CS228: Probabilistic Graphical Models

Homework 3

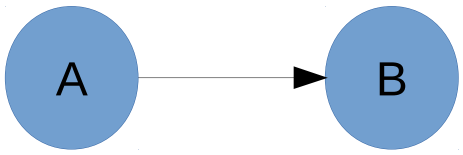
Luke Jaffe

Due: 02/17/2017
Submitted: 02/19/2017

Problem 1: MAP and MPE

Show that marginal MAP assignments do not always match the MPE assignments.

Observe a simple 2-node network $A \rightarrow B$:



Each variable is binary, and is associated with the following CPDs:

P(A)	
A_0	**A_1**
0.4	0.6

P(B|A)		
	B_0	**B_1**
A_0	0.4	0.6
A_1	0.51	0.49

To calculate the MPE, we take the argmax of the joint for each variable configuration:
$P(A,B) = P(A)*P(B|A)$
$P(A_0,B_0) = P(A_0)*P(B_0|A_0) = 0.4*0.4 = 0.16$
$P(A_0,B_1) = P(A_0)*P(B_1|A_0) = 0.4*0.6 = 0.24$
$P(A_1,B_0) = P(A_1)*P(B_0|A_1) = 0.6*0.51 = 0.306$
$P(A_1,B_1) = P(A_1)*P(B_1|A_1) = 0.6*0.49 = 0.294$

max $P(A,B) = P(A_1,B_0) = 0.306$
argmax over (A,B) of $P(A,B) = \{A=A_1, B=B_0\}$

To calculate the marginal MAP for each variable:
$P(A=A_0) = 0.4$
$P(A=A_1) = 0.6$
$P(B=B_0) = P(A_0)*P(B_0|A_0) + P(A_1)*P(B_0|A_1) = 0.16 + 0.306 = 0.466$
$P(B=B_1) = P(A_0)*P(B_1|A_0) + P(A_1)*P(B_1|A_1) = 0.24 + 0.294 = 0.534$

So according to the marginal MAP, the most probable assignment is $\{A=A_1, B=B_1\}$
As we can see, the MPE assignment and marginal MAP predict different values for B, showing that they do not always match.

Problem 2: Variable Elimination

(1) Derive an $O(n^3 d^3)$ algorithm for computing marginals $P(X_i, X_j)$ over all n^2 variable pairs X_i, X_j.

If the total runtime must be $O(n^3 d^3)$, and each of the n^2 pairs is treated separately, we must find an algorithm to compute each marginal in $O(nd^3)$ time.

Since unary factors can be folded into pairwise factors, let us specify the set of factors as follows:

$$\Phi = \{ \phi(X_1, X_2), \phi(X_2, X_3), \dots, \phi(X_{n-1}, X_n) \}$$

To compute a marginal for some X_i, X_j, we will iterate through this chain of factors, performing VE to eliminate the variables in the order $k = 1, \dots, n$, where $k \neq i$, $k \neq j$. Since there are exactly two factors containing most $(n-2)$ of the variables, this elimination will usually involve 2 factors w/ 3 total variables. Since VE takes exponential time in the number of variables, and there are 3 variables in $(n-2)$ VE steps, and each variable takes d possible values, it will take $O(d^3)$ to perform one such VE, and $O((n-2)d^3) = O(nd^3)$ to perform all of them (we can ignore the first and last variables in terms of time complexity).

Since there are n^2 pairs, and each marginal takes $O(nd^3)$ to compute, the total runtime $= O(n^3 d^3)$

b) Prove the following equation holds:

$$P(X_i, X_j) = \sum_{X_{j-1}} P(X_i, X_{j-1}) P(X_j | X_{j-1}) \quad (\text{for } i < j-1)$$

First, using Bayes' Rule:

$$P(X_i, X_{j-1}) = \frac{P(X_i | X_{j-1})}{P(X_{j-1})}$$

$$P(X_i, X_j) = \sum_{X_{j-1}} \frac{P(X_i | X_{j-1}) \cdot P(X_j | X_{j-1})}{P(X_{j-1})}$$

$X_i \perp X_j | X_{j-1}$, Since knowing X_{j-1} blocks the flow between X_i, X_j by separating the subgraphs containing X_i, X_j.

Using this, we can rewrite the equation:

$$P(X_i, X_j) = \sum_{X_{j-1}} \frac{P(X_i, X_j | X_{j-1})}{P(X_{j-1})} = \sum_{X_{j-1}} P(X_i, X_j, X_{j-1})$$

$$= P(X_i, X_j) \qquad \text{QED}$$

c) Create algorithm which is asymptotically faster than $O(n^3d^3)$ using the recursive relation from part b).

Let us rewrite the relation to remove conditional independencies to simplify:

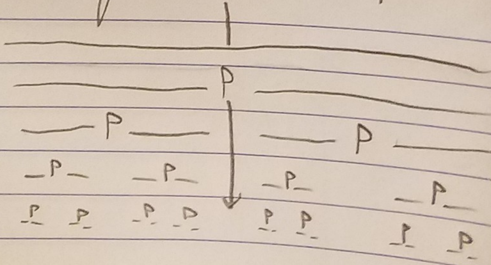
$$P(X_i, X_j) = \sum_{X_{j-1}} P(X_i, X_{j-1}) P(X_j \mid X_{j-1}) = \sum_{X_{j-1}} \frac{P(X_i, X_{j-1}) P(X_j, X_{j-1})}{P(X_{j-1})}$$

If finding $P(X_i, X_j)$ takes $O(nd^3)$, as shown in part a), then finding all of the above terms also takes $O(nd^3)$. Specifically, finding each of the 3 terms takes $O(nd^3)$ each, and $O(\cancel{3}nd^3) = O(nd^3)$.

Let us think of X_{j-1} as a pivot, X_p, such that all elements X_i, $i<p$ are separated from X_j, $j>p$, according to the above relation. We will first place this pivot in the center of the chain, so half of all variables are before and after. Next, we compute $P(X_i, X_p)$ for all i, $P(X_j, X_p)$ for all j. Using the relation, we can now compute $P(X_i, X_j)$ for all pairs $i<p$, $j>p$. This step takes $O(n)\cdot O(nd^3) = O(n^2d^3)$ and yields $\frac{n}{2}\cdot\frac{n}{2} = \frac{n^2}{4}$ of the n^2 pairs. (Takes $O(n)$ to compute pivot pairs)

Let us then move the pivot halfway between $1, p$, and repeat, then halfway between p, n and repeat. If we continue to split the remaining subsections in 2 using this strategy, it is clear that the splitting follows a log relationship, such that we must split $O(\log_2 n) = O(\log n)$ times to yield all n^2 pairs.

Thus, the final time complexity of this algorithm is $O(n^2 \log n\, d^3)$, which is asymptotically faster than $O(n^3d^3)$.



Problem 3: Clique Tree Calibration

a) If we modify a factor in some clique C_i, which message updates do we have to perform to recalibrate the tree?

If we modify (the values of) a factor in some clique C_i, we must update all messages in the tree leading away from C_i, (single pass), since these will be recursively affected by the change. This comes from the message update recurrence relation, which tells us that outgoing messages from some clique C_i are affected by the product of all incoming messages to C_i and the factor of C_i.

b) If we modify a factor in some clique C_1, but we just want the marginal over a single pre-specified variable X_k, which message updates do we have to perform?

To calculate the marginal over X_k, we must compute the product of a factor of X_k with all incoming messages to that factor, and then marginalize out all variables in that factor which are not X_k. This means we must recursively update all messages leading away from C_i up to any factor of X_k. In the best case, X_k is in C_i, and we do not need to perform any message updates. In the worst case, we must do a full single pass through the tree, as in part a (this will only happen if the cliques are on opposite ends of a tree with no forks i.e. a line).

Problem 4: Importance Sampling

a)

i. Can we compute P(x|e) exactly, in a tractable way?

No, computing this is intractable.

ii. Can we sample directly from P(X|e)?

No, we cannot sample directly from P(X|e).

iii. Can we compute P'(x|e) = P(x,e) exactly, in a tractable way?

Yes, we can compute P(x,e) exactly in a tractable way.

b) Q(X,E) is close to P(X,E)

i. Show how to sample from the posterior in Q.

Form a clique tree where all child/parent pairs are grouped as cliques. Now, incorporate the evidence E=e, and the distribution for the tree will be Q(X,E | E=e). After the clique tree has been calibrated, the belief at a clique over some X and its parents is proportional to Q(X, parents(X) | E=e). Using this, we can use Bayes' rule to compute Q(X | parents(X), E=e). Then, using the calculated CPDs, we can forward sample from the posterior in Q. This is done by sampling from the first variable, using it in neighboring cliques, and repeating through the clique tree.

ii. Reweight the samples according to the rules of importance sampling.

The weight function can be derived from this as:

w[m] =P'(x[m], e[m]) / Q(x[m], e[m] | e)

iii. Show the form of the final estimator Phat(X=x|E=e) for P(X=x|E=e), in terms of the samples from part i, and the weights from part ii.

The final estimator will take the form:

Phat(X=x | E=e) = [Sum over m of w[m]*1{x[m]=x}] / [Sum over m of w[m]]

Problem 5: Programming Assignment

a)

Code for constructClusterGraph:

```
def constructClusterGraph(yhat, H, p):
    '''
    :param - yhat: observed codeword
    :param - H parity check matrix
    :param - p channel noise probability

    return G clusterGraph

    You should consider two kinds of factors:
    - M unary factors
    - N each parity check factors
    '''
    N = H.shape[0]
    M = H.shape[1]
    G = ClusterGraph(M)
    domain = [0, 1]
    G.nbr = [[] for _ in range(M+N)]
    G.sepset = [[None for _ in range(M+N)] for _ in range(M+N)]
    ##############################################################

    # Set the variables of G
    G.var = range(M)

    # Set unary factors
    for i in range(M):
        val = None
        if yhat[i] == 1:
            val = np.array([p, 1-p])
        else:
            val = np.array([1-p, p])

        f = Factor(scope=[G.var[i]], card=[len(domain)],
                val=val,
                name="unary_{}".format(i))
        G.factor.append(f)

    # Set parity check factors
    for i in range(N):
        scope = (np.arange(M)[H[i]==1])
        card = [len(domain) for _ in range(len(scope))]
        poss = list(itertools.product(domain, repeat=len(scope)))
        val = np.zeros(card)
        for p in poss:
            val[p] = 1-np.count_nonzero(p)%2
        f = Factor(scope=scope, card=card,
            val=val, name="parity_{}".format(i))
        G.factor.append(f)

    # Initialize sepset of factors
    for i in range(M+N):
        si = set(G.factor[i].scope)
        for j in range(M+N):
            sj = set(G.factor[j].scope)
            # Only parity factors & unary factors are neighbors with each other
            if (len(si) == 1 and len(sj) > 1) or (len(si) > 1 and len(sj) == 1):
                shared = list(si.intersection(sj))
                # If list of shared vars is empty, don't set
                if len(shared) > 0:
                    G.sepset[i][j] = shared


    # Initalize neighbors of factors using sepset
    for i in range(M+N):
        for j in range(M+N):
            if G.sepset[i][j] is not None:
                G.nbr[i].append(j)

    # Iterate over the sepset to form messages
    for i in range(M+N):
        for j in range(M+N):
            sepset = G.sepset[i][j]
            if sepset is not None:
                card = [len(domain) for _ in range(len(sepset))]
                message = Factor(scope=sepset, card=card,
                    val=np.ones(card), name="msg_{}_{}".format(i, j))
                G.messages[(i, j)] = message.normalize()

    ###############################################################
    return G
```

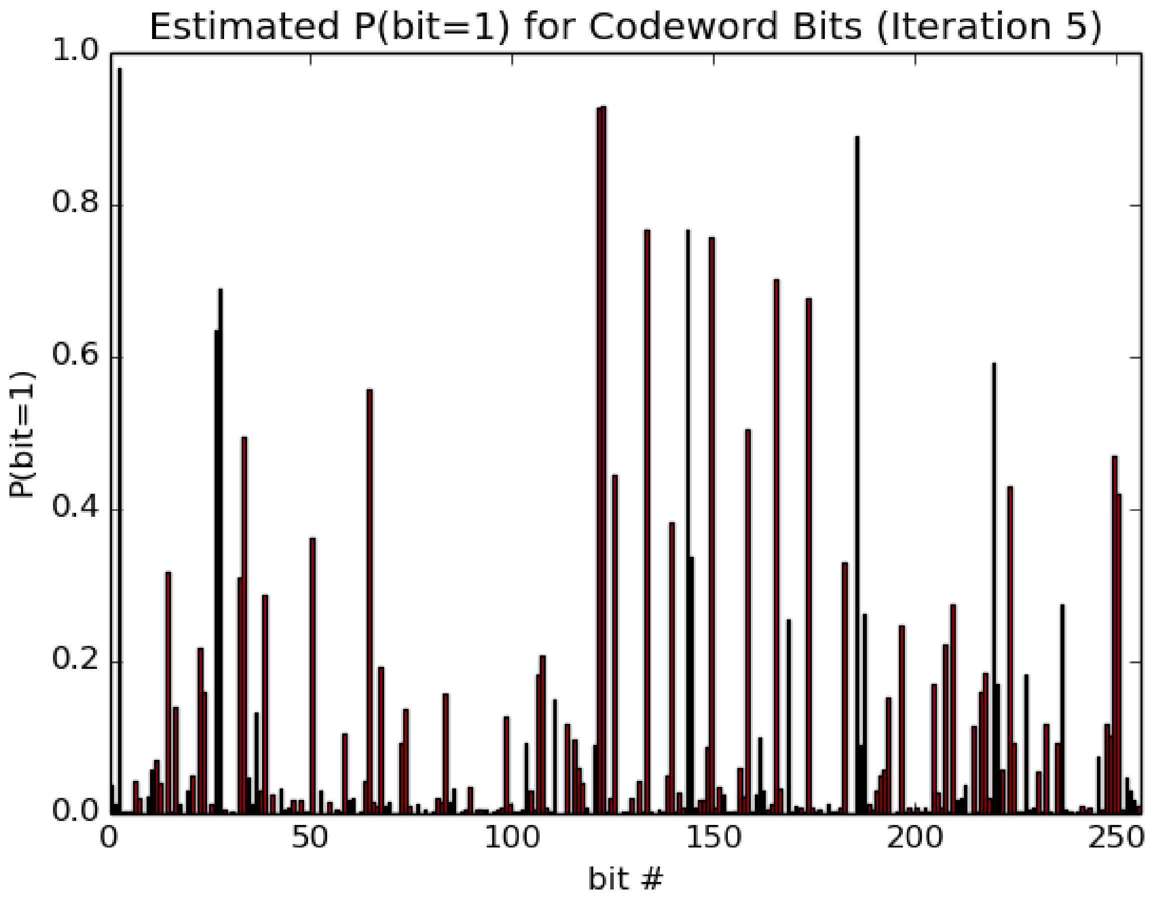
Test cases:
The first two test cases do not have the same parity for all factors, while the last case (0-vector) does.

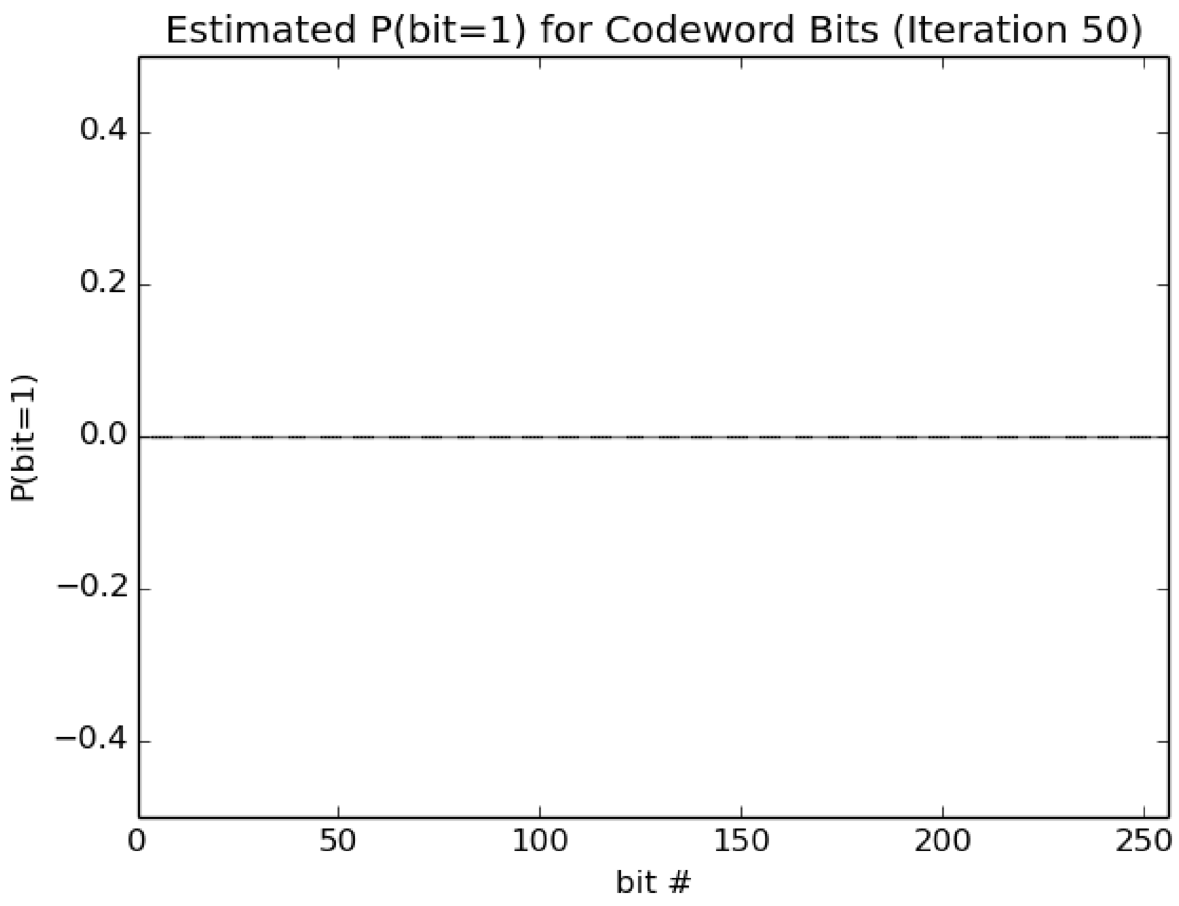
```
    ytest1 = [1, 0, 0, 0, 0]
    ytest2 = [0, 1, 0, 0, 0]
    ytest3 = [0, 0, 0, 0, 0]
```

Output:
```
Doing part (a): Should see 0.0, 0.0, >0.0
(0.0, 0.0, 0.77378093749999988)
```

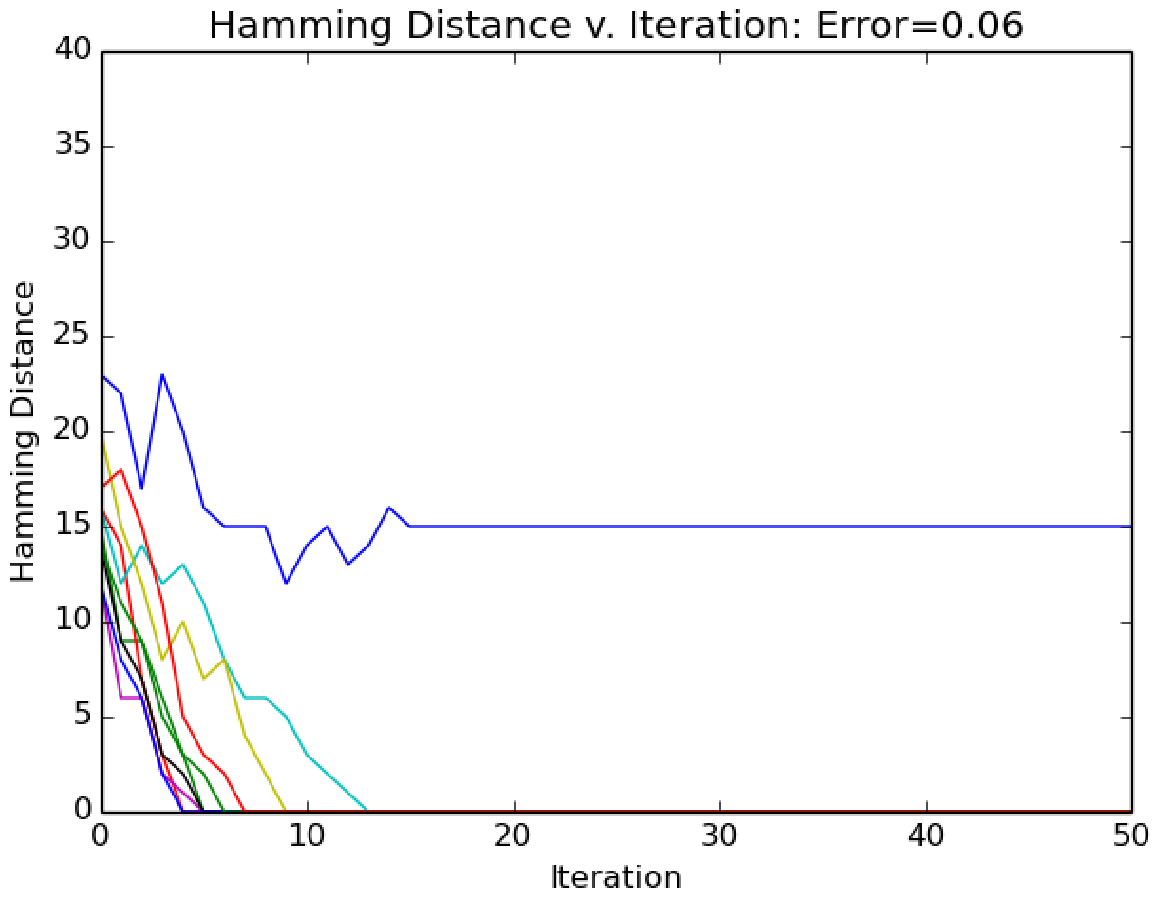
b) The ClusterGraph skeleton was used to construct a clique tree implementation of loopy belief propagation, as described in the lecture slides. In each iteration of the algorithm, all messages were passed between factors in both directions. As the messages were computed, they were stored back to the same dictionary, which seemed to increase the speed of convergence considerably. Marginals for each variable were computed by normalizing the product of the unary variable factor and all incoming messages. The argmax of these marginals was taken to get the predicted value of each resulting codeword bit.

c) The plot for the final iteration was very boring, since the resulting posterior probability vector was the 0-vector. For this reason, I also include a plot from iteration 5. If we decode the iteration 50 result by setting each bit to the maximum of its corresponding marginal, we would find the correct codeword.

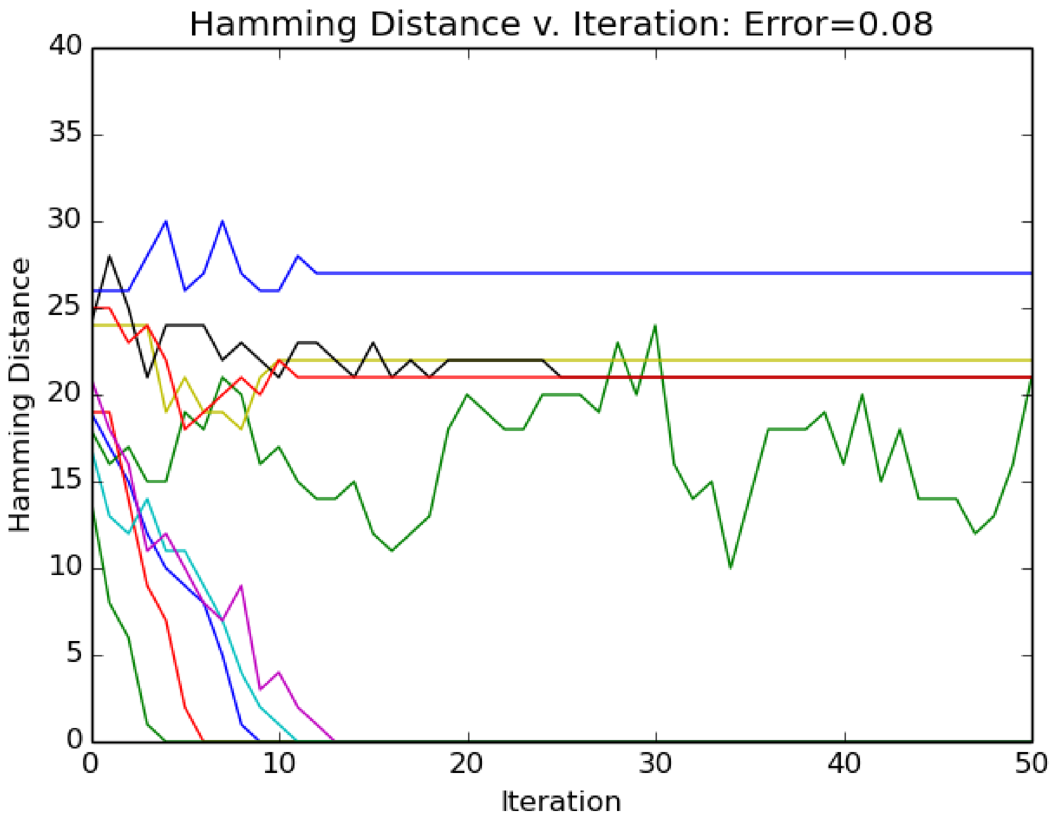


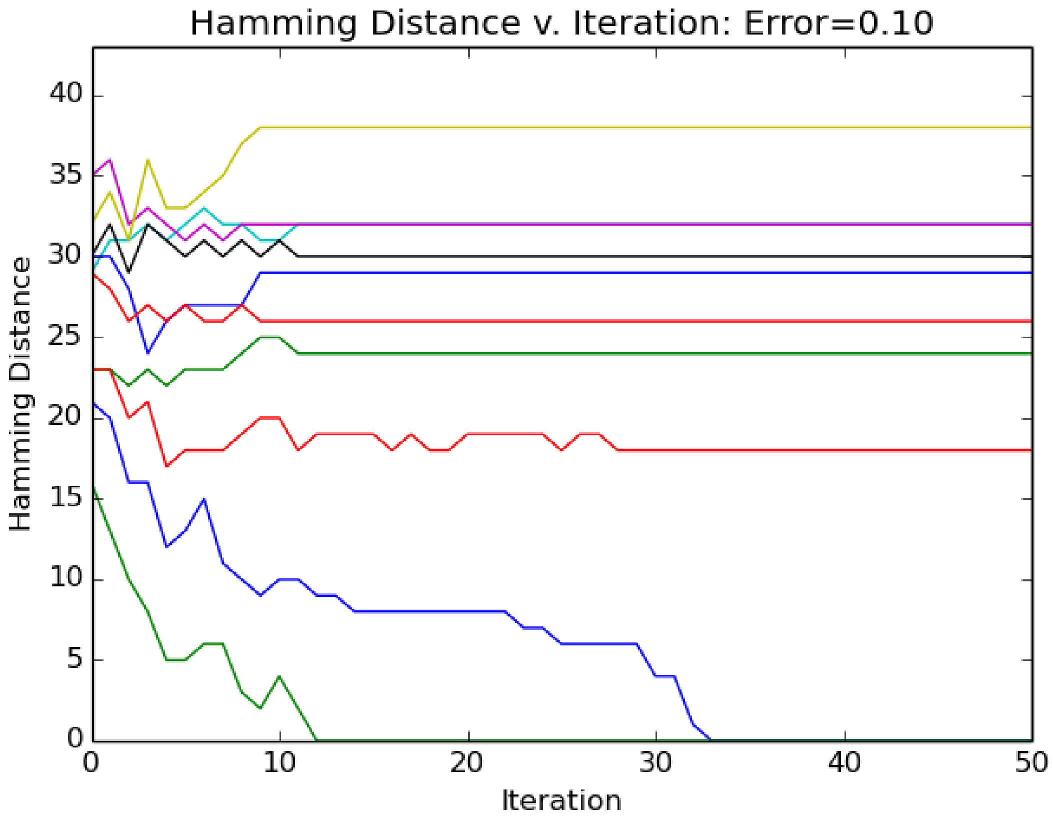


d) Since one of the trials converged at an incorrect solution for P(error) = 0.6, it is arguable whether the algorithm is reliable. If there can be an assurance made that for some error probability, the probability of converging to an incorrect codeword (or not converging at all) is vanishingly small, then the algorithm may be considered reliable for that scenario. Perhaps this is true for a lower probability of error than 0.6 for this case.



e) For higher error rates, it is clear that the probability of converging to an incorrect codeword increases (or not converging at all). We can see that while only one trial converges at an incorrect result for p(error)=0.6, about half are incorrect for p(error)=0.8, and all but two for p(error)=0.10. In some cases for p(error)=0.10, the converged codeword had more incorrect bits than the original noised codeword. In this way, it seems that p(error) should be 0.5 or less for the decoding to be reliable for this particular setup scheme.


Hamming Distance v. Iteration: Error=0.08
Hamming Distance
Iteration
0
5
10
15
20
25
30
35
40
0
10
20
30
40
50

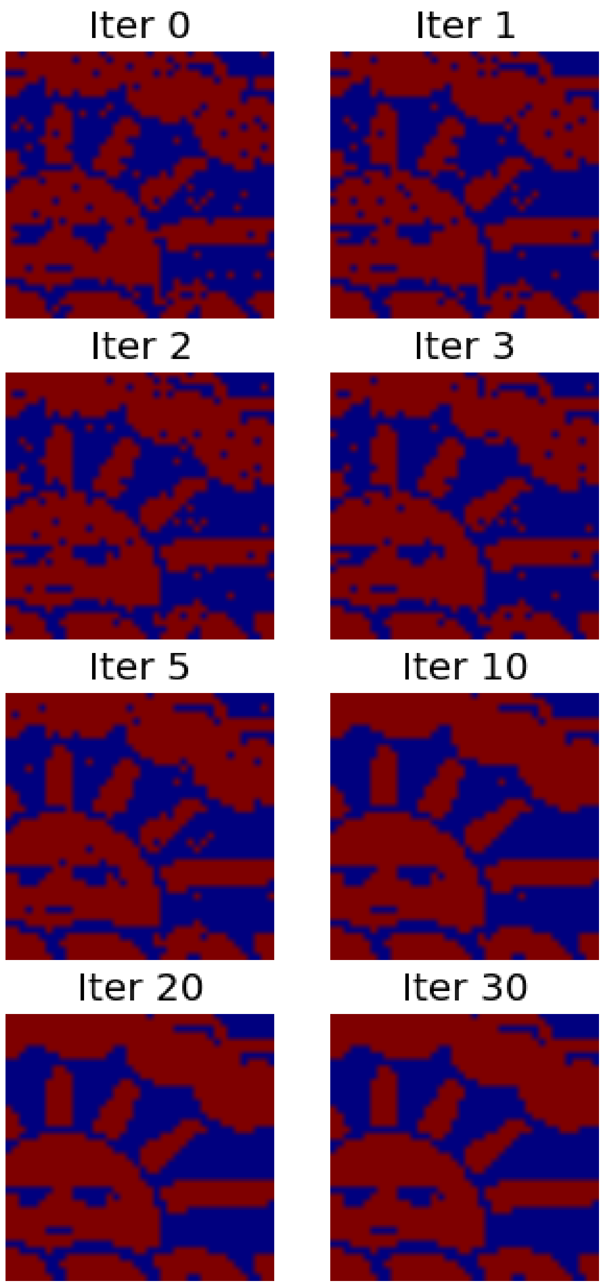

Hamming Distance v. Iteration: Error=0.10
Hamming Distance
Iteration
0
5
10
15
20
25
30
35
40
0
10
20
30
40
50

f) The hamming distance from the true codeword for the target iterations are included in a table. As shown in the plots, the original image is fully recovered for p(error)=0.6.

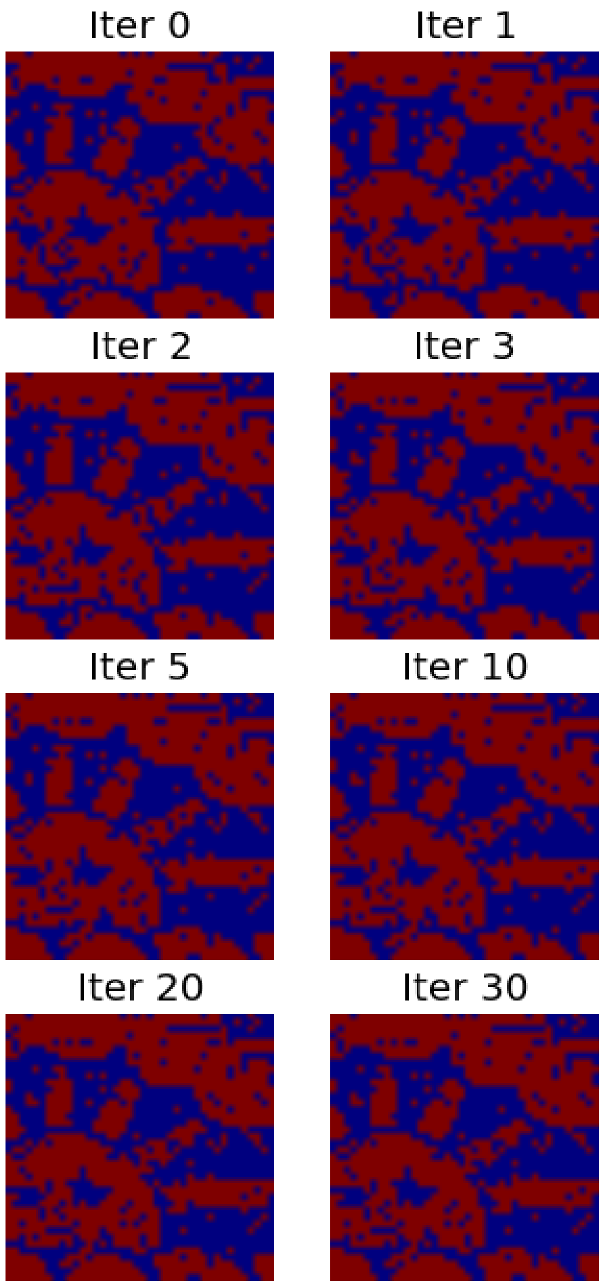
Hamming Distance for Different P(error)		
	Hamming Distance (from truth)	
Iteration	P(error) = 0.06	P(error) = 0.10
0	199	329
1	161	331
2	137	312
3	112	311
5	73	302
10	12	310
20	0	308
30	0	308

g) As shown by the included plots, the decoder converges to the correct result for P(error)=0.06, but not for P(error)=0.10. This aligns with the result for parts d/e. It seems that if a sufficient number of parity bits are corrupted, the decoder will always converge to an incorrect solution (or not converge at all). Still, for this example, the decoder was able to recover 21 bits for the higher error rate, meaning it has some utility. Multiple trials with different noise channels should be tested to see if it always converges, and if it does, if it converges to a result which recovers some bits.

Decoded Output: Error=0.06



Decoded Output: Error=0.10


Iter 0
Iter 1
Iter 2
Iter 3
Iter 5
Iter 10
Iter 20
Iter 30