The problem sets in this class will be done using Matlab. You may need to use the computers in the EECS labs to get access to Matlab. If you have problems accessing these computers, contact the GSIs at `gsi-cogsci131@lists.berkeley.edu`.

Collaboration Policy

Class policy is that collaboration is permitted when solving problems, but each student must:

1. Write his or her own code
2. Write the document containing his or her answers independently
3. List the people he or she worked with on his or her problem set

Submission Instructions

Problems will require you to either turn in Matlab code, written answers, or both. All of the files you need are provided for you – you just need to fill them in with your answers. **Make sure to fill in your name and collaborators for each part of the homework.**

All parts that include Matlab code come with a validation script called `validatePN.m` which you can run by calling `validatePN()` from Matlab (replace `N` with the problem number, e.g. for Problem 1, run `validateP1()`). Make sure you run these functions for each part of the homework before submitting your assignment. **If your code does not pass the validation script, you will NOT receive credit.** Note, however, that this is not a grading script; it is entirely possible for incorrect answers to pass the validation script.

You will be submitting your homework on bSpace. For each problem set, you will submit a single zip file containing your filled-in files from each problem. The zip file with all your solutions should be named:

`YourLastName_YourFirstName_ProblemSetNumber.zip`

For example, if your name is Jane Doe, your submitted file should be named `Doe_Jane_1.zip`. You can upload your zip file containing the answer files for the problem set by going to the assignment on bSpace, You can update your submission until the due date. If you make any changes to your submission after the due date, we will regard your problem set as having been turned in at the last time it was changed.

If you use Matlab on the computers in Corey 105 or log in with remote desktop, please note that files saved to the Desktop are removed when you log out. Instead, save any files to your user directory, `U:/`. The `U:/` drive can also be accessed through a shortcut on the Desktop.

Late Problem Set Policy

This problem set is due at the **start of class on February 20, 2014**. You get **THREE LATE DAYS** total for the entire semester, to account for foreseeable minor crises. If you turn in a

problem set late after using these late days, 20% of the total points on the problem set will be deducted for each 24 hours (or portion thereof) that it is late.

1 Learnability (2 points)

We discussed various approaches to learnability in class. In this problem you will explore some of their implications.

Part A

In order to prove his theorem, Gold made several assumptions about language and learning. Identify one assumption that differs from how language is actually learned and one that seems similar. Explain why you chose your answers. Write your answer to Part A in `problem1/writtenAnswersPartA.txt`.

Part B

Probably Approximately Correct (PAC) learning makes different assumptions about language learnability. Identify one assumption that differs between PAC learning and Gold's Theorem. Does this make the assumptions behind the PAC analysis more or less similar to the way humans learn language? Explain your answer. Write your answer to Part B in `problem1/writtenAnswersPartB.txt`.

2 Gold's Language-Learning Game (5 points)

For this problem, you will write Matlab code to implement the language-learning game explained in lecture and described by Gold in his 1967 paper. The game is intended to be a formal model of the way in which people learn natural languages. In the game, there is a learner who is trying to learn the language and a teacher who is supplying the learner with examples of valid sentences from the language. For our convenience, we will represent each sentence with a single number and each language as a set of valid sentences (numbers). In this version of the game, we will assume that there are only 10 possible languages: $L_1, L_2, \ldots, L_{10}$, and that each language L_i contains the sentences 1 through i, so $L_1 = \{1\}, L_2 = \{1, 2\}, L_3 = \{1, 2, 3\}, \ldots$, and $L_{10} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

The game proceeds as follows: the teacher first selects a target language that the learner must learn. The teacher then generates examples of valid sentences from the target language and tells them to the learner one at a time (we will assume that the teacher chooses each example randomly from the set of valid sentences). Each time the learner sees a new sentence, he gets one guess for what the target language is, based on all the observed sentences seen so far. In our version of the game, the game will be over when the learner correctly guesses the target language. The learner's strategy should be to always guess the first language that is compatible with the sentences seen so far. So if the sequence of observed sentences were $(3, 1, 1, 2, 1, 3, 2)$, the learner should guess L_3, and if the observed sentences were $(2, 9)$, the learner should guess L_9.

Make sure you have downloaded all the files from bSpace. The files you need for this problem are in the directory `problem2` and are entitled `goldGame.m`, `selectTargetLanguage.m`, `generateExample.m`, `guessLanguage.m`, and `writtenAnsers.txt`. You can open the file `goldGame.m` by typing `edit goldGame.m` in the Matlab command window. This is the main script used to play the game. Notice the spots in the script where 'THIS IS ONE OF YOUR FUNCTIONS' is written in the comments. It will be your job to finish the code by filling in the definitions for these functions in the respective `.m` files.

Part A

Complete the function definition in `selectTargetLanguage.m`, which randomly selects the target language. The template file is provided to you, so you need to write your code inside the file where it says 'ENTER YOUR CODE HERE'.

The function `selectTargetLanguage.m` takes one argument, `numLanguages`, an integer representing the number of possible languages to choose from. The function has a single output, `target`, consisting of a single integer representing the index of the selected target language.

Part B

Complete the function definition in `generateExample.m`, which randomly generates an example sentence from the target language. The template file is provided to you, so you need to write your code inside the file where it says 'ENTER YOUR CODE HERE'.

The function `generateExample.m` takes a single input, `validSentences`, a vector of the valid sentences in the language (recall that each sentence is represented as an integer). The function should produce a single output, `example`, one example sentence selected at random from the set of valid sentences. You can create random integers with the `randi` function (remember you can type `help randi` to find how to use the `randi` function).

Part C

Complete the function definition in `guessLanguage.m`, which guesses the target language based on all the previously-observed example sentences. The template file is provided to you, so you need to write your code inside the file where it says 'ENTER YOUR CODE HERE'.

The function `guessLanguage.m` takes two input arguments, `observations`, a vector of previous observations of valid sentences from the target language, and `possibleLanguages`, a cell array of vectors. Each cell in `possibleLanguages` corresponds to a single language, a single language is represented as a vector of sentence IDs permissible in that language.

The function should produce a single output, `guess`, an integer index corresponding to a guess as to which language is the target.

Upon completing the functions `selectTargetLanguage.m`, `generateExample.m`, and `guessLanguage.m`, make sure your guessing function eventually gets the right answer by running the `goldGame.m` script.

Part D

Gold's version of the language-learning game is different from ours: in his version, the learner is never told whether he correctly guessed the target language, and the game never ends. The learner is said to have won the game if there comes a point in time where he always guesses the correct language from that point onward. Additionally, the teacher must eventually show the learner an example of every valid sentence in the target language. If we played the game this way, would the language learner that you implemented always be able to win the game? Why or why not? Write your answer under Part D in `problem2/writtenAnswersPartD.txt`.

Part E

Assume that instead of 10 languages, we had an infinitude of languages L_1, L_2, L_3, ..., L_∞ with each language L_i containing all the sentences from 1 to i. If we played this game with Gold's version of the rules described above, would the language learner that you implemented always be able to win the game? Why or why not? Write your answer under Part E in `problem2/writtenAnswersPartE.txt`.

3 Animal Guessing Game (3 points)

In this problem you are going to play an animal guessing game in Matlab. In this game, players are shown some features of a particular animal and the goal is to guess which animal has these features. The player is first shown two features of the animal, and then a list of animals to guess from. If the player guesses incorrectly, more features are shown and they will get more chances to guess the animal. If they cannot guess the animal correctly after all the features have been displayed, then they lose.

Part A

Run the game by running the function `playAnimalGuessingGame.m` in the Matlab command window. This script will play the game for varying numbers of animals to choose from. Specifically, the game will run 15 times: displaying either 2, 4, 8, 16, or 32 animals to select an answer from, and running 3 iterations for each of these settings. The script `playAnimalGuessingGame.m` saves the results of the game, `trialData`, to the current directory upon completion. When finished, `trialData` will be a 3×5 matrix where the entry (i, j) corresponds to the number of guesses it took you to answer the game for the i^{th} iteration where j animals were shown to you to select from.

Part B

In this problem, we'll make a function that plots our results to get a better feel for our data. We'll first examine what the game data looks like on a regular scale and then compare it to the same data on a semi-logarithmic scale. Please complete the function definition in `plotAnimalGuessingGameResults.m`.

`plotAnimalGuessingGameResults.m` takes one input argument, `trialData`, the 3×5 matrix of scores that you get from completing Part A. `plotAnimalGuessingGameResults.m` returns three objects:

1. `meanTrialData` should contain the mean number of guesses per size of hypothesis space from your trial data; this should be a vector with 5 values.
2. `plot1` is our first plot of the results. Using Matlab's built-in function `plot`, write code to plot `meanTrialData` as a function of `hypothesisSizes`, a variable provided for you in the script. Please make sure that the output of your plotting code (within the `plot()` function) is assigned to the variable `p1`. Also be sure to include axis labels and a graph title using the functions `xlabel()`, `ylabel()`, and `title()`.
3. `plot2` is our second plot of the results. Now we will examine the game data using a semi-logarithmic scale for the x-axis. Using Matlab's built-in function `log2`, write code to plot `meanTrialData` as a function of the logarithms in base 2 of `hypothesisSizes`. Please make sure that the output of your plotting code (within the `plot()` function) is assigned to the variable `p2`. Include axis labels and a graph title using the functions using the functions noted above.

In the space provided in `writtenAnswersPartB.txt`, provide a brief written summary of the results (i.e. does the number of guesses increase or decrease when there are more animals to choose from? Is the relationship linear?). Please limit your response to one sentence per plot.

Part C

This game can be analyzed in terms of Probably Approximately Correct (PAC) learning.

Theorem. *For a given hypothesis space* $\mathbf{H}$*, of size* $|\mathbf{H}|$*, there exists finite* n *such that the probability that error is less than* ϵ *is greater than 1 -* δ*, satisfied by:*

$$n \geq \frac{1}{\epsilon}(\log |\mathbf{H}| + \log \frac{1}{\delta})$$

This theorem gives a lower bound for the number of examples, n, that a PAC learner must see to have confidence (1 - δ) that their error is less than an error threshold, ϵ. In our guessing game, consider the different numbers of animals to choose from as a hypothesis space where the number of animals displayed corresponds to the size of that hypothesis space. In the space provided in `writtenAnswersPartC.txt`, explain your results from part (b) above using the PAC theorem.

4 Tversky's Contrast Model (6 points)

Recall from lecture and readings that Tversky defined the similarity between objects a and b as:

$$S(a,b) = \theta\, f(A \cap B) - \alpha\, f(A - B) - \beta\, f(B - A)$$

A is the set of features of a, B is the set of features of b, f is an additive function from sets to numbers, and θ, α, β are free parameters all ≥ 0.

In this problem you will write Matlab functions to implement Tversky's contrast model.

Part A

Complete the function `commonFeatures.m` so that it takes two $1 \times n$ binary feature vectors **a** and **b** as arguments and returns `sumOfCommonFeatures`, the total number of features in common between **a** and **b**.

Part B

Complete the function `differences.m` so that it takes two $1 \times n$ binary feature vectors **a** and **b** as arguments and returns `sumOfDifferences`, the total number of features in **a** that are not contained in **b**.

Once you have implemented and these functions type `sim = tverskySim()` at Matlab's command line to compute the similarities over the animal dataset. Upon completion, this function will output the similarities in the matrix `sim`. If there is an error, use `validateP4.m` to check that your functions from Part A and B are working properly.

Part C

Complete the function `findDeerSim.m` to find animals that are most- and least-like a deer, using to the similarities calculated by `tverskySim()`. `findDeerSim` takes the `names` vector (loaded from `50animalbindat.mat`) as an argument and returns two outputs:

- `deerLike` : A 5×1 cell array such that `deerLike{1}` is the (non-deer) animal most similar to a deer and `deerLike{5}` is the fifth-most similar animal to a deer.
- `notdeerLike` : A 5×1 cell array such that `notdeerLike{1}` is the least similar animal to a deer and `notdeerLike{5}` is the fifth-least similar animal to a deer

As a hint to finding the animals most similar and least similar to a deer, try finding out a way to use the `sort()` function in Matlab.

Part D

Note that the function `tverskySim` (found in `problem4/tverskySim.m`) takes optional arguments, `theta, alpha, beta` where `theta` is the weight assigned to the common features between **a** and **b**, `alpha` is the weight for the distinctive features of **a**, and `beta` is the weight for the distinctive features of **b**. Complete the function `findRhinoAndSquirrelSim` so that it takes `theta, alpha`, and `beta` as arguments and returns `simRhinoToSquirrel`, the similarity of a rhino to a squirrel, and `simSquirrelToRhino`, the similarity of a squirrel to a rhino.

Part E

For `theta = 1.0, alpha = 0.5, beta = 2.5`, is `simRhinoToSquirrel` the same or different from `simSquirrelToRhino`? Why or why not? In the provided template `writtenAnswersPartE.txt`, explain how these results can be interpreted as counter-evidence for Tversky's notion that S(variant, prototype) $>$ S(prototype, variant). Make sure to explain the effect of the model weights.

Hint: Your function `findRhinoAndSquirrelSim` will need to call `tverskySim` to get the relevant similarity scores.

5 Prototype Theory (4 points)

In this problem, you will write Matlab code that implements the prototype theory of categorization using Shepard's universal law to calculate the similarity between pairs of stimuli.

Part A

Complete the function `prototype.m` (found in `problem5/prototype.m`), which takes four arguments:

- `category` : a character string containing feature for which we wish to calculate a prototype
- `features` : the vector of feature names loaded from `50animalbindat.mat`
- `data` : the feature matrix loaded from `50animalbindat.mat`

The function should return a single 1×85 binary feature vector, `p`, with a `1` in column j if the prototype has feature j and `0` otherwise.

The features of the prototype should be determined by a "majority rule" vote between the members of the category: if exactly *half or more* of the animals in the category have a particular feature, the category prototype should have that feature. Otherwise, it shouldn't.

When your function is called, the output should look something like this:

```
>> load 50animalbindat.mat
>> p = prototype('black', features, data)
p =

  Columns 1 through 13

     1     1     0     1     1     0     0     0     0     0     0     1     0

  ...

  Columns 73 through 85

     0     0     1     0     0     0     1     1     1     1     1     0     0
```

Part B

Complete the function `shepardSim`, which takes as input two $1 \times n$ binary feature vectors, `a` and `b`, and returns `simScore`, a value between 0 and 1 representing the similarity between the two inputs. Your function should calculate `simScore` using Shepard's law:

Shepard's "Universal" Law of Generalization. For $1 \times n$ binary feature vectors $\mathbf{a}$ and $\mathbf{b}$ define a function $d : \{0, 1\}^n \to \mathbb{Z}$ such that $d(\mathbf{a}, \mathbf{b})$ is the number of features (positions) by which $\mathbf{a}$ and $\mathbf{b}$ differ.[1] The similarity between $\mathbf{a}$ and $\mathbf{b}$ may be calculated as

$$s(\mathbf{a}, \mathbf{b}) = e^{-d(\mathbf{a},\mathbf{b})},$$

where the codomain of s is the real interval $(0, 1]$.

Hint: Try looking up the logical operator `xor` and thinking about how you could use it to complete `shepardSim.m`

Part C

Consider the subcategory of animals in the `data` matrix (loaded from `50animalbindat.mat`) which are "bulbous". The function `bulbousAnimals.m` (found in `problem5/bulbousAnimals.m`) takes three arguments:

- `names` : the vector of animal names loaded from `50animalbindat.mat`
- `features` : the vector of feature names loaded from `50animalbindat.mat`
- `data` : the feature matrix loaded from `50animalbindat.mat`

Complete the function so that it uses `prototype()` and `shepardSim()` to return three outputs:

- `bulbousPrototype` : the prototype for bulbous animals
- `bulbousSimilarities` : a 50×1 column vector containing the similarity scores between each of the 50 animals in the `data` matrix and your prototype
- `bulbousAnimalName` : the name of the animal which is most similar to `bulbousPrototype`.

[1] Because we are using binary feature representations, $d(\mathbf{a}, \mathbf{b})$ corresponds to the Hamming distance between $\mathbf{a}$ and $\mathbf{b}$