Overview

The TaskOneModel and TaskTwoModel classes fit models for their respective problems and test their accuracy and loss. An instance of these classes generates one fitted model.

TaskOneModel and TaskTwoModel train and test from using data stored in text files. These text files represent a wire diagram and their labels. The wiring configurations are created in the WireDiagram class. The GenerateWiringDiagrams class uses WireDiagram to store wiring configurations in a text file.

ModelTesting creates graphs of the models' performance. The main method in the class can be run to produce graphs for each model. **Note** if this is being run, the paths at the top of the GenerateWiringData class need to be changed to the new path of the text files.

The matrixManger class contains static methods for matrix and vector manipulation. The Wire class is used in GenerateWireDiagram.

The Data Set

1. Wire Representation
    a. The Wire class holds integer constants as the representation of each wire.
        i. ints 0-4 represent each wire or the absence of one
            1. NO_WIRE = 0
            2. RED_WIRE = 1
            3. BLUE_WIRE = 2
            4. YELLOW_WIRE = 3
            5. GREEN_WIRE = 4
        ii. Segments of the code that iterate through the wire colors or properties that correlate to a wire color will be ordered in the same way.
            1. In task two, there is a weight vector for each wire color, which is stored in a list. The list is ordered such that the first index is the red weight vector, the second is the blue weight vector, and so on.
    b. Encoding
        i. The Wire class has an encoder that takes in an integer that represents the wire and returnns a one hot encoded array representation.
        ii. For example:
            1. encodedWirecolor(NO_WIRE) returns [0, 0, 0, 0]
            2. encodedWirecolor(RED_WIRE) returns [1, 0, 0, 0]
    c. In diagram generation, an instance of the Wire class represents one single cell in the 20x20 diagram. It holds the color of the cell.
2. Wiring Configurations

a. An instance of the WiringDiagram class holds a 20x20 matrix of Wire instances. It follows the specified generation rules by modifying the color of each cell to form a valid diagram.
    i. In the following pseudo code I will to the matrix as wireDiagram
b. WiringDiagram uses three Integer ArrayList properties to keep track of its generation
    i. colorsChosen stores the order of which the colored wires are placed on the diagram .
        1. So when a wire is placed, its integer representation is added to the list.
    ii. rowsChosen stores the index of where a horizontally placed wire was laid.
    iii. colsChosen stores the index of where a vertically placed wire was laid.
c. WiringDiagram also has properties of whether it is dangerous and which wire to cut.
d. The generateDiagram takes an empty wireDiagram (initialized with Wire instances but all are set to NO_WIRE) and modifies rows and columns of Wires to create a valid diagram.
    i. It uses the fillingRow boolean to determine whether it is adding a wire to a column or row.
    ii. Generation Process
        1. initialize the empty wireDiagram
        2. With a 50/50 chance assign fillingRow as true or false
            a. First wire has a 50% chance to start on either a row or column
        3. Iterate over 4 times: //Loop once for each color
            a. Pick a random number 0-19
                i. Check if that index already has a wire by looking at the respective rowsChosen and colsChosen lists.
                    1. The one looked at depends on the fillingRow boolean.
                ii. Pick a another number if the index already has a wire
            b. Pick a random wire color that is not already in the colorsChosen array
            c. Based on fillingRow, fill the index of the column or row selected with the chosen color.
                i. Loop through the columns or rows and sets the color of each Wire to the chosen color
            d. Add the row/column filled to the rowsChosen/colsChosen
            e. Add the wire color chosen to the end of the colorsChosen array
            f. Set fillingRow = !fillingRow
                i. switch from row to column or column to row
        4. Lastly, set the labels for the diagram

a. The isDangerous property equals 1 if the red wire comes before the yellow wire in the colorsChosen list.
b. The wireToCut property equals 0 if isDangerous is 0, otherwise it is the element in the colorsChosen list.

iii. Example:

1. 

e. The encodeDiagram returns a string representation of the wireDiagram.
   i. It loops through wireDiagram and appends the wireColor to the end of a string.
      1. For example, the the first 5 elements in the first row of the above example would read:
         a. 13111

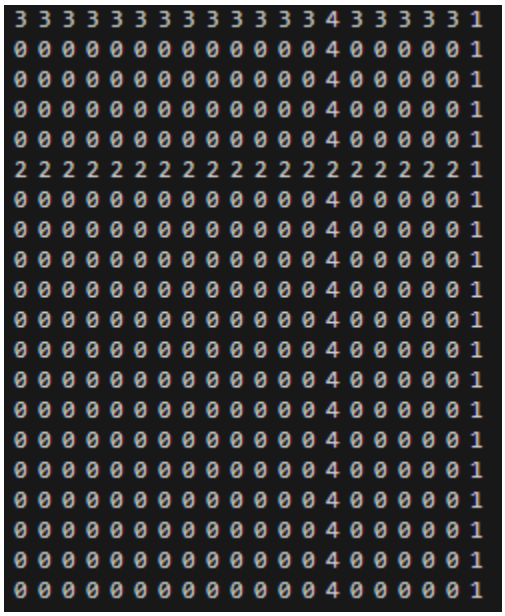
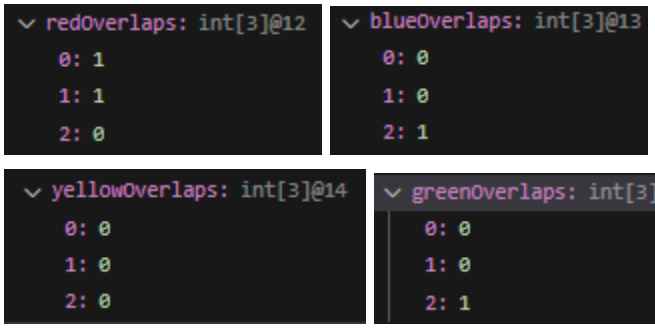
3. Data Generation
   a. The GenerateWireDiagram class creates multiple instances of the WireDiagram class and writes its information to text files.
      i. Each task has two text files, one training file and one testing file.
      ii. Thus, the models are trained on the same training inputs each time and tested on the same testing inputs each time to add some standardization between runs of the model.
   b. The text files are located in the data folder which is in the Project3 directory.
      i. The SafeOrDangerousTraining holds 5000 wire diagrams to train for task 1 and the SafeOrDangerousTesting holds 1000 wire diagrams to test the accuracy of the model.
      ii. WireCutTrainingData holds 5000 wire diagrams to train the task 2 model and WireCutTestingData holds 1000 diagrams to test its accuracy.
   c. Generation Methods
      i. The generateSafeDangerousWiringData method in GenerateWiringDiagrams writes a number of WireDiagrams for task 1 to a specified text file.

            1. The model ensures half of the diagrams are safe and half of the diagrams are dangerous so the model is evenly trained.
        ii. The generateWireToCutWiringData method writes a number of WireDiagrams for task 2 to a text file
            1. It makes sure that there is an equal amount of diagrams for each label.
                a. ¼ for each wire color
        iii. The format for each text file repeats:
            1. Line 1: encoded string form of a diagram
            2. Line 2: The safe or dangerous label (0 / 1)
            3. Line 3: The wire to be cut label (0 / 1 / 2 / 3 / 4)
            4. Line 4: Blank line
    d. The generation methods were called for each text file to populate them with their data.
4. File Reading
    a. The text files are read in their respective TaskOneModel or TaskTwoModel class but each method follows the same guidelines.
    b. Each file reader loops a set number of times depending on how many diagrams it will read.
        i. For the training sets in Tasks 1 and 2, they run a number of iterations based on the numSamples property of the model instance.
            1. If the model should be trained on a set of 2000 examples, then it would iterate 2000 times.
        ii. For the testing sets in Tasks 1 and 2, they run for a constant number of iterations equal to the size of the testing document.
            1. Runs 1000 times since the testing documents have 1000 entries written to them.
    c. Task 1 and Task 2
        i. In both tasks, the first line is read as the diagram representation which is then passed into a function that generates the feature space from it.
        ii. In Task 1, the 2nd line is read as the safe/dangerous label for the input and the 3rd and 4th line are skipped.
        iii. In task 2, the 2nd line is skipped and the 3rd line is read as which wire should be cut. The empty 4th line is skipped.

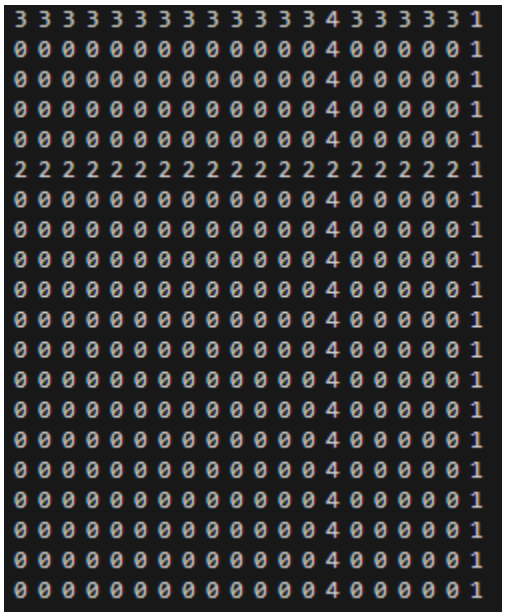
Task One

1. Overview
    a. The TaskOneModel class fits a model to a number of training samples from the text files.
    b. The constructor for TaskOneModel takes in a number of training samples, a learning rate, and a penalizing timer for l2 regression.
    c. TaskOneModel properties:

        i. Holds an array of weights which is the same length as the feature space
        ii. diagramMap
            1. A hashmap that maps a feature space to its corresponding label.
            2. Holds the data of the training samples
        iii. testingInputMap
            1. A hashmap that maps a feature space to its corresponding label.
            2. Holds the testing data for the model
            3. Set to a constant 1000 samples
2. Input
    a. The input space is a vector made up of the flattened wire matrix. The wire colors are encoded so that there is no inherent order among the colors. Thus, the input will be a 1600 length vector made up of the one hot encoded representation of each wire in the diagram.
    b. Implementation
        i. The readTrainingDiagrams and readTestingDiagrarms methods read the string representations of the diagrams stored in the text files.
        ii. The getFeatureSpace method takes in the string representation of a diagram and outputs the feature.
            1. For the input representation, it loops through the string representation of the diagram and appends the one hot encoded version of each wire to the feature space array.
            2. After this loop, the array is populated with the 1600 length vector of the inputs.
3. Output
    a. The model assigns a 0 or 1 to an input for safe or dangerous respectively.
    b. Implementation
        i. The safe/dangerous label is read in the read diagrams methods as the line under the string diagram representation.
        ii. This is the value in the key-value pair in the diagramMap and testingInputMap hashmaps.
4. Model
    a. A logistic regression model since the output is a binary classification.
        i. Thus, when assigning a label, the model will take the sigmoid of the weights and inputs to get a probability of the dangerous label being assigned.
    b. Feature Space
        i. The model uses the input vector (x_1, x_2, … x_{1600}) as its parameters.
        ii. The feature space will start with a 1 as a baseline feature. This will be followed by the input vector that represents the diagram.
            1. These features are linear since each represents if there is a specific color of wire at a pixel in the image.
        iii. There are 12 additional features appended to the array that represent whether a wire overlaps another.
            1. This is organized as 4 arrays, representing a wire color.

2. Each array has 3 elements that represent whether the wire overlaps another.
    a. Each has 3 elements since the wire can't overlap itself. Thus, the order of the elements is kept in red, blue, yellow , green but excludes the wire the array represents.
    b. For example:
        i. In the redOverlaps array, {0,0,0}
            1. The first element represents if the red wire overlaps the blue wire
            2. The second element represents if the red wire overlaps the blue, and the 3rd represents if red overlaps green.
            3. In this case none are being overlapped by red
    c. 
    d. 
    e. The above images show that the red wire overlaps the blue and yellow wires, the blue wire overlaps the green wire, the yellow wire does not overlap any wire, and the green wire overlaps the yellow wire.

iv. Thus the feature space is a vector of 1s and 0s with 1613 length
    1. 1 + 1600 + 12

c. Implementation
    i. The diagramMap and testingInputMap hashmaps hold the feature space (as defined above) as a key and its label as its value.
    ii. The readDiagram methods put the pair of getFeatureSpace(string diagram representation), label in the map.
    iii. getFeatureSpace()
        1. This method takes in the string diagram and outputs an Integer[] array of the feature space.
        2. The method creates a 1613 length array to store the feature space.
        3. This first element is always set to 1 to serve as a baseline feature.
        4. Next the one hot encoded vector of the diagram is appended after.
            a. Implementation is explained in the Input point above.
        5. neighborColorChecker()
            a. Helper method that checks if the given wire's pixel overlaps another wire.
            b. This uses the observation that a wire overlaps another when at any pixel:
                i. The overlapping wire has 2 neighbors of the same color
                ii. Or if its neighbor is a perpendicular wire.
            c. 
                i. In this example, the blue wire overlaps the green, since at the intersection point the blue cell has 2 green neighbors. The red wire overlaps the blue since at the intersection point, it has 1 blue neighbor and the blue wire is perpendicular to it.
            d. The neighborColorChecker() method thus checks for a given cell if the given wire color overlaps another wire

based on its neighbors in the diagram, and if so returns the color of the wire it overlaps.

6. getFeatureSpace() uses this helper method to fill out the overlap arrays.
    a. First, it creates a literal diagram representation of the map so it can easily access information on neighboring cells.
    b. Then it loops through the diagram:
        i. If it encounters a wire, it will run neighborColorChecker() with the respective color
        ii. If it returns a color, that means the wire overlapped the returned color wire and thus the respective element in the wireOverlap array is set to 1.
7. The overlap features are non linear since they take the linear features that represent the map and apply classifiers to them that incorporate the wire's neighbors. This can't be classified as a linear function of an input feature.

iv. By the end of the reading process, the diagramMap and testingInputMap are populated with the input's label mapped to the feature space.

5. Overfitting
    a. To prevent overfitting, the model utilizes l2 regression. This adds a penalizing term to the loss function which adds sensitivity to large weights.
6. Loss Function
    a. The loss function is log loss as follows:
        i. L(w) = -1/N * ∑(sum from i = 1 to N)[yi * ln(f(xi)) + (1 - yi) * ln(1 - f(xi))]
            1. N is the size of the set of inputs
            2. xi represents the ith input being iterated over
            3. yi represents label corresponding to xi
            4. f(xi) is the sigmoid function of the dot product between xi and the weights vector.
        ii. Sigmoid function:

$$S(x) = \frac{1}{1 + e^{-x}}$$

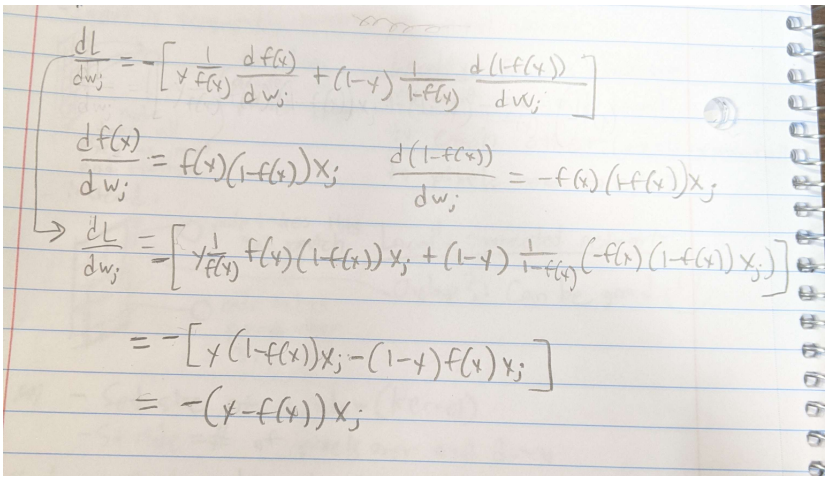
            1.
            2. x represents: dot product of (input vector xi) and (weight vector w)
    b. With regularization:
        i. L(w) = -1/N * ∑(sum from i = 1 to N)[yi * ln(f(xi)) + (1 - yi) * ln(1 - f(xi))] + λ * ∑(sum from j = 1 to N) wj ^2
            1. λ represents the penalizing factor which is given to the model at construction
            2. wj is the single weight at position j in the weight vector.
    c. The loss function is calculated in the calculateLoss() method.
        i. The function takes in a HashMap of Integer arrays to Integers

1. To calculate the loss over the testing set, set to testingInputMap
2. For loss over training set, input diagramMap

ii. Implementation

1. Calculate the total loss over -1/N * ∑(sum from i = 1 to N)[yi * ln(f(xi)) + (1 - yi) * ln(1 - f(xi))]
    a. Loop through the hashmap map and plug into the above equation:
        i. yi = entry.getValue
            1. The label of the input
        ii. f(x) = calculateSigmoid(entry.getKey , weights)
            1. weights is a property so it will always have access to it
            2. entry.getKey is the input feature space which is xi
    b. Add the result of these calculations to a variable totalLoss.
2. Calculate the regression factor:
    a. double l2Regression = 0
    b. Loop through each weight in the weights vector and add the square of the weight to l2Regression
    c. l2Regression *= λ
3. Return
    a. ((totalLoss)/inputMap.size()) + l2RegressionFactor

7. Gradient of the Loss
    a. The gradient of the loss will be an array where the jth index represents ∂Loss/∂wj where wj is the jth weight in the weights matrix.
    b. ∂Loss/∂wj

i. 

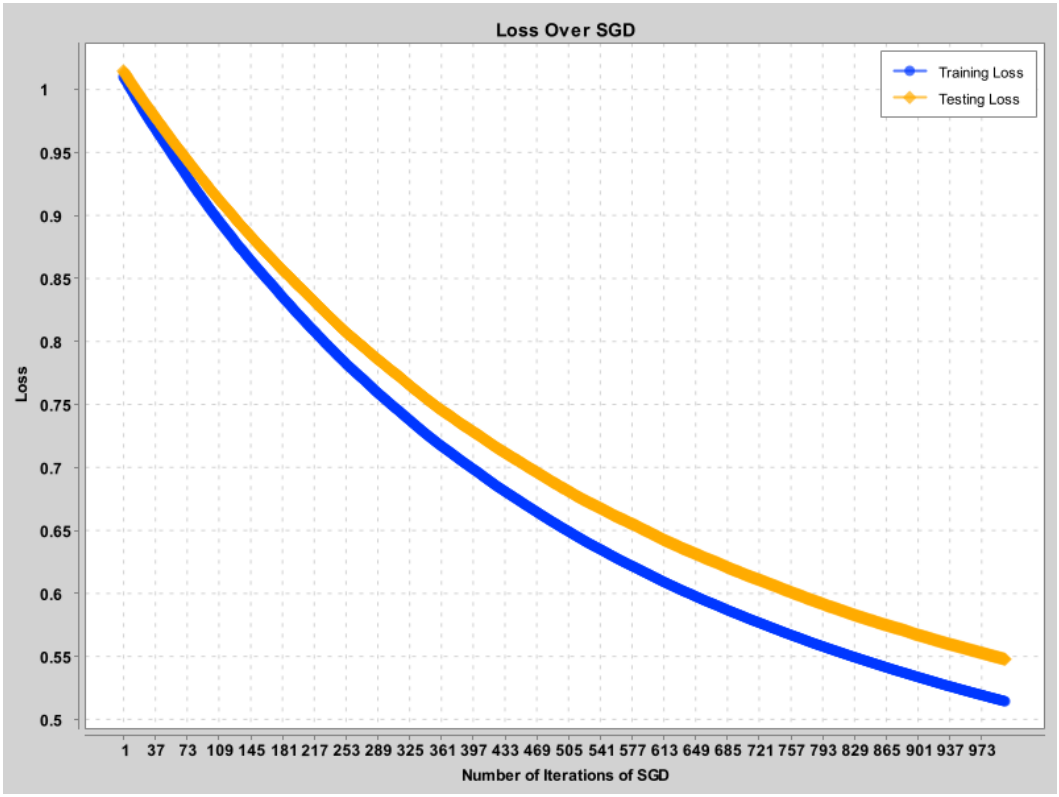
ii. Note f(x) = sigmoid of xi dotted with weights vector

    c. Add on the derivative of the l2 regression
        i. + 2 * λ * wj
    d. Implementation
        i. The computeLossGradient method returns the loss gradient array with the input vector and its label as parameters.

ii. calculate the y - f(x) term once before looping over the input.
    1. sigmoidMinusLabel = calculateSigmoid(input, weights) - label;
iii. Then loop through the input vector and assign:
    1. lossGradient[i] = input[i] * sigmoidMinusLabel + 2 * λ * weights[i]
iv. return the loss gradient

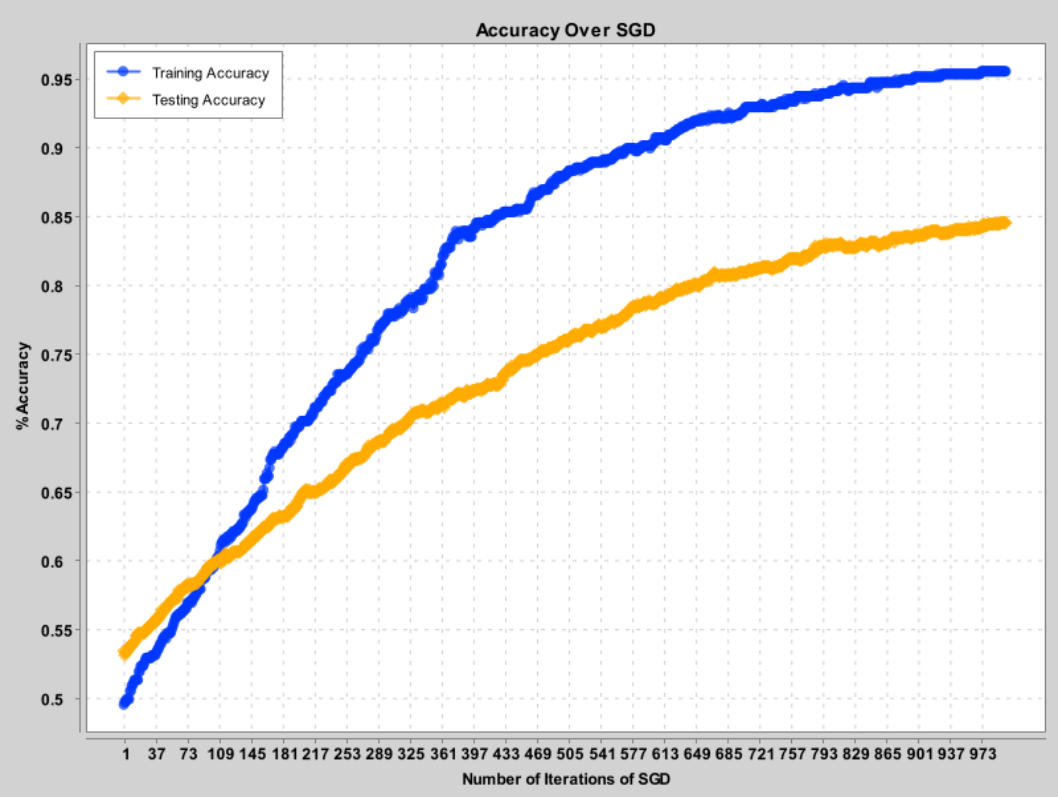
8. Training Algorithm
    a. The model uses Stochastic Gradient Descent to train the weights.
        i. The training is done in the SGD() method
    b. The method starts by reading the training and testing data which populates the diagramMap and testingInputMaps.
    c. Populate the weights array by selecting random values within a range. If the range is 0.2, the weights will be randomly assigned values between -0.1 and 0.1.
        i. Small weight ranges are used since the model has trouble converging at higher ones.
    d. For a number of iterations:
        i. Populate the randomEntries list with a number of entries from diagramMap.
            1. This is the set that SGD uses when updating the weights rather than the entire training set in GD.
            2. The size of the set is determined by the constant SGD_SAMPLE_SIZE.
        ii. Create an ArrayList of Double arrays called gradientList and populate it as follows:
            1. For each entry in randomEntries:
                a. gradientList.add(computeLossGradient(entry.getKey(), entry.getValue()));
            2. This will add the loss gradient of each input in the SGD set to gradientList
        iii. Create a Double array averageLossGradient and set each index to the average of the index in all gradients over gradientList.
        iv. Then for each weight in the weights vector:
            1. weights[i] = weights[i] - alpha * averageLossGradient[i]
                a. Updates the set of weights
            2. alpha is the learning rate set with the constructor
9. Predicting Accuracy
    a. The predictLabel method takes in an input and outputs its predicted label.
        i. It calculates the sigmoid of the input and the weights and and assigns 1 if the probability is over 0.50 and 0 otherwise.
        ii. The cutoff is 0.5 since the training samples are split evenly as well.
    b. The testSuccessRate method takes in a HashMap of Integer arrays to Integers
        i. It calls predict Label on every entry.key in the map and compares its output to the actual value (entry.value).
        ii. It returns the average success rate

c. The testSuccessRate method is called at each iteration in SGD and the result is stored in lists that are graphed to show the accuracy increase over runtime.
    i. The same is done for the loss function.

10. Selections in testing
    a. In testing, the alpha value was set to 0.01 as it showed clear improvement in accuracy and loss but didn't take too many iterations.
    b. Lambda was set to 0.01 as it prevented overfitting while keeping the accuracy high.
11. Loss Graph:
    a. 

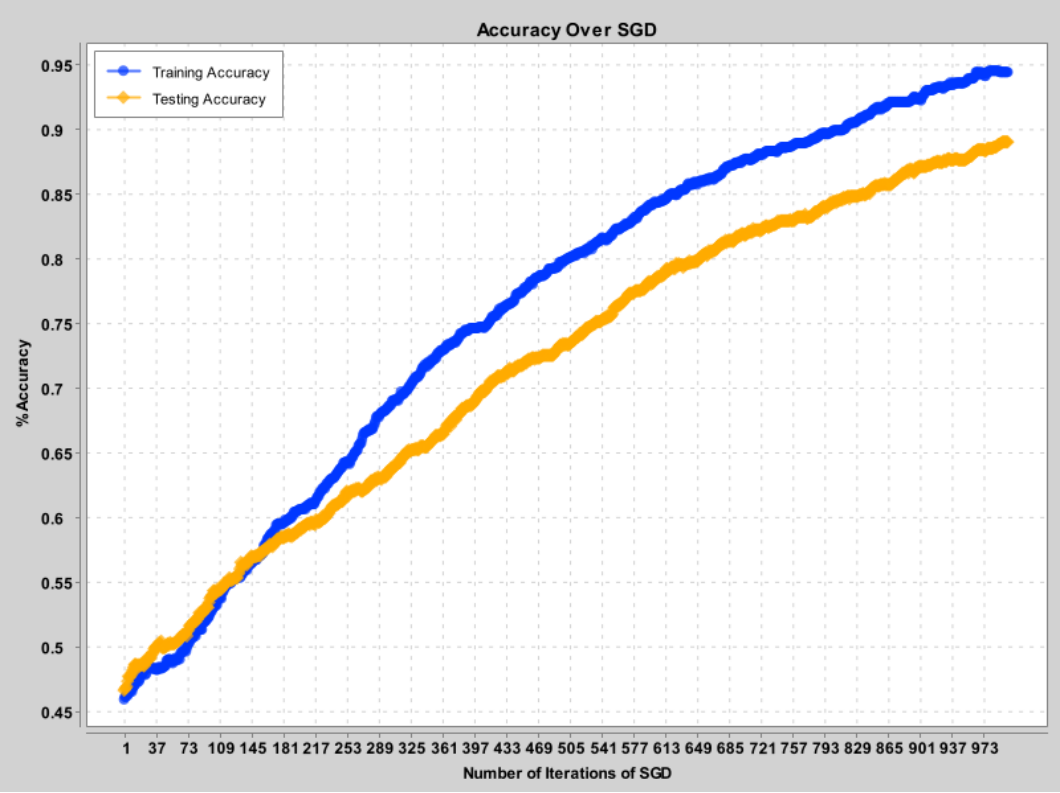
    b. This graph shows the loss over the training (Size of training set = 2000) and testing sets over iterations of SGD.
    c. Both consistently decrease, demonstrating learning.
    d. The testing loss does not increase after a number of iterations, demonstrating it is not overfit to the data.
12. Performance: Accuracy graph over 1000 iterations of SGD

a.



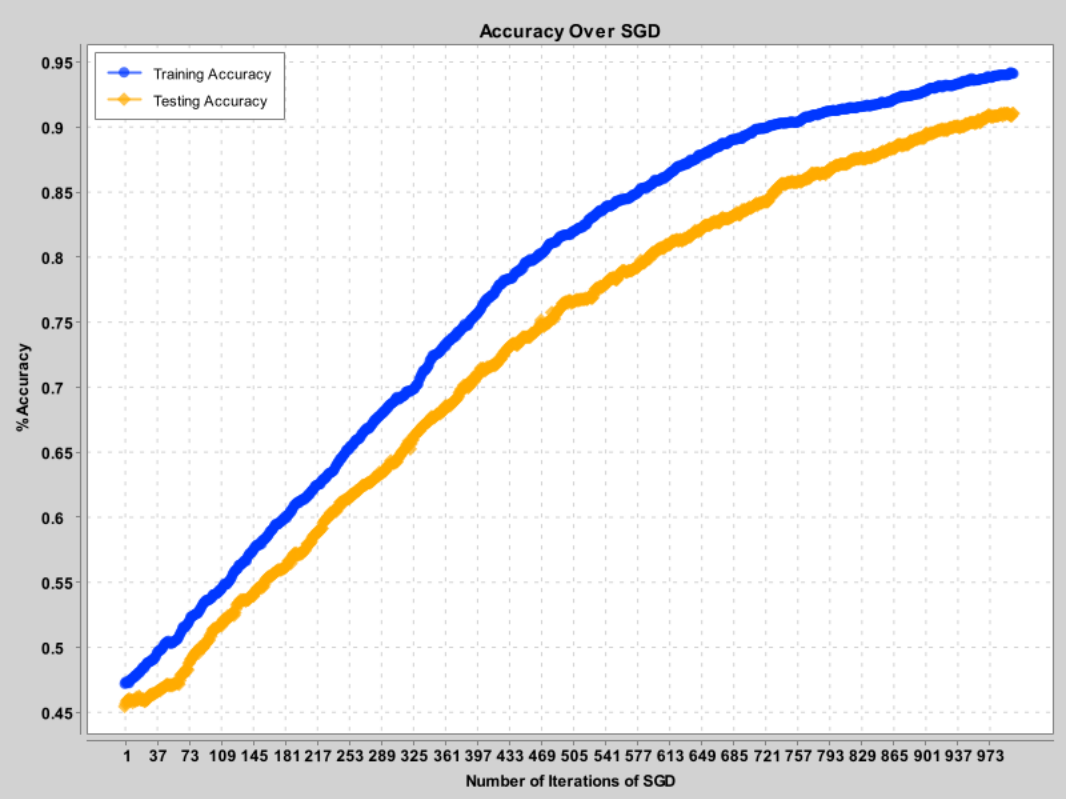
i. Accuracy with 500 samples in training set

b.



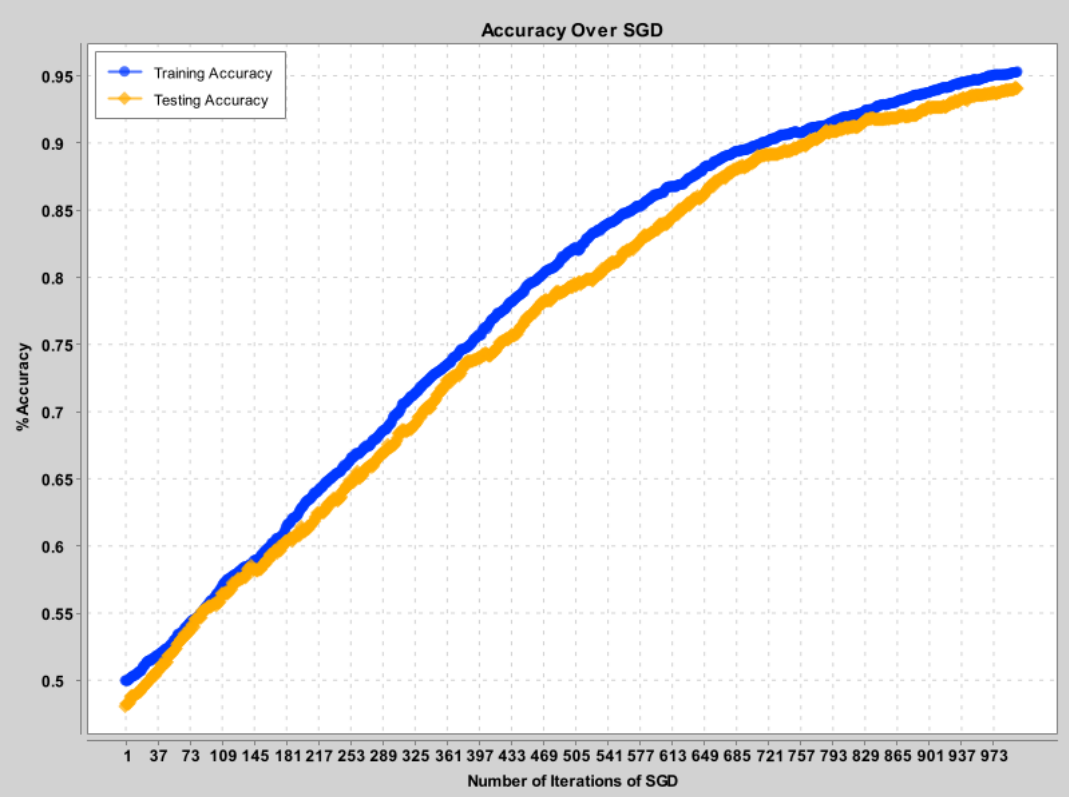
i. Accuracy with 1000 samples in training set

c.

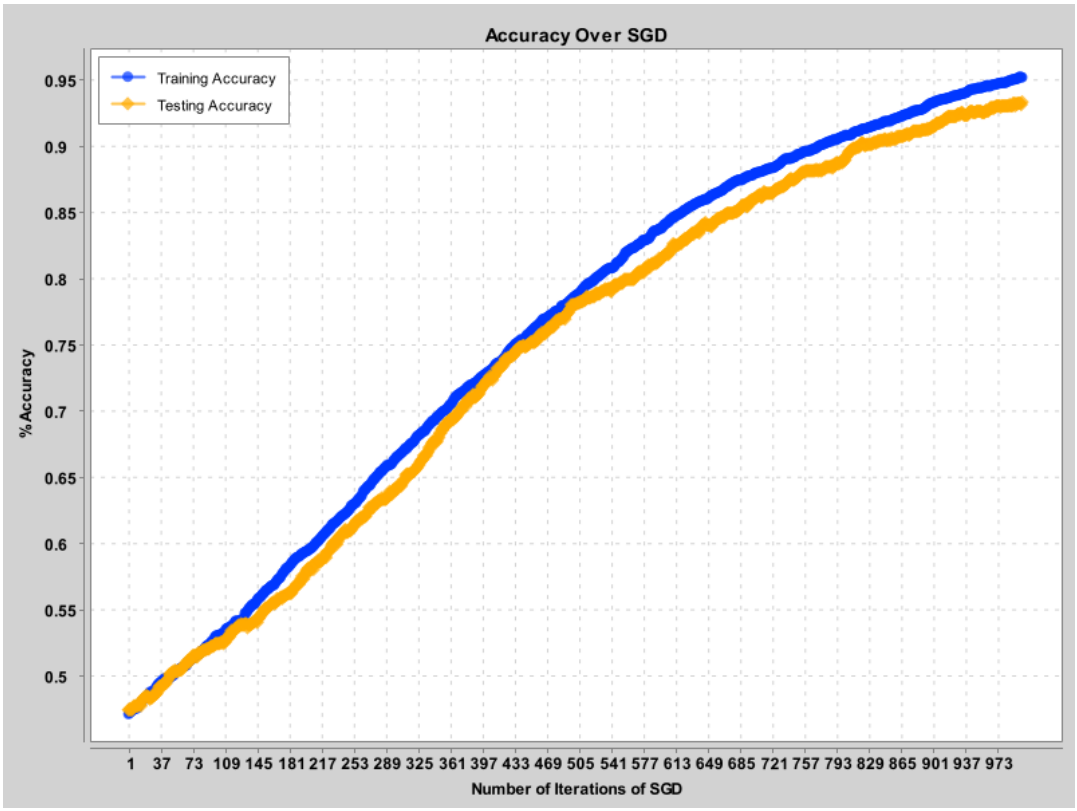


i. Accuracy with 2000 samples in training set

d.



i. Accuracy with 3000 samples in training set

e. 

   i. Accuracy with 5000 samples in training set

f. The accuracy graphs show a clear benefit from sample sizes above 1000
   i. Highest accuracy reached with 500 samples was 85% and highest with 1000 samples was 90%
g. At 1000 iterations the testing accuracy in the 3000 sample graph gets closer to the training than in the 2000 sample graph.
h. Though after 3000 samples there doesn't seem to be much more improvement in the graph.
i. These graphs demonstrate that the model generalizes well with high testing accuracy that is close to training accuracy.

Task Two

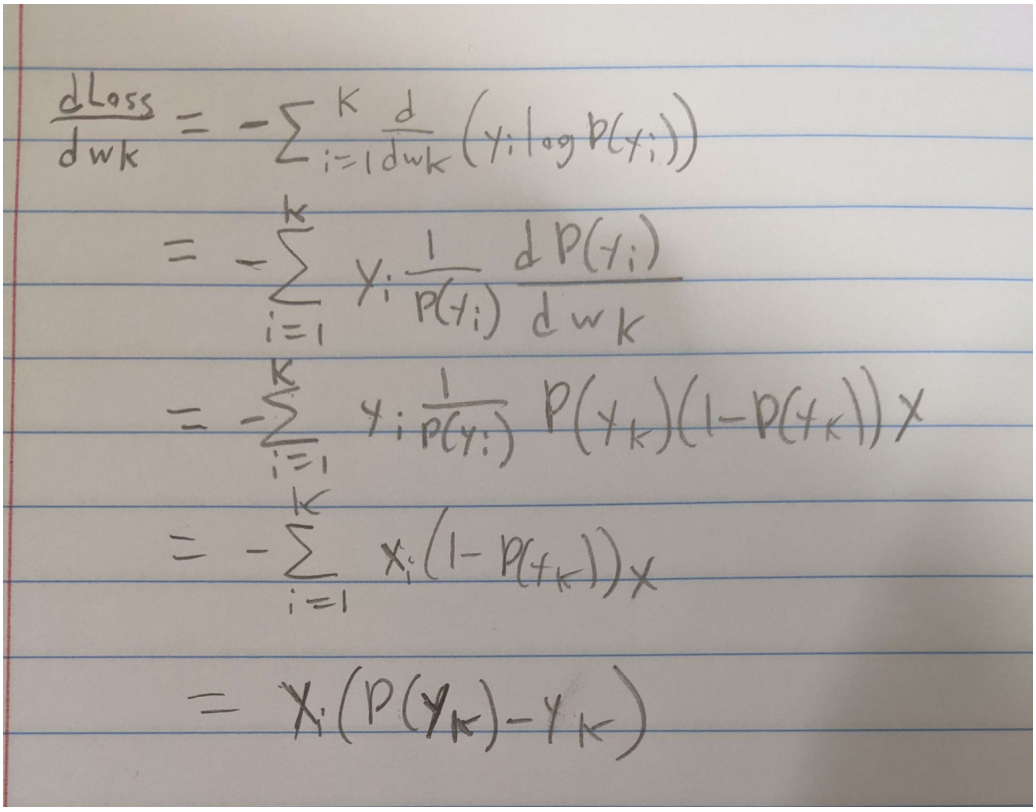
1. Overview
   a. The task two model incorporates similar algorithms and processes as task one's but differs in the output and the processes that lead to it.
   b. The constructor of TaskTwoModel takes in the same parameters as task one and uses them in the same way.
   c. Properties:
      i. Holds the same maps as taskOne: diagramMap and testingInputMap
         1. testingInputMap set to a constant 1000 samples

    ii. Instead of holding one vector of weights, it holds a list of 4 weight vectors (property name weightsList).
        1. list[0] = redClassification weights
        2. list[1] = blueClassification weights
        3. etc
2. Input and Feature Space
    a. The input and feature space are the same as in task one.
        i. The diagram still takes in an encoded diagram as input
        ii. The feature space represents the diagram and where wires overlap which is useful for determining which wire to cut.
    b. Thus, diagramMap and inputTestingMap are populated in the same way.
        i. The read diagram methods are modified to read the 3rd line as the label instead of the 2nd.
3. Output
    a. The model outputs a 1, 2, 3, or 4 which represents the wire that needs to be cut.
        i. The output is the integer representation of the wire instead of the encoded version since it would match the formatting of the label in the text file and the predictLabel method can input it in that form.
4. Model
    a. Logistic regression model will be used that will determine probabilities through Soft Max instead of the sigmoid function.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

        i.
        ii. j through K represents each weight vector in weightsList
        iii. Implementation in probabilityOfLabel
            1. Takes an input vector and weights vector as parameters.
            2. totalWeights = 0
            3. for each weight in weightsList
                a. totalWeights += eRaisedToDotProd(input, weightVector)
            4. return eRaisedToDotProd(input, weights) / totalWeights
    b. As previously mentioned, the feature space will be constructed with the same getFeatureSpace method in task one.
        i. Thus the same non linear overlap features will be used in addition to the input vector
5. Loss
    a. The loss function is cross entropy loss as follows:
        i. Loss = -1/N * ∑(sum from i = 1 to N)[yi * log(P(yi = 1)) + (1- yi)* log(1 - P(yi = 1))]
        ii. Add l2 regression term as well
    b. The loss function is calculated in the calculateLoss() method.

i. The function takes in a HashMap of Integer arrays to Integers
ii. for each entry in the map parameter
    1. encodedLabel = encodeWireColor(entry.getValue())
        a. Now the label is an array that lines up with the weightsList array.
            i. weightsList[0] will give the weight vector for the red wire classification and encodedLabel[0] will give whether the label was red wire or not.
    2. for each weight vector in weightsList
        a. totalLoss = totalLoss - encodedLabel[i] * ln(probabilityOfLabel(entry.getKey(), weightsList.get(i)))
iii. for each weight vector in weights vector
    1. for each weight in the weight vector
        a. l2RegressionFactor += weight^2
iv. l2RegressionFactor *= lambda
v. return ((totalLoss)/inputMap.size()) + l2RegressionFactor

6. This model also uses l2 regression to prevent overfitting
7. Gradient of the Loss
    a. 
    b. Add in the l2 regression term
    c. + 2 * λ * wj
    d. Implementation
        i. The computeLossGradient method takes in an input vector, weight vector, and label.

1. It will be called in the SGD method with varying weights being parsed in with their respective labels.
2. The label will be either 0 or 1, which will represent yk

ii. Construct a lossGradient double array as follows:
1. precalculate P(yk) - yk as probabilityMinusLabel = probabilityOfLabel(input, weights) - label
2. for each input
a. lossGradient[i] = (input[i] * probabilityMinusLabel) + (2 * lambda * weights[i])

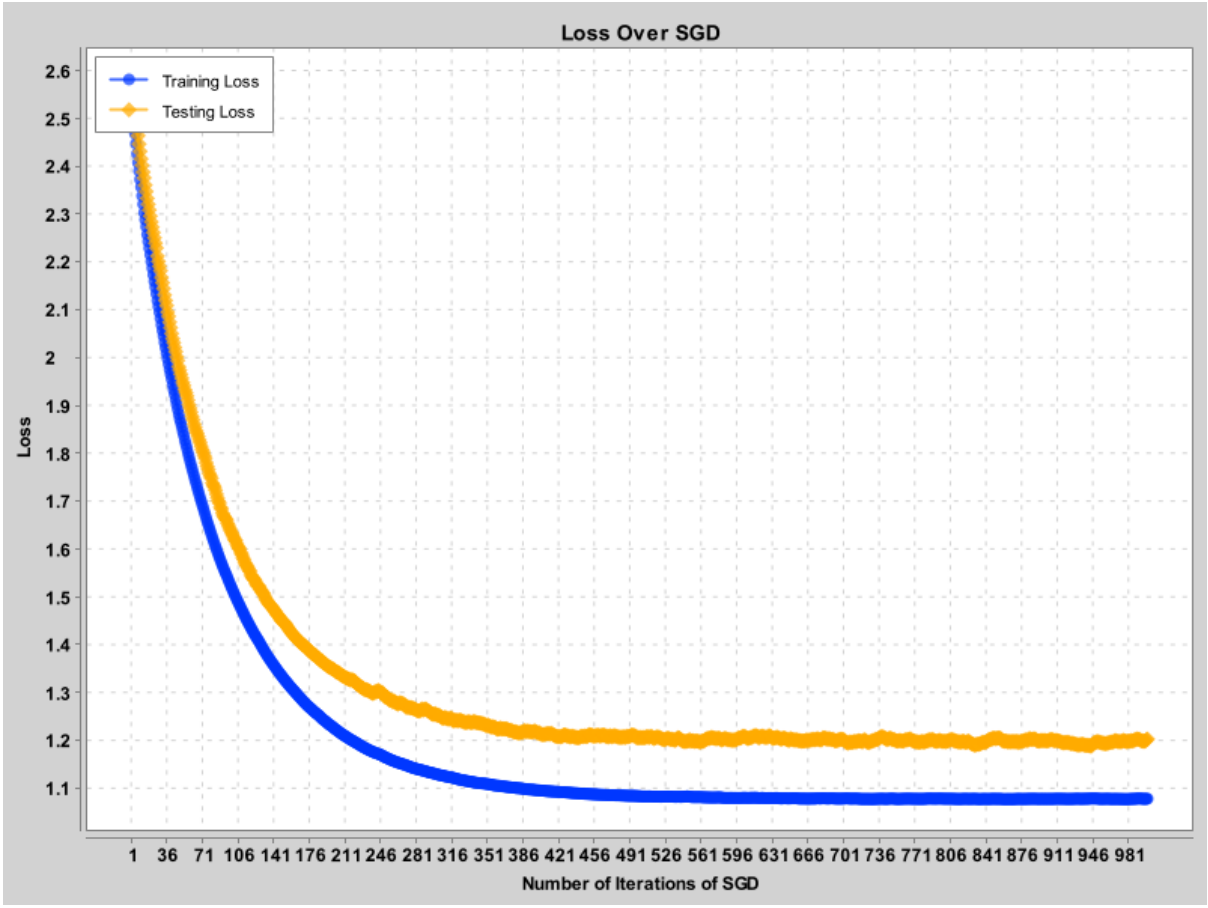
iii. return lossGradient

8. Training Algorithm
a. Also uses Stochastic Gradient descent implemented in SGD()
i. Same process as SGD() in TaskOneModel but now needs to update the list of weight vectors instead of just one
b. The method starts by reading the training and testing data which populates the diagramMap and testingInputMaps.
c. Populate each weight vector in weightsList with random weights
d. For a number of iterations
i. Populate the randomEntries list with a number of entries from diagramMap.
1. Same as in task one

ii. Create an ArrayList of ArrayLists of doubles called classGradients
1. This will hold an ArrayList of double arrays for each color.
a. The ArrayList of double arrays holds the list of lossGradients for that color. The number of lossGradients equals the number of randomEntries picked out for SGD.
2. For each entry in randomEntries
a. for each weight vector in weightsList
i. encodedLabel = encodeWireColor(entry.getValue())
ii. classGradients[i] = computeLossGradient(entry.getKey(), weightsList.get(i) ,encodedLabel[i])
1. Each weight vector will have its corresponding label (0 or 1) sent into the function

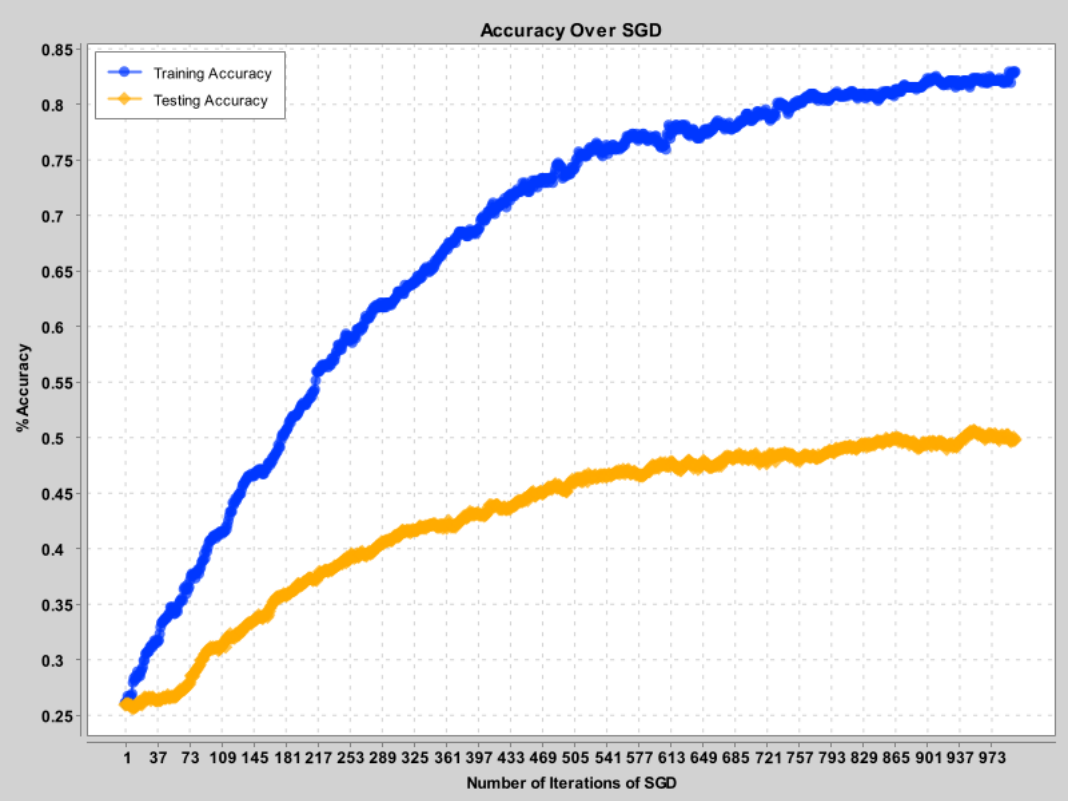
iii. Create an ArrayList of doubles called weightSetGradients
1. This will hold the averaged loss gradient vector for each color.
2. To populate, average the contents of each ArrrayList of doubles in classGradient and add that vector to weightSetGrradients.

iv. Then update each weight vector with its respective loss gradient in weightSetGradient
1. weightsList.get(i)[j] = weightsList.get(i)[j] - alpha * weightSetGradients.get(i)[j]

        a. weightsList.get(i)[j] represennts the jth weight in the ith weight vector.
    2. alpha is the learning rate set with the constructor
9. Prediction Accuracy
    a. The predictLabel method takes in an input and outputs its predicted label.
        i. It uses the probabilityOfLabel method on each wire weight and returns the numerical representation of the wire with the highest weight.
    b. The testSuccessRate method functions in the same way as it did in TaskOneModel
10. Selections
    a. Alpha = 0.01 was used again
    b. Lambda was set to 0.05, higher than in task 1 since there tended to be a larger difference between testing and training accuracy with lambda = 0.01
11. Loss Graph:
    a. 

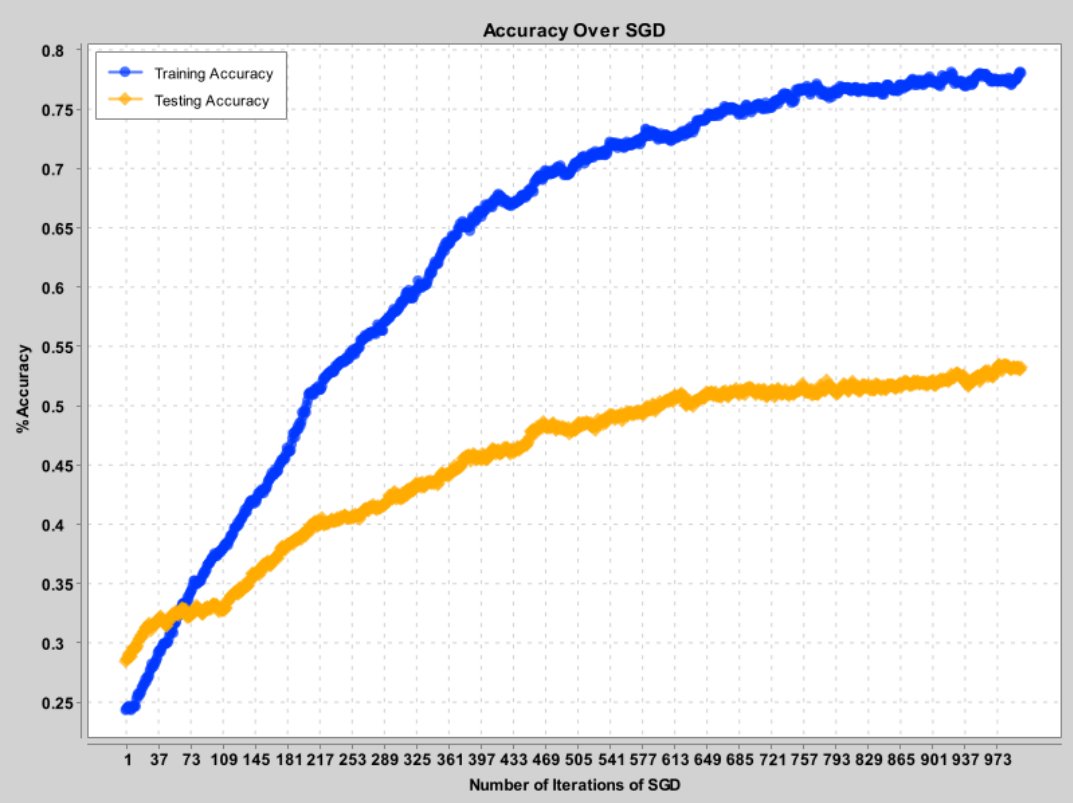
        i. This graph shows the loss over the training (Size of training set = 2000) and testing sets over iterations of SGD.
        ii. Both consistently decrease, demonstrating learning.
        iii. Testing Loss does not increase over iteration showing the model isn't overfitting to the training set
12. Performance Accuracy:

a.



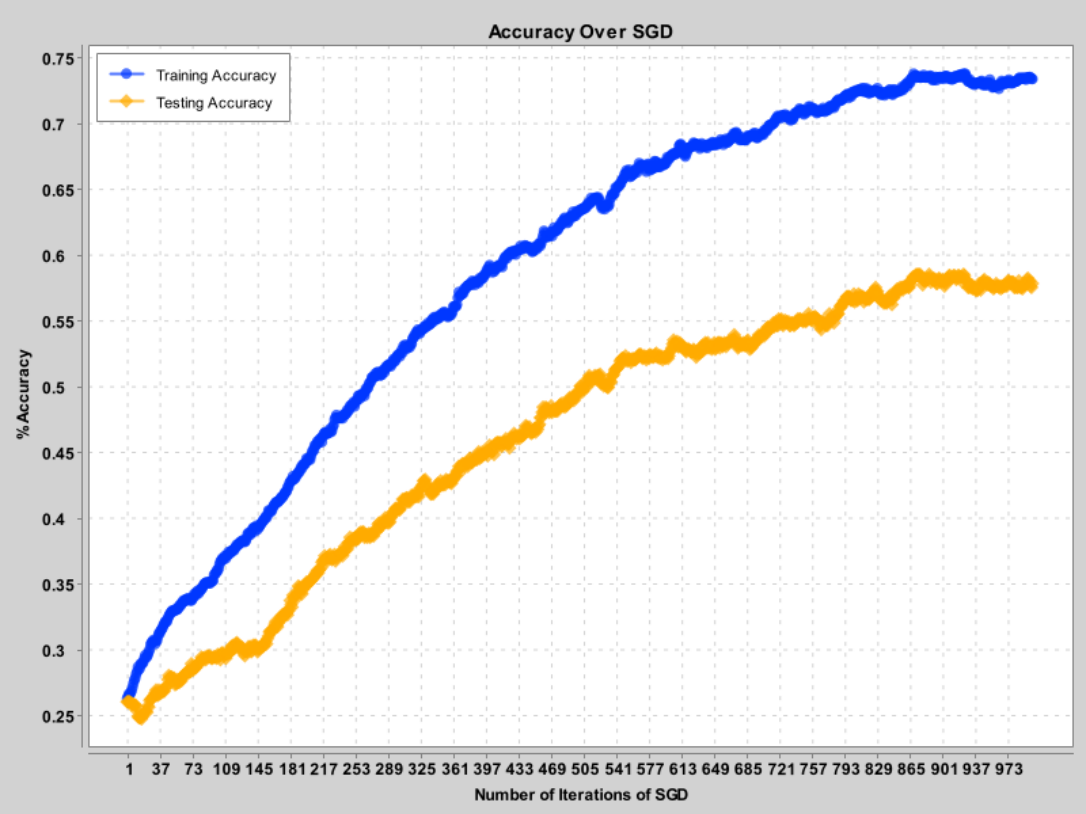
i. Accuracy with 500 training samples

b.



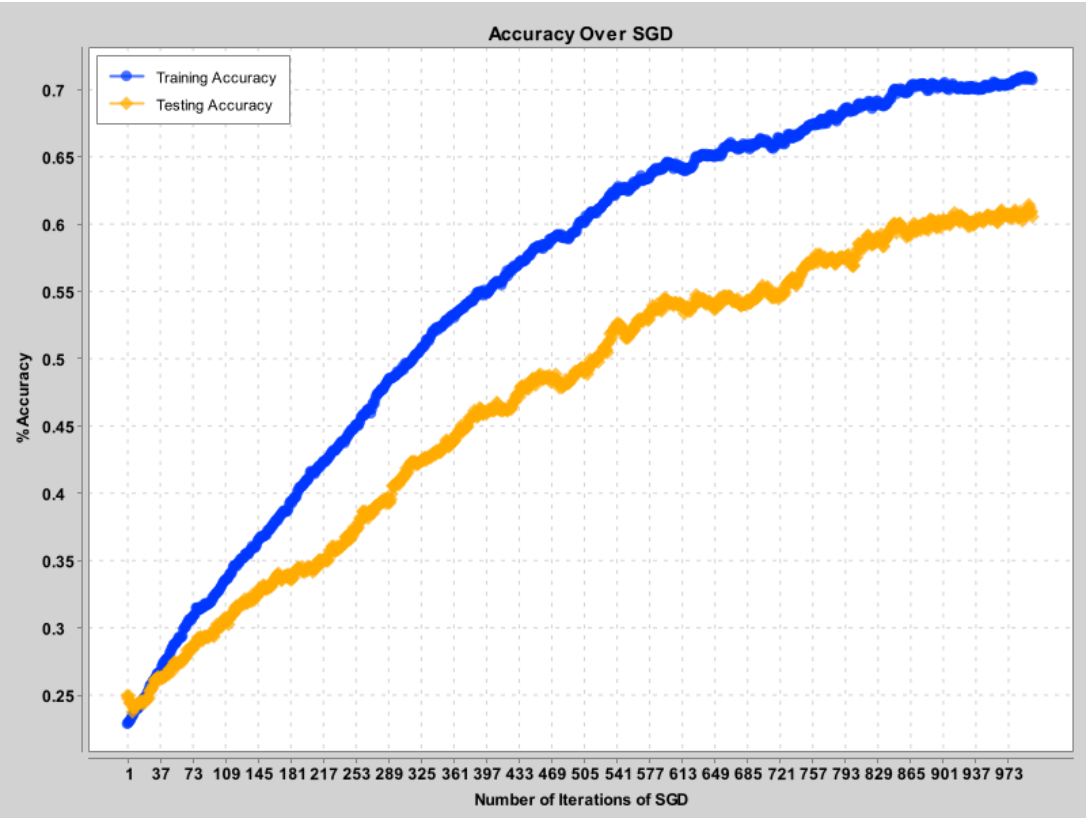
i. Accuracy with 1000 training samples

c.

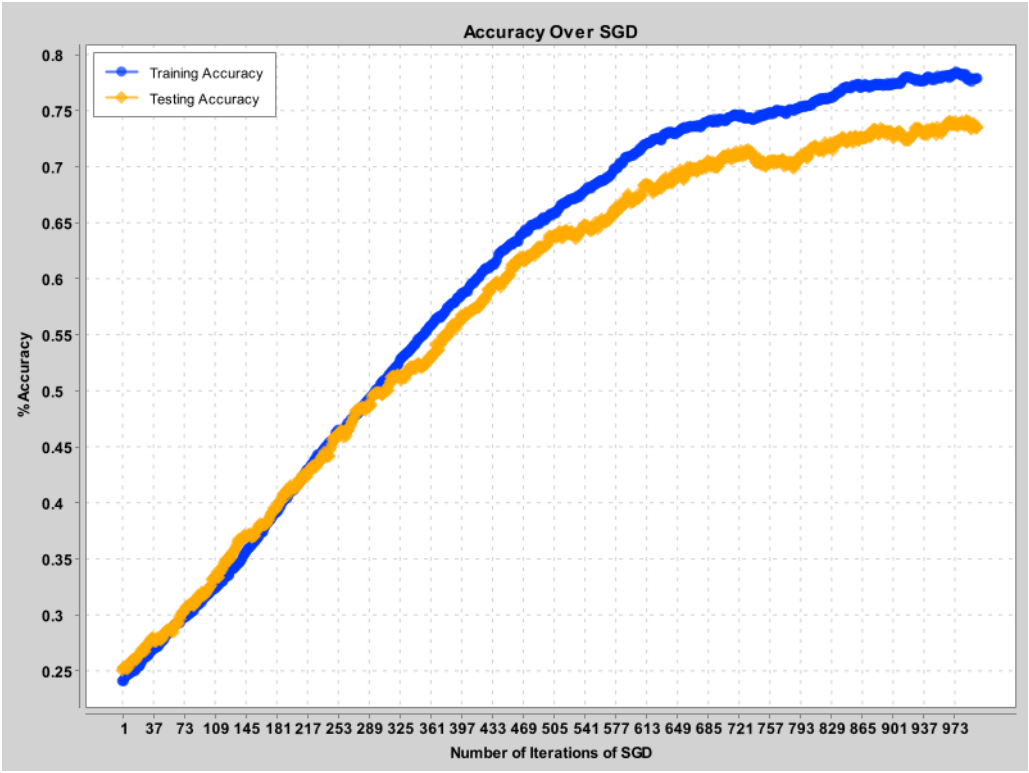


i. Accuracy with 2000 training samples

d.



i. Accuracy with 3000 training samples



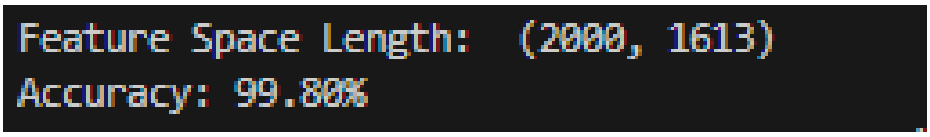
e.

i. Accuracy with 5000 training samples

f. The testing accuracy over SGD rises significantly with more training samples.
   i. At 1000 iterations, the 500 sample graph has 50% accuracy while the 5000 sample graph has 74% accuracy.
g. The accuracy consistently increasing over the runtime demonstrates the model is not overfit and the accuracy demonstrates it generalizes well to testing data.

Bonus

1. Overview
   a. I used the sklearn framework in python to fit models to task one and two problems.
   b. The python files are located in the Project3 directory
      i. TaskOneWithFramework.py
      ii. TaskTwoWithFramework.py
   c. Each uses text files written to by GenerateWireDiagrams.
      i. The text files can be found in a folder within the data folder called ExtraCreditData
      ii. Each text file has a set size to compare results with varying numbers of samples.
2. TaskOneWithFramework

a. In main, the program will read the training and testing data from specified file locations then use a method that generates a feature space from the text file input.
b. Then, a model is generated with
    i. model = LogisticRegression(max_iter=MAX_ITERATIONS)
        1. Set a maximum number of iterations to a constant
            a. In testing MAX_ITERATIONS is set to 1000, the same as TaskOneModel and TaskTwoModel
    ii. model.fit(X_train_feature_space, y_train)
        1. Fit the model with the input feature space and their respective labels.
    iii. The following lines predicts labels and assigns accuracy
        1. predictions = model.predict(X_test_feature_space)
        2. accuracy = accuracy_score(y_test, predictions)
        3. print("Accuracy: {:.2f}%".format(accuracy * 100))
    iv. Result on training set of 2000 with the same feature space from TaskOne:
        1. 

            a. 2000 inputs
            b. 1613 features
3. TaskTwoWithFrameworrk
    a. Reading the data and generating feature space is the same in TaskOneWithFramework.
    b. The model needs to be specified as multinomial
        i. model = LogisticRegression(multi_class='multinomial', max_iter=MAX_ITERATIONS)
        ii. The output also needs to be flattened to its one hot encoded version:
            1. y_train_flat = np.argmax(y_train, axis=1)
            2. y_test_flat = np.argmax(y_test, axis=1)
        iii. model.fit(X_train_feature_space, y_train_flat)
        iv. y_pred = model.predict(X_test_feature_space)
        v. accuracy = accuracy_score(y_test_flat, y_pred)
        vi. print(f"Accuracy: {accuracy:.4f}")
    c. Result on training set of 2000 with the same feature space from TaskTwo:
        i. 

4. Note:
    a. If either of these py files are being run, you will have to change the paths being read from at the top of main.
5. Solutions with Minimal Data
    a. The following data will show tests with different combinations of feature spaces and training sets.

b. Training Sets
    i. In the BonusData folder in Data, there are 6 text files for each task.
        1. There is a testing file for each that will have 1000 samples.
        2. Then there are training text files with sample sizes 100, 200, 500, 1000, 2000
c. Feature Spaces
    i. First feature space is the same used in task one and two:
        1. get_full_feature_space
    ii. Second is just the baseline and overlap features from the first feature space.
        1. get_overlap_feature_space
    iii. The 3rd feature space utilizes the overlap features and just the representation of the red and yellow wires.
        1. The red and yellow wire representation is done by creating matrices for each where a 0 in a cell means that the correlation cell in the wire diagram does not have that wire in it. A in a cell indicates that the correlation cell in the wire diagram does contain the wire.
        2. Each matrix would be 20x20 and only made up of 0s and 1s. Flattening and appending the two would give a vector of size 800.
        3. Thus the total feature space with this vector, baseline feature, and overlaps, would be 813 in length.
        4. The intuition behind this representation is task one needs to know if the red wire is laid after the yellow wire.
        5. get_red_yellow_feature_space
d. I will show the accuracy of combinations of these feature spaces and training set sizes in the following tables:
    i. Each data point in the table was found by changing:
        1. X_train, y_train parameters through which text file
        2. X_train_feature_space and x_test_feature_space through the feature space function
e.

Task One	Full Feature Space	Red Yellow Feature Space	Overlap Feature Space
100 samples	Accuracy: 72.40%	Accuracy: 83.50%	Accuracy: 99.70%
200 samples	Accuracy: 82.40%	Accuracy: 95.90%	Accuracy: 99.60%
500 samples	Accuracy: 97.90%	Accuracy: 99.40%	Accuracy: 99.70%
1000 samples	Accuracy:	Accuracy:	Accuracy:

	99.10%	99.80%	99.70%
2000 samples	Accuracy: 99.80%	Accuracy: 99.90%	Accuracy: 99.90%

i. The simpler, overlap feature space performs much better at lower sample sizes.
ii. The red yellow feature space outperforms the others at 1000 samples perhaps due to that point it is simpler than the full feature space but its extra features help enough for it to perform better than the overlap features.

f.

Task Two	Full Feature Space	Red Yellow Feature Space	Overlap Feature Space
100 samples	Accuracy: 0.3750%	Accuracy: 0.4770%	Accuracy: 0.9860%
200 samples	Accuracy: 0.4810%	Accuracy: 0.5550%	Accuracy: 0.9940%
500 samples	Accuracy: 0.6670%	Accuracy: 0.8270%	Accuracy: 0.9920%
1000 samples	Accuracy: 0.8820%	Accuracy: 0.9800%	Accuracy: 0.9890%
2000 samples	Accuracy: 0.9870%	Accuracy: 0.9890%	Accuracy: 0.9900%

i. These results show that the simpler feature space matters even more for the exclusive classification problem.
ii. The red yellow feature space outperforms the full feature space as a result of being simpler.

g. These tables show that successful classification can be made with the simple overlap feature space with very low sample sizes.
i. 13 features, 100 samples