2023

Olin Group

IMPACT PROJECT MCSBT

ALVAREZ, EDGARDO

BARCA, FEDERICO

MARINO, ADRIAN

VIEHHOFER, SEBASTIAN

ZABALLA, JOSE LUIS

[illegible]

[illegible]

[illegible]

Sebastian Viehhofer

[illegible]

"I hereby certify that this report and the accompanying presentation is my own original work in its entirety, unless indicated and referenced."

Executive Summary

Olin Group, a Spanish startup, born in 2011 with the main purpose of consolidating the Telco market in Spain identified after their first acquisitions a pressing need for data cleaning and standardization during their several acquisition processes. This requirement arose as a critical means to enhance operations and customer relationship management. In response, our team developed a solution integrating algorithms, and public APIs.

Applying the Cross-Industry Standard Process for Data Mining (CRISP-DM) methodology helped fashion a solution to standardize and verify data. This approach steered the team through understanding the data and devising an exhaustive cleaning plan. The data was characterized by a lack of standardization, frequent typographical errors, and missing details, which necessitated considerable correction. The team disassembled addresses into their respective components, rectified discrepancies using regular expressions, and employed Python's Pandas libraries for data cleaning and manipulation.

For the data modeling phase of CRISP-DM, Fuzzy matching algorithms were explored. These algorithms function by approximating strings and calculating the operations required to make them identical. The best-known Fuzzy Algorithms are: Levenshtein, Jaro-Winkler, and Hamming Distance. The Levenshtein algorithm, due to its capability to manage variations in address strings and compare longer strings, proved most effective.

However, the algorithms alone did not deliver acceptable results, prompting us to incorporate several public APIs. These included Open Street Maps, TomTom, Google Address Validation, Google Places, and Google Geocoding API for matching and validating our addresses. While OSM and TomTom, being free or less expensive, were our initial choices, Google Maps APIs ultimately outperformed them, delivering better results (over 70% accuracy).

Implementation, the final phase of the CRISP-DM model, involved building an authenticated web app using Python and Streamlit. This application enables Olin employees to upload a CSV file with raw data, which our solution then cleans and validates using Python libraries and Google's Geocoding and Places APIs. Additional features were added to the app, such as a form for validating new customer addresses and a geolocation map for pinning the provided addresses.

Significant improvements notwithstanding, there is a clear roadmap for future enhancements. These include developing an API for integrating our service with other applications, transitioning to a microservice architecture for improved maintainability, synchronizing our app with Olin's internal authentication, and adding a fourth tab on the web app to provide client based KPIs. These KPIs could identify business opportunities by filtering clients by region, socioeconomic power, home size, etc.

In conclusion, our solution, built upon the CRISP-DM framework, not only resolved Olin's address validation challenge but also laid a foundation for future enhancements. A tool has been effectively created for Olin to address not only their current issue of registering new clients from acquired businesses but also to tackle similar challenges with future acquisitions.

TABLE OF CONTENTS

TABLE OF FIGURES

1. Introduction

OLIN GROUP, based in Madrid, are experts in telecommunications services. The management team consists of professionals in the telecommunication market[1]. Led and founded by infrastructure firm Asterion Industrial Partner, the company's primary challenge is to enhance connectivity not only in cities but also in towns and villages across the country. Their goal is to deliver a "quality service, close, and differential" to their customers. They provide high-speed Olinfra network services to wholesalers and manage local operators for retail markets.

According to Olin, the Spanish telecommunication market might be the most competitive and most complex in the world. Four big players control almost 95% of the market share, while the last 5%, approximately 1 million customers, is shared between 500+ small & regional operators. The company is focused on supporting these wholesalers and retailers in regional markets in Spain, aiming to become the "[…] leading regional telecommunication platform in the Spanish market […]"[2]The way Olin Group is trying to accomplish this task, is by buying several smaller and regional companies. So far, they have acquired nine companies, and transfer these companies to a single integrated operation with a common operational & systems model.[3]

The acquisition of existing regional telecommunications providers brings great opportunities, but also great challenges. The integration of newly acquired data, both client data and infrastructure data, is one of the biggest challenges for Olin. The data they are receiving is mostly local data from the providers. The data has been collected by operators or local stores, where addresses are assumingly written by hand in the system. To determine where most customers are located, an overview via geolocation is useful. However, the data Olin receives is not always homogenous and readable data. This is due to so called "as is company systems", which have different standards, codifications, and qualities[4]. Different fields, sometimes even just one field, are used. Open text vs fixed text fields are another common issue the Olin group faces.

The task in this project is to deal with this "difficult" data and to convert addresses to correctly written and computer-readable addresses to be able to connect with a third-party service and get the real address and geolocation. As an input in the best case, a one field address such as "Avd. Lazarejo, 17, Portal 2, 3ª, 28232, Las Rozasde Madrid, Madrid" is being received. Other addresses received have the following outline: "ApartamentoEN EL PATIO ANDALUZCLUB ALDEAS DE TARAY3782º;MANGA DEL MAR MENOR, LA;Murcia" and are more challenging. As an output we are required to present the address divided to the fields Country: (España), Province (Madrid), City (Las Rozas), Neighborhood, Street Type (Avenida), Street Name (Lazarejo), Street Number (17), Additional Information (Portal 2) and Zip Code (28232).

Another requirement is to receive the Geolocation for these addresses and output them as well. The output should be an excel file with the "clean" addresses and the geolocation. The data is mostly limited to the provinces of Murcia, Alicante, and Galicia.

[1] (Olingroup, 2023)
[2] Loc. Cit.
[3] Loc. Cit.
[4] Loc. Cit.

While working on this project, complex challenges that consist of data reliability and the indifferent Spanish address system with several languages, such as Castellano, Galician, and Valencian, are anticipated.

The Cross-Industry Standard Process for Data Mining (CRISP-DM) methodology will be used to standardize and verify data. In the data modeling phase, a Fuzzy algorithm is useful. Several algorithms such as the Levenshtein, Jaro-Winkler, and Hamming Distance will be used to improve the algorithm readability for further use. To create the desired CSV file, implementing APIs, such as the Open Street Maps, TomTom, Google Address Validation, Google Places, and Google Geocoding API for matching and validating the addresses will create great value.

The first steps into solving this problem are data understanding and preparation, therefore they will be first detailed during the report. All the methods, algorithms and APIs tried during the project, as well as potential future implementations, can be found afterwards in the penultimate section before the conclusion. This structure ensures a logical flow and comprehensive understanding of the project.

2. Data Understanding

To understand the data, the first step is to perform an exploratory analysis on the dataset given. The initial data came as an excel file containing a total of 6,946 rows, each representing an address. These addresses contained three attributes: street name and street number, neighborhood, and province. Below is an example:

"Calle SAN MARTIN DE PORRES 10;VISTA ALEGRE;Murcia"

As observed, this address contains street name and number, neighborhood, and province separated by semicolons. If necessary, some had further information such as apartment number, usually after the street number, such as:

"ALCOY 22 1B;Xixona;Alicante"

Have in mind that the only unique part is the street name and number, while the neighborhood and provinces are largely not unique. For example, for the provinces (rightmost part in the address), there were a total of 12 unique values, as seen in Table 1.

Province	Quantity	Percent
Murcia	5723	82,4%
Alicante	956	13,8%
Valencia	148	2,1%
Pontevedra	80	1,2%
Madrid	28	0,4%
Others (7)	11	0,2%
TOTAL	**6946**	**100%**

Table 1. Distribution of province values within data obtained.

Note that 82% of the addresses are from Murcia followed by 14% from Alicante. Both provinces encompass 96% of the data.

In the case of neighborhoods, the attribute has a total of 140 unique variables. What stands out the most is that 25% of them come from 'La Manga del Mar Menor', a small touristic 21 km strip in Murcia. Interestingly enough, this will cause problems in the future given that most addresses within this neighborhood do not have a street number and are mostly written with the name of the apartment (instead of the street name). Additionally, various addresses with "Muxia" which's province is wrongly labelled as 'Alicante', were written in Galician. For example, "Rua" was written instead of "Calle".

Shifting the focus to the formatting of the whole address, it is important to observe that for obvious reasons not all addresses are formatted the same. Based on the observations, addresses can be broadly categorized into three: good (no typos), medium (existing typos), and unfindable (regardless of any cleaning methods, the address cannot be found). Below is an example of good addresses:

"Calle LOS INGLESES 66; LA PALMA;Murcia"

The medium-level addresses exhibited a very diverse range of typos that made it unreasonable to quantify their frequency and very difficult to establish specific patterns. Here are three different examples:

"BOTALÓN13;APARECIDA, LA:Murcia"

"PlazaREYES CATÓLICOS14;VISTA ALEGRE;Murcia"

"C/ CUATRO VIENTOS 7 PBJ;PUEBLA, LA;Murcia"

As for the unfindable addresses, they are essentially unfixable. Below is an example:

"ES EN EL PANTALAN PRINCIPAL – BARCO KUKETESCLUB NAUTICO LA ISLETA – AMARRE 174 1;MANGA DEL MAR MENOR, LA;Murcia"

As mentioned earlier, note that these typos encompass a wide range of discrepancies, making it challenging to precisely quantify the frequency of each specific one or even group them. As a result, due to the lack of a systematic pattern, it is unreasonable to provide any precise statistical measurements regarding the frequency distribution of individual address typos. Therefore, while acknowledging the presence of different address typos within the dataset, the focus is on implementing a solution that can effectively correct the address typos. Once the corrected addresses are obtained, the next step will be to quantify the accuracy of the solution.

The aforementioned analysis provides valuable insights into the structure, distribution, and typographical variations present in the dataset. This information lays the foundation for further investigation and the development of an accurate address correction solution.

3. Data Preparation

Following the thorough analysis of the dataset and the identification of structural and typos in the addresses, the next step in the project is the data preparation. This phase encompasses two vital processes: data normalization and data cleaning, aiming to enhance the quality, consistency, and usability of the dataset. For both processes, there was extensive and exclusive use of the following libraries: NumPy, Pandas, and Re.

3.1 Data Normalization

The main part of data normalization consists in changing some basic but repetitive typos available in the dataset. As the name suggests, the purpose of normalization is to standardize the data, which is going to be very helpful in future data usage. In the dataset, the normalization mainly consisted in changing accents and common typos within the street type. For example, numerous addresses had the street type written as "C/" or "calle" instead of "Calle". The comprehensive list of replacements made can be found in the appendix (Table A1). Afterwards, a for loop (Figure A1) was applied with the dictionary to obtain the resulting, cleaner csv file with columns named "addresses", "neighborhood", "province".

3.2 Data Cleaning

After performing data normalization to standardize and rectify basic typos within the dataset, the next crucial step in the data preparation process is data cleaning. Data cleaning aims to address inconsistencies, errors, and irregularities in the dataset that are less common. Given that the **addresses** column contains a significant amount of varied information, including street names, street numbers, floor information, and additional details, the first part of the data cleaning process involves extracting relevant data from this column and creating new columns to capture this information (3.2.1). The extracted columns include **type_of_street**, **street_number**, **floor**, and **street_name**. Following the extraction of relevant data into new columns, the second part of the data cleaning process revolves around correcting typos within these newly created columns (3.2.2). This step aims to achieve a more uniform and standardized representation of the data.

3.2.1 Extracting data and splitting columns

As mentioned earlier, the addresses column contains lots of relevant information, therefore, extracting and organizing this data into separate columns allows for a more structured analysis. To achieve this, a series of functions will be applied to the addresses column, resulting in the creation of the new columns. Specifically, the new columns are **type_of_street**, **street_number**, **floor**, and **street_name** (3.1.1 to 3.1.4). Keep in mind that after the creation of the functions, they were all applied to the address's column via the built-in .apply() method, which's output was saved in the new column.

3.2.1.1 type_of_street:

The **type_of_street** column provides information regarding the type of street based on the prefixes identified in the address. The extraction of this information involved the utilization of two functions: **type_street()** and **remove_prefix(add).**

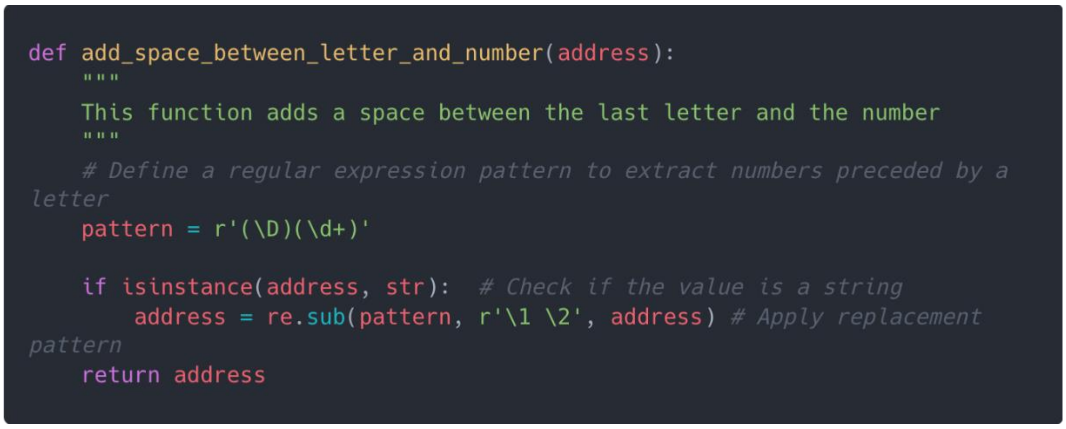
a. The **type_street()** function plays a crucial role in determining whether the address begins with any of the predefined prefixes and is responsible for extracting the street type (e.g., "CALLE", "AVENIDA") from the addresses by comparing the address against a list of known prefixes such as "CALLE" the function identifies a match. If a match is found, the corresponding prefix is returned. If no match is detected, an empty string is returned, indicating the absence of a recognizable street type.
b. **remove_prefix()** is responsible for eliminating prefixes from the address, ensuring a cleaner representation of the street name. By removing prefixes such as "C/" or "AVDA" the function enhances accuracy.

3.2.1.2 street_number:

It is important to note that to obtain the street number (and almost all other attributes within the addresses column), many addresses contained the data without spaces, which increases the difficulty to extract. Therefore, the solution came by using regular expressions, which are powerful tools used in pattern matching and data processing. They consist of a sequence of characters that define a search pattern, allowing for the identification and extraction of specific textual patterns within a larger body of text[5]. For example, function **add_space_between_letter_and_number(address)** adds a space between the last letter and the number found in the address. In this case, the regular expression is *pattern = r'(\D)(\d+)'*, which means:

- *(\D)* represents another capturing group that matches and captures a single non-digit character (in this case, the last letter of the address).
- *(\d+)* represents a capturing group that matches one or more consecutive digits (the number).

If the pattern exists (that is, the last letter of the address is followed by a number without a space) the **re.sub()** function is used to apply the replacement pattern *r'\1 \2'*, which adds a space between the first captured group (*\1*, the last letter) and the second captured group (*\2*, the number). The function **adds_space_between_letter_and_number(address)** looks like Figure 1.



```python
def add_space_between_letter_and_number(address):
    """
    This function adds a space between the last letter and the number
    """
    # Define a regular expression pattern to extract numbers preceded by a
letter
    pattern = r'(\D)(\d+)'

    if isinstance(address, str):  # Check if the value is a string
        address = re.sub(pattern, r'\1 \2', address) # Apply replacement
pattern
    return address
```

Figure 1. Code snippet showcasing the use of regular expression to add space between letter and number in address.

[5] (Adegeo, 2022)

3.2.1.3 floor:

This column was created to incorporate details about the floor in the address. The transformations applied to obtain the street number were helpful to obtain this column. Basically, only one function was applied:

- **extract_address_components(address)**: function employed a regular expression pattern to extract the street number and floor from the address.

3.2.1.4 street_name:

To establish this column only one function, **extract_street_name(address),** was passed to the remaining of the addresses' column. This successfully extracted the street name and created a new column named **street_name**.

3.2.1.5 Final DataFrame:

After applying all transformations and functions to the addresses, several new columns were created. These include **type_of_street**, **street_number**, **floor**, and **street_name**. As a result, the final clean DataFrame looks like this:

type_of_street	addresses	neighborhood	province
calle	los sanchez 7	pozo estrecho	murcia
calle	mayor 321 b	canteras	murcia
calle	aldeas de tarayapto 128	la manga del mar menor	murcia
avenida	calle cibeles 852	los narejos	murcia
calle	san nicolas 251 nda	corvera	murcia
	pintor bianqui 4	la palma	murcia
calle	ugrcollado jeronimos 9 a 3	gea y truyols	murcia

street_number	floor	street_name
7		los sanchez
321	b	mayor
128		aldeas de tarayapto
852		calle cibeles
251	nda	san nicolas
4		pintor bianqui
9	a 3	ugrcollado jeronimos

Table 2. Final DataFrame obtained after splitting the columns (first part).

3.2.2 Correcting typos within columns

After the demanding process of splitting the columns, more work needs to be done to correct typos within each of the columns created. Various techniques were employed to address these typos, including string rearrangement, filling missing values, ensuring consistent spacing between characters, and more. As a result, the following columns were enhanced through the application of these specific functions:

A. street_name:

- **remove_numbers()** function removes any remaining numbers from the address column.
- **remove_nnd()** and **remove_nd()** removes the 'nnd' and 'nd' from the address.
- **remove_parentheses()** removes any parenthesis and content inside of it.

B. street_number:

- **check_st_num(st_num)** validates and corrects street numbers within the 'street_number' column. Specifically, it checks that street numbers are not repeated (eg., 2121 to 21).

C. neighborhood:

- **rearrange_name()** designed to rearrange the neighborhood names within the 'neighborhood' column. It aims to standardize the format of neighborhood by moving some prefixes that were at the end to the front.

3.2.3 Final data preparation

After the cleaning process is complete, a new column called **full** is established that collects all the relevant information recently extracted from the columns, and which will be used to apply the various techniques and algorithms tested.

As seen later, to achieve the greatest results when using third-party APIs, a combination of the various cleansed values is required. For this **full** column the finished string ready to be sent to the API looks as such:

'type_of_street street_name, street_number, province, country'

4 Methods and Algorithms

4.1 Algorithms

There are several algorithms that can be used for matching the addresses with a complete database (string similarity and approximation). However, Fuzzy Algorithms are most reliable for this task[6]. Fuzzy matching algorithms approximate strings by comparing them and counting how many operations will be needed for both strings to be equal.

The more relevant ones are:

- **Levenshtein Algorithm**[7]: Measures the minimum number of character edits to transform one string to another. This includes insertions, deletions, and substitutions. For example: Corn vs Cork will count as 1.
- **Jaro-Winker Algorithm**[8]: Measures the edit distance between two sequences by considering the number of matching characters and transpositions needed. It is useful for comparing similar strings, especially one-word strings.
- **Haming Distance Algorithm**: Measures the number of positions at which two strings of equal length differ. Because the length of the strings needs to be similar, it will not run properly for this exercise.

Although the three algorithms are used for matching two strings, the Levenshtein algorithm was found to be the best suited for the task because of several reasons. First, it is capable of handling minor variations in the address strings such as typos or transpositions (common issues in data cleaning). It provides a quantitative measure of similarity by calculating the minimum number of edits required for the transformation. Finally, it is simple and easy to implement, making it the practical choice considering the lack of resources (computing power) available.

4.2 APIs

APIs (Application Programming Interfaces) are defined by Oxford as: "*a set of programming tools that enables a program to communicate with another program or an operating system and helps software developers create their own applications*".[9] It basically allows different software applications to communicate and interact with each other.

During this project explored several public APIs (free and paid) were explored:

- **Open Street Maps[10] API:** (OSM) is an API that provides access to OSM database, an open source and collaborative mapping project. Through this API it was possible to send cleaner address queries and retrieve the full address and geo location such as latitude and longitude. This API is mainly known for being open source and having global coverage. However, because it is free and open source, the results of the API for address validation may not be ideal or as efficient as others.

[6] (Silva, 2022)
[7] Loc. Cit.
[8] (Kulkarni, 2021)
[9] (Oxford, 2023)
[10] (OpenStreetMap, 2023)

- **TomTom API[11]:** Offers geocoding and address validation functionalities like OSM. TomTom API matches the queries with its geospatial database. It is cheaper than Googles APIs (USD 0.5 per 1,000 queries) and addresses retrieved include latitude and longitude.

- **Google Address Validation API[12]:** Part of Google Maps API suite (released in November 2022). Allows users to validate and standardize addresses by comparing them against the Google Maps database. The API also verifies the accuracy of the address by field, grading each part individually and confirming if the answer is valid, probably valid, or missing. It also gives back latitude and longitude and can detect if the building is residential or commercial. However, it is the most expensive API at USD 17 per 1,000 queries.

- **Google Places API[13]:** Also, part of Google Maps API suite. Provides access to Googles database also including landmarks, businesses, points of interest and addresses. Provides access to Google's database also including landmarks, businesses, points of interest, and addresses. It validates address information by retrieving additional details about the specific location such as business names, types, ratings, reviews, etc. However, it does not provide latitude and longitude coordinates. It costs USD 2 per 1,000 and is the cheapest API from Google Maps API suite tried.

- **Google Geocoding API[14]:** Last Google Maps API tried. It is mainly used to convert addresses into geographic coordinates (latitude and longitude). However, it will not be as effective when trying to correct the query as it is not its main purpose. It is also based on pay as you go and charges USD 5 per 1,000 queries.

[11] (TomTom, 2023)
[12] (Validation, 2023)
[13] (Places, 2023)
[14] (Geocoding, 2023)

5 Results and Discussion

The results of the API outcome are saved in a CSV file with the division of the desired fields and the geolocation. The Levenshtein algorithm gave us a CSV without geolocation. The results were evaluated by checking the accuracy and developed the best model out of it. Regarding the APIs, four different address validation methods were taken into consideration: the Open Street Maps API, the TomTom API, the Google Address Validation API, and a combination of Google Places API with Google Geocoding API. The results, advantages and disadvantages of each model are displayed in table 3.

Method	Results	Advantages	Disadvantages
Levenshtein	50%	• No need to use an API • Reduced costs	• Not great accuracy • Not able to geolocate addresses • Need a very big dataset • Needs great computing power
Open Street Maps API	45%	• Free API • Complete data + Geolocation service	• Low accuracy • Slow API • Misses many street numbers
TomTom API	55%	• Cheap API ($0.50 per 1000) • 2500 free calls per day	• Low accuracy • Slow API
Google Address Validation API	60%	• Complete data in only one API (Geolocation + Address) • Confidence interval	• High costs, $17 per 1000 calls • Not great accuracy
Google Places API + Google Geocoding API	76%	• Great accuracy • Low cost ($2 places + $5 Geocoding per 1000 calls) • Detects places (building names as "Urbanizaciones")	• Need to combine two APIs to get all the data (Addresses + Geolocation) • Not free • Directly depending on Google to maintain the prices (previously happened that prices escalated)

Table 3. Methods comparison with results, advantages, and disadvantages.

The results showed that the Levenshtein method had a 50% accuracy rate. Not only was the accuracy low but it was also unable to geolocate addresses, making it unsuitable. Using geolocation was an essential part of the task, this is why a focus on the already mentioned APIs is logical. Furthermore, Open Street Maps was disappointing as well, as it gave an accuracy of only 45% and was slow. It also did not give many street numbers. Nonetheless, it was used first because it is free. Moving on, the TomTom API had just a 5% increase in accuracy. In conclusion, the best results were obtained with the Google Address Validation APIs and the combination of the Google Places and Geocoding APIs. The highest value received, was using the API combination which yielded 76%, excluding the unfindable addresses. Not only did it give the best result, it also was cheaper than using the Address Validation API.

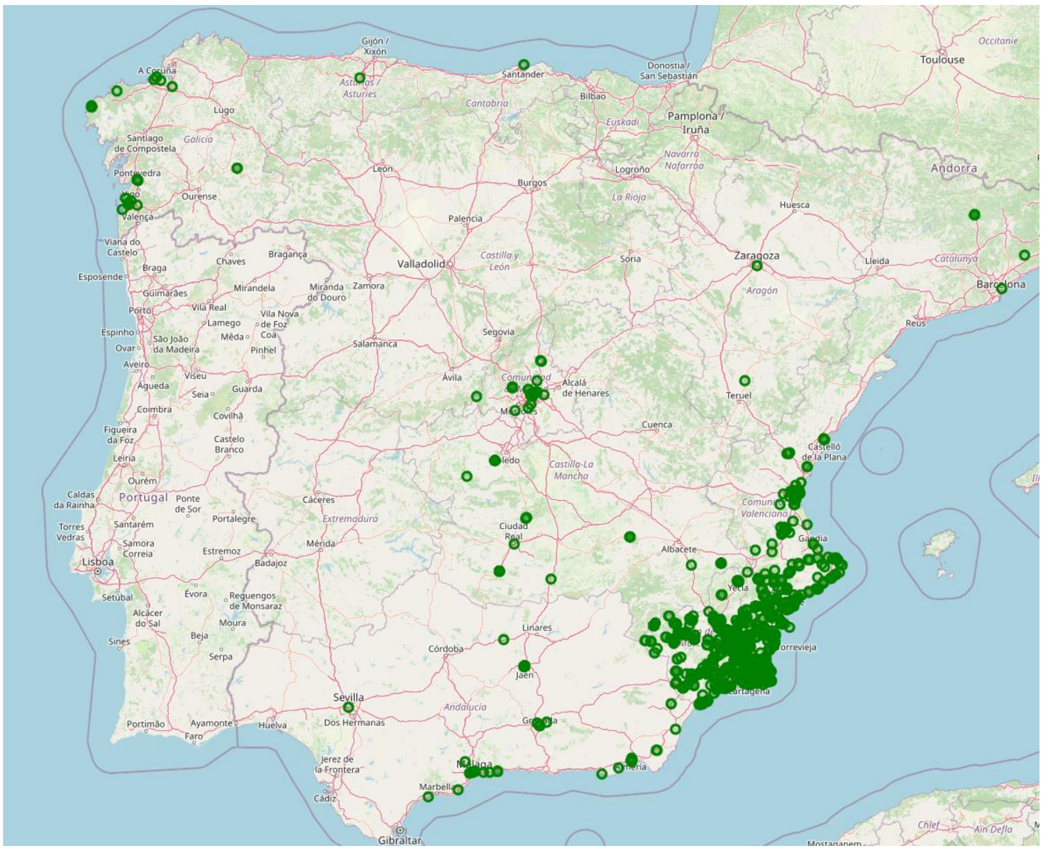


Figure 2. Display of the obtained final geolocations

In Figure 2, the results are displayed on a map of the Spanish mainland. As described in the data description above, most of the addresses are in the region of Alicante and Murcia. It shows the distribution of the customers of the newly acquired companies, which gives Olin an overview of the location distribution.

But why did some methods perform better than others? None of the methods gave an exceptional accuracy, which might reflect the complexity of the data received even after extensive cleaning. As discussed earlier, factors such as different standards, codifications, languages, and quality of the "as is company systems" data pose considerable challenges. But according to the observations, Google's combined Places and Geocoding APIs performed best. Googles databases are likely more extensive using the Google Maps database and frequently updated compared to other sources, which could explain the improved accuracy when using their APIs[15]. Also, combining both APIs allowed for more comprehensive data gathering, improving the chances of accurate geolocation and address validation.

[15] Alphabet Inc., 2023)

In the end the Google Places API + Google Geocoding API were used to generate the final CSV file with the outcome and geolocations. Receiving the best outcome was the challenge of the assignment. For future implementations cost, accuracy, and speed need to be considered to hit the project goals. How the implementation of future projects could look like is explained in the next sector, the implementation.

6 Implementation & Deployment

Once the stages of data understanding, data preparation, data cleaning, and modeling were done, the implementation was executed. For Olin Group, the relevance of this project lays in having a file with the correct addresses and the geolocation. That is why it was decided to create a web app to enable Olin users to interact in an easy way with the model. With this web app, the users can easily upload a file with not well-written addresses and retrieve a file with the correct addresses. On the other hand, Olin group employees can also validate new addresses, while filling in the form and then validating it through an API. Finally, they can search for customer addresses on an interactive map. In this section of the document, these features of the web app and the process of the deployment will be described.

For the project, an authenticated web app was created to facilitate Olin Group the interaction with the cleaning model. To create this web app, a code in Python was built using Streamlit, and the cleaning model, APIs, and the rest of resources used for the data process were incorporated. The webapp is divided into 3 sections, and for each section the user can interact in a different way and perform different activities.

On the first tab, the app gives the Olin employees the option to upload a CSV file. Once uploaded, the code, described on the "Data Preparation" and "Models and Algorithms" sections of this document, is used to prepare the data, clean it, call the APIs, and save the results. Afterwards, the cleaned data is shown on the web, and the user has the option to download the information as a CSV file, using the base64 Python library.

On the second tab, extra features for Olin users were developed, in case they need them in the future. In this tab, a form is displayed where Olin employees can write the address information of a new customer. The benefit of this form is that once the form is filled out, the user can click on the "validate" button, so the code that is connected to the APIs verifies the address. Once the code gets the result, it is shown to the user, so the user can validate if that is the correct address. Having this integration, Olin can ensure that all their new addresses are identified with geolocation and full well-written information, before storing them.

On the third tab, a map is shown. Whenever the employee gives a pair of coordinates, a pin shows that address. This is meaningful because in the future, Olin can complement the address information with the rest of their confidential customers' information. Doing this, in the future, whenever they want to search the geolocation of a specific customer ID, a set of customers of a specific region, or any other relevant information, the map will show the different pins to easily identify the geolocation of the desired data. For this Streamlit tab, the Python folium library was used to build the features.

Everything that has been explained so far is about the Python code, however, in order to make the whole application work, other files were needed too, including YAML files, pictures, database, Dockerfile and requirements files, and uploading all these in a Github repository. Cloud Native principles were used to deploy the application, taking advantage of the benefits of working with Dockerfiles, images and containers to build the web app.

First, a Dockerfile was created. For this case, the official Python base image was used, and the requirements file was stated, where all the libraries needed are described (not only to run Streamlit but also to run the data preparation, data cleaning, calling the APIs, the libraries for the maps, etc.). Finally, the web app used port 8080 to be exposed. Once the Dockerfile was ready, everything was updated in the Github repository.

Furthermore, for the deployment of the application on the internet, it was decided to use Google Cloud and specifically the Cloud Run service. To deploy it, the Github repository with all its files was replicated into the editor in the Google Cloud Console. Next, a docker image was created with its proper tag, and the Artifact Registry API was enabled, so the interaction with this service to upload Docker images and create containers based on those images was possible. Once Artifact Registry was enabled, a repository was created, and the docker image was pushed into it. Finally, the container was deployed in Cloud Run using the Docker image. The idea of following this process is to benefit from the CI/CD best practices, and even, in the future, use services such as Cloud Build to facilitate future app improvements and deployments.

Cloud Run was used in this project because it gives the option to scale when needed. Imagine a scenario where Olin Group wants to keep using the form to register new addresses, and hundreds of persons start working with this app. If Cloud services were not used, Olin Group would have 2 options, the first one is to buy servers to serve the average traffic and assume that when a pick of requests arises, the servers would not be able to serve the traffic, so the app will be inaccessible at that time. The second option is that Olin ensures it has enough servers, even for the request picks, but on average the servers would be underused. That is why Deploying the app in Cloud Run enables to have the servers needed according to the number requests the app is receiving.

Nevertheless, Cloud Run is not valuable only because of the technical benefits, but also because of business benefits. There are some other services in Google Cloud that would give the same scalability and flexibility as Cloud Run, such as App Engine Flexible. However, Cloud Run is the cheapest option for this implementation, allowing the app to serve for free 180.000 seconds per month with 1 million calls.

Finally, some of the recommendations for next steps in the implementation of this project are:

- Creating an API: The project was built where a user can interact with a front end. However, Olin might need to incorporate this service of cleaning the data in other applications and the best way to do this is to create an Application Programing Interface (API). Olin can connect the rest of their services with the API.
- Creating microservices: The whole front end was created in one single service, having only one container. This could be improved by dividing the different services into different containers. In that scenario, there should be one container for the login, another one for the “Cleaning Data” tab, another for the “New Address” tab, and one for the “Navigation” tab. Doing this will improve the web app in terms of maintainability, modularity, fault isolation, and monitoring.
- Olin’s Authentication: Another step to be considered in the implementation of the project inside Olin’s operations, is to connect the authentication of the web app with Olin’s Intranet. Right now, the app is using an external authenticator based on Streamlit. However, for Olin Group it would be better to control, maintain and access the webapp using the same access management used for their intranet.

The initial architecture proposed for the next steps recommendations is the following:

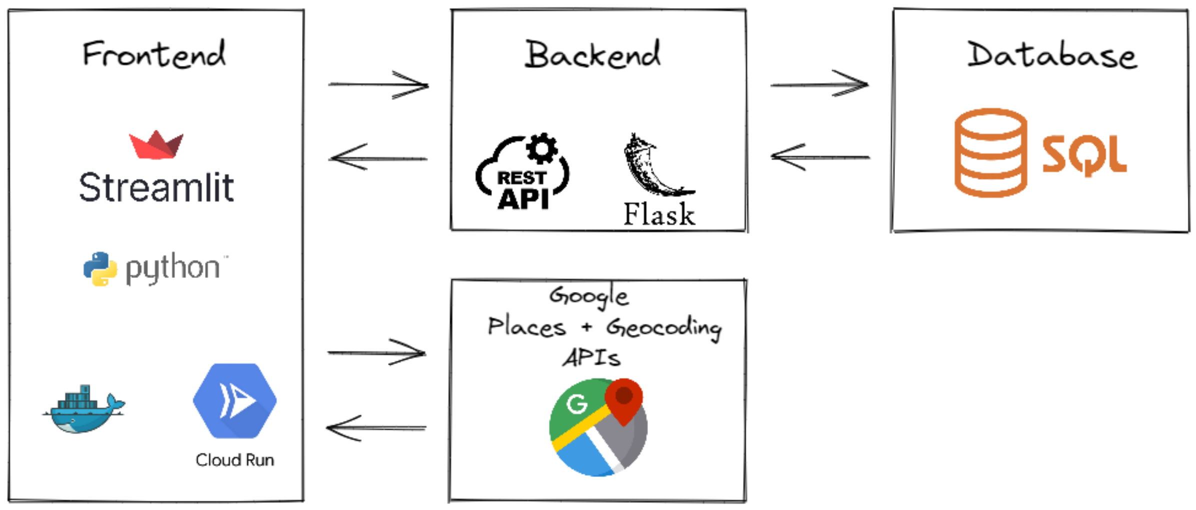


Figure 3. Proposed future architecture.

There are several improvements that can be made for the implementation of the project inside Olin's operations. In order to do this in the best way, we should work closely with the company to better understand their processes and the needs of the business. The idea of fully integrating the model into Olin's operations would give us the opportunity to fully understand the mission of the company and build additional features and processes for them to continue bringing value to their customers and their business.

7 Conclusion

In order to complete the assignment, which was to create an address validation algorithm as well as displaying the geolocation of provided addresses, several steps were needed to get to the optimal result. Data understanding, preparing, and cleaning the data, develop several models and their evaluation were some of the steps needed. The complex address data made the task very challenging, but after an evaluation of the different methods, the determination of the best solution was method for the task: an extensive data cleaning followed by a combined use of Google Places API and Google Geocoding API. With this method an accuracy of 76% of correct addresses was reached, even though it required using two APIs and is highly dependable on Google in forms of cost matter and reliability.

Following this approach, a user-friendly interface was developed for uploading, validating, and downloading address data for Olin. This solution is a steppingstone towards a long-term approach for Olin, bearing in mind that needs and technologies might change. Several promising possibilities for the future were identified, an API could be created to be integrated with the existing systems used by Olin Group. This could broaden the impact and utility of the address validation service. Refining the web application by splitting it into distinct microservices, each encapsulated in its own container, could be another improvement. This strategy can enhance maintainability, modularity, fault isolation, and monitoring, making the application more resilient and easier to manage in the long run. Finally, another upgrade worth exploring is the integration of the web app's authentication with Olin's Intranet. This step could provide a unified and secure access management system, while improving the overall user experience and security.

The project has been quite challenging and rewarding at the same time. It was challenging to find the right approach to come to the best solution, but there was great learning on the different models. Not only the competition of the challenge was achieved, but also set a first step to future improvements by not just addressing Olin`s current issue with one dataset, but also creating a solution to tackle similar issues in the future. A complete integration of the different subjects completed in the Master in Computer Science and Business Technology was implemented into one project, while creating a great learning experience by itself. The great organization of the diverse courses, as well as the great practical experience of the professors, helped to fulfill this task.

8 References

Adegeo. (2022). *Microsoft*. Retrieved from Microsoft: https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference

Alphabet Inc. (2023, June 30). *Google For Developers*. Retrieved from Use Places APIs and Geocoding with data-driven styling for boundaries: https://developers.google.com/maps/documentation/javascript/dds-boundaries/dds-use-maps-places-apis

Geocoding, G. (2023, July 04). *Google Geocoding API*. Retrieved from https://developers.google.com/maps/documentation/geocoding/overview

Kulkarni, S. (2021, March 26). *Medium*. Retrieved from Medium: https://srinivas-kulkarni.medium.com/jaro-winkler-vs-levenshtein-distance-2eab21832fd6

Olingroup. (2023). *About us*. Retrieved from olingroup.es: https://olingroup.es/en/about-us/

Olingroup. (2023). *IE MCSBTChallenge 2023.*

OpenStreetMap. (2023, July 04). *Open Street Map*. Retrieved from https://wiki.openstreetmap.org/wiki/API

Oxford. (2023, July 04). *Oxford Learners Dictionaries*. Retrieved from https://www.oxfordlearnersdictionaries.com/definition/english/api

Places, G. (2023, July 04). *Google Places API*. Retrieved from https://developers.google.com/maps/documentation/places/web-service/overview

Silva, E. (2022, July 15). *Redis*. Retrieved from Redis: https://redis.com/blog/what-is-fuzzy-matching/

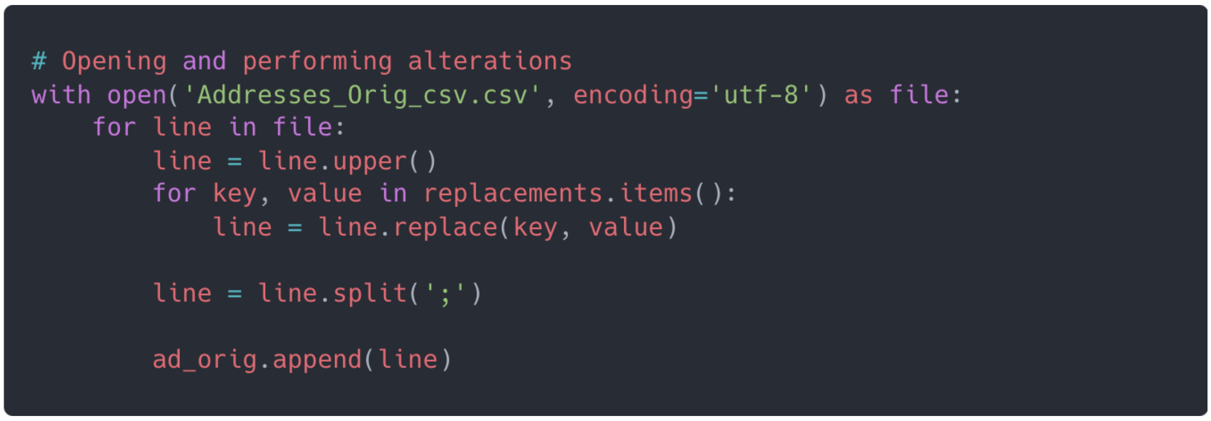
TomTom. (2023, July 04). *TomTom*. Retrieved from https://developer.tomtom.com/map-display-api/documentation/product-information/introduction

Validation, G. A. (2023, July 04). *Google Address Validation API*. Retrieved from https://developers.google.com/maps/documentation/address-validation/overview

9 Appendix

Province	Quantity
Á	A
É	E
Í	I
Ó	O
Ú	U
Ñ	N
Ü	U
C/	CALLE
C /	CALLE
AVDA	AVENIDA
CTRA	CARRETERA
URB.	URBANIZACION
URB	URBANIZACION
URB.	URBANIZACION
AVENIDAAVENIDA	AVENIDA
PLAZAPLAZA	PLAZA
"	
º	ND
\n	

Table A1. Replacements applied to normalize dataset



```python
# Opening and performing alterations
with open('Addresses_Orig_csv.csv', encoding='utf-8') as file:
    for line in file:
        line = line.upper()
        for key, value in replacements.items():
            line = line.replace(key, value)

        line = line.split(';')

        ad_orig.append(line)
```

Figure A1. Code snippet on replacements dictionary to create first clean csv file.