Big Data Technologies Project

Master in Data Science
University of Trento

CENSUS web application

Calculator of Earnings Nationally Scored via User Specifics

Lucia Hrovatin
223027
lucia.hrovatin@studenti.unitn.it

Elisa Paolazzi
224424
elisa.paolazzi-2@studenti.unitn.it

***Abstract*—The income of individuals or households determines the spendable budget for acquiring and owning essential items (e.g., accommodation) and constraints their features (e.g., house sqm. or location). This project aims at building a system able to estimate the income bracket of Italian citizens based on census data. A Random Forest algorithm performs the categorisation relying on subgroups of the Italian taxation system (Irpef).**

***Keywords*—Italian census data, Airflow and Celery executor, MySQL, RedisML, Random Forest**

I. Introduction

The project assignment was open to several interpretations due to its vagueness and potential wide scope. The assignment understanding hereby proposed implements an estimator of household income in the Italian framework. Whereas most of the available data reports aggregate values (e.g., average rates), accessible microdata is scarce and outdated. To overcome the resource scarcity, two generalisations are employed: an aggregation of data pinpointing to different years (2014-2016) and a classification method relying on income brackets. The income brackets follow the Irpef segmentation and are further specified into groups whose gap is around 7000 Euros. This structured and static data is downloaded from Banca d'Italia and Sole24Ore websites. The most natural choice is a batch processing pipeline that gets data from the sources, organises it into tables, and extracts insights by deploying a categorisation model.

II. System Model

A. System architecture

The project sets up a horizontally scaled and potentially fault tolerance batch processing pipeline. The system architecture, shown in Fig.1, is a sequence of stages that fetches raw data from the Banca d'Italia and Sole24Ore websites and returns a final classification model within a web application. Specifically, five stages are embedded into the system.

- *Collection and ingestion stage*
The first stage fetches data simulating the basic functionalities of an API. It requires the URL's resource scheme, sends an HTTP/1.1 request, and gets raw data from the webserver. Then, the documents are unzipped and the datasets of interest are selected. In order to guarantee data integrity and avoid local storage, this first stage is integrated into a *Directed Acyclic Graph* (DAG) in an Airflow environment. This architecture grants intermediate storage within the PostgreSQL backend database and a later retrieval via relative paths.

- *ETL stage*
Extract Transform Load (ETL) is a general term for describing a specific workflow which transfers data from one source to another. The input is a source producing data, which then is consumed by a sink returning the cleansed data as output. The entire process can be seen as a DAG [1], whose management becomes more complex when dealing with Big Data. Several software are able to orchestrate the process and enable the integration of heterogeneous systems and technologies [2]. Among them, Apache Airflow has been chosen. Even if the data handled by this system is not definable as Big Data, a resembled distributed environment (supported by Docker) is set up. Indeed, Airflow is scaled out with Celery Executor, which distributes the workload across worker node(s). The Celery queue is complemented by Redis message broker and PostgreSQL metadata database. As shown in Fig. 2, three main working blocks are present. The first, the metadata database, keeps a record of each task's status and acknowledges the scheduler. The scheduler determines how to process the tasks (second block) and sends messages to the worker node(s) employing a message broker (third block).

- *Data storage*
The computations of the ETL stage clean the datasets from useless data and load the results in the `project_bdt` database on MySQL. This storing procedure benefits from the hosting network automatically set up by Docker, but implies an *a priori* knowledge of the data organisation, embedded relations among tables, and attribute constraints (e.g., format). In this case, the documentation attached to the datasets [3]

TABLE I
Example of table stored in the database

Final database							
nquest	***nord***	***ncomp***	***anasc***	***sex***	***staciv***	***ireg***	***y***
PK int	int	int	int	int	int	int	int

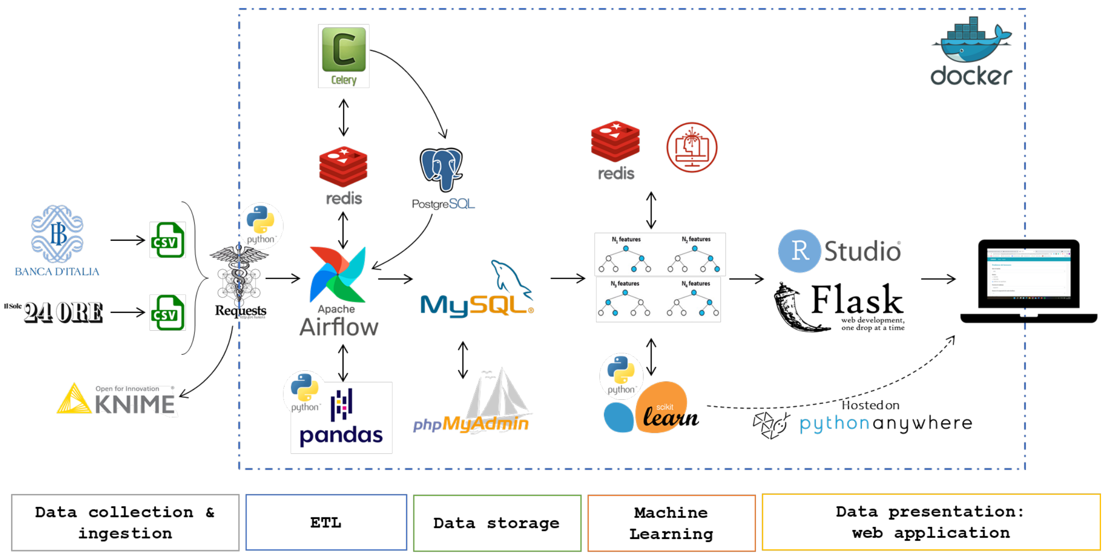


Fig. 1. Pipeline deploying the web application.

enables a straightforward definition of the schemata. The resulting tables, `final` and `final_individual`, share a common schema, reported in Table I, but store different data. While `final` gathers the household income and uniquely identifies families with the attribute *nquest* marked as primary key, the `final_individual` table considers singular income. Thus, each individual is a distinct record and is uniquely referenced via *nquest* and *nord*.

- *Serving layer: machine learning*

The pipeline serving layer corresponds to two classification models relying on Random Forest algorithm. Whereas the *RandomForestClassifier* provided by the Python library **scikit-learn** performs predictions for the web application hosted on a third-party server, another method exploits Redis and its machine learning module (RedisML). It works via a two-step procedure. The first step, training phase, trains and stores the model(s) in-memory; the second step, testing phase, recalls the model(s) and returns a prediction with extremely low latency. However, in-memory storage does not guarantee long-term retention and represents a significant drawback of this approach.

- *Data presentation*

The data pipeline ends by launching the *CENSUS* web application. The user interacts directly with the interface and receives the predicted income bracket based on previously trained and stored models. As cited above, the latency should be extremely low compared to the stable and user-friendly version of the web application hosted on the third-party server.

B. Technologies

- *Wide scope technologies*

KNIME. When working with new data, pursuing a rough understanding of its logic and semantics should be the first objective. Therefore, the data exploration was performed on KNIME platform, whose GUI allows creating and visualising modular data pipelines without coding.

Docker. Replicability and reproducibility are fundamental pillars for many academic disciplines. However, in software engineering the lack of portability, undocumented dependancies and configurations make reproducibility hard to achieve [4]. Docker overcomes this technical obstacle by offering an open-source technology based on *containers*. A container can be seen as a lightweight virtual machine that sets up a computational environment, including all necessary dependancies, configurations, and data within a single unit (i.e., *image*).

- *Ingestion stage*

Requests library. As stated above, the data ingestion is performed by the Requests Python library sending HTTP requests to the target URLs and getting back either zipped folders or `csv` files. Programmatically downloading the resources avoids the possible corruptions that may occur when storing or handling files on the local machine. Even though the exposed endpoints are supposed to be persistent and stable, the academic literature that references web content using URLs, often faces failures and inaccessible links: the *link rot* phenomenon [5]. Therefore, any server issue, page removal, and content relocation may break the pipeline and return an HTTP 404 error.

- *ETL stage*

Apache Aiflow with Celery Executor. With a certain degree of generalisation, the pipeline's core can be considered a DAG with different processing operators that transfer data from one source to another. The ETL procedure may face difficulties when dealing with Big Data complexity. A suitable software, such as Apache Airflow, is needed. The choice of this specific solution can be justified by the decoupling between the program itself and the managerial functions. Indeed, Airflow works with *operators* (here, PythonOperators), uses the same programming language to describe all the operations and keeps the DAGs in the same version control [2]. In the project, the Apace Airflow environment simulates a distributed system aiming at scalability and fault tolerance. The system spins up a Docker container for each component involved in the workflow. Whereas the webserver and scheduler are responsible for accessing DAG and task status, Celery triggers and adds the necessary tasks to the queue. The last two components, Redis and PostgreSQL, manage the flow of messages and information.

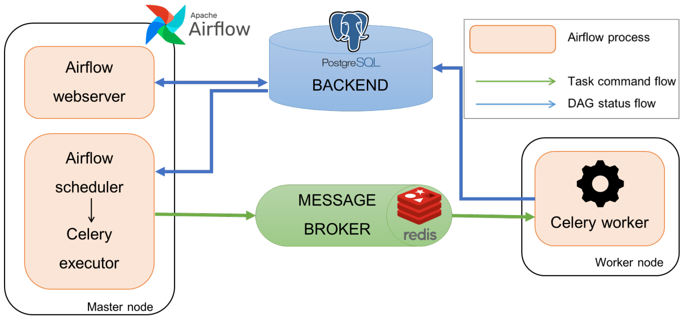


Fig. 2. Simulated distributed environment in Apache Airflow

Celery. Simpleness, flexibility, and reliability are some of Celery distributed task queue qualities. This service distributes and schedules tasks while other tools execute them. Celery can communicate with many different backends (i.e., Result Stores) and brokers (i.e., Message Transports). While any message broker should guarantee remote control and be monitored via **Flower** (administrator of Celery clusters), the result backed is SQLAlchemy. SQLAlchemy is an open-source object-relational mapper (ORM) that enables Celery to connect with SQL databases, which store metadata about the status of DAG and tasks.

Redis. Redis, *REmote DIctionary Server*, stores data structures in key-value manner. However, its scope may be enlarged to publish/subscribe messaging engine [6]. If implemented as message broker, Redis identifies a client (i.e., Airflow Scheduler) that publishes messages into channels using the *Redis Serialization Protocol* (RESP). The subscribers (i.e., Celery workers) receive the messages in the same order they were published. Even though the plethora of message broker services is broad, Redis shows outstanding speed and lightweight in message transmission [7].

PostgreSQL. PostgreSQL is a free and open-source object-relational database management system (ORDBMS). Its core features are extensibility, SQL compliance, referential integrity, and Multi-Version Concurrency Control (MVCC), which provides transaction isolation and eschews locking procedures [8]. In the Airflow environment, PostgreSQL is used as backend metadata database. It keeps a record of each task status and communicates it to webserver and scheduler. The association between PostgreSQL and Redis is deeply rooted in the implementations posted online, but a popular alternative poses MySQL as backend and RabbitMQ as message broker.

Pandas library. Even though it is not a proper technology, Pandas powerful functionalities have a central role in the functions run by PythonOperators.

- *Data storage*

MySQL. The tabular data handled in this project needed to be stored in a relational database. Among the open-source RDBMS, MySQL shows the most attractive features. Portability[1], support of various replication services, robustness, and a significant storage capability (or, at least, bigger than PostgreSQL) should clarify the choice. However, an additional feature tipped the balance in favour of MySQL. Whereas MySQL outperforms other relational databases in select queries, writing operations are rendered with similar efficiency [9]. Considering the existence of a tradeoff between efficiency and resources (i.e., latency) required in writing/reading operations, MySQL seems to be the most suitable tool for the project's needs.

phpMyAdmin. The correctness of SQL computations can be monitored on phpMyAdmin, a software tool written in the popular and general-purpose scripting language PHP. It administrates MySQL and supports a wide range of frequently used operation (e.g., databases, tables, and relations management) which can be performed either via user interface or SQL statements.

- *Serving layer: machine learning*

Despite the widespread interest in machine learning (ML), real-time usages still fail to support the prediction side of ML. Considering this limitation, the project presents two opposite approaches. The classical one, based on the RandomForestClassifier model of **scikit-learn** library, shows a consistent drawback when implemented in the real-time application. Thus, at every request it must parse the entire data, (re)train the model, and return the prediction for the input data. On the other hand, the second approach dives deeper into a more accessible ML pipeline using Redis functionalities.

RedisML and Redis. As stated above, Redis stores data in a key-value manner. The vagueness of the term *data* led to the development of RedisML module. It enhances the ML prediction/evaluation process by training the classification models, storing them in Redis, and retrieving them with extremely low latency. This process requires two components: a Redis instance and RedisML module, both preloaded in the used Docker container.

- *Data presentation*

RStudio. Data visualisation leads to a better understanding of data (i.e., Exploratory Data Analysis) and helps the user to focus on the most relevant aspects of data. These objectives are achieved by the graphs produced in RStudio, an open-source software that combines various components of R into one productive and seamless workbench. Even though more professional visualisation tools are available, RStudio fits the project's purposes.

Flask. The final interface is developed using Flask, which is a WSGI web application framework written in Python.

[1]Portability is a relevant feature, but it did not really influence the choice. Thus, this project implements the relational databases in Docker containers.

The design allows a quick and easy implementation, with the ability to adapt to complex applications and integrate Python codes. The Flask documentation [10] offers many suggestions for code structure and syntax. Moreover, in a Flask project, no dependancies or predefined layouts are applied: tools and libraries are chosen only if they suit the needs.

III. Implementation

To parallelise the project development and to easily access the code, a GitHub repository has been created and populated with the folders described below.

Starting from the Flask's GUI, the elements that enable its visualisation and general operation process are:

- `main.py` contains the route functions (defining variables, actions and events of each page) and the function required to launch the application on the local server
- `forms.py`, defines the app's form and its fields
- `templates`, contains the `HTML` template for each page. `layout.html` sets the general template and elements (navbar and footer), while `home.html` structures the form and the table showing user's data and prediction
- `static` folder with the `CSS` presentation file (layout, colors, and fonts) and the images exported from RStudio

A central role is played by `forms.py`, which manages the temporary collection of user's data and converts it into the most suitable format for the algorithms (i.e., integer number). Notice how the user's data are not further stored in any database. Moreover, the user enters a predefined input, based on the fields:

- *year of birth*, within 1932-2002. The choice of constraining the variable between these temporal boundaries is supported by three general assumptions. Firstly, the model is trained on data from surveys submitted to over 18 years old respondents. Secondly, teenagers (under 18) usually do not earn an income. Thirdly, the target audience does not include people born before 1932 due to their improbable interest in a web application.
- *gender* (male, female, prefer not to say)
- *province of residence*
- *number of family components*
- *marital status*, set accordingly to the documentation provided by the Banca d'Italia [3]

In order to engage and inform the user, a range of graphs is offered. They show interesting aspects of data from the Sole24Ore *Qualità della vita 2020* survey [11]. This extra content aims at providing an updated overview of the social welfare condition in Italy. The graphs' scripts are written in R (`Rmd` files), exported as images and integrated to the templates.

As cited above, enhancing reproducibility was a central objective of the project. Docker accomplishes this thanks to its containers and the configuration defined in `docker-compose.yml`:

- the official Apache-Airflow image implemented with Celery Executor, Flower monitor service, PostgreSQL as backend and Redis as message broker
- the official MySQL image and its web interface phpMyAdmin, used to create and populate the database after the ETL phase
- the Docker image `shaynativ/redis-ml` employed in the ML procedure

After the manual activation of Docker, also `runner.py` should be launched via command line. It starts the pipeline and, specifically, the following DAGs (source code in `airflow\dags`):

- `first_dag.py`, automatically requests and downloads the datasets
- `second_dag.py` deals with data cleansing and loads the transformed data to MySQL
- `third_dag.py` firstly joins the tables `carcom` with `rper` and with `rfam` using a `FULL OUTER JOIN`. It joins the tables either on the primary key *nquest* or on the couple *nquest-nord*, which identifies individuals instead of households. Secondly, it merges (`UNION`) the data from 2014 with 2016 leaving out the 2014 records of all the households that participated to both surveys. This implementation should update, where possible, the handled data. The final reference tables `final` and `final_individuals` are further updated using a `CASE` statement that constraints the income (y, in the tables) to nine discrete classes

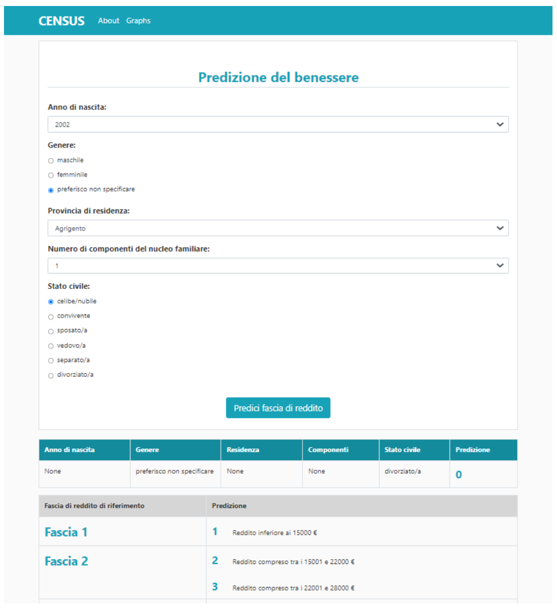
Any publisher/subscriber messaging service manages the workflow. Indeed, the DAGs are triggered by API when the *success* state of the previous DAG is received. While testing the system, the Airflow REST API often blocked the communication due to a requests' overload. To avoid blocks, the `sleep()` function freezes the code and allows a retrial only after 30 seconds. This temporal interval has been set using a trial and error procedure; therefore, it may still break down under other conditions.

The last folder is `src`, where `saver.py` is placed. It defines the Python connector for MySQL (`MySQLmanager`) and its functions for data retrieval and update. Also `collector.py` and `province_ita.json` files are encased in `src` and deal with the definition of Italian regions. Unfortunately, no dataset with geographical precision at the NUTS3 level (i.e., provinces) has been found. Thus, the original data are based on NUTS2, which are the region codes ranging from 1 to 20. To pursue consistency with the training data, an easy procedure transforms the input values. The user selects the province of residence and this input, a string with the province denomination, is matched with the relative region and converted to integer.

The last file in `src` is `classifier.py`. It contains the two ML models: the classification algorithm implemented using Python libraries and the RedisML approach. Both classifiers are *Random Forest* (RF) algorithms, a statistical learning model based on randomised ensemble of decision trees [12]. These RF algorithms take as input the user's data (already converted), and outputs an integer representing the predicted income bracket. Three reasons justify the choice of RF. Firstly, it has a good predictive accuracy even when predictors contain noise or are highly correlated (as in this case[2]). Secondly, the classification trees are independant and built with two levels of randomness, avoiding any risk of overfitting and predictor predominance [13]. Lastly, it outperforms the other tested classifiers: *Quadratic Discriminant Analysis* (QDA) and *k-Nearest Neighbours* (k-NN). The RF has a parameter, the degree of depth, that has been tuned using the validation set approach and misclassification error rate comparison [13]. The algorithms are trained with the tables `final_individual`, about single individuals, and `final`, referring to households. The value of marital status determines the table to consider. If the option *celibe/nubile* is chosen, then the prediction is based on `final_individual`. Furthermore, since the form offers the possibility of not specifying one's gender, the algorithms can exclude the *sex* attribute from the prediction.

IV. Results

The project's final result is a web application entitled *CENSUS* (Calculator of Earnings Nationally Scored via User Specifics). Since the target audience is Italian citizens, the interface's main language is Italian.



Predizione del benessere

Anno di nascita: 2002

Genere:
- maschile
- femminile
- [✓ preferisco non specificare]

Provincia di residenza: Agrigento

Numero di componenti del nucleo familiare: 1

Stato civile:
- [✓ celibe/nubile]
- convivente
- sposato/a
- vedovo/a
- separato/a
- divorziato/a

Anno di nascita	Genere	Residenza	Componenti	Stato civile	Predizione
None	preferisco non specificare	None	None	divorziato/a	0

Fascia di reddito di riferimento	Predizione
Fascia 1	1 Reddito inferiore ai 15000 €
Fascia 2	2 Reddito compreso tra i 15001 e 22000 €
	3 Reddito compreso tra i 22001 e 28000 €

Fig. 3. CENSUS home page

The access can be performed in two ways:

- by connecting to the web address (hosted on pythonanywhere [14] server)
- following the instructions contained in the `README.md` on GitHub repository.

The second option suits more proficient users, interested in a deeper understanding of the structure beyond the web application and its predictions. After launching the web application, the home page is returned, shown in Fig.3. It displays the form to fill up to receive the *income bracket prediction*, a description of each income bracket referenced to the five Irpef groups, and a choropleth map showing the Italian household expenditure distribution. Through the navbar, the user can easily reach two other pages:

- *About*, indicating project's objective, licenses, and direct links to databases and resources
- *Grafici*, containing graphs about social welfare in Italy

V. Conclusions

The deployed infrastructure handles this data in a satisfactory manner. However, some limitations and critical points could be found at both logical and semantic levels.
Thus, the project merges data from different years, assuming a similar welfare. The scarceness of data justifies this approach, but it is biased and may not be applicable in other contexts. For example, merging 2019/2020 would not be possible due to the Covid19 pandemic and its impact. Other semantic issues regard the usage of outdated data and some debatable choices in the web application (e.g., exclude under 18).
Logically, the infrastructure might show weaknesses when dealing with bigger data (for volume and variety). In several stages, criticisms could be pointed out :
- *Ingestion phase*: no precautions are taken about the possible corruption of an URL. An alternative approach would request the data only once and then persistently store it in a Data Warehouse or Data Lake.
- *ETL phase:* the Airflow environment has one worker node. This choice was meaningful in this framework, but it does not guarantee fault tolerance. A better implementation would enhance the plethora of workers to distribute the workload. Another debatable choice is the use of Redis instead of RabbitMQ. Even though they reach similar performances, Redis does not guarantee message reception, slows down when dealing with heavy messages, and is susceptible to data loss in the event of power failures or abrupt termination.
- *MySQL phase:* MySQL seemed a practical choice, but it has a worrying latency when loading tables for the first time (write operation). Alternatively, Data Marts could be extracted from the DW cited above and integrated in Apache Spark as Resilient Distributed Datasets (RDD). This would allow in-memory computations in a fault-tolerant and parallel manner. Moreover, Spark supports and optimises the ML process in Redis[3].
Overall, the system may also be enhanced by introducing a publisher/subscriber engine. It would better manage the workflow and strengthen the infrastructure by avoiding server shutdowns or blocks due to requests' overload or high latency.
A last remark regards the choice Airflow as orchestration engine. Its greater drawback refers to the *a priori* knowledge required when structuring the DAGs and the absence of a real pipeline optimisation [2]. However, the complex nature of Big Data makes optimisation and pipeline effectiveness strongly context-dependant even if data management and governance are well-defined.

[2]EDA's results are reported in exploratory_data_analysis.Rmd file

[3]Spark-RedisML combined have a great performance, as Databricks shown.

References

[1] T. Koivisto, "Efficient data analysis pipelines," 2019.

[2] P. Ceravolo, A. Azzini, M. Angelini, T. Catarci, P. Cudré-Mauroux, E. Damiani, A. Mazak, M. Van Keulen, M. Jarrar, G. Santucci, *et al.*, "Big data semantics," *Journal on Data Semantics*, vol. 7, no. 2, pp. 65–85, 2018.

[3] B. d'Italia, *Documentazione per l'utilizzo dei microdati*. 2018. https://www.bancaditalia.it/statistiche/tematiche/indagini-famiglie-imprese/bilanci-famiglie/documentazione/index.html.

[4] J. Cito, V. Ferme, and H. C. Gall, "Using docker containers to improve reproducibility in software and web engineering research," in *International Conference on Web Engineering*, pp. 609–612, Springer, 2016.

[5] D. Fetterly, M. Manasse, M. Najork, and J. L. Wiener, "A large-scale study of the evolution of web pages," *Software: Practice and Experience*, vol. 34, no. 2, pp. 213–237, 2004.

[6] F. Gutierrez, "Messaging with redis," in *Spring Boot Messaging*, pp. 81–92, Springer, 2017.

[7] M. Dunne, G. Gracioli, and S. Fischmeister, "A comparison of data streaming frameworks for anomaly detection in embedded systems," in *Proceedings of the 1st International Workshop on Security and Privacy for the Internet-of-Things (IoTSec), Orlando, FL, USA*, 2018.

[8] J. D. Drake and J. C. Worsley, *Practical PostgreSQL*. " O'Reilly Media, Inc.", 2002.

[9] C. Asiminidis, G. Kokkonis, and S. Kontogiannis, "Database systems performance evaluation for iot applications," *International Journal of Database Management Systems (IJDMS) Vol*, vol. 10, 2018.

[10] Pallets, *Flask documentation*. 2020. https://flask.palletsprojects.com/en/2.0.x/.

[11] I. Sole24Ore, *Qualità della vita*. 2020. https://github.com/IlSole24ORE/QDV.

[12] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[13] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning*, vol. 112. Springer, 2013.

[14] P. LLP, *PythonAnywhere*. 2021. https://www.pythonanywhere.com/l.