**A Brief Note about the Files in this Package**

As we have discussed previously, the feature engineering step of our system takes a preposterously long time on the full training data. As a result, we’ve included a file **(file.arff)** that has already been run through this process and is ready for modeling. Our shell script, **runit.sh**, will run only the modeling and results-reporting steps. The code for the portion of our system that does the initial cleaning, formatting, and feature engineering is included in **DELIMITING.txt** and **script.r**, and is both referenced at the appropriate steps in the outline of our solution below and annotated in the files themselves.

(**file.csv** is the same data as **file.arff**, formatted differently. This is included in case it proves illuminating to look at, but is not required or utilized by the system)

Thanks, and enjoy!

**About the Problem**

(Author: Kushagra Kumar)

Our task is to identify the characters from the famous TV series Friends based on the multi-party dialogues. Specifically: for each token in a line of dialogue that refers to a person (I, you, woman, coworker, etc.), our task is to correctly identify the referent of that token if it is one of the six main characters (Joey, Rachel, Phoebe, Ross, Monica, Chandler) or else classify it under “Other”.

The training data provided to us are the files friends.train.episode\_delim.conll and friends.train.scene\_delim.conll, such that first set of data is episode delimited while the second set of data is delimited on the basis of scene within each episode. Similarly, trial data files are friends.trial.episode\_delim.conll and friends.trial.scene\_delim.conll, delimited again on the basis of episodes and scenes, respectively but with less number of dialogues.

**Our Approach**

(Author: Marcus Walker)

A common approach to the kind of problem we’re facing is to feed everything into some variety of neural network- however, this approach relies on an abundance of training data. Even with a training set of ~150,000 instances, each of which is a token from the script dialogue, the nature of our problem is such that most of those instances do not correspond to reference tokens. In fact, there are only about 13,500 instances that are used for training, while the other instances provide context.

After reviewing the literature and our own past data adventures, we decided to use the contextual, non-reference tokens to engineer features that would add context to the reference token instances, and then apply some more traditional machine learning methods. Our intuition was that a decision tree would be ideal for the task, since it seemed likely that the features that ought to be checked would vary based on the reference token (e.g., the factors that narrow down the word “you” should be different from the factors that narrow down the word “sweetie”).

We begin with a .conll file (a text file format delimited by a variable number of whitespaces, for reasons natural philosophy dares not to speculate on) and must convert them to a usable format before engineering features and then subjecting the modified dataset to modeling. The classification variable will be *Entity.ID*, which is originally a numerical value corresponding to an individual referent (of ~400 types originally, transformed to 7 to meet the constraints of the problem). The steps we took are detailed below.

1. Convert the .conll file to a tab-delimited .txt file (functionally equivalent to a .tsv) using Python. (See **DELIMITING.txt**)
2. Read the tab-delimited file into an R data frame. (See **script.r** for steps 2-13)
3. Generate a *Sentence Number* feature so that blank instances (originally used to mark boundaries between sentences) can be removed, then remove blank and structural (marking beginning and end of document) instances.
4. Clean and transform the class variable *Entity.ID* so that the codes for the six main characters display as their names instead of their codes (e.g., “306” becomes “Rachel Green”) and as “Other” for all other referents. As discussed before, this transforms *Entity.ID* from a numerical feature with ~400 types to a character feature with 7 types.
5. Perform some simple data cleaning on the *Speaker* feature, which contains inconsistencies such as unintentional extra spaces and characters added to the end of an entry. (e.g., “Rachel Green” and “Rachel Green [“ showing up as two separate speakers).
6. Generate a *Lemma Reference Uncertainty* feature by counting the number of different referents (of 7) a lemma is used to refer to, and then discretizing the counts into three equal width bins (1 = least uncertainty, 3 = most uncertainty). These values are connected to the *Lemma* of each instance.
7. Generate a *Speaker’s Most Frequent Referent* feature by counting, for each speaker, how many times they refer to each of the 7 referents, and then returning the referent they refer to most often. These values are connected to the *Speaker* of each instance.
8. Generate *Previous Speaker* and *Next Speaker* features. For each instance, we look at for the most recent instance with a different speaker and the first subsequent instance with a different speaker, and take those *Speaker* names as values for the new features.
9. Generate *Previous Lemma*, *Second Previous Lemma*, *Next Lemma*, and *Second Next Lemma* features. As with the speaker features in the previous step, these features are generated by taking the values of *Lemma* from the two instances prior, one instance prior, and one and two instances subsequent.
10. Generate an *Implied Gender* feature. This simply checks whether the *Lemma* is “he”, “him”, “she”, or “her” and, if so, assigns a value of “M” or “F” to *Implied Gender*. Otherwise, it is assigned a value of “NA”.
11. The features *Frameset.ID*, *Word.Sense*, *Named.Entity.Tag*, and *Constituency.Tag* (from the original data) are discarded because they will not be used in modeling, and keeping the dataset size as small as it can be is always a good plan.
12. All “String” or “Character” features are converted to “Factor” in R, so that they will properly show up as “Nominal” features in Weka.
13. The modified dataset is written out to .arff format, which Weka prefers.
14. The .arff dataset is read into Weka and the J48 Java implementation of the C4.5 decision tree algorithm (see *J48 Algorithm* section below) is applied with *Entity.ID* as the classification variable. (See **weka.sh** for steps 14-15)
15. Weka writes detailed results out to a text file.
16. All onlookers independently, and without suggestion, provocation, or collusion, erupt in vigorous applause. (See what you are probably doing right now)

The results are discussed in RESULTS-2.txt, but briefly: J48 did achieve approximately 87% accuracy on the full training set, and outperformed every other algorithm we tested- including Random Forest, which had outperformed J48 on our Stage 1 dataset that lacked our engineered features.

**J48 Algorithm**

(Author: Marcus Walker)

J48 is a Java implementation of the general C4.5 decision algorithm.Much like a language model, its essential mechanic is measure and comparison of entropy. Rather than measuring the entropy of tokens based on conditional probability in a corpus, however, it uses a measure of entropy of the class variable (how uncertain is the classification of that variable) to choose data attributes and thresholds of those attributes’ values to partition the data such that the difference in entropy (information gain) is maximized.

In other, possibly better, words: the baseline entropy of the class variable is found by just looking at the proportions of the classifications in the data. Then, each attribute is evaluated for its ability to partition the data into subsets such that those subsets have *lower* entropy than the entire data set. The attribute with the greatest such ability, or the most information gain, is selected and the data is partitioned into subsets according to its values. Then, the process is iterated upon, using each of those new partitions as its own base set of data. Once a point is reached, along each of these iterate branches, where no further partitioning results in a significant decrease in entropy, each leaf of that branch makes its best guess at how to classify the class variable.

The result is a tree of conditional “checks” to make when classifying a new instance. The value of the topmost partitioning attribute is checked, and the corresponding branch of the tree structure is explored. At each node, this same check is performed, until a leaf is reached. The classification from that leaf is applied to the instance, and voila! Classification has been done!

Additional detail: using raw information gain (raw decrease in entropy) biases in favor of attributes with a high number of possible values. This leads to problematic and inaccurate classifications in a lot of cases. For that reason, J48 actually uses *normalized* information gain, which divides the raw information gain by the number of possible attribute values for that attribute.

**Team Member Roles**

(Author: Marcus Walker)

Our first, foremost, and shared role is that of excellent groupmate and committed, good-faith collaborator. We toyed with the idea of one of us taking on the role of slouching layabout, but despite a strong tradition of such a division of responsibilities, we didn’t find it to be appropriate for our particular project.

Additionally, we worked together brainstorming features in a way that is difficult to delineate. Ideas collide and share genetic information seemingly at random before careening off into the abyss, and the final ecology would be unrecognizable to the original ideas that spawned it. Consequently, that work doesn’t show up as a bullet point anywhere, because it would have to show up everywhere.

**William Jaros - Captain of Scripting, Packaging, and Managing**

* Shell scripting all of our disparate code fragments together
* Packaging and management of shell scripts, documents, files
* Fixing oodles of code and dependency disjoints
* Input/Output and Program-Running documentation
* Establishing and managing Git repository

**Kushagra Kumar - Potentate of Python, Formatting, and Fixing Everything**

* Python scripting to convert .conll files to tab-delimited .txt format
* Extensive troubleshooting in lieu of sleeping
* Documentation of Problem Description and Third Party Tool
* Troubleshooting again, because it deserves two bullet points

**Marcus Walker - Mathemagician of Data Frames, Modeling, and Long-Windedness**

* R scripting to clean data and implement feature engineering
* Weka scripting to generate classification model
* Documentation of Approach, J48 Algorithm, Team Member Roles
* Documentation of results analysis
* Worrying about how much sleep Kush gets

**Installing the Software**

(Author: Kushagra Kumar)

Provide the relative path of the directory UMDuluth-CS8761-Shannon on your Linux machine, after extracting it from the tar file. Then execute the following command:

./install.sh

As per the install.sh, the installation step can be divided into five steps:

1. Installing python using:

sudo apt-get install python3.6

2. Installing R using:

sudo apt-get install r-base

3. The zip folder for the third party tool Weka is obtained using either of the two methods:

wget <http://prdownloads.sourceforge.net/weka/weka-3-9-1.zip>

OR

git clone <https://github.com/bnjmn/weka.git>

4. This is followed by a call to unzip, which unzips Weka and creates a new directory weka-3-9-1, within the directory UMDuluth-CS8761-Shannon.

unzip weka-3-9-1.zip -d weka-3-9-1

5. Finally the zipped folder is removed using:

rm weka-3-9-1.zip

**Running the Program**

(Author: William Jaros)

The program can be run by navigating to the correct folder in command line and executing:

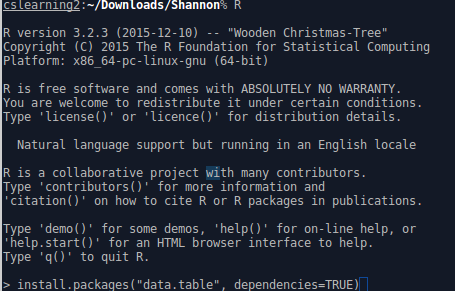
./runit.sh <EnterFileNameHere>

**NOTE:**

After running the above command, an error appears:



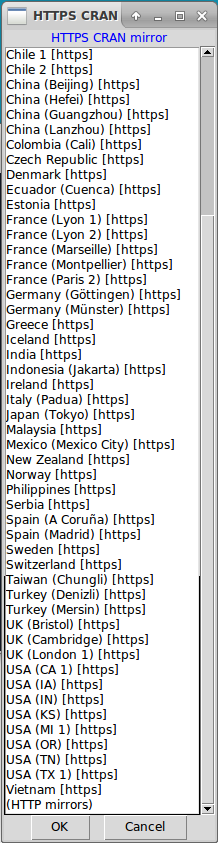
Open R by typing R in linux command line and give the command install.packages("data.table", dependencies=TRUE):



Then it asks to create a personal library for installing package, type “y” two times:



Select USA (IA) as the mirror:



All the packages would be downloaded. This takes some time.

Now run the command and it will produce all the required files:

./runit.sh <EnterFileNameHere>

Note: If more than one file needs to be run it will need to be run twice. The second file data will overwrite the first so changes to that will be changed in our future adjustments.

**Previous Expected Input/Output**

(Author: William Jaros)

**The expected input file should be a dialog separated word for word containing the following columns and data (It looks better in a text file with appropriate columns):**

**SeasonEpisode Scene IDToken ID WordForm POS Tag Constituency Tag Lemma Frameset ID Word Sense Speaker Named Entity Tag Entity ID Sentence**

**/friends-s01e02 3 0 Man NN (TOP(S(NP\*) man - - Rachel\_Green \* -**

**/friends-s01e02 3 1 , , \* , - - Rachel\_Green \* -**

**/friends-s01e02 3 2 I PRP (NP\*) I - - Rachel\_Green \* (306)**

**/friends-s01e02 3 3 never RB (ADVP\*) never - - Rachel\_Green \* -**

**/friends-s01e02 3 4 thought VBD (VP\* think - - Rachel\_Green \* -**

**/friends-s01e02 3 5 I PRP (SBAR(S(NP\*) I - - Rachel\_Green \* (306)**

**/friends-s01e02 3 6 'd MD (VP\* would - - Rachel\_Green \* -**

**/friends-s01e02 3 7 be VB (VP\* be - - Rachel\_Green \* -**

**/friends-s01e02 3 8 here RB (ADVP\*) here - - Rachel\_Green \* -**

**/friends-s01e02 3 9 .. RB (ADVP\*)))))))) .. - - Rachel\_Green \* -**

**The data should include things like word form, POS Tag, Lemma, and speaker name.**

**Although, the data doesn't need to be tab delimited when input - I've included that here**

**so the data is more clear.**

**The expected output creates 5 files:**

**file1.arff**

**file1.csv**

**file1filtered.arff**

**file1J48results.txt**

**tab\_delim\_file1.txt**

**The expected output, predictions, model descriptions and accuracy reports are detailed**

**in the results section.**

**Current Expected Input/Output**

(Author: William Jaros)

Our current method simplifies the Input/Output section of this project. One problem we ran into was the time it takes to run the input formatter (python space\_tab.py) and the r scripts (script.r) on the input. This task took an approximated 15 hours to complete. Because of this time complexity, we decided that in our final submission, we would run the formatting and r scripts beforehand and use the output file from that as our input instead of all the raw data. This allowed for the same data and results to be found, but in a far faster timeframe! To recap, the input in our current method is a properly-formatted arff file that that is provided in our submission. This file has the correct formatting to be ran through weka to obtain the results. Finally, the expected output is a single file (file1J48results.txt) which is further discussed in the results section.

**Third Party Tool (WEKA)**

(Author: Kushagra Kumar)

Waikato Environment for Knowledge Analysis (Weka), developed by the University of Waikato in New Zealand, is a collection of machine learning techniques used for real world data mining problems. Weka had been implemented in java. Weka comprises various tools for data clustering, pre-processing, regression, association rules and classification. It also includes visualization tools.

Weka explorer is an environment for exploring the data provided as input. The input that weka takes is .arff file, which is generated through R as described in the second step of “Our approach”. In order to associate tokens like “I”, “him”, “she” with a certain Entity.ID, first the attribute “Entity.ID” needs to be converted from string to nominal as J48 (classifier in weka) only works on nominal attribute. Filters from weka are used for converting from one data type to another. We used unsupervised filter and then used the attribute option followed by selecting string to nominal.

java -cp ./weka-3-9-1/weka-3-9-1/weka.jar weka.filters.unsupervised.attribute.StringToNominal -R 12 -i file1.arff -o file1filtered.arff

The above command was used to convert the file1.arff to file1filtered.arff such that file1filtered.arff has the attribute “Entity.ID” as nominal. The .arff file comprises three main declarations: @relation, @attribute and @data. @relation tells about the relation name which in our case is:

@relation data2

@attribute tells about the ordered sequence of all the attributes with their corresponding values. In our case, it’s:

@attribute Speaker {'',All,Barry,Carol\_Willick,Chandler\_Bing,'Chandler,\_Joey','Chandler,\_Joey,\_Phoebe,\_Ross',Frannie,Joey\_Tribbiani,Marsha,Monica\_Geller,Paul,Phoebe\_Buffay,Rachel\_Green,Robbie,Ross\_Geller,Susan\_Bunch,Waitress}

@data tells indicates the start of the data segment in the file. Again, in our case it’s:

@data

/friends-s01e01,0,1,'\'s',VBZ,(VP\*,be,-,-,Monica\_Geller,\*,?,?,?,?,?,?,?,?,?,?,?,?,?,1

/friends-s01e01,0,2,nothing,NN,(NP\*,nothing,-,-,Monica\_Geller,\*,?,?,?,?,?,?,?,?,?,?,?,?,?,1

/friends-s01e01,0,3,to,TO,(S(VP\*,to,-,-,Monica\_Geller,\*,?,?,?,?,?,?,?,?,?,?,?,?,?,1

Finally, the data in file1filtered.arff is run through 10-fold cross-validated J48 (classifier in weka) with default settings, classifying Entity.ID. The result with the percentage of correctly identified tokens (like I, him, her) are outputted to file1J48results.txt using the command:

java -cp ./weka-3-9-1/weka-3-9-1/weka.jar weka.classifiers.trees.J48 -t file1filtered.arff -x 10 -c 12 > file1J48results.txt