**Train.txt**

8 2

+ -20 1

+ 47 2

+ -39 1

+ -10 3

- -28 -1

- 0 -2

- 22 -2

- 85 -2

**Test.txt**

6 2

+ 9 1

+ -10 1

+ 0 1

- 10 -1

- -10 -1

- 0 -1

**Console output**

Enter name of training data: irisversitrain.txt

Enter name of test data: irisversitest.txt

Number of epochs: 10000

Learning rate: 0.005

Perceptron weights: 0.3246601013837885 -0.8308007910305479 0.12642916584105449 -0.6115071959753431 0.5264387673885347

Training performance: 0.720000

Testing performance: 0.706667

// Training.java

// This is the main driver for Perceptron training.

// The user will be prompted for training and test data, in

// addition to the number of epochs and learning rate.

**public** **class** Training {

**public** **static** Sample [] loadSamples(Scanner inputStream)

{

**int** nSamples = inputStream.nextInt();

**int** nFeatures = inputStream.nextInt();

Sample [] samples = **new** Sample [nSamples];

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

samples[i] = **new** Sample(nFeatures, inputStream);

}

**return** samples;

}

**public** **static** **void** main(String[] args) **throws** Exception

{

Scanner keyboard = **new** Scanner(System.*in*);

System.*out*.printf("Enter name of training data: ");

String trainFname = keyboard.nextLine();

System.*out*.printf("Enter name of test data: ");

String testFname = keyboard.nextLine();

System.*out*.printf("Number of epochs: ");

**int** nEpochs = keyboard.nextInt();

System.*out*.printf("Learning rate: ");

**double** learningRate = keyboard.nextDouble();

Sample [] trainData = *loadSamples*(**new** Scanner(**new** File(trainFname)));

Sample [] testData = *loadSamples*(**new** Scanner(**new** File(testFname)));

Perceptron perceptron = Perceptron.*train*(learningRate, trainData, nEpochs);

System.*out*.printf("Perceptron weights: %s%n", perceptron.toString());

System.*out*.printf("Training performance: %f%n", perceptron.performance(trainData));

System.*out*.printf("Testing performance: %f%n", perceptron.performance(testData));

}

}

// Sample.java

// Represents a simple data sample. This includes all of the

// numeric attributes and its classification (+ or -).

**import** java.util.Scanner;

**public** **class** Sample {

**public** **double** [] data;

**public** **boolean** isPositive;

**public** Sample(**int** nFeatures, Scanner inputStream)

{

data = **new** **double** [nFeatures + 1];

String c = inputStream.next();

isPositive = c.equals("+");

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

data[i] = inputStream.nextDouble();

}

data[nFeatures] = 1;

}

}

**Perceptron.java**

// Perceptron.java

// This class represents a Perceptron.

**public** **class** Perceptron {

**public** **double** [] weights;

**public** **double** learningRate;

/\*\*

\* Constructs a perceptron given the number of features in the data set

\* and the learning rate.

\* **@param** nFeatures

\* **@param** learningRate

\*/

**public** Perceptron(**int** nFeatures, **double** learningRate)

{

weights = **new** **double** [nFeatures];

**this**.learningRate = learningRate;

}

/\*\*

\* Returns the string representation of the perceptron. This is

\* simply the list of weights.

\* **@return** string representation of perceptron

\*/

**public** String toString()

{

String result = "";

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

result += " " + weights[i];

}

**return** result;

}

/\*\*

\* Computes the dot product between the sample and the percptron

\* weights.

\* **@param** sample sample item to be processed

\* **@return** dot product between weights and feature vector

\*/

**public** **double** processSample(Sample sample)

{

**double** sum = 0;

**int** nFeatures = sample.data.length;

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

sum += weights[i] \* sample.data[i];

}

**return** sum;

}

/\*\*

\* Applies the perceptron to a sample to arrive at a classification.

\* **@param** sample

\* **@return** true if the items is classified as a positive

\*/

**public** **boolean** classify(Sample sample)

{

**return** processSample(sample) > 0;

}

/\*\*

\* Shuffles the order of the samples

\* **@param** nSamples number of samples in training set

\* **@return** arrays of sample numbers, in random order

\*/

**private** **int** [] randomSampleOrder(**int** nSamples)

{

**int** [] sampleIDs = **new** **int**[nSamples];

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

sampleIDs[i] = i;

}

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

**int** tmpSlot = (**int**)(Math.*random*() \* nSamples);

**int** tmp = sampleIDs[i];

sampleIDs[i] = sampleIDs[tmpSlot];

sampleIDs[tmpSlot] = tmp;

}

**return** sampleIDs;

}

/\*\*

\* Applies the percptron learning algorithm once to each item

\* in the training set (in random order).

\* **@param** set array of training samples

\*/

**public** **void** nextEpoch(Sample [] set, **double** [] sampleWeights)

{

**int** nSamples = set.length;

**int** [] randomOrder = randomSampleOrder(nSamples);

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

Sample x = set[ randomOrder[i] ];

**int** nFeatures = x.data.length;

**double** in = processSample(x);

**double** desiredOutput = x.isPositive ? +1 : -1;

**double** err = desiredOutput - in;

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

weights[j] += learningRate \* err \* x.data[j];

}

}

}

/\*\*

\* Applies the perceptron learning algorithm to a set of training data.

\* This overloaded version of train includes sampleWeights, which could

\* be used in a an ensemble learning method such as Ada Boost.

\* **@param** learningRate learning coefficient

\* **@param** set set of training data

\* **@param** sampleWeights weights for each training sample

\* **@param** nEpochs number of epochs to train

\* **@return** trained perceptron

\*/

**public** **static** Perceptron train(**double** learningRate, Sample [] set, **int** nEpochs)

{

**int** nFeatures = set[0].data.length;

Perceptron perceptron = **new** Perceptron(nFeatures, learningRate);

**for** (**int** e=0; e<nEpochs; e++) {

perceptron.nextEpoch(set);

}

**return** perceptron;

}

/\*\*

\* Applies the perceptron to a set of samples and determines the

\* percentage of correct classifications.

\* **@param** set array of samples

\* **@return** percentage of samples correctly classified

\*/

**public** **double** performance(Sample [] set)

{

**int** corr = 0;

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

**double** output = processSample(set[i]);

**boolean** shouldBePositive = set[i].isPositive;

**if** (output > 0 && shouldBePositive) {

corr++;

} **else** **if** (output < 0 && !shouldBePositive) {

corr++;

}

}

**return** corr / (**double**)set.length;

}

}