Machine learning

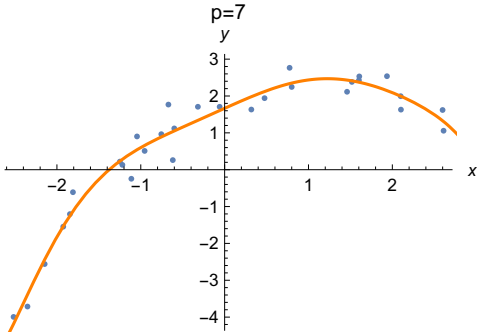
1. Regression

In this problem we'll fit polynomials on a synthetic dataset.

a) The underlying data generating function is $y = f(x) = 2 + x - 0.5\,x^2$. The data will have some noise on top of this function. Start by generating 30 data points of the form
$y_i = f(x_i) + \epsilon_i$,
where x_i are sampled from an uniform distribution on the interval $[-3, 3]$ and ϵ_i are sampled from a normal distribution with mean $\mu = 0$ and standard deviation $\sigma = 0.4$. You should end up with 30 pairs of numbers (x_i, y_i). Plot your dataset using ListPlot to see that everything looks ok. Hint: look up RandomVariate, UniformDistribution and NormalDistribution.

b) Define a function that fits your data using polynomials up to a order given as a parameter, that is, define something like: getFit[order_] := ... . Your function should return a polynomial of order 'order' in x with fitted coefficients. Hint: check the documentation for the built-in function Fit.

c) Make a plot where you show the data as points and your polynomial fit as a solid curve. You should obtain something like this (for order 7)



Generate a list of this kind of plots for polynomial orders from 0 to 10. Which fit is ' best'?

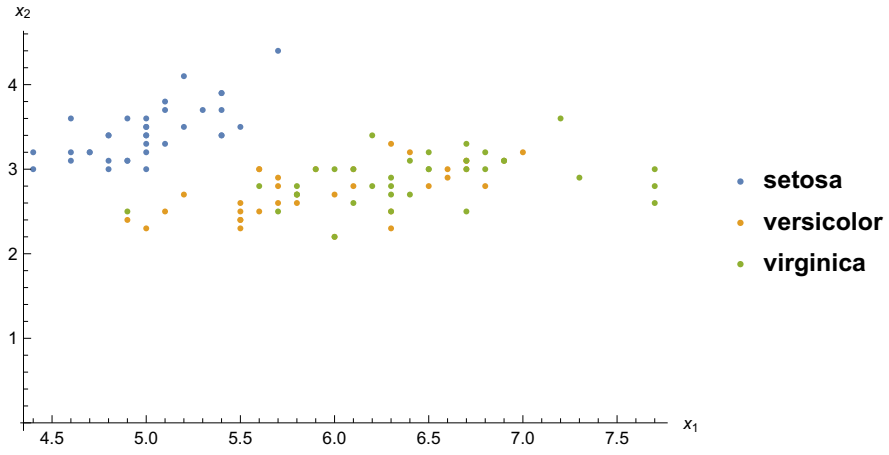
2. Flower recognition

Fisher's irises is a classic dataset for supervised learning. It contains measurements of three similar looking flower species (Iris

Setosa/Versicolor/Virginica). Each flower in the dataset has sepal length, sepal width, petal length and petal width measurements. That is, each data point is a set of four real numbers and a label. Load the data using the following commands

In[•]:=
```
resource = ResourceObject["Sample Data: Fisher's Irises"];
trainingData = ResourceData[resource, "TrainingData"];
testData = ResourceData[resource, "TestData"];
```

a) The data is four dimensional. Select a subset of two features and produce a scatterplot where different species are marked with different colors. You should obtain something like



Repeat for all six possible pairs of features. You'll notice that some features are more useful than others in telling different species apart.

b) Build a nearest neighbor classifier from scratch. That is, define a function that takes four numbers and outputs a class prediction. The idea is to search the closest point in the training dataset using Euclidian distance in $\mathbb{R}^4$ and output the class of that closest point as a prediction for the input point. Hint: look up documentation for TakeSmallestBy

c) Measure how many of the test examples you can classify correctly with your simple classifier

d) Use Classify on training data without specifying a method. This causes Classify to automatically search for an optimal classification algorithm for your data. What is the accuracy of this classifier?

3. Digit classification with Classify

a) A classic exercise in machine learning courses is the classification of hand

written digits. Let's use built-in functions to solve the problem in Mathematica. First, start by downloading the dataset from the online repository. Use the following commands

In[*]:=
```
resource = ResourceObject["MNIST"];
trainingData = RandomSample[ResourceData[resource, "TrainingData"], 5000];
testData = RandomSample[ResourceData[resource, "TestData"], 1000];
```

This downloads the 'MNIST'-dataset and takes 5000 and 1000 examples for training and validation (complete training dataset size is 60000, you can use all training samples if you have a fast computer and/or don't mind waiting). Print a few digits to see the data was loaded correctly

b) Use the built-in function 'Classify' on training data to construct a ClassifierFunction. Use the method "LogisticRegression" for classification. This performs a variant of linear regression on the training data.

c) Use the classifier for classifying digits in the test dataset. Evaluate the classifier on all test examples and manually calculate the accuracy of your classifier (the fraction of correctly classified test examples and total number of test examples).

d) Use the ClassifierMeasurements-function to find the accuracy of your classifier and check that it agrees with the number you obtained above.

e) Use the ClassifierMeasurementsObject from d-part to generate the confusion matrix plot for your classifier. Which digits are the most difficult to classify correctly? Hint: you can use cm["Properties"] (where 'cm' is the ClassifierMeasurementsObject) to get a complete list of available properties.

4. Digit classification with a neural network

Let's try to outperform the logistic regression classifier of the previous problem by training a neural network for the digit classification task.

a) Start by defining the following fully connected neural network

```
fcUninitialized =
 NetChain[{LinearLayer[500], Ramp, LinearLayer[784], Ramp, LinearLayer[10],
   SoftmaxLayer[]}, "Input" → NetEncoder[{"Image", {28, 28}, "Grayscale"}],
  "Output" → NetDecoder[{"Class", Range[0, 9]}]]
```

This corresponds to the following network

You'll notice that at this point the neural network is uninitialized. This means that the weights and biases within each of the linear layers remain undefined. Use the NetInitialize-function to initialize this network.

b) At this point the network can map images of digits to numbers. Since the parameters of the network are initialized to random values the output of the network is not very impressive. Evaluate the initialized network on a few training digits and verify that it produces a number in range [0, 9].

c) Train the network on the training set using the NetTrain-function. Give a few test examples to the trained network to see that now it performs much better.

d) Use ClassifierMeasurements to find the accuracy of your network. Did it perform better than logistic regression?

e) Modify the network by adding/removing layers and changing sizes of linear layers. Can you come up with a network that performs better than the original one?

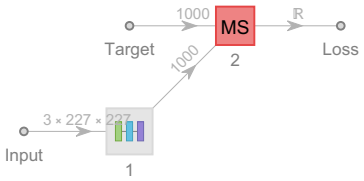
5. Adversarial attack on a convolutional neural network (CNN) (This is a harder problem, some previous knowledge on neural networks is recommended)

In this problem we'll write an algorithm for constructing adversarial examples for a CNN trained for image classification. The key observation is that the space of possible inputs in image classification is extremely large and it turns out that it is very difficult to train a neural network that behaves well for all possible inputs. We will exploit this property to construct noise we add to an image of our choosing in such a way that the image is classified incorrectly by the CNN. That is, we can take a picture of a panda and make the network think it is a gibbon simply by adding imperceptible, carefully constructed noise on top of the panda.

a) Download SqueezeNet from the Wolfram Neural Net Repository by using the NetModel-function. This network is pretrained on ImageNet competition data to yield good performance with a relatively small network. Also

download an image from the internet (e.g. picture of a cat) using the Import-function. This will be the image we use to fool the network. Use ImageResize to resize the image to 227x227 (input size of SqueezeNet), in this way we avoid some easy pitfalls later on.

b) Construct a training network using NetGraph.



The idea is to feed the output of SqueezeNet (class probabilities) to a MeanSquaredLossLayer(MS) together with a unit vector (indicating our desired class) to produce a real number, loss, measuring how far off we are from classifying the given image as our preferred class. See documentation for NetGraph and NetPort. You'll have to tell NetGraph to connect the "Output"-port of SqueezeNet to the "Input"-port of the MS-layer and also connect a "Target"-NetPort to the "Target"-port of the MS-layer.

c) Check that your network works. Give it the image you downloaded and a vector indicating a class to see it outputs a real number (loss). The vector should be an unit vector in $\mathbb{R}^{1000}$ because SqueezeNet has 1000 output classes.

d) Next we wish to use backpropagation to calculate gradients of our input image. The idea is to nudge our input image slightly in the direction of negative gradient until it is classified the way we want. See documentation for NetPortGradient and calculate the gradient of the "Input"-port of the training network. Shift your image in the direction of the negative gradient (it is a good idea to normalize the gradient somehow) and confirm that this modified image has a lower loss than the original image.

e) Lastly, put it all together in a function that repeatedly calculates the gradient of the input image and modifies it towards the negative gradient until either a maximum number of iterations has passed or the image is classified in the (incorrect) class of your choosing. This should not take too many iterations (less than 100) and the adversarial image should still look indistinguishable from the original image. You'll have to make some design choices like step sizes and such the way you think is best.