Foundations of Intelligent Systems - Lab 2 Write-up Jason St. George
Due 4/25/18

- **<u>Feature Selection</u>**: I chose to select 10 features to split data on and decided that wrapping all special characters and common words into singular features would give better performance than separating them and having less information. (i.e. if á and é were separate features instead of one). My choices are as follows in the FeatureExtraction.py file:

  0 Special Characters: These were chosen on the inference that common English contains few special characters such as umlaut's and that on average, Dutch would contain more.
  1 Common Dutch words: Assume good classification if we can find common words that only occur in Dutch.
  2 Common English words: Assume good classification if we can find common words that only occur in English.
  3 Double U: English rarely contains consecutive double U's in a sentence.
  4 Double K: English rarely contains consecutive double K's in a sentence.
  5 Double A: English rarely contains consecutive double A's in a sentence.
  6 Total double letter count: There exist more double letters in Dutch than in English.
  7 Average Word Length: More commonly, Dutch words are longer than English words.
  8 Contain Q: The letter Q is slightly more common in English.
  9 Contain X: The letter X is slightly more common in English.

- **<u>Decicision Tree:</u>** I took an object-oriented approach to the Decision Tree, which simplified my model over using dictionaries and/or lists as node objects. The class DecisionTreeNode is the main node object that holds the feature split on, default_children (in case it ends up being a leaf node next recursive call) and its actual children (other DecisionTreeNodes or DecisionTreeLeafs). The DecisionTreeNode object has a call method which allows the node object to be called like a function and then classifies a given example taken as argument via recursive tree traversal.

  The decision_tree function takes in the data, weights and calls internal functions that build and train the tree recursively. The action really happens in train_tree, which is the main algorithm modeled after the book pseudocode. A feature is chosen which has the most information gain looking at all the available features and the current entropy of the split and recursively calls the main algorithm again and adds the returned node to the children of the parent who called it, again using the book formulation. The last key function here is weighted_majority, which takes the examples of a node and computes the predicted label for that node based on the weights of the examples inside and returns a DecisionTreeLeaf.

- **Adaboost**: Adaboost takes in a learning algorithm L and the number of iterations K to run the algorithm. In this case, K will be how many stumps to create. We then train a decision stump with an even weight distribution on the first iteration, and decrease the weights for the correctly classified examples, and leave the incorrectly classified examples alone, such that they will be more highly valued next iteration. Then we append the stump to a list, normalize the weights, calculate the hypothesis stump weight and append it to W, a list containing the associated weights for each hypothesis. The algorithm returns a list of hypotheses and weights for each hypothesis so that we can classify test data. I found that K = 10 gave some improvement, but K >> 10 saw no real positive returns, even up to K = 1000. On test.dat, I ended up classifying 8 out of 10 correctly, with an 80% accuracy.

- **Usage**: To run this code, run main.py with cmd-line arguments:
  - main <training_data> <hypothesis_out><learning_type><test><labels>
    - training_data is the filename for the training data I include in the zip (trainingData.txt)
    - hypothesis_out is the filename to which the serialized model will be written
    - learning_type is either ‘ada’ or ‘dt’ for adaboost or decision tree
    - test is the filename for test data you wish to use (here is testData.txt)
    - labels is an optional argument if you wish to provide a new-line separated text file with the correct labels for test data. If labels is provided, then evaluate() will be called in main.py and will print the accuracy (percentage) of the predictions with the given model.