Natural Language Project 4: Autocomplete Tool for Journalists

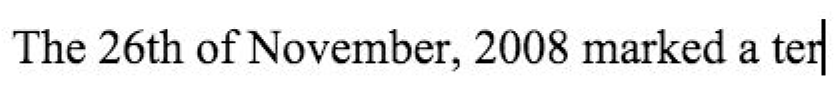
Jake Bass (jab783) and Ishaan Jhaveri (iaj8)

1. Problem or Task

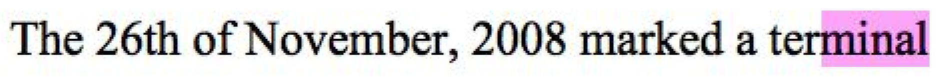
1.1 Overview

We want to design an autocomplete program for a word processor, particularly one that is used by journalists. The program will be made such that it can be installed onto any regular word processor, like Microsoft Word, Google Docs, an editor for LaTeX etc. It will work by suggesting an autocompleted word. After a user has typed one or more characters, the user could select that word, or just continue typing. Here is an example:

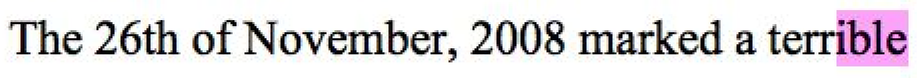
A user begins typing:


The 26th of November, 2008 marked a ter

with the intention of writing the word "terrible". The program might then suggest:


The 26th of November, 2008 marked a terminal

but the user doesn't want terminal so the user continues typing, without accepting the program's suggestion:


The 26th of November, 2008 marked a terrible

At this point, after the user has already typed "terr" the program predicts "terrible", and the user can accept the program's suggestion (by typing some customizable hotkey on their keyboard), and can move on to typing the next word. The example above has an incorrect guess from the program to demonstrate that incorrect guesses do not slow the user down, but correct guesses do speed the user up. Of course in reality we hope not to have too many incorrect guesses!

1.2 Overall Input/Output Behavior

The first step to using our program would be for the user to install it on the word processor of their choice. This is discussed at greater length later on in this report. Our program will start by processing our training data. As we talk about in the ***Data*** section, this will include both a standardized set of data, and some user-specific additions based on their history with the program. This data will be processed to create our classifier. Our program will then be able to provide suggestions for every word as the user is typing. Thus, the output of our program is each suggestion given to the user. In the previous section we have shown examples of how this will look in practice.

1.3 Our Motivation

Taking this class and learning about NLP algorithms, we have begun to notice how ubiquitous they are in our lives today. In particular, we see our iPhones suggesting autocompletion for words as we are typing them. We looked around online for a research paper or white paper on how the iPhone comes up with such suggestions, but couldn't find too much more than speculation by other people[1], who like us were interested in this technology, presumably because private corporations want to hide their proprietary advances in the field of an effective autocompletion tool. So we thought, since we now have a sufficient understanding of the algorithms and the tools to make our own autocomplete software we would devote project 4 to conceptualizing our own attempt at making such an autocomplete software.

The reason we chose the realm of journalism was threefold. Firstly, we figured that journalists are often in a situation where they need to type furiously, so a software that aided in this process is definitely desirable to them. We figured it could help them type faster, and it may have the auxiliary benefit of helping them make fewer spelling mistakes while typing a long stream of text, which would speed up the editing process for them. Secondly, we wanted to be able to think about some interesting features over and above the features we have already put substantial thought into over our other assignments. We thought that since Journalism deals with real world and often current events, it would be interesting to see how we could incorporate context from these things into the training corpus, so that they inform our suggested autocompletions too. We delve further into the discussion of such features in the section on ***Methods***. Finally, Autocomplete limits the task to only perfectly spelled words, which is why we choose the realm of journalism - to deal with user preferences of incorrectly spelled words, or adding new idiomatic words or specific vernacular to the dictionary, is an incredibly complex extension that deserves to be handled separately in its own project. However, there is a very

[1] http://www.slate.com/articles/technology/technology/2010/07/yes_ill_matty_you.html

natural intersection between these two problems and a successful software would likely integrate them both.

2. General Approach or System Architecture

2.1 General Approach

We first began thinking of this problem as a language modeling problem. Each token is a letter, and each n-gram is a partial word. Then predicting the rest of the word becomes a matter of predicting the next n-gram until the word is completed. For example:

Say you begin typing “he” with the intention of writing “hello”. The program sees the “he” and tries to predict the next token. If your model uses unigram probabilities it will probably predict “e” (leaving you with “hee”) because “e” is the most common letter in the english language. Thus, unigram probabilities is probably not the best model to use. With bigram probabilities it will predict the token that most often follows “e”, which may, for example be “a” (leaving you with “hea”), and so on till you reach the end of a word, say “heap” for example. In this case, such a word is incorrect, but a bigram or other n-gram model may have some limited utility.

But we thought, why not keep changing the model as more of the word is typed, or predicted, to improve performance. So when the program sees “he” typed by the user, it immediately uses trigram probabilities to predict the next possible letter, so it predicts the letter that most often follows “he”, giving a much higher probability of choosing “l”. Once it has chosen “l”, it now looks at its input as “hel” and sets about using 4-gram probabilities to predict the letter that is most likely to follow “hel”, and will probably settle on “l”, and so on. At each letter of course a valid prediction is the end of the word.

We thought this would be a good model for correctly predicting a correct word in English, but this model suffers from the obvious flaw that the program when using this model will always predict the same complete word for the same partial word. So “he” will always be completed as “hello” and never “hey”, “heaven”, or even just “he”!

To get around this, we needed to incorporate some more information with the input partial word. In fact, what to use as this additional information is the bulk of our project, as it occurred to us that it was pretty trivial to use a language modeling program as above, what would really be interesting and novel is gleaning as much information as possible about the context of a partial word, so as to give the best possible prediction.

It is worth noting that we discussed the possibility of using a Markov Model with letters as states but quickly dismissed the idea because, among other things, of the volume of data, and as a result the low and computationally intense observed probabilities.

2.2 System Architecture

The flow of our program from the input to the output is detailed at the end of the section on the ***Problem or Task***. As far as the high level system architecture goes we would have a module that extracts the feature vectors from the training set as well as the article the user is currently typing, using an extracting technique with Java and XML talked about below. The extracted input feature vector is passed to a module that computes the best predicted completion, and this module interfaces with the display on the Word Processor to display the suggested autocompletion to the user.

3. Data and Data Annotation

Our project will use "The New York Times Annotated Corpus" from the University of Pennsylvania's Linguistic Data Consortium (LDC), located at https://catalog.ldc.upenn.edu/LDC2008T19. This corpus contains "over 1.8 million articles written and published by the New York Times between January 1, 1987 and June 19, 2007". This will be more than enough training data for our algorithm. In fact, we plan on reserving some of this data for a validator that is talked about in ***Evaluation*** section. Because this data has already been cleaned etc., no manual annotation will be required. However, we do plan on making use of existing modules to classify the Part of Speech of each word. Additionally, all the data in the corpus in formatted in the News Industry Text Format (NITF), which comes in the form of .xml files that are accompanied by java tools built specifically to parse the data. We would employ these java tools to extract the raw text from the .xml. (We would only extract the text, as all metadata and other information, such as the title, author etc. are not useful for us.) We believe the New York Times is a perfect corpus for a program that is targeted to journalists, and will contain an accurate representation of the co-occurrence, collocation and other features that we will be looking for in the partial words we will be trying to predict when actual journalists are using our program.

One interesting addition to the training data is that every time a given user completes an article, that article is added to the their specific training corpus and the model is recalculated with this newest addition to the training corpus. In this way, each user will gradually accrue a personalized autocompletion tool that is optimized for their own uses and style. However, there are two challenges to tackle with such an approach. Firstly, how should we actually capture the influence of the user's own writing style? Simply adding 1, 2, 10 or even 1,000 of their articles is going to make almost no difference to the probabilities used in the model since they have been calculated over millions of articles. We would experiment different ways of doing this - we could add each user article to the corpus thousands of times, or we could create two separate classifiers (one with the original training data, one with the user-specific training data) and aggregate the results. The second challenge is formatting a writer's article in the same NITF format the the LDC uses for our other data. We would use python's nitf module to do this.

*(**Note**: we have attached a sample of our data as a .xml file in NITF format. As noted in the project description, we will not be able to access the actual data until we have subscribed. Consequently, we have done our best to explain the data above and attach the sample to show the format in comes in. Additionally, we have attached two more files from NITF.org. One is a raw xml file in NITF format, and*

the other is a styled HTML file made from the raw xml. This shows the transition that occurs in storing an article in NITF style. A further description of these files is given in the appendix.)

4. Methods and System Development

4.1 Introduction to Methodology

As we discussed above, we need to obtain contextual clues to inform our predictions for autocompleting partial words because we can't use only the Language Model (LM) discussed above. In this section we will discuss our approach for obtaining and using this context, and then conclude the section with how to use the already discussed LM to enhance our classifier.

We looked at multiple types of machine learning (ML) algorithms. After quickly deciding Markov Models would not be suited to this problem, we moved on to considering ML algorithms that rely on supervision to extract feature vectors and then make predictions. We decided that we can frame our problem as a ML problem pretty easily, using the context from a partially typed word as an input feature vector which we then have to classify, where the prediction or classification is just a predicted completion for the word. It is interesting to use this approach because once the user has typed 1 or 2 characters, the possible words that can be suggested as an autocompletion gets immediately limited to the words in the training corpus that start with those same letters. Here is an example of this:

Consider the following very simple corpus of all possible words that can be predicted as a completion for a partial word, only for the purpose of this hypothetical example:

CORPUS: {“hello”, “hey”, “heaven”, “egg”, “hat”}

Now imagine a user begins typing “he”

Immediately the corpus gets shrunk to {“hello”, “hey”, “heaven”}

and the classification task becomes one of classifying whether the program should predict “hello”, “hey” or “heaven”.

Thus, since the program would constantly be classifying only from subsets of the entire training corpus, we believe using a multi-class ML classifier, either a Multinomial Naive Bayes Classifier or Logistic Regression (we would test to see which one performed better) is a viable approach to obtain probabilities for a particular completion, which can then be incorporated with the LM probabilities to output the ultimate prediction. Note, we could treat it as a problem of classifying the entire completion, or even just classifying one letter at a time - we would try both approaches and compare them.

For example:

Imagine the same corpus as above. Now, a user begins typing “he”. The ML approach could either try classifying “he” as a partial word for “hello”, “hey” or “heaven” directly, treating these three labels as the possible predictions, and then suggesting the appropriate one to the user as an autocompletion.

However, it could also work where it sees “he” and then only tries to classify what the next letter should be, where the options are “l”, “y” or “a” (because these are the only possible next letters in the corpus for words starting with “he”). Say it finds that “l” is the most likely next letter, and classifies “l”, then it repeats this problem, now with “hel”, and so on till the end of the word.

We would try both approaches and see which one gave us better results.

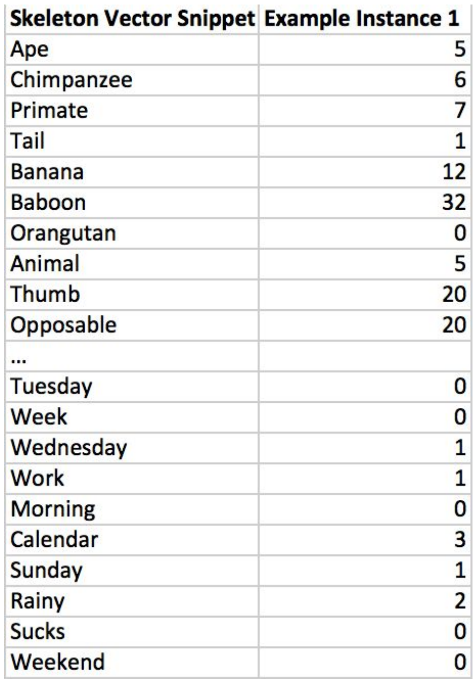
We use the rest of this section to talk about exactly what context we will extract from each partial word. The context is in the form of features that will go into the input feature vector which the program then has to classify.

4.2 Our Features

As we previously mentioned, our features involving language modeling will be discussed at the end of this section. We have two main feature types in our vector: collocational features, and co-occurrence features. In addition to those, we also discuss features based on the user’s geographical location, their previous history, and trending topics in the journalistic world.

4.2.1 Co-occurrence Features

Our main co-occurrence feature will be commonly co-occurring words. To extract commonly occurring words, we would first create a dictionary between each article and another dictionary. This dictionary will contain every word in that article and its count. Then, for a given label, we use the previous data structure to find the ten other words that occur most frequently in articles that contain the given label. (An essential part of this and all future calculations will be our noise filtration system. Words like “the” and “a” will obviously appear a massive number of times. However, as we have found from previous projects, there are other words that surprisingly act purely as noise. We will ignore all of these words. Choosing these words carefully will be essential for both the accuracy and complexity of our classifier. Our list is shown in the appendix. This set of ten key co-occurrence words will be added to our vector for each label. If a word appears in the ten word list for multiple labels it will still only appear as one feature, as it



Skeleton Vector Snippet	Example Instance 1
Ape	5
Chimpanzee	6
Primate	7
Tail	1
Banana	12
Baboon	32
Orangutan	0
Animal	5
Thumb	20
Opposable	20
...	
Tuesday	0
Week	0
Wednesday	1
Work	1
Morning	0
Calendar	3
Sunday	1
Rainy	2
Sucks	0
Weekend	0

makes no sense to have repeated words. Here is an example:

Consider the situation where a user is in the middle of an article and is writing "... mon" with the intention of writing the word "monkey". Above we have a snippet from the feature vector that will be generated for "mon". The skeleton vector shows what the features denote, and Instance 1 is the feature vector for "mon". The ellipses in the middle just show that the feature vector has other slots as well that are not shown. Clearly from this vector we can see that the words that the model treats as co-occurrence features of "monkey" occur much more than the words that the model treats as co-occurrence features of "monday" (here we are assuming a simplified case that the model is only trying to predict either "monkey" or "monday"), so clearly there is a much higher probability that the model will predict "monkey".

In addition to extracting features for commonly co-occurring words, we also would extract features for commonly co-occurring Named Entities (NE). There are two ways in which we would try to do this. The first method that we would try is essentially recreating the above method for common co-occurring words, but with Named Entities. Instead of finding the ten most common NEs, we would likely choose a number more like 3 or 5. However, we would experiment with what value is the best for both common words and Named Entities. The other method we would try for Named Entities is as follows. Instead of finding the most common NEs for each word, we would simply search the entire corpus for the fifty (this number would also need to be experimented with) most commonly occurring NEs. Then a slot would be added to the vector for the count of each of these words. We hypothesize that this method would be more successful, but we would certainly try both.

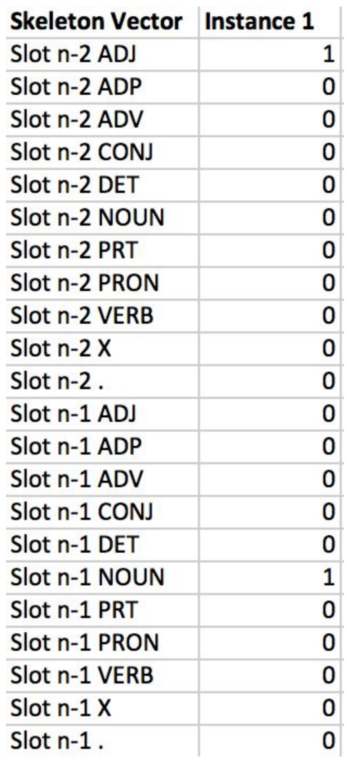
The next co-occurrence feature we would add would attempt to account for is the geographical location of the user (assuming we have access to this information). In order to do this, we would have to find some number of words that are commonly used based on this location. (E.g. Manhattan, Central Park, or subway might all be commonly occurring words for someone living in and writing about New York. Or, Congress, Senator, and constitution might be common for someone living in Washington D.C. and consequently covering politics.) There are two ways that we would try to cultivate this list. Firstly, we could restrict the corpus to articles from the same location and find the most commonly occurring words in those articles and just add n of them to the vector (once again we would experiment to find the best number n). Alternatively, we would consider creating the list manually, or at least using manual intervention or additions to the generated list for each location.

The last co-occurrence feature we would implement is adding features based on worldwide trending topics. This feature would account for any major events that would naturally be the talk of the whole world. We would extract these features by trying the same two methods as we would for the geographically-based features. This feature would also have a large amount of crossover with the geographically-based features, so they may both become more effective as predictors if we integrate them. We would do this by having various spheres of influence where features apply. For example, we could have occurrence features be added at the city-, state-, country-, continent-, and world-wide levels.

If the way we tackle co-occurrence leads to a vector that becomes too large and computationally intractable, we will need to adopt methods to reduce the size of the feature vector. We could either reduce the number of co-occurrence features for each word, or make it so that only the n most common words have co-occurring features in the feature vector skeleton. If we did this, other words would simply not have co-occurring features in the vector skeleton. We would make such decisions after testing to see how long the current approach would take.

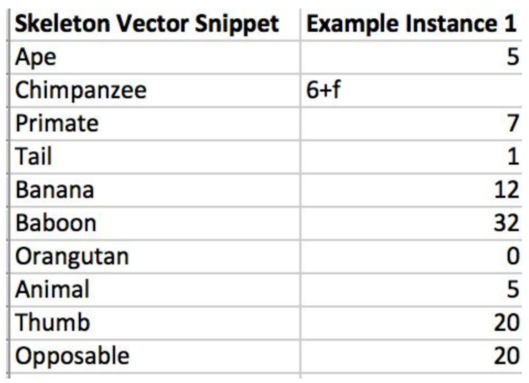
4.2.2 Collocational Features

We extract information about the part of speech and actual word of the words preceding the target partial word to be completed. We would use Google's sentence splitter (discussed further in the section on ***Implementation***) to split the articles into sentences, to ease the task of figuring out which words preceding the target word are actually in the same sentence (We do not consider collocation features over words from different sentences).



Skeleton Vector	Instance 1
Slot n-2 ADJ	1
Slot n-2 ADP	0
Slot n-2 ADV	0
Slot n-2 CONJ	0
Slot n-2 DET	0
Slot n-2 NOUN	0
Slot n-2 PRT	0
Slot n-2 PRON	0
Slot n-2 VERB	0
Slot n-2 X	0
Slot n-2 .	0
Slot n-1 ADJ	0
Slot n-1 ADP	0
Slot n-1 ADV	0
Slot n-1 CONJ	0
Slot n-1 DET	0
Slot n-1 NOUN	1
Slot n-1 PRT	0
Slot n-1 PRON	0
Slot n-1 VERB	0
Slot n-1 X	0
Slot n-1 .	0

We have two types of collocational features. The first one is the part of speech of the words preceding the target word. By target word, we mean the incomplete word that we are attempting to autocomplete. (As an additional caveat, here we are only considering words that appear in the middle of an adequately long sentence. We discuss the edge cases regarding words that occur at the beginning of the sentence at the end of our feature section.) We did this with a list of 12 possible parts of speech taken from nltk.pos_tag, including a marker for an unknown part of speech and for punctuation. We added in the POS by, for a given slot (for example the word immediately preceding the target word would be referred to as index n-1, the one two before would be n-1 and so on), there is a feature for each part of speech. The feature is assigned the value of 1 if that slot contains a word of that part of speech, and otherwise is assigned a 0. We would experiment with different window sizes, trying 1 word before, 2 words before and so on. Shown to the left is the feature vector skeleton and the actual feature vector in the situation that a user has typed "green grass i" with the intention of typing "green grass is". Instance 1 refers to a snippet from the feature vector for the "i" for which the user needs to predict a completion. Since "green" is an ADJ, there is a 1 at Slot n-2 ADJ and a 0 in all the other n-2 slots and since "grass" is a NOUN there is a 1 in at Slot n-1 NOUN and a 0 in all the other n-1 slots. Following this approach with collocational words' parts of speech allows us to incorporate the categorical parts of speech of nearby words feature into our current model which uses multinomial feature vectors.



Skeleton Vector Snippet	Example Instance 1
Ape	5
Chimpanzee	6+f
Primate	7
Tail	1
Banana	12
Baboon	32
Orangutan	0
Animal	5
Thumb	20
Opposable	20

The second one is the actual word that occurs before. We would tackle this in 2 ways and compare each of their results. The first is, if we saw a co-occurrence word in the window we would just add a fixed factor to the count, giving it much more weight as a co-occurrence feature, because it was also collocated with the partial word we are trying to predict. Shown to the left is an example of this approach. Using the example from above of trying to autocomplete the partial word "mon" imagine if the word Chimpanzee had co-occurred 6 times already in this article, but the most recent occurrence was the previous word, like for example the user was typing "... Chimpanzee, mon" then the count of Chimpanzee would increase from 6 to 6+*f*, for some *f*. The ideal *f* would be determined by experimentation and would be different depending on whether it was the preceding word, the word 2 words before, 3 words before and so on.

Another way we could treat co-location words is letting the words themselves be the features, and then using a categorical model distinct from the main model to give another supplemental probability of what the predicted word is likely to be. We conceptualize such a categorical classifier briefly below.

4.3 Edge cases

As we stated earlier, we must have a way to handle all of the edge cases related to words that appear near the beginning of a given sentence. To do so, when attempting to classify a word, we would record its index in the sentence (with index being in the range [0, len of sentence-1]). We also will have tested to find the optimal number of collocational features to include. For the sake of example, say we found that including the five words before the target word gave the best results. Now, when attempting to classify one of the first five words in the sentence, we would remove all collocational features that would not exist. For example, say we were attempting to classify "Ja" in the sentence "My name is Ja…". We would eliminate the collocational features referring to four and five slots before the target word. Thus, the target word "Ja" would only use "My", "name", and "is" for comparing with collocational features from our training set. In this way, we would only allow for relevant comparisons. While this does mean there is slightly less data to classify words that appear early on in sentences, we do not suspect this will substantially affect our results.

Another edge case we would attempt to handle is the capitalization of words that start a sentence. Using the previous example ("My name is Ja…"), "My" is much more frequently seen as "my". Consequently, this discrepancy would likely decrease the accuracy of our classifier. To account for this fact, we would first try to see if first word of every sentence is a Named Entity. If it is, we would leave it. If it is not, we would un-capitalize it. We believe that this un-capitalization would greatly help our classifier.

4.4 Categorical classifier

As mentioned above, we want to test out whether moving collocational features for the actual word that is collocated with a partial target word, to a separate categorical classifier would be better than including it by adding a fixed factor to the co-occurrence feature for that word. This has the added benefit of using words that are collocated with the target partial word that are not just limited to the already specified co-occurrence features, but can be any word at all. We would incorporate this by having a separate model where the skeleton vector has slots for the preceding n words' actual word, and then we use a categorical ML classifier instead of a multinomial one to classify an input vector with a predicted autocompletion.

4.5 Integration of Language Modeling

We have detailed how we would obtain probabilities for partial words using the Language Modeling (LM) approach as well as the feature vector approach. In this section we discuss some ways we would test out combining these two probabilities.

1. We could not combine these probabilities at all and only go with the machine learning (ML) approach - we have discussed above why we cannot only go with only the LM approach. If this gave us the best results, we would stick with this.
2. As discussed above, we could either use the ML approach to predict a whole word or only predict one letter at a time. If we only predicted one letter at a time, we could obtain a probability for each possible next letter from both models and then aggregate the two probabilities to get the final probabilities of each next letter, and use the one that has the highest probability. If we use the ML model to predict a whole word, we can derive an LM probability of this word as if each predicted letter had been predicted independently with an increasing n-gram each time as discussed above and then aggregate this probability with the probability of the prediction given by the feature vector model and then compare these probabilities for the final prediction.
3. Finally, we could include the LM probabilities as features somehow in the feature vector. This is a complicated approach which we think is out of the scope of this particular project, but could be an area of further research after actually completing this project.

4.6 Further Discussion

It is worth noting that the features that deal with worldwide trends and geographical location are currently only incorporated by manipulating our co-occurrence features. The only thing we do to incorporate user history and tendency is add the user's own articles to the training corpus. We considered adding features to account for this, but decided that this was also out of the scope of our project. However, it could a promising area for further research.

There is one other feature that our product would have that we haven't yet discussed. As the classification of words is a difficult task, we would expect that sometimes our classifier will calculate very low probabilities for every possible word. This would indicate that our model does not have a good idea of what word the user is attempting to type. To handle this case, we would introduce a probability threshold to our product. If the probability of the best prediction is too low (i.e. below our threshold), we would not display a prediction to the user. For example, when the user has started a sentence and has only typed the letter "T", it will be very difficult to predict what word they are typing. Therefore, we think our product would be better if it does not offer any suggestions when it does not have a good one. We would experiment to see at what level the threshold best augments the experience of our users.

5. *Implementation*

We will list the existing packages and libraries that we would make use of. Everything else in our project would be implemented from scratch.

We would make extensive use of the NLTK toolkit. We would use this for parsing the article into individual words for the corpus of possible autocompletion predictions and parsing individual words into individual letters as tokens to compute the n-gram probabilities of tokens at the letter level. We would also use it to part of speech tag the words in the articles, for use as collocational features.

For Named Entity Recognition, we would try both NLTK and python's NER package, because NLTK NER package does not recognize times and dates. If we don't like the performance of python's NER package, it would be easy to extend the NLTK package to include times and dates.

For the Machine Learning algorithms, we would write our own software to extract the feature vectors, but we would use Python's scikit learn library to actually perform the Naive Bayes classification or Logistic Regression.

For sentence splitting, we would use google's splitta library.

We would also have to integrate the java tools offered by the LDC (who we got our training corpus from) that assist in parsing xml documents.

As we talked about in the ***Data*** section, we would use python's nitf module to put the user data into an xml file in NITF format so it can be integrated with the rest of the training data.

Ultimately, we envision our product as an installation in the word processor of the user's choice. We would have to do additional research to figure out the easiest way to integrate our program with the APIs offered by the most common word processors like MS Word, Google Docs, etc.

6. Evaluation

As mentioned in the ***Data*** section, we decided to set aside some (roughly 15%) of our data for validation purposes. By doing this, we have an easy way to test iterations of our classifier throughout development. However, the data must be preprocessed in a way that simulates actual user data of partially typed sentences. We do this by taking the roughly 300,000 articles that were not used as part of the training data and breaking them up into sentences (as described in the ***Implementation*** section). Then, for each sentence, we proceed to break it up at a random point. We would do this by taking a random integer in the range of [1, length of sentence as a string-1]. We chose this range as it prevents picking an empty string. Then we check if the integer refers to an index in the middle of the word. We will say it is the middle of a word if both the character at the index and before the index are letters, not spaces or punctuation. If the index does refer to the middle of a word, we use it. If it does not, we pick another index randomly until we get an appropriate one. By ensuring that we are truncating in the middle of a word, we are guaranteeing there is a word to complete. Our algorithm is not designed to be used on complete words (i.e. trying to complete a word that is already completed). Then, we remove every character starting at the index (so the substring from [0, index) remains). We have now effectively created a partially typed word that is ready to be autocompleted. Since the words are chosen randomly, there will be a good distribution of words that begin a sentence (recall, the feature vectors for such words are different) and words that occur elsewhere in a sentence.

For each partial word, we would record what the actual complete word was from the training set and then compare that to the completed word suggested by the program, and obtain an accuracy which could then be used to judge which experiments lead to better results.

7. Conclusion

We see our proposed project as being an incredibly valuable tool for 21st century journalists. We have proposed a large variety of possible ideas and implementations for our classifier. While some may be unwieldy or prohibitively computationally complex, we won't be able to tell how this may limit our performance without testing. Additionally, we have addressed this problem by proposing some solutions to these potential shortcomings. We see the wide variety of possibilities offered as a strength of our report and as a catalyst for further innovation.

Finally, through the course of this project we have proposed some things that are out of the scope of what we would attempt for this project, but could be pursued as areas for further research. These are:

- Researching the best way to include the Language Modeling probabilities as a feature in the feature vector instead of trivially aggregating the LM probabilities with the ML probabilities
- More sophisticated learning from the user's habits. For example, if the user rejects the same prediction many times, the algorithm should adapt in some way.
- Changing the way we deal with user preferences, geographical location, trending topics etc. as discussed at the end of the ***Methods*** section.

8. Individual Member Contribution

We conceptualized and specified the problem together. Jake came up with the idea for something related to autocorrect or spell-check, Ishaan helped zero in on autocomplete. Ishaan began following the Language Modeling approach and Jake urged us towards combining this with some supervised Machine Learning algorithms. We found the data and conceptualized our feature vectors together. As for the sections, Ishaan wrote the bulk of sections 1,2 and 8, Jake wrote the bulk of sections 3 and 5. For section 4 Ishaan wrote about collocational features and combining the Language Model with the feature vector, Jake wrote about co-occurrence features and edge cases. For section 6, Jake wrote about sentence splitting and Ishaan wrote about the validator. We wrote section 7 together.

Appendix

List of Noise

Note this is a partially modified list of noise from our project on Word Sense Disambiguation. The noise list for this task will naturally be different, but of the same form, thus we have included this list as an example.

['the', 'and', 'or', 'it', 'what', 'is', 'was', 'be', 'at', 'by', 'its', 'not', 'until', 'very', '' 'to', 'of', '.', ',', "'s", 'for', '-', 'this', 'as', 'a', 'an', 'but', 'with', 'are', 'for', 'from', 'further', 'get', 'gets', 'got', 'me', 'more', 'most', 'mr', 'my', 'near', 'nor', 'now', 'of', 'onto', 'other', 'our', 'out', 'over', 'than', 'that', 'the', 'their', 'them', 'then', 'there', 'thereby', 'therefore', 'therefrom', 'therein', 'thereof', 'thereon', 'thereto', 'therewith', 'these', 'they', 'this', 'those', 'through', 'thus', 'to', 'too', 'was', 'we', 'were', 'what', 'when', 'where', 'whereby', 'wherein', 'whether', 'which', 'while', 'who', 'whom', 'whose', 'why', 'with', 'without', 'would', 'you', 'your', 'then', 'that']

Attached Files
ldcDataSample.jpg -- a picture of the sample data given by the LDC
rawNITFstyle.xml -- a sample NITF style xml file
articleGeneratedFromNITF -- This is a styled HTML file that shows the article that was generated from the previous sample NITF style xml file. It is accompanied by a folder with pictures in it for the article.