1 INTRODUCTION

This project contains some backgrounds which are not part of computer science. Here introduces adequate information in order to help getting start.

1.1 MOTIVATION

The first civil registration in Ireland was performed on 1864 [12]. Before that time census meterials were mostly lost or incomplete. So genealogical researches need to rely on parish records and also some 'census substitute' documents, such as land ownership and tenancy records[1].

However, for these documents, each of them usually does not contain enough information to indentify individuals. Some of them contains name and address, whereas others might contain only name. In order to fulfil missing information of one individual that scattered among many documents, *Record linkage* is one method to do so.

"Record linkage is used in historical research, social studies, marketing, administration and government as well as in genealogy" – Winstanley [13] section 2.2

Record linkage uses a person's name as a basis to link that person's information between many documents. Together with other coherant attributes to ensure the link is correct, a more complete information about that person can be obtained.

In addition, this is not only just for one person. We can assume the relationship of the person to others that might be close to, and apply the information to those people as well. For example, if we know that there is a record that is believed to consist of people from the same area in each page [14] (but no area or address is mentioned, or some is missing in the page). And we can find one or more person's addresses in that page by using record linkage. We might be able to apply those addresses to all people in that page as well.

Apparently linking or matching person's name is important in the process. Unfortunately, in the 19$^{\text{th}}$ century, in Ireland, there was no standard of the spelling of names, handwriting could be difficult to read and contractions or abbreviations were often used. Many people were not literate, so they asked literate people to write their names. This way even names with the same pronounciation and for the same individual could be written in many different ways, depending on who wrote them.

1 [13] section 1.1

In addition to the various ways of spelling one's name, people from this time also often use Irish names which equivalent to modern names, for example, Irish version of 'Smith' could be 'Gowan'. There are also some Irish prefixes like 'O'', 'M'', 'Mac', etc. When combined together this would result in 'O'Gowan' or 'M'Gowen', and so on.

An example list of possible equivalent Irish names of 'Smith' could be as follow.

> Smith, Smyth, Smythe, Smeeth, Going, Gowing, Maizurn, McGhoon, MaGough, M'Ghoon, MacGivney, MacGivena, M'Givena, MacGhoon, M'Evinie, McGivney, MacEvinie, McGivena, M'Givney, McEvinie, MacAvinue, M'Avinue, McAvinue, McCona, MaGowen, MaGowan, MaGovern, MaGeown, McGowan, McGoween, McGown, M'Cona, MeCowan, MeGowan, MacGown, MacGoween, MacGowan, MacCona, M'Gowan, M'Gowen, M'Gown, Ogowan, O'Gowan, Gowen, Gowan, Gow, Goan

At present time, when historical researchers try to trace people back using historical records, they would encounter this problem of name variations.

Various solutions have been created to find matching different names that refer to the same person. However, for our extent knowledge, there is yet no public system which encodes those solutions together and provides a service of name matching. This project is to create one system to archieve this.

1.2 RESEARCH QUESTIONS

From the motivation, we address our research questions as follow.

1. Can we provide a web service to match names, where matching can be a complicated process because of the way people record their names.
2. Can the web service act as a platform system for general names or words matching system so that it can be extended to other languages as well.

The first question derives directly from the motivation. The second question is an enhancement for the system. It can be designed as a more general purpose matching system rather than just specified only for Irish names. Therefore it should be extensible for any further matching algorithms to be developed in the future.

In addition to the web service, web interface is to be introduced as well for the purpose of user friendly usage, individual usage, and demonstration.

1.3 OBJECTIVE AND AIMS

The objective of this project is to provide a web service that encodes several of matching algorithms and produces matching results between two lists of names.

The project aims to be a part of a bigger system, such as genealogy research. These client systems, at some point, they might need a service of a name matching on demand, so then they can use this web service, providing their lists of name, algorithms be be used, and threshold as inputs, and get matching results for their further usage.

We would start by focusing on Irish *surname* first. For any further kind of names we would leave it for future works.

1.4 REPORT STRUCTURE

This report is separated into four parts, The Background, The Solution, and Appendix.

THE BACKGROUND: Current part, states about background of this project. Introduces the initial problem, also some historical situations and terms which are not resident to computer science. Also related works that are involved in the project.

THE SOLUTION: The implementation to solve the problem. Details about algorithm, tools, language, frameworks, etc. which being used in the project.

THE FOREWARD: Evaluation of its performance, conclusion of the outcome of the project. encountered problems, and future works for extending and improvements.

APPENDIX: The 'user manual' of the project. Presents technical aspects, for example, how to use the web service in real world situation, or how to create an environment to host this project.

2 RELATED WORK

From research questions on section 2.2, there are three aforementioned terms that will be core research fields of this project. These fields are *name matching*, *web service*, and *extensible platform*.

2.1 NAME MATCHING

There are many methods for matching names. This project encodes various of them at the starting state.

2.1.1 *Edit distance*

> Edit distance is a way of quantifying how dissimilar two strings (e.g., words) are to one another by counting the minimum number of operations required to transform one string into the other. – Edit distance, Wikipedia [15]

An direct string operation way of comparing two string could work with name matching too. One of the edit distance variant, *Levenshtein distance* [16] is chosen to be implemented in this project.

2.1.2 *Soundex*

Soundex [17] encodes a name (or any string) into a 4 character code which represents an essence of its sound as pronounced in English. The idea is to encode letters with similar sound into the same group, and ignore vowels (unless it is the first letter). For example, 'Smith' is translated to `S530`, and 'Simon' is translated to `S550`.

Irish Soundex[1] is a modified version of Soundex, aims to improve capability of a traditional one upon Irish surnames. By applying rules accroding to the langauage characteristics and make some adjustment to distinguish names properly.

Both Soundex variants are also implemented in the project.

1 [13] Appendix 3.

2.1.3 *Lookup Table*

In 1901, Robert Edwin Matheson, the assistant registrar-general in Dublin, developed a name classification system [18] for an aid of register indexing and searching. He used a report on surnames in Ireland extracted from civil registers [19] in 1894 as a base of his system[2].

He gathered information from registry offices, focusing on people or members of close families. When these people made official register records with the office, they might use different variant of their surnames. For example, Mr. Green can be registered as dead by his son using the name Huneen.

With these information, Matheson classified the surnames in Ireland into 2091 groups. For example, group 753 consists of these names.

> Green, Greenan, greenaway, greene, grene, Guerin, Houneen, Huneen, MacAlasher, MacAlesher, MacGlashan, MacGlashin, MacIllesher, M'Alasher, M'Alesher, McAlasher, McAlesher, McGlashan, McGlashin, McIllesher, M'Glashan, M'Glashin, M'Illesher, Oonin.

This classification also includes multiple mapping between names. One name can belong to one or more group. For example, 'Green' belongs to groups 753, 754, 768, and 1350.

By using this classification information, we can construct a lookup table for Irish names by having names in the same group hold the same reference number.

2.2 WEB SERVICE

One convenient way to bring this service to public is to create a *web service*. A web service is a tool or function that can be accessed by other programs over the web (via http) [20]. A result from web service is designed to be used by computer programs rather than humans.

There are many ways to implement web services. Two famous ones are *Simple Object Access Protocol (SOAP)* and *Representational State Transfer (REST)*. Both has their own advantages [21]. We decided to implement our service using REST due to its simplicity and scalability [22][23].

At this initial state, data resulting from our web service is in `JSON` [24] format. Since it is widely used in web development and becoming more and more popular [25]. However, our service can be extended into any other format easily as well, such as traditional `XML`.

2 [13] section 2.3.

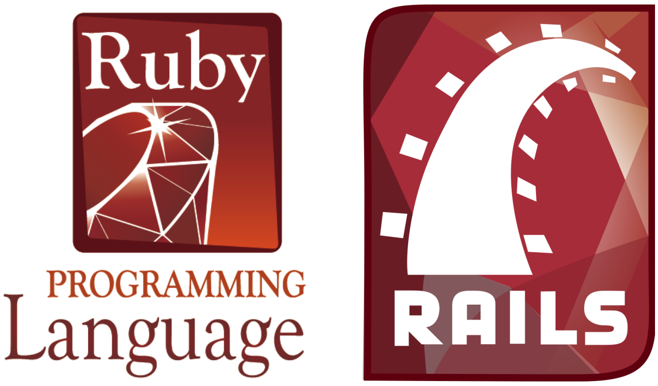


Figure 1: Ruby programming language (left)
and Ruby on Rails framework (right).

2.3 EXTENSIBLE FRAMEWORK

"Ruby is designed to make programmers happy."
– Venners [26]

Our system is implemented in *Ruby* [27] programming language. Ruby is a well-balanced language, it can be used as an traditional object-oriented language [28] and also capable of performing functional programming [29], thus making it very flexible and versatile.

The system sits on top of *Ruby on Rails* (or *Rails*, in short) [30] framework. Rails is a mature and stable framework that has been in web development for decades [31]. So it has a great support and a large community bebind. A great choice for building a sustainable system.

Rails is capable of both web service and web interface. By sharing the same algorithm we could provide a service for both programs (targeted by web service) and humans (targeted by web interface).

Part I

THE SOLUTION

3 NAME MATCHING ALGORITHMS

This chapter describes the details how the project is implemented. Note that algorithms and codes listed here are written in Ruby programming language, which is the main language of the project.

We will start off by detailing bundled matching algorithms here. Each matching algorithm calculates the *similarity score* between two strings.

The score is ranging between 0.0 to 1.0, where 0.0 means two strings are completely different and 1.0 means both are extactly matched.

Also note that string inputs here is in all in the uppercase format, in order to prevent letter-case difference.

3.1 LEVENSHTEIN DISTANCE

This algorithm measures the difference between two strings. It tells the minimum number of opearations needed to change string to another. These opeations are insertions, deletions, or substitutions. Consider these following examples.

- SM**I**TH → SM**Y**TH
  the minimum operation to change is 1, which is to substitute I to Y, therefore Levenshtein distance for these two strings is 1.
- GOWAN → **MC**GOWAN
  2 insertions of M and C is required.
- SM**I**TH**E** → SM**Y**TH
  1 deletion of e and 1 substitution of i to y are required.

The implementation used in the project is done by Battley [32][1]. Once the distance is calculated, it will be compared to the length of the longer string between the two (or if they are the same length, use that length).

For example, Levenshtein distance between 'SMYTH' and 'SMITHE' is 2, compare 2 to length of the longer string, 'SMITHE', which is 6. So the *similarity score* of these two strings are $6 - (2/6) = 0.667$.

The code of this algorithm is as in listing 2, note that `name` and `@base_name.name` are two strings to be matched.

1 levenshtein.rb

```
def cal_score
  @value = Text::Levenshtein.distance(@name, @base_name.name)
  size = [@name.size, @base_name.name.size].max
  @score = ((size - @value).to_f / size)
end
```

Listing 1: Levenshtein distance implementation.

3.2 SOUNDEX

Soundex encodes a string into a 4 character code representing an essence of its sound as pronounced in English. It operates in the following steps.

1. Take the first letter of a string.
2. Encode each remaining letters into a group following table 1. Discards A, E, I, O, U, H, W, and Y
3. Remove two adjacent same characters.
4. If a group of a first letter is the same as the second letter, remove the second letter.
5. Trim or pad with zeros as necessary, making the result 4 characters long.

GROUP	LETTERS
1	B, F, P, V
2	C, G, J, K, Q, S, X, Z
3	D, T
4	L
5	M, N
6	R
-	A, E, I, O, U, H, W, Y

Table 1: Soundex letter group.

Let us follow these steps by step, consider we are going to encode the string 'PFISTTER'.

1. Take first letter of 'PFISTTER'.
   PFISTTER $\rightarrow$ P

2. Encode remaining letter 'FISTTER'.
   PFISTTER → P1-233-6 → P12336
3. Remove two adjacent same characters.
   PFISTTER → P12336 → P1236
4. P is also in group 1, so remove the second 1 letter.
   PFISTTER → P1236 → P236
5. P236 is 4 characters long, so no need to be trimmed or padded.
   PFISTTER → P1236 → P236

Therefore, soundex of 'PFISTTER' is `P236`.

The implementation of soundex (listing 3) in this project is adapted from Winstanley's *Irish soundex* implementated in Visual Basic[2]. The code is commented following the same aforementioned steps.

```
def self.soundex(name)
  # Take the first letter of a string.
  result = name.first

  # Encode remaining letters.
  name[1..name.length].split('').each do |n|
    result = result + category(n).to_s
  end

  # Remove two adjacent same characters.
  result.gsub!(/([0-9])\1+/, '\1')

  # If category of 1st letter equals 2nd character, remove 2nd
      character.
  if result.size >= 2 && category(result[0]).to_s == result[1]
    result.slice!(1)
  end

  # Trim or pad with zeros as necessary.
  result = if result.size == 4
             result
           elsif result.size > 4
             result[0..3]
           else
             result.ljust(4, '0')
           end
end
```

Listing 2: Soundex implementation.

The `category` function implements soundex grouping table (table 1) as in listing 4.

2 [13] Appendix 3.

```
def self.category(c)
  if c.match(/[AEIOUHWY]/).present?
    ""
  elsif c.match(/[BPFV]/).present?
    1
  elsif c.match(/[CSKGJQXZ]/).present?
    2
  elsif c.match(/[DT]/).present?
    3
  elsif c.match(/[L]/).present?
    4
  elsif c.match(/[MN]/).present?
    5
  elsif c.match(/[R]/).present?
    6
  else
    ""
  end
end
```

Listing 3: Soundex grouping table implementation.

Now that we encode two strings to be matched in soundexes. We then calculate the *similarity score* of these two soundexes using these steps.

- Compare first characters of each soundex, if they are different, *similarity score* is 0, otherwise move to next step.
- Compare the rest 3 digits by using Levenshtein distance (section 4.1) to calculate the distance between them.

For example, *similarity score* between 'SMITH' and 'SPEED', which soundexes are `S530` and `S130` respectively, is 0.75 (1 substitution from *5* to *1*, so 1 difference of length 4).

The code of this soundex *similarity score* is as in listing 5.

```
def soundex_distance_score(s1, s2)
  if s1.first != s2.first
    0 # Different category, so they suppose to be completely
        different.
  else
    (s1.size - Text::Levenshtein.distance(s1, s2).to_f ) / s1.size
  end
end
```

Listing 4: Soundex similarity score implementation.

3.3 IRISH SOUNDEX

Irish soundex is another variant of traditional soundex. It determines characteristics of Irish names and normalised them to modern names. This algorithm also follows Winstanley's *Irish soundex*[3].

Irish names might contain some prefix, e.g. *Mc* or *O*, which are obstructive to soundex result. These prefixes are to be discarded. Moreover, there is no initial soft *C* in Irish names, instead *K* is used. So the first letter *C* is changed to *K*. It is implemented as in listing 6.

3 [13] Appendix 3.

```
def self.soundex(name)
 # Change initial ST. to SAINT.
 name = name.match(/^ST\./).present? ? "SAINT
     #{name[3..name.length]}" : name

 # Discard Irish prefixes.
 name = if name.match(/^O /).present?
        name[1..name.length].gsub(' ', '')
       elsif name.match(/^O'/).present?
        name[2..name.length].gsub(' ', '')
       elsif name.match(/^MC/).present?
        name[2..name.length].gsub(' ', '')
       elsif name.match(/^M'/).present?
        name[2..name.length].gsub(' ', '')
       elsif name.match(/^MAC/).present? && name != 'MAC'
        name[3..name.length].gsub(' ', '')
       else
        name
       end

 # Change initial C to K.
 name = name.strip.gsub(/^C/, 'K')

 # Call to traditional soundex.
 return {
   :label => name,
   :soundex => Soundex.soundex(name)
 }
end
```

Listing 5: Irish soundex implementation.

Irish soundex algorithm in this project calls traditional soundex described in section 4.2 to minimise repeated code. It also calculates *similarity score* the same way soundex does, as in listing 5.

3.4 LOOKUP TABLE

Lookup table is constructed from Robert Edwin Matheson's classification of Irish names. All classification information is stored in a Database, using PostgreSQL, which is a powerful, open source object-relational database system [33].

Matheson classified the surnames in Ireland into 2091 groups. Each group has one or more names, and on the other hand, each name belongs to one or more group.

In this section, we will consider the names as strings input. So the term *string* will be used in consistent with previous sections.

For example, considering the string *ACHESON*, this string belongs to two groups, 4 and 42. So *ACHESON* will have 2 records in the database. One with reference to group 4 and another with reference to group 42.

Next, let us consider group 4 and 42.

> 4 → ACHESON, ACHISON, AITCHISON, ATCHESON, ATCHIESON, ATCHISON, ATKINSON
> 42 → ACHESON, ARKESON, ATKINS, ATKINSON

By combining two groups together, these are all possible strings that match *ACHESON* according to Matheson's classification.

Now is the process to match two strings using Lookup table, suppose two strings are *ACHESON* and *ATKINS*.

1. Find references of *ACHESON*. We get references for group 4 and 42. Note the use of `pluck` method to select reference attribute (`ref`) here[4].

```
LookupTableRecord.where(:name => 'ACHESON').pluck(:ref)
=> [4, 42]
```

2. Find reference to matching string *ATKINS* and also specify the reference groups from step 1. If there is no match `where` method will return empty array. `present?` method is used to check the result if it is not empty[5].

```
LookupTableRecord.where(:ref => [4, 42], :name =>
    'ATKINS').present?
=> true
```

   By specifying both matching string and references group, we can ensure the matching name is also in the one of the same reference group of the base name. In this case, we conclude that there is a match between *ACHESON* and *ATKINS* via group 4 or 42 (or more specifically, 42, because *ATKINS* belongs to group 41 and 42).

4 Use pluck as a shortcut to select one or more attributes without loading a bunch of records just to grab the attributes you want. http://api.rubyonrails.org/classes/ActiveRecord/Calculations.html#method-i-pluck

5 http://api.rubyonrails.org/classes/Object.html#method-i-present-3F

If a reference is found on both steps, *similarity score* for lookup table of the two strings is 1.0. If the system fails to find any reference on any step, consider a no match and the score is 0.0.

By following these steps, the implementation of lookup table is as in lisiting 7.

```
def cal_score
  # Look for a reference for base name.
  base = LookupTableRecord.where(:name => @base_name.name)

  @score = if base.nil? # Could not find reference for base name,
      no matches.
          0
        else
          # Find any reference that has 1) same name 2) same
              reference.
          base = base.map(&:ref)
          refs = LookupTableRecord.where(:ref => base, :name =>
              @name)

          if refs.present?
            @label = (base & refs.map(&:ref)).join(', ')
            @value = "Matched"
            1
          else # Could not find reference for matching name, no
              matches.
            0
          end
        end
end
```

Listing 6: Lookup table implementation.

4 ARCHITECTURE

Now that we already know how to match (by comparing and calculating *similarity score* from section 4), we then proceed to a bigger picture. This section describes how the architecture of overall system is.

4.1 INITIAL IDEA

Let us start by the basic idea of this project.

As mentioned in section 2.3, the objective of this project is to provide a web service that produces matching result between two *lists* of names. As we see the word *list* here, that means our inputs are not only a pair of names, but rather two lists. In real world use, this list can be large, a hundred or thousand, depending on the client who uses the system.

We will introduce two terms, *base name* and *to-match name*. *Base name* acts as a base and will be matched against each *to-match name* in their list, from start until the end, then proceed to the next *base name*, match against the whole *to-match name* list again, and so on.

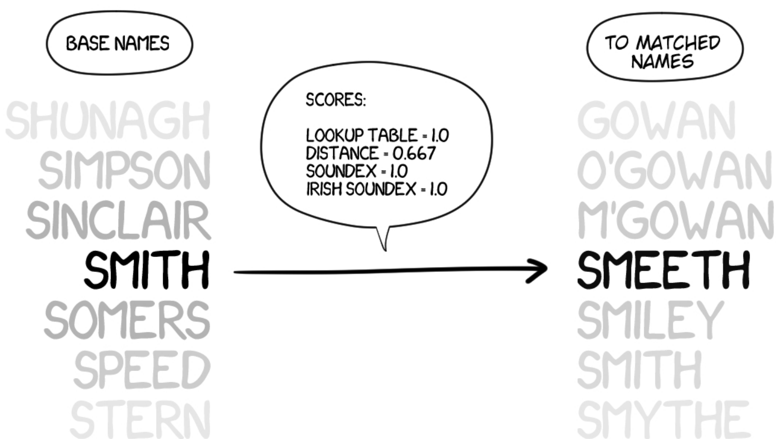


Figure 2: *Base name* 'SMITH' comparing to *to-match name* 'SMEETH'. Scores of each matching algorithms are presented in the bubble above the arrow.

Figure 2 shows a snapshot of an attempt to match between *base name* 'SMITH' against *to-match name* 'SMEETH'. *Similarity score* for each matching algorithms have been calculated. And by these scores, we can calculate *overall score* for 'SMITH' and 'SMEETH'.

So the next step is to match current *base name*, 'SMITH', against next *to-match name*, 'SMILEY'.

Once *base name* 'SMITH' completes all *to-match name*'s in their list, the system then process to the next *base name*, 'SOMERS', and start over the matching process against the whole *to-match name* list again, from start to the end.

4.2 WEIGHTING MATCHING ALGORITHMS

We realised that, for matching names, each matching algorithms should not be treated as all the same priority. For example, for Irish names, it would be better if we favour *Irish soundex* over the traditional *Soundex*, because it produces more accurate result.

By this idea we also implement *weight* for each matching algorithm. We will suggest initial values, but also allow client to change these values. Table 2 states these suggested initial weights.

MATCHING ALGORITHM	WEIGHT
Levenshtein distance	1
Soundex	3
Irish soundex	6
Lookup table	10

Table 2: Matching algorithm weights.

By summarising products of each matching algorithm *similarity score* and its weight, dividing by sum of all weight, we can obtain *overall weighted score* (OWS). This sentence can be represented by equation 1.

$$OWS = \frac{\sum_{i=1}^{n} (s_i \times w_i)}{\sum_{i=1}^{n} w_i} \tag{1}$$

Where s and w are *similarity score* and weight of matching algorithm i respectively. n is number of available matching algorithms.

This *overall weighted score* will represent each matching and all results will be sorted by this score. Usage and calculation of this weighting will be described in more detail in the next section (5.3).

4.3 Actual system

Following our basic idea from previous sections, we then design the architecture of our system.

Suppose we have two inputs, list of *base names* of length b, and list of *to-match names* of length t. We need to process the matching for $b \times t$ times. We call this single matching between *base name* and *to-match name* as *matching cycle*.

In this following figure 3 we once again show a snapshot of an attempt to match between *base name* 'SMITH' against *to-match name* 'SMEETH'. But now in a *matching cycle* style.

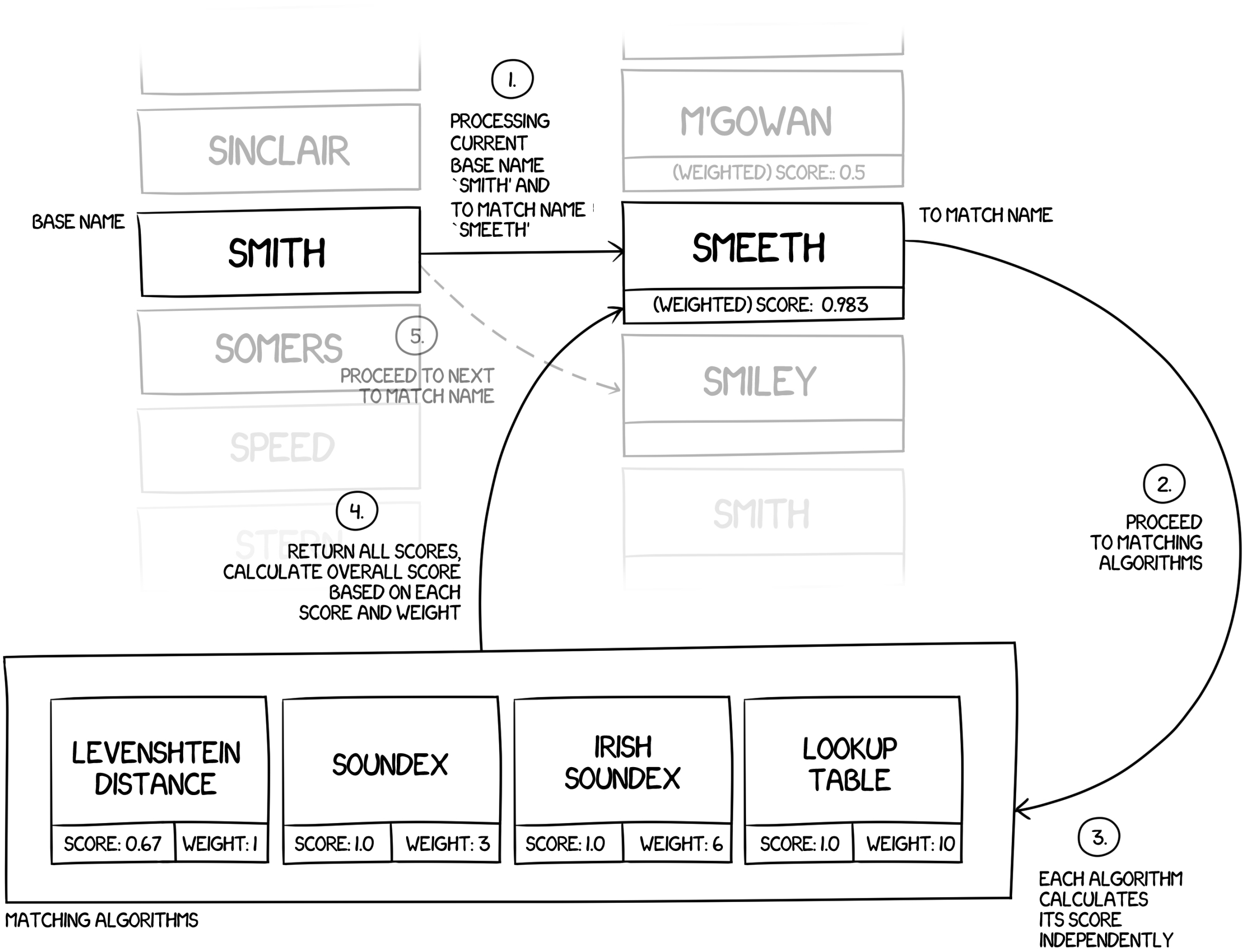


Figure 3: One matching cycle.

One matching cycle consists of 5 steps as shown in figure 3.

1. Processing current *base name* 'SMITH' and *to-match name* 'SMEETH'
2. Proceed to matching algorithms.

3. Each algorithm calculates its score indepedently.
   a) Levenshtein distance between 'SMITH' and 'SMEETH' is 2 (1 substitution of *I* to *E* and 1 insertion of *E*). So *similarity score* is $(6-2) \div 6 = 0.667$ where 6 comes from thelength of the longer string, 'SMEETH'.

      Weight of this algorithm is 1.

      $\therefore$ WEIGHTED SCORE $= 0.667 \times 1 = 0.667$
   b) Soundex of both 'SMITH' and 'SMEETH' are `S530` so their *similarity score* is 1.0.

      Weight of this algorithm is 3.

      $\therefore$ WEIGHTED SCORE $= 1.0 \times 3 = 3.0$
   c) Irish soundex of both 'SMITH' and 'SMEETH' are `S530` so their *similarity score* is 1.0.

      Weight of this algorithm is 6.

      $\therefore$ WEIGHTED SCORE $= 1.0 \times 6 = 6.0$
   d) References between 'SMITH' and 'SMEETH' is found in the Lookup table via group 1897. so their *similarity score* is 1.0

      Weight of this algorithm is 10.

      $\therefore$ WEIGHTED SCORE $= 1.0 \times 10 = 10.0$
4. Return all scores and then calculate *overall weighted score* for 'SMITH' and 'SMEETH'. Sum of the scores is $0.667 + 3.0 + 6.0 + 10.0 = 19.667$. Sum of the weights is $1 + 3 + 6 + 10 = 20$. Therefore the *overall weighted score* is $19.667 \div 20 = 0.983$.
5. Matching cycle for 'SMITH' and 'SMEETH' is finished with *overall weighted score* 0.983. Now the system will proceed to the next *to-match name* 'SMILEY'. Matching cycles for *base name* 'SMITH' will continue until the end of *to-match name* list. After that it will start matching cycles for *base name* 'SOMERS' from the start of *to-match names*, and so on.

Once all cycles are fully finished for every *base names* and *to-match names*, we will get all *overall weighted scores* ready. So we can sort and present in web interface (section 5.8), or return as a result in web service (section 5.7).

4.4 THRESHOLDING THE RESULTS

Suppose there are a thousand of *to-match names*, there could be many irreverent results that are not likely to match each *base name*. For example *overall weighted scores* of 'SMITH' and 'CROMBIE' is just 0.007.

Client may opt-out these irreverent results by specifying a floating number *threshold*. Any *to-match name* with *overall weighted scores* lower than *threshold* will be discarded from the result.

4.5 DATA FLOW

In the previous section we describe the essence of this project, how we use matching algorithms to calculate score of similarity between two strings. We know how to process the data. Now in order to make the system becomes useable. We need to consider two more things.

- How to gather inputs from clients.
- How to present or return results to clients.

From research questions (section 2.2) we mentioned two ways to communicate with clients, by *web service* and *web interface*.

Clients who use the system as a web service will send inputs directly without any medium in between, and will receive result back in form of agreed format, e.g. `JSON`.

On the other hand, clients who use the system via web interface will use a form in a web interface (web page) provided by the system to provide inputs, and results will be presented in another page after client submitted the form.

SERVICE TYPE	INPUT SOURCE	RESULT FORMAT
Web service	Web/mobile/desktop application	`JSON`
Web interface	Provided form	Web page result

Table 3: Service types and their inputs and results.

In the next section (5.6) we will describe how we gather inputs and provide results.

4.6 MVC

Our system is based on Ruby on Rails, which is a MVC[1] framework. We will use Rails architecture to encapsulate our system be these following means.

VIEW: where the form for web interface is implemented. It creates a web page with inputs for use to fill in. Client can inputs names

1 Model-View-Controller [34]

manually, or upload a file containing names. He also can choose whether to use any available matching algorithms.

Inputs from *view* are then passed to *controller*.

View is also responsible in displaying result to web interface clients, and generating `JSON` result for web service clients.

CONTROLLER: receives inputs from different sources, inputs from form of web interface style, or direct input from clients using web service style. Inputs will be pre-processed, such as separating lines from file input, removing white spaces, or converting input to upper-case.

Once inputs are ready, *controller* then passes these inputs to *model*, where our matching system lies in.

After inputs are processed, *controller* receives results back from *model*, then *controller* will decide which kind of result it needs to return from input source. It will then pass results to appropriate *view*.

MODEL: this is where we implement our whole matching system in. Model constructs *base names* and *to-match names* from received inputs, then invoke matching algorithms, as described in section 5.3. *Model* will pass results back to *controller* after finished.

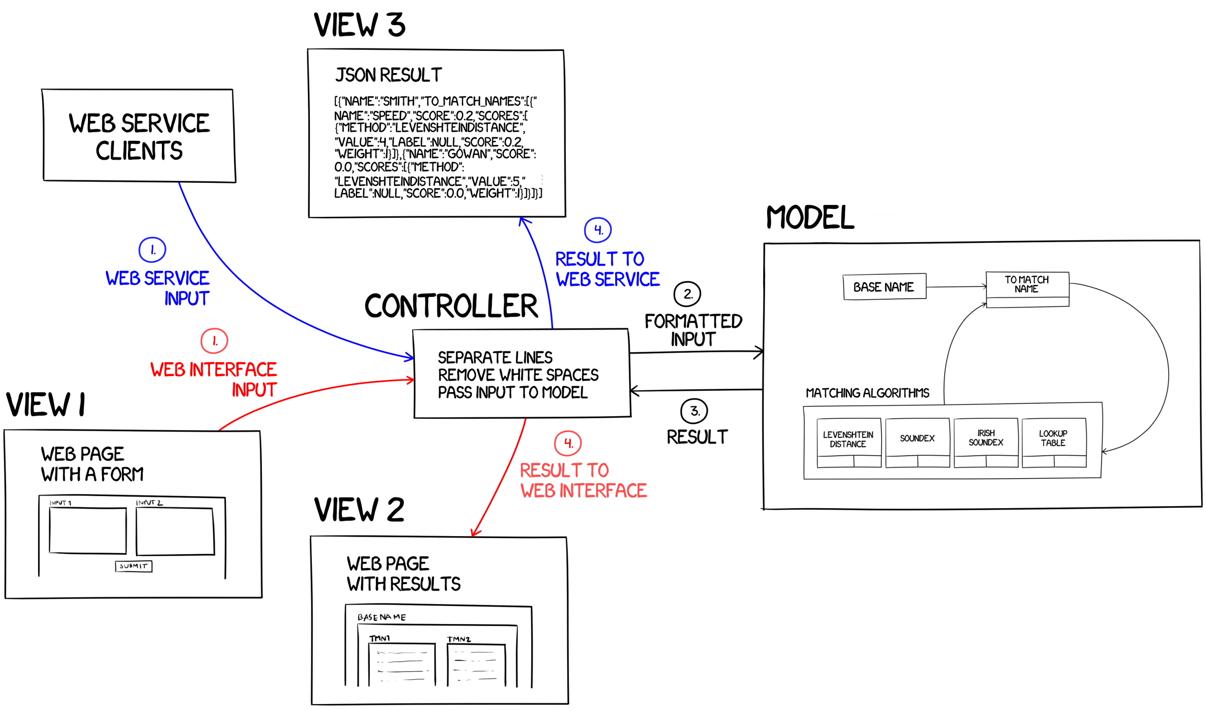


Figure 4: Data flow from web service client and web interface client.

4.7 WEB SERVICE

From figure 4, web service clients use the system by directly pass inputs to the *controller*. Recalling from previous sessions, all possible inputs to the system are as follow.

1. Base names – separated by new lines[2].
2. To-match names – separated by new lines.
3. Matching algorithms – specify needed algorithm names along with its weight (section 5.2).
4. Threshold (section 5.4).
5. Standard list – will be introduced in section 7.1. Specify `true`, `t`, or `1` to use the standard list.

Currently a preferable format is `JSON`. In listing 8 shown a sample `JSON` input for using web service, let us call this `sample.json`.


```json
{
  "base_names":"Smith",
  "to_match_names":"Smythe\r\nO'Gowan",
  "matching_algorithms":{
    "1":{"name":"LookupTable", "weight":"10"},
    "2":{"name":"LevenshteinDistance", "weight":"1"},
    "3":{"name":"Soundex", "weight":"3"},
    "4":{"name":"IrishSoundex", "weight":"6"}
  },
  "threshold":"0",
  "standard_list":""
}
```


Listing 7: `sample.json`.

`sample.json` is an attempt to match between *base name* 'SMITH', and *to-match name* 'SMYTHE' and 'O'GOWAN'. Using 4 matching algorithms, and threshold as 0. We will submit this `sample.json` input to the system using cURL [35] in command line.

```
$ curl -H "Accept: application/json" -H "Content-type:
    application/json" -X POST -d @sample.json
    http://localhost:4001/match.json
```

2 Preferably `\n` or `\r`, see http://stackoverflow.com/questions/1761051/difference-between-n-and-r.

We use HTTP POST [36] to submit `sample.json` to the web service, currently running locally, so the `url` using here is `http://localhost:4001/match.json`. The formatted `JSON` result is shown in listing 9, note that the result of matching between 'SMITH' and 'O'GOWAN' is truncated just for readability.

```
[
  {
    "base_name": "SMITH",
    "to_match_names": [
      {
        "to_match_name": "SMYTHE",
        "overall_weighted_score": 0.983,
        "scores": [
          {
            "method": "LookupTable",
            "value": "Matched",
            "label": "1897",
            "score": 1,
            "weight": 10
          },
          {
            "method": "LevenshteinDistance",
            "value": 2,
            "label": null,
            "score": 0.667,
            "weight": 1
          },
          {
            "method": "Soundex",
            "value": "S530 <=> S530",
            "label": null,
            "score": 1,
            "weight": 3
          },
          {
            "method": "IrishSoundex",
            "value": "S530 <=> S530",
            "label": "SMYTHE",
            "score": 1,
            "weight": 6
          }
        ]
      },
      {
        "to_match_name": "O'GOWAN",
        "overall_weighted_score": 0.5,
        ...
```

Listing 8: Result from `sample.json`.

From the results, *overall weighted score* between 'SMITH' and 'SMYTHE' is 0.983, higher than 'SMITH' and 'O'GOWAN' (0.5), so the former is sorted before the latter.

To generate these results in JSON, we use Jbuilder [37], a template for generating JSON structures. The template we use is shown in listing 10.

```
json.array! @matched_names do |matched_name|
  json.base_name matched_name.name

  json.to_match_names do
    json.array! matched_name.to_match_names do |tmn|
      json.to_match_name tmn.name
      json.overall_weighted_score tmn.score

      json.scores do
        json.array! tmn.scores do |s|
          json.method s.class.name
          json.value s.value
          json.label s.label
          json.score s.score
          json.weight s.weight
        end
      end
    end
  end
end
```

Listing 9: Jbuilder template for generating JSON results.

4.8 WEB INTERFACE

Our system provides a web interface with inputs form. All possible inputs are equivalent to web service as follow.

1. Base names – clients can fill the names in *input 1*, separated by new lines. alternatively, clients can also upload a file containing names separated by new lines. if the web interface detects that the file input is present, direct text input will be discarded.
2. To-match names – the same way of *base name*, using *Input 2*.
3. Matching algorithms – clients can choose available algorithms from the list along with its weight using checkboxes. Uncheck to opt-out any algorithms.
4. Threshold – clients can specify floating number threshold using input box.

5. Standard list – will be introduced in section 7.1. clients can check the checkbox to use the standard list.

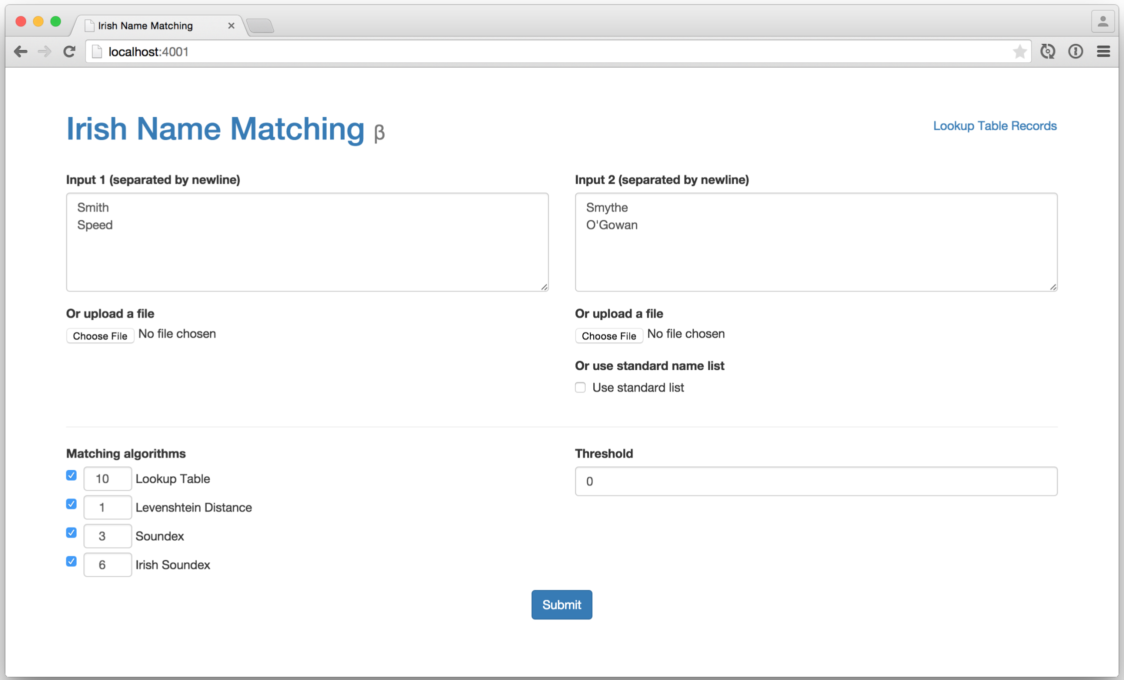
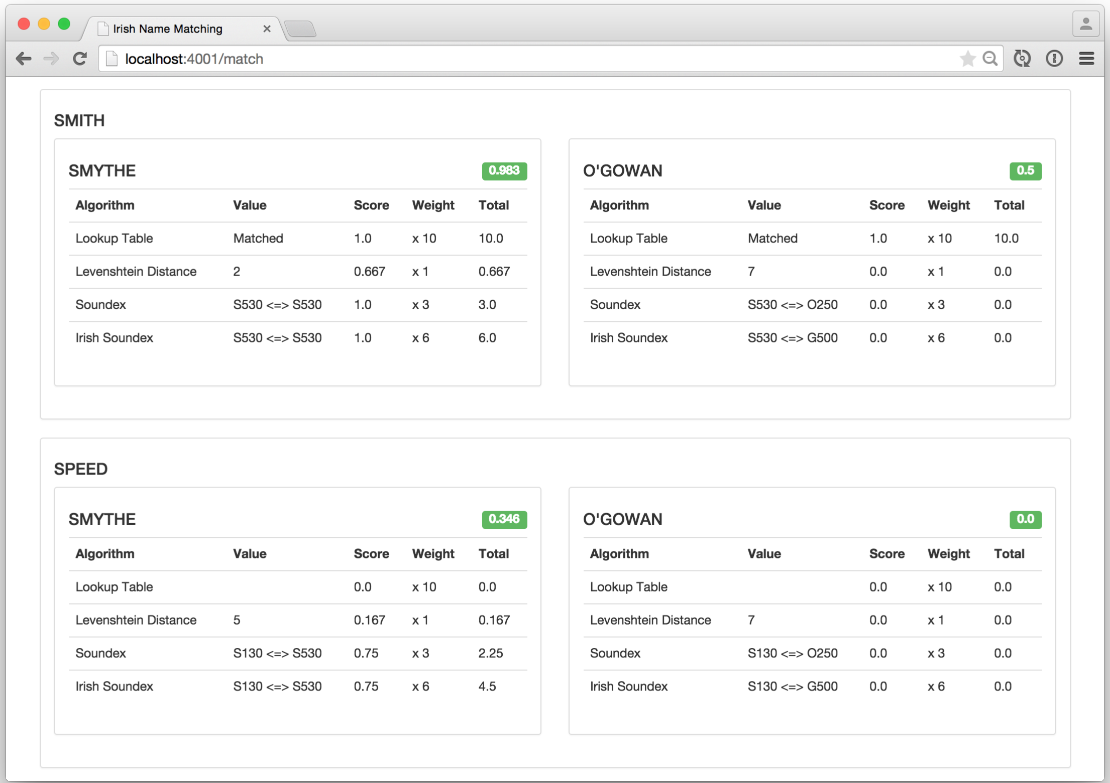


Figure 5: Web interface with input forms.

Figure 5 is an attempt to match between *base name* 'SMITH' and 'SPEED', and *to-match name* 'SMYTHE' and 'O'GOWAN'. Using 4 matching algorithms, and threshold as 0. After finish filling all inputs client may press the blue 'Submit' button to begin matching.



Irish Name Matching

localhost:4001/match

SMITH

SMYTHE 0.983

Algorithm	Value	Score	Weight	Total
Lookup Table	Matched	1.0	x 10	10.0
Levenshtein Distance	2	0.667	x 1	0.667
Soundex	S530 <=> S530	1.0	x 3	3.0
Irish Soundex	S530 <=> S530	1.0	x 6	6.0

O'GOWAN 0.5

Algorithm	Value	Score	Weight	Total
Lookup Table	Matched	1.0	x 10	10.0
Levenshtein Distance	7	0.0	x 1	0.0
Soundex	S530 <=> O250	0.0	x 3	0.0
Irish Soundex	S530 <=> G500	0.0	x 6	0.0

SPEED

SMYTHE 0.346

Algorithm	Value	Score	Weight	Total
Lookup Table		0.0	x 10	0.0
Levenshtein Distance	5	0.167	x 1	0.167
Soundex	S130 <=> S530	0.75	x 3	2.25
Irish Soundex	S130 <=> S530	0.75	x 6	4.5

O'GOWAN 0.0

Algorithm	Value	Score	Weight	Total
Lookup Table		0.0	x 10	0.0
Levenshtein Distance	7	0.0	x 1	0.0
Soundex	S130 <=> O250	0.0	x 3	0.0
Irish Soundex	S130 <=> G500	0.0	x 6	0.0

Figure 6: Web interface results page.

Figure 6 shows the web interface result of this matching. There are two *base names* and two *to-match names*, so the results are matchings of total $2 \times 2 = 4$ names. Each outer box is results of matching between each *base names*, with the outer box there is an inner box containing details of each *to-match names*.

From the results, *overall weighted score* (each green labelled box) between 'SMITH' and 'SMYTHE' is 0.983, higher than 'SMITH' and 'O'GOWAN' (0.5), so the former is sorted before the latter.

5 EXTENDING THE SYSTEM

In this section we show how we design our system to be extensible in terms of adding new matching algorithms, also using existing shared methods, or sharing new methods.

5.1 MATCHING ALGORITHMS INHERITANCE

All matching algorithms (chapter 4) inherit the same superclass, `MatchingAlgorithm` (shown in listing 11). They all use the same class constructor (in Ruby, it is called the *initialize* method). To create an instance of `MatchingAlgorithm`, current *base name*, *to-match name* and *weight* are passed as parameters.

We use *The Strategy Pattern*[1] as a design pattern. In `MatchingAlgorithm` the `cal_score` method is declared and also meant to be overridden, so every matching algorithm needs to override this method using their own matching logic. Each matching algorithm class will call `cal_score` to calculate its scores. `cal_score` is also private to be only used within the subclasses themselves.

We also define `soundex_distance_score` method to be shared between soundex algorithms. Any further shared methods can be declare here as well.

`MatchingAlgorithm` class is shown in listing 11.

1 [38], page 24

```ruby
class MatchingAlgorithm
  WEIGHT = 1 # Default weight of every matching algorithm.

  attr_accessor :name,
    :base_name,
    :value,
    :label,
    :score,
    :weight,
    :weighted_score

  def initialize(params = {})
    @name    = params.fetch(:name)
    @base_name = params.fetch(:base_name)
    @weight  = params.fetch(:weight)

    cal_score
    @score = @score.round(3)
    @weighted_score = (@score * @weight).round(3)
  end

  private

  def cal_score
    raise NotImplementedError
  end

  def soundex_distance_score(s1, s2)
    if s1.first != s2.first
      0 # Different category, so they suppose to be completely
          different
    else
      (s1.size - Text::Levenshtein.distance(s1, s2).to_f ) /
          s1.size
    end
  end
end
```

Listing 10: `MatchingAlgorithm` class.

For example of a concrete matching algorithm, we have already shown some `cal_score` overridings. For *Levenshtein distance* is as in in listing 2. Here we will show whole `LevenshteinDistance` class, which is a subclass of `MatchingAlgorithm`, as in listing 12.

```
class LevenshteinDistance < MatchingAlgorithm
  private

  def cal_score
    @value = Text::Levenshtein.distance(@name, @base_name.name)
    size = [@name.size, @base_name.name.size].max
    @score = ((size - @value).to_f / size)
  end
end
```

Listing 11: `LevenshteinDistance` class.

Also for *Lookup table* is as in listing 7. Here we will show whole `LookupTable` class, which is another subclass of `MatchingAlgorithm`, as in listing 12. Note that `LookupTable` overrides default *weight*, which `LevenshteinDistance` does not.

```
class LookupTable < MatchingAlgorithm
  WEIGHT = 10 # Overriding default weight.

  private

  def cal_score
    ...
  end
end
```

Listing 12: `LookupTable` class.

5.2 EXPORTING A CLASS METHOD

When we mentioned soundex implementation in listing 3, we introduced `self.soundex` method instead of `cal_score`. Defining a method with `self.` is to create a *class method* [39]. *Class method* can be called directly without creating instance of the class, it is the same as *static method* in Java. For example as in listing 14.

```
[7] pry(main)> Soundex.soundex('SMITH')
=> "S530"
```

Listing 13: Calling class method *Soundex.soundex*.

By defining this method to be a *class method*, it can be reused in other class as well. In listing 15 we will show whole `Soundex` class, which is another subclass of `MatchingAlgorithm`.

```
class Soundex < MatchingAlgorithm
  WEIGHT = 3

  def self.soundex(name)
    ...
  end

  private

  def self.category(c)
    ...
  end

  def cal_score
    name_soundex  = self.class.soundex(@name)
    base_name_soundex = self.class.soundex(@base_name.name)

    @value = "#{base_name_soundex} <=> #{name_soundex}"
    @score = soundex_distance_score(name_soundex,
        base_name_soundex)
  end
end
```

Listing 14: `Soundex` class.

Note that we have already covered `self.soundex` and `self.category` implementation in listing 3 and 4 respectively, so both are truncated for readability. Here we focus on the `cal_score` overriding on `Soundex` class. The use of `self.class.soundex` in `cal_score` refers to `Soundex.soundex`. And `soundex_distance_score` is defined in `MatchingAlgorithm`.

As for Irish soundex, it also contains its own `self.soundex` *class method*. But this `self.soundex` also calls `Soundex.soundex` to use original soundex code, as in listing 16 (extracted from `IrishSoundex.soundex` implementation, listing 6).

```
# Call to traditional soundex.
return {
  :label => name,
  :soundex => Soundex.soundex(name)
}
```

Listing 15: `IrishSoundex.soundex` calls to `Soundex.soundex`.

In listing 17 we will show whole `IrishSoundex` class, which is another subclass of `MatchingAlgorithm`.

```
class IrishSoundex < MatchingAlgorithm
  WEIGHT = 6

  def self.soundex(name)
    ..
  end

  private

  def cal_score
    name_soundex  = self.class.soundex(@name)
    base_name_soundex = self.class.soundex(@base_name.name)

    @value = "#{base_name_soundex[:soundex]} <=>
        #{name_soundex[:soundex]}"
    @label = name_soundex[:label]
    @score = soundex_distance_score(name_soundex[:soundex],
        base_name_soundex[:soundex])
  end
end
```

Listing 16: `IrishSoundex` class.

Note that we have already covered `self.soundex` implementation in listing 6, so it is truncated for readability. Here we focus on the `cal_score` overriding on `IrishSoundex` class. The use of `self.class.soundex` in `cal_score` refers to `IrishSoundex.soundex`. And `soundex_distance_score` is defined in `MatchingAlgorithm`.

5.3 IMPLEMENTING NEW MATCHING ALGORITHMS

Currently there are 4 matching algorithms derived from `MatchingAlgorithm`.

1. `LevenshteinDistance`

2. `Soundex`
3. `IrishSoundex`
4. `LookupTable`

The implementations of all of these classes are within the same file as their superclass, `MatchingAlgorithm`, the path is `app/models/matching_algorithm.rb`. The structure of this file is shown in listing 18.

```
class MatchingAlgorithm
  ..
end

class LevenshteinDistance < MatchingAlgorithm
  ..
end

class Soundex < MatchingAlgorithm
  ..
end

class IrishSoundex < MatchingAlgorithm
  ..
end

class LookupTable < MatchingAlgorithm
  ..
end
```

Listing 17: `matching_algorithm.rb`.

To add a new matching algorithm, suppose we were to implement another soundex for Indian [40]. We would call this `IndianSoundex`. Here is the list of steps to do so.

1. Modify the file `app/models/matching_algorithm.rb` where all the matching algorithms are in. Append the class definition to the file.

```
..

class LookupTable < MatchingAlgorithm
  ..
end

class IndianSoundex < MatchingAlgorithm
  private

  def cal_score
    # To be implemented.
  end
end
```

2. Override `cal_score`, fulfil the algorithm for Indian soundex.
3. You may create *class method* `self.soundex` to follow the same pattern as `Soundex` and `IrishSoundex`, making it reusable too.
4. You may also use *class method* of soundex and Irish soundex by calling to `Soundex.soundex` and `IrishSoundex.soundex`.
5. You may also use shared method `soundex_distance_score` to calculate distance score like two other soundexes do.

You may consider adding more shared methods to `MatchingAlgorithm` superclass if considered appropriate, i.e. many subclasses will use it. On the other hand, if you need some method just inside the class, consider create just private methods.

Part II

THE OUTCOME

6 EVALUATION

After we finished implementing our system, we began to evaluate our system in terms of performances of response speed and memory usage. To evaluate the system, we need some decent amount of sample data, so here we introduce the standard name list.

6.1 INTRODUCING STANDARD NAME LIST

In subsection 3.1.3, we mentioned Robert Edwin Matheson, who developed a classification of Irish names. He classified the surnames in Ireland into 2091 groups. Adam Winstanley's work on this classification [13] looked through these groups and came up with a total 12,944 names in this classification. We use all these names to build up our lookup table (section 4.4).

We also decide to use all these records as a standard name list, for example, a client may want to match the *base name* 'MONAHAN' for all any possible matching *to-match names*. A client has an option to choose to match 'MONAHAN' with all 12,944 names in our standard list.

For web service clients, specify `standard_list` as `true`, `t`, or `1` to use the standard list. For example in listing 19, note that `to_match_names` is left blank and `standard_list` value is 1.

```
{
  "base_names":"Monahan",
  "to_match_names":"",
  "matching_algorithms":{
    "1":{"name":"LookupTable", "weight":"10"},
    "2":{"name":"LevenshteinDistance", "weight":"1"},
    "3":{"name":"Soundex", "weight":"3"},
    "4":{"name":"IrishSoundex", "weight":"6"}
  },
  "threshold":"0",
  "standard_list":"1"
}
```

Listing 18: Sample `JSON` with a standard name list option.

For web interface clients, check the "Use standard list" checkbox to use the standard list, as shown in figure 7.

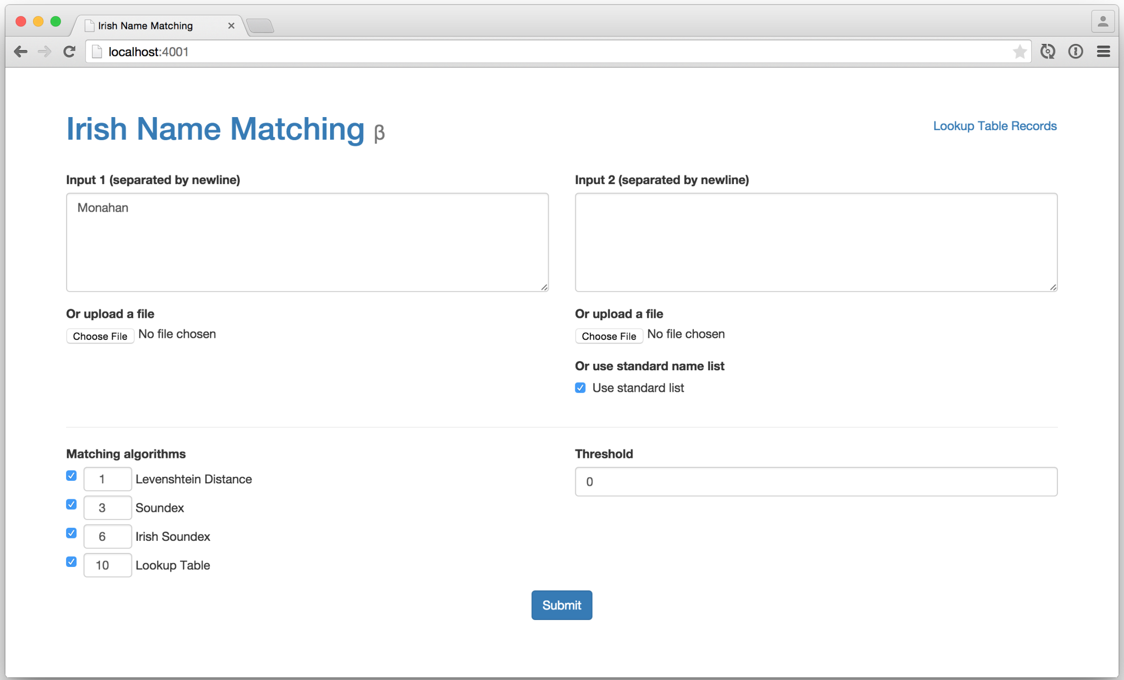


Figure 7: Web interface with a standard name list option.

Using a standard name list option generates many results. Client is suggested also specify a proper threshold (section 5.4) to discard irreverent results.

In listing 20 is a result of matching between *base name* 'MONAHAN' and the standard name list, using threshold as 0.9. Results' detailed scores are truncated for readability.

```
[
  {
    "base_name": "MONAHAN",
    "to_match_names": [
      {
        "to_match_name": "MONAHAN",
        "overall_weighted_score": 1,
        ..
      },
      {
        "to_match_name": "MOYNAHAN",
        "overall_weighted_score": 0.994,
        ..
      },
      {
        "to_match_name": "MONOHAN",
        "overall_weighted_score": 0.993,
        ..
      },
      {
        "to_match_name": "MONEHAN",
        "overall_weighted_score": 0.993,
        ..
      },
      {
        "to_match_name": "MOYNIHAN",
        "overall_weighted_score": 0.988,
        ..
      },
      {
        "to_match_name": "MOYNAN",
        "overall_weighted_score": 0.979,
        ..
      }
    ]
  }
]
```

Listing 19: Results of matching *base name* 'MONAHAN' with a standard name list.

6.2 TEST ENVIRONMENT SETUP

We run, test, and profile our system locally, using these following environmental setup.

TEST MACHINE: MacBook Pro (Retina, 13-inch, Mid 2014).

- Processor – 2.6 GHz Intel Core i5
- Memory – 8 GB 1600 MHz DDR3
- Hard disk – APPLE SSD SM0256F

RUBY: 1.9.3p125 (2012-02-16 revision 34643) with GC-Patched MRI[1].

RUBY ON RAILS: version 4.2.0.

DATABASE: PostgreSQL version 9.3.5.

PROFILING TOOLS: rails-perftest [41] 0.0.6.

6.3 RESPONSE SPEED

We test response speed of our system by matching *base name* 'SMITH' with the standard name list of total 12,944 names. Each matching algorithm is tested separately first and then altogether at last.

Listing 21 is our JSON setup for response speed testing. Matching algorithms are varies between each scenario and all use default weights.

```
{
  "base_names":"Smith",
  "to_match_names":"",
  "matching_algorithms":{
    ..
  },
  "threshold":"0",
  "standard_list":"1"
}
```

Listing 20: JSON setup for performance testing.

To conduct testing, we use *rails-perftest* [41] to run our test cases. Table 4 shows the test result in response speed aspect.

MATCHING ALGORITHMS	RESPONSE SPEED (MS)
Levenshtein distance	1,337
Soundex	2,024
Irish soundex	2,456
Lookup table	24,293
All 4 algorithms	28,786

Table 4: Response speed for each matching algorithms.

1 Installing GC-Patched MRI [41].

From the results, Levenshtein distance is the fastest matching algorithm because it has the simplest logic among the four. Soundex and Irish soundex are second and third because they involve more string converting logic, and Irish soundex has more steps. Lastly, lookup table involves many database queries so that makes it much more slower than the rest.

6.4 MEMORY USAGE

We also test memory usage of our system by matching *base name* 'SMITH' with the standard name list of total 12,944 names. Each matching algorithm is tested separately first and then altogether at last.

Listing 21 is still our `JSON` setup for response speed testing. Matching algorithms are varies between each scenario and all are using default weights. Table 5 shows the test result in memory usage aspect.

MATCHING ALGORITHMS	MEMORY USAGE (BYTES)
Levenshtein distance	48,518,621
Soundex	53,066,150
Irish soundex	69,534,598
Lookup table	244,302,744
All 4 algorithms	373,544,727

Table 5: Memory usage for each matching algorithms.

From the results, memory usage for each algorithm follows response speed fashion. Levenshtein distance and two soundexes use much less memory compared to lookup table.

7 CONCLUSION

We successfully developed an extensible web service system to match names. The system is initially encoded with 4 matching algorithms, Levenshtein distance, soundex, Irish soundex, and lookup table. We also present a web interface for a client to use the system from the web browser.

The system is designed to be extended with simple inheritance, thus developer can understand and develop further algorithm easily. In early state simple design is enough to serve the purpose, so we follow the *Kiss principle* [1].

However, we have encountered some problem, also there are still many rooms for future works. We will describe these in following sections.

7.1 ENCOUNTERED PROBLEM

The major problem is that the current system takes too long to process and also use too much memory. It has not been properly optimised in term of performance. These following techniques might improve our system furthermore.

7.1.1 *Memoization*

Memoization is the process of storing a computed value to avoid duplicated work by repetitive calls. While each algorithm calculats *similarity score*, there might be many repetitive calculations or database queries.

Ruby has a conditional assignment operator `||=` [2] which is commonly used for memoization. By doing so, it can improve performance of the system and reduce the number of database calls [3], thus shorten response time and lower memory usage.

7.1.2 *find_in_batches*

Matching large amount of name causes high memory consumption and may lead to out of memory situlation, especially in environment which memory are crucial and expensive such as remote server.

Rails provides `find_in_batches` which operates an array in batches, thus greatly reducing memory consumption [4]. We can apply the same principle to our *base name* and *to-match name*, also to the *controller* (section 5.6).

7.1.3 *Replace RDBMS with NoSQL*

Current database system (section 7.2) is a *Relational database management system (RDBMS)* [5] and the system relies on traditional database queries. By replacing this with high speed NoSQL database such as Redis [6], which is one of the fastest NoSQL [7], we can obtain better performance while using lookup table algorithm.

7.2 FUTURE WORKS

Our system can be further enhanced in many mays. Here are sample ideas for upcoming features of the system.

7.2.1 *More phonetic algorithms*

A phonetic algorithm [8] indexes words by their pronunciation. Soundex is also one of them. We can implement more matching algorithms based on them. For example, Kolner Phonetik [9] is similar to soundex and works well on German words. Daitch-Mokotoff soundex [10] is an improved soundex working well to match surnames of Slavic and Germanic origin.

7.2.2 *Inheritance for similar matching algorithms*

Currently all matching algorithms inherit `MatchingAlgorithm` class. In future, if there are many similar ones or can be categorised in the same group, we can create another level of inheritance so they can share common methods. For example, consider the soundex case, with Kolner Phonetik and Daitch-Mokotoff soundex we can create inheritance with `Soundex` class as in listing 1.

```ruby
class Soundex < MatchingAlgorithm
  WEIGHT = 3

  def self.soundex(name)
  ..
  end

  private

  def self.category(c)
  ..
  end

  def cal_score
  ..
  end

  # Moved from MatchingAlgorithm class to be more specific to
      soundexes.
  def soundex_distance_score(s1, s2)
  ..
  end
end

class IrishSoundex < Soundex
..
end

class KolnerPhonetik < Soundex
..
end

class DaitchMokotoffSoundex < Soundex
..
end
```

Listing 21: Soundex inheritance.

7.2.3 *Improve web interface result*

Current web interface result is as in figure 6, just a list of boxes detailed with matching algorithm scores. When it comes to large number of inputs, thousands of boxes will be generated and could overwhelm both browser and client themselves.

We can improve result display by implement a visualised graph base on the results, there are many libraries [11] that are capable if generating interactive graph. d3js is another well option for starting from scratch.

BIBLIOGRAPHY

[1] apache.org. *The Kiss Principle*. 2015. URL https://people.apache.org/~fhanik/kiss.html. accessed May 28th, 2015.

[2] Peter Cooper. *What Ruby's ||= (Double Pipe / Or Equals) Really Does*. October 2011. URL http://www.rubyinside.com/what-rubys-double-pipe-or-equals-really-does-5488.html. accessed May 29th, 2015.

[3] Gavin Miller. *The Basics of Ruby Memoization*. November 2013. URL http://gavinmiller.io/2013/basics-of-ruby-memoization. accessed May 29th, 2015.

[4] Arne Hartherz. *Use find_in_batches to process many records without tearing down the server*. 2011. URL http://makandracards.com/makandra/1181-use-find_in_batches-to-process-many-records-without-tearing-down-the-server. accessed May 29th, 2015.

[5] Wikipedia. *Relational database management system*. 2015. URL http://en.wikipedia.org/wiki/Relational_database_management_system. accessed May 29th, 2015.

[6] redis.io. *Redis*. 2015. URL http://redis.io. accessed May 29th, 2015.

[7] Conor Branagan and Patrick Crosby. *Understanding the Top 5 Redis Performance Metrics*. 2013. URL https://www.datadoghq.com/wp-content/uploads/2013/09/Understanding-the-Top-5-Redis-Performance-Metrics.pdf. accessed May 29th, 2015.

[8] Wikipedia. *Phonetic algorithm*. 2015. URL http://en.wikipedia.org/wiki/Phonetic_algorithm. accessed May 29th, 2015.

[9] Wikipedia. *Kolner Phonetik*. 2015. URL http://de.wikipedia.org/wiki/Kolner_Phonetik. accessed May 29th, 2015.

[10] Wikipedia. *Daitch-Mokotoff soundex*. 2015. URL http://en.wikipedia.org/wiki/Daitch-Mokotoff_Soundex. accessed May 29th, 2015.

[11] Phattara Wangrungarun. *Chart / Graph*. 2015. URL https://delicious.com/phatograph/Chart%20%2F%20Graph. accessed May 29th, 2015.

[12] welfare.ie. *History of Registration in Ireland*. 2015. URL https://www.welfare.ie/en/downloads/GRO-History.pdf. accessed May 8th, 2015.

[13] Adam Winstanley. *Identifying People on the Morpeth Roll*. July 2014. Postgraduate Diploma in Genealogical, Palaeographic & Heraldic Studies 2013-14.

[14] Christopher Ridgway. *The Morpeth Roll: Ireland identified in 1841*. 2013.

[15] Wikipedia. *Edit distance*. 2015. URL http://en.wikipedia.org/wiki/Edit_distance. accessed May 4th, 2015.

[16] Wikipedia. *Levenshtein distance*. 2015. URL http://en.wikipedia.org/wiki/Levenshtein_distance. accessed May 4th, 2015.

[17] National Archives and Records Administration. *The Soundex Indexing System*. May 2007. URL http://www.archives.gov/research/census/soundex.html. accessed May 4th, 2015.

[18] Robert Edwin Matheson. *Varieties and synonymes of surnames and christian names in Ireland*. 1901. URL https://archive.org/details/varietiessynony00math. accessed May 4th, 2015.

[19] Robert Edwin Matheson. *Special report on surnames in Ireland*. 1894. URL https://archive.org/details/cu31924029805540. accessed May 4th, 2015.

[20] Vincent Ramdhanie. *What is a 'web service' in plain English?* October 2008. URL http://stackoverflow.com/a/226159/459794. accessed May 5th, 2015.

[21] The Java EE 6 Tutorial. *Types of Web Services*. 2013. URL http://docs.oracle.com/javaee/6/tutorial/doc/giqsx.html. accessed May 5th, 2015.

[22] John Mueller. *Understanding SOAP and REST Basics*. January 2013. URL http://blog.smartbear.com/apis/understanding-soap-and-rest-basics/. accessed May 5th, 2015.

[23] Steve Francia. *REST Vs SOAP, The Difference Between Soap And Rest*. January 2010. URL http://spf13.com/post/soap-vs-rest. accessed May 5th, 2015.

[24] Introducing JSON. *json.org*. 2015. URL http://www.json.org. accessed May 5th, 2015.

[25] Hussain Fakhruddin. *JSON or XML – Which Data Format Is Better For Developers?* April 2015. URL http://teks.co.in/site/blog/json-or-xml-which-data-format-is-better-for-developers/. accessed May 5th, 2015.

[26] Bill Venners. *The Philosophy of Ruby*. September 2003. URL http://www.artima.com/intv/rubyP.html. accessed May 6th, 2015.

[27] ruby lang.org. *Ruby, a programmer's best friend*. 2015. URL https://www.ruby-lang.org/en/. accessed May 6th, 2015.

[28] tutorialspoint.com. *Object Oriented Ruby*. 2015. URL http://www.tutorialspoint.com/ruby/ruby_object_oriented.htm. accessed May 6th, 2015.

[29] ruby lang.org. *About Ruby*. 2015. URL https://www.ruby-lang.org/en/about/. accessed May 6th, 2015.

[30] rubyonrails.org. *Web development that doesn't hurt*. 2015. URL http://rubyonrails.org. accessed May 6th, 2015.

[31] Kresimir Bojcic. *What are the Benefits of Ruby on Rails? After Two Decades of Programming, I Use Rails*. 2013. URL http://www.toptal.com/ruby-on-rails/after-two-decades-of-programming-i-use-rails. accessed May 6th, 2015.

[32] Paul Battley. *Text, Collection of text algorithms*. 2015. URL https://github.com/threedaymonk/text. accessed May 9th, 2015.

[33] postgresql.org. *PostgreSQL*. 2015. URL http://www.postgresql.org. accessed May 11th, 2015.

[34] Wikipedia. *Model-view-controller*. 2015. URL http://en.wikipedia.org/wiki/Model-view-controller. accessed May 13th, 2015.

[35] curl.haxx.se. *cURL, a command line tool and library for transferring data with URL syntax*. 2015. URL http://curl.haxx.se. accessed May 22nd, 2015.

[36] Wikipedia. *POST (HTTP)*. 2015. URL http://en.wikipedia.org/wiki/POST_(HTTP). accessed May 22nd, 2015.

[37] Rails. *Jbuilder*. 2015. URL https://github.com/rails/jbuilder. accessed May 22nd, 2015.

[38] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. 2004.

[39] railstips.org. *Class and Instance Methods in Ruby*. May 2009. URL http://www.railstips.org/blog/archives/2009/05/11/class-and-instance-methods-in-ruby/. accessed May 26th, 2015.

[40] Santhosh Thottingal. *Phonetic Comparison Algorithm for Indian Languages*. July 2009. URL http://thottingal.in/blog/2009/07/26/indicsoundex/. accessed May 26th, 2015.

[41] Ruby on Rails. *rails-perftest – Performance Testing Rails Applications*. 2015. URL https://github.com/rails/rails-perftest. accessed May 27th, 2015.