Katie Hippe
MATH 381
December 1, 2023

Etymologies Project

I. Problem Description

I chose to model the etymological pathways words can take on their journeys from Latin to English. Despite English being a Germanic language, many words ultimately come from Latin roots due to various conquests and influences from Latin being the main language of academia throughout Europe. In modern English, many of the grammatical influences have fallen out of fashion (as the two languages aren't very similar grammatically) but the words have remained, especially in the field of science, medicine, and law. Morphologically, Latin can be more complicated than English, as nouns are declined between five patterns, three genders, and seven declensions, and verbs can have over two hundred different forms, depending on person, tense, voice, mood, and more. When transferring from Latin to English, certain stems and forms are used for different purposes, but this makes it difficult to predict which ones will make it through and in what ways they'll be used in English.

In many cases, Latin has already done the hard work for us in transferring over to English. For example, we can trace from root verb *cedo, cedere, cedi, cessus*, combined with prefix *ad-* to become *accedo, accedere, accede, accessus*, then from the fourth principal part, has a variety of endings that give meaning. We can add the adjectival *-ibilis* suffix to get *accessibilis*, giving English adjective 'accessible' and from there, noun 'accessibility.' Or, we could add a noun suffix to get *accessio, accessionis* and from there, English 'accession.' We could isolate the stem, giving the English noun 'access.' For a verb, we can take from the second principal part *accedere* and use the stem plus 'e' on the end to follow English morphology and get 'accede.' We could add a noun suffix to get *accessor, accessoris*, and get English 'accessory' or 'accessor.' So from just one verb, already five different pathways to seven English words, each coming from a different modification in the original Latin.

So how best to model these transitions? Since such similar words exist already in Latin, it made the most sense to me to start with a collection of the various Latin words and focus on the specific transitions to English. Instead of jumping from *accessus*, I collected the different Latin extensions *accessibilis, accessio,* and *accessor* and compared them directly to the potential English counterparts, such that no middle steps would be cut out. That is, at least within Latin. Due to language spreading across Europe, and the unfortunate existence of France in between Italy and England, many English words came by way of French, causing significant spelling and pronunciation differences. Other words contain Latin stems but were modified via English patterns already in existence, like 'accessorize,' or different inflected forms of nouns and verbs, like 'accessories' and 'accedes,' 'acceded,' or 'acceding.' Since there are so many factors at play, it's hard to create an accurate and succinct model.

II. Description of Model

My model focuses on a very small and specific transition between Latin and English. I chose to start with certain forms of Latin nouns, verbs, adjectives, and adverbs (conjunctions and other parts of speech were less interesting and tended to be passed down verbatim or not at all) and set them

right next to the closest English descendent. This way, I could see the exact changes between words. From our earlier example, instead of comparing *cedo* with 'accessibility,' I compared the much closer *accessibilis* and 'accessibility.' Even further, I only compared the nominative singular form of any noun, with added constraint masculine for any adjective or participle (or the genitive if the stem wasn't accurately described by the nominative), as English doesn't differentiate between cases for any nouns except pronouns. While this removes a lot of nuances from the original Latin, it reduces the complexity of the model.

When making these pairs, I separated them up into prefix, stem, suffix, and/or ending, as well as any inner pieces caused by compounding words, then split up the corresponding English to match. For example, *accessibilis* became *ac-cess-ibilis* matched with 'ac-cess-ible,' where *cess* remains isolated as the original stem instead of *a-ccess*, where the extra *c* is caused by assimilation of the prefix. So, my dictionary contains occasionally arbitrary choices I made of where to split up words, but hopefully with enough variety that it covers many user inputs. One could split the *-ibilis* suffix further into *-ibil-is*, but I felt that would decrease accuracy in the model, as there are certainly third declension nouns that drop the *-is* ending in their journey to English, but you would never see an English noun ending in '-ibl.' There is certainly bias in the model, but to describe in greater detail, below is a section from the dictionary:

male-vol-entia male-vol-ence	mira-culum mira-cle
minus-cula minus-cule	miser-ia miser-y
medit-atio medit-ation	modern-us modern-
memor-abilis memor-able	moder-atus moder-ate
merc-es merc-y	moment-um moment-
mem-ento mem-ento	morbid-us morbid-
miser-abilis miser-able	multi-plic-atio multi-plic-ation

To make the model easier, I made sure each component of the Latin word matched up with the corresponding English, even if that meant matching an ending to a blank space, like in the example of *modern-us* to 'modern- .' Note that the full dictionary is not included in this report, but I could provide it if wanted.

My code was designed to handle any input file that was organized in this specific way, so it could be used to analyze other language transfer processes. Each line was taken at a time, split up into the Latin and English, then each component of the Latin word was separated out and matched with the corresponding component in the English. I then made the transition matrix in a bit of a roundabout way: first, all of the components were put into a map from the original Latin to a list of all possible English correspondents, including repeats. Then I took each list, counted occurrences, and divided them by the length of the list to get the decimal probability of each change, and created a new map from the original Latin to a map of English possibilities to their respective probabilities. The generative process took user input for a Latin word, split up into its components ahead of time by the user (and thus doesn't account for user error or malicious intent) and for each component, found the entry in the map if it existed. I generated a random number between 0.0 and 1.0 and, for each possibility in the nested map, added that probability to a counter variable until the random number was strictly less than the counter variable. Whichever ending triggered this condition would be added to the output word.

Below is the code (somewhat squished to fit this document). Please note that while it is functional for the purposes of this project, there are many ways the code quality could be improved.

```java
// this class analyzes a set of mappings between words of different languages and creates
// connections, allowing us to input our own word in one language and use probabilities
// to determine a likely transition
import java.util.*;
import java.io.*;

public class Main {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner console = new Scanner(System.in);
        Random rand = new Random();

        System.out.println("Welcome! Press any key to begin. To exit, press 'E'");
        Map<String, Map<String, Double>> diachrons = new TreeMap<>();
        createMatrix(diachrons);
        String choice = console.nextLine();

        while (!choice.equalsIgnoreCase("E")) {
            System.out.println("Enter a latin root, following conventions:");
            choice = console.nextLine();
            if (!choice.equalsIgnoreCase("E")) {
                testWord(diachrons, rand, choice);
            }
        }
    }

    // helper method that creates a map of each possible Latin to component to all
English possibilities
    public static void createMap(Map<String, List<String>> transitions) {
        try {
            File inFile = new File("./resources/words.txt"); // hard coded input file
            Scanner s = new Scanner(inFile);

            while (s.hasNext()) {
                // cuts up the line into two words
                String long_line = s.nextLine().trim();
                int space = long_line.indexOf(" ");
                String long_latin = long_line.substring(0, space);
                String long_english = long_line.substring(space + 1);

                // compares each syllable and makes a new mapping if applicable
                boolean hasNextComponent = true;
                String short_latin = long_latin; //initializers
                String short_english = long_english;

                while (hasNextComponent) {
                    if (long_latin.contains("-")) { // if more syllables to look at
                        int dash = long_latin.indexOf("-");
                        short_latin = long_latin.substring(0, dash);

                        int eng_dash = long_english.indexOf("-");
                        short_english = long_english.substring(0, eng_dash);

                        long_latin = long_latin.substring(dash+1);
                        long_english = long_english.substring(eng_dash+1);
```

```
                    } else {
                        short_latin = long_latin;
                        short_english = long_english;
                        hasNextComponent = false; // reached end of the word
                    }

                    // make connections
                    if (!transitions.containsKey(short_latin)) { //create new mapping
                        transitions.put(short_latin, new ArrayList<>());
                    }
                    transitions.get(short_latin).add(short_english);
                }
            }

        } catch (FileNotFoundException e){
            //ERROR crash program
            System.out.println("Error: " + e);
            e.printStackTrace();
            System.exit(1);
        }
    }

    // creates a transition matrix (one sided) from a Latin component to possible English
components
    public static void createMatrix(Map<String, Map<String, Double>> diachrons) {
        Map<String, List<String>> transitions = new TreeMap<>();
        createMap(transitions);

        for (String key : transitions.keySet()) {
            Map<String, Double> counts = new TreeMap<>();

            // organizes all possible transitions to get counts
            for (int i = 0; i < transitions.get(key).size(); i++) {
                String option = transitions.get(key).get(i);
                if (!counts.containsKey(option)) {
                    counts.put(option, 1.0);
                } else {
                    counts.put(option, counts.get(option) + 1);
                }
            }
            Map<String, Double> percents = new TreeMap<>();
            for (String nested_key : counts.keySet()) { // divide counts by total for
percents
                double percent = counts.get(nested_key) / transitions.get(key).size();
                percents.put(nested_key, percent);
                diachrons.put(key, percents);
            }
        }

        // System.out.println(diachrons); //(uncomment this to see what the transition
probabilities look like!)
    }

    // given a word, splits it up into components and checks whether we have a mapping,
then
    // changes the component based on transition probability
    public static void testWord(Map<String, Map<String, Double>> diachrons, Random rand,
```

```
String root) {
        StringBuilder leaf = new StringBuilder();

        while (root.contains("-")) {
            String short_root = root.substring(0, root.indexOf("-"));
            root = root.substring(root.indexOf("-")+1);

            if (diachrons.containsKey(short_root)) { // if we have a mapping, use it!
                leaf.append(changeComponent(diachrons.get(short_root), rand));
            } else {
                leaf.append(short_root); // otherwise word remains unchanged
            }
        }

        // repeat above for final part of word, not included in while loop
        if (diachrons.containsKey(root)) {
            leaf.append(changeComponent(diachrons.get(root), rand));
        } else {
            leaf.append(root);
        }

        System.out.println("English word: " + leaf);
        System.out.println();
    }

    // helper method that figures out how to change components based on transition matrix
    public static String changeComponent(Map<String, Double> row, Random rand) {
        double trans = rand.nextDouble();
        double placehold = 0.0;

         for (String key : row.keySet()) {
             placehold += row.get(key);
             if (trans < placehold) {
                 return key;
             }
         }

         return ""; // if something went wrong
    }
}
```

III. Model Experiments

I tested the model first by checking words that were already in the dictionary and seeing what I came up with. Because I made the dictionary, it's biased to the words I already know, with patterns I came into this project knowing, so there's many more words that follow a certain pattern. For some examples, the input *male-vol-entia*, which corresponds to English 'malevolence,' became 'malevolential,' because there also exists a mapping between *con-fid-entia* and 'confidential.' This word doesn't already exist in English (probably just equivalent to malevolent), but now we have a pretentious way of saying something wishes ill intent. I tested *modern-us*, which became 'modernal,' which I thought was funny because *-us* only has a 1.9% chance of becoming 'al.' I put *modern-us* through a second time and got the existing English equivalent 'modern.' I tested *class-icus* and got 'classic,' but could have gotten 'classical,' also an existing English word.

From there, I looked up a list of English words derived from Latin and tested if my model could predict them accurately. The first, *agend-a*, doesn't change and becomes English 'agenda.' However, my model only preserves the final *a* 18% of the time, so produced 'agend.' When I put in *clam-or*, my model produced 'clame,' as the *or* ending had a 50/50 chance of not changing or changing to an 'e.' *Atroc-itas* became 'atrocity' with 100% certainty, as all of my input examples switched the *-itas* ending to '-ity.' *Fer-ox* also became 'ferocious' with 100% certainty, which was fun as the *-ox* ending comes from a PIE suffix meaning 'looking' or 'appearing,' and is anglicized very similar to *-osus*→'-ious,' a suffix meaning 'full of.' *Con-glomer-atus* also became 'conglomerate' with nearly 100% certainty. However, *dirig-ere* can only become 'dirig,' 'dirige,' or 'dirigion,' instead of the actual 'dirigible' I was trying to get. None of the example in my dictionary behave like this, because the '-ible' ending comes from French modifications, so I added that example to my dictionary and now had a small chance of getting the existing transition, better than the previous zero. *Fut-ilis* became 'futilian,' equivalent in meaning to the intended 'futile.'

Many Latin-to-English transitions behave in this extremely predictable manner, which isn't as interesting. When making the dictionary, I found myself writing again and again words ending with *-tio* becoming 'tion' in English, or *-tas* becoming 'ty' (with various assimilation vowels added). While I was glad that my model reflected this accurately, the more interesting transitions, such as the nine different possibilities that *-us* could become, were often wrong. The generated words were similar enough to English that you could use them and be understood, but rarely did I get the exact correct transition when working with ambiguous components. This makes sense considering my model works off of probabilities, so if more words act a certain way, then this one exception isn't going to be more likely to act as an exception, but I wonder if there's more to the word itself that a more complicated model could capture to be more accurate.

So, satisfied that the model was working as well as I wanted it to, I went directly to the Latin to create some new English words. This part was fun, and as language is malleable, I'm hoping some of these will catch on.

From Latin *madido, madidare, madidavi, madidatus*, we get 'madidate' from the fourth principal part, meaning to make wet or moist, or in a particular sense, to make drunk or intoxicate. I split up the Latin to be *madid-atus*, as *madid* is the common stem that I wanted to preserve. So, one could say, about an eventful night out, that they got incredibly madidated, and in using this word, everyone would think that they still were.

From *maculatio, maculationis*, meaning a spot or a stain, we can create ‘maculation.’ (I looked this up and learned that this is in fact already a word, and according to usage data there’s been a strong uptick since 2019 in its popularity.)

From *feteo, fetere*, meaning to stink or smell bad, we get ‘fete,’ a verb meaning the same thing (we can compare this to the already existing ‘fetid,’ which comes from a slightly different route but similar root words). For fun, we can pretend this verb acts as normally as an English verb can and could tell someone they are feting up a classroom. The word ‘fete’ already exists in English as a noun meaning a celebration, or a verb meaning to honor or entertain lavishly, which makes for an interesting homonym.

For an especially committed historical reenactor, *fustibalator, fustibalatoris* yields English fustibalator, one who fights with the sling-staff. We might compare this with Shakespearean ‘fustilarian,’ a now-obsolete noun describing someone who uses a cudgel instead of a sword, while implies lowly or commoner status. A fustibalus is a (relatively) primitive weapon, but because it hurls heavy rocks at high speeds, it was an especially destructive weapon, so it’s hard to use it in a similarly insulting manner.

Finally, from *sarcinosus, -a, -um*, we get sarcinous, meaning heavily burdened, an excellent adjective for current students. We could compare this with existing ‘onerous,’ which has a similar *-osus* to ‘ous’ pipeline, or ‘encumber,’ also from Latin but with a significant French detour. In Latin, *sarcinosus* describes something specifically heavily burdened, as opposed to *onerosus* merely meaning burdened, so the main advantage of this word would be to one-up others.

IV. Strengths and Weaknesses

This model words well for very specific transitions between Latin and English, and only because of how much of the heavy lifting is already done by the Latin. While the dictionary I created contains about 500 different examples, and I tried to get a variety of words with different transitions, there are certainly many that I missed. Additionally, the model only examines component to component, without regard for the rest of the word. In English, we often have a silent ‘e’ at the end of a word, like in ‘genuine,’ which comes from *genuinus*. However, there were many more examples in my dictionary, like in the section above *modernus* to ‘modern’ where the *-us* ending was removed, than when it was replaced with an ‘e,’ so my code tended to anglicize *genuinus* merely as ‘genuin.’ Another example where this structure fails would be *multiplicare*, from which we get English ‘multiply.’ To keep my model consistent, I would split the Latin into *multi-plic-are*, which would correspond to ‘multi-ply- .’ Thus we have created a link between *plic-* ply and *are-* , but there are other examples in my dictionary where *are* doesn’t correspond to a blank, so I could put back *multi-plic-are* into my code and get ‘multiply+garbage.’ There are other examples of a dropped suffix and change in stem, so the translation can get garbled as the code doesn’t know that we’ve stopped. Another change is the *i* to ‘j’ change that occurs when *i* acts a consonant instead of a verb. My model correctly changes *iect* to ‘ject,’ like in ‘eject,’ but wouldn’t necessarily get this letter switch outside of the examples in the dictionary.

The code itself could also be improved, like making sure the methods aren’t doing too much individually and improving the user interface. The user experience also is only worthwhile if it’s me or someone who understands the structure I’ve created, as the decisions for where to split up words needs to be consistent throughout both the user’s dictionary and their input Latin. The

dictionary that the program uses is hard-coded in my program, which doesn't allow the user themselves to change it if they wanted to try their own. Also, the code to make the transition matrix relies on the dictionary being complete before it's analyzed. If I had more time, I would want to figure out a method to add another line to the dictionary and thus more information to the existing transition matrix without having to recreate it from scratch.

This model has lots of potential to improve, but works reasonably well within a small, detailed input structure. In the future, I would want to expand the dictionary to include some weirder examples with more change from Latin to English, as well as a method that might generate possible inflected forms of the generated English words. Considering how irregular our conjugation tends to be, that task could be its own project. Additionally, this project has already whittled down the potential Latin inputs to the ones that I thought would be most interesting, so it could be expanded to include other parts of speech and words. And finally, I used Classical Latin and modern English, without considering the routes through Late Latin, French, and other languages that words could go through, so it might be interesting to make another similar model from Latin to French, then French to English and see what we come up with. In some cases, it would certainly make the model more accurate.