return lista_prodotti
```

Se si ha la necessità di leggere i dati in modo diverso, e quindi non tramite scraping, si può implementare tranquillamente un altro meccanismo all'interno del metodo *get_offers*.

Generazione grafici

L'ultima parte è stata quella della generazione dei grafici. Avendo a disposizione tutti i dati è bastato semplicemente leggerli dal database ed organizzarli attraverso l'utilizzo della libreria *matplotlib*.

La classe *GestoreGrafici* si occupa della creazione dei grafici e include un meccanismo di listeners che servono per notificare quando un grafico è stato prodotto, motivo per il quale estende la classe astratta *Ascoltabile*.

```
class Ascoltabile:
    __metaclass__ = ABCMeta

    @abstractmethod
    def addListeners(self, listeners: list = Ascoltatore):
        raise Exception("NotImplementedException")

    @abstractmethod
    def removeListener(self, listener: Ascoltatore):
        raise Exception("NotImplementedException")

    @abstractmethod
    def notify(self, operation, *args):
        raise Exception("NotImplementedException")
```


Il metodo *ottieniGrafici* è incaricato di produrre i grafici e notificare i listener quando la creazione è finita.

```
def ottieniGrafici(self, scraper: GenericScraper, dataInizio=None, dataFine=None, multiplePriceForDay=False, discontinuo=True):
    tuttiIProdotti = DatabaseManager.selectProduct(scraper, "")

    for prodotto in tuttiIProdotti:
        # nomeProdotto = prodotto[1][0:prodotto[1].__len__() - 1]
        nomeProdotto = prodotto[0].strip("\n")
        # print(nomeProdotto)
        self.ottieniGrafico(scraper, nomeProdotto, dataInizio=dataInizio, dataFine=dataFine,
                           multiplePriceForDay=multiplePriceForDay, discontinuo=discontinuo)

    self.notify("fine", DatabaseManager.getTable(scraper), ">>>>> Generazione grafici completata <<<<<")
```

Come prima cosa viene letta dal database la lista di tutti i prodotti presenti. Successivamente, per ognuno di essi, si chiama il metodo *ottieniGrafico*, il quale legge tutti i dati relativi ad un prodotto dal database e crea il grafico.

```
def ottieniGrafico(self, scraper: GenericScraper, nomeProdotto: str, dataInizio=None, dataFine=None,
                  multiplePriceForDay=False, discontinuo=True, costruisciGrafico = True, folder=None):

    if folder is not None:
        cartella_output = os.path.join(folder, DatabaseManager.getTable(scraper))
    else:
        cartella_output = DatabaseManager.getTable(scraper)

    # Solo per cercare nel db
    nomeProdotto = nomeProdotto.replace("'", "")

    prodotti = DatabaseManager.selectProduct(scraper, nomeProdotto, dataInizio=dataInizio, dataFine=dataFine, multiplePriceForDay=multiplePriceForDay)
    prezzi = self.__ottieniPrezzi(prodotti)
    date = self.__ottieniData(prodotti)

    nomeProdotto = self.normalizza_nome_prodotto(nomeProdotto)

    # Crea la cartella se non esiste
    Path(cartella_output).mkdir(parents=True, exist_ok=True)

    if self.__listeners is not None:
        self.notify("prezzi", DatabaseManager.getTable(scraper), nomeProdotto)

    if costruisciGrafico:
        self.costruisciGrafico(nomeProdotto, date, prezzi, cartella_output, discontinuo=discontinuo)
    else:
        return nomeProdotto, date, prezzi, cartella_output, discontinuo
```

Il metodo *__ottieniPrezzi* restituisce una lista contenente tutti i prezzi del prodotto.

```
def __ottieniPrezzi(self, prodotti: list) -> list:
    prezzi = list()
    for prodotto in prodotti:
        prezzi.append(prodotto[3])

    return prezzi
```

Mentre il metodo `__ottieniData` restituisce la lista di tutte le date del prodotto.

```
def __ottieniData(self, prodotti: list) -> list:
    date = list()
    # last_month = "0"
    for prodotto in prodotti:
        data = prodotto[prodotto.__len__() - 1]
        data = data.strftime('%d/%m')
        date.append(data)

    return date
```

Il metodo `normalizza_nome_prodotto` serve per rimuovere alcuni caratteri che creerebbero problemi alla libreria `matplotlib`.

```
@staticmethod
def normalizza_nome_prodotto(nomeProdotto):
    nomeProdotto = nomeProdotto.replace("\", " pollici")
    nomeProdotto = nomeProdotto.replace("'", "")
    nomeProdotto = nomeProdotto.replace(":", " ")
    nomeProdotto = nomeProdotto.replace("/", "-")

    return nomeProdotto
```

Successivamente vengono avvisati i listener, se ce ne sono, e viene chiamato il metodo `costruisciGrafico`, il quale genererà il grafico e lo salverà nella cartella di destinazione.

```
def costruisciGrafico(self, nomeProdotto: str, date: list, prezzi: list, cartellaOutput: str, discontinuo=False):
    if discontinuo:
        prezzi = self.rimuoviValoriSuperflui(prezzi)

    pl.title(nomeProdotto)
    pl.plot(date, prezzi, "r-")
    pl.plot(date, prezzi, "ro")
    pl.ylabel('Prezzi')
    pl.xlabel('Giorni')
    pl.grid()
    pl.xticks(rotation=75)
    pl.savefig(cartellaOutput + '/' + nomeProdotto + ".png")
    pl.close()
```

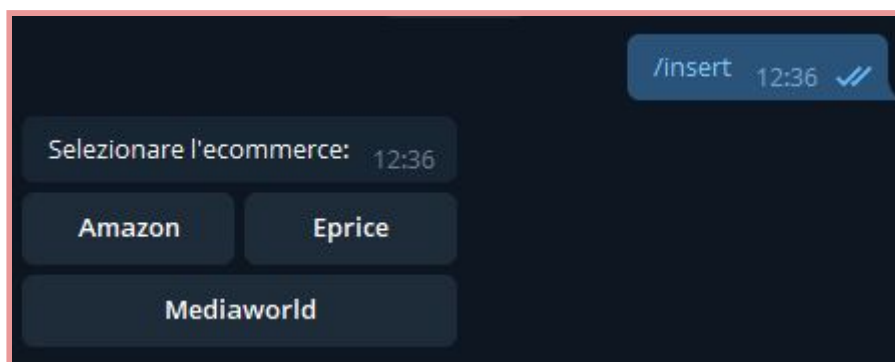
Bot per la gestione dei prodotti da analizzare

Per semplificare la gestione dei prodotti da analizzare è stato realizzato un bot telegram che permette di aggiungere nuovi prodotti da monitorare o rimuovere quelli esistenti. Lo script è hostato su un raspberry, e grazie a questo bot si possono gestire agevolmente i prodotti senza doversi collegare direttamente al raspberry. Quando viene richiesto un inserimento, il bot chiede per quale e-commerce si vuole inserire il prodotto.

```
def __insert(self, update, context):
    id = update.message.from_user['id']
    if id in self.__enabled_users:
        self.__handler[id] = self.__insertHandler
        self.__userdata[id] = []
        self.__step[id] = 0
        keyboard = [
            [
                InlineKeyboardButton("Amazon", callback_data='amazon'),
                InlineKeyboardButton("Eprice", callback_data='eprice'),
            ],
            [InlineKeyboardButton("Mediaworld", callback_data='mediaworld')],
        ]

        reply_markup = InlineKeyboardMarkup(keyboard)
        update.message.reply_text('Selezionare l\'ecommerce:', reply_markup=reply_markup)
```

L'utente può rispondere a questa domanda tramite gli appositi bottoni dell'interfaccia di telegram.



Quando si interagisce con questi pulsanti viene eseguito il seguente metodo

```
def __genericButton(self, update: Update, context: CallbackContext):
    id = update.callback_query.message.chat.id
    if self.__handler[id] == self.__insertHandler:
        self.__buttonInsert(update, context)
    elif self.__handler[id] == self.__buttonRemove:
        self.__buttonRemove(update, context)
```

`__handler` è un dizionario che ha come chiave l'id dell'utente che invia il messaggio, e come valore una funzione. Serve per ricordare cosa vuole fare l'utente. Se ad un determinato utente è associato il metodo `__insertHandler` allora viene

```
def __buttonInsert(self, update: Update, context: CallbackContext) -> None:
    id = update.callback_query.message.chat.id
    self.__step[id] = 1
    query = update.callback_query
    # CallbackQueries need to be answered, even if no notification to the user is needed
    # Some clients may have trouble otherwise. See https://core.telegram.org/bots/api#callbackquery
    query.answer()
    self.__userdata[id].append(query.data)
    print(self.__userdata[id])
    query.edit_message_text(text="Inserisci l'url di {}".format(query.data))
```

eseguito il metodo `__buttonInsert`, il quale chiede l'url del prodotto.

Dopo che l'url è stato inserito, verrà eseguita la funzione `__genericHandler`, che a sua volta invocherà il metodo associato all'utente, tramite il dizionario `__handler`.

```
def __genericHandler(self, update, context):
    id = update.message.from_user['id']
    self.__handler[id](update, context)
```

`__genericHandler` è un metodo che viene eseguito quando vengono inviati messaggi che non corrispondono a comandi del bot (non iniziano con `/`, ma sono messaggi testuali).

Quando si invia il messaggio `/insert` al bot, il metodo `__insert` associa all'id dell'utente il metodo `__insertHandler`, il quale chiede prima il nome del prodotto e poi comunica che è stato inserito.

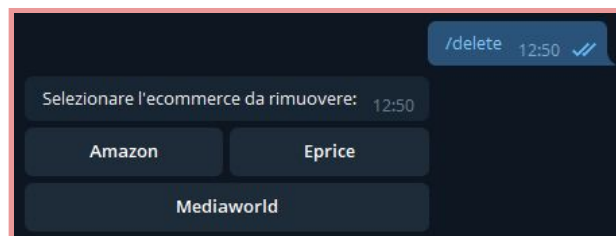
```
def __insertHandler(self, update, context):
    print("=== INSERT ===")
    id = update.message.from_user['id']
    message = update.message.text
    if id in self.__enabled_users:
        if self.__step[id] == 1:
            self.__userdata[id].append(message)
            txt = "Inserisci il nome del prodotto"
            self.__send_message(update, context, txt, id)
            print(self.__userdata[id])
            self.__step[id] = 2
        elif self.__step[id] == 2:
            self.__userdata[id].append(message)
            print(self.__userdata[id])
            self.__insertintofile(id)
            self.__cleanVariables(id)
            self.__send_message(update, context, "Prodotto inserito correttamente.", id)
```


Quando invece viene inviato il comando `/delete` viene eseguito un metodo analogo il quale però associa all'utente il metodo `__buttonRemove`. Viene chiesto di selezionare l'e-commerce da cui rimuovere il prodotto.

```
def __delete(self, update, context):
    id = update.message.from_user['id']
    if id in self.__enabled_users:
        self.__handler[id] = self.__buttonRemove
        self.__userdata[id] = []
        self.__step[id] = 0
        # self.__send_message(update, context, text="Selezionare l'e-commerce", chat_id=id)
        keyboard = [
            [
                InlineKeyboardButton("Amazon", callback_data='amazon'),
                InlineKeyboardButton("Eprice", callback_data='eprice'),
            ],
            [InlineKeyboardButton("Mediaworld", callback_data='mediaworld')],
        ]

        reply_markup = InlineKeyboardMarkup(keyboard)
        update.message.reply_text('Selezionare l\'e-commerce da rimuovere:', reply_markup=reply_markup)
```

Il metodo `__buttonRemove` chiede prima di selezionare il prodotto da rimuovere, sempre attraverso l'utilizzo di bottoni, e successivamente lo elimina.



```

def __buttonRemove(self, update: Update, context: CallbackContext):
    id = update.callback_query.message.chat.id
    query = update.callback_query
    if self.__step[id] == 0:
        input_file = "files/" + query.data + "_product_list.txt"

        prodotti = FileUtility.readFromFile(input_file)

        reply_markup = InlineKeyboardMarkup(self.__format_keyboard(prodotti, 2))

        bot = context.bot
        bot.edit_message_text(
            chat_id=query.message.chat_id,
            message_id=query.message.message_id,
            text="Selezionare il prodotto da rimuovere:",
            reply_markup=reply_markup
        )
        query.answer()
        self.__userdata[id].append(input_file)
        self.__step[id] = 1
    elif self.__step[id] == 1:
        FileUtility.deleteFromFile(self.__userdata[id][0], query.data)
        query.answer()
        bot = context.bot
        bot.edit_message_text(
            chat_id=query.message.chat_id,
            message_id=query.message.message_id,
            text="Prodotto eliminato con successo"
        )
        self.__cleanVariables(id)

```

L'interfaccia grafica

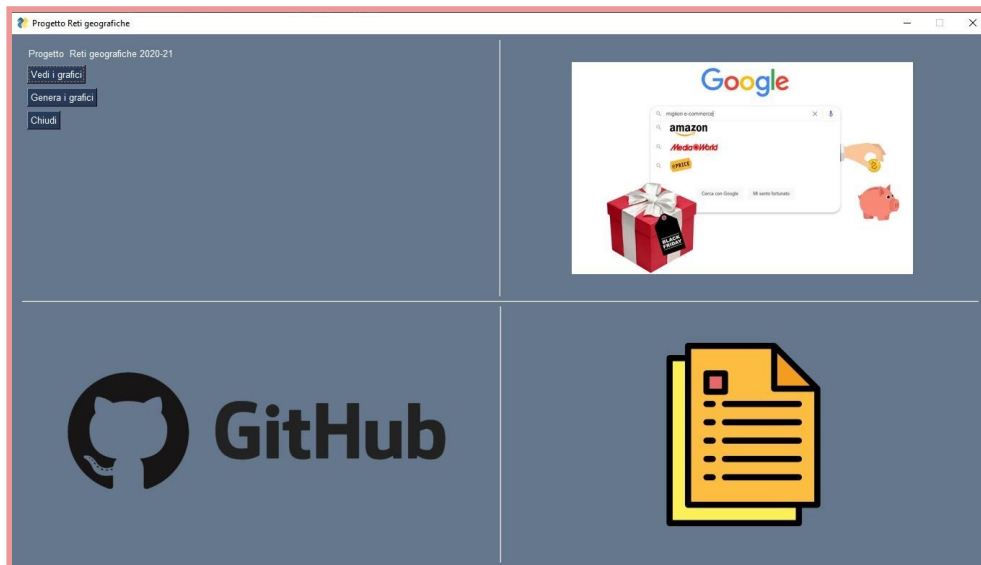
La parte relativa all'interfacciamento con l'utente è stata realizzata grazie alla libreria PySimpleGui, uno strumento semplice ed innovativo, che permette di definire interfacce in maniera veloce.



L'interfaccia realizzata si compone di quattro pannelli:

- **Pannello iniziale:** Contiene un collegamento verso il pannello di visualizzazione e quello di generazione dei grafici ed il link alla documentazione e al codice GitHub del progetto.
- **Pannello visualizzazione grafici:** Permette la visualizzazione dei grafici (devono essere prima generati) effettuando l'opportuna selezione dalla lista sulla parte sinistra e visualizzando il relativo grafico sulla parte destra dell'interfaccia. Contiene anche informazioni sul report del prodotto selezionato.
- **Pannello generazione grafici:** Permette la generazione dei grafici, personalizzabili con diversi parametri mostrati all'interno dell'interfaccia (analizzati in seguito). Una volta selezionate tutte le opzioni, premendo sul tasto "genera" saranno generati e salvati i grafici all'interno dell'apposita cartella.
- **Pannello status generazione grafici:** Questo pannello viene visualizzato dopo aver premuto il tasto "genera" del pannello precedente e contiene fino a tre progress bar che evidenziano l'avanzamento della generazione dei grafici.

Pannello iniziale



(interfaccia iniziale dell'applicazione)

E' il pannello visualizzato all'avvio dell'applicazione, è composto da una griglia contenente 4 celle, la prima permette di muoversi all'interno dell'interfaccia andando al pannello di visualizzazione o generazione grafici, la seconda contiene il logo del progetto, la terza permette di accedere al codice GitHub (al click verrà aperta una pagina del browser) e la quarta contiene un link alla documentazione.

```
button_column = [
    [
        sg.Text("Progetto Reti geografiche 2020-21")
    ],
    [
        sg.Button("Vedi i grafici")
    ],
    [
        sg.Button("Genera i grafici")
    ],
    [
        sg.Button("Chiudi")
    ]
]

logo_viewer_column = [
    [sg.Image(key="-LOGOIMAGE-", size=(640, 350), filename="../immagini/copertina.png", enable_events=True)],
]

github_viewer_column = [
    [sg.Image(key="-GITHUBIMAGE-", size=(640, 350), filename="../immagini/github.png", enable_events=True)],
]

doc_viewer_column = [
    [sg.Image(key="-DOCIMAGE-", size=(640, 350), filename="../immagini/documenti.png", enable_events=True)],
]

layout = [
    [
        sg.Column(button_column, size=(651, 350)),
        sg.VSeparator(),
        sg.Column(logo_viewer_column),
    ],
    [
        sg.HSeparator()
    ],
    [
        sg.Column(github_viewer_column),
        sg.VSeparator(),
        sg.Column(doc_viewer_column),
    ],
]
```

(codice creazione layout)

Come è possibile notare dal codice, sono presenti due colonne (definite in quattro liste, una per cella), ognuna contenente due celle separate da una linea verticale. Inoltre per dividere la prima riga dalla seconda è presente un ulteriore separatore orizzontale, questa volta.

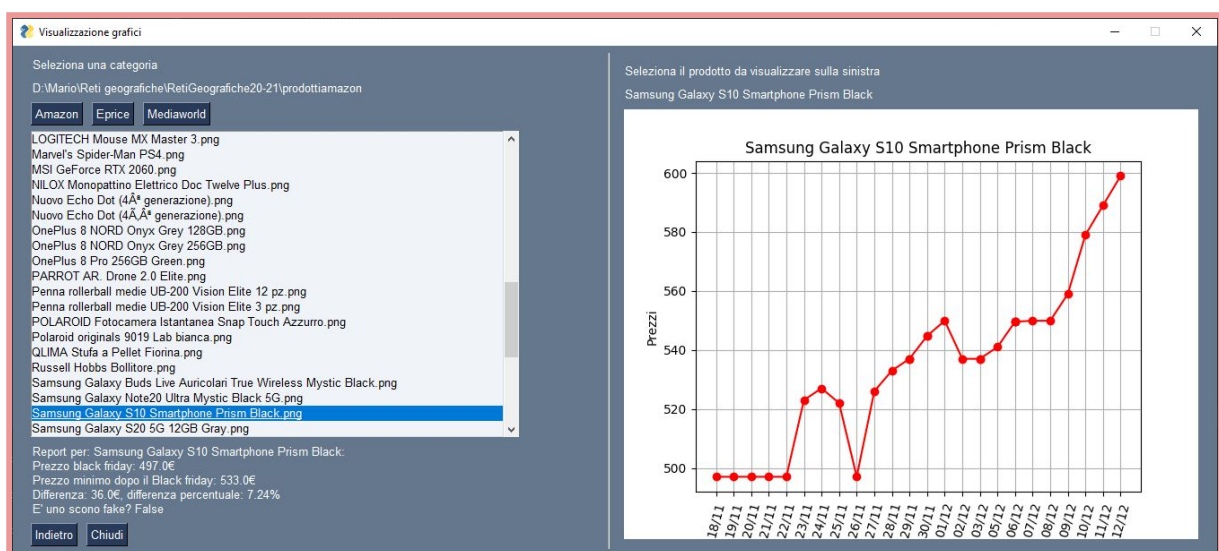
```
while True:
    event, values = window.read()

    if event == "Chiudi" or event == sg.WIN_CLOSED:
        window.close()
        break
    elif event == "Vedi i grafici":
        window.close()
        showgraph()
        break
    elif event == "Genera i grafici":
        window.close()
        generateGraph()
        break
    elif event == "-DOCIMAGE-":
        webbrowser.open('https://github.com/mariomamo/RetiGeografiche20-21/tree/main/documentazione', new=2)
    elif event == "-GITHUBIMAGE-":
        webbrowser.open('https://github.com/mariomamo/RetiGeografiche20-21', new=2)
```

(flusso degli eventi per il pannello iniziale)

Il flusso di eventi viene gestito attraverso un loop che rimane in ascolto (attraverso il metodo read) fino al verificarsi di un'interazione con l'interfaccia che si conclude con la gestione del relativo evento. Per questo pannello, quando viene cliccato un pulsante viene aperta la relativa interfaccia, mentre se si clicca su un'immagine viene aperta la pagina browser come descritto in precedenza.

Pannello visualizzazione grafici



(pannello visualizzazione grafici)

All'interno di questo pannello è possibile visualizzare i grafici relativi a ciascun e-commerce selezionando prima il sito web, cliccando sul pulsante e successivamente la lista sottostante sarà riempita con tutti i nomi dei prodotti analizzati. Selezionando un prodotto sarà possibile vedere il relativo grafico con l'andamento dei prezzi generato attraverso Matplotlib ed un piccolo report che indica i prezzi che il prodotto ha avuto durante il giorno del black friday ed il prezzo minore a cui è stato venduto successivamente, in base a questo si determina se lo sconto è stato effettivo oppure no.

```
stats = GestoreGrafici.load_sconto_info(nomeprodotto, path)
if stats is not None:
    window["-REPORT-"].update(f"Report per: {stats.nome}:\n"
                               f"Prezzo black friday: {stats.prezzo_bf}€\n"
                               f"Prezzo minimo dopo il Black friday: {stats.prezzo_dopo}€\n"
                               f"Differenza: {stats.differenza_prezzo}€, differenza percentuale: {stats.percentuale_sconto}%\n"
                               f"E' uno sconto fake? {stats.is_sconto_fake}")
else:
    window["-REPORT-"].update(f"Report non disponibile")
```

(scrittura del report)

Il codice relativo alla creazione del pannello è simile a quello visto in precedenza (per maggiori informazioni è possibile consultare il codice su GitHub) e la gestione degli eventi prevede anche in questo caso un ciclo con attesa dell'interazione dell'utente. Per quanto concerne il report visualizzato, questo consiste essenzialmente in una lettura del file che lo contiene di tutti i prodotti suddivisi per e-commerce e la relativa impaginazione formattata e mostrata attraverso una componente testuale di PySimpleGui come è possibile vedere nel codice soprastante.

Generazione del report

```
Apple iPad Grigio Siderale (10,2 pollici, Wi-Fi, 32GB) 380.21 389.0 2.31 8.79 False
Apple iPad mini 5 Grigio Siderale (Wi-Fi, 64GB) 400.0 398.99 -0.25 -1.01 True
Apple iPad Pro Argento (11 pollici, Wi-Fi, 128GB) 858.45 832.9 -2.98 -25.55 True
Apple iPad Pro Grigio siderale (12,9 pollici, Wi-Fi, 128GB) 1069.0 1006.88 -5.81 -62.12 True
Apple iPhone 11 Nero (64GB) 614.52 616.87 0.38 2.35 True
Apple iPhone 11 Pro Grigio Siderale (64GB) 857.9 868.25 1.21 10.35 True
Apple iPhone 11 Pro Max Argento(64GB) 1099.9 1006.25 -8.51 -93.65 True
```

(estratto del report per Epice)

Il file del report (per ogni e-commerce) visualizzato in precedenza viene creato durante la generazione dei grafici mediante un'apposita funzione. Inizialmente vengono caricate due liste di prodotti (con i relativi prezzi) del giorno del black friday e quello precedente visto che già si erano verificati alcuni sconti (i giorni 26 e 27 Novembre 2020) scegliendo il prezzo minimo di entrambi i giorni ed i prezzi successivi al black friday a partire dalla data 28 Novembre 2020.

```
prodotti_black_friday = DatabaseManager.get_prezzi_tutti_i_prodotti(scraper, "2020-11-26", "2020-11-27")
prodotti_dopo = DatabaseManager.get_prezzi_tutti_i_prodotti(scraper, "2020-11-28", "2020-12-26")
```

(caricamento dei prezzi per un dato ecommerce)

Una volta caricati i prezzi è possibile passare all'analisi che prevede il calcolo della differenza di prezzo tra il prezzo minimo successivo al black friday e quello relativo a quel giorno ed il valore percentuale dello sconto. Se la percentuale di sconto è inferiore alla soglia definita (del 2% dovuta a possibili fluttuazioni di prezzo) allora lo sconto proposto il giorno del black friday viene ritenuto "fake" ossia non valido.

```

differenza = round(prod_dopo.prezzo_minimo - prod_bf.prezzo_minimo, 2)
percentuale_sconto_oggi = round((differenza * 100) / prod_bf.prezzo_minimo, 2)

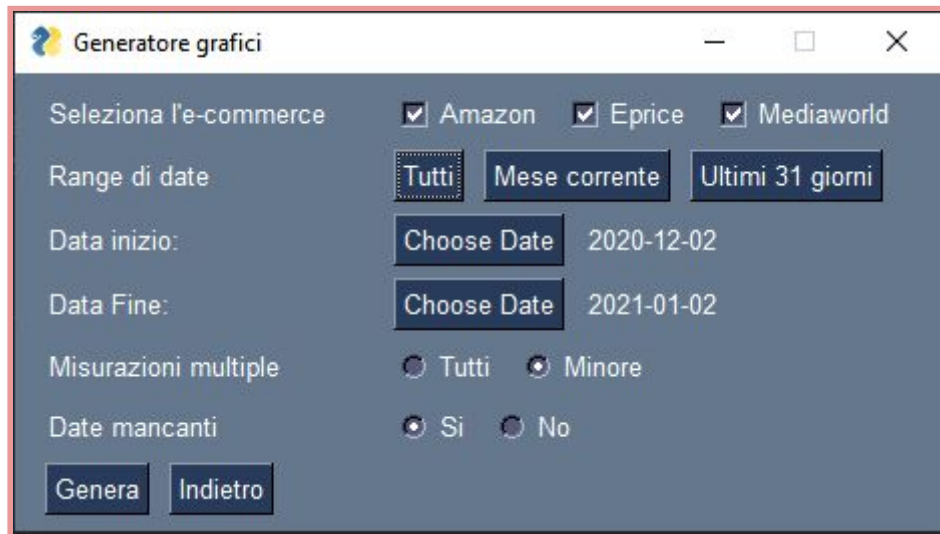
if percentuale_sconto_oggi <= soglia_percentuale:
    prod_dopo.is_fake_sconto = True

```

(l'analisi dei prezzi)

Al termine dell'analisi per ciascun prodotto, i risultati sono salvati all'interno del report e sono pronti per essere letti dal modulo dell'interfaccia utente.

Pannello generazione grafici



(pannello generazione grafici)

Grazie a questo pannello è possibile scegliere da una serie di opzioni in che modo si vogliono generare i grafici. Le opzioni mostrate sono varie ed in particolare permettono di:

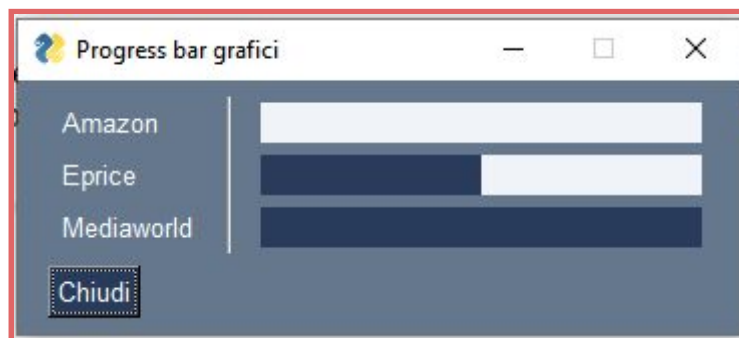
- **Selezionare l'e-commerce:** permette di scegliere per quali e-commerce generare i grafici.
- **Range di date:** permette di scegliere le date da utilizzare per la generazione dei grafici, a scelta tra l'ultima possibile, il mese corrente, gli ultimi 31 giorni oppure scelta personalizzata (bisogna inserire la data dal calendario al click di "choose date").

- **Misurazioni multiple:** permette di scegliere se si vuole la misurazione minore (per giorno) oppure si vogliono includere tutte le misurazioni effettuate all'interno del grafico.
- **Date mancanti:** permette di scegliere come saranno gestite le date mancanti all'interno del database (se per qualche ragione non è stato possibile ottenere il prezzo). Selezionando "Sì" la linea dei prezzi risulterà spezzata verso le date con misurazioni mancanti mentre selezionando "No" la linea risulterà continua.

```
p = ProgressWindow()
p.open(checked_scraper, data_inizio, data_fine, multipleprice, missingdata)
```

Una volta selezionate le opzioni che si vogliono utilizzare è possibile cliccare su "Genera" che lancerà il pannello successivo ed inizierà la generazione dei grafici.

Pannello status generazione grafici



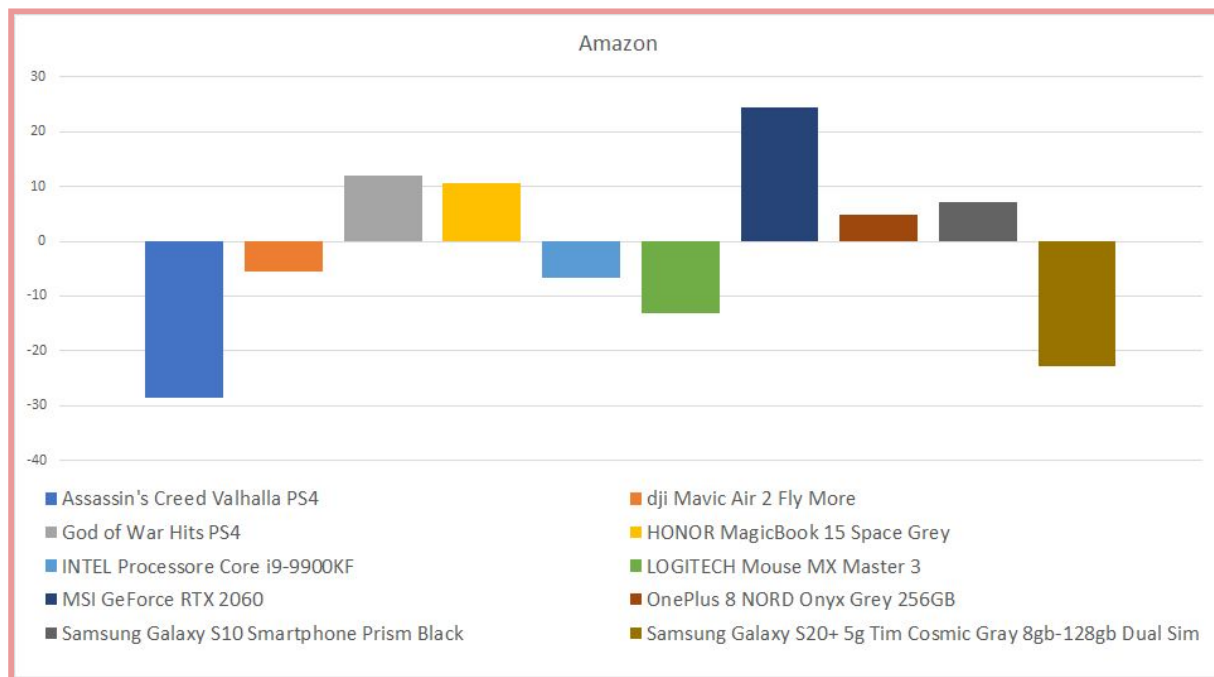
(pannello status avanzamento)

Lo scopo di questa interfaccia è solo quella di avvisare l'utente dell'avanzamento della generazione dei grafici per ognuno degli e-commerce precedentemente selezionati (per ognuno appare la propria progress bar, se scelta). Al termine dell'operazione, cliccando il tasto chiudi, verrà aperto nuovamente il pannello di generazione grafici.

Conclusioni

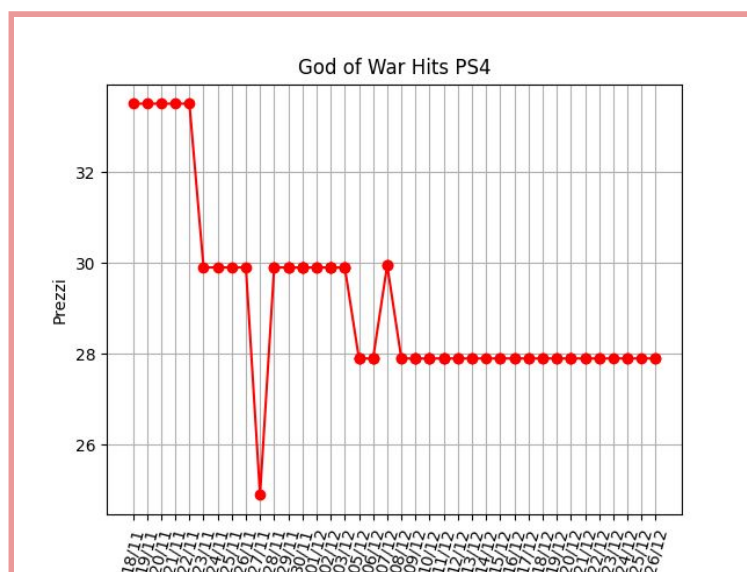
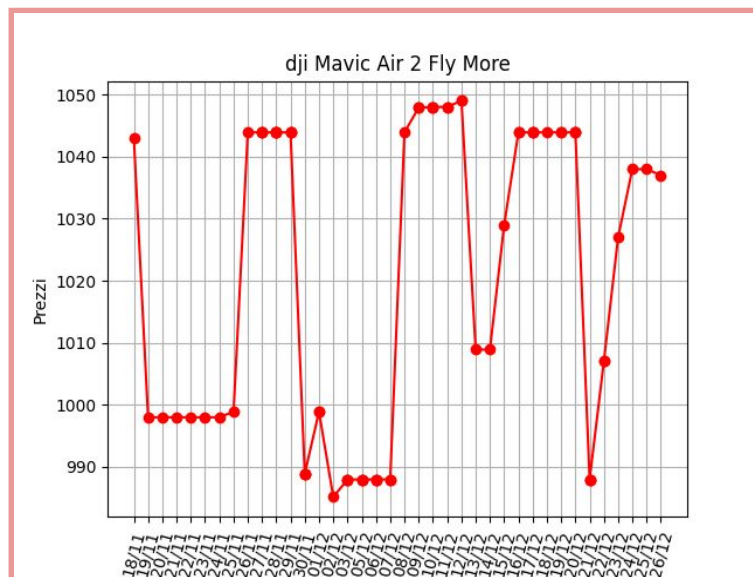
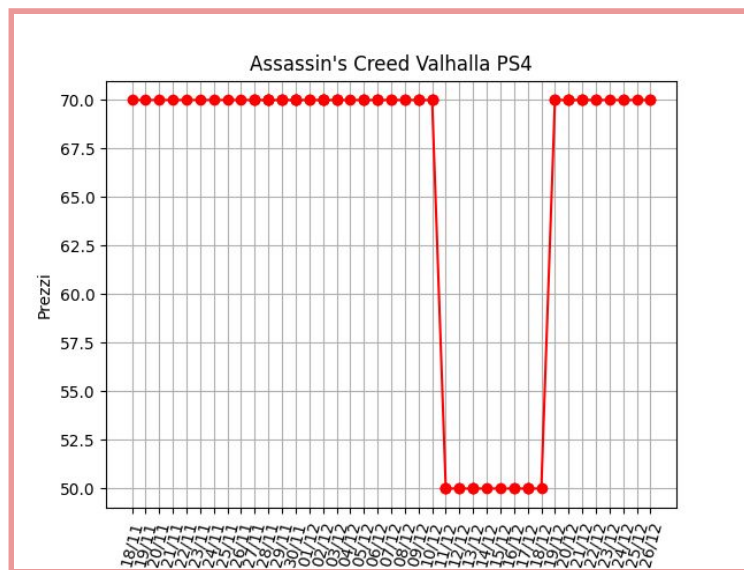
I seguenti grafici rappresentano la percentuale di sconto su un campione di 10 prodotti selezionati per ogni e-commerce. I grafici indicano la percentuale di sconto (ottenuta come mostrato in precedenza), per ognuno dei prodotti, che può essere positiva (se durante il black friday vi è stato un effettivo sconto) oppure negativa.

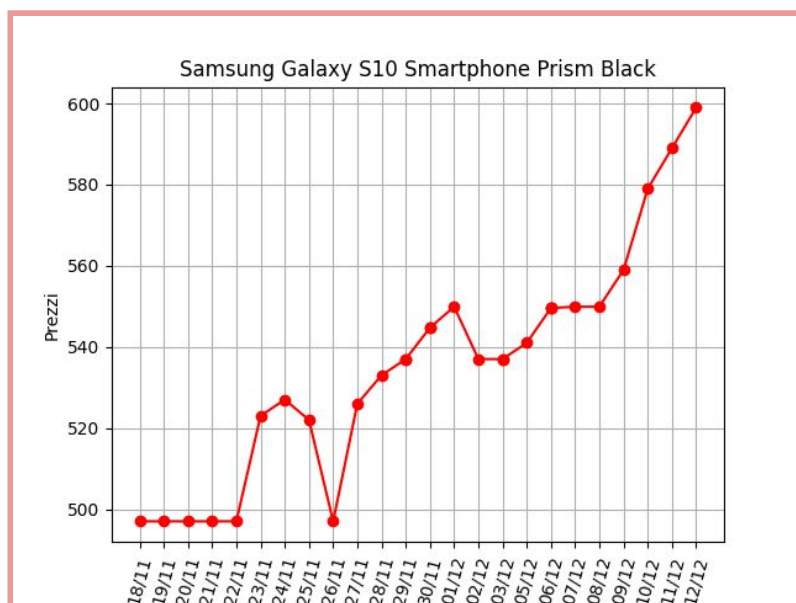
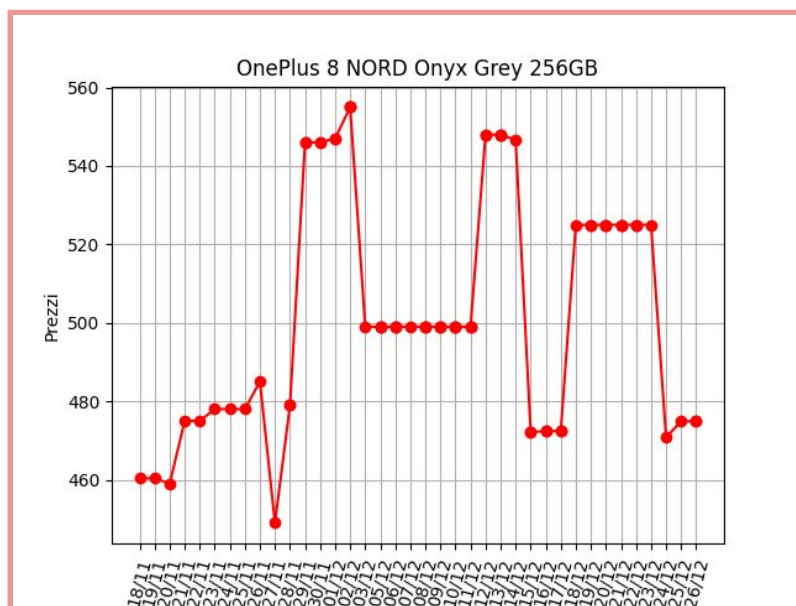
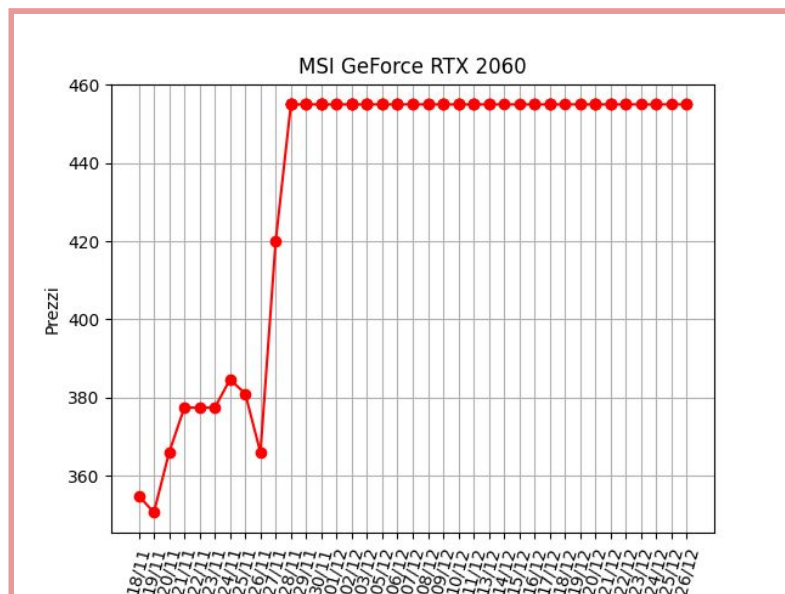
Amazon

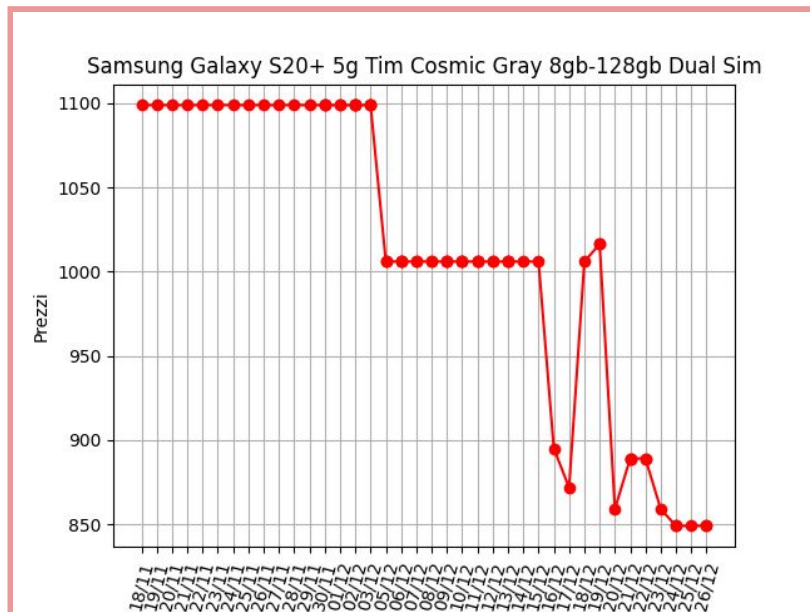


Il report generato per l'e-commerce Amazon ha evidenziato in generale una fluttuazione dei prezzi con cadenza quasi giornaliera. In alcuni casi il prezzo relativo al black friday è risultato essere effettivamente uno sconto valido rispetto ai prezzi offerti nei giorni successivi a questo evento mentre in altri no. Su un totale di 73 prodotti analizzati, sono risultati essere effettivamente in sconto 20 per una percentuale del 14,6%.

Grafici dei prodotti selezionati

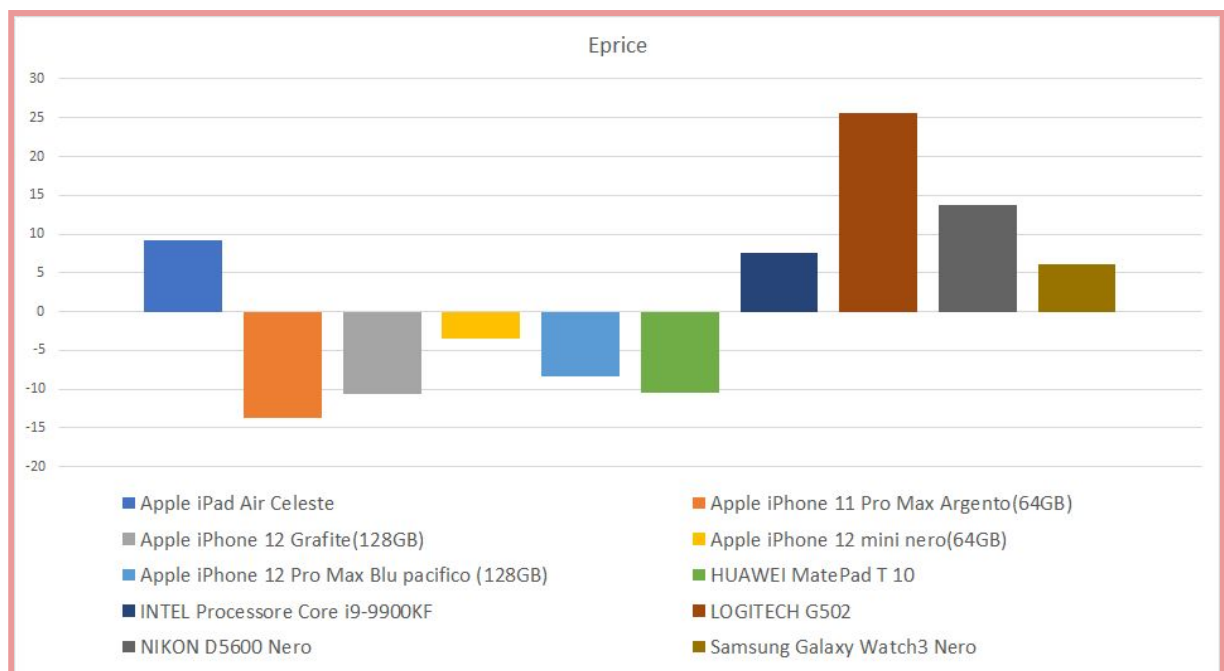






E' possibile notare in generale dai grafici una grande fluttuazione dei prezzi, in particolare per il dji Mavic Air 2 Fly More la cui varianza è elevata. Inoltre il Samsung Galaxy S10 Smartphone Prism Black è risultato essere quello più scontato durante l'evento ed il Samsung Galaxy S20+ 5g Tim Cosmic Gray 8gb-128gb Dual Sim è quello che ha subito una maggior diminuzione del prezzo nei giorni successivi.

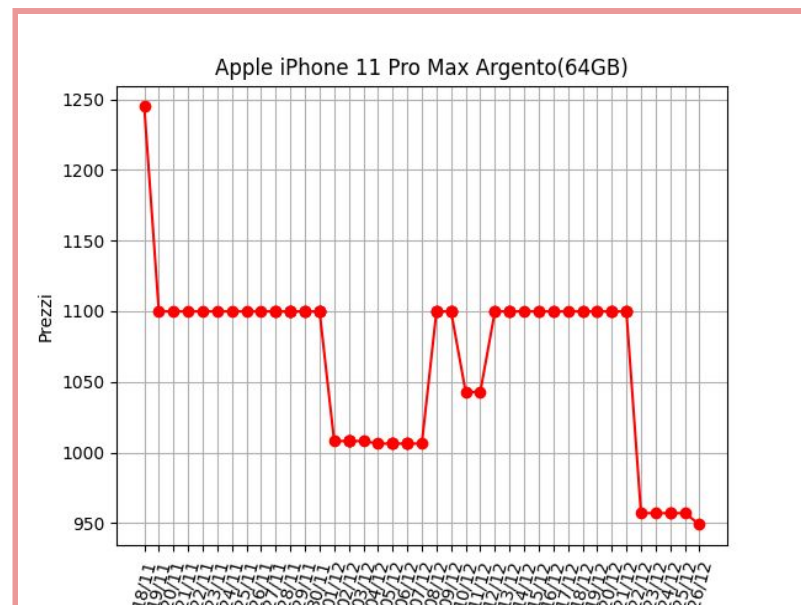
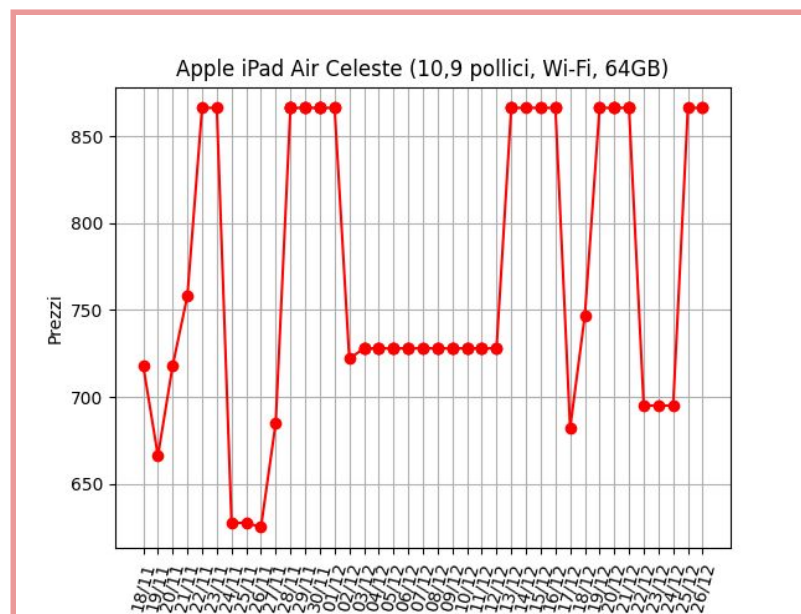
Eprice



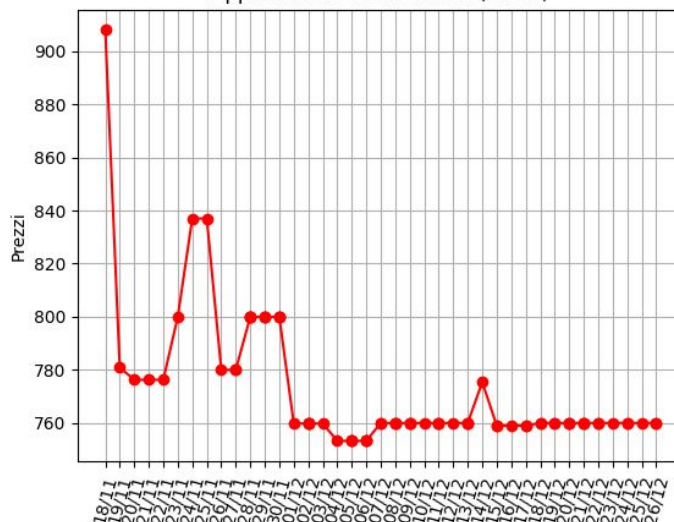
Il report generato per l'e-commerce Eprice ha evidenziato durante il periodo del black friday alcuni sconti rilevanti (con una grossa percentuale rispetto al prezzo del

prodotto) anche se in percentuale estremamente bassa, infatti, in alcuni casi è stato riscontrato un prezzo superiore durante l'evento rispetto al periodo immediatamente successivo. Su un totale di 52 prodotti, 13 sono risultati in effettivo sconto per una percentuale del 6,76%.

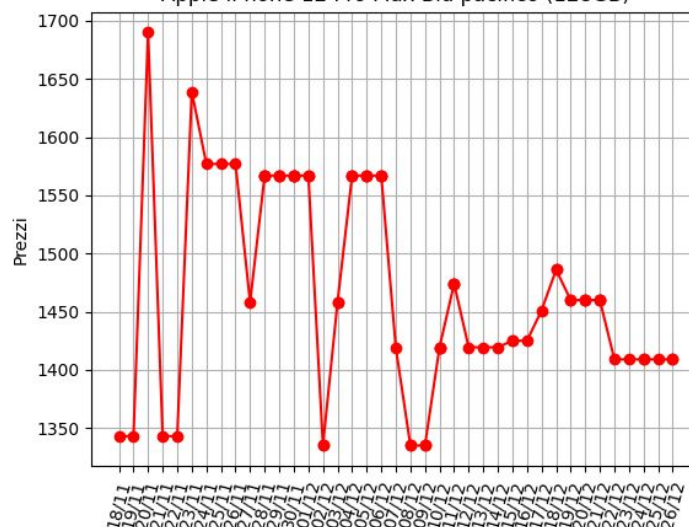
Grafici dei prodotti selezionati



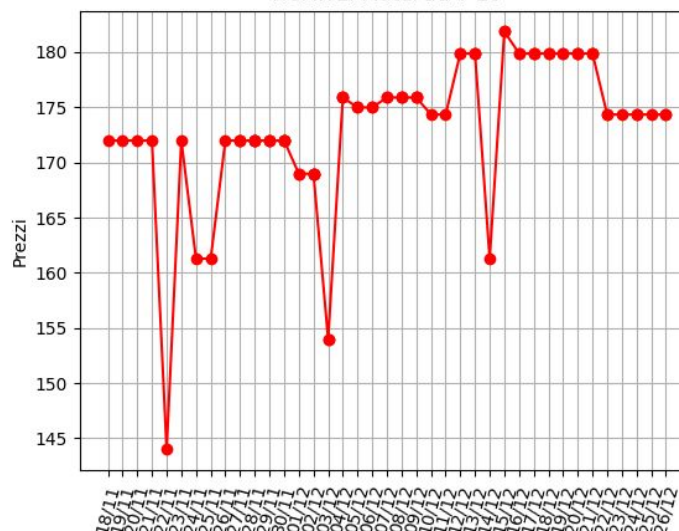
Apple iPhone 12 mini nero(64GB)



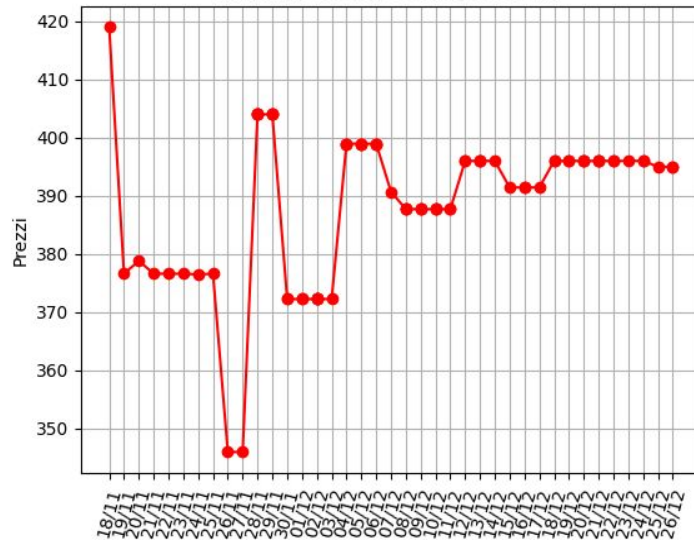
Apple iPhone 12 Pro Max Blu pacifico (128GB)



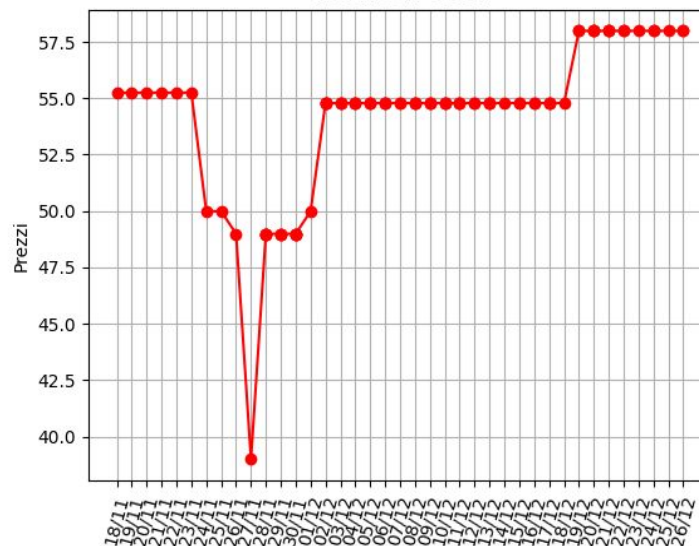
HUAWEI MatePad T 10



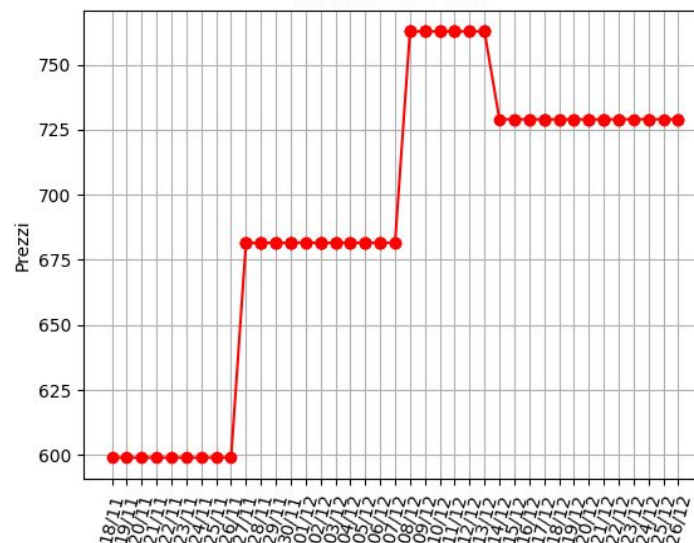
INTEL Processore Core i9-9900KF

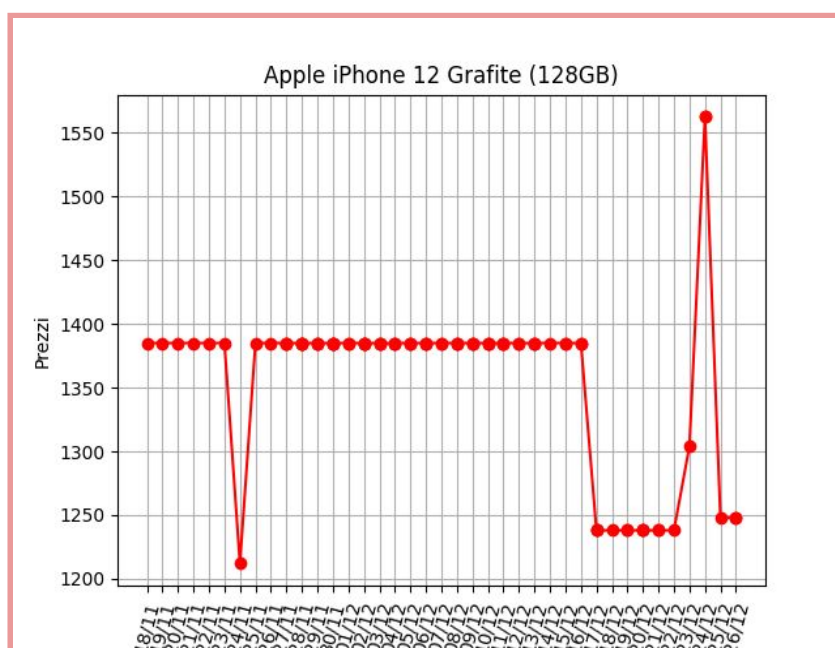
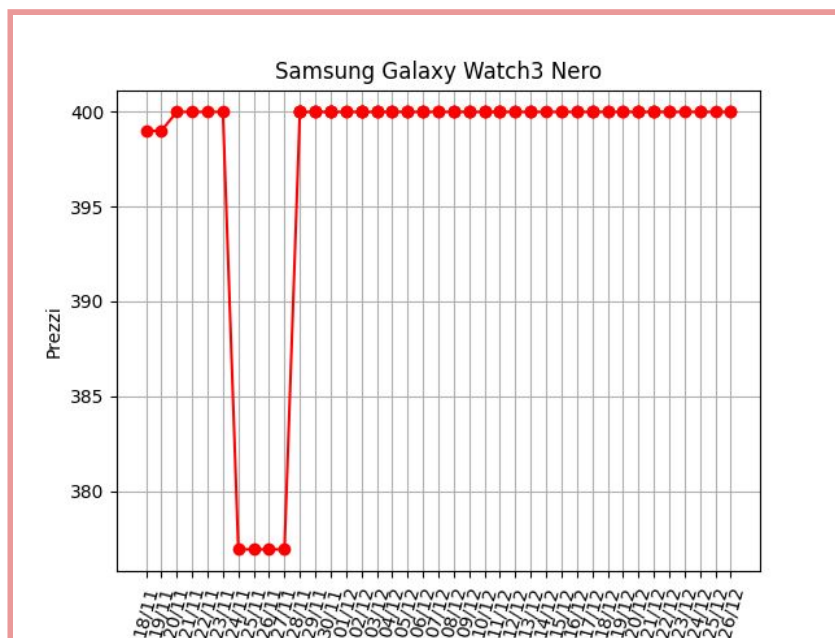


LOGITECH G502



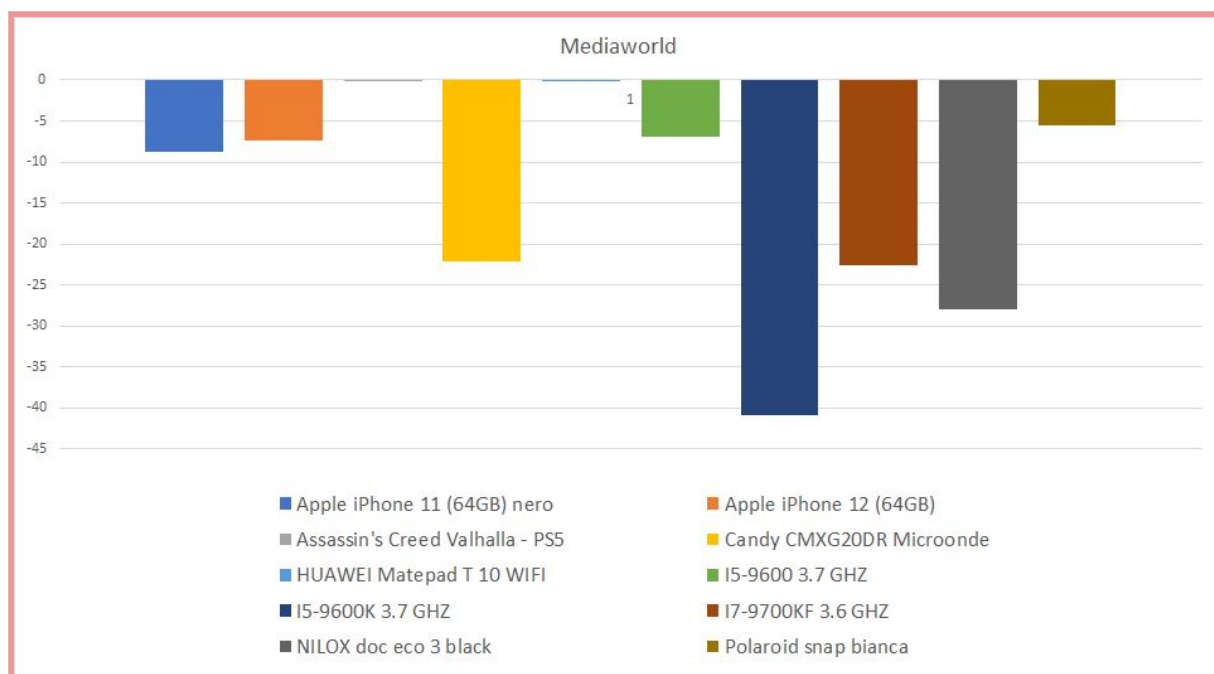
NIKON D5600 Nero





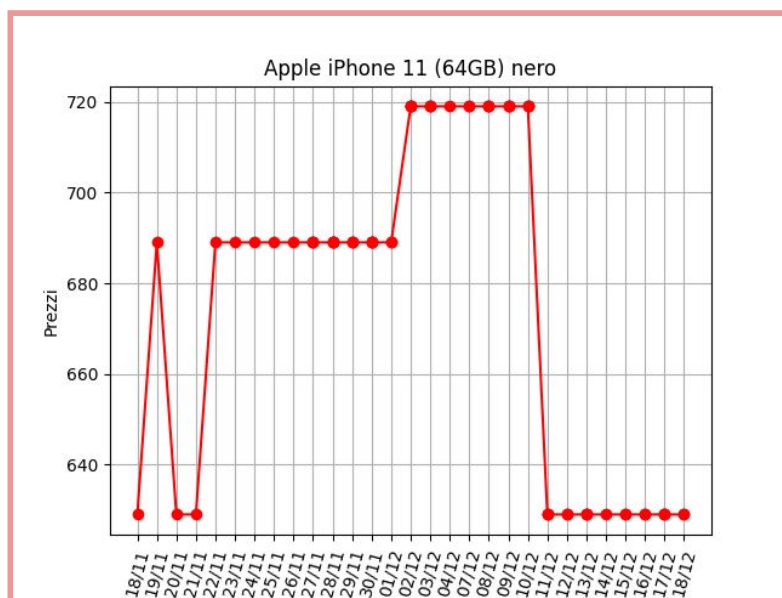
In questo caso è possibile vedere che i prezzi presentano una stabilità maggiore rispetto ad Amazon. In particolare la NIKON D5600 Nero è risultata essere quella più in sconto durante l'evento (considerando che nei giorni successivi il prezzo è solo aumentato) mentre l' Apple iPhone 11 Pro Max Argento(64GB) è il prodotto ad aver subito maggior ribasso durante il periodo immediatamente successivo (ed è quindi risultato non esser scontato).

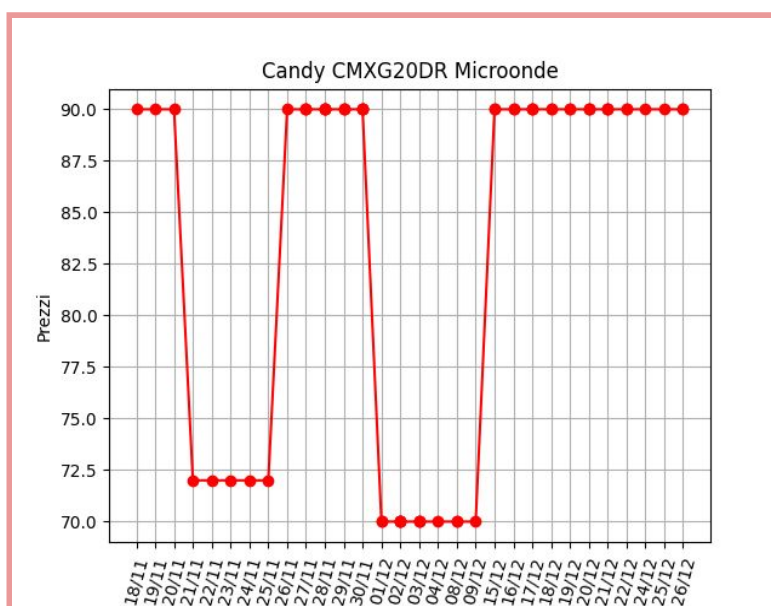
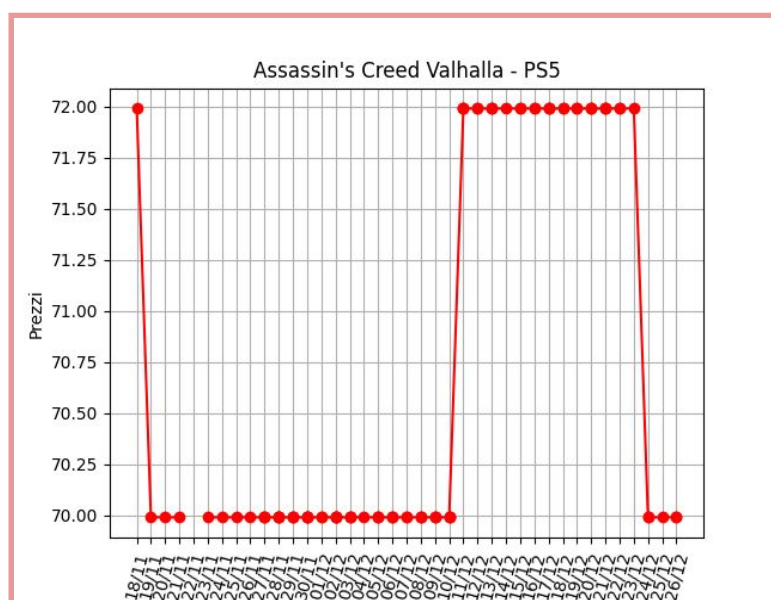
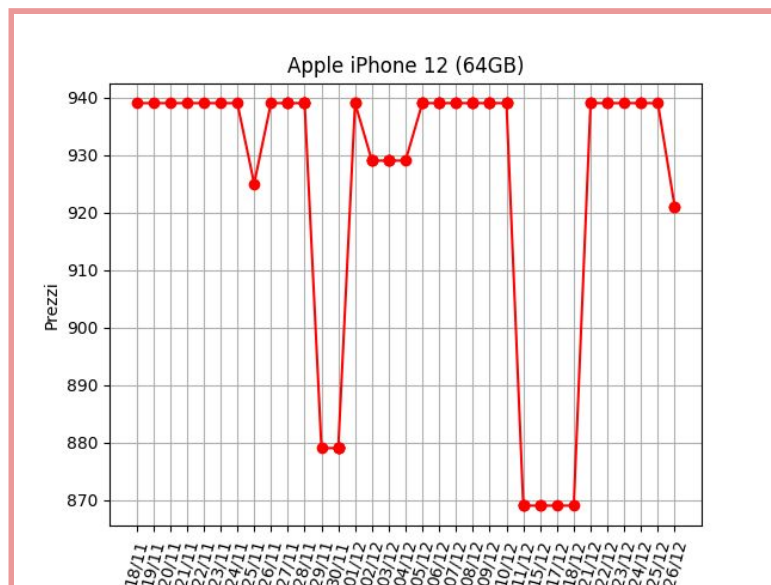
Mediaworld

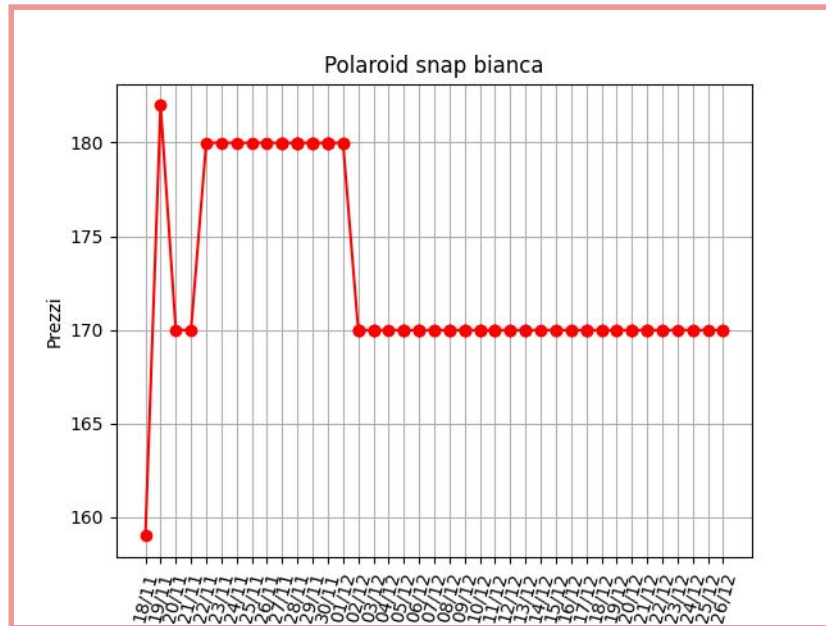


Il report generato per l'e-commerce Mediaworld ha evidenziato prezzi alti rispetto alla concorrenza durante l'evento con alcuni aumenti nei giorni precedenti. Gli effettivi sconti rilevati sono pochi e di percentuale molto bassa, mentre come si evince dal grafico, i prezzi in aumento (con valore negativo, ossia la quasi totalità del grafico) sono abbastanza alti. Tuttavia, su un totale di 85 prodotti analizzati, 15 sono risultati in effettivo sconto per una percentuale del 12,75%.

Grafici dei prodotti selezionati







La fluttuazione dei prezzi per questo e-commerce è risultata abbastanza stabile, considerando che non vi è concorrenza tra i vari possibili venditori (essendo questo una catena fisica). In particolare per l'Apple iPhone 11 (64GB) nero si è riscontrata la diminuzione di prezzo più grande nei giorni successivi all'evento mentre per il HUAWEI Matepad T 10 WIFI il prezzo è prima rimasto stabile ma è successivamente solo aumentato.

Conclusioni finali

In conclusione, sono stati analizzati un totale di 210 (73 per Amazon, 52 per Eprice ed 85 per Mediaworld) prodotti appartenenti a diverse categorie e sono stati raccolti 21036 record all'interno del database (6931 per Amazon, 5356 per Eprice e 8749 per Mediaworld). E' possibile dire in generale che gli effettivi sconti in relazione al numero di prodotti offerti da questi colossi sono estremamente bassi (in media solo l'11,37%). Dei tre e-commerce analizzati il migliore è risultato Amazon, principale promotore dell'evento che offre quindi i prezzi più competitivi sul mercato, al secondo posto si piazza Mediaworld, che pur essendo meno organizzato (vedi i problemi di scalabilità e organizzazione del sito) è risultato migliore rispetto al terzo classificato (soltanto per numero di prodotti in effettivo sconto, anche se la percentuale effettiva è estremamente bassa rispetto agli altri), ossia Eprice che ha offerto la percentuale di sconto sul numero di prodotti più bassa.

