James Mccune

Machine Learning - Project 5

a)

Y: 110 X : 10.0

Y: 11 X : -1.0

Y: 59 X : 7.0

Y: 11 X : 1.0

Y: 19 X : 3.0

Y: 11 X : 1.0

Y: 26 X : 4.0

Y: 14 X : -2.0

Y: 91 X : 9.0

Y: 46 X : 6.0

Y: 46 X : 6.0

Y: 10 X : 0.0

b)

Linear Regression without Regularization

y = 8.882239382239389 + 7.895752895752898x

MSE = 154.09362934362935

c)

Lambda( 0.1 ) : In-Sample Error is ( 107.80686473475485 ) : Validation Error is ( 156.34752411980287 ) Lambda( 1.0 ) : In-Sample Error is ( 108.75865717858063 ) : Validation Error is ( 156.22908934508595 )

Lambda( 10.0 ) : In-Sample Error is ( 123.25580293046376 ) : Validation Error is ( 170.72875335512268 )

Lambda( 100.0 ) : In-Sample Error is ( 289.2884925518699 ) : Validation Error is ( 395.7137821148088 )

d) The final lambda was 1.

e)

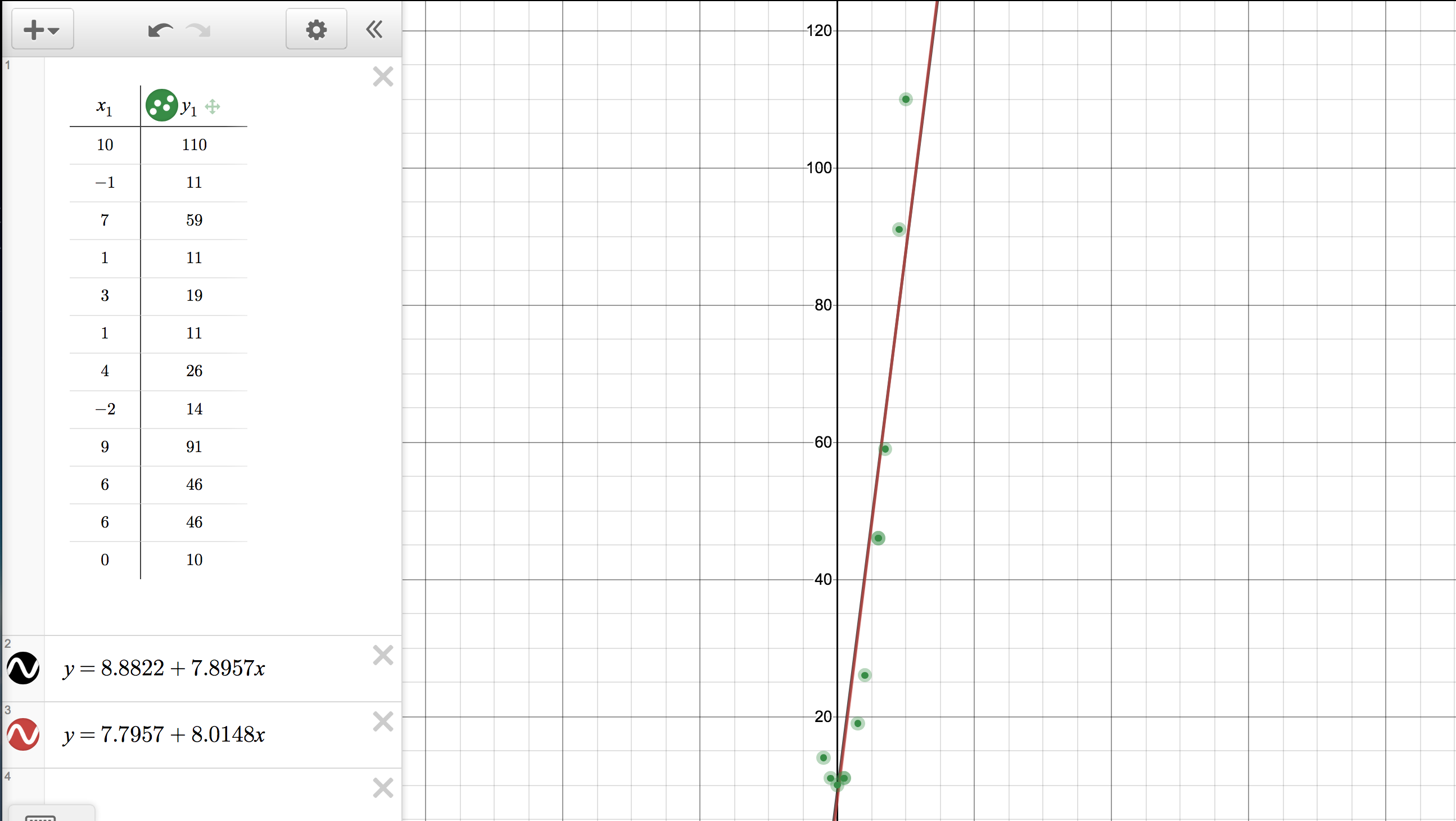
Linear Regression with Regularization. Lambda is 1.0

y = 7.7957833815626225 + 8.014882182720132x

MSE = 154.71987671323942

Below is the graph of the data and equations. The linear regression with regularization is colored red and

The simple linear regression is colored black. It’s difficult to distinguish the lines but they are both there.



Below is the code.

**LinearRegression.java**

public class LinearRegression {

private double[] weights;

private DataSet trainingData;

private double lambda = 0;

private double error;

public LinearRegression(DataSet trainingData){

this.trainingData = trainingData;

weights = new double[trainingData.getRecords().get(0).getValues().length + 1];

}

public LinearRegression(DataSet trainingData, double lambda){

this.trainingData = trainingData;

this.lambda = lambda;

weights = new double[trainingData.getRecords().get(0).getValues().length + 1];

}

//solve with cross validation and select best model parameter

public static LinearRegression solveWithVFoldCrossValidation(DataSet trainingData, int folds, double[] lambda){

LinearRegression chosen;

int validationSetSize = trainingData.getRecords().size() / folds;

double[] trainingError = new double[lambda.length];

double[] validationError = new double[lambda.length];

//for each model parameter change

for(int model = 0; model < validationError.length; model++){

DataSet[] dataAndValidationSets;

//for each fold in the training set

for(int fold = 0; fold < folds; fold++){

//seperate dataset into training and validation sets based on fold

int start = (fold == 0) ? 0 : ((validationSetSize \* fold));

int end = (fold == 0) ? (validationSetSize - 1) : ((validationSetSize \* (fold + 1)) - 1);

dataAndValidationSets = trainingData.divideForValidation(start, end);

//create a linear regression model and train it on the training set

LinearRegression candidate = new LinearRegression(dataAndValidationSets[0], lambda[model]);

candidate.solveWithRegularization(lambda[model]);

//calculate and sum the error from the training and validation sets

trainingError[model] += LinearRegression.getMeanSquaredError(candidate, dataAndValidationSets[0]);

validationError[model] += LinearRegression.getMeanSquaredError(candidate, dataAndValidationSets[1]);

}

trainingError[model] /= lambda.length;

validationError[model] /= lambda.length;

}

//find the model that generated the least error

int leastError = 0;

for(int i = 0; i < validationError.length; i++){

if(validationError[i] < validationError[leastError]) leastError = i;

System.out.println("\t\tLambda( " + lambda[i] + " ) : In-Sample Error is ( " + trainingError[i] + " ) : Validation Error is ( " + validationError[i] + " )");

}

chosen = new LinearRegression(trainingData, lambda[leastError]);

chosen.solveWithRegularization(lambda[leastError]);

return chosen;

}

//standard linear regression

public void solve(){

Matrix data = trainingData.dataAsMatrix();

Matrix classes = trainingData.classAsMatrix();

Matrix result = Matrix.multiply(Matrix.transpose(data), data);

result = Matrix.multiply(Matrix.inverse2(result), Matrix.transpose(data));

result = Matrix.multiply(result, classes);

extractWeights(result.asArray());

error = LinearRegression.getMeanSquaredError(this, trainingData);

}

//linear regression with regularization

public void solveWithRegularization(double lambda){

Matrix data = trainingData.dataAsMatrix();

Matrix classes = trainingData.classAsMatrix();

Matrix result = Matrix.multiply(Matrix.transpose(data), data);

result = Matrix.add(result, Matrix.getIdentityMatrix(Matrix.getHeight(result), lambda));

result = Matrix.multiply(Matrix.inverse2(result), Matrix.transpose(data));

result = Matrix.multiply(result, classes);

extractWeights(result.asArray());

error = LinearRegression.getMeanSquaredError(this, trainingData);

}

public double getErrorForRegularization(DataSet dataset){

double error = 0;

Matrix data = dataset.dataAsMatrix();

System.out.println(Matrix.toString(data));

Matrix classes = dataset.classAsMatrix();

Matrix result1 = Matrix.scale(Matrix.transpose(classes), 1/Matrix.getHeight(data));

System.out.println(Matrix.toString(result1));

Matrix h = Matrix.multiply(Matrix.transpose(data), data);

h = Matrix.add(h, Matrix.getIdentityMatrix(Matrix.getHeight(h), lambda));

h = Matrix.multiply(data, Matrix.inverse2(h));

h = Matrix.multiply(h, Matrix.transpose(data));

System.out.println(Matrix.toString(h));

Matrix result2 = Matrix.subtract(Matrix.getIdentityMatrix(Matrix.getHeight(h)), h);

result2 = Matrix.multiply(result2, result2);

System.out.println(Matrix.toString(result2));

result1 = Matrix.multiply(result1, result2);

result1 = Matrix.multiply(result1, classes);

System.out.println(Matrix.toString(result1));

return error;

}

public static double getMeanSquaredError(LinearRegression model, DataSet testData){

double[][] data = testData.dataAsArray();

double[][] result = testData.classAsArray();

double sum = 0;

for(int row = 0; row < data.length; row++){

double y = 0;

for(int col = 0; col < data[row].length; col++){

y += model.weights[col] \* data[row][col];

}

sum += Math.pow(result[row][0] - y, 2);

}

return (sum/data.length);

}

public String printEquation(){

StringBuilder stringBuilder = new StringBuilder();

stringBuilder.append("y = ");

for(int i = 0; i < weights.length; i++){

if(i == 0) stringBuilder.append(weights[i] + " + ");

else if(i == 1 && weights.length > 2) stringBuilder.append(weights[i] + "x + ");

else if(i == 1 && weights.length == 2) stringBuilder.append(weights[i] + "x");

else if(i == weights.length - 1) stringBuilder.append(weights[i] + "x^" + i);

else stringBuilder.append(weights[i] + "x^" + i + " + ");

}

return stringBuilder.toString();

}

//extract the weights from a double[][] into a double[]

private void extractWeights(Double[][] result){

if(result != null){

if(result[0].length == 1){

weights = new double[result.length];

for(int row = 0; row < result.length; row++){

weights[row] = result[row][0];

}

}

}

}

public double getLambda(){

return lambda;

}

public double getError(){

return error;

}

}

**Matrix.java**

**public** **class** Matrix {

**private** Double[][] matrix;

**public** Double[][] asArray(){

**return** matrix;

}

**public** Matrix(**int** height, **int** width, **double** cellValue){

matrix = **new** Double[height][width];

**for**(**int** r = 0; r < height; r++){

**for**(**int** c = 0; c < width; c++){

matrix[r][c] = cellValue;

}

}

}

**public** Matrix(**int** height, **int** width){

matrix = **new** Double[height][width];

}

**public** Matrix(**double**[][] array){

**if**(array != **null**){

matrix = **new** Double[array.length][array[0].length];

**for**(**int** r = 0; r < matrix.length; r++){

**for**(**int** c = 0; c < matrix[r].length; c++){

matrix[r][c] = array[r][c];

}

}

}

}

**public** **static** String toString(Matrix matrix){

**if**(matrix == **null**) **return** "";

StringBuilder stringBuilder = **new** StringBuilder();

**for**(**int** i = 0; i < *getHeight*(matrix); i++){

**for**(**int** j = 0; j < *getWidth*(matrix); j++){

stringBuilder.append(String.*format*("%6.6f\t", *getCell*(matrix, i, j)));

}

stringBuilder.append("\n");

}

**return** stringBuilder.toString();

}

**public** **static** Integer getHeight(Matrix matrix){

**if**(matrix != **null**) **return** matrix.matrix.length;

**return** **null**;

}

**public** **static** Integer getWidth(Matrix matrix){

**if**(matrix != **null**) **return** matrix.matrix[0].length;

**return** **null**;

}

**public** **static** Double getCell(Matrix matrix, **int** row, **int** col){

**if**(matrix != **null**) **return** matrix.matrix[row][col];

**return** **null**;

}

**public** **static** **void** setCell(Matrix matrix, **int** row, **int** col, **double** value){

**if**(matrix != **null**) matrix.matrix[row][col] = value;

}

**public** **static** Matrix getCofactorMatrix(Matrix matrix){

**if**(matrix == **null**) **return** **null**;

**if**(*getHeight*(matrix) != *getWidth*(matrix)) **return** **null**;

Matrix comatrix = **new** Matrix(*getHeight*(matrix), *getWidth*(matrix), 0);

**int** sign = 1;

**for**(**int** row = 0; row < *getHeight*(matrix); row++){

**for**(**int** col = 0; col < *getWidth*(matrix); col++){

Matrix sub = *getSubMatrix*(matrix, row, col);

sign = (**int**) Math.*pow*(-1., row+1 + col+1);

*setCell*(comatrix, row, col, sign \* *determinant*(sub));

}

}

**return** comatrix;

}

//transposes given matrix

**public** **static** Matrix transpose(Matrix matrix){

**if**(matrix == **null**) **return** **null**;

Matrix transpose = **new** Matrix(*getWidth*(matrix), *getHeight*(matrix));

**for**(**int** mc = 0; mc < *getHeight*(transpose); mc++){ //for each column in matrix

**for**(**int** mr = 0; mr < *getWidth*(transpose); mr++){ //for each row in matrix

*setCell*(transpose, mc, mr, *getCell*(matrix, mr, mc));

}

}

**return** transpose;

}

//helper function used to cross-out a row and column from a larger matrix and return the sub-matrix

**public** **static** Matrix getSubMatrix(Matrix matrix, **int** row, **int** col){

**if**(matrix == **null**) **return** **null**;

**if**(*getHeight*(matrix) == 1 && *getWidth*(matrix) == 1) **return** matrix; //matrix cannot be reduced

Matrix sub = **new** Matrix(*getHeight*(matrix) - 1, *getWidth*(matrix) - 1);

**int** mr = 0;

**int** mc = 0;

**int** sr = 0;

**int** sc = 0;

**while**(sr < *getHeight*(sub) && mr < *getHeight*(matrix)){ //while within row boundries

**if**(mr == row){ //found row to skip

mr++;

**continue**;

}

**while**(sc < *getWidth*(sub) && mc < *getWidth*(matrix)){ //while within column boundries

**if**(mc == col){ //found col to skip

mc++;

**continue**;

}

*setCell*(sub, sr, sc, *getCell*(matrix, mr, mc));

sc++;

mc++;

}

mc = 0;

sc = 0;

sr++;

mr++;

}

**return** sub;

}

**public** **static** Matrix getIdentityMatrix(**int** size, **double** scale){

Matrix identity = **new** Matrix(size, size);

**for**(**int** ir = 0; ir < *getHeight*(identity); ir++){

**for**(**int** ic = 0; ic < *getWidth*(identity); ic++){

*setCell*(identity, ir, ic, ir == ic ? scale : 0);

}

}

**return** identity;

}

**public** **static** Matrix getIdentityMatrix(**int** size){

Matrix identity = **new** Matrix(size, size);

**for**(**int** ir = 0; ir < *getHeight*(identity); ir++){

**for**(**int** ic = 0; ic < *getWidth*(identity); ic++){

*setCell*(identity, ir, ic, ir == ic ? 1 : 0);

}

}

**return** identity;

}

**public** **static** Matrix scale(Matrix matrix, **double** scale){

Matrix scaled = **new** Matrix(*getHeight*(matrix), *getWidth*(matrix));

**for**(**int** row = 0; row < *getHeight*(scaled); row++){

**for**(**int** col = 0; col < *getWidth*(scaled); col++){

*setCell*(scaled, row, col, scale \* *getCell*(matrix, row, col));

}

}

**return** scaled;

}

**public** **static** Matrix subtract(Matrix left, Matrix right){

**if**(left == **null**) **return** **null**;

**if**(right == **null**) **return** **null**;

**if**(*getHeight*(left) != *getHeight*(right)) **return** **null**;

**if**(*getWidth*(left) != *getWidth*(right)) **return** **null**;

Matrix diff = **new** Matrix(*getHeight*(left), *getWidth*(left));

**for**(**int** row = 0; row < *getHeight*(diff); row++){

**for**(**int** col = 0; col < *getWidth*(diff); col++){

*setCell*(diff, row, col, *getCell*(left, row, col) - *getCell*(right, row, col));

}

}

**return** diff;

}

**public** **static** Matrix add(Matrix left, Matrix right){

**if**(left == **null**) **return** **null**;

**if**(right == **null**) **return** **null**;

**if**(*getHeight*(left) != *getHeight*(right)) **return** **null**;

**if**(*getWidth*(left) != *getWidth*(right)) **return** **null**;

Matrix sum = **new** Matrix(*getHeight*(left), *getWidth*(left));

**for**(**int** row = 0; row < *getHeight*(sum); row++){

**for**(**int** col = 0; col < *getWidth*(sum); col++){

*setCell*(sum, row, col, *getCell*(left, row, col) + *getCell*(right, row, col));

}

}

**return** sum;

}

**public** **static** Matrix multiply(Matrix left, Matrix right){

**if**(left == **null**) **return** **null**;

**if**(right == **null**) **return** **null**;

**if**(*getWidth*(left) != *getHeight*(right)) **return** **null**;

Matrix result = **new** Matrix(*getHeight*(left), *getWidth*(right));

**for**(**int** lr = 0; lr < *getHeight*(left); lr++){ //for each row in left

**for**(**int** rc = 0; rc < *getWidth*(right); rc++){ //for each column in right

**double** sum = 0;

**for**(**int** lc = 0; lc < *getWidth*(left); lc++){ //for each column in left

sum += *getCell*(left, lr, lc) \* *getCell*(right, lc, rc); //calculate sum

}

*setCell*(result, lr, rc, sum);

}

}

**return** result;

}

//calculates the determinant of a 3x3 matrix

**public** **static** Double determinant3(Matrix matrix){

**if**(matrix == **null**) **return** **null**;

**if**(*getHeight*(matrix) != 3 && *getWidth*(matrix) != 3) **return** **null**;

Double determinant = 0.;

determinant += *getCell*(matrix, 0, 0) \* *getCell*(matrix, 1, 1) \* *getCell*(matrix, 2, 2);

determinant += *getCell*(matrix, 0, 1) \* *getCell*(matrix, 1, 2) \* *getCell*(matrix, 2, 0);

determinant += *getCell*(matrix, 0, 2) \* *getCell*(matrix, 1, 0) \* *getCell*(matrix, 2, 1);

determinant += *getCell*(matrix, 2, 0) \* *getCell*(matrix, 1, 1) \* *getCell*(matrix, 0, 2);

determinant += *getCell*(matrix, 2, 1) \* *getCell*(matrix, 1, 2) \* *getCell*(matrix, 0, 0);

determinant += *getCell*(matrix, 2, 2) \* *getCell*(matrix, 1, 0) \* *getCell*(matrix, 0, 1);

**return** determinant;

}

//calculates the determinant of a 2x2 matrix

**public** **static** Double determinant2(Matrix matrix){

**if**(matrix == **null**) **return** **null**; //undefined

**if**(*getHeight*(matrix) != 2 && *getWidth*(matrix) != 2) **return** **null**;

**if**(*getHeight*(matrix) != *getWidth*(matrix)) **return** **null**; //not square

**return** (*getCell*(matrix, 0, 0) \* *getCell*(matrix, 1, 1)) - (*getCell*(matrix, 1, 0) \* *getCell*(matrix, 0, 1));

}

//calculates the inverse of a 2x2 matrix

**public** **static** Matrix inverse2(Matrix matrix){

**if**(matrix == **null**) **return** **null**; //undefined

**if**(*getHeight*(matrix) != *getWidth*(matrix)) **return** **null**; //not square

**if**(*getHeight*(matrix) != 2) **return** **null**; //not a 2x2

Matrix inverse = **new** Matrix(*getHeight*(matrix), *getWidth*(matrix));

**double** determinant = 1/((*getCell*(matrix, 0, 0) \* *getCell*(matrix, 1, 1)) - (*getCell*(matrix, 1, 0) \* *getCell*(matrix, 0, 1)));

*setCell*(inverse, 0, 0, *getCell*(matrix, 1, 1) \* determinant);

*setCell*(inverse, 1, 1, *getCell*(matrix, 0, 0) \* determinant);

*setCell*(inverse, 0, 1, *getCell*(matrix, 0, 1) \* -determinant);

*setCell*(inverse, 1, 0, *getCell*(matrix, 1, 0) \* -determinant);

**return** inverse;

}

//calculates the determinant of a NxN matrix given its cofactor matrix

**public** **static** Double determinant(Matrix matrix, Matrix cofactor){

**if**(matrix != **null** && cofactor != **null**){

**if**(*getHeight*(matrix) == *getWidth*(matrix) && *getHeight*(cofactor) == *getWidth*(cofactor) && *getHeight*(matrix) == *getHeight*(cofactor)){ //both are square

**double** determinant = 0;

**for**(**int** col = 0; col < *getHeight*(cofactor); col++){

determinant += *getCell*(matrix, 0, col) \* *getCell*(cofactor, 0, col);

}

**return** determinant;

}

}

**return** **null**;

}

//recursively find determinant via cofactor expansion always choosing 1st row as expansion

**public** **static** Double determinant(Matrix matrix) {

**if**(*getHeight*(matrix) != *getWidth*(matrix)) **return** **null**; //does not exist, matrix is not square

**if**(*getHeight*(matrix) == 3) **return** *determinant3*(matrix);

**if**(*getHeight*(matrix) == 2) **return** *determinant2*(matrix);

**double** determinant = 0;

**int** sign = 1;

**for**(**int** mc = 0; mc < *getWidth*(matrix); mc++) {

Matrix sub = *getSubMatrix*(matrix, 0, mc);

determinant += sign \* *getCell*(matrix, 0, mc) \* *determinant*(sub);

sign \*= -1;

}

**return** determinant;

}

}

DataSet.java

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

import java.util.Random;

public class DataSet implements Iterable<Record>{

private String name;

private FileReader dataFile;

private String[] attributeNames;

private List<Record> records;

public DataSet(String name, FileReader dataFile){

this.name = name;

this.dataFile = dataFile;

records = new ArrayList<>();

}

public DataSet(String name){

this.name = name;

records = new ArrayList<>();

}

/\*

\* Returns 2 DataSets, set[0] is the training set set[1] is the validation set

\*/

public DataSet[] divideForValidation(int validationSetStart, int validationSetEnd){

DataSet[] sets = new DataSet[2];

for(int i = 0; i < sets.length; i++){

sets[i] = new DataSet(name);

sets[i].attributeNames = attributeNames;

}

for(int i = 0; i < records.size(); i++){

if(i >= validationSetStart && i <= validationSetEnd){

sets[1].records.add(records.get(i)); //add to validation set

}

else{

sets[0].records.add(records.get(i)); //add to training set

}

}

return sets;

}

public void generateRandom(int numberRecords){

Random random = new Random();

for(int i = 0; i < numberRecords; i++){

attributeNames = new String[2];

attributeNames[0] = "X";

attributeNames[1] = "Y";

double[] values = new double[2];

values[0] = random.nextInt(13) - 2;

values[1] = (values[0] \* values[0]) + 10;

records.add(new Record(this, i, values));

}

}

public void generateSudoRandom(){

int[] x = {6, 0, 7, 0, 4, 0};

for(int i = 0; i < x.length; i++){

attributeNames = new String[2];

attributeNames[0] = "X";

attributeNames[1] = "Y";

double[] values = new double[2];

values[0] = x[i];

values[1] = (values[0] \* values[0]) + 10;

records.add(new Record(this, i, values));

}

}

public Matrix dataAsMatrix(){

if(records == null) return null;

Matrix matrix = new Matrix(records.size(), records.get(0).getValues().length + 1);

for(int i = 0; i < Matrix.getHeight(matrix); i++){

for(int j = 0; j < Matrix.getWidth(matrix); j++){

if(j == 0) Matrix.setCell(matrix, i, j, 1);

else Matrix.setCell(matrix, i, j, records.get(i).getValues()[j-1]);

}

}

return matrix;

}

public double[][] dataAsArray(){

double[][] matrix = new double[records.size()][records.get(0).getValues().length+1];

for(int i = 0; i < matrix.length; i++){

for(int j = 0; j < matrix[i].length; j++){

if(j == 0) matrix[i][j] = 1;

else matrix[i][j] = records.get(i).getValues()[j-1];

}

}

return matrix;

}

public Matrix classAsMatrix(){

Matrix matrix = new Matrix(records.size(), 1);

int i = 0;

for(Record record : records){

Matrix.setCell(matrix, i, 0, record.getClassification());

i++;

}

return matrix;

}

public double[][] classAsArray(){

double[][] matrix = new double[records.size()][1];

int i = 0;

for(Record record : records){

matrix[i][0] = record.getClassification();

i++;

}

return matrix;

}

public void parseCSV() throws IOException{

String seperator = ",";

BufferedReader bufferedReader = new BufferedReader(dataFile);

attributeNames = bufferedReader.readLine().split(seperator);

int recordNumber = 1;

String[] values;

String line;

while((line = bufferedReader.readLine()) != null){

values = line.split(seperator);

records.add(new Record(this, recordNumber, values));

recordNumber++;

}

bufferedReader.close();

}

@Override

public Iterator<Record> iterator() {

return records.iterator();

}

public String[] getAttributeNames() {

return attributeNames;

}

public List<Record> getRecords() {

return records;

}

public String toString(){

StringBuilder stringBuilder = new StringBuilder();

stringBuilder.append("--" + name + " DATA SET--\n");

for(Record record : records){

stringBuilder.append(record.toString());

}

return stringBuilder.toString();

}

}

**Record.java**

**public** **class** Record {

**private** DataSet parentDataSet;

**private** **int** recordNumber;

**private** Double[] values;

**private** **int** classification;

**public** Record(DataSet parentSet, **int** recordNumber, String[] values){

**this**.parentDataSet = parentSet;

**this**.recordNumber = recordNumber;

**this**.values = **new** Double[values.length-1];

**for**(**int** i = 0; i < values.length-1; i++)

**this**.values[i] = Double.*parseDouble*(values[i]);

classification = Integer.*parseInt*(values[values.length-1]);

}

**public** Record(DataSet parentSet, **int** recordNumber, **double**[] values){

**this**.parentDataSet = parentSet;

**this**.recordNumber = recordNumber;

**this**.values = **new** Double[values.length-1];

**for**(**int** i = 0; i < values.length-1; i++)

**this**.values[i] = values[i];

classification = (**int**)values[values.length-1];

}

**public** Double[] getValues(){

**return** values;

}

**public** **int** getClassification() {

**return** classification;

}

**public** String toString(){

StringBuilder stringBuilder = **new** StringBuilder();

stringBuilder.append("Record #" + recordNumber + " :: Y: " + classification);

stringBuilder.append(" \tValues:");

**for**(**int** i = 0; i < values.length; i++) stringBuilder.append("\t" + parentDataSet.getAttributeNames()[i] + " :: " + values[i] + " ");

stringBuilder.append("\n");

**return** stringBuilder.toString();

}

}

Main.java

**public** **class** Main{

**public** **static** **void** main(String[] args) {

**double**[] lambda = {.1, 1, 10, 100};

**int** numSamples = 12;

LinearRegression linearRegressionWithoutRegularization;

LinearRegression linearRegression;

DataSet trainingData = **new** DataSet("TRAINING");

trainingData.generateRandom(numSamples);

System.***out***.println(trainingData.toString());

/\*

\* Linear Regression without Regularization

\*/

System.***out***.println("Linear Regression without Regularization");

linearRegressionWithoutRegularization = **new** LinearRegression(trainingData);

linearRegressionWithoutRegularization.solve();

System.***out***.println(linearRegressionWithoutRegularization.printEquation());

System.***out***.println("MSE(train) = " + linearRegressionWithoutRegularization.getError() + "\n");

/\*

\* Linear Regression with regularization and 3 fold cross validation

\*/

linearRegression = LinearRegression.*solveWithVFoldCrossValidation*(trainingData, 3, lambda);

System.***out***.println("\nLinear Regression with Regularization. Lambda is " + linearRegression.getLambda());

System.***out***.println(linearRegression.printEquation());

System.***out***.println("MSE(train) = " + linearRegression.getError());

}

}