Data Management Project

Comensoli Paolo, Gazzola Michele, Usai Alberto

883147, 825267, 886731

Università degli Studi di Milano Bicocca

MSc Data Science

Anno accademico 2021/2022

Settembre, 2022

Codici, notebook e dataset sono disponibili qui: Drive

I. Introduzione

Durante la ricerca di un immobile in affitto, uno studente o una famiglia potrebbero essere interessati anche ai servizi immediatamente disponibili nelle vicinanze di tale immobile, come ad esempio la presenza di un asilo o di una biblioteca. Questi servizi potrebbero aiutare a trovare la zona più adatta alle proprie esigenze e giustificare eventuali differenze di prezzo. Tali informazioni sono molto importanti soprattutto nelle grandi città, come ad esempio Milano. Tuttavia, nei portali online, presenti attualmente sul mercato, manca un'integrazione di questo tipo, tra annunci immobiliari e servizi disponibili alle persone. Infatti, è possibile filtrare sulle diverse proprietà di un immobile, come numero di stanze, metratura, presenza di un parcheggio o aria condizionata, ma non è possibile specificare il fatto che si voglia, ad esempio, una piscina a non più di mezzo chilometro. Lo scopo di questo progetto è dunque quello di creare una possibile base dati, sottostante ad un portale online, che permetta agli utenti di effettuare delle ricerche mirate alle proprie esigenze.

II. Descrizione del problema

Non esistendo una base dati contenente le informazioni necessarie per questo nuovo portale si è reso necessario incrociare diverse fonti dati. Per quanto riguarda l'elenco degli annunci immobiliari in affitto e le loro proprietà, abbiamo deciso di affidarci al noto sito web Idealista[1]. Idealista è un'azienda spagnola, fondata nel 2000, che offre, tra i vari servizi, un portale immobiliare in Italia, Spagna e Portogallo. Al suo interno è possibile cercare case, appartamenti o stanze in affitto, con diverse proprietà e per diversi range di prezzo. La seconda fonte dati invece riguarda l'elenco dei servizi disponibili in una certa area, come scuole, impianti sportivi o negozi. In questo caso abbiamo utilizzato la versione online di Pagine Gialle[2]. Da questo abbiamo potuto ricavare nome dell'attività, tipologia dell'attività, indirizzo e quartiere coperto.

Per il nostro progetto ci siamo focalizzati unicamente sul territorio corrispondente alla città di Milano. L'obiettivo è quello di unire i due set di dati per fornire all'utente la possibilità di ispezionare i servizi vicini ad un particolare annuncio immobiliare, o viceversa, guardare gli annunci immobiliari vicini ad una particolare attività.

Il tutto verrebbe poi reso disponibile ai potenziali fruitori tramite un'applicazione web. All'interno di questa verranno rese disponibili tutte le informazioni riguardanti gli immobili, esattamente come su Idealista, al quale rimanderemo nel caso l'utente decida di contattare il venditore. Il valore aggiunto è che nel momento in cui verrà selezionato un particolare annuncio verranno caricati tutti i servizi

[1] https://www.idealista.it/it/

[2] https://www.paginegialle.it/

limitrofi. Di default, nella nostra base dati, memorizzeremo per ogni immobile, i servizi presenti in un range di 500 metri, in maniera tale da tenere il sistema molto veloce. L'utente potrà poi impostare i propri filtri o modificare il range. Questa operazione richiederà una ricerca specifica. I dati verranno controllati e aggiornati ogni 24 ore.

III. Dataset immobiliare

Dal sito di Idealista si vuole estrarre l'elenco degli annunci, relativi agli affitti, attivi in questo momento sul territorio della città di Milano. Agli sviluppatori Idealista mette a disposizione un'interfaccia web (API) per poter cercare ed estrarre tutti i dati, catastali e non, relativi agli immobili in affitto presenti all'interno loro portale. I dati vengono aggiornati ogni 24 ore e per poter utilizzare tale API è necessario fare richiesta tramite apposito form.
Abbiamo fatto richiesta per due volte, ma, in entrambi i casi, non abbiamo ricevuto un riscontro. Inoltre, non sapendo se le API avessero delle limitazioni, come un numero massimo di chiamate giornaliere, abbiamo deciso di cambiare approccio e utilizzare tecniche di scraping. Con Web Scraping si intende un insieme di metodi utilizzati per raccogliere informazioni da Internet. In genere, si tratta di un software che simula la navigazione umana sul web per raccogliere informazioni specifiche da diversi siti internet. Per costruire ed aggiornare questo dataset la procedura si divide in due step:

1. **Download elenco annunci attivi**: Si itera tra le pagine presenti su Idealista, estraendo titolo e indirizzo url dei vari blocchi "div" rappresentanti i singoli annunci.

2. **Download delle informazioni e delle proprietà di ogni annuncio**: Raccolti tutti i riferimenti alle pagine le si scaricano o le si aprono, e si estraggono tutte le proprietà che costituiscono le caratteristiche dell'immobile, come la grandezza in metri quadri, il numero di stanze, l'indirizzo, etc.

La suddivisione in due passaggi si è resa necessaria per rendere l'esecuzione di raccolta e di controllo di dati già presenti più rapida. Inoltre abbiamo dovuto affrontare una limitazione di Idealista nel mostrare gli elementi di una richiesta. Durante una ricerca, infatti, non è possibile andare oltre la pagina 60 (che corrisponde a 1800 annunci), e di conseguenza potrebbe non essere possibile estrarre tutti gli annunci disponibili. Abbiamo quindi appli-

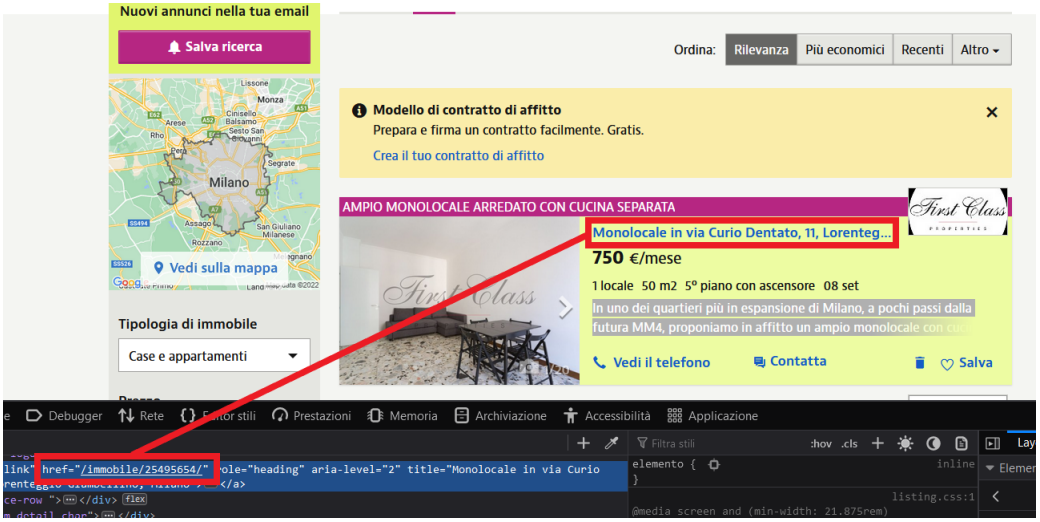

Figure 1: *Idealista risultati ricerca*

cato, uno ad uno, i possibili filtri applicabili alla ricerca, creando diversi set di dati. Gli annunci di ogni ricerca, identificati dalla coppia ¡titolo, url¿, venivano poi salvati su un file csv. Al termine dell'esecuzione abbiamo unito tutti i file csv ed eliminato i duplicati. In totale abbiamo applicato 23 diversi filtri diversi, riferiti al numero di stanze, al numero di bagni, e possibili spazi/servizi, come garage o aria condizionata.

Eliminati i duplicati abbiamo ottenuto circa 4.300 elementi, rispetto ai circa 4.800 disponibili. Idealista ordina sulla base dei più rilevanti e dei più recenti, quindi gli annunci non presenti, e che sono quindi stati scartati, appartenevano comunque ad annunci particolarmente datati. A questo punto era necessario effettuare lo scraping, e quindi il parsing, delle singole pagine degli annunci e memorizzare le loro proprietà in una base dati alternativa al csv.

Inizialmente, per entrambe le procedure, effettuavamo una richiesta get per la pagina specifica e poi tramite Beautiful Soup facevamo il parsing della pagina. Tuttavia, immediatamente, o dopo poche iterazioni, Idealista identificava che si trattasse di un sistema automatico e tutte le richieste successive venivano bloccate. A questo punto abbiamo deciso di simulare il comportamento di un utente umano attraverso Selenium[3]. Si tratta di un progetto open source che raggruppa una serie di strumenti e librerie per l'automazione dei browser. Di fatto si tratta della libreria più utilizzata per questo genere di task.

Per ogni url disponibile viene aperta una nuova sessione del browser, in questo caso Firefox, tramite Selenium e si effettua lo scraping dell'annuncio, scaricando le proprietà fisiche dell'immobile, le apparecchiature, la posizione e le informazioni del venditore. Le pagine di Idealista sono organizzate in maniera molto chiara e gli identificatori o le classi css dei vari blocchi sono immediate nel nome. Ad esempio per il prezzo avevamo la classe "*info-data-price*" oppure le informazioni dettagliate sulla proprietà erano nel div con classe "*details-property*".

Annunci diversi potevano avere più o meno

[3]https://www.selenium.dev/

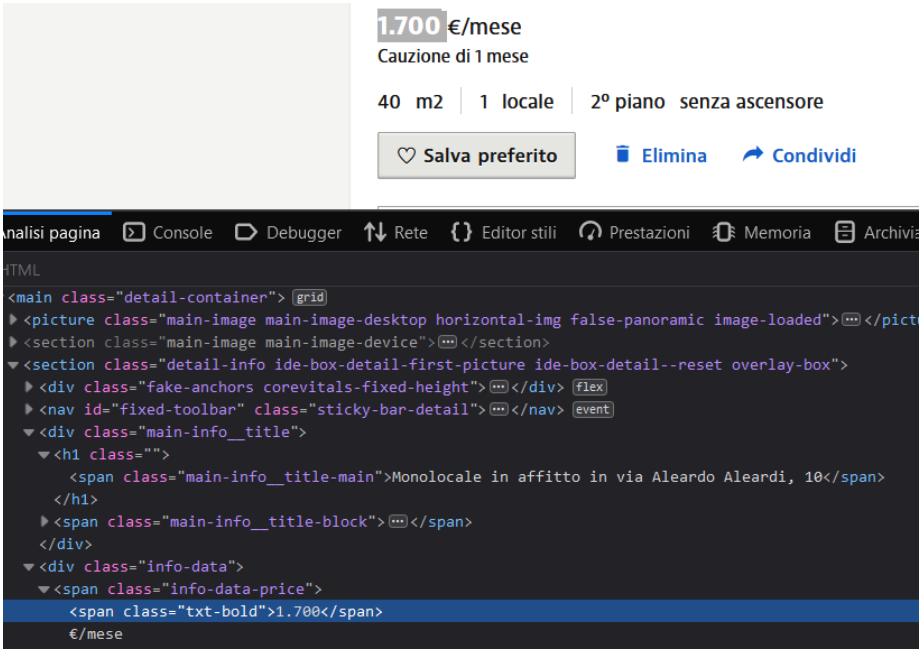

Figure 2: *div del prezzo*

```python
#Features
features = driver.find_element(By.CLASS_NAME,'info-features')
if len(features.find_elements(By.CLASS_NAME, "info-urgent")) ≠ 0:
    house['occasione'] = True
else:
    house['occasione'] = False

features_details = features.find_elements(By.TAG_NAME,'span')
features_len = len(features_details)

if features_len ≥ 2:
    house['metratura'] = features_details[0].text

if features_len ≥ 4:
    house['numero locali'] = features_details[2].text

if features_len ≥ 5:
    house['posizione'] = features_details[4].text
```

Figure 3: *Campi opzioni della proprietà*

campi, in riferimento alle proprietà degli immobili, e di conseguenza abbiamo deciso di utilizzare un approccio dinamico e non a struttura fissa per quanto riguarda la memorizzazione dei dati. Per tale motivo abbiamo utilizzato un database documentale, nel nostro caso MongoDB. MongoDB permette di avere accesso, in maniera del tutto gratuita, ad un'istanza remota in un macchina condivisa. Le caratteristiche di questa istanza sono ovviamente molto basilari, ma per i nostri scopi e per la quantità di dati che avremmo dovuto memorizzare era più che sufficiente. La connection string per connettersi al database remote è la seguente:

```
conn strin: mongodb+srv://sa:
↪  DataMan2022@cluster0.hdw3axi.mongodb.net/test
```

Come già verificatosi nel caso di Beautiful Soup, anche utilizzando Selenium, dopo un certo numero di esecuzioni, 150 circa, il sistema rilevava che si trattasse di un software automatico e di conseguenza iniziava a bloccare le richieste successive mostrando una pagina in cui era richiesto il superamento di un recaptcha.
Per ovviare a questo problema abbiamo utilizzato un servizio di proxy. Come primo tentativo abbiamo provato ad utilizzare i proxy forniti gratuitamente dal sito https://free-proxy-list.net/. Tuttavia questi proxy venivano immediatamente identificati come bot alla prima iterazione della procedura di scraping. Abbiamo così deciso di abbonarci alla prova di 3 giorni del servizio premium. Per $2.95 abbiamo avuto a disposizione una lista di 3000 proxy, aggiornata ogni ora.
Con una semplice chiamata HTTP ci veniva restituita una lista di proxy da utilizzare. In questo modo è stato possibile procedere con

We update the proxy list **every 30 minutes**. Open this API URL to get it.

http://list.didsoft.com/get?email=p.comensoli@campus.unimib.it&pass=[redacted]
pid=http3000&showcountry=no

Figure 4: *Richiesta elenco proxy*

```
def import_data(url, proxy):

    webdriver.DesiredCapabilities.FIREFOX['proxy'] = {
        "httpProxy": proxy,
        "sslProxy": proxy,
        "proxyType": "MANUAL",
    }
```

Figure 5: *Impostazione proxy Firefox*

un proxy fino a quando non veniva bloccato. A quel punto è stato sufficiente passare al successivo, procedendo così fino a fine lista. Una volta completata abbiamo proceduto con una nuova chiamata, ricominciando da capo la lista. In questo modo la procedura ha funzionato ed è riuscita a coprire l'intero set di url in qualche ora.
Utilizzando Firefox come browser nella procedura di scraping, è possibile impostare il proxy da utilizzare direttamente nei parametri di quest'ultimo senza utilizzare librerie aggiuntive. Disponendo ora di tutti i dati, e di tutti gli indirizzi, prima di procedere con i task di pulizia e miglioramento dovevamo trovare ed estrapolare un set di dati per i servizi.

IV. Dataset Servizi

Anche il secondo set di dati è stato ricavato attraverso tecniche di web scraping, in questo caso applicate al sito "paginegialle.it". Sul sito sono presenti numerose categorie di interesse, tra le quali sono state identificate quelle presenti in questo dataset. Nello specifico sono state identificate 5 aree tematiche, all'interno delle quali sono presenti diverse categorie:

- **Fare la spesa**: 'alimentari', 'discount', 'frutta e verdura', 'macellerie', 'panetterie', 'pescherie'
- **Servizi**: 'asili nido', 'banche', 'lavanderie', 'ricevitorie', 'tabaccherie'
- **Mangiare**: 'bar', 'ristoranti', 'gelaterie'
- **Pubblica utilità**: 'farmacie', 'parrocchie',

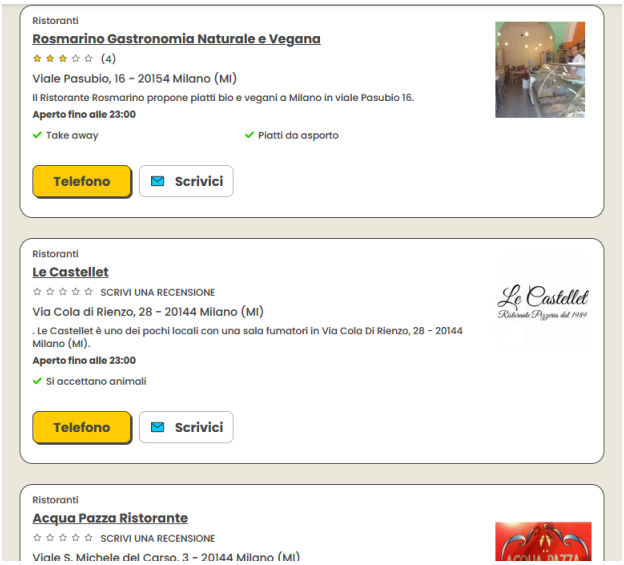

Figure 6: *Ricerca su pagine gialle*

'scuole', 'uffici postali'
- **Sport e tempo libero**: 'cinema', 'librerie', 'palestre', 'teatri'

Durante la navigazione sul sito per ciascuna ricerca vengono mostrati un massimo di 200 risultati. Pertanto, per evitare una raccolta parziale dei dati interessati, per ciascuna categoria si è scelto di ricercare i punti di interesse per quartiere. In questo modo è stato possibile fare in modo che, per ciascuna ricerca, gli elementi restituiti fossero meno di 200 e fosse pertanto possibile visualizzarli tutti. Ciò ha comportato in alcuni casi delle ripetizioni. Per ciascun oggetto nel dataset viene quindi riportato il nome, la categoria, l'indirizzo e il quartiere di appartenenza.

V. Data Editing

Operazioni eseguite:

- Rimozione duplicati
- Normalizzazione delle vie
- Ricerca coordinate e filtraggio

Nel dataset dei servizi non sono presenti valori mancanti. Solo in 7 casi l'indirizzo non è completo (compare solo città e CAP) e, pertanto, si è deciso di eliminare tali documenti.
L'espediente dei filtri ha causato la presenza di numerose ripetizioni all'interno del dataset. Nello specifico abbiamo riscontrato che uno stesso luogo di interesse, con lo stesso indirizzo, può appartenere a più di un quartiere. Questo avviene quando il luogo di interesse si trova al confine tra più quartieri.
Un'altra ripetizione individuata all'interno del dataset riguarda invece le categorie: uno stesso luogo può essere catalogato in diverse categorie.
Le ripetizioni sono state gestite mantenendo un solo documento, all'interno del quale sono stati inseriti dei nuovi campi contenenti rispettivamente la lista dei quartieri in cui il luogo descritto dal documento veniva localizzato e la lista delle categorie in cui veniva catalogato.
Anche nel primo dataset sono state individuate delle ripetizioni. In questo caso alcuni annunci venivano ripetuti interamente (stesso 'url'), pertanto tali ripetizioni sono state eliminate. In altri casi, invece, erano presenti più versioni dello stesso annuncio (stesso nome, indirizzo, posizione nel palazzo e metratura), che differivano per alcuni dettagli e per il campo 'last update', che indica l'ultima data di modifica dell'annuncio. In questo caso è stata mantenuta la versione con la data più recente, mentre le altre sono state eliminate.
In entrambi i dataset, infine, sono stati standardizzati gli indirizzi che non sempre vengono riportati nella forma corretta (ad esempio il numero civico veniva riportato prima della Via). Gli indirizzi sono quindi stati uniformati e riportati tutti al formato standard (via/corso/piazza + nome della via + numero civico + CAP (se disponibile) + città + sigla della provincia).
Per poter unire i due dataset è necessario disporre delle coordinate geografiche, sia degli


```
{
    "category": "alimentari",
    "name": "Tropical Market",
    "address": "70, Via Elio Adriano - 20128 Milano (MI)",
    "quartiere": "Quartiere Adriano"
},
{
    "category": "alimentari",
    "name": "La Bottega delle Cialde Vpp di Cagnazzi Simone",
    "address": "Via Padova, 281 - 20127 Milano (MI)",
    "quartiere": "Quartiere Adriano"
},
```


Figure 7: *Elenco JSON estratto*

annunci, che dei servizi. MongoDB ci viene incontro da questo punto di vista e ci permette di confrontare due punti geospaziali diversi e di calcolare la distanza.
Una volta normalizzati i dati relativi agli indirizzi siamo quindi passati al calcolo delle coordinate geografiche. Inizialmente abbiamo utilizzato le api di Google Maps, ma nella loro forma gratuita sono limitate sul numero di richieste che si possono mandare in una giornata. Alla fine abbiamo utilizzato le api di Bing Maps[4]. Le api di Microsoft, nella forma gratuita, permettono un massimo di 125.000 richieste durante un anno solare, senza limitazioni sul numero giornaliero.
Il formato della richiesta era il seguente:

```
request_url =
↪  f'http://dev.virtualearth.net/REST/v1/
↪  Locations/IT/MILANO/{addressLine}?
↪  key={BING_MAPS_KEY}'
-------------------------------------
response = requests.get(request_url)
coordinates = response.json()
↪  ['resourceSets'][0]['resources'][0]
↪  ['geocodePoints'][0]['coordinates']
print(coordinates)
-------------------------------------
[45.4452, 9.17748]
```

[4]https://docs.microsoft.com/en-us/bingmaps/

Il risultato, in caso di risposta con successo, erano i punti di latitudine e longitudine. La procedura è stata applicata a tutti gli elementi dei due dataset.
Alcuni punti, come si può vedere in Figura [8], non sono stati riconosciuti correttamente. In questo caso si può procedere in diversi modi:

- Eliminazione degli annunci non riconosciuti correttamente
- Utilizzo di una seconda libreria, come le api di Google Maps
- Ricerca manuale delle coordinate geografiche

Nel nostro progetto abbiamo deciso di procedere con la cancellazione di tutti quegli annunci che non appartenevano all'area di Milano.
Nei due dataset per quanto riguarda la pulizia del testo non sono state effettuate grandi modifiche perchè non era necessario. Abbiamo utilizzato solamente una funzione che eliminava la punteggiatura all'interno del dataset che non era appropriata e l'abbiamo applicata alle variabili che la necessitano.

Figure 8: *Coordinate ottenute*

```
PUNCTUATIONS = string.punctuation.replace('Âº',' ')

def remove_punctuation(text):
    trans = str.maketrans(dict.fromkeys(PUNCTUATIONS, ' '))
    return text.translate(trans)
```

Figure 9: *Pulizia dati*

```
def remove_whitespaces(text):
    return " ".join(text.split())

houses['title'] = houses['title'].apply(lambda x: remove_whitespaces(x))
houses['location'] = houses['location'].apply(lambda x: remove_whitespaces(x))
houses['metratura'] = houses['metratura'].apply(lambda x: remove_whitespaces(x))
houses['posizione'] = houses['posizione'].apply(lambda x: remove_whitespaces(x))
houses['decriptions'] = houses['decriptions'].apply(lambda x: remove_whitespaces(x))
houses['propertyDetails'] = houses['propertyDetails'].apply(lambda x: remove_whitespaces(x))
houses['constructions'] = houses['constructions'].apply(lambda x: remove_whitespaces(x))
houses['equipments'] = houses['equipments'].apply(lambda x: remove_whitespaces(x))
houses['advertiser'] = houses['advertiser'].apply(lambda x: remove_whitespaces(x))
houses['fullAddress'] = houses['fullAddress'].apply(lambda x: remove_whitespaces(x))
houses['numberOfPictures'] = houses['numberOfPictures'].apply(lambda x: remove_whitespaces(x))
```

Figure 10: *Pulizia dati*

Oltre a questa, un'atra funzione per unificare gli spazi bianchi tra le parole applicandola a tutte le variabili del dataset.

VI. Integrazione

Come anticipato, un modo a nostra disposizione per poter integrare spazialmente i due set di dati è tramite il calcolo della distanza, in metri, tra le coordinate di ogni annuncio rispetto all'elenco dei servizi. La nostra idea per il nostro portale è quella di caricare in maniera rapida i servizi disponibili in un raggio di 500 metri dall'abitazione e permettere all'utente solo in un secondo momento di impostare i filtri e le distanze che più preferisce.

Per fare questo abbiamo deciso di utilizzare un approccio "nested". All'interno di ogni documento "house" inseriremo una lista di documenti "service", che corrispondono ai servizi la cui distanza massima è di mezzo chilometro dalle coordinate dell'annuncio. Come prima cosa dobbiamo impostare un indice geospaziale nelle due collezioni.

```
db.houses.create_index([("loc", GEO2D)])
db.services.create_index([("loc", GEO2D)])
```

Fatto questo tramite una query in mongo possiamo impostare la distanza massima che desideriamo, nel nostro caso 500 metri.

```
for house in houses.find():
    services_at_500_m = []
    loc = house['loc']
    geo_query =
    ↪  {"loc":{"$nearSphere":{"$geometry":
    ↪  {"type": "Point", "coordinates": loc},
    ↪  "$maxDistance": 500}}}

    for service in services.find(geo_query):
        services_at_500_m.append(service)

    house['services'] = services_at_500_m
    houses_services.insert_one(house)
```

Il risultato viene salvato in una nuova collezione, chiamata "houses_services". Questa integrazione risulta essere molto rapida e immediata, soprattutto grazie al fatto che abbiamo eliminato i possibili dati sporchi e incompleti.

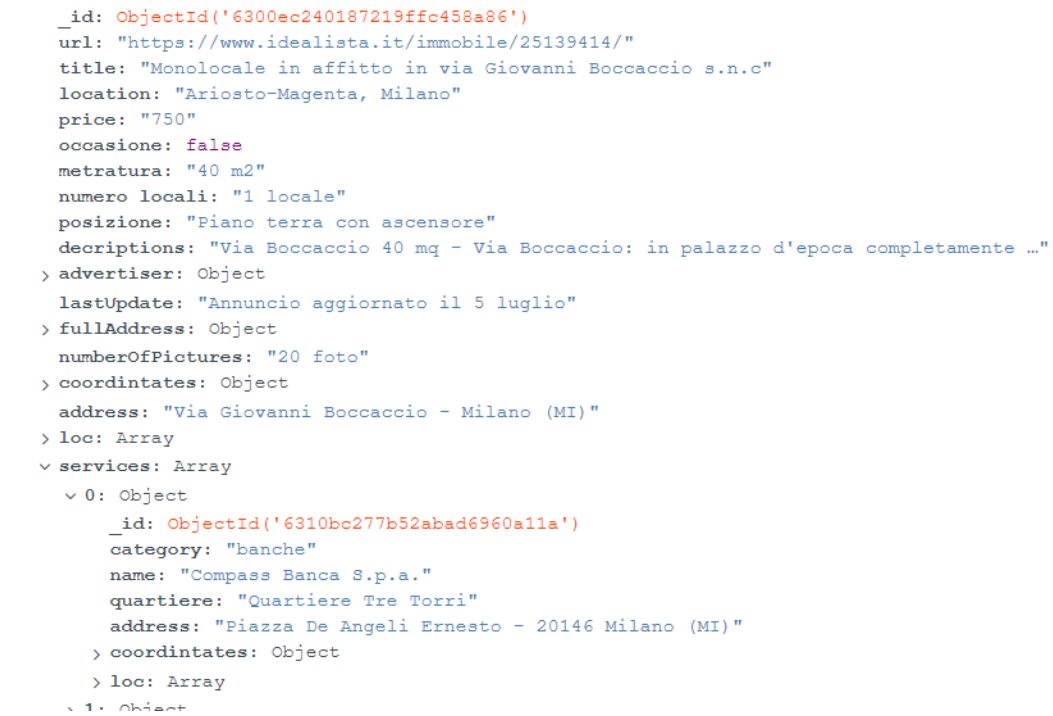

Figure 11: *Primo elemento nuova collection*

Al caricamente di un annuncio sarà dunque istantaneo anche il caricamento dei servizi più prossimi a tale immobile.

VII. Applicativo finale

Una volta ottenuto il nuovo dataset, con gli annunci e relativi servizi più prossimi, abbiamo ipotizzato un possibile software da rivolgere all'utente finale. Si tratta di un portale web in cui l'utente riceve, oltre alle informazioni prettamente legate all'immobile, anche le informazioni sui servizi disponibili nell'area. L'utente poi potrà impostare i propri filtri e il sistema sottostante modificherà query di pymongo.

I dati relativi agli annunci, come idealista, sarebbero aggiornati su base quotidiana, mentre i servizi verrebbero aggiornati meno frequentemente come una volta alla settimana o ogni due settimane.

VIII. Conclusioni e possibili sviluppi

I dati sugli affitti potrebbero essere recuperati anche da altri siti oltre a Idealista, come ad esempio Subito oppure OffroCasa. In questo caso ci sarebbe un rischio maggiore di duplicati, che potrebbero differire anche dalla struttura in cui sono disponibili nei vari portali.
Oltre agli annunci anche l'elenco dei servizi potrebbe essere espanso cercando altre fonti dati, o integrando tali dati con informazioni disponibili tramite social, per avere anche indicazioni quali valutazioni, etc.
Per quanto riguarda l'aggiornamento della base dati relativa agli annunci, ogni 24 ore, nel momento in cui si scarica l'elenco degli url degli annunci sarebbe necessario confrontarli con quelli presenti attualmente nella collezione di Mongo. Gli url presenti solo in Mongo dovrebbero essere cancellati in quanto probabilmente scaduti, mentre quelli mancanti dovrebbero essere aggiunti. Di conseguenza

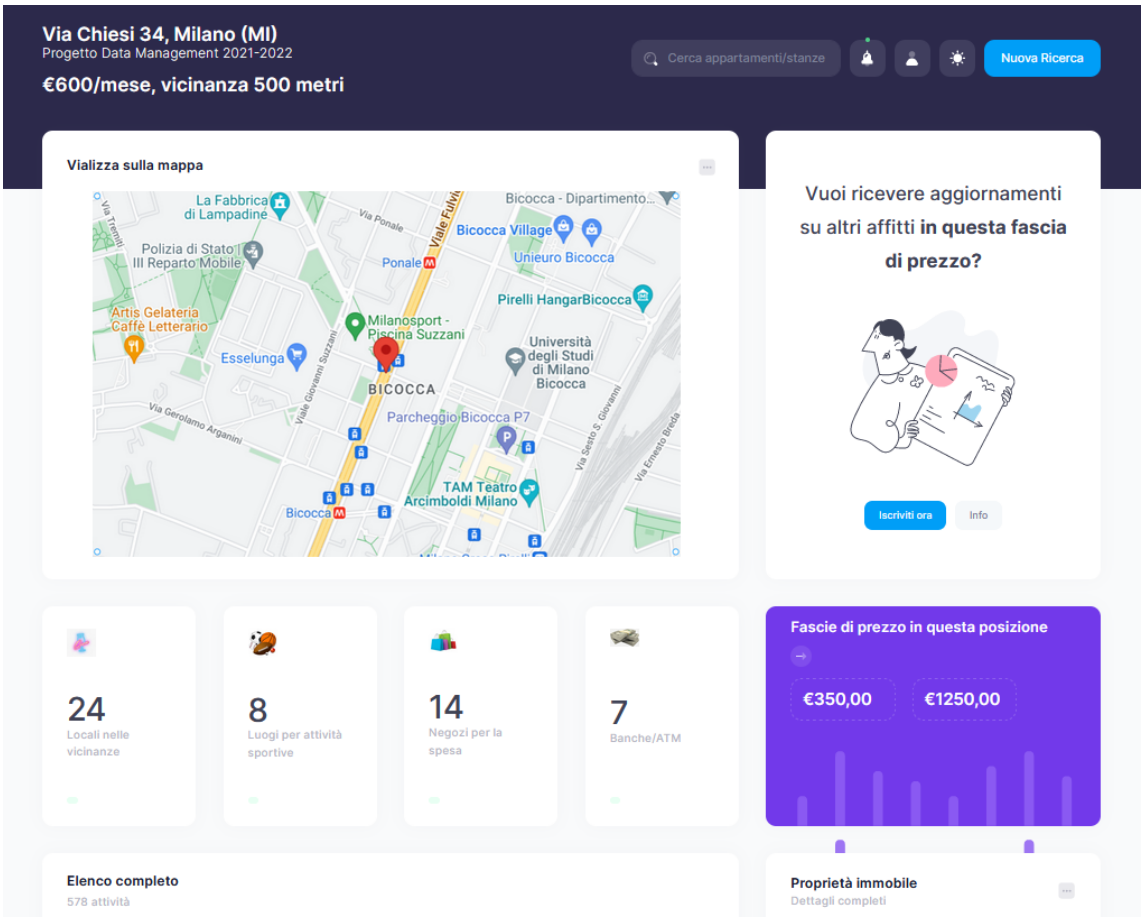

Figure 12: *Mockup applicativo*

l'esecuzione dell'intera pipeline dovrebbe essere più veloce rispetto ad una esecuzione ex novo, in quanto è molto difficile un ricambio completo degli annunci in un lasso di tempo breve.