map_gloss.py
Tutorial and Reference
AGGREGATION Inference
University of Washington

Prepared by Michael Lockwood, June 2015

Table of Contents

Tutorial

1 Getting Started

This tutorial is meant to be a graded approach at introducing the user to the map_gloss.py script. The map_gloss.py process allows users to automatically correct and map gloss values to a compressed and hopefully more accurate set. Morphemes and words can be attached to this process as well. A user who desires to only take advantage of the collection and compression of glosses may select between two functions. This allows ignorance of the cumbersome details of the classes and functions within map_gloss.py.

1-1 GLOSSING

An overview of the script will show several classes. Two functions at the bottom are not members of classes. These are the classes that provide an API like ability for a user to merely input data into a function instead of calling the functions of each class in the correct order. Two options are available. The first is called glossing:

`glossing(data, user_input=False, user_preference=2, restart=True)`

`data` The glossing() function requires that data arrive in a specific format. At the very least it needs to be organized into tuples. The first element [0] of the tuple will contain the ISO-639 code for the language. The second element [1] must contain a list of the glosses that the system will process. Tuples can be stored into a list which can fill this data requirement.

`user_input` The user has the option to provide feedback during the training stage. This allows the user to control how ties are broken for the Levenshtein distance method, whether glosses are English words instead of glosses, and how to input a new gloss occurrence if a match has not been found in the system. The python interface will propose questions to the user and then automatically call the functions to resolve the input. The program structure will remember these values and recall them upon seeing the same gloss again.

`user_preference` The user can select which values of the glosses they would like for output. The system records a Leipzig, GOLD (ontology), or author based gloss for every input gloss. The default is to select the author gloss. The author gloss system is very special in that it learns which input values may be the same and will return the most common. Often times perfect and imperfect are called by different glosses even by the same author. This will converge them to the most likely and return that value if the author preference is selected.

`restart` The restart option will toggle between reading in previously trained files or not. The default is to retrain every time so that observation values are not distorted by calling the same data time and time again. However, if the user utilizes the manual method then restart should be set to False to avoid deleting previous work. Outputs of all classes are stored as pickle files. There are individual and combined options. The glossing() function by default will export a map_gloss.pkl file which combines all of the data structures into a single file.

1-2 XIGT GLOSSING

The xigt_glossing() function parallels the previous glossing function. It has the added benefit of directly working with enriched XIGT files to extract and process data for the user. If a user only needed glosses from XIGT this would essentially allow the user to avoid working with XIGT on their own. It is important that the user has the XIGT directory in their PYTHONPATH. Please see the AGGREGATION website for more information about how to configure the PYTHONPATH for the purpose of loading XIGT and working with other AGGREGATION tools:

http://lemur.ling.washington.edu/trac/aggregation/wiki/Setup

Not suprisingly, the format for the xigt_glossing() function is not dissimilar from glossing():

xigt_glossing(data, user_input=False, user_preference=2, restart=True)

data The xigt_glossing() function must still intake data organized into tuples. The first element [0] of the tuple will contain the ISO-639 code for the language. The second element [1] will instead by the path to an enriched XIGT file. If the file is not a XIGT file the script will fail because it will load and attempt to process the file as if it were XIGT.

user_input The user_input operates the same as with glossing().

user_preference The user_preference operates the same as with glossing().

restart The restart operates the same as with glossing().

1-3 INITIAL OUTPUTS

By default both the glossing() and the xigt_glossing() functions will return a list of output glosses in the same order as the input. Note that the xigt_glossing() will also perform morpheme mapping with glosses but this will not impact or be present in the output. These scripts and their respective outputs will work well for general purposes although it is recognized that some tasks require more complex addressing of languages by ISO-639 and gloss. The rest of the section addresses how to understand the MapGloss class and its functions

2 MapGloss

The MapGloss class hosts functions that direct the mechanics of the other classes in the map_gloss.py module. Its functions are the most important for the user to understand for manipulation of each of the other classes and ultimately to affect the desired gloss and morpheme outputs. MapGloss is the only class that does not have a constructor. It does not have objects because its role is to facilitate the management of the objects in other classes.

2-1 TRAINING

The training stage relies on two functions. The first is called training() and it has the role of actually training each gloss (and morpheme) by building all of the necessary object values. Once every gloss has been trained the prepare_decoding() function will set the references section which maps centroids of similar concepts.

training(iso, gloss, morpheme='')

iso This is the ISO-639 value for the gloss. The data formats of glossing and xigt_glossing format this automatically.

gloss This is the gloss that will be learned. The algorithm will either add an observation of it if it already exists or will build a new object for it. Note that gloss objects are stored in tuples of (iso, gloss). This is to distinguish glosses by language so that the statistics of one language do not affect another.

morpheme This optional variable allows the user to store morpheme to gloss and gloss to morpheme translations. This is helpful for systems trying to target morphology. If the morpheme is the empty string the system will not process it. If there is a value then it will process. The xigt_glossing() will automatically call and process morphemes.

After all of the glosses have been trained then the prepare_decoding() may be called to form centroids in the references value of each language and gloss object. These centroids represent unique concepts like 'Imperfect'. As an example if the centroid included 'IPFV', 'Imperfect', 'Imperfective', 'IMPF', and 'IMPV' each of their references would be a dictionary storing all of the other values. This will not impact inference procedures but it provides equivalencies for evaluation purposes.

prepare_decoding()

The prepare_decoding() function takes no arguments. It operates completely internal to map all of the references. This should only be called once after training all glosses.

2-2 DECODING

The decoding stage takes glosses and processes them to return the user desired output. By default this is the author gloss but as subsection **2-5 SET USER VARIABLES** will demonstrate, this value is adjustable to select Leipzig or GOLD values instead.

decoding(iso, gloss)

iso This is the ISO-639 value for the gloss receiving the decoding.

gloss This is the gloss that will be decoded. The algorithm will return the value of the gloss that matches the user preference (0 = Leipzig, 1 = GOLD, 2 = author).

2-3 COMBINED EXPORT AND LOAD

Originally map_gloss.py relied on the pickle format to import and export data structures. This is an important task for two reasons. If the user desires to use a manual approach to affect the results then they will likely want to save their effort; exporting their outputs would achieve this. Secondly and more importantly, it is likely that someone will build different scripts for their inference and evaluation (presumably other functions with glosses and morphemes as well). The references need to be identical between the versions in order to support this. Unfortunately the pickle structure turned out to be inadequate for storing classes and objects, even with the Pickler and Unpickler modules which were intended for this very purpose. Thus a new data storage format was designed specifically for this purpose; called CLSO for CLS – class and O – object. Please see section **11 CLSO** for detailed documentation of the CLSO data format. The combined export function places all objects in a single CSLO file called 'map_gloss.clso'.

export_combined_structures()

Calling this function will automatically write all of the objects in all map_gloss.py classes to the 'map_gloss.clso' file. When the script is run again (assuming MapGloss.restart=False) the user can reload the objects by calling load_combined_structures():

load_combined_structures()

Again, this will function automatically and load all of the objects for the user as long as the 'map_gloss.clso' file is the current working directory. Otherwise it will try to load individual CSLO files. For each of the individual files that fails, the class has a method to restore default values.

2-4 INDIVIDUAL EXPORT AND LOAD

As mentioned in the previous subsection **2-3 COMBINED EXPORT AND LOAD** map_gloss.py uses the CLSO data format. Should the combined loading fail the system will automatically try to load individual structures. This is because if individual loads fail each class has a solution for loading default values. Otherwise the only difference between the combined and individual export and load is that the combined produces one 'map_gloss.clso' file while the individual option produces files for each class called by the class name with the '.clso' file extension. An example is 'leipzig.clso'

export_individual_structures()

A call to export_individual_structures() will automatically write individual files for each of the classes in map_gloss.py.

load_individual_structures()

A call to import_individual_structures() will automatically load individual files for each of the classes in map_gloss.py. Again, if the files do not exist it will use defaults defined by each class.

2-5 SET USER VARIABLES

The user variables are class variables for MapGloss that affect the processing of certain functions.

set_max_distance(value)

value An integer that indicates the maximum distance for the Levenshtein string distance algorithm.

The max_distance variable controls when the Levenshtein distance becomes too much for the system to automatically try to map it from a word to gloss. If a gloss cannot be matched within the max_distance then it will be added to the Lexicon class and be considered a word only.

set_user_input(value)

value A Boolean that toggles between manual and automatic modes. Set to True it will allow user feedback (manual). Set to False and it will allow the learning methods to make all of the decisions (automatic).

The user_input will control how a lot of processes take place. The next subsection **2-6 USER ADDITIONS** can only take place when the MapGloss is processing manually. The max_distance impacts the automatic and manual differently. It will be the ultimate decision maker to determine if a value is an actual gloss or a word in automatic but merely be a decision boundary for manual in which case all values within the distance are possible for the user to select (if they share a minimal string distance value).

set_user_preference(value)

value An integer that sets the desired decoded output for the user. A 0 will return Leipzig, a 1 will return GOLD, and a 2 will return the most probable author tag based on trained observations.

The user_preference is important for determining what the system outputs. This is likely critical for those taking the outputs directly and manipulating them as a stand alone element for inference or evaluation. It is instead recommended to interface with map_gloss.py during inference so that the desired value can be called direction for each of the objects in the structure (this is much more efficient than sorting through a list of all of the occurrences of each gloss in decoding). Nonetheless this option will remain for those preferring to take the list of output glosses.

set_restart(value)

value A Boolean that toggles between whether to load an existing CLSO file or restart the training.

In terms of runtime the restart will not have an effect. It is critical that restart is set to False for evaluation purposes. A best practice would be to follow what was implemented for tense, aspect, and mood: in infer_tam.py the restart was set to True to allow refreshing the values but then is set to false in eval_tam.py so that the references would stay in tact.

clear_class_data()

The clear_class_data() resets all of these values to their default values.

2-6 USER ADDITION

The user addition works in tandem with the manual mode to prompt questions of the user to improve the output of the system. It will ask questions about whether a gloss is a gloss or a word, whether the gloss should be paired with a close matching Leipzig or GOLD value, or if the value is its own gloss.

⚠ This function may be deprecated as manual has not been tested for some time. Reasonable attempt was made to place and adjust the function to fit the current properties of map_gloss.py but no guarantee is made that this will function as the user intends.

user_addition(gloss)

gloss The gloss that the system has prepared for the user to review manually.

Calling user_addition() will interact with the Leipzig, Gold, and Delphi classes to place the gloss in the correct place with the corresponding Leipzig, Gold, and author values as appropriate. Note that the default is to assume that user defined values are pseudo=True indicating that they are not actual Leipzig or Gold values. This will not affect their processing but it will signal to anyone loading the CLSO produced by this user that the Leipzig and Gold value(s) may not be standard.

2-7 GET EXAMPLES

The final function of the MapGloss class simply returns dummy data to test with the glossing() function.

get_example()

Calling get_example() will return Yakima Sahaptin (yak) and Russian (rus) values that can be passed to the glossing() function direction.

3 Leipzig and Gold

The Leipzig glossing standards and the GOLD ontology form the standard for map_gloss.py concepts. When the system reads an input gloss it will attempt to figure out which Leipzig and GOLD value is represented by the gloss. By default there must be an equivalent Leipzig and GOLD tag for each other. When an action occurs against one the system will automatically perform it with its counterpart of the other. Therefore, in the event that one of them does not have a gloss (in practice this is almost always Leipzig), the object will have a value called *_psuedo* set to True that will indicate author creation of the gloss. By default when a user enters values interactively through the manual process, the *_pseudo* value will be set to True indicating authorship of the value. This section explores how to load, export, add, delete, and manage the Leipzig and Gold classes which represent the mapping standards. This will use Leipzig as the example although every Leipzig function is also in Gold.

3-1 OBJECT CONSTRUCTOR

The object constructors for both Leipzig and Gold require both equivalent values. No Leipzig can be free of a Gold value pair and vice-versa. They also have an optional pseudo value that indicates whether the value is actually Leipzig or Gold or if it has been invented for the purposes of equivalency. Note that the constructor will automatically place the object in a lookup for the class called objects. This occurs for every class in map_gloss.py except for MapGloss which does not have objects.

__init__(self, leipzig, gold, pseudo=False)

leipzig This is the leipzig value of the Leipzig object.

gold This is the gold equivalent of the Leipzig object.

pseudo This optional variable indicates if the leipzig value is standard or invented.

The above __init__() is the Leipzig class constructor, the constructor for Gold is very similar.

__init__(self, gold, leipzig, pseudo=False)

gold This is the gold value of the Gold object.

leipzig This is the leipzig equivalent of the Gold object.

pseudo This optional variable indicates if the gold value is standard or invented.

3-2 LOAD

The load_leipzig() and load_gold() functions will first attempt to load a CLSO file for each of their classes if MapGloss.restart has been set to False. Otherwise they will load a default dictionary of values and their counterparts.

load_leipzig(restart=False)

restart A boolean value that toggles between loading the CLSO file or the default data structure.

Calling load_leipzig() and load_gold() initializes the base and/or CLSO file Leipzig and Gold objects.

3-3 EXPORT

The export_leipzig() and export_gold() functions take all of the objects stored in Leipzig.objects and Gold.objects and dump them to the 'leipzig.cslo' and 'gold.cslo' files.

export_leipzig()

3-4 DELETE

The delete_leipzig() and delete_gold() functions will take a Leipzig or Gold value (respectively) as input and delete that value for its class. They also will find the counterpart and delete it from the other class. This maintains the balance that Leipzig and Gold values must have equivalents. In the normal map_gloss.py routine this is never called but may have use for those attempting the manual mode.

delete_leipzig(leipzig_value)

leipzig_value A string value for the Leipzig gloss that the user wishes to delete.

Note that calling delete_leipzig(leipzig_value) will also delete the Gold equivalent of the leipzig_value. The same is try of calling delete_gold(gold_value) for the gold_value's Leipzig equivalent.

3-5 GET VALUES

The Leipzig and Gold classes have a set of get functions that return values stored within the object.

get_leipzig(self)

This will return the Leipzig value for the gloss.

get_gold(self)

This will return the Gold value for the gloss.

get_pseudo(self)

This will return the pseudo value for the gloss.

get_iso_codes(self)

This will return a dictionary of ISO-639 values as keys if the gloss was observed for the language.

get_iso_list(self)

This returns a list of keys for the iso codes dictionary instead of the dictionary itself.

4 Delphi

The Delphi class organizes values that should be corrected or avoided by the learning process. Delphi methods are those that generally involve a group of profession experts collaboratively agreeing on a central solution. These typically work when modeling does not. In the context of map_gloss.py the Delphi class is the set of values that will be automatically flagged as incorrect and will be automatically corrected. They take no context and are therefore considered absolute. They cannot override Leipzig or Gold values though, that is an important check within the system.

4-1 OBJECT CONSTRUCTOR

The object constructor for Delphi requires Leipzig and Gold equivalents.

__init__(self, delphi, leipzig, gold)

delphi This is the delphi value of the Delphi object. This would be a gloss value that should be corrected to a corresponding Leipzig and/or Gold value before any learning method can influence it.

leipzig This is the leipzig equivalent of the Delphi object.

gold This is the gold equivalent of the Delphi object.

The above __init__() will set the Delphi object and add it to the Delphi.objects dictionary.

4-2 LOAD

The load_delphi() function will first attempt to load the Delphi CLSO file if MapGloss.restart has been set to False. Otherwise it will load a default dictionary of values and their counterparts.

load_delphi(restart=False)

restart A boolean value that toggles between loading the CLSO file or the default data structure.

Calling load_delphi() initializes the base and/or CLSO file Delphi objects.

4-3 EXPORT

The export_delphi() function takes all of the objects stored in Delphi.objects and dumps them to the 'delphi.cslo' file.

export_delphi()

4-4 DELETE

The delete_delphi() function will take a Delphi value as input and delete that value for its class. In the normal map_gloss.py routine this is never called but may have use for those attempting the manual mode.

delete_delphi(delphi_value)

delphi_value A string value for the Delphi gloss that the user wishes to delete.

4-5 GET VALUES

The Delphi class has a set of get functions that return values stored within the object.

get_delphi(self)

This will return the Delphi value for the gloss.

get_leipzig(self)

This will return the Leipzig value for the gloss.

get_gold(self)

This will return the Gold value for the gloss.

get_iso_codes(self)

This will return a dictionary of ISO-639 values as keys if the gloss was observed for the language.

get_iso_list(self)

This returns a list of keys for the iso codes dictionary instead of the dictionary itself.

5 Language

The Language class stores lists of Gloss, Segment, Levenshtein, Lexicon, and Morpheme object references for a language in order to facilitate organizing and retrieving objects. The Language objects should be initialized with ISO-639 values.

5-1 OBJECT CONSTRUCTOR

The object constructor for Language requires only an iso value.

__init__(self, iso)

iso The ISO-639 code for the language to initialize.

The above __init__() will set the Language object and add it to the Language.objects dictionary with a key equal to the iso code.

5-2 LOAD

The load_language() function will first attempt to load the Language CLSO file if MapGloss.restart has been set to False. Otherwise it will load a default empty dictionary.

load_language(restart=False)

restart A boolean value that toggles between loading the CLSO file or the default data structure.

Calling load_language() initializes the base and/or CLSO file Gloss objects.

5-3 EXPORT

The export_language() function takes all of the objects stored in Language.objects and dumps them to the 'language.cslo' file.

export_gloss()

5-4 DELETE

The delete_language() function will take a Language iso value as input and delete that value from the class. In the normal map_gloss.py routine this is never called but may have use for those attempting the manual mode.

delete_language(iso_value)

iso The ISO-639 code for the language of the gloss that the user would like to delete.

5-5 REFERENCES

There may be several possible equivalents for the same author tags. In map_gloss.py the terminology adopted for these clusters is centroids because they group values to a central theme based on the Gold value. These centroids conceptually group the same valued glosses together. So where 'IMPF' was meant to be 'Imperfect' it is combined with 'IPFV' and 'Imperfect' as well as any other authored glosses mapped to 'Imperfect'.

set_references(self)

The set_references() function processes all glosses in the class and builds a Language specific references dictionary. This dictionary contains the most probable author gloss for every centroid. This function will also call the set_references() function of each Gloss object so that they automatically inherit from the Language object's references.

5-6 PRINT VALUES

Before the CLSO class which handles a human and machine readable format for loading and exporting data, the print_values() function allowed the user to see the values of the Gloss class. CLSO does have byte writing and reading properties so the print_values() may work well for those trying to write CLSO files to bytes.

print_values(self)

This will print a diagnostic for the all Gloss objects in the Language object's glosses. This diagnostic contains all of the values for the object written in an easy to read format. Note that it is strongly recommended to become familiar with the CLSO class and format because it has more capabilities than print_values().

5-7 GET VALUES

The Gloss class has a set of get functions that return values stored within the object.

get_iso(self)

This will return the ISO-639 value for the language.

get_glosses(self)

This will return a dictionary of all glosses in the language. Find the Gloss object by calling Gloss.objects[(self._iso, gloss)] if and only if gloss refers to a key in get_glosses().

get_segments(self)

This will return a dictionary of all segments in the language. Find the Segment object by calling Segment.objects[(self._iso, segment)] if and only if segment refers to a key in get_segments().

get_morphemes(self)

This will return a dictionary of all morphemes in the language. Find the Morpheme object by calling Morpheme.objects[(self._iso, morpheme)] if and only if morpheme refers to a key in get_morphemes().

get_observed(self)

This will return a dictionary with Gold values as keys. This dictionary contains all of the centroids for the language and is used to clarify the most common author value.

get_references(self)

This will return a dictionary of all matching reference glosses for the Language.

6 Gloss

The Gloss class stores all of the gloss values for each of the languages by (iso, gloss). This allows for very easy and precise retrieval of all glosses for each language. The glosses interact heavily with the Segment and Morpheme classes as well as any learning class (such as Levenshtein and TBL).

6-1 OBJECT CONSTRUCTOR

The object constructor for Gloss requires both the gloss and iso values.

__init__(self, iso, gloss)

iso The ISO-639 code for the language to which the gloss pertains.

gloss The current gloss for either training or decoding purposes.

The above __init__() will set the Gloss object and add it to the Gloss.objects dictionary with the key (iso, gloss).

6-2 LOAD

The load_gloss() function will first attempt to load the Gloss CLSO file if MapGloss.restart has been set to False. Otherwise it will load a default empty dictionary.

load_gloss(restart=False)

restart A boolean value that toggles between loading the CLSO file or the default data structure.

Calling load_gloss() initializes the base and/or CLSO file Gloss objects.

6-3 EXPORT

The export_gloss() function takes all of the objects stored in Gloss.objects and dumps them to the 'gloss.cslo' file.

export_gloss()

6-4 DELETE

The delete_gloss() function will take a Gloss value as input and delete that value for its class. In the normal map_gloss.py routine this is never called but may have use for those attempting the manual mode.

delete_gloss(iso_value, gloss_value)

iso The ISO-639 code for the language of the gloss that the user would like to delete.

gloss The gloss that the user would like to delete.

6-5 SET VALUES

The set_values() function processes a gloss and determines what its Leipzig, Gold, and author values are. By default it will check first if the gloss is Leipzig or Gold. If it is, then the author tag will match the input. Otherwise it will try to learn based on the learning methods. Delphi is a quasi learning method and the user can toggle it indirectly. By default it is set to run first and this is highly recommended. If the Delphi method does not produce a result then it will attempt the next learning method. While it is not recommended the user may change the order of learning methods. Form a list calling the methods by name and set the value of MapGloss.learning to equal that value. This may be very helpful if the user would like to add a learning method (they would also have to alter set_values()) but again do so with caution.

set_values(self)

The user only has to call set_values() from the object itself and the process will occur automatically. Note that this process additionally calls methods to segment the gloss. Frivolously calling set_values() may inflate training counts of segments. The automated training method from MapGloss calls set_values() for the user; it may be best to avoid calling set_values() unless processing a small set of glosses very intentionally.

6-6 SET AUTHOR

The set_author() function attempts to select the best author value for the gloss. This should be completed after training and before decoding. Part of the prepare_decoding() function from MapGloss is to run the set_author() for all of the Gloss objects.

set_author(self)

For more information about the observations see subsection **6-10 OBSERVATIONS** which discusses how the author values are counted to determine the most likely value for the gloss concept/centroid.

6-7 SEGMENTS

Segments occur when a gloss contains a gloss as a subset of its text with additional text. For example '1SG' contains two gloss segments, '1' and 'SG'. These may be critical to certain processes so every gloss is tested for segmentation.

add_segment(self, segment)

segment The segment discovered within the gloss.

The add_segment will add the observation of a segment to the gloss. By default this should identically equal the observations of the gloss. This is just a check to make sure the segment functions were called correctly.

set_segmentation(self)

The set_segmentation does a majority of the work in finding segments and handling them. It first looks through all of the Leipzig and Gold glosses. If it finds a match based on regular expression it will call add_segment to report the match and increment its count. It then will instantiate a Segment object for the iso value of the gloss if an object does not already exist. The object's set_combination() function will then be called which increments the observation of the gloss for the segment. These glosses containing a subsuming gloss with additional text are called combination glosses for the purposes of map_gloss.py. Finally, the set_segmentation() upon finding a match will call the _set_segmentation() function for the text preceding and following the matched gloss. For example, if the gloss were '1SG', the '1' may be recognized first. Then set_segmentation() would call _set_segmentation() for '' as the preceding value and 'SG' as the antecedent.

_set_segmentation(self)

The _set_segmentation() function operates recursively. It will continually find inset values of glosses until matches are no longer found. This needs to be separate from the set_segmentation() for two reasons. The first is to extend the inference capabilities of map_gloss by allowing a segment that has no Leipzig, Gold, or Delphi equivalent but is likely distinct from the meaning of the segment(s) of the gloss. An example might be '1ANIM' standing for a created version of first-animate; this would recognize ANIM as a valid segment and possible gloss. It will otherwise be ignored during the learning routes. Secondly, this should apply if and only if a gloss has been found within a word. It would not be desired to have several segments that are actually words. The design intends for _set_recursion() to be called only from set_segmentation(self) and in this manner act as a private variable. By setting the recursion this way, set_segmentation() only handles matches and _set_recursion() can handle segments that do not match.

6-8 MORPHEMES

The map_gloss.py script began as a module to merely track glosses but it became apparent that adding morpheme relations to glosses would not only be easy to accomplish with the updated map_gloss.py structure but beneficial to certain procedures. The Morpheme class allows a user to query morphemes and find the corresponding counts of gloss appearances but this functionality also exists with glosses to morphemes. Both are calculated from the Gloss class with the set_morpheme() function.

set_morpheme(self, morpheme)

morpheme The morpheme aligned with the gloss.

The xigt_glossing will handle this functionality automatically. But if the user wishes to build their own input and output of morphemes then calling the set_morpheme will be important for constructing the morphemes correctly in both the Gloss and Morpheme classes.

6-9 OBSERVATIONS

The observations of a gloss are stored in two formats. The Gloss class stores the total count of the occurrences of the gloss. The Language class stores the Gold value equivalent of the gloss mapped to the gloss. Because this happens for all glosses, if two distinct author values overlap on the same Gold value, the scripting will make sure to set the author gloss for both of them to the most common one. This is important to determine which author gloss should be exported to a choices file if the user prefers the choices file output to use author tags.

add_observation(self)

The add_observation() will automatically increase the count of the gloss by 1 for the user.

6-10 REFERENCES

As observations notes there may be several possible equivalents for the same author tags. In map_gloss.py the terminology adopted for these clusters is centroids because they group values to a central theme based on the Gold value. These centroids conceptually group the same valued glosses together. So where 'IMPF' was meant to be 'Imperfect' it is combined with 'IPFV' and 'Imperfect' as well as any other authored glosses mapped to 'Imperfect'.

set_references(self)

The set_references() function simply points to centroids developed at the Language class level for the glosses and sets the references of the gloss to the centroid of the Language class for that gloss. In fact the standard procedure is for the Language class to call this function for every gloss by language.

6-11 PRINT VALUES

Before the CLSO class which handles a human and machine readable format for loading and exporting data, the print_values() function allowed the user to see the values of the Gloss class. CLSO does have byte writing and reading properties so the print_values() may work well for those trying to write CLSO files to bytes.

print_values(self)

This will print a diagnostic for the Gloss object that contains all of the values for the object written in an easy to read format. Note that this is preferably called from the Language class which can perform this for all glosses in the language. But the function works for individual glosses if the user desires. Note that it is strongly recommended to become familiar with the CLSO class and format because it has more capabilities than print_values().

6-12 GET VALUES

The Gloss class has a set of get functions that return values stored within the object.

get_iso(self)

This will return the ISO-639 value for the language to which the gloss belongs.

get_input_gloss(self)

This will return the original gloss which represents the object value.

get_output_gloss(self)

This will return the preferred output for the gloss. This is used during decoding to return the correct value of the gloss whether it be Leipzig, Gold, or author. Recall that the preference for the method is set by MapGloss.set_user_preference().

get_leipzig(self)

This will return the Leipzig value for the gloss. Calling this before MapGloss.prepare_decoding() will return an empty value.

get_gold(self)

This will return the Gold value for the gloss. Calling this before MapGloss.prepare_decoding() will return an empty value.

get_author(self)

This will return the author value for the gloss. Calling this before MapGloss.prepare_decoding() will return an empty value.

get_delphi(self)

This will return the Delphi value for the gloss which is Boolean indicating whether the learning method was Delphi or not.

get_levenshteini(self)

This will return the Levenshtein value for the gloss which is Boolean indicating whether the learning method was Levenshtein or not.

get_segments(self)

This will return a dictionary of segments and counts for the gloss. Note the counts for the segments should be the same as the gloss itself.

get_morphemes(self)

This will return a dictionary of matching morphemes and counts for the gloss.

get_observed(self)

This will return the count of the number of occurrences for the gloss (language specific).

get_references(self)

This will return a dictionary of all matching reference glosses for the gloss.

7 Segment

The Segment class stores segmented glosses produced from the Gloss class. These segments are stored by ISO-639 value and segment value which will either be a recognized gloss or fragment leftover from the segmentation process (presumably many fragments are glosses that have not been added to Leipzig or Gold).

7-1 OBJECT CONSTRUCTOR

The object constructor for Segment requires both the segment and iso values.

`__init__(self, iso, segment)`

`iso` The ISO-639 code for the language to which the segment pertains.

`segment` The current segment.

The above __init__() will set the Segment object and add it to the Segment.objects dictionary with the key (iso, segment).

7-2 LOAD

The load_segment() function will first attempt to load the Segment CLSO file if MapGloss.restart has been set to False. Otherwise it will load a default empty dictionary.

`load_segment(restart=False)`

`restart` A boolean value that toggles between loading the CLSO file or the default data structure.

Calling load_segment() initializes the base and/or CLSO file Segment objects.

7-3 EXPORT

The export_segment() function takes all of the objects stored in Segment.objects and dumps them to the 'segment.cslo' file.

`export_segment()`

7-4 DELETE

The delete_segment() function will take a Segment value as input and delete that value for its class. In the normal map_gloss.py routine this is never called but may have use for those attempting the manual mode.

`delete_segment(iso_value, segment_value)`

`iso` The ISO-639 code for the language of the segment that the user would like to delete.

`segment` The segment that the user would like to delete.

7-5 COMBINATIONS

The combinations values of each Segment object is a dictionary of all of the gloss values that contained the segment. The value of each gloss key is the count of how many times that gloss occurred.

set_combination(self, gloss)

gloss The gloss that contains the segment.

The set_combination() is processed automatically from the Gloss class but if the user overrides that automation then it is possible to set combination values directly with the set_combination() function.

7-6 OBSERVATIONS

The observations is the total number of times that the segment has occurred in the language regardless of gloss. It should equal the sum of all combination values.

add_observation(self)

The add_observation() will automatically increase the count of the segment by 1 for the user.

7-7 GET VALUES

The Segment class has a set of get functions that return values stored within the object.

get_iso(self)

This will return the ISO-639 value for the language to which the segment belongs.

get_segment(self)

This will return the original segment which represents the object value.

get_combinations(self)

This will return a dictionary of all glosses which contain the segment as keys with the count of their occurrence as the value.

get_observed(self)

This will return the count of the number of occurrences for the segment (language specific).

8 Levenshtein

The Levenshtein takes an iso and a gloss value and learns the best matching standard Leipzig or Gold gloss by Levenshtein distance. If the best match has a distance greater than the MapGloss.max_distance value than the Levenshtein will send the value to the Lexicon class and consider the gloss a word.

8-1 OBJECT CONSTRUCTOR

The object constructor for Levenshtein requires both the gloss and iso values.

__init__(self, iso, gloss)

iso The ISO-639 code for the language to which the gloss pertains.

gloss The current gloss for training with the Levenshtein learning method.

The above __init__() will set the Levenshtein object and add it to the Levenshtein.objects dictionary with the key (iso, gloss).

8-2 LOAD

The load_levenshtein() function will first attempt to load the Levenshtein CLSO file if MapGloss.restart has been set to False. Otherwise it will load a default empty dictionary.

load_levenshtein(restart=False)

restart A boolean value that toggles between loading the CLSO file or the default data structure.

Calling load_levenshtein() initializes the base and/or CLSO file Levenshtein objects.

8-3 EXPORT

The export_levenshtein() function takes all of the objects stored in Levenshtein.objects and dumps them to the 'levenshtein.cslo' file.

export_levenshtein()

8-4 DELETE

The delete_levenshtein() function will take a Levenshtein value as input and delete that value for its class. In the normal map_gloss.py routine this is never called but may have use for those attempting the manual mode.

delete_levenshtein(iso_value, gloss_value)

iso The ISO-639 code for the language of the gloss that the user would like to delete.

gloss The gloss that the user would like to delete.

8-5 SET LEVENSHTEIN

The set_levenshtein() function is a wrapper for the actual learning components of the Levenshtein learning algorithm. The function does some preprocessing checks. If a gloss has already been defined as a word it will not allow processing of the gloss for Levenshtein edit distance because it will assume it is a word. Note that due to the learning method structure if a gloss is already matched to Leipzig or Gold the Levenshtein will not process them (if it did it would just return the gloss anyways because its best match would be to itself).

set_levenshtein(self)

A large portion of the set_levenshtein() function is dedicated toward the manual method. This will print options that ask for user input. The automated method will skip this entirely. The methodology is to call two functions that build the Levenshtein values by comparing all Leipzig and Gold values. Only those tied for the minimum distance and less than the maximum distance threshold will be allowed for comparison. The manual component simply allows the user to select from a set of possible matches in addition to stating that the gloss is actually a word or it is a gloss itself. Otherwise the first minimum tag will be automatically selected. It will then set a _levenshtein value inside the object which will be retrieved by a Gloss object if the object ran the Levenshtein learning method.

8-6 HANDLE LEVENSHTEIN

The _handle_levenshtein() function is an intermediary function between the set_levenshtein() and _calculate_levenshtein() functions. It reranks the what is considered the minimum distance and tied pairs based on what _calculate_levenshtein() returns. It then passes its results back to set_levenshtein().

_handle_levenshtein(self, gloss, lookup, lookup_tag)

gloss The current Leipzig or Gold gloss being tested for a match.

lookup The current minimum Levenshtein distance.

lookup_tag A list of all tied gloss values for minimum Levenshtein distance.

Once it processes it will return a new lookup and lookup_tag pair if the lookup value was lower than the previous or append a value to the lookup_tag if the scores were equal. Otherwise it will return the same value because the examined gloss had a higher Levenshtein distance.

8-7 CALCALUATE LEVENSHTEIN

The _calculate_levenshtein() function is a very standard implementation of the Levenshtein distance algorithm that returns the edit distance between two strings.

_handle_levenshtein(self, a, b)

a Either the actual input gloss or a Leipzig or Gold gloss, whichever is shorter.

b Either the actual input gloss or a Leipzig or Gold gloss, whichever is longer.

This will return the value of the distance between the gloss and the current Leipzig or Gold gloss that the system is testing. Note that trying to cut off the algorithm when it exceeds the current minimum value sometimes leads to errors or incorrect results. The algorithm runs extremely quickly so this theoretical inefficiency does not cost much user or CPU processing time.

8-8 GET VALUES

The Levenshtein class has a set of get functions that return values stored within the object.

get_iso(self)

This will return the ISO-639 value for the language to which the Levenshtein gloss belongs.

get_gloss(self)

This will return the original gloss which represents the object value.

get_levenshtein(self)

This will return a list of the Leipzig, Gold, and author values that the Levenshtein object matched. The author tag will at first always be the gloss input.

9 Lexicon

The Lexicon class most closely associates with the Levenshtein class and is used to indicate that glosses are actually words in the Language. In future versions of map_gloss it is likely that the inference methods for what is a word will improve.

9-1 OBJECT CONSTRUCTOR

The object constructor for Lexicon requires both the gloss and iso values.

__init__(self, iso, gloss)

iso The ISO-639 code for the language to which the gloss pertains.

gloss The current gloss that is considered a lexical item.

The above __init__() will set the Lexicon object and add it to the Lexicon.objects dictionary with the key (iso, gloss).

9-2 LOAD

The load_lexicon() function will first attempt to load the Lexicon CLSO file if MapGloss.restart has been set to False. Otherwise it will load a default empty dictionary.

load_lexicon(restart=False)

restart A boolean value that toggles between loading the CLSO file or the default data structure.

Calling load_lexicon() initializes the base and/or CLSO file Lexicon objects.

9-3 EXPORT

The export_lexicon() function takes all of the objects stored in Lexicon.objects and dumps them to the 'lexicon.cslo' file.

export_lexicon()

9-4 DELETE

The delete_lexicon() function will take a Lexicon value as input and delete that value for its class. In the normal map_gloss.py routine this is never called but may have use for those attempting the manual mode.

delete_lexicon(iso_value, gloss_value)

iso The ISO-639 code for the language of the gloss that the user would like to delete.

gloss The gloss that the user would like to delete.

9-5 GET VALUES

The Lexicon class has a set of get functions that return values stored within the object.

get_iso(self)

This will return the ISO-639 value for the language to which the Lexicon gloss belongs.

get_gloss(self)

This will return the original gloss which represents the object value.

10 TBL

The TBL class represents Transformation Based Learning. This class has not been finished and thus documentation has not yet taken place. The TBL class is another learning method to substantiate the Delphi and Levenshtein methods. Error analysis has revealed that sometimes an author will confuse Leipzig values or choose a really poor abbreviation that will become confused with another gloss during the Levenshtein learning. TBL will follow the Delphi and Levenshtein methods to learn based on a set of training data and correct errors during decoding. It is a somewhat advanced version of TBL in that its target and source are different value sets. Normally the feedback process for target and source is the same. In Brill tagging the gold standard is the correct part-of-speech (POS) tag while the source is the currently inferred version. In inference and glossing methods it is too resources intensive to create gold standard glosses at this point. The TBL will instead use evaluation procedures from choices files as the gold standard and correct inputs by making assumptions about which values were confused. This class is still very much under construction.

11 Morpheme

The morpheme class stores counts of how often a morpheme occurs in each language. This storage extends to counting the number of times each gloss occurred with the morpheme.

11-1 OBJECT CONSTRUCTOR

The object constructor for Morpheme requires the ISO-639 code for the morpheme.

__init__(self, iso, morpheme)

iso Set the iso value for the language.

morpheme Set the morpheme value that was discovered in the current gloss.

11-2 LOAD

The load_morpheme() function will first attempt to load the Morpheme CLSO file if MapGloss.restart has been set to False. Otherwise it will load a default empty dictionary.

load_morpheme(restart=False)

restart A boolean value that toggles between loading the CLSO file or the default data structure.

Calling load_morpheme() initializes the base and/or CLSO file Morpheme objects.

11-3 EXPORT

The export_morpheme() function takes all of the objects stored in Morpheme.objects and dumps them to the 'morpheme.cslo' file.

export_morpheme()

11-4 DELETE

The delete_morpheme() function will take a Morpheme value as input and delete that value for its class. In the normal map_gloss.py routine this is never called but may have use for those attempting the manual mode.

delete_morpheme(iso_value, morpheme_value)

iso The ISO-639 code for the language of the morpheme that the user would like to delete.

morpheme The morpheme that the user would like to delete.

11-5 GLOSSES

The glosses dictionary stores all of the glosses that have ever occurred with the morpheme during training. Expect that there will be more glosses than one and that in general there are more glosses than morphemes. The gloss is the key for this dictionary and the count is the value.

set_glosses(self, gloss)

gloss The gloss that contains the morpheme.

The set_glosses() function is processed automatically from the Gloss class but if the user overrides that automation then it is possible to set gloss values directly with the set_glosses() function.

11-6 OBSERVATIONS

The observations is the total number of times that the morpheme has occurred in the language regardless of gloss. It should equal the sum of all combination values.

add_observation(self)

The add_observation() will automatically increase the count of the morpheme by 1 for the user.

11-7 GET VALUES

The morpheme class has a set of get functions that return values stored within the object.

get_iso(self)

This will return the ISO-639 value for the language to which the morpheme belongs.

get_morpheme(self)

This will return the original morpheme which represents the object value.

get_glosses(self)

This will return a dictionary of all glosses which contain the morpheme as keys with the count of their occurrence as the value.

get_observed(self)

This will return the count of the number of occurrences for the morpheme (language specific).

12 CLSO

The CSLO data format was created by Michael Lockwood to store CLS - class and O - object data. The pickle.Pickler and pickler.Unpickler methods did not suit the needs of multiple class hierarchies. Thus the CLSO format was developed to handle the exporting and loading of class and object data so that subsequent runs to do not have to reprocessed. This format currently is in string format so the user can read the output. Eventually this will be converted to utf-8 encoded bytes.

12-1 OBJECT CONSTRUCTOR

The object constructor for CLSO only requires a file name without the .clso extension which then because the object lookup value.

__init__(self, file, option='r')

> file This is the file name for the CLSO file. When initialized it will be opened with the value of whatever option is. The default will make the CLSO file a reader.
>
> option This sets the type of file manipulator the CLSO file will be. Options include 'r' for read, 'w' for write, and 'a' for append. Currently bytes are not supported but that will be added in subsequent versions of CLSO.

Once the object is initialized any of the functions can be called without further need to open or handle the file.

12-2 LOAD

The load() function will automatically load all of the objects in the CLSO file if and only if the class of the object is located within the current module or namespace.

load(self)

Simply call the load() function and all of the objects will be initialized automatically.

12-3 DUMP

The dump() function allows a user to dump an object and all of its values to a CLSO file.

dump(self, obj)

> obj The object value that will be dumped to a CLSO file.

Note that the entire process is dynamic; simply passing an object will provide all of the information the CLSO class needs to build and then later re-instantiate the object data.

12-4 BULK DUMP

The bulk_dump() function provides a very specific ability to dump multiple objects at the same time. It does require some specific data structure requirements. It will process all of the objects in a class and to do this all of the class objects need to be stored in a class variable called objects. Objects would be a dictionary where values are objects. Keys would related directly to the class itself.

bulk_dump(self, cls)

cls The class value whose objects will all be dumped to a CLSO file.

The cls value must be a string of the class. Python eval() and exec() functions will convert the string to the needed variables and values.

12-5 CLOSE

When a file no longer has use to the user, the user may close the file by calling the close() function. This essentially acts as a wrapper to file.

close(self)

Reference

class MapGloss

training(iso, gloss, morpheme='')
prepare_decoding()
decoding(iso, gloss)
clear_class_data()
load_individual_structures()
export_individual_structures()
load_combined_structures()
export_combined_structures()
set_max_distance(value)
set_user_input(value)
set_user_preference(value)
set_restart(value)
user_addition(gloss)
get_example()

class Leipzig

__init__(self, leipzig, gold, pseudo=False)
load_leipzig(restart=False)
export_leipzig()
delete_leipzig(leipzig_value)
get_leipzig(self)
get_gold(self)
get_pseudo(self)
get_iso_codes(self)
get_iso_list(self)

class Gold

__init__(self, gold, leipzig, pseudo=False)
load_gold(restart=False)
export_gold()
delete_gold(gold_value)
get_gold(self)

get_leipzig(self)
get_pseudo(self)
get_iso_codes(self)
get_iso_list(self)

class Delphi

__init__(self, delphi, leipzig, gold)
load_delphi(restart=False)
export_delphi()
delete_delphi(delphi_value)
get_delphi(self)
get_leipzig(self)
get_gold(self)
get_iso_codes(self)
get_iso_list(self)

class Language

__init__(self, iso)
load_languages(restart=False)
export_languages()
delete_language(iso_value)
get_iso(self)
get_glosses(self)
get_segments(self)
get_morphemes(self)
get_observed(self)
set_references(self)
get_references(self)
print_values(self)

class Gloss

__init__(self, iso, gloss)
load_glosses(restart=False)
export_glosses()
delete_gloss(iso_value, gloss_value)
get_iso(self)
get_input_gloss(self)

get_output_gloss(self)
get_leipzig(self)
get_gold(self)
set_author(self)
get_author(self)
get_delphi(self)
get_levenshtein(self)
set_values(self)
add_segment(self)
set_segmentation(self)
_set_segmentation(self)
get_segments(self)
set_morpheme(self, morpheme)
get_morphemes(self)
add_observation(self)
get_observed(self)
set_references(self)
get_references(self)
print_values(self)

class Segment

__init__(self, iso, segment)
load_segment(restart=False)
export_segment()
delete_segment(iso_value, segment_value)
get_iso(self)
get_segment(self)
set_combination(self)
get_combinations(self)
add_observation(self)
get_observed(self)

class Levenshtein

__init__(self, iso, gloss)
load_levenshtein(restart=False)
export_levenshtein()
delete_levenshtein(iso_value, levenshtein_value)
get_iso(self)

get_gloss(self)
set_levenshtein(self)
_handle_levenshtein(self, gloss, lookup, lookup_tag)
_calculate_levenshtein(self, a, b, lookup)
get_levenshtein(self)

class Lexicon

__init__(self, iso, gloss)
load_lexicon(restart=False)
export_lexicon()
delete_lexicon(iso_value, lexicon_value)
get_iso(self)
get_gloss(self)

class TBL

class Morpheme

__init__(self, iso, morpheme)
load_morpheme(restart=False)
export_morpheme()
delete_morpheme(iso_value, morpheme_value)
get_iso(self)
get_morpheme(self)
set_gloss(self)
get_glosses(self)
add_observation(self)
get_observed(self)

class CLSO

__init__(self, file, option='r')
load(self)
dump(self, obj)
bulk_dump(self, cls)
close(self)