A significant advantage that human cryptographers have over artificial intelligence programs is their ability to understand language. Even if a solver finds a decryption key that decrypts a piece of ciphertext into a string of valid words, it isn’t as good as a human as determining whether or not the words have meaning together as a sentence. Furthermore, the solver might not know every valid word, and it definitely won’t know every colloquial word. Adding in noise creates a huge challenge because there is literally no way to guarantee knowing which characters were and weren’t noised. To help deal with these challenges, we had our solver learn *n*-gram frequencies for both characters and words, since character *n*-grams help validate partial words, and word *n*-grams help validate whether or not a string of words has meaning.

We attempted to solve this problem with two different methods. In our first method, we modeled the task as a search problem. In this search problem, a state is represented by a mapping from letters to letters. For example, the state {‘A’ : ‘B’, ‘M’ : ‘C’, ‘K’ : ‘E’} means we have decided that the letter ‘A’ in our ciphertext should be decrypted to a ‘B’, ‘M’ should be ‘C’, and ‘K’ should be ‘E’. Therefore, our starting state is simply the empty mapping, {}. An action is simply a mapping from a single letter to another letter, such as {‘B’ : ‘G’}. Taking action *a* at state *s* results in a state *s* ∪ *a*. A state is our goal if it has a mapping for all 26 letters in the alphabet, or at least for all the letters that appear in the ciphertext. For a given state, the available actions are all the possible mappings for the first letter that appears in the ciphertext (from left to right) that does not yet have a mapping. The heuristic for A\* search is based on the *n*-grams created from the partial mapping represented by a state. The more infrequent the *n*-grams, the higher the heuristic. Unfortunately, this approach had a significant problem with runtime, so we decided to rethink our approach.

We next tried to model the task as a Bayesian network. The network includes a variable for each character in the ciphertext, a variable for each mapping, and a variable for each word in the ciphertext. All character variables for the same ciphertext letter are connected to a mapping variable, and the character variables are constrained to take on the same value that the mapping variable takes on (i.e. the probability of a character variable taking on value *x* is 1 if the mapping variable took on *x* and 0 otherwise). The mapping variables are all connected to each other, and they are constrained such that they all take on different values. There is also a word variable for each word in the ciphertext, which is connected to each of the character variables corresponding to it. The value it takes on is constrained to be the word formed by joining the values that the character variables take on.

We then used a version of Gibbs sampling to solve the problem. We initialize our mapping variables to a random complete assignment, but we had to modify the repeated step in order to satisfy the constraint that different characters cannot map to the same character. For a variable *X*, instead of computing the weight of the assignment resulting from assigning *X* a letter, we compute the weight of the assignment resulting from switching the values of *X* and another variable *Y* (which may or may not be *X*). We then switch the values with probability proportional to the computed weights. We compute the weight of the assignment by scoring the sentence yielded by decrypting the ciphertext with the assignment. The score is based on a combination of the unigram, bigram, and trigram frequencies of characters, and the unigram and bigram frequencies of words. We used Laplace smoothing for the *n*-gram counts so nothing was ever assigned a score of 0, to account for both valid *n*-grams we lack data on and to account for noise causing invalid *n*-grams.

To help account for noise, we decided to increase the score for word unigrams and bigrams that were very close to valid word unigrams and bigrams. For example, the word “PARLITMENT” would receive the minimal possible score because it’s invalid, but it should get some credit for being very close to “PARLIAMENT”, especially since the middle ‘t’ is likely to have been noised. Therefore, we decided to compute a sentence score using an algorithm based off beam search. For every word in the sentence, we consider the set of valid words that could be generated by noising one letter. We start from the beginning of the sentence and try to assign each word a valid word. For example, the word “NEO” could be assigned “CEO”, “NEO”, “NEW”, or “NET”. We keep a candidate list of partial assignments and continue assigning words to each word in the sentence, keeping only the top partial assignments at each step, where the partial assignments are weighted with word unigrams and bigrams. Keeping the word unchanged (assigning “NEO” the word “NEO” in the above example) boosts the score, while changing it decreases the score. When our Gibbs sampling reaches an assignment that is mostly correct, this beam search is essentially able to guess what the original sentence was without noise.