COMP 402 Project Proposal

**Abstract**

The difficulties in learning a language today are numerous. Many would not be willing or able to spend the time or money on effective courses. That leaves more impersonal, nontraditional forms of learning, such mobile and computer applications. Due to the limitations of learning from a computerized system, there can be an overreliance on memorization of a limited set of phrases. This can be exacerbated by the language to be learned being niche, such as dead languages like Latin and languages that have few native or fluent speakers from whom to learn. However, this need not be the unfortunate reality. Through natural language processing, a field that has been quickly growing since its inception, computer programs have further capability to work with natural language input, a lofty feat considering that many programs would otherwise work on specific string data. The use of natural language processing would allow users of an application like the one described in this paper to more naturally interact with the application in ways that would otherwise be limited or nonexistent in their possibility.

To this end, the application described in this paper would encourage the user to gain further understanding of the Latin language by teaching them through composition. That is, by practicing writing original sentences, the user should be able to gain a higher level of understanding than what would be obtained through simple memorization.

By means of the various models available to machine learning programs, the GUI support of Python’s Tkinter library, and many other features, this program will be able to analyze the grammatical structure of a user’s Latin sentence translated from English, and be able to make any errors in the translated sentence known to the user for improvement. The application will also be able to have simplified lessons on Latin grammar available to the user for learning purposes.

The project will be done in stages throughout the course of the spring semester, following the structure of COMP 402’s structure of biweekly checkpoints. This will be done by gradually building machine learning models to process certain sections of the available data. For example, for the first checkpoint, the machine learning models that are capable of tagging a word’s part of speech and collecting a noun/adjective’s information will be developed.

**Background and Introduction**

Within the field of machine learning, there exists a subsection of study known as natural language processing, often abbreviated to NLP. The purpose of NLP is, as the name suggests, to train models to interact with natural linguistic structures, like the sentences a person would say or write. Over the years since NLP’s inception, machines have been trained to perform numerous tasks to interact with users, such as voice recognition and activation, production of seemingly natural and semantically correct sentences, and parsing natural language input from users for various purposes, such as detection of errors or categorizing articles by semantic themes.

The project currently underdevelopment was inspired by the similar projects of both the Natural Language Toolkit (NLTK) and, more directly, the Classical Language Toolkit (CLTK). The project is meant to also compete with the various tools for learning languages that can be found out in the market today, differing in that this project will focus on the composition of Latin sentences rather than mostly rely on the memorization of a small collection of curated phrases. A result of overreliance on memorization of phrases in a good number of the easily accessible applications in the market leaves a great deal to be desired when it comes to true understanding of the language. While it is true that the memorization that many other applications provide is useful in some regards, such as for learning vocabulary, the shortcomings still remain, such as not being able to test for proficiency. [1, 2] Composition, on the other hand, by typing and writing out sentences encourages one to understand the underlying grammar and structure of the language, and doing so would promote fluency through practice.

From the influence of the CLTK in particular, it is mostly in the form of an attempt to improve upon the experiment part of speech tagger that is presently available for the Latin language by building a new set of models to compete with it. [3] In addition to this, and to apply these attempts to a reasonable application, the idea came that the new models should be sufficient enough to be able to impart useful information to even beginning users of the Latin language.

**Description and Analysis**

This program will be available as a desktop application, as (if it is allowable to be honest) the author is currently inexperienced with web and mobile application development. Still, seeing as how most people in the modern world have, at least, access to a traditional computer in some way, such as through a laptop or desktop computer, this should not impact the availability of the application, though it will decrease the ease with which users will be able to access it in passing amounts of time. That being said, for the purposes of this capstone project, the application is not meant to reach a finalized, ready-for-market state at the end of the semester. It should be noted that it could indeed be possible to have the trained model (the state in which the model no longer needs the computational power to be trained on large datasets) persist and be ported to either the web or mobile devices through methods found within this program’s machine learning library, Scikit-Learn. [4]

As a general description of how this program shall work, it shall display a sentence in English for the user to translate into Latin on their own, allowed to make mistakes, with a trained model to parse such a sentence for grammatical structure. The program will be able to compare an expected Latin grammatical structure generated with the English sentence and the structure of the Latin sentence that the model produced, such that errors can be found and explained (or at least made known) to the user. For now, these errors are to be limited to grammatical errors, with lesser emphasis on semantic error. That is, the program should be able to discern how a sentence should be formed and how the user forms it, though what the original English sentence means and whether the user’s translation matches that meaning will not be considered, unless, somehow, there is time left within the semester to implement it to some degree. Through this error handling, it is expected that a user is taught the basics of grammar and sentence structure, though vocabulary will not be something that will likely be learned without outside effort from the user. Furthermore, since Latin, for the most part, has no required word order, semantic errors will also be given little-to-no emphasis.

Of course, it is unreasonable and unrealistic to assume that simply having the user write sentences without proper lessons would yield any useful information. Even in the unlikely chance that a beginner user understands some of the grammatical forms that words take, it very well may be the case that the user is not fully cognizant of why these forms are used. To remedy this issue, it is necessary for the application to provide lessons on the basics of Latin grammar, such as noun cases and the moods of verbs.

While random generation of English sentences and corresponding expected Latin structure might not be feasible within the timeframe of a single semester, itself being a project worthy of its own capstone, a collection of curated sentences and structures will work as a form of proof of concept.

Naturally, since Python has extensive support for machine learning and tools used therein through libraries such as Scikit-Learn and Pandas, the program in question will be written in Python. Likewise, Python also has the capacity for developing graphical user interfaces. Because of this, the application will take on the style of object oriented programming, using classes and methods to store and use data.

As an example case of how the user might use the application, suppose that the sentence that the user must translate is, in English, “The mother says her son sings”. Since it is (currently) unfeasible to generate new, randomized sentences, this means that the English sentence is curated and has a predefined grammatical structure, which would be a (singular, feminine, nominative noun), (a present, active, indicative, first person, singular, indicative verb), (a singular, masculine, accusative noun), (a singular, masculine, accusative adjective), and (a present, active, infinitive verb). Unbeknownst to the user, this structure is stored away in a secluded portion of the program, which will be discussed in the future with the Model View Control architecture. The user is presented with the original, English sentence and a prompt to translate it into Latin through a series of text fields within the application’s graphical user interface. Likewise, in the graphical user interface will be another text field, one that differs from the other in that it is indeed editable, and this will be where the user types their translated sentence. Also in the interface will be a series of buttons that can be pressed, one to signal to a controller section of the program to pass the user’s translation to model the section, and a series of other buttons to cause the view section to display lessons on the basics of Latin grammar for reference. After the user translates the sentence, the user will hit the aforementioned button, and the translation will be sent to machine learning models in one of the other sections. From there, the machine learning models will attempt to predict the translation’s grammatical structure. Once this is done, the grammatical structure will be sent back for the user to view, and it will be compared with the expected grammatical structure such that errors may be, at the very least, highlighted to make themselves plainly known to the user.

**Foundations**

At the forefront of any end user program, an understandable, appealing, and effective graphical user interface, otherwise known as a GUI, is responsible for displaying information to the user and, where possible and applicable, passing user input to the lower-level parts of the program. Seeing as how many users would perhaps not themselves be tech-savvy, asking them to use the program through a command line, one of the few ways to interact with running programs with no dedicated GUIs, would be unreasonable. To that end, many libraries and frameworks have been created, with one in particular even being integrated into base Python, Tkinter. [5] For the sake of simplicity, then, Tkinter will suffice for displaying the graphical components of the program. It is imperative that all parts of the program that would need graphical interface support are known early, so that there is always some way to interact with and/or extract information from the program.

This would be a good opportunity to discuss how the program would, throughTkinter, accomplish its goals in imparting information to and taking input from the user. As the basic description of the program’s main functionality suggests, the main 3 parts of the GUI should be three fields of displayable text, one editable and two non editable, with one of the two non editable text fields being dedicated to displaying an English sentence to translate and the other to display errors in the user’s translation. As well as that, there should be a form to signal to the GUI to pass the data entered into the editable text field to lower-level parts of the program, such as a button. This would be done through a specific style of GUI programming, known as a Model-View-Controller architecture, which is used to limit the dependencies of otherwise dependent portions of the program. [6, 7] In this specific architecture, as the name suggests, the program is composed of 3 sections: a model, a controller, and a view. The simplest of these to explain would be the view. The view is the portion of the program dedicated to interpreting and displaying the internal data to the user. That is, the view is what the user actually sees on the screen. It follows, then, that the view shall contain all the elements of the GUI, which are called widgets, that were previously described. This would include the three text fields, the visual representation of the button to signal when the translation is done, and the window in which all of the other widgets are contained. [6, 7] The view, of course, would need data to display and something that would make such data more than predefined text on a screen. To fill such a niche would be the model, the representation of the internal workings of the program and the collection of methods by which to manipulate input into output. [6, 7] To avoid confusion between this definition of model and the definition of model used in machine learning, they will be labelled as MVC model and ML model, respectively. The MVC model is responsible for hiding away the internal workings from the user and instead make the results of such workings into a more understandable format to be sent to the view. This is, of course, the general description of what MVC models do; what this program’s will do will be store the English sentence’s text, said sentence’s Latin equivalent’s expected grammatical structure, the grammatical structure of the user’s translation, and trained ML model within variables, as well as any methods defined to assist in the program’s functionality. Finally, in between the view and the MVC model is the controller, the portion of the program that is responsible for handling event cases between the two other portions. [6, 7] That is to say, the controller is what is responsible for handling the passing of information and signals to use methods to the other parts of the application, and is what the user directly interacts with in the application. In general, the controller first passes information to calls on the MVC model to update the low-level data, and then calls to the view to update itself according to the changes made in the previous step. Event cases need not be only from user input as well; the other portions, especially the MVC model, may also send signals to the controller. For example, if it were to be desirable to save the state of the user’s translation on a regular basis, a timer inaccessible to the user in the MVC model may be connected to the controller to request the translation from the view. Seeing as how Python is an object oriented programming language, these three portions of the program will come in the form of classes, each portion being given a class to represent it. These classes will act as wrapping classes for the entirety of the program, encompassing their respective sections as according to the above information.

As natural language processing is meant to be able to handle text data in most cases, and since ML models themselves must be trained on numeric data, there are a series of steps to transform text data into numeric representations. Firstly, of course, there must be text data to begin with. This typically takes the form of a corpus, a large collection of text data. These large collections can come from various sources, such as books, and it is not uncommon for multiple corpora to be used for training, so as to provide a large degree of representation of natural language. Corpora of various languages and various types of sources can be found in similarly varied locations. For the sake of this project, though, a sufficient array of corpora are provided and available through methods in the CLTK repertoire, chiefly the get\_corpus\_reader() method. [3] Likewise, CLTK also provides lexica in the same manner, which provides information needed for training. Lexica, unlike other corpora, contain the grammatical information for words in their text, allowing for training without having to input the data’s information manually beforehand, which is critical for effective time usage for any project. [3] As such, since grammatical data is indeed relevant to the application’s functionality, it is in reality the lexica that is required for the purposes of training the ML model. There are a few ways in which the grammatical data in these lexica are represented, fittingly but confusingly called grammars. A genre among these grammars is called treebanks, themselves being a subset dependency grammars, according to which are what many of the available lexica for Latin are formatted. [Bamman and Crane] One example would be the Ancient Greek and Latin Dependency Treebanks, which format, as the name suggests, words according to their dependencies in a tree-like structure, such as subjects and objects being reliant on their respective verbs. [9] As will be discussed in the future, though, the exact structure of the grammar is not particularly important for this program, simply requiring that the structure be traversable to retrieve its information. Luckily, trees are indeed traversable, and through a variety of means as well. It should be noted, however, that although the specific structure by which the information of words in a sentence is stored is unimportant for the sake of this application, dependency treebank grammars are not completely useless by default. Indeed, with more time, more complex methods by which to predict, in a ML model sense, and assess the grammatical structure of a user’s errors could be produced. This is because dependency treebank grammars are what are known as context sensitive grammars. [8] This means that information of not only what the form of the word is but also how it is being used in a sentence is apparent through the structure of the grammar. This means that, rather than simply checking if all the necessary and correct forms of words are present, the dependencies between these words would also be able to be assessed. That is to say, by use of this, though it is beyond the scope of this project, the program would be able to dynamically explain errors in the This would work to also having the user practice proper syntax.

Though the data is now obtained, there must also be a way to convert it to numeric data, since most machine learning models are incapable of using text data. To that end, various forms of vectorizers exist to transform the text into a collection of numbers. One such vectorizer, and one of the more popular ones, would be CountVectorizer, which transforms the text into the frequency of tokens (usually words) within the text. While in an analysis of a text as a whole, this can be further improved by using a TF-IDF transformer, which decreases the dependence on stop words, words that appear too frequently to yield much useful information. TF-IDF proves to be less effective for individual words as data, as each letter is potentially important. [3, 2] TF-IDF may prove useful in tagging parts of speech by being able to identify Latin’s own stop words, such as the indeclinable “et”.

There is a concession to be made about the ML model that will do this task of parsing and classifying the words in the user’s sentence; there will not be only one model, but a collection of them. Firstly, due to the dissimilarity to classifying a word as a particular part of speech and classifying a word based on its grammatical information, the two tasks will be split accordingly, with one ML model to do the former, and a collection of other ML models to do the latter. To increase the accuracy of classifying each word with a known part of speech, many similar ML models will be made for each part of speech, rather than have one giant ML model classifying each word’s inflected grammatical information. Notably, this comes with at least one notable advantage. For example, this ensures that each part of speech is used to train ML models modularly, and because of this, the hyperparameters, which determine how ML models are trained, can be tuned for each part of speech. This prevents the case of relying on one set of hyperparameters to be, potentially unsuitably, used for all parts of speech.

As before, the ML models mentioned above will be classifiers, rather than other types, like regressors. This is because, obviously, the tasks that these ML models are to undertake are ones of classification, like determining what part of speech a word is or determining if a noun is singular or plural. There are many ML models that can be used for this purpose, creating their own genre of ML model, in fact. These can include anything from KNearestNeighbors to DecisionTrees. Due to the fact that there are so many different classifiers that could be used, finding the one that would best suit the program at each different instance of a ML model being used is imperative. KNearestNeighbors is a good contender for the gathering of inflected data, as words with the same inflected information tend to be formed with similar endings. The following logic, then, is that words with the same inflected information will have greater similarity with each other than with words with different inflected information. Since KNearestNeighbors merely looks to stored data points to compare with new data points, KNearestNeighbors will likely, and hopefully correctly, classify words with identical inflected information under the same category. [4] The primary hyperparameter for KNearestNeighbors, of course, would be k, the number of neighbors to compare to the data point to be predicted. Considering too few neighbors, of course, would lead to an underfitting scenario where the ML model does what is functionally identical to guesswork. However, considering too many neighbors would lead to an overfitting scenario where the ML model has become so specialized, looking for near exact matches between old and new data points, that it cannot reliably correctly predict the class of a new piece of data.

**Implementation**

The current plan for implementation is to incrementally build models that will handle identifying and declining/conjugating the various parts of speech. The first things to be implemented would be the very basic systems of the program; the GUI, at least with some basic functionality in the view and controller and with a basic MVC model should be the first thing to be implemented, so that testing the interactivity of the application at all stages is possible. From there, the main difficulty will be in implementing the various ML models that will compose the main “model”. As always, there is a degree of uncertainty when it comes to plans in the future, and the courses of action described in the following sections may not be adhered to with inflexibility. As the GUI is not among the more particularly demanding tasks within this project, it will be worked on over the course of break between semesters, and should be ready for demonstration by the first biweekly checkpoint within the spring semester. Another task to perform over the break would be testing different ML models from among the possible contenders for each task, so as to get practical comparisons between different ML models trained for the same task.

To the end of making the GUI between semesters, the aforementioned Tkinter library will be used. Tkinter incorporates multiple means by which the widgets in the application’s window may be made. For example, the main window through which the user will see all the other widgets is itself a widget, being a top-level widget class called a Tk. [5] Referring back to the concept of the Model View Controller architecture, this means that the View will contain very few fields itself, needinging, as is currently known, only a Tk object. The ScrolledText class is also provided by the library, which has the capability to display text within the window and be able to be scrolled to accommodate longer sentences. [5]

The first among the ML models will be the most basic, being a part of speech tagger. The point of this would be to be able to distinguish between the various parts of speech found within Latin. These parts of speech would be things such as nouns, adjectives, and verbs, notably. As stated before, this ML model will be a classifier. Due to the fact that Latin is inflective, meaning that it contains grammatical information within the forms of its words, there are only a few cases of ambiguity among the words to be classified, which, of course, should be handled accordingly. For example, the singular, ablative/dative, masculine noun “amico” might be confused for a singular, first person indicative verb by the ML model, since they both tend to end with “-o”. The greatest form of ambiguity would be between nouns and adjectives, as they share similar formations due to the rules of complement. Some points of ambiguity, though do not majorly impact the grammar of the word. For example, participles, such as “paratus”, are verbal adjectives, meaning that classifying them as either a verb or an adjective would be technically correct, though, due to their forms, they will most likely be classified as adjectives. The initial plan for this part of speech classifier was to update and retrain it, simply adding support for classifying parts of speech as the other models were developed to support said part of speech. However, after a revelation that adjectives could use the same ML models for classifying most of their inflected information leaves open a gap of time, and thus a change of plans is going to be made. This part of speech classifier will take the place of the time that would be spent on the noun ML model, shifting the ML models for noun and adjectives to the

Following with Latin’s inflective nature, it makes sense to train models to gather the grammatical information found within words. The first among these, arbitrarily, will be nouns and adjectives. As stated before, nouns and adjectives are rather similar to each other, given that the rules of complement dictate that adjectives must match their complementary nouns in number, case, and gender. However, assuming that the part of speech classifier model can discern between the two, this can be used to the advantage of the program. Both nouns and adjectives would need to be classified by a ML model for case (nominative, accusative, dative, genitive, or ablative), number (singular or plural) and gender (masculine, feminine, or neuter). Adjectives would only need a small further classification of degree (positive, comparative, and superlative). While in the past, it seemed that adjectives would need their own singular ML model for classification of their inflected information, thus warranting the need of a separate checkpoint to develop and explain, it is apparent now that it is possible to allow for nouns and adjectives to be classified mostly by the same ML model. This frees up some time around the third biweekly checkpoint that would otherwise be dedicated to developing ML models for adjectives, time that can now be spent performing other tasks, perhaps by more rigorously testing the ML models or by adding support for other less critical parts of speech, such as adverbs. Adding support for nouns and adjectives is scheduled to be completed by the time of the second biweekly checkpoint, January 21st.

The next part of speech to contend with would be verbs. Verbs are among the most complex parts of speech within Latin, requiring that they be classified according to its number (singular or plural), voice (active or passive), person (first, second, or third), and mood (indicative, imperative, infinitive, or subjunctive). As before, participles can be classified as either verbs or adjectives, so using the previous classifier ML model for nouns and adjectives would be acceptable for gathering the participle’s inflected grammatical information. As mentioned in previous sections of this paper, the ML models designed for classifying verbs will be quite similar to the ones used for classifying nouns and adjectives, working through the same methodology with the same reasoning. The goal to add classifiers for verbs within the program’s collection of ML models is scheduled to be completed by February 18th, the time of the third biweekly checkpoints.

With those, in truth, most of the major parts of Latin grammar are covered. Further time would be dedicated to fixing whatever issues are remaining, such as, potentially, the ambiguity between nouns and adjectives. Adding support for more auxiliary parts of speech, such as adverbs or prepositions, could also be done within this remaining time.

While the treebank dependency structure is potentially useful for more in depth analysis of grammatical structure, and would certainly be useful in training ML models to help generate new sentences for translation, it is not entirely what is the drawing point for their use in this application. Instead a small portion of the information found within the format that these treebanks are found are what are indeed needed. Within each node of the treebank, there is a certain field known as pos. [9] This pos field contains all the information needed for the word that the node represents, including its part of speech and all the inflected information that the word could possibly have, all in the form of a string. The string or, more specifically, its characters represent the specific combination of the aforementioned information that the word has. Since combinations can be made, and since there are in fact a limited number of combinations, the ML models for this project may indeed be classifiers, with each combination potentially a class that the ML model can predict. However, as stated before, the program will divide machine learning tasks between multiple ML models, reducing the number of classes upon which the ML models will need to be trained.

**Conclusion**

In summation, to create a program by which to create and exercise the understanding of grammar within the Latin language is by no small degree feasible. Indeed, to do so through natural language processing allows for the more naturalistic approach to learning grammar than what might be afforded by the memorization techniques of popular applications currently available to most users. While this application may not be in such a state so as to compete with such others by the end of the spring semester, it would be remiss to dismiss its potential. While aspirations are currently to simply make a reasonable means of classifying Latin words so that they can be used in a simple translation exercise project, it nonetheless represents a starting point for further work in developing grander applications that allow users to learning languages, not just Latin, through means other than memorization of phrases and vocabulary.

**Bibliography and Sources Cited**

[1] Wang et al. 2020. A Contribution-oriented Self Directed Mobile Learning Ecology Approach To Improving EFL Students’ Vocabulary Retention And Second Language Motivation. *Educational Technology and Society* 23, 1, 15 pages. Retrieved from <https://www.jstor.org/stable/26915404>.

[2] Chin-Hsi Lin and Mark Warschauer. 2015. Online Foreign Language Education: What Are The Proficiency Outcomes. *The Modern Language Journal* 99, 2, 4 pages.

[3] Kyle P. Johnson et al. 2019. Classical Language Toolkit Docs: Latin. Retrieved from <https://docs.cltk.org/en/latest/latin.html>.

[4] Scikit-Learn. User Guide: Contents - Scikit-Learn 0.23.2 Documentation. Retrieved from <https://scikit-learn.org/stable/user_guide.html>.

[5] Python. 3.9.0 Documentation. Retrieved from <https://docs.python.org/3/>.

[6] Yihui Xie et al. 2014. Reactive Programming For Interactive Graphics. *Statistical Science* 29, 2, 13 pages. Retrieved from https://www.jstor.org/stable/43288470.

[7] Dragos-Paul Pop and Adam Altar. 2013. Designing An MVC Model For Rapid Web Application Development. *Annals of DAAAM and Proceedings* 24, 1, 8 pages. Retrieved from <https://libprox.monm.edu/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=a9h&AN=109386171>.

[8] Aravind J. Joshi. 1991. Natural Language Processing. *Science* 253, 5025, 4 pages. Retrieved from https://www.jstor.org/stable/2879169.

[9] David Bamman and Gregory Crane. The Ancient Greek and Dependency Treebanks. Retrieved from http://static.perseus.tufts.edu/docs/latech.pdf.