Appendices

1. Geocoding

1.1. Uncertainty of location names

Most users of rootsweb.com who generated the family trees in our study are in the U.S. and Canada. We mainly focus on these family trees in the U.S. and their origins in other countries. Non-US and Canada locations were established much earlier, and there are more extinct places, more misspellings and spelling variations of the place names due to the change in language over time. For the U.S., changes also occurred through time with the territories acquired in the North West and the West and subdivided places in the North and Northeast. These changes pose substantial challenges in mapping the population demography and migration, and thus, studying the demographic and geographic expansion of the country. Once states were established, there were some locations that were put into a neighboring state, but these were few and involved only small parcels of land: for example some towns in Rhode Island became part of Massachusetts in 1861 and others moved from Massachusetts to Rhode Island; Alexandria moved back and forth between Virginia and the District of Columbia. More dramatically, the Honey Warin 1939 between Iowa (then a Territory) and Missouri involved a strip along the length of the boundary between them.

Place names were entered by users who relied on their personal memories, records, genealogies and census, or an educated guess of where a person in their tree was born or died. Location names entered by users may be geocoded with no problems, however, the entry could still be inaccurate. The only way to check the entries would be to identify the same individual in a census, or a genealogical record to see whether the locations match. In this study, we consider locations that we can match with a place name in our reference dataset as accurate. Future studies are needed to identify individuals in censuses and other sources to compare locations.

There are several challenges for geocoding due to the uncertainty of location names included in birth and death records. We classify uncertain location names into a category named “confusing”. We review these uncertain location names and adjust our reference datasets and our matching criteria to resolve some of the uncertainties and reduce the number of unmatched records.

1. Although the process of geocoding matches the place name with one location, sometimes two alternative locations were entered by users because the likelihood of location of birth or death were in two or more places. For example, a birthplace entry includes "North Carolina or Russia". Sometimes the conflict arises from the use of two different events for the location such as the location of baptism versus birthplace, or death location versus burial place.
2. Some place names include "prob. MA", which states the probable location is MA – likely to be Massachusetts, U.S. However, the location is not known with certainty.
3. Because we are using different reference datasets, there are conflicting matches for place names, especially when abbreviations are used. For example, CA matches Canada when using the country reference data, while it matches California using the US state reference data. Another example is "Beverley; WA", which may refer to "Beverley; West Australia" in the country reference set or "Beverley, Washington, USA" in the U.S. reference set. In similar cases in which location name and potential state or country name were used in lastfield1 and lastfield2, we picked the U.S. location over other locations in the world. This is because we “WA” and the location name “Beverley” both match with the U.S. reference dataset, which is prioritized and checked before the third reference dataset that Australia reference location names are in.
4. The US and states reference sets include full names, two letter, three letter and other common abbreviations, and common alternative spellings, and misspellings that we identified with Levenshtein similarity method. Some common spellings and misspellings are: [“American”, “Amerika”] for the US; [“Ark”, “Arkansa”, “Arkanas”] for Arkansas; [“C A”, “Californien”, “Callifornia”] for California; [“Illinoise”, ”Illionois”] for Illinois; and [“Tenn”, “Tenneessee”] for Tennessee.

5. There are errors in the original data. For example, a place name is entered as "Texas Co.; Montana". The abbreviation “Co.” is used to refer to “County”; however, we cannot find Texas County in Montana.

1.2. Reference data

Our reference data consist of five datasets:

1. US and US states (full names, abbreviations that include two letter, three letter, and other common abbreviations, and common misspelling of names.
2. UK and historical UK place names
3. Canada, Netherland, Germany, Australia names and their first administrative division names
4. Other Europe country names
5. Other country names

1.3. Geocoding workflow

Place names require standardization since some records include special characters, numbers and multiple spaces. We used a multi-step filtering and standardization of place names in both the reference and the family tree data. We split the place name into three fields: “Name”, “lastField2” and “lastField1”, to categorize different levels of location hierarchy such as place names, state names and country names in the U.S. While we use “lastField1” and “lastField2” for a strict criterion of an "exact match" with the reference data, we use the “Name” field to match the place name based on "contains" criteria, which includes partial match of a place name in the reference dataset. Because the geocoding criterion are different, the reference datasets are also different for each of these three fields. In each of the reference sets we include values for match and values for exclusion to remove any unwanted matches. For example, the value “NO” matches with Norway in the country reference set when it appears in lastField1 or lastField2, while “NO” is used for exclusion if it appears in the “Name” field because “NO” is flagged with an exclusion rule in place names reference set since it is used to denote “number”. Our workflow consists of three steps. In step 1, we clean the place names and extract the fields:

1. We replace all numbers (0-9), special symbols (like !, @, #...) except “;” and space with space. The output is called: “Name”.
2. We extract the last field, i.e., the last meaningful segment, which is not equal to space or null, and not separated by semicolon. The output is called: “lastField1”
3. We extract the last field, i.e., the last meaningful segment separated by space. The output is called: “lastField2”

In step 2, we match the records with references based on lastField1 and lastField2, both of which are checked with our reference data using the “exact match” criterion. We compare place names with the rules and the reference datasets in the following order:

1. We check the record with the “confusing” reference data when the record may refer to different places. For example, CA may mean Canada or California. We add place names associated with Canada and California in the confusing reference data to resolve the uncertainty. This is done when we can parse the name into “lastField1” or “lastField2”, and “Name”.
2. We check whether the record match with lastField1, which is separated by semicolon. For example, NO (Norway), AS (Australia) are in the country reference data. However, if these abbreviations appear in middle of names, they may have other meanings (e.g., NO: number) rather than place names.
3. We check the record with the confusing reference data which includes names such as West Virginia, New Mexico, Austria-Hungary, and their variants. We generate the confusing reference set iteratively when we try to geocode unmatched place names in the confusing category. For example, a place name “Virginia” may be geocoded to West Virginia or Virginia because lastField1 and lastField2 do not exist in this record and “contains” criterion is used. To resolve this uncertainty, we update the confusing reference set and add a rule to match

the record with “Virginia” when the place name cannot be parsed into those three fields.

4. We check the record with the reference datasets in the following order: U.S. reference data (4th), U.K. reference data (5th), Canada/German/Australia/Netherland (6th), Europe countries (7th), Other countries (8th), historical countries (9th). Both the lastField1 and lastField2 are used to match these reference data.

In step 3, we match the records with the references based on the Name field. Name field should contain the corresponding name in the reference data.

1. We check the record with “confusing” reference data.
2. We check the record with “contains” reference data.
3. We check the record with the reference datasets in the following order: U.S. reference data (3rd), U.K. reference data (4th), Canada/German/Australia/Netherland (5th), Europe countries (6th), Other countries (7th), historical countries (8th). The reference data used here is different from the reference data in step 1. For example, DE is deleted from U.S. reference data in step 2, since DE may have other meanings when it appears in the “Name” field.

All geocoded records are saved into eight tables based on their locations with appropriate attributes attached: confusingTable, usTable, ukTable, cagnTable (Canada / Australia / Germany / Netherlands), europeTable, othercountryTable, historicalcountryTable and unmatchedTable.

1.4. Evaluation

We checked the geocoding accuracy by checking the alternative matches in our reference sets and searching online sources (e.g., Wikipedia, Google, etc.) for the matched and unmatched place names. We geocoded all place names at state level for the U.S. and country level for the rest of the world. We first prioritized geocoding with the U.S. States reference dataset, and then checked the U.K. reference set and other country reference datasets. For place names that did not match with U.S. reference dataset, we checked whether the record match with the other country reference sets, and the place name had to have the state and country name to be matched with those reference sets. If country name did not exist, we classified the record in “confusing” or in other words, “uncertain” category. This is rather a conservative approach that leaves many records unmatched. However, this was a choice we made to keep the uncertainty minimal because most of our analysis rely on location information. We applied the “winner takes all” approach to match uncertain place names with country information based on their frequency of occurrence to reduce the number of place names in the confusing category. Then, we matched the place names without a country or state name with the most common location occurred in those place names with the country or state name.

We selected 1% stratified sample from the geocoded data. We selected one geocoded place from every hundred place names 1st, 101st, 201st, 301st, and so on, to evaluate the geocoding accuracy. Overall, 98.05% of the selected records can be geocoded to the state and country level with exact match criterion. Among the unmatched records, we were able to match 49 out of 135 by adding the new exceptions into our reference datasets. For example, a place name that included an Australian State without the country name, “New South Wales”, was wrongly geocoded to “Wales” which existed in “Wales” in the U.K. reference set. This was because records without a country name or U.S. state name are compared with the reference datasets using the “contains” criteria. “New South Wales” was matched with “Wales” in the U.K. reference set because of the prioritized order of search in the reference datasets.

There are several reasons why we did not use a geocoding software to check the accuracy. First, errors also exist in available geocoding software and packages. Second, our data is user-generated and unsuitable to be directly used for most geocoding software. Third, our reference datasets are from multiple sources and historical, and we employed different matching criterion for each of the reference data sources. These tasks are hard or even impossible to complete with the existing geocoding software and tools.

1.5. Limitations and future work

We geocode place names before eliminating the duplicates in trees. However, the process of deduplication in which we check whether locations match could be used to identify conflicting information, which we could use to evaluate the uncertainty of location names. The summary statistics of how many pairs match, and when they match what the differences could be used to enhance our geocoding methodology. Other genealogical databases have the user verify a place against an atlas with coordinates they maintain and prompt until they get a match. However, enforcing individuals to make a guess would increase the number of geocoded records as well as the uncertainty of locations.

There are several improvements to be made to the reference data. First, we included the common misspellings in our reference data and geocoded using the exact match criterion. However, in future work, we could use parts of the correct names to capture most of the misspelling names, for example, when a name contains “Pennsyl” it could be matched with “Pennsylvania” with a confidence score. Second, several reference location names are not used in step 2 because they relate to multiple locations, for example, IN, and NO. However, we may use, for example “NO”, when there are no numbers following the place name. Third, we can add more reference data based on the unmatched names. We plan to geocode the unmatched data to place names in the U.S.

In addition to improving the reference data, our geocoding workflow could be improved. First, there is a need to store temporal snapshots of locations to better handle geocoding of historical place names. This may require storing country, state, and place names and their corresponding references for the most detailed temporal resolution if available. The resolution may be a decade or even a year if available. Second, we do not currently resolve some of the confusing category such as the issue of the inclusion of two different places: “Virginia or NC”. In future work, we plan to use the cleaned family tree data for resolving such conflicts. For example, we may use the nearest event such as the birth of a sibling or a child, and the death of a spouse. We could compare such events when there are two alternative locations in the confusing category to see if the nearest event matches with one of them or not. We have not used family relationships in the geocoding process. Because we already use location information to match individuals in multiple trees. Family tree relationships could be a valuable source for example, for checking the distances between spouses' birthplaces. We plan to evaluate our geocoding using family relationships in the future; however, such evaluation is challenging as the data, the trees we are studying, have high geographic mobility and diffusion over North America. Although individuals usually marry others who are from same ethnicity, origin or locations, we expect to see substantial variation of spousal relationships given the context of our study and time periods. We plan to study particularly spousal relationships in future work, that could potentially be useful to improve our geocoding process.

We have started to geocode at the county level in the US. However, the evaluation process is rather more difficult, and at this stage we report the results at the state level. In our latest experiment, we were able to geocode 99.77% of locations to the state level, while we were able to geocode 78.63% to the county level in the US.

2. Algorithms

In this section, we describe algorithms we created for steps 4 and 5 in our methodology described in section 3.1.

2.1. Saving person records into blocks

For each GEDCOM file, we first extract and save persons with detailed information including gender, birth year, birthplace, first name, last name, individual's family tree id, mother's, father's and spouse's information into blocks. Each GEDCOM file may include more than one family tree with unique id for each file. The algorithm 2.1 produces three outputs: (1) The blocks of persons indexed by birthplace and gender and sorted by birth year. We built each block (index) based on gender and birthplace of persons to improve the efficiency of the search queries to match identical persons in multiple trees. We then sort the persons in the same block by birth year. (2) A Hash map of individuals in which the key is block id, whereas the value is the person object that contain all features of an individual.

2.1. Algorithm for saving individual records into blocks for efficient processing

---

Input: *G:* GEDCOM file collection. Each GEDCOM file *g* ϵ G include individual records.

Each individual record *i* ϵ *g* include features: *id:* an individual's unique identification number within a tree, *bp*: birthplace, *ge*: gender, *by*: birth year, *ln*: last name, *fn*: first name, *dp*: death place, *dy*: death year, *f*: father, m: mother and *S*: spouse list and *FTID*: family tree id that is unique and equals to *g.id* (GEDCOM file id).

Output: *B:* The blocks of persons indexed by birthplace and gender and sorted by birth year.

personHashByBlock: The hash map of person hash maps by block. The key is block id and the values are the hash map of person objects that contain all attributes of a person within a block.

personID_Block: Hash map to store block id for each person.

personHashByTree: The hash map of person hash maps by each tree. The key is tree id and the values are the hash map of person objects that contain all attributes of a person within a tree. *personHashByTree* is used in Algorithm 2.3. Tree Cleaning and Deduplication.

```
initialize B as blocks, personHashByBlock, personID_Block, personHashByTree
rcnt = 0
foreach gedcom g ϵ G
    foreach person i ϵ g
        if i.ln, i.fn, i.by, i.bp and i.ge are empty or not valid
            continue
        p = createPerson(i)
        // assign unique tree id
        p.tid = i.FTID
        if i.f !=null
            p.father = createPerson(i.f)
        if i.m !=null
            p.mother = createPerson(i.m)

        foreach spouse s ϵ S && S.length < 3
            p.spouse = createPerson(s)
        blockid = i.bp + "'&" + i.ge
        if blockid is not in blocks
            blocks.add (blockid)
        personHash = personHashByBlock.get(blockid)
        if personHash == null
            initialize personHash

        personHashTree = personHashByTree.get(p.tid)
        if personHashTree == null
            initialize personHashTree

        p.id = rcnt
        personHash.put(p.id, p)
        personHashTree.put(p.id, p)
        personID_Block.put(p.id, blockid)
        personHashByBlock.add(blockid, personHash)
        personHashByTree.add(p.tid, personHashTree)
        rcnt += 1
    if rcnt > 2,000,000 or end of gedcom files
        blocks.sortByBirthYear()

```

2.2. Fuzzy matching and connecting family trees

Fuzzy matching consists of two steps: Weights assignment algorithm to match similar individuals and spousal pairs matching and tree clustering algorithms for connecting family trees.

2.2.1. Weights assignment for matching similar individuals

We use fuzzy feature weighting to add more weights to features that can be used to distinguish persons. Weights are determined based on available information about the person, his/her parents and spouse (i.e., the first spouse if a person has multiple spouses). The total weight (score) is 100 (%), and the initial match score is 25 because gender (15) and birthplace (10) are the same within each block. The weight for each feature is listed below:

Birth year: We compare the birth years of persons', their parents and spouses. If the difference is the maximum of 5 years, then the weight equals to 0.

- If birth years of persons are the same, the weight equals to 10.
- If birth years of persons' fathers are the same, then the weight equals to 2.
- If birth years of persons' mothers are the same, then the weight equals to 2.
- If birth years of persons' spouses are the same, the weight equals to 2.

Death year: We compare the death years of persons', their parents and spouses. If the difference is the maximum of 5 years, then the weight equals to 0.

- If death years of persons are the same, then the weight equals to 3.
- If death years of persons' fathers are the same, then the weight equals to 1.
- If death years of persons' mothers are the same, then the weight equals to 1.
- If death years of persons' spouses are the same, then the weight equals to 1.

First name: Using Levenshtein similarity, we compare the first name of persons, their parents and spouses.

- If persons' first names match exactly, then the weight equals to 10.
- If persons' mothers' first names match exactly, then the weight equals to 7.
- If persons' fathers' first names match exactly, then the weight equals to 7.
- If persons' first spouses' first names match exactly, then the weight equals to 7.

Last name: We employ a gender-based comparison for last names. If persons to be compared are males, then we compare their last names directly. If persons to be compared are females, then we compare their last names, their fathers' and husbands' last names if they exist. We use the highest matching score out of the three comparisons. If persons' last names match exactly, then the weight equals to 10.

Birthplace: We compare the geocoded birthplace of persons' parents and spouses. Birth places of persons are the same since they are in the same block.

- If persons' fathers' birthplaces match, then the weight equals to 2.
- If persons' mothers' birthplaces match, then the weight equals to 2.
- If persons' spouses' birthplaces match, then the weight equals to 2.

Deathplace: We compare the geocoded deathplace of persons, their parents and spouses.

- If persons' death places match, then the weight equals to 3.
- If persons' fathers' death places match, then the weight equals to 1.
- If persons' mothers' death places match, then the weight equals to 1.
- If persons' spouses' death places match, then the weight equals to 1.

2.2.1. Weight Assignment Algorithm for Fuzzy Matching

Input: *B:* Sorted blocks by birth year. Each person *p* include features: *id:* a person's unique identification number within a tree, *bp*: birthplace, *ge*: gender, *by*: birth year, *ln*: last name, *fn*: first name, *dp*: deathplace, *dy*: death year, *f*: father, m: mother and *S*: spouse list and *tid*: family tree id.

simJW(name1, name2): Returns the Jaro-Winkler similarity of two names
simLV(name1, name2): Returns the Levenshtein similarity of two names

maxSimLVLN(person1, person2): Returns the maximum Levenshtein similarity of two female persons' fathers', mothers' and spouses' last names.

Output: *simPairs*: The hash map of similar pairs of persons.

calculateFuzzyMatchScore(p1, p2): Returns the fuzzy match score between two persons.

getpairKey(p1, p2): Returns the unique comparison key for two persons p1 and p2.

```
initialize simPairs
foreach blockid b ϵ B
    for i = 0 to b.size()
        for j = i + 1 to b.size()
            p1 = b.get(i)
            p2 = b.get(j)
            if p2.by - p1.by > 5
                break
            if simJW(p1.fn, p2.fn) < 0.7 and simJW(p1.ln, p2.ln) < 0.7
                continue
            key = getPairKey(p1, p2)
            score = calculateFuzzyMatchScore(p1, p2)
            if score >= 67
                simPairs.put(key, [p1, p2])

function getPairKey(p1, p2):
    key = “”
    if p1.tid < p2.tid
        if p1.id < p2.id
            key = p1.tid + “_” + p1.id + “_” + p2.tid + “_” + p2.id
        else
            key = p1.tid + “_” + p2.id + “_” + p2.tid + “_” + p1.id
    else
        if p1.id < p2.id
            key = p2.tid + “_” + p1.id + “_” + p1.tid + “_” + p2.id
        else
            key = p2.tid + “_” + p2.id + “_” + p1.tid + “_” + p1.id
    return key

function calculateFuzzyMatchScore(p1, p2):
    // gender and birthplace known within each block:
    score = 25

    //compare birth years:
    score = score + 10 - abs(p2.by – p1.by) * 2
    score = score + 2 - abs(p2.f.by -p1.f.by) * 0.4
    score = score + 2 - abs(p2.m.by - p1.m.by) * 0.4
    score = score + 2 - abs(p2.s.by – p1.s.by) * 0.4

    //compare death years:
    score = score + 3 - abs(p2.dy - p1.dy) * 0.6
    score = score + 1 - abs(p2.f.dy - p1.f.dy) * 0.2
    score = score + 1 - abs(p2.m.dy - p1.m.dy) * 0.2
    score = score + 1 - abs(p2.s.dy - p1.s.dy) * 0.2

    //compare first names:
    score = score + (3 - simLV(p1.fn, p2.fn)) * 3.3
    score = score + (3 - simLV(p1.m.fn, p2.m.fn)) * 2.3
    score = score + (3 - simLV(p1.s.fn, p2.s.fn)) * 2.3
    score = score + (3 - simLV(p1.f.fn, p2.f.fn)) * 2.3

    //compare last names:
    if p1.ge == male
        score = score + (3 - simLV(p1.ln, p2.ln)) * 3.3
    else
        score = score + maxSimLVLN(p1, p2)

    //compare birth places:
    if p1.f.bp == p2.f.bp
        score = score + 2
    if p1.m.bp == m.f.bp
        score = score + 2
    if p1.s.bp == p2.s.bp
        score = score + 2

    //compare deathplaces
    if p1.dp == p2.dp
         score = score + 3
    if p1.f.dp == p2.f.dp
         score = score + 1
    if p1.m.dp == m.f.dp
         score = score + 1
    if p1.s.dp == p2.s.dp
         score = score + 1

    return score

```

2.2.2. Spousal pairs matching algorithm

Given the suspected (candidate) pairs of persons, we applied the spousal pairs matching algorithm to identify the candidate husband-wife pairs that are similar in two trees. For each matching person pair, we first check whether they have spouses, and whether their spouses also have a match score equal to or greater than 67. We then create a hash map of candidate husband-wife pairs with their unique family tree ids. Husband-wife pairs are then used to connect and group trees into tree clusters.

2.2.2. Spousal Pairs Matching and Tree Clustering Algorithm

Input: *simPairs*: The hash map of similar pairs of persons. *personHashByBlock:* Hash map of person hash maps by block. *personID_Block:* Hash map to store block id for each person.

connectedComponents(Graph g): Given a graph g, this method returns all connected components into a list of subgraphs reverse-sorted by the size of nodes in each graph. Thus, the largest cluster is the first subgraph in the list. The largest connected component equals to the input graph g if all nodes are connected.

Output: *shwPairs*: The list of suspected (candidate) husband wife pairs with person ids and tree ids. There could be more than one matching husband-wife pairs for connecting the two trees.

gethwPairsKey: Returns the unique comparison key for husband-wife pairs of person1-spouse1 and person2-spouse2.

treeGraph: The graph of trees in which a node represents a family tree by id, a link represents the connection between two trees that is derived from the matching husband-wife pairs*.*

treeClusters: The set of subgraphs of treeGraph each of which consists of connected trees.

```
initialize shwPairs, treeGraph, treeClusters
foreach pair sp ϵ simPairs
    person1 = p.getValues[0]
    person2 = p.getValues[1]
    tree1 = person1.tid
    tree2 = person2.tid
    spouse1_block = personID_Block.get(person1.s)
    personHashSpouse1 = personHashByBlock.get(spouse1_block)
    spouse1 = personHashSpouse1.get(person1.s)
    spouse2_block = personID_Block.get(person2.s)
    personHashSpouse2 = personHashByBlock.get(spouse2_block)
    spouse2 = personHashSpouse2.get(person2.s)
    keySpouse = getPairKey(spouse1, spouse2)
    if simPairs.get(keySpouse) != null
        keyhwPair = gethwPairsKey(person1, spouse1, person2, spouse2)
        shwPairs.put(hwkey, [tree1, person1, spouse1, tree2, person2, spouse2])
        treeGraph.addLink(tree1, tree2)

treeClusters = connectedComponents(treeGraph)

function gethwPairsKey(person1, spouse1, person2, spouse2):
    keyhwPair = “”
    spousePair1 = “”
    if person1.id < spouse1.id
        spousePair1 = person1.tid + “_” + person1.id + “_” + spouse1.id
    else
        spousePair1 = person1.tid + “_” + spouse1.id + “_” + person1.id

    spousePair2 = “”
    if person2.id < spouse2.id
        spousePair2  = person2.tid + “_” + person2.id + “_” + spouse2.id
    else
        spousePair2 = person2.tid + “_” + spouse2.id + “_” + person2.id

    if person1.tid < person2.tid
        keyhwPair = spousePair1 + “_” + spousePair2
    else
        keyhwPair = spousePair2 + “_” + spousePair1

    return keyhwPair
```

2.3. Tree cleaning and deduplication

The output of the fuzzy match algorithm generates a list of candidate husband-wife pairs that are used to connect the trees and form the tree clusters. During this process, some trees become redundant because all the information in a tree can already exist in a newly formed tree cluster. We clean the trees and remove the duplicates within each tree cluster using the algorithms described in sections 2.3.1, 2.3.2. and 2.3.3. In tree cleaning and deduplication process, we use the cleaned and geocoded trees created through Steps 1-4 in our methodology. In Step 4, we use the algorithm defined in Appendix 2.1. to generate personHashByTree, the data structure that contains a hash map of person objects by each family tree.

2.3.1. Tree cleaning

First, we go through each tree cluster that consists of trees that have matching husband-wife pairs and clean trees within each cluster using a set of rules described below. We first remove persons:

- who do not have any parents, children, or a spouse.
- who only had little or no information (e.g., persons who did not have first name or birth year).
- who have inconsistent temporal information such as a record in which the birth year is greater than the death year or a person's age is greater than 120.
- who have inconsistent links. We classify a link as inconsistent if:
    a) The age difference between a person and his/her spouse was greater than 60.
    b) A person's birth year was less than 12 years of his/her father or mother's birth year.

We then examine and reconstruct the family relationship based on the following two rules:

- A person can have only one father and/or mother. We removed the parental links for persons who had multiple fathers or mothers.
- The parent-child relationship is bidirectional, which means A is listed as a child of B, then B would be one of the parents of A.

2.3.1. Tree Cleaning Algorithm

Input: *personHashByTree:* The hash map of person hash maps by each tree.

removeParent(person, parent: mother or father): Removes parent from person object.

Output: *personHashByTree:* The hash map of persons by trees.

```
foreach treeCluster tc ϵ treeClusters
    foreach tree t ϵ tc
        personHash = personHashByTree.get(t.id)
        foreach person p ϵ personHash
            // if person does not have descendants and spouse
            if (p.f == null and p.m == null) or p.C == null or p.S ==null
                personHash.removePerson(p.id)
                continue
            // if person has little or inconsistent temporal information
            if p.by == null or p.fn == null or p.by > p.dy or abs(p.s.by – p.by) > 60
                personHash.removePerson(p.id)
                continue
            // remove person with inconsistent links
            if (p.by - p.f.by) < 12
                p.f = null
            if (p.by - p.m.by) < 12
                p.m = null
            // if person has multiple fathers or mothers
            if p.f is Array
                p.f = p.f[0]
            if p.m is Array
                p.m = p.m[0]
            // if parent-child relation is bidirectional
            personHashFather = personHashByTree.get(p.f)
            father = personHashFather.get(p.f)
            if p not in father.C
                removeParent(p, father)
            personHashMother = personHashByTree.get(p.m)
            mother = personHashFather.get(p.m)
            if p not in mother.C
                removeParent(p, mother)

```

2.3.2. Iterative tree search for identifying the “true” duplicate spouse pairs

We conducted a relation-based iterative search to identify the “true” duplicate spouse pairs. For each candidate (suspected) husband-wife pair detected in the fuzzy matching process:

1. Check whether the duplicate pairs' parents (i.e., husband's mother and father, and wife's mother and father) are already in the candidate husband-wife pair list.
2. If father-mother pairs are already in the candidate husband-wife pairs list, then classify the candidate husband-wife pair as true (matching) husband-wife pair and skip step 3. If not, continue with step 4.
3. For each candidate duplicate husband-wife pair, compare both mothers' and fathers' information by calculating a score of conflicting information. Given two persons, calculate a score of conflicting information that range between 0 and 1. The score reflects whether the two records are from different persons if the score is above the threshold of 0.3. Unlike the fuzzy matching score, conflict score considers only a few but major features of person records, that are gender, birthplace, death place, birth year, death year, first name and last name. For example, if gender does not match, then the score is 0. We check whether the conflicting scores of mothers' (husband's mother and wife's mother) and fathers' (husband's father and wife's father) exceed the predefined threshold of 0.3. If the score is below the threshold for at least one of the parents' comparisons, then classify the candidate husband-wife pair as true (matching) husband-wife pair.
4. Check whether the duplicate pairs' child (ren) has/have spouses and whether at least one of the child-spouse pairs is already in the candidate husband-wife pair list. Classify the candidate duplicate child-spouse pairs as true (matching) husband-wife pair and skip step 5. If not, continue with step 5.
5. Calculate the conflict score calculation to each child-spouse pairs if they were not already in the candidate list of husband-wife pairs. Add the child-spouse pairs into the suspect duplicate husband-wife lists if there is no conflict information.

2.3.2. Iterative Tree Search for Identifying the "True" Matching Spouse Pairs

Input: *shwPairs*: The list of suspected (candidate) husband-wife pairs with person ids and tree ids.

shwPairsVisited: The list of visited pairs.

personHashByTree: The hash map of persons by trees.

maxSimJWLN(person1, person2): Returns the maximum Jaro-Winkler similarity of two female persons' fathers', mothers' and spouses' last names.

getpairKey(p1, p2): Returns the unique comparison key for two persons p1 and p2.

gethwPairsKey(p1, s1, p2, s2): Returns the unique comparison key for husband-wife pairs of person1-spouse1 and person2-spouse2.

treeClusters: The set of subgraphs of treeGraph each of which consists of connected trees.

Output: *hwPairs*: The list of true duplicate husband-wife pairs with person ids and tree ids.

calculateConflictScore(p1, p2): Given two persons, this function returns a score between 0 and 100, which reflects whether the two records are different persons. Conflict score is based on gender, birthplace, death place, birth year, death year, first name and last name.

conflictScoreHash: The hash map to store a pair of persons with their conflict score.

```
initialize hwPairs
while shwPairs.length > 0
    shw = shwPairs.next()
    if shw.key in shwPairsVisited
        continue
    shwPairsVisited.add(shw.key)
    tree1 = shw[0]
    p1 = shw[1]
    s1 = shw[2]
    tree2 = shw[3]
    p2 = shw[4]
    s2 = shw[5]
    p1_p2_key = getPairKey(p1, p2)
    s1_s2_key = getPairKey(s1, s2)

    // parents of persons p1 and p2
    personHashTree1 = personHashByTree.get(tree1)
    f1 = personHashTree1.get(p1.f)
    m1 = personHashTree1.get(p1.m)
    personHashTree2 = personHashByTree.get(tree2)
    f2 = personHashTree2.get(p2.f)
    m2 = personHashTree2.get(p2.m)
```

```
keyParents = *gethwPairsKey*(f1, m1, f2, m2)
// parents of spouses s1 and s2
fs1 = personHashTree1.get(s1.f)
ms1 = personHashTree1.get(s1.m)
fs2 = personHashTree2.get(s2.f)
ms2 = personHashTree2.get(s2.m)
keySParents = *gethwPairsKey*(fs1, ms1, fs2, ms2)

hwTruePair = false
if *shwPairs*.get(keyParents) != null and *shwPairs*.get(keySParents) != null
    hwTruePair = true
else
    // check the fathers
    f1_f2_key =  *getPairKey*(f1, f2)
    score_f1f2 = *conflictScoreHash*.get(f1_f2_key)
    if score_f1f2 == null
        score_f1_f2 = *calculateConflictScore(f1, f2)*
        *conflictScoreHash*.put(f1_f2_key, score_f1_f2)
    // check the mothers
    m1_m2_key =  *getPairKey*(m1, m2)
    score_m1m2 = *conflictScoreHash*.get(m1_m2_key)
    if score_m1m2 == null
        score_m1_m2 = *calculateConflictScore(m1, m2)*
        *conflictScoreHash*.put(m1_m2_key, score_m1_m2)

    // check the fathers of the spouses
    fs1_fs2_key =  *getPairKey*(fs1, fs2)
    score_fs1_fs2 = *conflictScoreHash*.get(fs1_fs2_key)
    if score_fs1_fs2 == null
        score_fs1_fs2 = *calculateConflictScore(fs1, fs2)*
        *conflictScoreHash*.put(fs1_fs2_key, score_fs1_fs2)
    // check the mothers of the spouses
    ms1_ms2_key =  *getPairKey*(ms1, ms2)
    score_ms1ms2 = *conflictScoreHash*.get(ms1_ms2_key)
    if score_ms1ms2 == null
        score_ms1_ms2 = *calculateConflictScore(ms1, ms2)*
        *conflictScoreHash*.put(ms1_ms2_key, score_ms1_ms2)

    // check the conflict scores to determine the true matches
    if score_f1_f2 <= 0.3 or score_m1m2 <= 0.3 or
    score_fs1_fs2 <= 0.3 or score_ms1_ms2 <= 0.3
        hwTruePair = true
        *shwPairs*.put(keyParents, [tree1, f1, m1, tree2, f2, m2])
        *shwPairs*.put(keySParents, [tree1, fs1, ms1, tree2, fs2, ms2])

if hwTruePair == true
    *hwPairs*.put(shw.key(), shw)
    *hwPairs*.put(keyParents, [tree1, f1, m1, tree2, f2, m2])
```

```
            hwPairs.put(keySParents, [tree1, fs1, ms1, tree2, fs2, ms2])
        // check whether the child-spouse pairs are already in the suspected
        // husband-wife pairs, shwPairs. If not, add each of the child-spouse pairs to
        // hwPairs if there is no conflict information
        children1 = p1.C
        children2 = p2.C
        foreach child1 ϵ children1
            if child1.s == null
                continue
            else
                foreach child2 ϵ children2
                    if child2.s !=null
                        c1 = personHashTree1.get(child1.id)
                        c2 = personHashTree2.get(child2.id)
                        c1_c2_key =  getPairKey(c1, c2)
                        score_c1c2 = conflictScoreHash.get(c1_c2_key)
                        if score_c1c2 == null
                            score_c1_c2 = calculateConflictScore(c1, c2)
                            conflictScoreHash.put(c1_c2_key, score_c1_c2)
                        if score_c1c2 > 0.3
                            continue

                        cs1 = personHashTree1.get(c1.s)
                        cs2 = personHashTree2.get(c2.s)
                        cs1_cs2_key =  getPairKey(cs1, cs2)
                        score_cs1cs2 = conflictScoreHash.get(cs1_cs2_key)
                        if score_cs1cs2 == null
                            score_cs1_cs2 = calculateConflictScore(cs1, cs2)
                            conflictScoreHash.put(cs1_cs2_key, score_cs1_cs2)
                        if score_cs1cs2 <= 0.3
                            keySC= gethwPairsKey(c1, c2, cs1, cs2)
                            if shwPairs.get(keySC) != null
                                shwPairs.put(keySC, [c1.tid, c1, cs1, c2.tid, c2, cs2])
                           hwPairs.put(keySC, [c1.tid, c1, cs1, c2.tid, c2, cs2])

    function calculateConflictScore(p1, p2):
        conflictScore = 0
        if p1.gender != p2.gender
            return conflictScore
        if p1.bp != p2.bp
            conflictScore = 0.3
        if p1.dp != p2.dp
            conflictScore = conflictScore + 0.2

        birthyearAbsDif = abs(p1.by - p2.by)
        if birthyearAbsDif <= 5
            conflictScore = conflictScore + birthyearAbsDif * 0.04
        else
```

```
            conflictScore = conflictScore + 0.2

        deathyearAbsDif = abs(p1.by - p2.by)
        if deathyearAbsDif <= 5
            conflictScore = conflictScore + birthyearAbsDif * 0.04
        else
            conflictScore = conflictScore + 0.2

        nameSimilarity = simJW(p1.fn, p2.fn)
        if nameSimilarity < 0.7
            conflictScore = conflictScore + 0.2
        else
            conflictScore = conflictScore + (1- nameSimilarity) * 0.67

        if p1.ge == male
            lastnameSimilarity = simJW(p1.ln, p2.ln)
        else
            lastnameSimilarity = maxSimJWLN(p1.ln, p2.ln)
        if lastnameSimilarity < 0.7
                conflictScore = conflictScore + 0.2
            else
                conflictScore = conflictScore + (1- lastnameSimilarity) * 0.67

        if p1.bp != p2.bp
            conflictScore = conflictScore + 0.2
        if p1.bp != p2.bp
            conflictScore = conflictScore + 0.2
        return conflictScore

```

2.3.3. Identifying representative person identification (id) numbers

We extract the representative person from the duplicates based on the amount of information. If two records were classified as duplicates, the record with more known information (e.g., gender, first name, last name, birthplace, death place, birth year, death year, father, mother, spouse, and children) was selected as representative person. If the two or more records have the same amount information, then we randomly chose one of the records as the representative person. We use the representative person to substitute other duplicate person records. Going through the true duplicate husband-wife pair list, we remove each duplicate person record until there are no more duplicate pairs.

2.3.3. Identifying representative person identification (id) numbers

Input: *hwPairs*: a list of true husband wife pairs with person ids and tree ids.

Output: *personHashByTree:* Removed duplicates from person hash maps by tree.

representativeIDHash: The hash map of each person id to a unique representative id. All person ids that point to the same representative ids are merged, and thus duplicates are removed.

duplicatesGraph: The graph in which each node is a person id, and a link is the connection between two persons, which means the two persons are the same individual.

duplicatesList: The list that contains lists of person ids that refer to the same person. *duplicatesList* is created by using *connectedComponents* function on *duplicatesGraph.*

connectedComponents(Graph g): Given a graph g, this method returns all connected components into a list of subgraphs reverse-sorted by the size of nodes in each graph. Thus, the largest cluster is the first subgraph in the list. The largest connected component equals to the input graph g if all nodes are connected.

calculateInformationScore(person): Returns a score that reflects the number of available features in a person record (i.e., gender, first name, last name, birthplace, death place, birth year, death year, father, mother, spouse and children). Each feature counts as one, and the score ranges between 0 and 11 (the total number of features).

```
initialize duplicatesGraph
foreach pair hw ϵ hwPairs
    tree1 = hw[0]
    person1 = hw[1]
    spouse1 = hw[2]
    tree2 = hw[3]
    person2 = hw[4]
    spouse2 = hw[5]
    duplicatesGraph.addLink(person1, person2)
    duplicatesGraph.addLink(spouse1, spouse2)
duplicatesList = connectedComponents(duplicatesGraph)

// assign representative id based on the amount of information
foreach duplicates d ϵ duplicatesList
    max_score = 0
    representativeID = -1
    foreach person p ϵ d
        p_score = calculateInformationScore(person)
        if p_score > max_score
            max_score = p_score
            representativeID = p.id
    foreach person p ϵ d
        representativeIDHash.put(person.id, representativeID)

```

```
// remove duplicate records
foreach treeCluster tc ϵ treeClusters
    foreach tree t ϵ tc
        personHash = personHashByTree.get(t.id)
        foreach person p ϵ personHash
            representativeID = representativeIDHash.get(person.id)
            if representativeID == null
                representativeIDHash.put(person.id, person.id)
            else
                personHash.remove(person)
                continue

function calculateInformationScore(person):
    score = 0
    if person.ge !=null
        score++
    if person.ln !=null
        score++
    if person.fn !=null
        score++
    if person.by !=null
        score++
    if person.dy !=null
        score++
    if person.bp !=null
        score++
    if person.dp !=null
        score++
    if person.m !=null
        score++
    if person.f !=null
        score++
    if person.S !=null
        score++
    if person.C !=null
        score++
    return score

```

3. Regression Results

We chose the state level population proportion (the number of individuals alive in the U.S. in 1880 in family trees divided by the number of individuals in 1880 Census) as the dependent variable. Our candidate independent variables were state level percentage of following population segments: white, farmer, individuals greater than 54 years old, male, individual's birthplace in the same state as the 1880 location, and individual's birthplace in foreign countries. The percentage of individual's birthplace in the same state as the 1880 location equals to the number of individuals born in the same state as the 1880 locations divided by the total population in 1880 in each state. The percentage of individual's birthplace in foreign countries equals to the number of individuals born in the other countries divided by the total population in 1880 in each state. Figure 3.1 illustrates the dependent variable and independent variables by states. The family trees contain a higher proportion of the population in the Middle and Eastern states of West Virginia, Indiana, Kentucky, Tennessee and Arkansas. It is interesting that in the South where there was a large Black population, the proportion of the population in trees is less than elsewhere and other states which are less well represented in trees have high proportions of foreign born. There were relatively more white and foreign born populations in the upper Eastern U.S. There were relatively more farmers in the middle U.S.

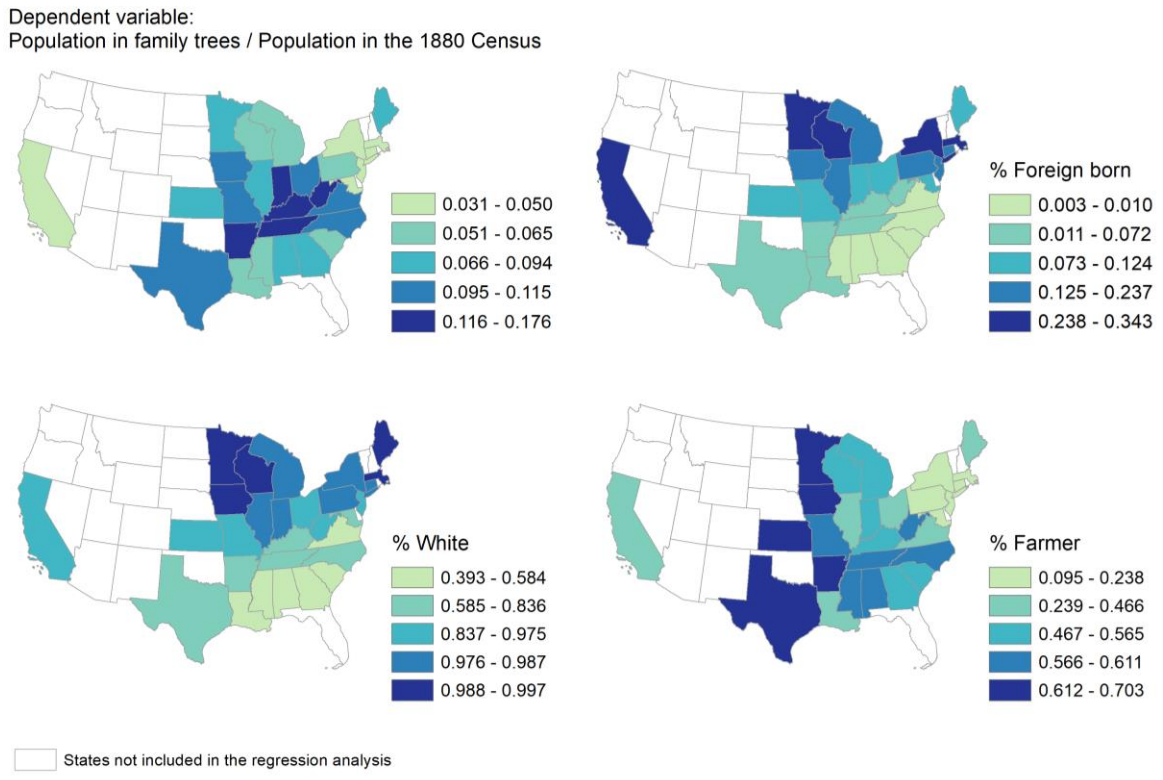


Figure 3.1. Dependent variable: 1880 family tree population / 1880 Census population and independent demographic variables.

Table 3.1 shows the regression result. The coefficients of the males and individuals greater than 54 years old were not significant. When we removed these two variables from the model, the coefficient of the percentage of individuals' birthplaces in the same state became not significant. Thus, we excluded the birthplace variable from the model, the coefficient of all independent variables became significant, see Table 3.2.

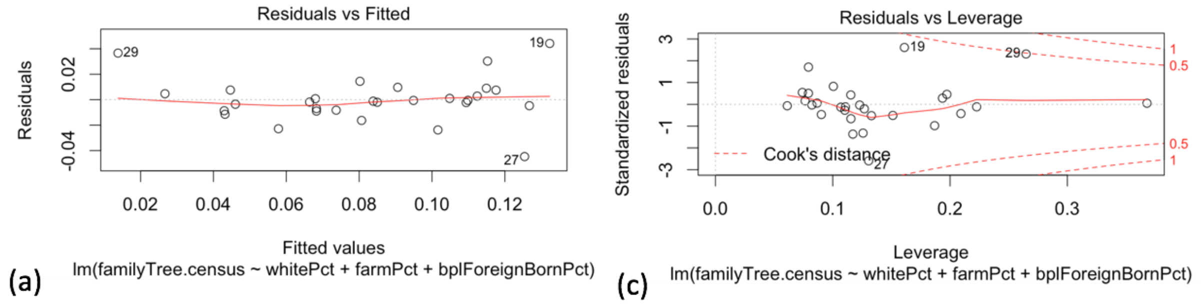
Table 3.1 Regression result 1.

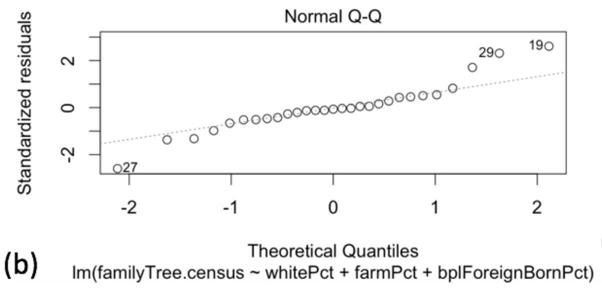
Variable	Coefficients	Standard error	t value	Probability (> |t|)
Intercept	-0.28660	0.13224	-2.167	0.041310*
White percentage	0.14958	0.02720	5.500	1.58e-05***
Farm Percentage	0.10380	0.02717	3.821	0.000933***
Birthplace in the same state percentage	0.09339	0.03653	2.557	0.017983*
Birthplace in foreign country percentage	-0.19319	0.06245	-3.094	0.005304**
Age more than 54 percentage	-0.44886	0.26691	-1.682	0.106767
Male percentage	0.38135	0.22802	1.672	0.108593
Other model performance parameters				
Multiple R-squared	0.8386			
Adjusted R-squared	0.7946			
F statistics	19.05 on 6 and 22 DF, p-value: 1.097e-07			

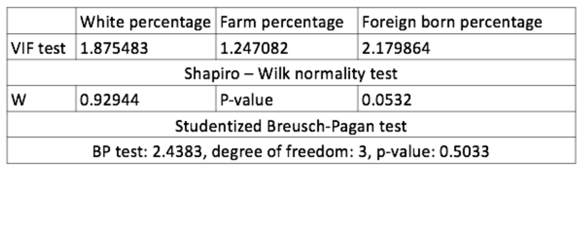
Table 3.2. Regression result 2.

Variable	Coefficients	Standard error	t value	Probability (> |t|)
Intercept	-0.03018	0.01991	-1.516	0.14216
White percentage	0.10376	0.02359	4.399	0.000177***
Farm Percentage	0.11858	0.02214	5.357	1.48e-05***
Birthplace in foreign country percentage	-0.22897	0.04790	-4.780	6.60e-05***
Other model performance parameters				
Multiple R-squared	0.7687			
Adjusted R-squared	0.7409			
F statistics	27.69 on 3 and 25 DF, p-value: 4.111e-08			

We conducted further analyses to test whether the four assumptions of linear regression were satisfied (Figure 3.2). The residuals versus fitted values plot (Figure 3.2.a) showed no obvious sign of deviation in the residuals. The Shapiro normality test showed that the residuals were normally distributed at the 0.05 significance level. The studentized Breusch-Pagan test showed that the residuals had equal variance. The residuals versus leverage plot (Figure 3.2.b) indicates there were no influential cases. In addition, the multi-collinearity check showed that all variance inflation factors were less than 10. Therefore, multi-collinearity was not an issue. These test results showed that the linear regression was robust, and all model assumptions were met (Figure 3.2.c and d).







	White percentage	Farm percentage	Foreign born percentage
VIF test	1.875483	1.247082	2.179864
Shapiro – Wilk normality test			
W	0.92944	P-value	0.0532
Studentized Breusch-Pagan test			
BP test: 2.4383, degree of freedom: 3, p-value: 0.5033			

(d)

Figure 3.2. Tests for evaluating the regression model and results.