Università degli Studi di Milano Bicocca

Scuola di Scienze

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di laurea in Informatica

MoonUtilities: una dashboard per la business intelligence

Relazione della prova finale di:

Stella Cervini

Matricola 847060

Anno Accademico 2020-2021

Abstract

Nello svolgimento delle sue attività, un’azienda produce dei dati, i quali riguardano aspetti eterogenei della vita della stessa e, se adeguatamente misurati, portano valore all’interno dell’impresa.

Al fine di estrarre informazioni utili dai dati, è necessario che l’utente che intende usufruire di tale conoscenza (solitamente il management) si interroghi sugli obiettivi che desidera raggiungere e ponga domande coerenti con i dati che ha a disposizione. Una volta che queste sono state formulate, la risposta è successivamente estratta dalle informazioni raccolte. Per questo motivo, i dati necessitano di essere collezionati, elaborati ed esposti in modo chiaro, così da poter essere compresi dall’utilizzatore finale, il quale, successivamente, può interpretarli al fine di prendere delle decisioni maggiormente motivate ed efficaci.

Il termine “business intelligence” fa riferimento all’insieme di questi processi aziendali, ossia quei processi che hanno lo scopo di raccogliere dati ed analizzare informazioni strategiche.

L’obiettivo di questo elaborato è di mostrare quanto svolto durante lo stage curricolare, ovvero la realizzazione di una dashboard in linguaggio R per la visualizzazione di dati aziendali.

L’azienda in questione si occupa di offrire servizi di fornitura luce e gas a clienti e partner. Collabora da un lato con i fornitori di tali beni e dell’altro con le agenzie immobiliari, proponendo loro i prodotti di fornitura di energia elettrica e gas offerti dai grossisti. Il compito delle agenzie è quello di presentare e vendere tali prodotti al consumatore finale. In questo contesto, per l’azienda risulta fondamentale riuscire a monitorare l’operato dei collaboratori.

Lo strumento sopra citato permette di verificare l’andamento delle vendite sulla rete delle agenzie, consentendo di controllare in tempo reale l’attività e accertare che sia in linea con gli obiettivi precedentemente stabiliti. In questo modo, il management può intervenire tempestivamente nel caso in cui le stime non siano rispettate.

In questa relazione sono illustrati, oltre alle ragioni e le necessità che hanno spinto allo sviluppo della dashboard, i mezzi utilizzati per la sua realizzazione e sono discusse le scelte implementative, tra i quali possono essere citati, per esempio, la modalità di importazione dei dati, i pacchetti utilizzati e l’approccio adoperato per la creazione e visualizzazione dei grafici.

Indice

1. Introduzione 4
2. Contesto: Moon Energy S.r.l. 5
    2.1 Descrizione dell'attività aziendale 5
    2.2 Struttura 5
    2.3 Rete Moon 6
    2.4 Applicazione della business intelligence 8
3. Strumenti utilizzati 10
    3.1 Introduzione della dashboard Moon Utilities 10
    3.2 Il linguaggio R 10
    3.3 Aspetti principali dell'implementazione 14
    3.4 Gestione dell'interattività 19
4. Dati 24
    4.1 Raccolta 24
    4.2 Importazione 25
    4.3 Pulizia e trasformazione 29
    4.4 Visualizzazione 36
5. Ottimizzazione 42
6. Conclusioni 46
Sitografia 48

1. Introduzione

L'analisi dati viene definita come l'insieme di metodologie e processi volti all'esplorazione, pulizia, trasformazione e modellazione di dati precedentemente raccolti, con l'intento di estrarre conoscenza utile [1].

L'analisi dati si può segmentare in varie branche caratterizzate da tecnologie, finalità e approcci eterogenei fra loro, i quali portano, di conseguenza, a definire ambiti differenti.

In questo elaborato, ci si focalizzerà sulla business intelligence (BI), ambito strettamente correlato all'analisi dati. Per business intelligence s'intende il processo di raccolta, archiviazione e analisi dei dati generati dalle operazioni aziendali. La BI si concentra su un'analisi descrittiva, così da poter riepilogare dati passati ed attuali. Per questo aspetto, si differenzia dalla business analytics (BA), che privilegia l'analisi predittiva, cioè con il fine di prevedere ciò che potrebbe accadere in un futuro più o meno prossimo [2].

Un primo esempio storico di questo approccio è rappresentato dagli studi di Frederick Taylor, il quale nel XIX secolo analizzò l'organizzazione e i procedimenti all'interno delle fabbriche e teorizzò ciò che è stato definito come "taylorismo" oppure "scientific management", ovvero una teoria che mira a ottimizzare la produttività aziendale attraverso un migliore impiego della forza lavoro [3]. Raccogliendo delle osservazioni, Taylor fu quindi in grado di estrapolare informazioni fondamentali e prendere le decisioni che meglio potevano portare ai cambiamenti desiderati alla linea di produzione.
Lo sviluppo dei calcolatori e l'evoluzione della tecnologia informatica hanno permesso una svolta digitale dell'analisi dati, resa possibile dall'introduzione di mezzi in grado di raccogliere, immagazzinare e analizzare grandi quantità di dati.

Oggi si è compreso quanto l'analisi dei dati all'interno delle aziende possa supportare la dirigenza aziendale nel prendere delle decisioni motivate e maggiormente contestualizzate, aiutando le attività commerciali ad operare in modo più efficiente, tanto da divenire un concetto fondamentale all'interno delle strategie aziendali.

Allo scopo di compiere uno studio sui dati e di visualizzarne i risultati, è utile sviluppare degli strumenti che possano facilitare la consultazione delle informazioni e la loro comprensione. Questa relazione volge a descrivere la realizzazione di una dashboard che possa soddisfare questo bisogno.

2. Contesto: Moon Energy S.r.l.

In questo capitolo verrà descritta l'attività dell'azienda Moon Energy S.r.l., committente dello sviluppo della dashboard.

2.1 Descrizione dell'attività aziendale

Moon Energy S.r.l. si occupa di offrire servizi di fornitura luce e gas a clienti e partner, implementando soluzioni con configurazione B2B (business-to-business) e B2C (business-to-consumer), personalizzando l'offerta in modo da incontrare le esigenze personali del singolo.

L'attività può essere suddivisa in quattro fasi [4]:

- *analisi*: con l'obiettivo di offrire un servizio che possa soddisfare al meglio il cliente, fatture e contratti relativi a forniture di energia elettrica e gas vengono esaminati e confrontati con l'ausilio di parametri istituzionali e commerciali;
- *offerta*: si ricerca il servizio che corrisponde alla soluzione migliore per il cliente in termini di qualità ed economia, in modo da consigliare la proposta più vantaggiosa;
- *consulenza*: il cliente riceve un servizio di supporto attraverso un consulente dedicato, il quale provvede a soddisfare le necessità relative al contratto, sia prima della vendita che dopo la sua stipulazione;
- *gestione*: essendo le tariffe in continuo aggiornamento, si rende necessario rivedere i contratti al fine di proporre il prezzo più conveniente al cliente. Inoltre, periodicamente vengono svolte delle verifiche con lo scopo di acquisire informazioni sull'indice di gradimento dei servizi forniti.

2.2 Struttura

Moon Energy S.r.l. ha funzione di intermediario tra *mandanti*, ovvero società che producono e distribuiscono energia elettrica e gas, e l'*utente finale,* ossia il soggetto che usufruisce del servizio di fornitura. Tuttavia, spesso il collegamento non è diretto, bensì passa attraverso soggetti definiti come *corner* e *segnalatori*. Questi sono in prevalenza agenzie immobiliari che scelgono di lavorare con Moon Energy S.r.l. ed entrano a far parte della rete.

La differenza tra corner e segnalatori consiste nell'assunzione di competenza. Il corner si occupa di tutta la procedura di affiancamento del cliente descritta precedentemente, dall'analisi fino alla gestione, mentre il segnalatore ha solo il compito di comunicare i

dati del cliente a Moon Energy S.r.l., la quale provvederà a portare a termine le varie fasi del procedimento.

Le relazioni che sussistono tra l'azienda e gli altri soggetti sopra citati possono essere semplificate come nell'immagine in Fig. 1.

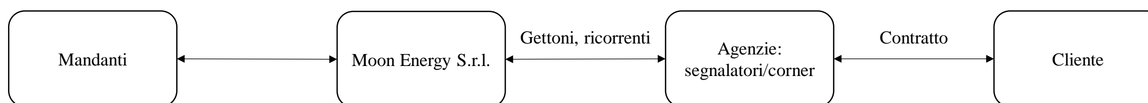

Fig. 1: schema riassuntivo delle relazioni sussistenti tra mandanti, Moon Energy S.r.l., corner/segnalatori e clienti.

Come riportato in precedenza, in questo ambito, i mandanti sono quelle società che producono energia elettrica e gas naturale. Queste imprese sono partner di Moon Energy S.r.l.. Ciò significa che quest'ultima si impegna a distribuire i loro prodotti e, allo stesso modo, le agenzie sono incaricate di vendere i prodotti dei mandanti (sotto commissione di Moon Energy S.r.l.), in cambio di una percentuale o di una somma prefissata calcolata sul valore dei contratti stipulati, denominata *gettone*, e di *ricorrenti*, vale a dire dei compensi periodici.

2.3 Rete Moon

La struttura rappresenta in Fig. 1 è una descrizione semplificata delle relazioni tra l'azienda e l'esterno. Infatti, l'organizzazione reale è più complessa ed è opportuno introdurre la *rete Moon*.
Per comprendere cosa si intende per "rete Moon" è utile, in principio, fare un'osservazione sulle agenzie. Di fatto, le agenzie immobiliari possono essere dei singoli punti vendita, dei franchising oppure fare parte di società le quali, a loro volta, possono essere contenute in holding di dimensioni maggiori che raggruppano più organizzazioni. In questi ultimi casi, è verosimile che si crei una gerarchia interna per la gestione, dove al livello inferiore si trova un'agenzia, mentre in cima è posto il gruppo aziendale. All'interno di questo ordinamento si possono trovare anche dei livelli intermedi, necessari per il corretto funzionamento del sistema.

La rete Moon è l'insieme di tutte le gerarchie a cui appartengono corner e segnalatori (d'ora in poi denominati *point* per semplificare), dove il grado maggiore è riservato alla holding e a seguire sono presenti tutti gli altri livelli dei business partner, fino a raggiungere l'agenzia immobiliare.

È interessante notare che, essendo i vari gruppi delle entità indipendenti, ciascuno potrebbe avere una scala gerarchica differente. La rete Moon gestisce questa diversità applicando un modello standard, creato considerando la gerarchia con complessità maggiore tra quelle presenti nella rete (la complessità viene misurata in base al numero di livelli che una gerarchia possiede: più è alto il numero di livelli, maggiore sarà considerata la complessità).

In Fig. 2 viene riportato lo schema che rappresenta la rete Moon.

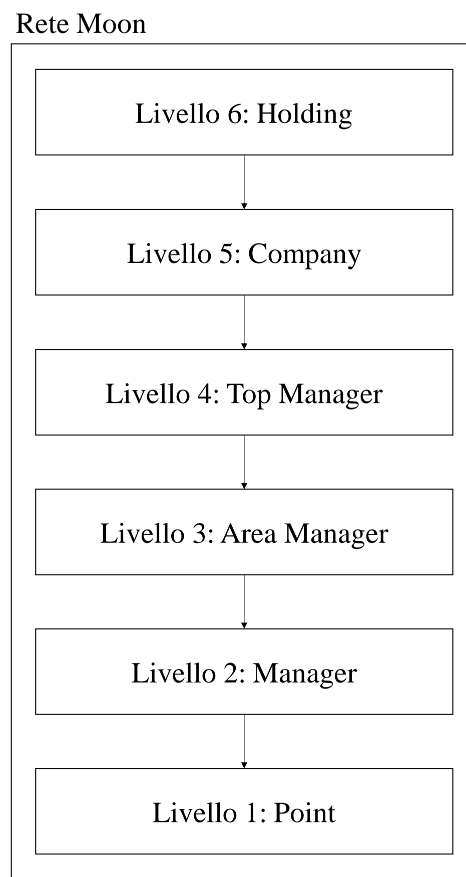

Fig. 2: schema riassuntivo della rete Moon.

Come appena spiegato, questa disposizione si riferisce al caso più complesso. Ci si potrebbe interrogare su cosa accada quando si presenta la necessità di rappresentare un point che non fa parte di nessun gruppo oppure quando non tutti i livelli sono coperti, dato che ogni gruppo ha la propria organizzazione interna.

In questi casi, si è scelto di inserire al posto dei campi non occupati il valore zero in modo che abbia funzione di valore fittizio (o *placeholder*), sia all'interno del database ma anche a livello logico. Questa implementazione è motivata dal fatto che è risultato più semplice in ambito di implementazione di database avere un numero fisso di livelli, nonostante questi siano "vuoti", rispetto a creare una struttura di lunghezza variabile che si adatti al

numero di livelli effettivi. Sarà competenza dell'utente interpretare in modo corretto la configurazione.
È fondamentale conoscere i livelli relativi ad ogni point, sia esso corner o segnalatore, poiché ricorrenti e gettoni sono da distribuire su tutta la rete a cui appartiene.
Nel momento in cui il point entra a far parte della rete, viene inserito all'interno del portale Moon e, oltre a tutte le informazioni anagrafiche, è richiesta l'introduzione delle relazioni sussistenti con i livelli soprastanti, se esistono.

Solitamente, tra le agenzie e gli altri soggetti della gerarchia esiste una relazione n ad 1, vale a dire che un'agenzia ha esattamente un solo manager, area manager, top manager, company ed holding. Al contrario, un area manager può essere responsabile di più manager, i quali si occupano di più agenzie, così come un top manager può coordinare più area manager. Similmente, ad una company possono appartenere più top manager ed una holding è un insieme di company.
La suddivisione in livelli gerarchici provoca la conseguenza di avere all'interno della rete dei soggetti con ruoli eterogenei e, coerentemente, anche dei privilegi, funzioni e mansioni differenti.

Vista la continua espansione della rete Moon, è essenziale riuscire a tenere traccia di tutti gli attori che ne fanno parte e, grazie a questa struttura, le relazioni tra soggetti sono idealmente sempre consistenti e aggiornate.

2.4 Applicazione della business intelligence

Per la direzione aziendale diventa fondamentale avere un quadro chiaro dell'operato dell'azienda, con particolare riguardo all'espansione della rete e il lavoro dei vari point. Senza di esso, infatti, non è possibile conoscere l'andamento delle vendite e, successivamente, rimunerare i point per le loro prestazioni. Pertanto, risulta utile possedere uno strumento che possa contenere le informazioni utili e visualizzarle in modo coerente per poterne comprendere il significato.

"Con la locuzione business intelligence ci si può solitamente riferire a: un insieme di processi aziendali per raccogliere dati ed analizzare informazioni strategiche; la tecnologia utilizzata per realizzare questi processi; le informazioni ottenute come risultato di questi processi." (*Business Intelligence*, Wikipedia) [5].

Considerando l'ambito descritto nei capitoli precedenti, è possibile notare che la realizzazione della dashboard, intesa come strumento per la visualizzazione di dati raccolti ed analizzati, rientra a pieno nella definizione sopra riportata.

È con riferimento alla BI che è stato deciso di sviluppare *MoonUtilities*, una dashboard che potesse riassumere i dati generati dal lavoro dell'azienda, estesa a tutta la propria rete.

3. Strumenti utilizzati

Questo capitolo si occupa di discutere le scelte progettuali effettuate e le motivazioni che hanno portato a tali decisioni.
In particolare, viene introdotto il linguaggio R e le peculiarità che hanno condotto alla sua scelta per l'implementazione della dashboard.
Inoltre, sono illustrate le caratteristiche principali del software, tra cui gli aspetti essenziali per lo sviluppo e gli elementi maggiormente significativi.

3.1 Introduzione della dashboard *MoonUtilities*

Una dashboard è un tipo di interfaccia grafica, contraddistinta da viste immediate che riguardano gli aspetti d'interesse di un particolare dominio o processo aziendale [6].
Questi oggetti grafici rendono le dashboard un mezzo potente per monitorare l'attività aziendale, in quanto mostrano le metriche più rilevanti in un'unica visuale.

La funzione principale della dashboard in questione è di contenere e visualizzare, tramite grafici e tabelle, le informazioni raccolte, in modo da costituire un pannello di controllo.

Questo cruscotto è utilizzato da più utenti. Precisamente, oltre ai soggetti che operano internamente a Moon Energy S.r.l., lo strumento è sviluppato in modo tale da essere utile anche a manager, company e holding che appartengono alla rete Moon, così che ognuno possa occuparsi della propria sottorete.
Ovviamente, le informazioni a cui ogni utente ha accesso sono solo quelle di propria competenza, così da permettere di mantenere la riservatezza dei dati ma anche la loro correttezza e coerenza con la realtà.

3.2 Il linguaggio R

La dashboard *MoonUtilities* è stata creata in linguaggio R, tramite l'IDE di programmazione RStudio.

R è un linguaggio di programmazione libero e un ambiente per il calcolo statistico e la rappresentazione grafica. È ampiamente utilizzato in questo settore e fa parte dei linguaggi più adoperati nell'ambito dell'analisi dati [7].
Il linguaggio fornisce una grande varietà di funzioni per la statistica e la costruzione di grafici, i quali rappresentano uno dei suoi punti di forza. Infatti, è possibile rappresentare i dati tramite grafici di alta qualità e personalizzabili, in modo da poter includere tutte le

informazioni necessarie per far comprendere meglio all'osservatore ciò che si vuole rappresentare [8].

Inoltre, R è fortemente estensibile, grazie ai pacchetti, spesso sviluppati da utenti esperti del linguaggio. Un pacchetto è un contenitore che include codice, dati e documentazione in una collezione standardizzata che è possibile installare sulla propria macchina e includere nei progetti, se necessario.
L'opportunità di utilizzare i pacchetti è una caratteristica particolarmente favorevole poiché permette di migliorare la produttività dello sviluppatore, riducendo il tempo impiegato a scrivere codice e fornendo funzioni ottimizzate e testate. Includere nel proprio codice una funzione che è già stata definita e non doverla scrivere da capo, permette di potersi concentrare su come applicare la soluzione al problema da risolvere piuttosto che ripensare alla soluzione stessa [9]. Un semplice esempio è rappresentato dal calcolo della media aritmetica di un certo numero *n* di valori. È possibile calcolare quanto desiderato eseguendo prima la somma di tutti i numeri e successivamente dividere tale somma per n. Queste due operazioni, anche se molto banali, richiedono più di una riga di codice e maggiore è il numero di elementi da considerare, maggiore sarà la complessità del calcolo. Per evitare errori, è possibile usare la funzione `mean()` [1] del pacchetto {base} [2], già implementata e collaudata in R. Nonostante il caso appena riportato non sia complicato, è esemplificativo del concetto che si voleva dimostrare.

Per fare un esempio riguardante alcuni dei pacchetti di cui si è usufruito durante lo sviluppo del codice, è possibile citare {shiny} e {bs4Dash}, i quali permettono di sviluppare delle applicazioni web in R, in questo caso una dashboard. Le applicazioni create tramite questi package sono dette "*Shiny app*".
Grazie a questi strumenti è stato possibile creare la user interface (UI) in modo rapido e curare la parte di interazione dinamica con l'utente, sfruttando le funzioni già implementate al loro interno. Inoltre, utilizzando questi package, risulta semplice integrare l'aspetto di analisi dati con il front-end dell'applicazione, senza necessità di scrivere ulteriore codice in HTML, CSS o JavaScript.
È interessante osservare che, in realtà, il codice scritto in linguaggio R per la creazione della UI della dashboard è codice HTML (viene interpretato successivamente come tale [10]). Il vantaggio di questa progettazione è che l'utente non è tenuto a conoscere i

[1] Le parentesi indicano che si tratta di una funzione.
[2] Per convenzione, i pacchetti sono indicati all'interno di parentesi graffe.

dettagli del linguaggio HTML per scrivere il codice occorrente a creare siti web. Inoltre, la sintassi necessaria a creare gli elementi di un sito web in linguaggio R potrebbe risultare più lineare rispetto a quella del classico HTML.

```
actionButton(
inputId = 'carica',
label = 'Aggiorna',
icon = icon('angle-double-right'),
width = '100%')
```

```
<button id="carica" style="width: 100%;" type="button" class="btn
btn-default action-button"> <i class="fa fa-angle-double-right"
role="presentation" aria-label="angle-double-right icon"></i>
Aggiorna </button>
```

Snippet 1: esempio di codice in linguaggio R e relativo codice HTML.

Il codice mostrato nello Snippet 1 mostra un esempio di quanto appena affermato, fornendo il codice necessario per la creazione di un bottone in linguaggio R e la relativa interpretazione in HTML, ottenuta eseguendo il codice all'interno di RStudio.

Per quanto riguarda la UI, in R è quindi possibile sviluppare qualsiasi elemento si possa scrivere in HTML. Questo vale anche per la personalizzazione dell'aspetto della dashboard. In questo caso, sfruttando il pacchetto {fresh} è stato possibile allineare l'apparenza del cruscotto al resto dei prodotti dell'azienda, come il portale in cui inserire le anagrafiche dei clienti per i nuovi contratti. La modifica ha riguardato principalmente i colori della dashboard, dato che quelli offerti di default dai pacchetti sopra citati non erano in linea con la brand identity del marchio.

Tuttavia, è possibile che non siano disponibili delle funzioni già implementate per realizzare quando si desidera sviluppare. Per fare un esempio si può citare una situazione incontrata durante lo sviluppo della dashboard. L'obiettivo era di inserire all'interno della stessa un elemento che, quando premuto dall'utente, modificasse la pagina (*tab*) visualizzata fino al momento del *click*, spostando il controllo ad un'altra pagina. Durante la realizzazione, non è stata trovata alcuna funzionalità già integrata che eseguisse questa operazione, nonostante sia una pratica diffusa nella programmazione web.

Il linguaggio R colma questa lacuna permettendo l'integrazione di codice JavaScript, HTML o CSS all'interno del progetto, tramite una lista di funzioni chiamate *tags*, appartenenti al pacchetto {tags}. Ogni funzione in questa lista crea un tag HTML di cui si può usufruire nella Shiny app [11]. Alcuni esempi di tags utilizzati

nell'implementazioni sono `tags$img()`, utilizzato per inserire l'immagine del logo aziendale all'interno della dashboard, e `tags$i()`, per modificare alcune sezioni del testo e creare l'effetto corsivo. Per raggiungere lo scopo sopra descritto, è stato utilizzato `tags$script()` per inserire la funzione JavaScript che eseguisse quanto richiesto e `tags$a()`, per chiamare la funzione ed eseguirla a seguito di un *on click* sull'elemento indicato.

```
tags$script(HTML("
        var openTab = function(tabName){
          $('a', $('.sidebar')).each(function() {
            if(this.getAttribute('data-value') == tabName) {
              this.click()};
          });
        }"))

HTML(paste(tags$a(tags$h5("Info",
       tags$i(fa('arrow-alt-circle right'),
       style = "font-size: 15px;")),
       onclick = "openTab('contratti')", href="#", style =
       'color: white;')))
```

Snippet 2: esempio di codice JavaScript inserito all'interno della dashboard.

Affinché la dashboard potesse essere pubblicata, si è reso necessario utilizzare *shinyapps.io*, una piattaforma self-service che permette di pubblicare online una applicazione Shiny in poco tempo [12]. Tramite questo servizio, è possibile ottenere l'URL relativo alla dashboard e condividere l'applicazione. È necessario puntualizzare che la Shiny app non è accessibile da chi non è possesso dell'URL.

L'azienda ha bisogno che i dati contenuti nella dashboard restino privati. Per assicurare la riservatezza, la dashboard non sarà semplicemente condivisa tramite l'URL, bensì è stato deciso di inserirla tramite un *iframe* HTML all'interno di una pagina nell'area privata del sito Moon Energy S.r.l.. Così facendo, per avere la possibilità di visualizzare lo strumento, è necessario possedere delle credenziali di accesso fornite e controllate dall'azienda. Queste credenziali attribuiscono anche un ruolo all'utente a cui sono associate, corrispondente alla funzione che svolge all'interno della rete Moon. Pertanto, è possibile che non tutti i ruoli possano accedere alla dashboard, ma, ad esempio, solo chi possiede un ruolo più alto nella gerarchia.
Questo accorgimento, oltre a permettere di controllare chi possa visualizzare i dati, rappresenta la soluzione alla necessità di filtrare i dati in base all'utente che accede alla dashboard, come viene illustrato nel capitolo successivo.

La decisione finale di utilizzare questo particolare linguaggio è stata, inoltre, affiancata da un rapido processo di studio di fattibilità e prototipazione. Una volta definite le caratteristiche e le funzionalità principali che lo strumento avrebbe dovuto contenere, si sono svolte delle ricerche e delle prove su quanto si sarebbe successivamente implementato. In particolare, si sono presi in esame i pacchetti e i metodi che avrebbero potuto rappresentare la soluzione per lo sviluppo delle necessità e si sono creati esempi e campioni. Così facendo, si sono potute verificare le potenzialità del linguaggio R, riscontrando che potesse fornire i mezzi necessari all'implementazione. Grazie alle sue peculiarità, R e tutti i suoi elementi sono risultati gli strumenti adatti per creare la dashboard.

3.3 Aspetti principali dell'implementazione

In precedenza, è stato citato il pacchetto {shiny}, affermando che si tratta dello strumento necessario per la realizzazione di Shiny app in linguaggio R. In particolare, queste applicazioni sono caratterizzate da due componenti fondamentali [13]:

- *user interface (UI)*: definisce l'apparenza dell'applicazione;
- *server*: funzione che definisce le attività che svolge l'applicazione.

Al fine di creare effettivamente una Shiny app, queste due parti sono passate come argomenti alla funzione `shinyApp()`, la quale crea un oggetto di tipo *shiny* dalla coppia UI/server [14].

La UI può essere vista come un oggetto al cui interno è presente il codice HTML fornito ad ogni utente che accede alla dashboard. Di conseguenza, chiunque acceda all'applicazione visualizza lo stesso contenuto. Al contrario, il server presenta una complessità maggiore, dato che quando un singolo utente interagisce con l'applicazione, gli altri non devono vedere le alterazioni che sta apportando alla pagina. Per garantire l'indipendenza, {shiny} invoca il server ogni volta che una nuova sessione ha inizio, sia quando un nuovo utente ha accesso alla dashboard, sia quando lo stesso utente la apre in due o più schede diverse del browser. Questo permette ad ogni sessione di avere uno stato autonomo e che le variabili create all'interno della funzione siano isolate [15]. Infatti, la funzione che rappresenta il server viene chiamata passandole tre parametri, ovvero *input*, *output* e *session*. Quest'ultimo è quello che garantisce l'indipendenza, in quanto univoco per ciascuna sessione e gli altri parametri sono strettamente collegati ad esso.

Nelle applicazioni di grandi dimensioni, oltre a questi due elementi, è utile avvalersi di altri componenti, dato l’alto rischio di ritrovarsi con i file di UI e server con un numero molto elevato di righe, rendendo la gestione più complicata. Per questo motivo, è utile sfruttare funzioni e moduli.

La definizione di *funzione* non appartiene solo al linguaggio R, poiché si tratta di un concetto comune a numerosi linguaggi di programmazione. In informatica e in programmazione, una funzione è “un particolare costrutto sintattico di un determinato linguaggio di programmazione che permette di raggruppare, all'interno di un programma, una sequenza di istruzioni in un unico blocco, espletando così una specifica (e in generale più complessa) operazione, azione (o elaborazione) sui dati del programma stesso in modo tale che, a partire da determinati input, restituisca determinati output” (*Funzione (Informatica)*, Wikipedia) [16].

Nel caso della dashboard, utilizzare delle funzioni ha portato a due vantaggi. Il primo è che le funzioni permettono di evitare le ridondanze nel codice, semplificando il debugging dell’applicazione. Infatti, in qualunque linguaggio di programmazione non è buona norma avere del codice duplicato poiché può risultare dispendioso dal punto di vista computazionale e aumenta la difficoltà del mantenimento. In secondo luogo, permettono di suddividere la Shiny app in file differenti, facilitando il controllo del codice. Le funzioni consentono di creare parti indipendenti dell’app, sia lato UI che lato server, ma svolgono bene il loro compito solo se il codice si trova interamente da un lato o dall’altro [17].

Per operare su codice che copre entrambi, è necessario utilizzare i moduli. Un *modulo* è una coppia di funzioni UI e server ed ha una struttura molto simile a quella di una app completa. Lato UI, un modulo è una funzione che genera delle specifiche relative alla user interface, mentre lato server è una funzione che specifica “cosa fare” all’applicazione. In termini più informali, i moduli, intesi come la coppia di funzioni sopra descritte, si possono definire come “pezzi” di UI e server che vanno a comporre l’interezza dei due componenti.
La particolarità dei moduli è che le funzioni che vi appartengono sono sviluppate in modo da creare un *namespace*: ciò che è contenuto al loro interno è indipendente dal resto dell’applicazione e non è visibile dall’esterno.
Siccome i moduli sono delle funzioni, esattamente come quest’ultime permettono il riutilizzo del codice [18], aspetto che ne favorisce l’utilizzo da parte del programmatore.

Moduli e funzioni sono stati impiegati nella realizzazione della dashboard, sfruttando così i vantaggi della loro implementazione. In particolare, le funzioni sono state utilizzate per la creazione di grafici e tabelle mentre i moduli sono stati utilizzati per gestire le *tab* della dashboard. Una "tab" è intesa come una voce del menù della dashboard che corrisponde ad una pagina dell'applicazione: cambiando tab, viene modificato il contenuto della dashboard.
In Fig. 3 viene mostrato il menù della dashboard con le varie tab.

Fig. 3: dettaglio della dashboard MoonUtilities (menù).

Ad ogni tab corrisponde, quindi, una coppia di funzioni UI e server che costituiscono, una volta chiamate, l'interezza della pagina e, nel complesso, la dashboard. Per esempio, il codice della voce "Contratti" del menù in Fig. 3 è costituito dalle due funzioni del modulo contenuto nel file *mod_contratti.R*. Le due funzioni in questione sono `contrattiUI()` e `contrattiServer()`, necessarie appunto per creare la UI e il server della tab (cosa viene visualizzato e cosa viene fatto). Queste due funzioni sono chiamate rispettivamente nei file *ui.R* e *server.R*, così da poter essere utilizzate. Questa costruzione è simile per tutti i moduli della dashboard.

È necessario specificare che le due funzioni devono lavorare in coppia. Per metterle in relazione, è necessario che prendano come argomento un id, il quale deve essere uguale per entrambe ed univoco all'interno dell'app. In assenza di questo, le due funzioni non possono essere connesse tra loro e viene meno il corretto funzionamento del modulo.
Questo meccanismo ha permesso di snellire notevolmente i file ui.R e server.R che altrimenti avrebbero avuto una dimensione troppo elevata, rendendo molto difficoltose le eventuali modifiche. Inoltre, essendo ogni modulo un'entità indipendente, è possibile isolare il comportamento di ogni singolo componente al fine di comprenderne al meglio il funzionamento e correggere eventuali errori in modo rapido e scorrevole.

Oltre a motivi di semplicità implementativa, la suddivisione in tab e moduli è motivata da ragioni logiche: ogni pagina contiene delle informazioni differenti e la sua realizzazione è contenuta in unico file indipendente dagli altri, rispecchiando i diversi contenuti.

Il contenuto principale delle varie tab può essere sintetizzato come di seguito:

- Home: contiene delle viste che forniscono la sintesi dei dati maggiormente significativi, come quanti contratti sono stati stipulati durante l'anno e il mese corrente e quanti recessi sono avvenuti negli stessi periodi;
- Productivity è una sezione composta da quattro elementi.
  - Contratti: qui sono presenti grafici che forniscono ulteriori informazioni riguardo a contratti e recessi, entrando maggiormente nel dettaglio delle loro caratteristiche, quali mandanti, tipologia (luce/gas) e data di stipulazione o avvenuto recesso;
  - Agenzie: racchiude dati riguardanti corner attivi, agenzie e segnalatori;
  - Dettagli: riporta delle tabelle contenenti le righe del database nel dettaglio;
  - Rete: la pagina racchiude un albero interattivo che rappresenta la gerarchia (se esiste) a cui appartiene l'utente che accede alla dashboard stessa;
- Performance: questa pagina contiene indici di performance relativi alla rete.

In Fig. 4 è riportata tutta la struttura del codice, evidenziando i moduli e le dipendenze tra i vari componenti. L'estensione ".R" indica che si tratta di un file del progetto in linguaggio R.

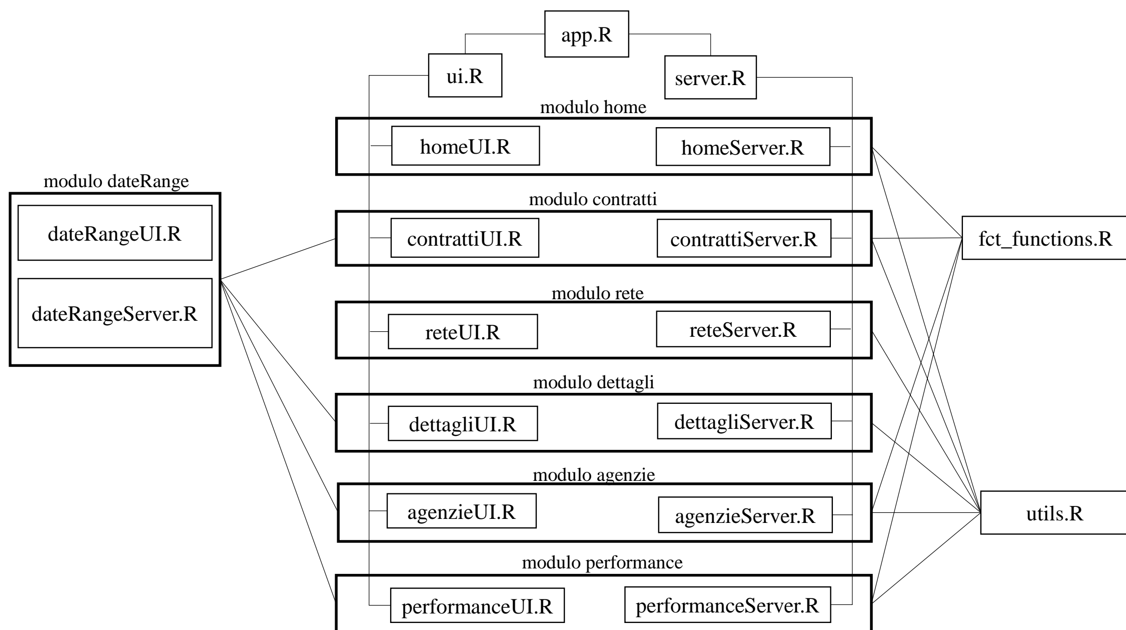

Fig. 4: struttura di tutto il codice della dashboard MoonUtilities.

La figura rappresenta quanto illustrato in precedenza. Inoltre, è possibile notare che il modulo `mod_dateRange.R()` non viene inserito direttamente in ui.R o server.R come accade per gli altri moduli, bensì viene incapsulato in essi. Questo accade perché è stata sfruttata la proprietà di riutilizzo del codice. I moduli relativi alle sezioni "Contratti", "Dettagli", "Agenzie" e "Performance" hanno in comune la possibilità di filtrare i dati, e di conseguenza le informazioni rappresentate nei grafici, secondo un lasso temporale variabile. Le funzioni del modulo in questione creano una coppia di input di testo che, se premuti, fanno apparire i calendari tramite i quali l'utente può selezionare le date [19] e ricevono il valore della selezione dell'utente. Successivamente, questo valore è passato al modulo in cui sono contenute le funzioni. In questo modo, non vi è codice ridondante e la UI relativa all'oggetto di input appare omogenea in tutte le pagine, così come il funzionamento della selezione.
Dalla Fig. 4 è anche possibile notare che due file, *fct_functions.R* e *utils.R*, non appartengono a nessun modulo. Questo accade semplicemente perché non sono moduli, ma file singoli che contengono funzioni e altro codice di appoggio utilizzato da quasi tutti gli altri file.

La suddivisione del codice presentata in precedenza è suggerita nel libro *"Engineering Production-Grade Shiny Apps"* (Colin Fay, Sébastien Rochette, Vincent Guyader, Cervan Girard, 2022) [20]. Secondo gli autori, un'applicazione Shiny è caratterizzata da due aspetti che dovrebbero rimanere separati, al fine di mantenere bassa la complessità del codice:

- *application logic*: consiste nei componenti che rendono la Shiny app interattiva;
- *business logic*: comprende le parti, come algoritmi e funzioni, che rendono l'applicazione specifica per l'ambito in cui opera. Si distinguono dalla application logic per il fatto che queste possono esistere anche al di fuori di un ambiente interattivo.

Un altro fattore che potrebbe favorire la linearità del codice è l'utilizzo di convenzioni per la denominazione dei file che compongono il progetto. Per esempio, i file che contengono i moduli hanno prefisso "mod_", le funzioni al loro interno terminano con "UI" o "Server" in base al ruolo che svolgono ed il file "utils.R" contiene delle funzioni di supporto.
Suddividere il codice e utilizzare delle convenzioni per scegliere come chiamare i file, favorisce la manutenzione e permette anche ad altre persone di comprendere la struttura

del codice, secondo la massima "*Small is beautiful*": maggiore la frammentazione del codice, maggiori saranno i vantaggi. Logicamente questo principio è da applicare con criterio.

Nonostante queste convenzioni siano state applicate nello sviluppo, non è stato possibile seguirle in ogni occasione. Ad esempio, le *best practice* prevedono che l'importazione dei dati (aspetto solitamente non interattivo) sia separata dal resto del codice. Ciò risulterebbe maggiormente coerente con quanto esposto nei paragrafi precedenti, ma non è stato possibile adottare la convenzione perché, per come è stata progettata la dashboard, si è previsto che la chiamata stessa sia dinamica, dato il filtro su utente e ruolo. Oltre a ciò, l'import dei dati deve essere una delle prime attività svolte dalla dashboard, altrimenti le successive funzioni in fase di costruzione restituiscono degli errori, non trovando nell'ambiente i valori su cui devono operare. Per questo motivo, l'importazione dei dati è inclusa nel file server.R, le cui istruzioni sono eseguite subito dopo aver costruito la UI, e non in un file a sé stante.

3.4 Gestione dell'interattività

Una delle caratteristiche essenziali della dashboard è la possibilità di filtrare i dati in base ad un determinato lasso di tempo specificato dall'utente. Questa funzionalità è stata integrata considerando che per il management risulta fondamentale avere la capacità di confrontare tra loro precisi periodi temporali, osservando l'andamento delle attività dell'azienda.

L'interattività con l'utente non riguarda solo la selezione delle date perché all'interno della dashboard sono presenti diversi controlli usati dall'utente per scegliere i filtri da applicare ai dati.

Shiny permette di ottenere un prodotto interattivo sfruttando la programmazione reattiva, ovvero un paradigma di programmazione che permette di creare e gestire un grafo delle dipendenze sussistenti tra i vari elementi dell'applicazione. Queste relazioni consentono agli elementi che vi appartengono di aggiornarsi automaticamente ogni volta che un input da cui dipendono viene modificato [21].

Fig. 5: esempio di rappresentazione di un grafo delle dipendenze.

Queste logiche sono definite nel server di una Shiny app ed è per questo motivo che la funzione ha come argomento i tre parametri input, output e session: gli output sono collegati agli input in una determinata sessione e dipendono da questi, dove per "dipendenza" si intende quanto affermato sopra, ovvero se un input viene modificato, anche gli output ad esso collegato subiscono la variazione.
Questo meccanismo permette agli output di essere sempre in linea con gli input e consente al programmatore di non doversi occupare della gestione del flusso di aggiornamento.

Nel linguaggio R, questo proposito è raggiunto attraverso delle *reactive expression*, cioè una espressione legata ad un oggetto di input e che ne utilizza il valore [22]. Concettualmente, sono strettamente collegate alla programmazione basata su eventi, la quale ha come fondamento l'idea di eseguire delle istruzioni in risposta al verificarsi di determinati eventi esterni.

Queste espressioni presentano due aspetti particolarmente vantaggiosi. Il primo riguarda la *laziness*, vale a dire che una reactive expression non viene eseguita fino a quando non è espressamente chiamata. Nel caso della dashboard, questo si verifica in diverse occasioni, ad esempio, ogni volta che l'utente cambia tab: fino a quando questo evento non si verifica, gli elementi della pagina non saranno calcolati. Il secondo vantaggio è che il valore di una espressione di questo genere, dopo essere stato calcolato, viene salvato in cache [23]. È quindi sufficiente eseguire le istruzioni una sola volta per ottenere un risultato che potrà essere utilizzato ripetutamente, fino a quando un evento non farà cambiare il suo valore. Anche in questo caso, comunque, l'entità sarà salvata nella cache e potrà essere utilizzata nelle modalità appena descritte.

Per quanto riguarda la dashboard *MoonUtilities*, per la creazione delle reactive expression, è stata largamente utilizzata la funzione `reactive()`. Solitamente, queste contengono i risultati delle query eseguite sui dati. Questa speciale tipologia di espressione si è resa necessaria poiché, come espresso in precedenza, l'utente ha la possibilità di scegliere in base a quali parametri filtrare le informazioni. Queste decisioni sono rese possibili grazie a degli oggetti di input HTML, quali slider, bottoni e check box. Tuttavia, è possibile accedere al valore selezionato solamente all'interno di un ambiente reattivo, altrimenti si verifica un errore. Per questo motivo tutte le interrogazioni: sono racchiuse in un blocco reactive, dipendono dagli input selezionati e modificano autonomamente il proprio contenuto in base alle loro dipendenze.

Un'ulteriore peculiarità di questa tipologia di oggetto riguarda la modalità con in cui vi si accede. Infatti, per utilizzare il contenuto di una reactive expression è necessario chiamare l'espressione come se si chiamasse una funzione, ovvero indicando il nome dell'espressione seguito da delle parentesi tonde. Sotto questo punto di vista, una reactive expression può essere interpretata come una funzione che non prende nessun parametro come argomento.

A questo punto della discussione, è necessario ricordare che un output che dipende da un determinato input cambia *ogni volta* che quest'ultimo subisce una modifica. Nonostante questo aspetto risulti molto comodo, durante lo sviluppo della dashboard ha portato a delle problematiche legate appunto all'aggiornamento continuo degli elementi di output. Un esempio concerne la selezione delle date. Nel dettaglio, l'utente ha la possibilità di selezionare sia la data di inizio che quella di fine relative al periodo di tempo che desidera prendere in considerazione. Se l'intenzione era di modificarle entrambe, allora tutto il contenuto della pagina veniva aggiornato due volte, sia alla selezione della prima che della seconda data, causando una latenza nella visualizzazione dei grafici. Inoltre, il duplice calcolo risulta inefficiente sotto il profilo computazionale.
Questa circostanza si verificava perché, all'interno del grafo, gli oggetti filtrati avevano una dipendenza sulla data iniziale del periodo e una su quella finale: nel momento in cui una delle due era modificata, l'elemento dipendente si aggiornava di conseguenza.

Per evitare situazioni di questo genere, {shiny} mette a disposizione degli strumenti per controllare il flusso che attraversa il grafo delle dipendenze. Questi permettono anche di specificare *quando* eseguire determinate operazioni. Le funzioni sono state utilizzate frequentemente; in diverse occasioni si è reso necessario gestire il momento in cui creare i vari elementi della UI, controllando che i grafici avessero a disposizione tutti i dati prima di procedere alla loro visualizzazione. In assenza di questi accertamenti, si generano degli errori e la dashboard non viene caricata.

Tra i mezzi fruibili, nell'implementazione della dashboard sono stati utilizzati principalmente i metodi `eventReactive()` e `observeEvent()`, i quali permettono di isolare gli elementi al loro interno e di coordinare la creazione delle dipendenze [24].
I due metodi `eventReactive()` e `observeEvent()` hanno in comune lo scopo di gestire gli eventi e, per questo motivo, presentano una sintassi simile. I due parametri essenziali per il loro funzionamento sono l'evento a cui reagire e le operazioni da eseguire in risposta a questo trigger. Nella pratica, sono stati applicati i due metodi in modo tale da eseguire

le operazioni solo a seguito di un click da parte dell'utente su un bottone inserito nella UI.

Nonostante possa sembrare che le due funzioni compiano la stessa operazione, queste sono da utilizzare in due occasioni differenti. Come specificato dalla documentazione di {shiny} [25], `observeEvent()` è da utilizzare quando si presenta la necessità di *eseguire un'azione* in reazione ad un evento, mentre ci si avvale di `eventReactive()` quando si deve *calcolare un valore*, sempre in risposta ad una azione esterna.
Per fare un esempio di come sono stati adottati questi principi, si consideri la tab "Agenzie". Questa contiene una panoramica sulle agenzie in un determinato lasso temporale selezionato dall'utente. L'implementazione presentava le due problematiche sopra citate, vale a dire gli errori dovuti alla prematura creazione dei grafici e il loro duplice caricamento al selezionare delle date.
La soluzione è stata quella di inserire un bottone "Carica" per innescare:
- `eventReactive()`: reagisce al click del tasto e salva in una variabile di tipo reactive le date selezionate;
- `observeEvent()`: viene innescato dallo stesso elemento e fa visualizzare tutti i grafici della pagina.

Una ulteriore funzione utilizzata per la gestione di valori reactive è `observe()`, la quale genera un observer[3] per l'espressione passatagli come argomento. Un observer in linguaggio R è simile ad una reactive expression: può leggere valori reactive e chiamare espressioni reactive, aggiornandosi automaticamente quando si verifica un cambiamento delle proprie dipendenze. Tuttavia, si differenzia dalle reactive expression per il fatto che non fornisce un risultato e non può essere usato come input per altre espressioni. In più, si differenziano per la loro strategia di esecuzione. Mentre le reactive expression hanno una *lazy evaluation*, ovvero non vengono rieseguite appena cambiano le loro dipendenze ma attendono fino al momento in cui non sono invocate da qualche altro oggetto, gli observer hanno una *eager evaluation* (valutazione impaziente), cioè vogliono essere eseguiti nuovamente appena le loro dipendenze cambiano [26].

Questa implementazione permette inoltre di non doversi preoccupare di gestire singolarmente tutte le query necessarie ai grafici, poiché queste, dipendendo dagli input,

[3] Un observer si può descrivere come una dipendenza su un oggetto, la quale reagisce automaticamente ai cambi di stato di quest'ultimo. Questo concetto è comune a diversi linguaggi di programmazione ed è chiamato "observe pattern" [27].

restano in attesa di una loro modifica, la quale avviene appunto solo al premere di "Carica" o di qualche altro evento esterno. Il flusso dei dati, quindi, non genera errori e risulta coerente con quanto si desiderava implementare.

4. Dati

I dati sono le fondamenta del progetto. L'intera dashboard si struttura intorno alla loro analisi, passando attraverso i vari passaggi di questo processo. In questo capitolo, è descritta l'evoluzione dei dati, illustrando le fasi di raccolta, importazione, pulizia e trasformazione e, infine, visualizzazione.

4.1 Raccolta

In questo contesto, per raccolta s'intende l'insieme dei processi volti all'acquisizione dei dati e alla loro memorizzazione. Questa fase non sarà descritta nel dettaglio, poiché è precedente allo sviluppo della dashboard e quindi non di competenza dell'elaborato. Risulta comunque utile affrontare brevemente l'argomento per comprendere meglio i passaggi successivi.

Moon Energy S.r.l. possiede un database MySql sul cloud contenente tutto lo storico necessario al corretto svolgimento della sua attività. Ad esempio, sono presenti diverse tabelle relative alle informazioni anagrafiche dei collaboratori della rete, ai contratti e ai recessi, alle provvigioni ma anche tabelle riguardanti gli identificativi ed informazioni ulteriori relative agli utenti.
Le aree di interesse nel contesto dello sviluppo della dashboard sono contratti, recessi ed agenzie.
I dati riguardanti questi tre settori possono essere inseriti nel DB in diversi modi.
Il primo è l'inserimento diretto sulla base dati da parte del back office. In questa situazione, vengono raccolte tutte le informazioni e successivamente viene creata una tabella o vengono integrate quelle già presenti. È questo il caso dell'inserimento delle relazioni di gerarchia per i point che erano già censiti all'interno del sistema. Per questi soggetti non è necessario aggiungere una nuova riga alla tabella delle agenzie, ma solo aggiungere le colonne di interesse e riempirle con le indicazioni trasmesse dall'esterno (ad esempio, un file Excel inviato all'azienda da una holding).
Per quanto riguarda l'inserimento dei contratti, questi non sono inseriti direttamente nel DB ma passano attraverso l'area privata del sito di Moon Energy S.r.l.. Il secondo metodo di inserimento riguarda quindi l'inserimento dei contratti attraverso una maschera apposita, scritta in linguaggio PHP. L'inserimento di un contratto su questa maschera comporta la creazione di una riga all'interno della tabella dei contratti sulla base di dati. Un processo simile avviene per i recessi.

Per contratti e recessi, i dati sono aggiornati anche in base alle informazioni trasmesse dai mandati. Infatti, una volta che il contratto è stato inserito nel sistema, è poi compito del distributore attivare effettivamente le utenze. Il processo di attivazione comporta l'evoluzione dello stato in cui si trova il contratto, il quale deve sempre essere in linea con la realtà affinché il cliente possa essere informato sulle tempistiche, se necessario.

4.2 Importazione

Per eseguire una qualsiasi analisi sui dati, è necessario *avere* dei dati da analizzare. Questo significa che i dati necessitano di essere portati all'interno dello spazio di lavoro.

Nonostante il linguaggio R fornisca diversi pacchetti e funzioni per accedere a database non in locale, si è scelto di non implementare questa soluzione: sarebbe risultata poco pratica e avrebbe lasciato spazio a possibili modifiche involontarie alla base dati durante l'analisi, dovuti principalmente alla esigua conoscenza iniziale del database.
Dunque, è risultato più agevole importare i dati in formato JSON. Si tratta di un formato per lo scambio di dati ed ha la caratteristica di essere *leggibile all'uomo*, ovvero ha una sintassi coppia-valore la cui semantica è interpretabile correttamente dall'uomo in linguaggio naturale. Grazie alla sua sintassi, per le macchine è invece facilmente generabile e analizzabile [28].

In particolare, questa implementazione permette di eseguire le query per recuperare i dati direttamente sul database, evitando di dover importare nel codice diverse tabelle e successivamente eseguire le query. Più specificatamente, si elude l'esigenza di eseguire alcune delle operazioni di join dopo l'importazione, dato che si tratta di operazioni onerose ed è conveniente eseguirle prima di importare i dati. In questo modo, si velocizzano i tempi di calcolo e caricamento della pagina e si risparmia potenza computazionale, ricevendo un unico file JSON contenente la tabella con tutte le informazioni utili.

I dati da analizzare sono raccolti tramite una procedura scritta in linguaggio PHP. Il codice consente di collegarsi alle tabelle del database desiderate ed eseguire su di esse le query SQL. Per l'importazione dei dati è stato utilizzato il pacchetto {jsonlite}, in particolare la funzione `fromJSON()` [29].

Questa prende in input il link per la chiamata al database sopra citato e restituisce un oggetto R. L'output derivante dall'esecuzione di queste istruzioni è una struttura nel formato JSON standard.

tipo_cliente	character [1529]	'B' 'D' 'D' 'D' 'D' …
luce_gas	character [1529]	'L' 'L' 'G' 'L' 'G' …
Stato	integer [1529]	0 0 0 0 0 0 …
descrizione_stato	character [1529]	'Attivo' 'Attivo' …

Fig. 6: esempio di struttura del JSON vista da RStudio.

Di conseguenza, i dati restituiti e importati attraverso questo meccanismo si presentano all'interno del progetto come una struttura già nota in partenza, ovvero compaiono come un oggetto contenente una raccolta di diversi array caratterizzati da un nome e dal tipo di dato che contengono.

Per ottenere tutte le informazioni necessarie per la direzione aziendale, sono compiute solamente tre chiamate, che portano a tre JSON differenti, ovvero "contratti", "recessi" e "agenzie". Questi contengono la totalità dei dati occorrenti per l'analisi dell'attività e la creazione dei grafici.
L'implementazione di questo processo fornisce la possibilità ad un collega più esperto di gestire le chiamate alla base di dati, frazionando il flusso di lavoro necessario per lo sviluppo della dashboard.

È interessante notare che questi tre file non sono di grandi dimensioni. Ad esempio, il file contenente le informazioni dei contratti è composto da circa 1500 righe e 25 colonne. Nonostante non sia disponibile una grande quantità di dati, è comunque possibile estrarre una notevole quantità di conoscenza in grado di portare consapevolezza e competenza all'interno dell'azienda.

Come accennato in precedenza, la dashboard è incapsulata in un iframe nell'area privata del sito dell'azienda, strategia che è stata implementata per fare in modo che l'utente che accede alla dashboard veda solo i dati di propria pertinenza.
Ogni utente ha collegate al proprio account delle informazioni, tra cui l'identificativo utente, univoco per ogni soggetto, e l'identificativo del ruolo, il quale indica la carica che il soggetto ricopre all'interno della gerarchia.

Quando si accede all'area privata del sito di Moon, *id_ruolo* e *id_utente* sono recuperati e concatenati al link della dashboard utilizzato dall'iframe. Per esempio, se id_utente è "12345" e id ruolo è "56789", allora il link risultante può essere pensato come

https://shinyapps.io/... /?id_ruolo=56789&id_utente=12345[4]

È possibile eseguire questa operazione poiché il server in Shiny include un parametro *session*. Si tratta di un oggetto di tipo *enviroment* che può essere usato per accedere alle informazioni e funzionalità attinenti alla sessione [30]. Tra i vari elementi disponibili in questo oggetto, vi sono anche le informazioni relative al client, tra cui i componenti dell'URL della pagina. Questi parametri sono quindi recuperati con l'operazione `session$clientData$url_search`.

Per avere accesso all'oggetto session, essendo un ambiente, è necessario inserire le istruzioni per recuperare i parametri all'interno di `observer()`, metodo descritto nel capito precedente.

Queste due variabili sono concatenate al link dello script PHP che recupera i dati dal database e li restituisce al programma in formato JSON.

Il link per la chiamata al database è, pertanto, nella forma

https://database/... /?id_ruolo=56789&id_utente=12345

Lo script PHP recupera gli id ed esegue la query al DB filtrando i dati in relazione a queste informazioni: sono restituiti solo i dati relativi a contratti, recessi e rete dell'utente 12345 con ruolo 56789.

Una volta che la procedura PHP è terminata, il JSON derivante dall'esecuzione della query è costruito e viene recuperato dal metodo contenuto nella dashboard. A questo punto è possibile usare i dati per eseguire tutte le operazioni previste, volte ad analizzare i dati e visualizzarne i risultati.

L'utente ha quindi accesso alla dashboard tramite l'iframe sulla pagina menzionata precedentemente.

[4] Per motivi di privacy e sicurezza, non è possibile rendere noto il link. L'obiettivo è solo quello di illustrarne il funzionamento.

In Fig. 7 è illustrato lo schema degli scambi id descritto sopra.

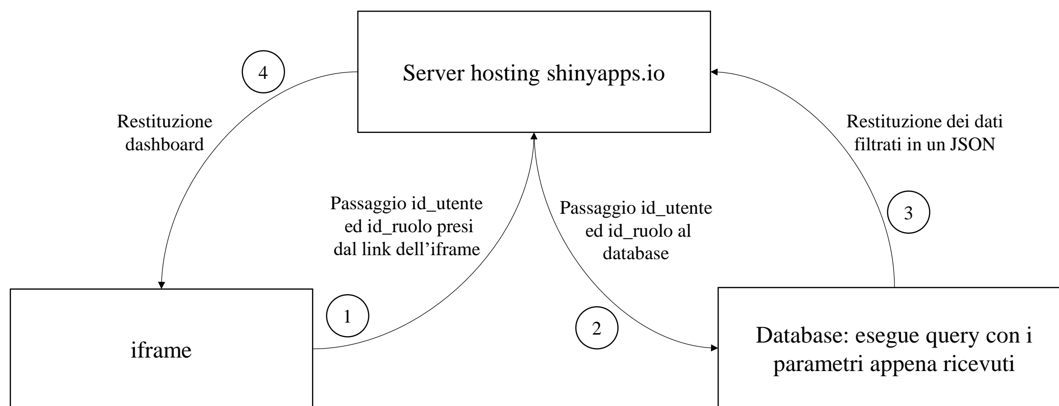

Fig. 7: schema illustrativo del passaggio dei parametri.

Tuttavia, i dati ricevuti attraverso questa procedura non sono facilmente sfruttabili nello stato attuale, poiché hanno una forma inadatta per poter essere utilizzati dalle funzioni che svolgono le analisi. Per questo motivo necessitano di una ulteriore modellazione.

Il linguaggio R fornisce degli strumenti per gestire ed elaborare dati in formato JSON. In questo caso, si è scelto di trasformare quanto è restituito in una struttura chiamata *tibble*, attraverso la funzione `as_tibble()` del pacchetto {tidyr}[31]. Si tratta di una rielaborazione del classico data frame[5], distinguendosi da quest'ultimo per essere "*lazy and surly*" (pigro e scontroso): questo implica che i dati in formato tibble ottimizzano le operazioni eseguite su di essi ma si "lamentano" di più (restituiscono un numero maggiore di errori e warning), costringendo il programmatore ad affrontare fin dal principio i problemi con la struttura e portando spesso a soluzioni più pulite e significative [33]. Concettualmente, una variabile di tipo tibble è molto simile ad una tabella, ovvero appare come un insieme ordinato di righe e colonne. Tramite questa trasformazione, si ottengono dei dati completamente strutturati ed organizzati, raggiungendo delle architetture in una forma utile per essere esplorate.

Il package {tidyr} fa parte di un insieme di pacchetti utili all'analisi dati, chiamato {tidyverse} [34]. Questo pacchetto è composto da ulteriori pacchetti di dimensioni minori utili per compiere i processi di importazione, pulizia, trasformazione e visualizzazione sui dati. In questo caso sono stati utilizzati i pacchetti {jsonlite} e {tidyr} per

[5] Un data frame è una struttura dati composta da una collezione di variabili *tightly coupled* e possiede delle proprietà simili alle liste ed alle matrici. I data frame sono la struttura alla base della maggior parte della modellazione in linguaggio R [32].

l'importazione, {tidyr} e {dplyr} per la pulizia dei dati, {dplyr} per la trasformazione e {plotly} per la visualizzazione.

4.3 Pulizia e trasformazione

Per pulizia dei dati si intende la memorizzazione dei dati in uno stato consistente, in modo tale che ci sia coerenza tra la semantica del dataset e il modo in cui sono organizzati. Sostanzialmente, se i dati sono puliti "ogni colonna è una variabile e ogni riga è un'osservazione" (tradotto da *R for Data Science*, Hadley Wickham, Garrett Grolemund, 2016) [35]. A questa affermazione, è possibile aggiungere che "ogni cella è un singolo valore" (tradotto da *tidy data*, tidyr.tidyverse.org, Hadley Wickham, Maximilian Girlich) [36]. Insieme, questi due aspetti definiscono i *tidy data*.
Lavorare su dati ordinati permette di avere una struttura sempre consistente e di potersi focalizzare sulle interrogazioni e sulla visualizzazione dei dati.

Per quanto riguarda il processo di trasformazione, questo può essere descritto come insieme di tre operazioni fondamentali [37]:

- restringere i dati alle osservazioni a cui si è interessati;
- generare nuove variabili in funzione di quelle che già si hanno a disposizione;
- calcolare un insieme di statistiche riassuntive ed esplicative dei dati che si vuole esplorare.

Comunemente, si usa riferirsi alle fasi di pulizia e trasformazione come "*wrangling*" (disputa, litigio), poiché si tratta di operazioni che richiedono un lavoro dispendioso da parte del programmatore che spesso si trova in difficoltà di fronte alle complessità presentate dai dati.

Le tre strutture restituite dalle chiamate della fase di importazione sono salvate in altrettante strutture di tipo tibble e sono rese globali, così che tutti i moduli possano vederle ed utilizzarle.
Senza questo accorgimento, i dati, essendo importati all'interno di un elemento `observe()` del file server.R, non riuscirebbero ad "uscire", essendo ogni espressione di questo genere relativa solamente al contesto in cui è contenuta.

Una volta salvati nelle variabili, inizia la fase di pulizia. Questa consiste principalmente nel cambiare il tipo di dato contenuto nelle varie colonne e rimuovere i valori *NA* (not available). Sistemare questi aspetti in una fase preliminare permette di evitare possibili

errori durante lo sviluppo e allo stesso tempo di eseguire queste operazioni una sola volta, senza ripeterle prima di eseguire ciascuna query.

Prima di compiere le operazioni iniziali, viene eseguito un controllo sulla struttura su cui si vuole operare: se questa non contiene righe, allora le istruzioni non sono attuate. Controlli di questo genere sono comuni all'interno del codice poiché è altrettanto comune che la chiamata al database oppure l'applicazione di un filtro restituiscano una tabella con zero righe. Ciò accade per la tabella "contratti", per esempio, quando si applica un filtro per le date e si sceglie come intervallo temporale una domenica oppure un periodo di festività. In queste occasioni, è molto probabile che non siano stati stipulati contratti e di conseguenza non ci siano dati per il periodo considerato, ottenendo una struttura vuota.

Le operazioni di trasformazione sono invece relative all'effettiva esplorazione dei dati ed estrapolazione delle informazioni richieste. Si tratta della fase che ha occupato la quantità di tempo maggiore, siccome ha richiesto particolare minuziosità. Infatti, dopo aver compreso le esigenze espresse dal management, è stato necessario capire *quali* interrogazioni eseguire sui dati, al fine di ottenere informazioni che risultassero corrette e rappresentative della realtà, e *come* eseguirle, considerando di evitare ripetizioni ed errori.

La quasi totalità delle operazioni sopra citate sono state svolte con l'ausilio delle funzioni appartenenti al pacchetto {dplyr}. Si tratta di una grammatica che fornisce un insieme di verbi per la manipolazione di dati contenuti in dei data frame [38], dove per *verbi* s'intende un insieme di funzioni che corrispondono alle operazioni di manipolazioni più comuni ed utilizzate.
Questo pacchetto viene utilizzato all'interno del codice della dashboard per eseguire le query sulle tabelle in locale, ma può anche essere utilizzato per compiere interrogazioni su basi di dati in remoto, dato che possiede un back-end generalizzato per comunicazioni di questo genere [39].
Usufruire della sintassi del pacchetto permette di scrivere le query mantenendo un linguaggio comune in tutto il codice del progetto, ovvero il linguaggio R, evitando di inserire istruzioni in un altro linguaggio per la loro realizzazione.

Oltre a ciò, considerato che le istruzioni del package {dplyr} sono tradotte automaticamente in SQL, non è necessario conoscere a fondo tale linguaggio [40].

Ad esempio, ipotizzando di avere una tabella denominata “contratti” con diverse colonne, tra cui `id_agenzia` e `agenzia`, le seguenti due query svolgono la stessa interrogazione, restituendo il medesimo risultato.

```
SELECT id_agenzia, agenzia FROM contratti;
```

```
contratti %>%
    select(id_agenzia, agenzia)
```

Snippet 3: esempio di codice SQL (sopra) e in linguaggio R, pacchetto {dplyr} (sotto).

Nello Snippet 3 è riportato il codice delle operazioni eseguite sulla tabella “contratti”, le quali restituiscono le colonne selezionate. Come si può notare, entrambi i linguaggi, seppur differendo in apparenza, condividono la stessa istruzione: “select”. Questa somiglianza enfatizza il fatto che tutte le query possibili in SQL sono riproducibili utilizzando i verbi del pacchetto {dplyr}.

Sempre dallo Snippet, è possibile notare i simboli `%>%` (*pipe*). Si tratta di un operatore del pacchetto {magrittr}, molto utilizzato in coppia con le funzioni di {dplyr}, con l’obiettivo di diminuire il tempo impiegato nella scrittura del codice e renderlo maggiormente leggibile e mantenibile.
Tutti i verbi del pacchetto preso in considerazione hanno, per definizione, come primo argomento l’elemento su cui devono effettuare l’operazione. L’operatore *pipe* permette di passare il risultato di una istruzione alle successive, in modo da non riscrivere operazioni molteplici volte. L’esempio analizzato, se non utilizzasse questo metodo, sarebbe scritto nella forma equivalente:

```
select(contratti, id_agenzia, agenzia)
```

In questa dimostrazione, trattandosi di un caso molto semplice, non vengono evidenziate particolarmente le motivazioni che possono spingere a sfruttare i vantaggi che offre l’operatore `%>%.` Ciò nonostante, nelle situazioni più complicate, si è dimostrata un’ottima opportunità per semplificare il codice e la comprensione delle query, come verrà mostrato in seguito.

In precedenza, è stato affermato che la fase di pulizia modifica alcuni aspetti dei dati contenuti nelle celle delle tabelle. Una parte di codice applicato a questa fase è riportato nello Snippet 4. L’obiettivo dello Snippet è quello di modificare la colonna `dtvar_tipo` della tabella “agenzie”, ammesso che il tibble non sia vuoto.

```
if(nrow(agenzie) != 0){
          agenzie <<- mutate(agenzie,
                         dtvar_tipo = case_when(
                         is.na(dtvar_tipo) ~ as.Date('2022-01-01'),
                         !is.na(dtvar_tipo) ~ as.Date(dtvar_tipo)))}
```

Snippet 4: esempio di codice per la pulizia dei dati.

Come è possibile osservare, la prima istruzione svolge il controllo sulla dimensione della tabella, verificando che il numero totale delle righe sia diverso da zero. Le operazioni successive compiono le modifiche desiderate e salvano i cambiamenti nella tabella originale, tramite l'operazione di assegnamento globale "`<<-`". In particolare, l'obiettivo di questo blocco di codice è modificare il tipo di dato contenuto nella colonna `dtvar_tipo`, trasformandolo, grazie all'utilizzo della funzione `as.Date()`, in un oggetto di classe *Date*, il quale rappresenta i valori come date di un calendario, fornendo la possibilità di applicare tutte le proprietà di una simile entità. Ad esempio, è possibile sfruttare delle funzioni per spostarsi da una data all'altra oppure ottenere l'ultimo giorno del mese della data presa in considerazione. In questo modo, si rendono automatiche alcune operazioni, senza la necessità di dover gestire ogni singolo caso o eccezione. In assenza di questa modifica non è possibile eseguire molte delle query richieste. Infatti, una volta importato il tibble "agenzie", all'interno delle celle di questa specifica colonna sono presenti stringhe di caratteri, tipologia non adatta a compiere le operazioni sopra citate.
L'istruzione `mutate(…)` esegue le modifiche sulle celle della colonna, sfruttando il metodo `case_when(…)`: se il contenuto della cella della colonna `dtvar_tipo` è NA, allora sostituisce il valore con una data prefissata (decisa dal management e coerente con il contesto), altrimenti trafroma la stringa in un oggetto Date. Questo metodo è equivalente all'istruzione CASE WHEN presente in SQL, dove ~ (tilde) corrisponde a THEN.

Un simile procedimento, con tutti gli accorgimenti del caso, è applicato ad ogni tabella subito dopo la sua importazione. Una volta che le procedure hanno terminato la propria esecuzione, si ottengono delle tabelle complete, senza celle vuote che possano generare errori successivamente e contenenti dei valori convertiti in un tipo di dato che risulta logicamente coeso con l'informazione che devono rappresentare. Si raggiunge così il proposito fondamentale della fase di pulizia, ovvero preparare il più possibile i dati prima di iniziare il processo di trasformazione.

È di interesse osservare che la fase di pulizia non è svolta esclusivamente come fase preliminare: questi processi sono ripresi e aggiornati ogni qual volta che si presenta una

nuova problematica che riguarda la struttura dei dati e che può essere risolta ritornando su queste istruzioni. Casi di questo genere hanno riguardato l'inserimento di una nuova colonna all'interno della tabella oppure la necessità di modificare alcuni aspetti della tabella per eseguire dei calcoli particolari nella fase successiva.

Il processo di trasformazione ha lo scopo di preparare tutte le informazioni per la fase di visualizzazione. Anche in questo caso si è fatto uso dei verbi del pacchetto {dplyr} e ciascuna query è inserita in oggetto reactive individuale per i motivi illustrati nel capitolo 3.4.

Alcuni dei contenuti richiesti dal management sono stati molto semplici da reperire. Spesso si trattava solamente di somme di una particolare entità oppure di filtrare i dati in base ad un particolare valore, considerando un determinato periodo temporale. In questa sezione sono mostrati alcuni esempi di interrogazioni, con lo scopo di illustrare la complessità che è stata affrontata e talune delle informazioni contenute all'interno della dashboard.

```
#totale contratti mese attuale
 totContrattiMese <- reactive({
      contratti %>%
      filter(data_inserimento >= dataInizio6 &
               data_inserimento <= Sys.Date()) %>%
      summarise(tot = n())
   })

#totale segnalatori attivi
segnalatoriAttivi <- reactive({
      contratti %>%
           filter(tipo_agenzia == 'S') %>%
         distinct(id_agenzia) %>%
         summarise(tot = n())
})
```

Snippet 5: esempio di due query in linguaggio R utilizzando il pacchetto {dplyr}.

Il primo blocco dello Snippet 5 mostra come si ottiene il totale dei contratti relativo al mese attuale. Qui è possibile notare il verbo `filter(…)`, equivalente a WHERE in SQL, che filtra i dati in base alle date. La funzione `summarise(…)` crea una nuova struttura, che in questo caso possiede una sola riga e una sola colonna, chiamata `tot`, contenente la somma desiderata ottenuta grazie alla funzione `n()`, utile per calcolare le occorrenze. Il secondo blocco di codice vuole estrarre il numero totale dei segnalatori attivi, ovvero quei

[6] `DataInizio` è una variabile che contiene il primo giorno del mese attuale. Si aggiorna automaticamente nel tempo e la sua definizione è contenuta nel file utils.R. Il file contiene altre funzioni necessarie per calcolare valori di supporto di questa natura.

segnalatori che hanno svolto almeno un'attività. Le istruzioni sono simili a quelle dell'esempio precedente, con l'aggiunta del verbo `distinct()`, il quale, similmente a DISTINCT in linguaggio SQL, seleziona le righe della tabella senza considerare le occorrenze multiple.

Talvolta, alcune delle interrogazioni sono risultate più complesse ed hanno occupato una quantità maggiore di tempo per essere realizzate. È questo il caso dello Snippet riportato di seguito.

```
#totale contratti per agenzia in un determinato lasso temporale
         contra_agen <- reactive({
             contr_little()  %>%
             right_join(agenzie,
                          by = c('id_agenzia' = 'id_agenzia',
                         'agenzia' = 'agenzia')) %>%
         mutate(tot = case_when(is.na(stato) ~ 0,
                                  !is.na(stato) ~ 1)) %>%
             group_by(id_agenzia, agenzia, data_inserimento) %>%
             summarise(tot = sum(tot)) %>%
             arrange(desc(tot))
         })

         agenzia_somma <- reactive({
             contra_agen() %>%
             ungroup() %>%
             filter((data_inserimento >= dateScelte()[1] &
                     data_inserimento <= dateScelte()[2]) |
                     is.na(data_inserimento)) %>%
             group_by(id_agenzia, agenzia) %>%
             summarise(tot = sum(tot))
         })
```

Snippet 6: esempio di una query più complessa.

I due blocchi di codice riportati sopra hanno come obiettivo quello di creare una nuova tabella nella quale per ogni agenzia è indicato il numero totale di contratti stipulati in un periodo di tempo selezionabile dall'utente. Lo scopo era di ottenere il risultato equivalente dell'istruzione SQL riportata dallo Snippet 7.

```
 SELECT b.id, sum(CASE
                   WHEN a.id_agenzia is null then 0
                   ELSE 1
              END)
          tot_contratti
FROM contratti a right_join agenize b on b.id = a.id_agenzia and
a.data_inseriemento <= dataScelta and a.data_inseriemento >=
dataScelta
GROUP BY b.id
ORDER BY 'tot_contratti' DESC
```

Snippet. 7: corrispondente codice SQL della query {dplyr} dello Snippet 6.

L'ideale sarebbe stato quello di poter ottenere il risultato desiderato in questo modo, con una sola istruzione. Tuttavia, questo non è stato possibile a causa dell'impossibilità di effettuare l'operazione di `case_when(…)` usando come condizione il fatto che una riga sia nulla. Di conseguenza, l'interrogazione è stata suddivisa in due parti. La prima compie un `rigth_join(…)` per unire la tabella `contr_little()`[7] (riduce il numero di colonne del file "contratti" selezionando solo le colonne di interesse) con la tabella "agenzie". Successivamente, sfrutta la funzione `mutate(…)` per aggiungere alla tabella risultante una nuova colonna contenente il valore uno o zero, a seconda se la colonna `stato` contenga un valore nullo o meno. Questo è l'equivalente dell'operazione CASE WHEN del caso SQL.

Questo metodo funziona tenendo presente che la tabella "contratti" contiene tutti i contratti stipulati, dove ogni riga corrisponde ad un contratto, mentre la tabella "agenzie" contiene tutte le agenzie. Quindi, tutte le agenzie presenti in "contratti" sono anche contenute in "agenzie" ma non vale il contrario, perciò se un'agenzia si trova in "agenzie" ma non in "contratti", significa che non ha mai svolto un'attività di stipulazione di contratto per un cliente. Essendo `stato` una colonna appartenente unicamente a "contratti" e contenendo sempre un valore non nullo per ogni riga, dopo l'operazione di join, se questa risulta vuota, allora significa che l'agenzia non ha mai stipulato un contratto e quindi il totale dei contratti stipulati deve essere zero. La tabella risultante avrà un numero di righe pari alla somma del numero di righe delle tabelle "contratti" e "agenzie". A questo punto, le istruzioni successive raggruppano per agenzia e data, sommando il contenuto della colonna `tot` e ordinano l'intera tabella in ordine decrescente.

Una generica riga della tabella finale ottenuta è del tipo:

id_agenzia	agenzia	data_inserimento	tot
1	Agenzia	2022-01-15	3

La colonna `data_inserimento` è fondamentale poiché successivamente permetterà di applicare il filtro indicato dall'utente, sommando tutte le righe con valore appartenente all'intervallo dato.

[7] Non si tratta di una funzione vera e propria ma di una reactive expression ottenuta tramite `reactive()`. Per questo motivo, l'invocazione viene seguita dalle parentesi tonde.

Si noti che in assenza dell'operatore *pipe* non sarebbe possibile strutturare le query in un unico blocco ma diventa necessario eseguire ogni singola istruzione e salvarla all'interno di una variabile, così da poterla passare come argomento all'operazione successiva, aumentando la complessità e la possibilità di compiere degli errori.

4.4 Visualizzazione

La visualizzazione dei dati è quell'attività svolta con l'obiettivo di comunicare graficamente delle informazioni in modo che la rappresentazione risulti chiara ed efficace. Si tratta di un ambito delicato poiché errori nella raffigurazione o dei grafici ambigui possono portare chi guarda a percepire il contenuto in modo errato e fuorviante. Alla base della data visualization si trova il principio secondo il quale ogni grafico esegue un processo di *mapping*[8] tra la quantità che si vuole rappresentare e un elemento grafico quantificabile che possa rappresentarla. Più formalmente, questo principio è noto come *proportional ink*, ovvero "quando si utilizza un'area di un grafico per rappresentare un valore numerico, l'area di tale regione deve essere direttamente proporzionale al valore corrispondente" (tradotto da *The Principle of Proportional Ink*, Bergstrom, West, 2016) [42]. Se un grafico non possiede questa caratteristica, allora è inconsistente. L'insieme delle quantità e delle rispettive rappresentazioni, in uno stesso contesto, compone un grafico.

All'interno della dashboard sono presenti diverse tipologie di grafici, a seconda dei dati che occorreva rappresentare e dell'informazione che si desiderava trasmettere. In particolare, sono stati largamente utilizzati *pie plots*, *box*, *bar plots* e *side-by-side bar plots*, *line plots* e, in più, *tabelle*.
Per la realizzazione di questo aspetto, sono state seguite alcune delle linee guida contenute nel libro *Fundamentals of Data Visualization* (Claus O. Wilke) pubblicato per O'Reilly Media, Inc (2019) [43]. Questo scritto vuole porsi come guida pratica per realizzare grafici attraverso i mezzi offerti dal linguaggio R. I principi proposti sono stati adattati alle diverse situazioni che si sono presentate durante lo sviluppo, riservandosi di applicarli solo qualora portassero degli effettivi miglioramenti oppure fossero in linea con lo specifico caso.

[8] Termine che indica un meccanismo per mettere in relazione due oggetti allo scopo di riportare mutamenti, richieste ed elaborazioni da un oggetto ad un altro. [41]

Per la realizzazione di tutti i grafici è stata utilizzata la libreria {plotly}, un pacchetto per costruire grafici interattivi e web-based, sfruttando la libreria JavaScript open source *plotly.js* [44]. È stato scelto questo pacchetto perché, tra i numerosi package offerti dal linguaggio R per la creazione di grafici, {plotly} è l'unico che permette l'interazione dell'utente con il grafico, senza l'introduzione di ulteriore codice JavaScript, dato che le funzionalità che permettono l'interattività sono già integrate all'interno della libreria. L'utente ha a disposizione diversi strumenti che compiono operazioni varie. Per esempio, è possibile aumentare o ridurre la dimensione degli elementi del grafico, spostarsi sugli assi, salvare il grafico come immagine e, più importante, chi utilizza la dashboard può sfruttare il *mouse hover* per mostrare ulteriori dettagli dei dati rappresentati. Questa caratteristica è particolarmente favorevole poiché permette di non rendere immediatamente visibili tutte le informazioni, ma di rivelarle solo a seguito di un evento esterno da parte dell'utente, così da mantenere il grafico più pulito. L'interattività con gli elementi della dashboard permette di comunicare i dati in modo più accattivante, così da attrarre l'attenzione del soggetto che ne fa uso. Tuttavia, è utile ricordare che è necessario bilanciare gli elementi statici e quelli dinamici; l'interattività non deve essere fonte di distrazione. Infatti, l'obiettivo è sempre quello di trasmettere le informazioni in una forma indubbia, permettendo al contenuto di essere lampante e preciso.

I dati che sono visualizzati all'interno di *MoonUtilities* si possono raggruppare in tre categorie e, per ognuna di esse, si distingue la tipologia di grafico più adatta:

- *amounts*: si tratta di singoli numeri (es.: medie, somme) relativi a un determinato aspetto di interesse. Per questo tipo di dati, è possibile utilizzare dei box[9], se si tratta di un singolo valore, e dei grafici a barre, poste orizzontalmente o verticalmente, dove l'altezza di ogni barra rappresenta il valore di una particolare categoria presa in esame. Se si vogliono rappresentare più proprietà, è possibile affiancare più barre;

Fig. 8: esempio di un box.

[9] Si tratta di un elemento grafico che contiene il valore da rappresentare e una breve descrizione di cosa rappresenta. È anche possibile includere un'icona per caratterizzare meglio il contenuto.

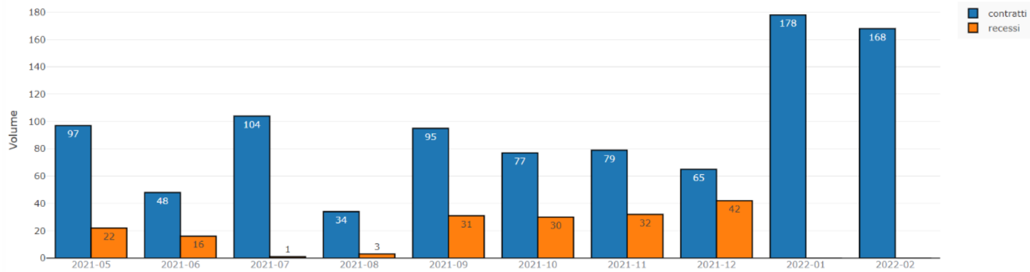

Fig. 9: esempio di un grafico a barre affiancate.

- *proportions*: per proporzioni s'intende che il dato da rappresentare può essere suddiviso in delle parti, la cui somma forma il totale. Un esempio è la suddivisione dei point in corner e segnalatori: la somma delle due quantità fornisce l'intero insieme delle agenzie. In questi casi, è possibile utilizzare grafici a torta, grafici a barre oppure grafici a barre impilate.

  All'interno della dashboard è stato scelto di utilizzare i grafici a torta (Fig. 10) e a barre, ritenendo che questi enfatizzino meglio contenuti di questo genere;

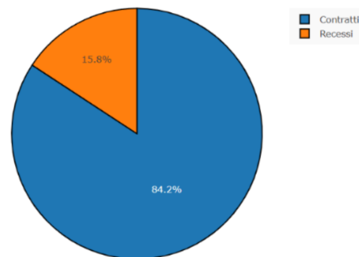

Fig. 10: esempio di un pie plot (grafico a torta).

- *time series*: si tratta di dati ordinati in base al tempo. In queste situazioni, l'ideale è sfruttare grafici a linee in due dimensioni, dove sull'asse delle ascisse sono riportate le date mentre sulle ordinate il valore delle osservazioni. Si tratta di una rappresentazione particolarmente significativa quando si desidera osservare i cambiamenti del dato osservato nell'intervallo considerato.

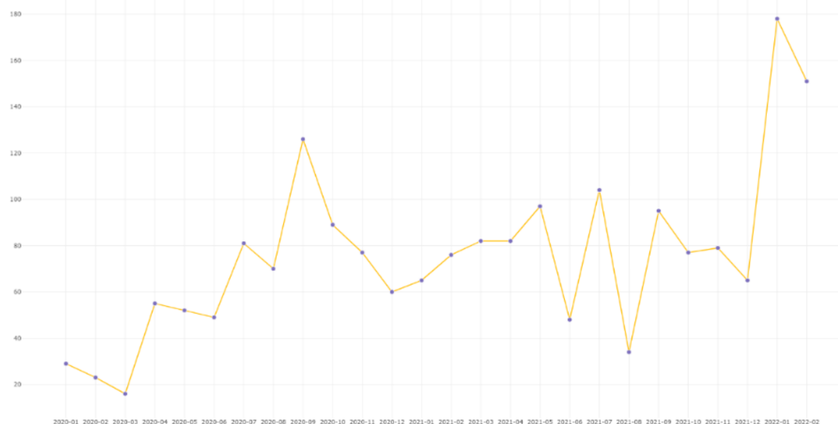

Fig. 11: line plot per serie temporale.

La suddivisione appena illustrata è stata acquisita sempre dal libro *Fundamentals of Data Visualization* (Claus O. Wilke) ed è utilizzata come linea generale per la creazione dei grafici. È importante ricordare che non esistono regole ferree per la rappresentazione di un determinato tipo di dato, bensì bisogna sempre scegliere la rappresentazione che si adatta meglio al contenuto da rappresentare e che possa mettere in luce le informazioni che si vogliono mostrare [45].

La giusta visualizzazione per il risultato dell'interrogazione che si desidera rappresentare spesso non rientra in una delle categorie appena esposte. In queste occasioni, è necessario scegliere singolarmente quale grafico utilizzare, in modo che l'informazione sia coerente con la rappresentazione. Ad esempio, per l'amministrazione è necessario conoscere il numero e la percentuale di contratti stipulati suddivisi per tipologia: luce e gas. Questo suggerisce l'utilizzo di un grafico a torta, il quale, tuttavia, presenta una problematica: il confronto visivo tra proporzioni non è immediato. Infatti, questa tipologia di rappresentazione è caratterizzata dal fatto che i valori sono mappati tramite l'ampiezza degli angoli. Tuttavia, chi osserva, non valuta l'ampiezza degli angoli, bensì l'area, aspetto che nei grafici a torta è percepito in modo differente rispetto alle aree rappresentate nei grafici a barre. Questo accade perché l'occhio è maggiormente allenato nel valutare le *distanze* e non le *aree* [46]. Nei grafici a barre, l'area è un aspetto secondario, poiché ciò che viene percepito è la distanza del valore dal punto iniziale, ovvero l'asse delle ascisse (nel caso in cui queste siano poste verticalmente).

Per questo motivo è stato scelto di affiancare un grafico a barre con colonne ordinate al grafico a torta. Così facendo, il paragone tra le varie parti diventa più semplice e intuitivo.

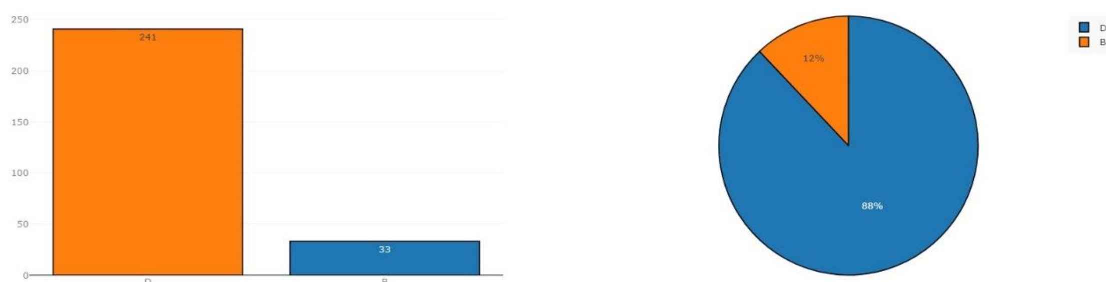

Fig. 12: grafico a barre e a torta errati.

In Fig. 12 sono riportati i grafici di quanto si voleva dimostrare. Ordinare le colonne del grafico a barre serve ad enfatizzare la differenza tra le varie grandezze, fornendo un supporto maggiore per l'interpretazione.

Tuttavia, l'esempio riporta un errore, che potrebbe condurre l'utente ad una interpretazione scorretta. Infatti, i colori del grafico a destra non rappresentano la stessa

suddivisione di quello a sinistra. È sufficiente osservare il colore blu: a sinistra rappresenta la quantità minore, mentre a destra quella maggiore. Questo è dovuto ad una mappatura non corretta dei colori all'interno dei due grafici. Per risolvere questa problematica, è sufficiente modificare la mappatura dei colori all'interno dei grafici per far sì che non si contraddicano.

È fondamentale che quando si sviluppano dei grafici che hanno lo scopo di rappresentare la medesima quantità o delle informazioni tra loro correlate, questi siano coordinati, così da non permettere errori di interpretazione. Oltre a ciò, una caratteristica sostanziale da tenere in considerazione riguarda il punto di vista dal quale si osservano i dati e di conseguenza si creano i grafici. Infatti, colori e rappresentazioni differenti possono trasmettere messaggi diversi, spostando l'attenzione dell'utente e alterando la sua percezione dei fatti. Un esempio banale è utilizzare il colore verde per raffigurare aspetti positivi, come la crescita delle vendite, mentre il rosso per un aspetto negativo, come un calo delle stesse. Sebbene talvolta l'impiego di questa convenzione possa facilitare la lettura, è bene ricordare che non sempre si ottiene una buona rappresentazione, dato che potrebbe peccare di oggettività, distinguendo il "buono" dal "cattivo". In molte situazioni, non esiste una distinzione in tale senso e crearla in fase di visualizzazione non è significativo. Per questo motivo, all'interno della dashboard, si è scelto di limitare approcci di questo genere, lasciando che fosse l'utente ad interpretarne il contenuto in modo autonomo. Da questa prospettiva, si è cercato di modellare il contenuto della dashboard in modo da incontrare le esigenze dell'utente, cercando di rispondere ai quesiti che quest'ultimo si potrebbe porre osservando i dati.

Per quanto riguarda le tabelle, sono state implementate tramite il pacchetto {DT}, il quale fornisce un'interfaccia alla libreria JavaScript *DataTables* [47]. Questo strumento è stato scelto per le sue numerose funzioni, quali l'ordinamento degli elementi all'interno delle colonne, con la possibilità di filtrare i dati e di salvarli in formato Excel in locale. Sfruttando le operazioni già implementate all'interno del package, non è necessario costruirle da zero, ottenendo diversi vantaggi, come illustrato nel capitolo 3.2.

Un aspetto che ha accomunato grafici e tabelle è la leggibilità, intesa come facilità dell'utente di osservare un elemento e capirne il significato, senza elementi grafici che possano risultare fastidiosi. Per questo motivo, si è posta particolare attenzione nel tenere pulita l'immagine, eliminando le scritte ridondanti e aggiungendo solo quelle necessarie. Per esempio, si è scelto di non inserire un titolo in ogni grafico, ma di lasciarlo solo

all'interno della pagina, dato che spesso sono presenti più grafici affiancati, come nel caso in Fig. 12. Similmente, anche l'uso delle legende è stato limitato, inserendole solo dove strettamente necessario. Al contrario, per fornire maggiore completezza alle immagini, sono stati sempre inseriti i nomi degli assi, così da poter comunicare quali grandezze fossero coinvolte nella raffigurazione.

Nella sezione 3.2, è stato citato il pacchetto {fresh} [48], introducendolo come un mezzo per personalizzare l'apparenza della dashboard. Anche questo fattore è compreso nella fase di visualizzazione. Il suo utilizzo è motivato dalla volontà di rendere l'intera pagina piacevole all'apparenza e congruente con quanto contiene, oltre alla pagina in cui è contenuta. In particolare, attraverso questo pacchetto è possibile creare un tema, tramite la funzione `create_theme()`, direttamente applicabile all'intera UI, attraverso una chiamata all'interno del file ui.R. Questo metodo consente di modificare i valori delle variabili CSS incluse nel tema di Bootstrap delle applicazioni Shiny e di attribuire loro i valori desiderati. Insieme al pacchetto {fresh} è stato utilizzato il pacchetto {bs4Dash} [49], il quale permette di creare una Shiny app che utilizza un'interfaccia Bootstrap 4 AdminLTE3. In questo modo, si ottiene un prodotto reattivo, moderno e dall'aspetto più familiare per l'utente, dato che questo design è comune a molti siti che potrebbe già aver incontrato.

In conclusione, la visualizzazione è un aspetto fondamentale dell'analisi dati e deve essere eseguita con cura, tenendo a mente *cosa* è necessario visualizzare, *come* realizzarlo e *chi* usufruirà del contenuto. Per la realizzazione della dashboard, questi aspetti sono stati spesso in conflitto, poiché non sempre quanto è gradevole da osservare risulta anche significativo oppure ottimale dal punto di vista computazionale. Attraverso gli aspetti illustrati in questo capitolo è stato possibile conciliare le varie richieste avanzate dalla direzione aziendale, ottenendo consistenza e veridicità.

5. Ottimizzazione

In informatica, per ottimizzazione si intende un processo in grado di cambiare certe caratteristiche di un sistema al fine di migliorare l'impiego delle risorse [50]. Il concetto di ottimizzazione è strettamente correlato con l'efficienza, la cui definizione formale può essere scritta come il rapporto di lavoro svolto (W) per unità di sforzo (Q):

$$\boldsymbol{\eta} = \mathbf{W/Q}$$ [51]

Questa nozione può essere trasportata nel campo dell'informatica affermando che per efficienza s'intende "la velocità con cui il calcolatore può svolgere un determinato compito, data una parte di codice" (*algorithmic efficiency*, "*Efficient R programming*", Colin Gillespie, Robin Lovelace (2016) [52]).

Una peculiarità del linguaggio R è la capacità di eseguire una stessa operazione in molteplici modi differenti, caratteristica ottenuta grazie alla sua flessibilità ed estensibilità tramite i pacchetti (vedi 3.2, *Il linguaggio R*). In questo contesto, per "efficienza" si intende trovare una soluzione al problema da risolvere, tale che questa sia sufficientemente veloce in termini di efficienza computazionale ma anche di efficienza della produttività del programmatore [53]. Questo significa che, data la grande varietà di funzioni disponibili in questo linguaggio di programmazione, non bisogna dimenticare che se l'obiettivo è semplicemente quello di *fare*, piuttosto di capire *perché* una soluzione sia migliore di un'altra, esplorare tutte le possibilità sarebbe un utilizzo inefficiente del tempo per il programmatore, dato che esistono molteplici vie. In questo senso, in linguaggio R è più semplice trovare *una* soluzione per svolgere un determinato compito, rispetto ad altri linguaggi di programmazione, mentre trovare la via *migliore* è un'attività più complessa, dato l'elevato numero di strade percorribili.

Un primo passo per scrivere codice efficiente in linguaggio R è scegliere il giusto pacchetto. Infatti, diversi package sono ottimizzati in modo da fornire prestazioni migliori o collaborare meglio tra loro (per esempio {tidyverse}). Così, per alcune operazioni ottenere l'efficienza computazionale risulta abbastanza semplice.

Per verificare, tra un insieme di funzioni, quale sia la migliore in termini di tempi di esecuzione si può utilizzare la funzione `microbenchmark()` del pacchetto {microbenchmark}.

Questo metodo misura il tempo necessario per eseguire una o più espressioni passate come argomento [54].

Il metodo è stato applicato per confrontare la funzione `fromJSON()` dei pacchetti {rjson} e {jsonlite}, tramite le istruzioni mostrate nello Snippet 8.

```
microbenchmark(
        jsonlite::fromJSON(txt = "https://..../contratti.php")[10],
        rjson::fromJSON(file = "https://..../contratti.php")
        )
```

Snippet 8: esempio di utilizzo della funzione `microbenchmark()`.

Il risultato prodotto dell'esecuzione di questo Snippet è riportato in tabella.

Unit: milliseconds							
expr	*min*	*lq*	*mean*	*median*	*uq*	*max*	*neval*
jsonlite::fromJSON()	180,74	210,89	240,88	232,84	261,16	418,94	100
rjson::fromJSON()	521,40	573,84	669,46	613,39	694,50	1232,76	100

Le due espressioni sono state eseguite cento volte (*neval*) e i valori della tabella fanno riferimento ai tempi di esecuzione, espressi in millisecondi. Ogni riga è attinente ad una diversa espressione testata, mentre le colonne fanno riferimento all'espressione analizzata (*expr*, qui sono abbreviate per motivi di spazio) al tempo minimo (*min*) e massimo (*max*) riscontrato, quartile inferiore e superiore (rispettivamente *lq* e *uq*), media e mediana (*mean* e *median*) delle osservazioni. In questo modo, si ottiene la conferma di quale funzione sia meglio utilizzare. In questo caso, la funzione `fromJSON()` del pacchetto {jsonlite} fornisce dei tempi migliori, molto inferiori rispetto a quelli della stessa funzione del pacchetto {rjson}. È possibile trarre la medesima conclusione anche osservando il grafico in Fig. 13, raffigurante i risultati ottenuti contenuti nella tabella.

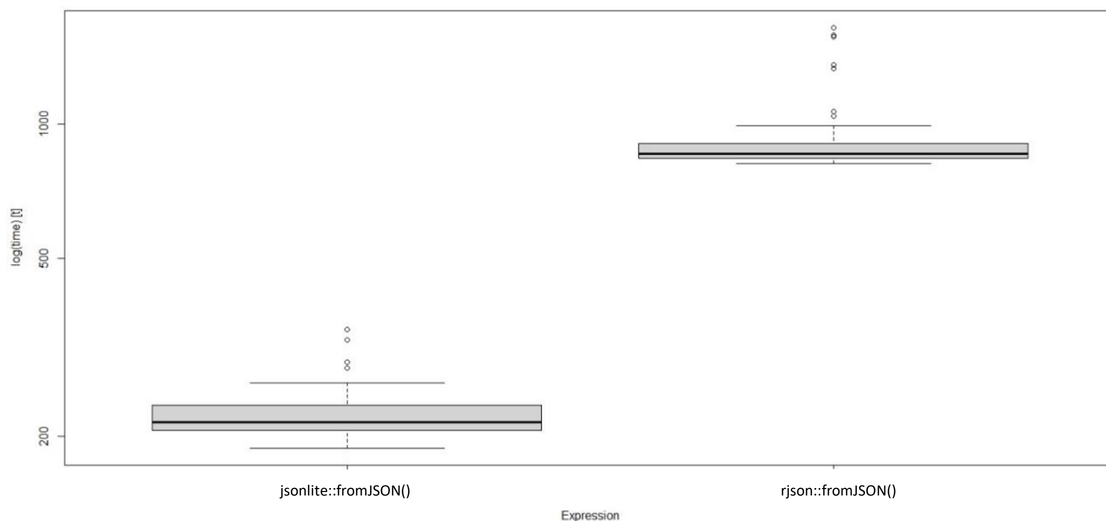

Fig. 13: boxplot del confronto `microbenchmark()`.

[10] I link non sono mostrati interamente per motivi di privacy.

Un procedimento simile è applicabile ogni qual volta ci si trovi in dubbio riguardo a quale funzione utilizzare, in modo da avere conferma su quale sia la soluzione migliore in termini di tempi di esecuzione.

Altri accorgimenti utilizzati per rendere la dashboard più performante sono le reactive expression e l'operatore pipe, aspetti discussi nei capitoli precedenti. Queste ottimizzazioni riguardano i due aspetti presentati in precedenza. Infatti, l'efficienza in linguaggio R è stata descritta sia come efficienza computazionale che in termini di produttività del programmatore. In questo caso, le reactive expression si occupano del primo aspetto, tanto è vero che il principale vantaggio di queste espressioni è che vengono salvate in cache e non sono calcolate nuovamente fino a quando non cambiano le loro dipendenze. Invece, l'operatore pipe non presenta particolari vantaggi dal punto di vista computazionale, bensì è utile per semplificare processi particolarmente complessi e rendere il programmatore più efficiente.

Talvolta, anche in seguito all'implementazione di questi aspetti, è possibile che all'interno dell'applicazione ci siano delle latenze nel caricamento dei vari elementi. Una simile situazione si palesa quando all'interno del codice è presente un elemento che crea un *collo di bottiglia*, rallentando l'intera esecuzione. Per rintracciare le istruzioni che causano dei problemi, è utile servirsi della funzione `profvis()` del pacchetto {profvis}. Questa esegue quanto le viene passato come argomento, restituendo un widget interattivo all'interno di RStudio, tramite il quale è possibile esplorare il profiling delle istruzioni [55]. Un esempio di processo di profilazione è riportato in Fig. 14, dove è stato applicato all'intero codice contenuto nel file ui.R, così da verificare cosa potrebbe causare dei ritardi in fase di costruzione della dashboard.

Nell'immagine sono riportate solo le istruzioni che hanno tempo di esecuzione maggiore di zero e che risultano quindi significative. Una volta rimosse queste righe di codice, è possibile osservare che, in totale, l'intero script impiega poco più di 2000 millisecondi per essere eseguito e che le uniche righe di codice che non hanno svolgimento immediato sono quelle relative all'importazione dei pacchetti e alla chiamata ad altri file, ovvero le varie istruzioni `library()` e `source()`. Una possibile ragione per cui questo si verifica è che le due funzioni devono recuperare degli elementi dall'esterno e successivamente restituirli e salvarli in memoria, aspetto visibile anche all'interno della figura osservando la colonna *Memory*. Queste piccole operazioni non sono visibili, ma si può intuire che impieghino una quantità di tempo maggiore, indipendentemente da quanto mostrato

nell’immagine. In particolare, l’istruzione maggiormente gravosa è `library(‘plotly’)`, necessaria per l’importazione del pacchetto {plotly} all’interno del progetto.

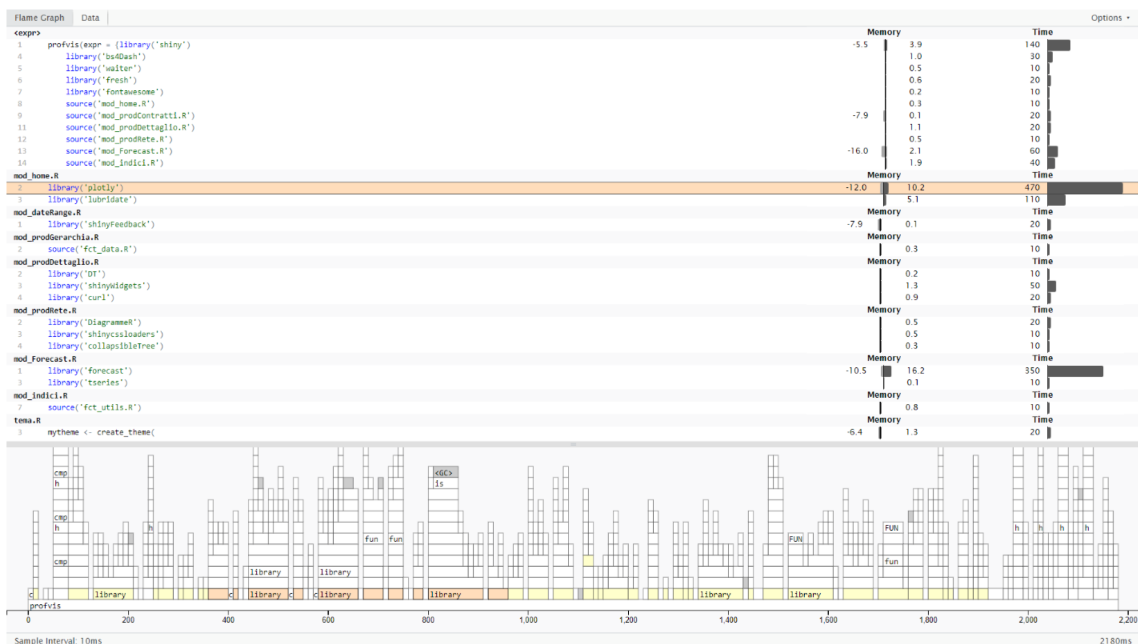

Fig. 14: esempio output `profvis()`.

Da quanto è possibile osservare in questo esempio, una gestione corretta dei pacchetti può portare a dei miglioramenti, sia in termini di memoria che in termini di tempo. Tenendo presente il concetto per cui in linguaggio R esistono molteplici funzioni per compiere le stesse operazioni, un approccio per migliorare questo aspetto è quello di importare inizialmente tutti i pacchetti fondamentali e sfruttare al massimo le funzioni contenute in essi, evitando di selezionare altre librerie per eseguire la medesima funzionalità, se già implementata nel pacchetto di cui non è possibile privarsi. Logicamente, è necessario adattare questo meccanismo a ciascuna situazione e applicarlo solo se fornisce effettivamente delle soluzioni migliori. Per verificare di selezionare i pacchetti e le funzioni maggiormente vantaggiosi, è sempre utile sfruttare le funzioni `microbenchmark()` e `profvis()` in modo iterativo, controllando quali implementazioni possano portare a codice maggiormente efficiente.

6. Conclusioni

Durante lo sviluppo della dashboard *MoonUtilities*, sono emersi alcuni punti che riguardano delle problematiche relative allo strumento, che potrebbero manifestarsi nel caso di crescita dell'azienda.
Un caso concerne quanto potrebbe accadere nell'ipotesi in cui l'impresa aumenti le dimensioni della propria rete nel corso degli anni, con logico aumento del numero dei contratti stipulati e dei dati da mantenere, importare all'interno dell'applicazione e analizzare. È infatti possibile che, dato l'aumento di informazioni, si riscontrino delle latenze all'interno dell'applicativo, più o meno gravi. Una possibile soluzione per evitare sconvenienti di questo genere potrebbe essere di non estrarre tutto lo storico, ma solo i dati strettamente necessari, in base a quanto specificato dal management. In questo modo, si considererebbero solo i dati più attuali, prendendo in esame solo gli anni più recenti. Nel caso si necessiti di visualizzare informazioni antecedenti alle date prestabilite, si inserirebbero all'interno della dashboard dei comandi specifici. Così facendo, si alleggerirebbe il flusso di lavoro complessivo, rendendolo più gravoso solo in caso di bisogno.

Il secondo argomento riguarda la sicurezza. Come illustrato in precedenza, la dashboard sarà inserita in un iframe nell'area privata del sito dell'azienda. Questo comporta che il link sia visibile nel codice HTML della pagina e che chi vi ha accesso possa trovarlo. Nonostante l'utente non sia a conoscenza del meccanismo di scambio ID e nemmeno del proprio ID o degli ID di altri utenti, è comunque importante che non possa portare l'URL al di fuori della pagina. In questo caso, risulta utile nascondere l'URL oppure rendere maggiormente complesso il processo delle chiamate, camuffando gli ID all'interno del link.

Al momento della scrittura, la dashboard *MoonUtilities* non è ancora stata resa disponibile ad utenti esterni all'azienda, dato che sono necessari ulteriori analisi sulla correttezza dei dati e sulla loro coerenza, ma ha già portato alle prime decisioni strategiche della direzione. Possedere uno strumento da cui è possibile visualizzare facilmente e in tempo reale tutte le informazioni utili per il corretto svolgimento delle proprie mansioni è sicuramente un enorme vantaggio. Un esempio di strategia in fase di avviamento a seguito dell'introduzione della dashboard riguarda l'esame della rete di corner e segnalatori: conoscere quanto producono i vari attori della rete consente di capire se si stanno

riscontrando i risultati prestabiliti ed intervenire in caso contrario. Congiuntamente, è possibile osservare l'espansione geografica della rete e iniziare a stendere un piano riguardante futuri ampliamenti.

Quanto illustrato in questo elaborato è l'essenza della dashboard *MoonUtilities*. In un prossimo futuro, saranno introdotte nuove funzionalità in base alle esigenze dell'azienda, così da poter consegnare a tutti gli utenti uno strumento completo ed efficiente.

Sitografia/bibliografia

[1] *Analisi dei dati*, da Wikipedia, l'enciclopedia libera: https://it.wikipedia.org/wiki/Analisi_dei_dati#:~:text=Nell'ambito%20della%20scienza%20dei,supportino%20le%20decisioni%20strategiche%20aziendali.

[2] *Business intelligence o business analytics: Qual è la differenza e quale ti serve?*, Tableau (articolo del sito ufficiale): https://www.tableau.com/it-it/learn/articles/business-intelligence/bi-business-analytics

[3] *A Brief History of Analytics*, by Keith D. Foote on September 20, 2021: https://www.dataversity.net/brief-history-analytics/

Scientific management, da Wikipedia, l'enciclopedia libera: https://en.wikipedia.org/wiki/Scientific_management#:~:text=Scientific%20management%20is%20a%20theory,economic%20efficiency%2C%20especially%20labor%20productivity.&text=Scientific%20management%20is%20sometimes%20known,its%20pioneer%2C%20Frederick%20Winslow%20Taylor.

Taylorismo, Andreas Fasel, traduzione di Martin Kuder (Versione del 14.08.2012): https://hls-dhs-dss.ch/it/articles/013883/2012-08-14/

What Is Business Analytics?, by NDMU (September 21, 2018): https://online.ndm.edu/news/analytics/what-is-business-analytics/

[4] Moon Energy S.r.l., sito ufficiale: https://www.moonutilities.com/

[5] *Business* intelligence, da Wikipedia, l'enciclopedia libera: https://it.wikipedia.org/wiki/Business_intelligence

[6] *Dashboard (business)*, da Wikipedia, l'enciclopedia libera: https://en.wikipedia.org/wiki/Dashboard_(business)

[7] *Top 6 Programming Languages for Data Science in 2021*, DASCA (Data Science Council of America), 2021: https://www.dasca.org/world-of-big-data/article/top-6-programming-languages-for-data-science-in-2021

[8] *What is R? Introduction to R*, The R Foundation: https://www.r-project.org/about.html

[9] *Efficient workflow: Package selection* (sezione 4.4) in *Efficient R programming*, Colin Gillespie, Robin Lovelace, O'Reilly Media, (2016): https://bookdown.org/csgillespie/efficientR/workflow.html#package-selection

[10] *Layout, themes, HTML: Under the hood* (sezione 6.6) in *Mastering Shiny* Hadley Wickham, O'Reilly Media, (2020): https://mastering-shiny.org/action-layout.html%23under-the-hood#under-the-hood

[11] *Shiny HTML Tags Glossary*, Garrett Grolemund, (last updated on 08 august 2017): https://shiny.rstudio.com/articles/tag-glossary.html

[12] *shinyapps.io user guide*, shinyapps.io team (2021-12-21): https://docs.rstudio.com/shinyapps.io/

[13] *Your first Shiny app: Introduction* (sezione 1.1) in *Mastering Shiny* Hadley Wickham, O'Reilly Media, (2020): https://mastering-shiny.org/basic-app.html

[14] *The basic parts of a Shiny app* (last updated on 28 JUN 2017): https://shiny.rstudio.com/articles/basics.html

[15] *Basic reactivity: The server function* (sezione 3.2) in *Mastering Shiny* Hadley Wickham, O'Reilly Media, (2020): https://mastering-shiny.org/basic-reactivity.html

[16] *Funzione (informatica),* da Wikipedia, l'enciclopedia libera: https://it.wikipedia.org/wiki/Funzione_(informatica)

[17] *Functions* (capitolo 18) in *Mastering Shiny* Hadley Wickham, O'Reilly Media, (2020): https://mastering-shiny.org/scaling-functions.html

[18] *Shiny modules* (capitolo 19) in *Mastering Shiny* Hadley Wickham, O'Reilly Media, (2020): https://mastering-shiny.org/scaling-modules.html

[19] *Create date range input* dalla documentazione ufficiale di Shiny: https://shiny.rstudio.com/reference/shiny/1.6.0/dateRangeInput.html

[20] *Structuring Your Project: Structuring your app: Business logic and application logic* (sezione 3.3.1) in *Engineering Production-Grade Shiny Apps* (Colin Fay, Sébastien Rochette, Vincent Guyader, Cervan Girard, 2022): https://engineering-shiny.org/structuring-project.html#structuring-your-app

[21] *Basic reactivity* (capitolo 3) in *Mastering Shiny* Hadley Wickham, O'Reilly Media, (2020): https://mastering-shiny.org/basic-reactivity.html

[22] *LESSON 6: Use reactive expressions* dalla documentazione ufficiale di Shiny: https://shiny.rstudio.com/tutorial/written-tutorial/lesson6/

[23] *Why reactivity?*: *Why do we need reactive programming?: Reactive programming* (sezione 13.2.4) in *Mastering Shiny* Hadley Wickham, O'Reilly Media, (2020): https://mastering-shiny.org/reactive-motivation.html#reactive-motivation

[24] *Reactive building blocks* (capitolo 15) in *Mastering Shiny* Hadley Wickham, O'Reilly Media, (2020): https://mastering-shiny.org/reactivity-objects.html

[25] *Event handler* dalla documentazione ufficiale di Shiny: https://shiny.rstudio.com/reference/shiny/latest/observeEvent.html

[26] *Create a reactive observer* dalla documentazione ufficiale di Shiny: https://shiny.rstudio.com/reference/shiny/0.14/observe.html

[27] *Observer pattern*, da Wikipedia, l'enciclopedia libera: https://en.wikipedia.org/wiki/Observer_pattern

[28] *Introduzione a JSON*: https://www.json.org/json-it.html

[29] *toJSON, fromJSON: Convert R objects to/from JSON*, Powered by DataCamp: https://www.rdocumentation.org/packages/jsonlite/versions/1.7.3/topics/toJSON,%20fromJSON

[30] *Session object* dalla documentazione ufficiale di Shiny: https://shiny.rstudio.com/reference/shiny/1.6.0/session.html

[31] *tidyr* (versione 1.2.0): https://tidyr.tidyverse.org/index.html

[32] *data.frame: Data Frames*, Powered by DataCamp: https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/data.frame

[33] *tibble (part of the tidyverse): Overview*: https://tibble.tidyverse.org/index.html

[34] *tidyverse: R packages for data science:* https://www.tidyverse.org/

[35] "*R for Data Science*", Hadley Wickham, Garrett Grolemund, O'Reilly Media , 2016 (paragrafo "tidy"): https://r4ds.had.co.nz/introduction.html

[36] *tidy data*, tidyr.tidyverse.org, Hadley Wickham, Maximilian Girlich: https://tidyr.tidyverse.org/articles/tidy-data.html#tidy-data

[37] "*R for Data Science*", Hadley Wickham, Garrett Grolemund, O'Reilly Media, 2016 (paragrafo "transform"): https://r4ds.had.co.nz/introduction.html

[38] *dplyr* (versione 1.0.8): https://dplyr.tidyverse.org/index.html, https://dplyr.tidyverse.org/articles/dplyr.html

[39] [40] *Database Queries With R*: https://db.rstudio.com/getting-started/database-queries

[43] *Fundamentals of Data Visualization*, Claus O. Wilke, O'Reilly Media (2019): https://clauswilke.com/dataviz/index.html

[41] *Significato di Mapping*: https://www.ehiweb.it/significato/m/mapping/

[44] *Getting Started with Plotly in R*: https://plotly.com/r/getting-started/

[45] *Visualizing proportions* (capitolo 10) in *Fundamentals of Data Visualization*, Claus O. Wilke, O'Reilly Media (2019): https://clauswilke.com/dataviz/visualizing-proportions.html

[42] *Visualization: The Principle of Proportional Ink*, Carl Bergstrom, Jevin West: https://www.callingbullshit.org/tools/tools_proportional_ink.html

[46] *The principle of proportional ink: Direct area visualization* (sezione 17.3) in *Fundamentals of Data Visualization*, Claus O. Wilke, O'Reilly Media (2019): https://clauswilke.com/dataviz/proportional-ink.html#direct-area-visualizations

[47] *DT: An R interface to the DataTables library*: https://rstudio.github.io/DT/

[48] *fresh* (versione 0.2.0.900): https://dreamrs.github.io/fresh/index.html

[49] *bs4Dash* (versione 2.0.4.9000): https://rinterface.github.io/bs4Dash/index.html

[50] *Ottimizzazione*, Vocabolario on line: https://www.treccani.it/vocabolario/ottimizzazione/

[51] *Introduction, What is efficiency* (sezione 1.2) in *Efficient R programming*, Colin Gillespie, Robin Lovelace, O'Reilly Media, (2016): https://bookdown.org/csgillespie/efficientR/introduction.html#what-is-efficiency

[52] *Introduction, What is efficiency* (sezione 1.2) in *Efficient R programming*, Colin Gillespie, Robin Lovelace, O'Reilly Media, (2016), definizione "*algorithmic efficiency*": https://bookdown.org/csgillespie/efficientR/introduction.html#what-is-efficiency

[53] *Introduction, What is efficient R programming?* (sezione 1.2) in *Efficient R programming*, Colin Gillespie, Robin Lovelace, O'Reilly Media, (2016): https://bookdown.org/csgillespie/efficientR/introduction.html#what-is-efficient-R-programming

[54] *microbenchmark: Sub-millisecond accurate timing of expression evaluation*: https://www.rdocumentation.org/packages/microbenchmark/versions/1.4.9/topics/microbenchmark

[55] *Profvis*: https://www.rdocumentation.org/packages/profvis/versions/0.3.7