12 Gradient Descent Methods

Lab Objective: *Iterative optimization methods choose a search direction and a step size at each iteration. One simple choice for the search direction is the negative gradient, resulting in the method of steepest descent. While theoretically foundational, in practice this method is often slow to converge. An alternative method, the conjugate gradient algorithm, uses a similar idea that results in much faster convergence in some situations. In this lab we implement a method of steepest descent and two conjugate gradient methods, then apply them to regression problems.*

The Method of Steepest Descent

Let $f : \mathbb{R}^n \to \mathbb{R}$ with first derivative $Df : \mathbb{R}^n \to \mathbb{R}^n$. The following iterative technique is a common template for methods that aim to compute a local minimizer $\mathbf{x}^*$ of f.

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \tag{12.1}$$

Here $\mathbf{x}_k$ is the kth approximation to $\mathbf{x}^*$, α_k is the *step size*, and $\mathbf{p}_k$ is the *search direction*. Newton's method and its relatives follow this pattern, but they require the calculation (or approximation) of the inverse Hessian matrix $Df^2(\mathbf{x}_k)^{-1}$ at each step. The following idea is a simpler and less computationally intensive approach than Newton and quasi-Newton methods.

The derivative $Df(\mathbf{x})^\mathsf{T}$ (often called the *gradient* of f at $\mathbf{x}$, sometimes notated $\nabla f(\mathbf{x})$) is a vector that points in the direction of greatest **increase** of f at $\mathbf{x}$. It follows that the negative derivative $-Df(\mathbf{x})^\mathsf{T}$ points in the direction of steepest **decrease** at $\mathbf{x}$. The *method of steepest descent* chooses the search direction $\mathbf{p}_k = -Df(\mathbf{x}_k)^\mathsf{T}$ at each step of (12.1), resulting in the following algorithm.

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k Df(\mathbf{x}_k)^\mathsf{T} \tag{12.2}$$

Setting $\alpha_k = 1$ for each k is often sufficient for Newton and quasi-Newton methods. However, a constant choice for the step size in (12.2) can result in oscillating approximations or even cause the sequence $(\mathbf{x}_k)_{k=1}^\infty$ to travel away from the minimizer $\mathbf{x}^*$. To avoid this problem, the step size α_k can be chosen in a few ways.

- Start with $\alpha_k = 1$, then set $\alpha_k = \frac{\alpha_k}{2}$ until $f(\mathbf{x}_k - \alpha_k Df(\mathbf{x}_k)^\mathsf{T}) < f(\mathbf{x}_k)$, terminating the iteration if α_k gets too small. This guarantees that the method actually descends at each step and that α_k satisfies the Armijo rule, without endangering convergence.

- At each step, solve the following one-dimensional optimization problem.

$$\alpha_k = \underset{\alpha}{\operatorname{argmin}}\ f(\mathbf{x}_k - \alpha Df(\mathbf{x}_k)^\mathsf{T})$$

  Using this choice is called *exact steepest descent*. This option is more expensive per iteration than the above strategy, but it results in fewer iterations before convergence.

Problem 1. Write a function that accepts an objective function $f : \mathbb{R}^n \to \mathbb{R}$, its derivative $Df : \mathbb{R}^n \to \mathbb{R}^n$, an initial guess $\mathbf{x}_0 \in \mathbb{R}^n$, a convergence tolerance `tol` defaulting to $1e^{-5}$, and a maximum number of iterations `maxiter` defaulting to 100. Implement the exact method of steepest descent, using a one-dimensional optimization method to choose the step size (use `opt.minimize_scalar()` or your own 1-D minimizer). Iterate until $\|Df(\mathbf{x}_k)\|_\infty <$ `tol` or $k >$ `maxiter`. Return the approximate minimizer $\mathbf{x}^*$, whether or not the algorithm converged (`True` or `False`), and the number of iterations computed.

Test your function on $f(x, y, z) = x^4 + y^4 + z^4$ (easy) and the Rosenbrock function (hard). It should take many iterations to minimize the Rosenbrock function, but it should converge eventually with a large enough choice of `maxiter`.

The Conjugate Gradient Method

Unfortunately, the method of steepest descent can be very inefficient for certain problems. Depending on the nature of the objective function, the sequence of points can zig-zag back and forth or get stuck on flat areas without making significant progress toward the true minimizer.

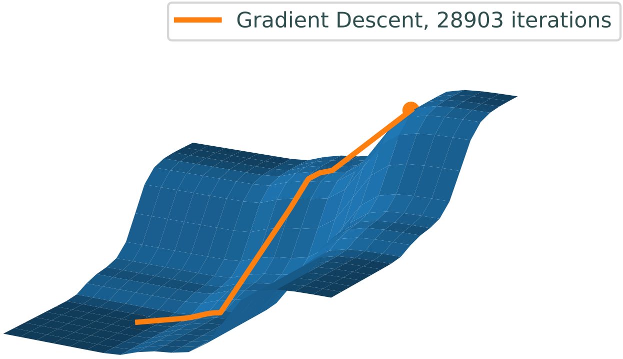


Figure 12.1: On this surface, gradient descent takes an extreme number of iterations to converge to the minimum because it gets stuck in the flat basins of the surface.

Unlike the method of steepest descent, the *conjugate gradient algorithm* chooses a search direction that is guaranteed to be a descent direction, though not the direction of greatest descent. These directions are using a generalized form of orthogonality called *conjugacy*.

Let Q be a square, positive definite matrix. A set of vectors $\{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_m\}$ is called Q-*conjugate* if each distinct pair of vectors $\mathbf{x}_i, \mathbf{x}_j$ satisfy $\mathbf{x}_i^{\mathsf{T}} Q\mathbf{x}_j = 0$. A Q-conjugate set of vectors is linearly independent and can form a basis that diagonalizes the matrix Q. This guarantees that an iterative method to solve $Q\mathbf{x} = \mathbf{b}$ only require as many steps as there are basis vectors.

Solve a positive definite system $Q\mathbf{x} = \mathbf{b}$ is valuable in and of itself for certain problems, but it is also equivalent to minimizing certain functions. Specifically, consider the quadratic function

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^{\mathsf{T}} Q\mathbf{x} - \mathbf{b}^{\mathsf{T}}\mathbf{x} + c.$$

Because $Df(\mathbf{x})^{\mathsf{T}} = Q\mathbf{x} - \mathbf{b}$, minimizing f is the same as solving the equation

$$\mathbf{0} = Df(\mathbf{x})^{\mathsf{T}} = Q\mathbf{x} - \mathbf{b} \quad \Rightarrow \quad Q\mathbf{x} = \mathbf{b},$$

which is the original linear system. Note that the constant c does not affect the minimizer, since if $\mathbf{x}^*$ minimizes $f(\mathbf{x})$ it also minimizes $f(\mathbf{x}) + c$.

Using the conjugate directions guarantees an iterative method to converge on the minimizer because each iteration minimizes the objective function over a subspace of dimension equal to the iteration number. Thus, after n steps, where n is the number of conjugate basis vectors, the algorithm has found a minimizer over the entire space. In certain situations, this has a great advantage over gradient descent, which can bounce back and forth. This comparison is illustrated in Figure 12.2. Additionally, because the method utilizes a basis of conjugate vectors, the previous search direction can be used to find a conjugate projection onto the next subspace, saving computational time.

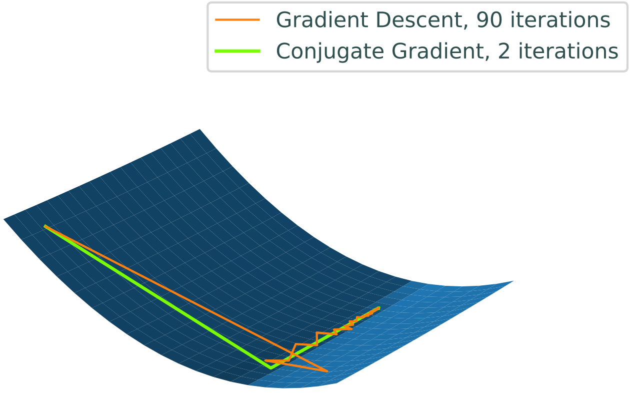


Figure 12.2: Paths traced by Gradient Descent (orange) and Conjugate Gradient (red) on a quadratic surface. Notice the zig-zagging nature of the Gradient Descent path, as opposed to the Conjugate Gradient path, which finds the minimizer in 2 steps.

Algorithm 12.1

```
procedure CONJUGATE GRADIENT($\mathbf{x}_0$, $Q$, $\mathbf{b}$, tol)
    $\mathbf{r}_0 \leftarrow Q\mathbf{x}_0 - \mathbf{b}$
    $\mathbf{d}_0 \leftarrow -\mathbf{r}_0$
    $k \leftarrow 0$
    while $\|\mathbf{r}_k\| \geq$ tol, $k < n$ do
        $\alpha_k \leftarrow \mathbf{r}_k^{\mathsf{T}}\mathbf{r}_k / \mathbf{d}_k^{\mathsf{T}} Q \mathbf{d}_k$
        $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{d}_k$
        $\mathbf{r}_{k+1} \leftarrow \mathbf{r}_k + \alpha_k Q \mathbf{d}_k$
        $\beta_{k+1} \leftarrow \mathbf{r}_{k+1}^{\mathsf{T}}\mathbf{r}_{k+1} / \mathbf{r}_k^{\mathsf{T}}\mathbf{r}_k$
        $\mathbf{d}_{k+1} \leftarrow -\mathbf{r}_{k+1} + \beta_{k+1}\mathbf{d}_k$
        $k \leftarrow k + 1.$
    return $\mathbf{x}_{k+1}$
```

The points $\mathbf{x}_k$ are the successive approximations to the minimizer, the vectors $\mathbf{d}_k$ are the conjugate descent directions, and the vectors $\mathbf{r}_k$ (which actually correspond to the steepest descent directions) are used in determining the conjugate directions. The constants α_k and β_k are used, respectively, in the line search, and in ensuring the Q-conjugacy of the descent directions.

Problem 2. Write a function that accepts an $n \times n$ positive definite matrix Q, a vector $\mathbf{b} \in \mathbb{R}^n$, an initial guess $\mathbf{x}_0 \in \mathbb{R}^n$, and a stopping tolerance. Use Algorithm 12.1 to solve the system $Q\mathbf{x} = \mathbf{b}$. Continue the algorithm until $\|\mathbf{r}_k\|$ is less than the tolerance, iterating no more than n times. Return the solution $\mathbf{x}$, whether or not the algorithm converged in n iterations or less, and the number of iterations computed. Test your function on the simple system

$$Q = \begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 1 \\ 8 \end{bmatrix},$$

which has solution $\mathbf{x}^* = \left[\frac{1}{2}, 2\right]^{\mathsf{T}}$. This is equivalent to minimizing the quadratic function $f(x, y) = x^2 + 2y^2 - x - 8y$; check that your function from Problem 1 gets the same solution.

More generally, you can generate a random positive definite matrix Q for testing by setting setting $Q = A^{\mathsf{T}}A$ for any A of full rank. Note, for values of $n \leq 5$ this method is not stable enough to always converge in exaclty n iterations. Try using the code given below to test your function for values of $n < 5$.

```
>>> import numpy as np
>>> from scipy import linalg as la

# Generate Q, b, and the initial guess x0.
>>> n = 4
>>> A = np.random.random((n,n))
>>> Q = A.T @ A
>>> b, x0 = np.random.random((2,n))

>>> x = la.solve(Q, b)      # Use your function here.
>>> np.allclose(Q @ x, b)
True
```

Non-linear Conjugate Gradient

The algorithm presented above is only valid for certain linear systems and quadratic functions, but the basic strategy may be adapted to minimize more general convex or non-linear functions. Though the non-linear version does not have guaranteed convergence as the linear formulation does, it can still converge in less iterations than the method of steepest descent. Modifying the algorithm for more general functions requires new formulas for α_k, $\mathbf{r}_k$, and β_k.

- The scalar α_k is simply the result of performing a line-search in the given direction $\mathbf{d}_k$ and is thus defined $\alpha_k = \underset{\alpha}{\operatorname{argmin}}\ f(\mathbf{x}_k + \alpha\mathbf{d}_k)$.
- The vector $\mathbf{r}_k$ in the original algorithm was really just the gradient of the objective function, so now define $\mathbf{r}_k = Df(\mathbf{x}_k)^\mathsf{T}$.
- The constants β_k can be defined in various ways, and the most correct choice depends on the nature of the objective function. A well-known formula, attributed to Fletcher and Reeves, is $\beta_k = Df(\mathbf{x}_k)Df(\mathbf{x}_k)^\mathsf{T}/Df(\mathbf{x}_{k-1})Df(\mathbf{x}_{k-1})^\mathsf{T}$.

Algorithm 12.2

```
procedure NON-LINEAR CONJUGATE GRADIENT($f$, $Df$, $\mathbf{x}_0$, tol, maxiter)
    $\mathbf{r}_0 \leftarrow -Df(\mathbf{x}_0)^\mathsf{T}$
    $\mathbf{d}_0 \leftarrow \mathbf{r}_0$
    $\alpha_0 \leftarrow \underset{\alpha}{\operatorname{argmin}} f(\mathbf{x}_0 + \alpha\mathbf{d}_0)$
    $\mathbf{x}_1 \leftarrow \mathbf{x}_0 + \alpha_0\mathbf{d}_0$
    $k \leftarrow 1$
    while $\|\mathbf{r}_k\| \geq$ tol, $k <$ maxiter do
        $\mathbf{r}_k \leftarrow -Df(\mathbf{x}_k)^\mathsf{T}$
        $\beta_k = \mathbf{r}_k^\mathsf{T}\mathbf{r}_k/\mathbf{r}_{k-1}^\mathsf{T}\mathbf{r}_{k-1}$
        $\mathbf{d}_k \leftarrow \mathbf{r}_k + \beta_k\mathbf{d}_{k-1}$.
        $\alpha_k \leftarrow \underset{\alpha}{\operatorname{argmin}}\ f(\mathbf{x}_k + \alpha\mathbf{d}_k)$.
        $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k\mathbf{d}_k$.
        $k \leftarrow k + 1$.
```

Problem 3. Write a function that accepts a convex objective function f, its derivative Df, an initial guess $\mathbf{x}_0$, a convergence tolerance defaultin to $1e^{-5}$, and a maximum number of iterations defaultin to 100. Use Algorithm 12.2 to compute the minimizer $\mathbf{x}^*$ of f. Return the approximate minimizer, whether or not the algorithm converged, and the number of iterations computed.

Compare your function to SciPy's `opt.fmin_cg()`.

```
>>> opt.fmin_cg(opt.rosen, np.array([10, 10]), fprime=opt.rosen_der)
Optimization terminated successfully.
```

```
         Current function value: 0.000000
         Iterations: 44
         Function evaluations: 102  # Much faster than steepest descent!
         Gradient evaluations: 102
array([ 1.00000007,  1.00000015])
```

Regression Problems

A major use of the conjugate gradient method is solving linear least squares problems. Recall that a least squares problem can be formulated as an optimization problem:

$$\mathbf{x}^* = \min_{\mathbf{x}} \|A\mathbf{x} - \ \mathbf{b}\|_2,$$

where A is an $m \times n$ matrix with full column rank, $\mathbf{x} \in \mathbb{R}^n$, and $\mathbf{b} \in \mathbb{R}^m$. The solution can be calculated analytically, and is given by

$$\mathbf{x}^* = (A^\mathsf{T} A)^{-1} A^\mathsf{T} \mathbf{b}.$$

In other words, the minimizer solves the linear system

$$A^\mathsf{T} A\mathbf{x} = A^\mathsf{T}\mathbf{b}. \tag{12.3}$$

Since A has full column rank, it is invertible, $A^\mathsf{T} A$ is positive definite, and for any non-zero vector $\mathbf{z}$, $A\mathbf{z} \neq 0$. Therefore, $\mathbf{z}^\mathsf{T} A^\mathsf{T} A\mathbf{z} = \|A\mathbf{z}\|^2 > 0$. As $A^\mathsf{T} A$ is positive definite, conjugate gradient can be used to solve Equation 12.3.

Linear least squares is the mathematical underpinning of *linear regression*. Linear regression involves a set of real-valued data points $\{y_1, \ldots, y_m\}$, where each y_i is paired with a corresponding set of predictor variables $\{x_{i,1}, x_{i,2}, \ldots, x_{i,n}\}$ with $n < m$. The linear regression model posits that

$$y_i = \beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + \cdots + \beta_n x_{i,n} + \varepsilon_i$$

for $i = 1, 2, \ldots, m$. The real numbers $\beta_0, \ldots, \beta_n$ are known as the parameters of the model, and the ε_i are independent, normally-distributed error terms. The goal of linear regression is to calculate the parameters that best fit the data. This can be accomplished by posing the problem in terms of linear least squares. Define

$$\mathbf{b} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}, \quad A = \begin{bmatrix} 1 & x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ 1 & x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_n \end{bmatrix}.$$

The solution $\mathbf{x}^* = [\beta_0^*, \beta_1^*, \ldots, \beta_n^*]^\mathsf{T}$ to the system $A^\mathsf{T} A\mathbf{x} = A^\mathsf{T}\mathbf{b}$ gives the parameters that best fit the data. These values can be understood as defining the hyperplane that best fits the data.

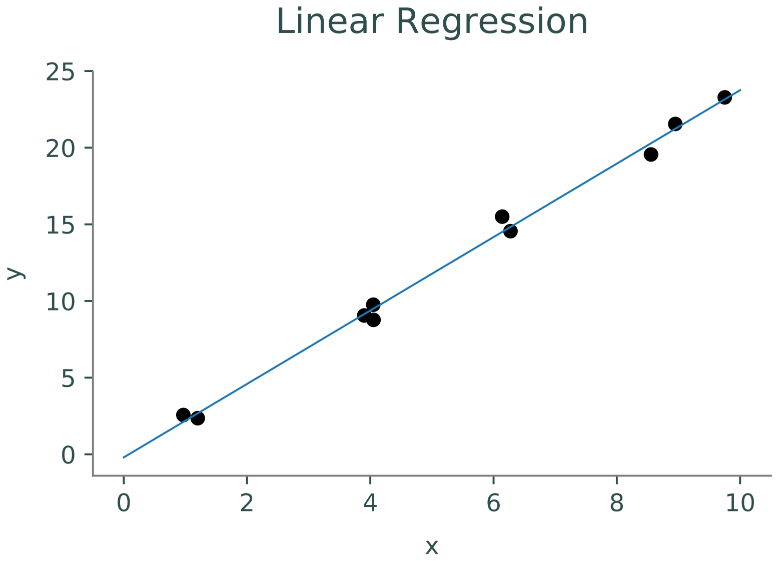


Figure 12.3: Solving the linear regression problem results in a best-fit hyperplane.

Problem 4. Using your function from Problem 2, solve the linear regression problem specified by the data contained in the file[a] `linregression.txt`. This is a whitespace-delimited text file formatted so that the i-th row consists of $y_i, x_{i,1}, \ldots, x_{i,n}$. Use `np.loadtxt()` to load in the data and return the solution to the normal equations.

[a] Source: Statistical Reference Datasets website at `http://www.itl.nist.gov/div898/strd/lls/data/LINKS/v-Longley.shtml`.

Logistic Regression

Logistic regression is another important technique in statistical analysis and machine learning that builds off of the concepts of linear regression. As in linear regression, there is a set of predictor variables $\{x_{i,1}, x_{i,2}, \ldots, x_{i,n}\}_{i=1}^m$ with corresponding outcome variables $\{y_i\}_{i=1}^m$. In logistic regression, the outcome variables y_i are binary and can be modeled by a *sigmoidal* relationship. The value of the predicted y_i can be thought of as the probability that $y_i = 1$. In mathematical terms,

$$\mathbb{P}(y_i = 1 \,|\, x_{i,1}, \ldots, x_{i,n}) = p_i,$$

where

$$p_i = \frac{1}{1 + \exp(-(\beta_0 + \beta_1 x_{i,1} + \cdots + \beta_n x_{i,n}))}.$$

The parameters of the model are the real numbers $\beta_0, \beta_1, \ldots, \beta_n$. Note that $p_i \in (0, 1)$ regardless of the values of the predictor variables and parameters.

The probability of observing the outcome variables y_i under this model, assuming they are independent, is given by the *likelihood function* $\mathcal{L} : \mathbb{R}^{n+1} \to \mathbb{R}$

$$\mathcal{L}(\beta_0, \ldots, \beta_n) = \prod_{i=1}^{m} p_i^{y_i} (1 - p_i)^{1-y_i}.$$

The goal of logistic regression is to find the parameters $\beta_0, \ldots, \beta_k$ that maximize this likelihood function. Thus, the problem can be written as:

$$\max_{(\beta_0,\ldots,\beta_n)} \mathcal{L}(\beta_0, \ldots, \beta_n).$$

Maximizing this function is often a numerically unstable calculation. Thus, to make the objective function more suitable, the logarithm of the objective function may be maximized because the logarithmic function is strictly monotone increasing. Taking the log and turning the problem into a minimization problem, the final problem is formulated as:

$$\min_{(\beta_0,\ldots,\beta_n)} -\log \mathcal{L}(\beta_0, \ldots, \beta_n).$$

A few lines of calculation reveal that this objective function can also be rewritten as

$$\begin{aligned}-\log \mathcal{L}(\beta_0, \ldots, \beta_n) = &\sum_{i=1}^{m} \log(1 + \exp(-(\beta_0 + \beta_1 x_{i,1} + \cdots + \beta_n x_{i,n})))+\\ &\sum_{i=1}^{m}(1 - y_i)(\beta_0 + \beta_1 x_{i,1} + \cdots + \beta_n x_{i,n}).\end{aligned}$$

The values for the parameters $\{\beta_i\}_{i=1}^n$ that we obtain are known as the *maximum likelihood estimate* (MLE). To find the MLE, conjugate gradient can be used to minimize the objective function.

For a one-dimensional binary logistic regression problem, we have predictor data $\{x_i\}_{i=1}^m$ with labels $\{y_i\}_{i=1}^m$ where each $y_i \in \{0, 1\}$. The negative log likelihood then becomes the following.

$$-\log \mathcal{L}(\beta_0, \beta_1) = \sum_{i=1}^{m} \log(1 + e^{-(\beta_0 + \beta_1 x_i)}) + (1 - y_i)(\beta_0 + \beta_1 x_i) \tag{12.4}$$

Problem 5. Write a class for doing binary logistic regression in one dimension that implement the following methods.

1. `fit()`: accept an array $\mathbf{x} \in \mathbb{R}^n$ of data, an array $\mathbf{y} \in \mathbb{R}^n$ of labels (0s and 1s), and an initial guess $\boldsymbol{\beta}_0 \in \mathbb{R}^2$. Define the negative log likelihood function as given in (12.4), then minimize it (with respect to $\boldsymbol{\beta}$) with your function from Problem 3 or `opt.fmin_cg()`. Store the resulting parameters β_0 and β_1 as attributes.
2. `predict()`: accept a float $x \in \mathbb{R}$ and calculate

$$\sigma(x) = \frac{1}{1 + \exp(-(\beta_0 + \beta_1 x))},$$

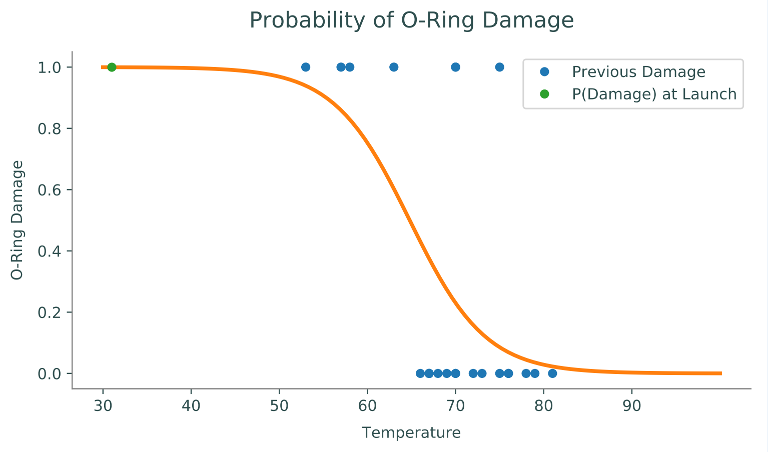
where β_0 and β_1 are the optimal values calculated in `fit()`. The value $\sigma(x)$ is the probability that the observation x should be assigned the label $y = 1$.

This class does not need an explicit constructor. You may assume that `predict()` will be called after `fit()`.

Problem 6. On January 28, 1986, less than two minutes into the Challenger space shuttle's 10th mission, there was a large explosion that originated from the spacecraft, killing all seven crew members and destroying the shuttle. The investigation that followed concluded that the malfunction was caused by damage to O-rings that are used as seals for parts of the rocket engines. There were 24 space shuttle missions before this disaster, some of which had noted some O-ring damage. Given the data, could this disaster have been predicted?

The file `challenger.npy` contains data for 23 missions (during one of the 24 missions, the engine was lost at sea). The first column ($\mathbf{x}$) contains the ambient temperature, in Fahrenheit, of the shuttle launch. The second column ($\mathbf{y}$) contains a binary indicator of the presence of O-ring damage (1 if O-ring damage was present, 0 otherwise).

Instantiate your class from Problem 5 and fit it to the data, using an initial guess of $\boldsymbol{\beta}_0 = [20, -1]^\mathsf{T}$. Plot the resulting curve $\sigma(x)$ for $x \in [30, 100]$, along with the raw data. Return the predicted probability (according to this model) of O-ring damage on the day the shuttle was launched, given that it was 31°F.


Probability of O-Ring Damage
O-Ring Damage
Temperature
● Previous Damage
● P(Damage) at Launch