Tutorial 7: Steepest Gradient Descent

Tutorial objectives:

- **Revise and solidify understanding of Steepest Gradient Descent (SGD) algorithm (in matrix form)**
- **Implement the Least Squares (LS) method for polynomial regressing using the Numpy library**
- **Implement the SGD algorithm for polynomial regressing using the Numpy library**
- **Implement the SGD algorithm for polynomial classification (logistic regression) using the Numpy library**
- **Reinforce understanding of optimisation concepts and SGD-based parameter search that fit the regression and classification models to training data**
- **Observe how complicated mathematical rules of the SGD boil down to simple code that works over a matrix library (such as Numpy)**

Optimisation concepts

1. What's the difference between LS and SGD?

2. What's the difference between linear regression and logistic regression?

3. What is the cross-entropy loss?

4. Would logistic regression over a polynomial hypothesis be capable of solving a non-linear separable classification problem ?

Linear regression and LS

Let's go over matrix-based calculations for training and prediction of a linear regression model using a polynomial hypothesis. Recall that polynomial of degree k is a weighted sum of combinations of multiplied input attributes with k terms or less (including the option to multiply an attribute by itself k times). And so, for input consisting of D attributes, the output of the polynomial of degree 1 would be:

$$\hat{y} = w_1 x_1 + w_2 x_2 + \ldots + w_D x_D + w_{D+1},$$

where w_i is the weight multiplying input x_i, and w_{D+1} is the bias weight.A shorter notation for expressing this sum is

$$\hat{y} = \sum_{i=1}^{D} w_i x_i + w_{D+1},$$

Suppose we had N points, and indexed attribute d of the n^{th} point as x_{nd}, then the output for the n^{th} point would be:

$$\hat{y}_n = \sum_{i=1}^{D} w_i x_{ni} + w_{D+1},$$

In the matrix form we can express the entire calculation as:

$$\begin{bmatrix} x_{11} & \ldots & x_{1D} & 1 \\ \vdots & \ddots & \vdots & \vdots \\ x_{N1} & \ldots & x_{ND} & 1 \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_D \\ w_{D+1} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{D} w_i x_{1i} + w_{D+1} \\ \vdots \\ \sum_{i=1}^{D} w_i x_{Ni} + w_{D+1} \end{bmatrix} = \begin{bmatrix} \hat{y}_1 \\ \vdots \\ \hat{y}_N \end{bmatrix}$$

Now let's consider a polynomial of degree 2 on an input of D attributes. In addition to the weighted sum of the terms from the polynomial of degree 1 we could also have any (or all) of the weighted sum of all unique combinations of $x_i x_j$ for $i = 1, \ldots, D$ and $j = 1, \ldots, D$. We could express this as the following sum:

$$\hat{y} = \sum_{i=1}^{D} \sum_{j \geq i}^{D} w_{ij} x_i x_j + \sum_{i=1}^{D} w_i x_i + w_{D+1},$$

This is a bit hard to imagine in abstract, so let's consider a specific case. Suppose we have a problem with input of $D = 3$ attributes. The full polynomial of second degree on that input would be:

$$\hat{y} = w_{11} x_1^2 + w_{12} x_1 x_2 + w_{13} x_1 x_3 + w_{22} x_2^2 + w_{23} x_2 x_3 + w_{33} x_3^2 + w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4.$$

Note that this scheme the indexing of weights is not consecutive as there is no w_{10} nor w_{20} nor w_{21} nor w_9 – all that matters is that these weights are different from each other

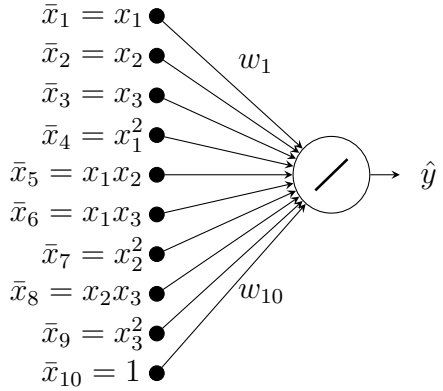
in the notation. Now, again, suppose we had N points, and attribute d of the n^{th} point was denoted as x_{nd}. The N outputs according to the equation above, for the polynomial of degree 2 and input consisting of 3 attributes, could be conveyed through the following matrix computation:

$$\begin{bmatrix} (x_{11})^2 & x_{11}x_{12} & x_{11}x_{13} & (x_{12})^2 & x_{12}x_{13} & (x_{13})^2 & x_{11} & x_{12} & x_{13} & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ (x_{N1})^2 & x_{N1}x_{N2} & x_{N1}x_{N3} & (x_{N2})^2 & x_{N2}x_{N3} & x(_{N3})^2 & x_{N1} & x_{N2} & x_{N3} & 1 \end{bmatrix} \begin{bmatrix} w_{11} \\ w_{12} \\ w_{13} \\ w_{22} \\ w_{23} \\ w_{33} \\ w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} = \begin{bmatrix} \hat{y}_1 \\ \vdots \\ \hat{y}_N \end{bmatrix}.$$

Why bother with all these matrices? Well, because they lead to a very simple expression of a polynomial hypothesis as

$$\hat{\mathbf{y}} = \bar{X} \cdot \mathbf{w}, \tag{1}$$

where $\bar{X}$ is a $N \times \bar{D}$ matrix of various combinations of multiplication of D attributes of the input concatenated with a column of all 1's, $\mathbf{w}$ is the vector of $\bar{D}$ weights, and $\hat{\mathbf{y}}$ is a vector of N output values. Does that look familiar? It should. It should seem somewhat similar to the computation of the output of the Perceptron model from last week. The only difference here is that the effect of the bias weight, represented as w_{D+1}, is incorporated into the feature matrix (as column of 1's) and the weight vector (as its last weight), input is not X, but its feature representation as $\bar{X}$, and there is no hard-limit function. And since we talk about single output value, the output and weights are not matrices, but vectors... but you can think of them as single column matrices. The point is that you can think of a linear regressor as a perceptron-line model, which for our example of a 3-attribute input and the feature space of polynomial of degree 2, would be something like this:



where $\bar{x}_i$ represents the i^{th} feature. The diagonal line inside the circle in the diagram above represents a linear activation function (recall that in perceptron, the weighted sum was converted to either 0 or 1 through the hard-limiting activation function, but here, the output is just the weighted sum of input features). Note that the actual input samples for the above model have only 3 attributes: x_1, x_2, and x_3. The 10 inputs of the polynomial feature space are made up of different multiples of x_1, x_2 and x_3 (and just value of 1 for the bias weight). This transformation of the input into extra attributes that consist of various functions of the input is the process of casting the input into a feature space. In the feature space, we are essentially solving the following linear regression problem:

n	$\bar{\mathbf{x}}_n$										y_n
	$(x_{n1})^2$	$x_{n1}x_{n2}$	$xn1x_{n3}$	$(x_{n2})^2$	$x_{n2}x_{n3}$	$(x_{n3})^2$	x_{n1}	x_{n2}	x_{n3}	1	
1	.	.	.	.	.	.	.	.	.	.	.
2	.	.	.	.	.	.	.	.	.	.	.
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
N	.	.	.	.	.	.	.	.	.	.	.

Recall from Lectures 13 and 14, that polynomial regressor is a linear model (linear in parameters). Given N data samples with corresponding N outputs in a vector $\mathbf{y}$, we can forming the feature matrix $\bar{X}$ according to the degree of desired polynomial and solve the Least Squares (LS) equation

$$\mathbf{w} = (\bar{X}^T \cdot \bar{X})^{-1} \cdot \bar{X}^T \cdot \mathbf{y}. \tag{2}$$

to obtain the weight vector $\mathbf{w}$ that minimises the Mean Squared Error (MSE) between $\mathbf{y}$ and $\hat{\mathbf{y}}$ defined in Equation 1.

The LS analytical solution is not guaranteed to always work well ($\bar{X}$ might not be invertible, or the chosen degree of the polynomial might not might be powerful to fit the data), but when it does, it's a very quick way to fit a function to the data.

Exercise 1: For this exercise you will be guided through an implementation of a simple program for linear regression fit of a polynomial hypothesis using the Least Squares method. Create a new project in PyCharm (select Anaconda's cosc343 environment for the interpreter). Download `PolynomialRergressor.py` and `helper.py` from Blackboard into your project folder. Create a new script and type in the following code:

```
from helper import *
import matplotlib.pyplot as plt

X, y = load_reg1dataset()
```

```
plt.scatter(X,y,c='b')
plt.show()
```

Run the is script. It loads a data sample as a 150×1 numpy array `X` (150 points with a single attribute) and the corresponding outputs in an 150×1 numpy array `y`. The call to the scatter function plots `X` against `y` as blue points (`c='b'` stands for colour the points blue).

Now to solve for the LS equation, you need to transform input X into a feature matrix $\bar{X}$ that is different for different degree of the polynomial. While conceptually this pretty straight forward, programmatically it's a bit of a hassle...so the code for this transformation is provided for you in `PolynomialRegressor.py`. At the top of you script add the following import

```
from PolynomialRegressor import PolynomialRegressor
```

and below, right after the scatter call (but before `plt.show()`) add the following code:

```
k = 1
Xb = PolynomialRegressor.input_to_poly_features(X,degree=k)
```

`Xb` (which is the variable name for $\bar{X}$) is a matrix of inputs in the feature space. For a polynomial of degree $k = 1$, for this dataset, it would be just `X` with an extra column of all 1's. For a polynomial of degree $k = 2$, it would be a 3×1 matrix with a column for x, x^2, and all 1's. For now, stick with polynomial of degree 1.

To solve for $\mathbf{w}$ you can use Equation 2, which in Numpy is:

```
w=np.dot(np.dot(np.linalg.inv(np.dot(Xb.T,Xb)),Xb.T),y)
```

To break this expression down:
`np.dot(Xb.T,Xb)` is equivalent to $\bar{X}^T \cdot \bar{X}$,
`np.linalg.inv(np.dot(Xb.T,Xb))` is equivalent to $(\bar{X}^T \cdot \bar{X})^{-1}$,
`np.dot(np.linalg.inv(np.dot(Xb.T,Xb)),Xb.T)` is $(\bar{X}^T \cdot \bar{X})^{-1} \cdot \bar{X}^T$, and
`np.dot(np.dot(np.linalg.inv(np.dot(Xb.T,Xb)),Xb.T),y)` is $(\bar{X}^T \cdot \bar{X})^{-1} \cdot \bar{X}^T \cdot \mathbf{y}$.

The length of the vector $\mathbf{w}$ simply falls out out of the rules of matrix multiplication ($\bar{X}^T \cdot \bar{X}$ gives a $\bar{D} \times \bar{D}$ matrix, inverting it doesn't change the size, dot producing with $\bar{X}^T$ gives a $\bar{D} \times N$ matrix and dot producting that with $\mathbf{y}$ results in a $\bar{D} \times 1$ column vector). Thus, the length of $\mathbf{w}$ ends up being the same as the number of columns in `Xb`.

To check the fit, you need to display the function over a range of data. Here's the code to create a set of 200 equidistant test samples spanning the range from the minimum to maximum value of `X`:

```
xmin = np.min(X)
xmax = np.max(X)
xtest = np.linspace(xmin,xmax,200)
Xtest = np.expand_dims(xtest,axis=1)
```

Next, you need to transform `Xtest` into the feature space of your polynomial

```
Xbtest = Polynomial.input_to_poly_features(Xtest,degree=k)
```

where k is the same degree as what you used for transformation of `X` into `Xb`. Next, compute the output of the polynomial hypothesis for the found weights `w` like so:

```
ytest = np.dot(Xbtest,w)
```

Finally, plot the output of the test data on the same scatter figure (that has blue training points) as red line visualising the resulting polynomial function:

```
plt.plot(xtest,ytest,c='r')
```

Oh, and make sure all this code goes before `plt.show()` (in other words, `plt.show()` should be at the end of you script).

You should see a plot with a red line going across the blue points. It won't be a great fit, but the best a linear polynomial can do for this data. Keep increasing the polynomial degree and checking the LS fit until you're happy that with its consistency over the training data.

Linear regression and SGD

Steepest Gradient Descent (SGD) is an optimisation method that uses derivatives of the loss function to update the parameters of the model iteratively. It is not as instantaneous as Least Squares, but it is more general, as it will work for non-linear as well as linear models, as well as for arbitrary loss function (as long as that function is differentiable). But for now, we'll stick with regression and the MSE loss.

Recall, that for regression the loss function we want to minimise is the mean squared error:

$$J_{\text{MSE}} = \frac{1}{N}\sum_{n=1}^{N}(y_n - \hat{y}_n)^2,$$

where $\hat{y}_n$ is the output of the model for the n^{th} sample in the training data, and y_n is the corresponding target output. In matrix form, this is the same as:

$$J_{\text{MSE}} = \frac{1}{N}(\mathbf{y} - \hat{\mathbf{y}}) \cdot (\mathbf{y} - \hat{\mathbf{y}})^T,$$

where $\mathbf{y}$ and $\hat{\mathbf{y}}$ are N-dimensional vectors, the latter being the result of computation shown in Equation 1.

The negative gradient of the J_{MSE} above with respect to weight vector $\mathbf{w}$ works out to be:

$$-\frac{\partial J_{\text{MSE}}}{\partial \mathbf{w}} = \frac{1}{2}\Big(\bar{X} \cdot (\mathbf{y} - \hat{\mathbf{y}})\Big)$$

Thus, the following expressions provide the logic of the SGD updates that reduce the MSE loss:

$$\Delta\mathbf{w} = \frac{1}{2}\Big(\bar{X}^T \cdot (\mathbf{y} - \hat{\mathbf{y}})\Big) \tag{3}$$

$$\mathbf{w} := \mathbf{w} + \frac{\alpha}{N}\Delta\mathbf{w}, \tag{4}$$

where $\alpha > 0$ is the learning parameter that is related to the size of the step taken along the loss surface in the direction of the negative gradient. Note the similarity of the above equations to the perceptron learning rule you implemented in the previous tutorial.

Exercise 2: For this exercise you need to implement the SGD updates in `PolynomialRegressor.py`. The script already contains the `PolynomialRegressor` class that follows the Sklearn library format. It contains all the code for the setup of variables, transformation of X into feature space $\bar{X}$ of the requested polynomial, and inferring the size and random initialisation of the weight vector. But it is missing the most key aspect – the SGD update rules, which are the key to the whole training of the model. Just like in the last tutorial, your task is to add code to the `fit(X,y)` method that implements the appropriate training rules. In this case, they should be based on the SGD algorithm with the updates according to the equations as given by Equations 1, 3, and 4. Your code should be placed within the scope of the loop

```
for i in range(self.max_iter):
```

in the `fit(X,y)` method of the `PolynomialRegressor` class. In that scope you will have available: `Xb`, an $N \times D$ Numpy array of inputs in feature space; `y`, an $N \times 1$ Numpy array of target output values; `self.w`, an $N \times 1$ Numpy array of weights (already initialised to random values if running `fit(X,y)` for the first time); `N`, the number of points, and `self.learning_parameter`, which is set in the class initialiser. For the logic that follows, you have to make sure that the output of the model, in each iteration, is saved to variable `yhat`.

Once you've done with your implementation test it. Create another script and type in the following code:

```
from helper import *
from PolynomialRegressor import PolynomialRegressor
import matplotlib.pyplot as plt

X, y = load_reg1dataset()

model = PolynomialRegressor(degree=1, learning_rate=0.1, max_iter=10)

for epoch in range(100):
    model.fit(X,y)
    model.plot_poly_function(X,y,
          titleStr="Epoch %d" % ( (epoch+1)*model.max_iter))

plt.ioff()
plt.show()
```

The above code loads the same dataset as you used in the previous exercise. Then it instantiates an object of `PolynomialRegressor` type, which sets up a polynomial model of specified degree to be trained using the provided learning rate (which is saved to `self.learning_parameter` inside the object) `max_ter` iterations at a time. The loop that follows invokes the `fit` method 100 times (inside which SGD will perform its updates over `max_ter` epochs) and plots the results using a method `plot_function` of the `PolynomialRegressor` that is already provided for you. This visualisation recreates the same plot you made in the previous exercise. However, since you want this plot to be updated, you need to disable the "Show plots in toolwindow" option in PyCharm→Preferences→Tools→Python Scientific. From the previous exercise you should a pretty good idea what to set the degree of your polynomial to to have a chance of a decent fit of the data. You might want to adjust the `learning_rate` to speed up or slow down the updates, and the `max_iter` value to skip more or fewer iteration before each visualisation update.

Are you noticing the slowing down of the updates as the algorithm progresses? Why do you think this happens?

Logistic regression and SGD

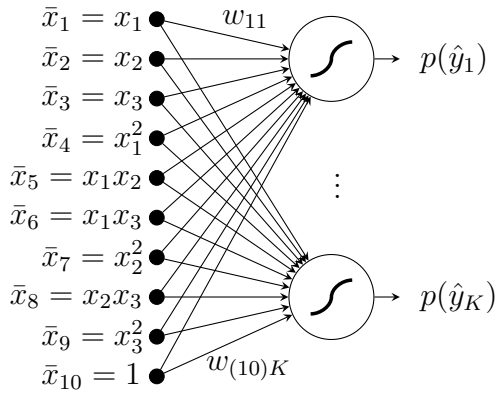
For classification, the output of the polynomial model needs to be passed through a sigmoid function:

$$\hat{Y}_p = \sigma_{\mathrm{s}}(\bar{X} \cdot W), \tag{5}$$

where $\sigma_{\mathrm{s}}(v) = \frac{1}{1+e^{-v}}$ is applied to every element of the result of $\bar{X} \cdot W$. The dot product inside the sigmoid function is very much the same as in Equation 1, except the weight vector is replaced with a $\bar{D} \times K$ matrix allowing K outputs (to support multi-class classification). Each row of matrix $\hat{Y}$ will now contains K values between 0 and 1, which we interpret as probabilities of that output being on (and thus labelling input as being (or not being) a member of the corresponding class. If it helps, the entire computation written out in matrix is as follows:

$$\begin{aligned}
\begin{bmatrix} p(\hat{y}_{11}) & \dots & p(\hat{y}_{1K}) \\ \vdots & \ddots & \vdots \\ p(\hat{y}_{1N}) & \dots & p(\hat{y}_{NK}) \end{bmatrix} &= \begin{bmatrix} \sigma_{\mathrm{s}}\Big(\sum_{i=1}^{\bar{D}-1} \bar{x}_{1i} w_{i1} + w_{\bar{D}1}\Big) & \dots & \sigma_{\mathrm{s}}\Big(\sum_{i=1}^{\bar{D}-1} \bar{x}_{1i} w_{iK} + w_{\bar{D}K}\Big) \\ \vdots & \ddots & \vdots \\ \sigma_{\mathrm{s}}\Big(\sum_{i=1}^{\bar{D}-1} \bar{x}_{Ni} w_{i1} + w_{\bar{D}1}\Big) & \dots & \sigma_{\mathrm{s}}\Big(\sum_{i=1}^{\bar{D}-1} \bar{x}_{Ni} w_{iK} + w_{\bar{D}K}\Big) \end{bmatrix} \\
&= \sigma_{\mathrm{s}}\left(\begin{bmatrix} \bar{x}_{11} & \dots & \bar{x}_{1\bar{D}-1} & 1 \\ \vdots & \ddots & \vdots & \vdots \\ \bar{x}_{N1} & \dots & \bar{x}_{N\bar{D}-1} & 1 \end{bmatrix} \begin{bmatrix} w_{11} & \dots & w_{1K} \\ \vdots & \ddots & \vdots \\ w_{\bar{D}1} & \dots & w_{\bar{D}K} \end{bmatrix} \right),
\end{aligned}$$

where $\hat{x}_{ni}$ is the element in row n and column i of feature matrix $\bar{X}$ and w_{ij} is the weight value connecting feature i to output j. As a perceptron-like diagram, for the case of a polynomial of degree 2 over a 3-input attribute, this model would be represented by the following figure:



where $\bar{x}_i$ is the i^{th} feature. The S-shaped line inside the circles indicates the sigmoid activation function.

Recall, that for logistic regression we want to minimise the cross-entropy loss, which for single output model would be:

$$J_{\text{CE}} = -\frac{1}{N}\sum_{n=1}^{N}\Big(y_n \ln p(\hat{y}_n) + (1-y_n)\ln\Big(1-p(\hat{y}_n)\Big)\Big),$$

Extending this to K outputs, this loss becomes:

$$J_{\text{CE}} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\Big(y_{nK} \ln p(\hat{y}_{nK}) + (1-y_{nK})\ln\Big(1-p(\hat{y}_{nK})\Big)\Big).$$

The computation of the negative gradient of J_{CE} with respect to weight matrix W is quite complicated, involves derivatives of the sigmoid function and logs... Amazingly (well, it's not a coincidence, but rather the choice of using cross-entropy is because of that reason), after a lot of computation many terms that have logs and exponents cancel out. In the matrix format, the negative gradient boils down to the following expression:

$$-\frac{\partial J_{\text{CE}}}{\partial W} = \bar{X}^T \cdot \Big(Y \odot (1-\hat{Y}_p) - (1-Y)\odot \hat{Y}_p\Big),$$

where $\cdot$ denotes the dot product, and $\odot$ denotes element-wise product. Specifically:

$$\begin{aligned} Y \odot (1-\hat{Y}p) &= \begin{bmatrix} y_{11} & \dots & y_{1K} \\ \vdots & \ddots & \vdots \\ y_{N1} & \dots & y_{NK} \end{bmatrix} \odot \begin{bmatrix} 1-p(\hat{y}_{11}) & \dots & 1-p(\hat{y}_{1K}) \\ \vdots & \ddots & \vdots \\ 1-p(\hat{y}_{N1}) & \dots & 1-p(\hat{y}_{NK}) \end{bmatrix} \\ &= \begin{bmatrix} y_{11}(1-p(\hat{y}_{11})) & \dots & y_{1K}(1-p(\hat{y}_{1K})) \\ \vdots & \ddots & \vdots \\ y_{N1}(1-p(\hat{y}_{N1})) & \dots & y_{N1}(1-p(\hat{y}_{NK})) \end{bmatrix} \end{aligned}$$

Thus, the following expressions give the SGD update of the model parameters:

$$\Delta W = \bar{X}^T \cdot \Big(Y(1-\hat{Y}_p) - (1-Y)\hat{Y}_p\Big) \tag{6}$$

$$W := W + \frac{\alpha}{N}\Delta W, \tag{7}$$

where α is the learning parameter and $\hat{Y}$ is computed as shown in Equation 5

Exercise 3: For this exercise you need to implement the SGD updates for `PolynomialClassifier.py` (which you can download from Blackboard). The script already contains the `PolynomialClassifier` class that follows the Sklearn library format. The code inside sets up the polynomial model, implements the logic of the `predict(X)` method, and inside `fit(X,y)` infers K from `y`, transforms `X` into `Xb`, transforms `{y}` into one-hot encoding matrix `Y` and initialises the weight matrix appropriately. All you need to do is to augment `fit(X,y)` method with he code implementing the training algorithm based on the SGD updates as given by Equations 5, 6, and 7. Once again, your goes into the `fit(X,y)` method within the scope of the loop

```
for i in range(self.max_iter):
```

Within the scope of that loop you have available: `Xb`, an $N \times D$ Numpy array of inputs in feature space; `Y`, an $N \times K$ Numpy array of target class labels in one-hot encoding format; `self.W`, an $N \times K$ Numpy array of weights (already initialised to random values if running `fit(X,y)` for the first time); `N`, the number of points, and `self.learning_parameter`, which is set in the class initialiser. To apply sigmoid function over all elements of a matrix you are provided with a static method `PolynomialClassifier.sigmoid(V)`, which expects V to be a $N \times K$ Numpy array. For the logic that follows your code inside the `fit(X,y)` to work, the output of your model in each epoch should be saved to `Yphat`, an $N \times K$ Numpy array.

Hints!

- Subtracting a Numpy array from a scalar value (like 1) can be done with the standard subtraction, '-', operator – Numpy, recognising that one of the operands is a scalar, will do the operation elementwise;
- elementwise multiplication of two Numpy arrays of the same size can be done with the standard multiplication, '*', operator.

Once you've done with your implementation of SGD, test it. Create another script and type in the following code:

```
from helper import *
from PolynomialClassifier import PolynomialClassifier
import matplotlib.pyplot as plt

X, y = load_xor()

model = PolynomialClassifier(degree=1, learning_rate=0.1, max_iter=10)

for epoch in range(100):
```

```
    model.fit(X,y)
    model.plot_classified_regions(X,y,
          titleStr="Epoch %d" % ( (epoch+1)*model.max_iter))

plt.ioff()
plt.show()
```

The code above loads the XOR classification dataset. Then it instantiates an object of `PolynomialClassifier` type, which creates a polynomial logistic regression model of specified degree to be trained using the given learning rate (which is saved to `self.learning_parameter` inside the object) `max_ter` iterations at a time. The loop that follows invokes the `fit` method 100 times (inside which SGD will perform its updates over `max_ter` epochs) and plots the results using a method `plot_classified_regions` of the `PolynomialClassifier` that is already provided for you (similar to the visualisation provided in the `Perceptron.py` you worked with in the last tutorial). If you paid attention in the lectures, you'll know that polynomial of degree 1 is not capable of solving the XOR classification problem. Try to increase the degree to see if it works. You might want to adjust the `learning_rate` to speed up or slow down the updates, and the `max_iter` value to skip more or fewer iteration before each visualisation update.

See how well the classification works the 2D version of the iris dataset. Simply replace loading of the xor data with

```
X, y = load_iris2D()
```

and (possibly) adjust the degree of the polynomial.

If you get all of that working, try the spiral problem

```
X, y = load_spiral()
```

Are you patient enough to wait for the polynomial to fit it well?