**Examples collection procedure:**

All the data has been collected from Wikipedia in English and Dutch languages. Process was to pick up multiple paragraphs of text from any current page – so as to have samples which can belong to the same writing style and choice of words to ease tree learning with multiple examples. To have variance in data distribution to match up with any real-world or test case scenario model can face, care was taken to choose chunks from multiple pages which can be written by multiple authors. Also, Pages were chosen from multiple sections of Wikipedia, such as pages related to history or computer science or current events etc. to have diversity in the data collected.

All data was collected manually and put into a two file one for English and another for Dutch.

Did some basic data cleaning such as removing some special characters which could be properly encoded by text editor or by file reader function in Python to avoid any issues in training.

Also, some basic work was done to avoid citation marks and numbers form the text chosen since numbers do not help us in classifying languages such as English and Dutch since they use same numeric notations.

All collected data was then read by a program which can split it into the samples of length 10, 20 or 50 and saved to different files. Also, Reader function appended information about what language current text belongs to, based on from which file data was read.

Below is the specifics about the distribution of dataset. Dataset Files have also been provided with names:

*Training files:*

ten\_words\_sample.dat

twenty\_words\_sample.dat

fifty\_words\_sample.dat

*Testing Files:*

ten\_words\_test.dat

twenty\_words\_test.dat

fifty\_words\_test.dat

|  |  |  |
| --- | --- | --- |
| **Dataset Type** | **Train Set** | **Test Set** |
| 10 Words | 440 Dutch, 426 English | 31 Dutch, 30 English |
| 20 Words | 211 Dutch, 213 English | 16 Dutch, 16 English |
| 50 Words | 76 Dutch, 74 English | 15 Dutch, 16 English |

Script create\_train\_sets.py was used to generate these sets.

**Features selection and reasoning:**

**Procedure for selecting features:**

Feature selection has been primarily done with observation. Once we have collected data for English and Dutch languages, we observed which sort of patterns or structure are present in one of the languages and not in other. If there are some patterns, which occur in one of the languages but only occur in another language with lesser frequency, threshold counts have been set , for example, more than 2 occurrence of ‘oo’. Such counts have been chosen by intuition and trial and error. For some of the features like average word length feature, multiple sources like [this](http://www.ravi.io/language-word-lengths) provide analysis of average word lengths in different languages; such insight was used to set the threshold values.

**Feature 1] Presence of the ‘articles’ in English language:**

Reasoning: Articles used in both of the language differ and hence to detect English language, we can check if the data being passed has English articles in it, which are ‘a’, ‘and’, ‘the’.

Behavior: If present, this feature would be set to True, False otherwise.

**Feature 2] Presence of the ‘articles’ in Dutch language:**

Reasoning: Dutch has multiple articles, which should be present in pure English statements. Thus, we can use check of if such articles are present, which would help in discrimination.

Use of two features for checking if articles from one of the languages is present, can potentially give a relatively strong discriminator.

Behavior: If present, this feature would be set to True, False otherwise.

**Feature 3]** Average word length in Dutch language is higher than English, and 9 can be a value to discriminate the two languages with some fair amount. Value of average length 9 obtained from [this](http://www.ravi.io/language-word-lengths). Enlish also has word length just below 9, so this feature is not a strong feature but still can act in ensemble we a weak learner.

Behavior: Returns False if above criteria held, True otherwise.

**Feature 4] Check if common Dutch words are present.**

Reasoning: This feature would check if the passed in sentence has Dutch words which are frequently used but are not in English language. This feature uses list of such words that has been defined with 100 such frequent words to refer from.

Behavior: Returns False if input has one of these words, True otherwise.

**Feature 5] Check for filler words in Dutch**

Reasoning: Filler words are used in any language but are not necessarily frequent words. Dutch has filler words like omdat (Dutch for ‘because’) and van (Dutch for ‘from’).

Behavior: We can use presence of these for as a discriminator.

**Feature 6] Check for common in dutch but rare in English patterns with which words usually end with:**

Reasoning: Patterns like ‘sch’ and ‘tsj’ do occur in end of Dutch words but not in English. This feature checks if content passed in has such words.

Behavior: If so, language can be Dutch and hence return False. True, otherwise.

**Feature 7] Check for sub-strings appearing in the middle in Dutch with high frequency but not in English.**

Reasoning: Dutch words have substrings which appear in the middle of sentence such as ‘zi’, ‘ji’, ‘iz’

Which appear in Dutch with high frequency.

Behavior: This feature checks for such presence and returns False if present, True otherwise.

**Feature 8] Check if input has multiple long words:**

Check if passed in input has more than 5 words with length greater than 8.

Reasoning: Dutch has a lot of long words, which does not happen in English with that much frequency.

Behavior: False if such feature occurs, True, otherwise.

**Feature 9] Check for word-endings:**

Reasoning: Words in Dutch can end with characters such as ‘r’, ‘i’, ‘n’, ‘e’. Such characters at the end of words are rare in English.

Behavior: This feature is set to False, if this is the case and True, otherwise.

**Feature 10] Check for high frequency double occurrence of ‘oo’**

**Reasoning:** Words in Dutch have multiple occurrences with two sub-sequent ‘oo’s. Since such patterns occur in English as well but with less frequency, we can check if such occurrence have count greater than 2.

**Behavior:** Returns False if present, True otherwise.

**Description of Decision tree learning:**

Decision trees help us to model problems where we need to make decision about the class to which input belongs to. It’s assumed that such decisions can be made by successively asking different questions and classifying data successively. After we have asked sufficient number of questions for given data, we get results of classification.

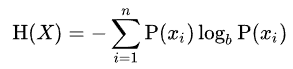
Questions we ask are the features in such modeling. We can ask questions with all the features we have in sequence, but we should not be asking same question twice since it does not help with classification.

Also, if we choose feature to split on at random, we can exhaust all the possible questions we can ask, and end up with an ‘impure’ data which is separable with different decision tree.

For optimal structure, to choose the node to split on, we use entropy and information gain attributes.

Entropy of data is measure of impurity of data, and if good classification is desired, lower the entropy after we asked question to split data on, the better.

Entropy can be given as:



(Reference: <https://en.wikipedia.org/wiki/Decision_tree>)

Entropy calculations in this implementation are done in get\_entropy function().

Information gain is difference between the entropy of the node before and after the split. TO get the entropy after split, we take the weighted sum of the entropies of the children.

For our example of the language classification, features that we need to choose from are defined manually and have been explained in previous sections.

Each of the input samples that we have is preprocessing to convert the natural language text to the feature map, which is a Boolean feature map with True and False.

So, find the split on any feature, we make a decision of if we split on this feature, i.e. with True and False possibilities, how many of the data samples which have this feature True are from English languages and how many are from Dutch. We do the same computation, on the data that can be generated if we were to set the feature under consideration to False.

In this specific implementation, when data samples after split and have English class, they get put into the left node and Dutch samples in the right node.

If we have multiple samples left to choose from, we will find the information gain for each of the samples and then choose the feature with the maximum gain for optimal tree structure for classification.

Above procedure has been coded in ‘get\_best\_split\_node()’ function.

Creation of the entire decision tree is part of build\_dtree() function, which has two base cases for the stopping criteria of tree building:

1] We ran out of features to split on.

2] We ran out of samples to classify.

**Results with decision tree:**

|  |  |  |
| --- | --- | --- |
| **Train-Test Specs** |  | **Test set Accuracy** |
| Train and Test on 10 words sample: |  | 96.77% |
| Train on 10 words and test on 20 words samples |  | 100% |
| Train on 10 words and test on 50 words samples |  | 100% |
| Train and Test on 20 words sample |  | 100% |
| Train on 20 words and test on 10 words samples |  | 95.16% |
| Train on 20 words and test on 50 words samples |  | 100% |
| Train and Test on 50 words sample |  | 100% |
| Train on 50 words and test on 10 words samples |  | 91.9% |
| Train on 50 words and test on 20 words samples |  | 93.75% |

**Analysis of Decision Tree performance on different train and test sets:**

Common pattern that we observe is that we get higher accuracy when model is trained on lower length sequences and tested on longer sequences. Reason for this can be that our features are based on occurrences of certain events in text, probability of which increases as we increase sample length and hence increased accuracy and vice-versa.

**Adaptive Boosting:**

Adaptive boosting learning is based on an assumption that it may not be possible to have good enough combination of all the features in a single decision tree(strong classifier) which is able to classify the data with high accuracy. Rather, if we use multiple simple classifiers, aka weak classifiers) in ensemble, where one classifier tries to get the samples mis-classified by previous classifier correctly classified. Such weak learners are also called stumps and in this implementation, they are one single decision for one of the chosen features.

One weak learner alone is not of much use but in ensemble, with one classifier helping another, such structures are able to learn complex data distributions, even if a single decision tree cannot. Ensemble learners generally provide more generality in the data distribution they can learn.

We still need to learn the weights for how to combine the results from these different classifiers together. We successively update those weights and stop when we converge to weight combination that can help us predict data sample classification with good enough accuracy. AdaBoost formulations define the classifier weight update rule.

Also, in order for one classifier to know which of the data samples are more important to get correctly classified, we will assign weights to the data samples themselves and update those weights depending on whether they get correctly classified or not. Such weights form a probability distribution which sums to 1. If one of the samples gets incorrectly classified then its weights will be increased and relative weights of the ones which were correctly classified will reduce later when we normalize the weights.

In this implementation, we use adaptive\_boosting() function for the procedure explained above.

Other supplementary functions are get\_simple\_entropy\_boost() and get\_best\_split\_node\_boost() which return the entropy values in AdaBoost setting and the index of best feature to create next stump from.

**Changes in entropy calculations in AdaBoost as compared to the regular decision tree:**

We count the number of samples to get classified into one class or another when computing the entropy values for standard decision tree. For AdaBoost, since examples have weights associated and we desire to correctly classify the samples on which another stump made mistake last time i.e. the ones with the higher weights; we use weights of the samples getting classified into one class or another, instead of count. Basic mathematical formulations will remain the same. Such change essentially tries its best to correctly split apart the instances weigh were mis-classified last time.

Above procedure is part of the entropy\_boost() function.

**Changes in finding the best feature to split on:**

The only change in this part is to use new weighted entropy values instead of plain entropy values to determine which of the features is best able to classify current data. And returns the index of such stump.

Above changes have been incorporated in get\_best\_split\_node\_boost() function.

In the implementation, we are using set of 15 weak learners that we can choose.

**An insight:**

From the few inputs checked, stumps 1,2,6 seem to be being used pretty frequently for good predictions.

**Results with AdaBoost:**

**Analysis of AdaBoost performance on different train and test sets:**

|  |  |  |
| --- | --- | --- |
| **Train-Test Specs** |  | **Test set Accuracy** |
| Train and Test on 10 words sample: |  | 91.2% |
| Train on 10 words and test on 20 words samples |  | 93.75% |
| Train on 10 words and test on 50 words samples |  | 100% |
| Train and Test on 20 words sample |  | 100% |
| Train on 20 words and test on 10 words samples |  | 92.1% |
| Train on 20 words and test on 50 words samples |  | 100% |
| Train and Test on 50 words sample |  | 100% |
| Train on 50 words and test on 10 words samples |  | 91.9% |
| Train on 50 words and test on 20 words samples |  | 93.75% |

As we see, Each of the model that has been trained on lower length sequences shows better accuracy on larger sequences. For example, model that has been trained on 10 word sequences shows better performance on 20 and 50 word tests while model of 20 words shows better performance on 50 words test than 20 words.

Possible reason might be that features defined in this implementation check for occurrences of some patterns which might be words, sub-strings average lengths of patterns etc. As we use higher length samples that was optimized to find such patterns in lower length sequences, model can do this job with ease.

However, reverse is not true. If we train the model on longer sequences like 50 and then test on 10 words sample, performance degrades as length reduces. This can be explained by opposite reasoning of increase in accuracy as explained above i.e. model was optimized to find such patterns in large data pool and since feature’s probability of occurrence increases as length does, model does not perform equally well on lower length samples.