6 Logistic Regression

Lab Objective: *Understand the basic principles of Logistic Regression and binary classifiers. Apply this to a dataset.*

Linear regression is unsuitable for predicting probabilities, because the resulting model may take values in any fixed interval in $\mathbb{R}$, but a probability-predicting model can only take values in the interval $[0,1]$. *Logistic regression* is a form of regression that always takes its values in the interval $[0,1]$ and as such, is a popular method for predicting probabilities and for constructing *classifiers*. As in linear regression, in a classification problem we have a random variable Y, conditioned on an input $X \in \mathbb{R}^d$. However, in *binary classification* problems the random variable Y is binary, that is, $Y \in \{0,1\}$. A *binary classifier* is any function f taking values in $\{0,1\}$. For example, $\mathbf{x} \in \mathbb{R}^d$ could be the pixel intensities of an image, and the classifier f gives 1 if the image is a picuture of a duck and 0 otherwise. The goal of a classification problem is to choose a classifier $\widehat{f}$ so that $(X, \widehat{f}(X))$ is a good approximation for (X,Y).

Logistic Regression

Logistic regression relies heavily on the *logistic function*, also known as the *sigmoid function*, $\text{sigm} : \mathbb{R} \to (0,1)$ given by

$$\text{sigm}(x) = \frac{1}{1+e^{-x}}. \tag{6.1}$$

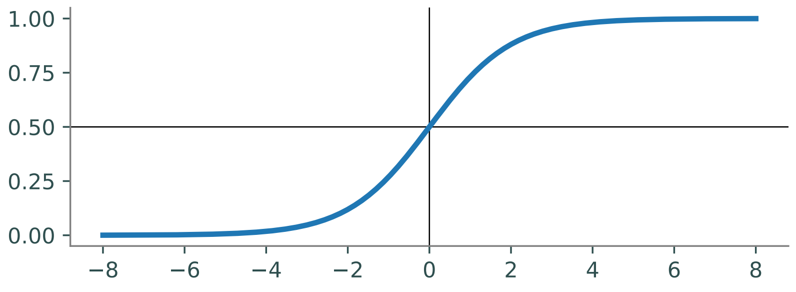

Figure 6.1: Sigmoid Function

This function works well for classifying objects based on probabilities, because it has some key properties that translate well into probability theory. Of particular note, the graph can translated by adding a constant, giving the form $\text{sigm}(\beta_1 t + \beta_0)$. A larger value of β_1 makes the ramp up from 0 to 1 steeper, while a smaller value of β_1 makes it less steep. The trick behind logistic regression is to find the values of β_i such that the resulting sigmoid function best classifies the data.

In logistic regression models we have a random variable Y with support $\{0, 1\}$, where Y is conditioned on another random variable X, with support in $\mathbb{R}^d$. The distribution of Y, given X, is assumed to be Bernoulli

$$Y \,\big|\, X \sim \text{Bernoulli}(\text{sigm}(X^\mathsf{T}\boldsymbol{\beta})),$$

so that

$$P(Y \,\big|\, X) = \text{sigm}(X^\mathsf{T}\boldsymbol{\beta}) = \frac{1}{1 + \exp(-X^\mathsf{T}\boldsymbol{\beta})}.$$

As in the case of linear regression, we usually add a constant feature $X_0 = \mathbf{1}$ to X and a corresponding coefficient β_0 to $\boldsymbol{\beta}$, so that $X^\mathsf{T}\boldsymbol{\beta} = \beta_0 + \beta_1 X_1 + \cdots + \beta_d X_d$. Given a draw of length n of the form $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ we wish to estimate $\boldsymbol{\beta}$. The maximum likelihood estimator is a good choice. To find this estimator, first observe that the likelihood of $\boldsymbol{\beta}$, given the data, is

$$\begin{aligned} L(\boldsymbol{\beta} \,\big|\, D) &= \prod_{i=1}^{n} P(Y = y_i, X = \mathbf{x}_i \,\big|\, \boldsymbol{\beta}) \\ &= \prod_{i=1}^{n} P(Y = y_i \,\big|\, X = \mathbf{x}_i, \boldsymbol{\beta}) P(X_i). \end{aligned}$$

which is equivalent to maximizing

$$\prod_{i=1}^{n} P(Y = y_i \,\big|\, X = x_i, \boldsymbol{\beta}) = \prod_{i=1}^{n} p_i^{y_i}(1 - p_i)^{1 - y_i}$$

where

$$p_i = P(Y = 1 \,\big|\, \mathbf{x}_i, \boldsymbol{\beta}) = \text{sigm}(\mathbf{x}_i^\mathsf{T}\boldsymbol{\beta}) = \frac{1}{1 + \exp(-\mathbf{x}_i^\mathsf{T}\boldsymbol{\beta})}.$$

Taking the negative logarithm turns this into a convex minimization problem, and a little math shows that

$$\ell(\boldsymbol{\beta} \,\big|\, D) = \sum_{i=1}^{n} \left(y_i \log(1 + \exp(-\mathbf{x}_i^\mathsf{T}\boldsymbol{\beta})) + (1 - y_i) \log(1 + \exp(\mathbf{x}_i^\mathsf{T}\boldsymbol{\beta})) \right). \tag{6.2}$$

The convexity of this problem implies there is a unique minimizer $\widehat{\boldsymbol{\beta}}$ of $\ell(\boldsymbol{\beta} \,\big|\, D)$.

Problem 1. Create a Python classifier called `LogiReg` that accepts an $(n \times 1)$ array y of binary labels (0's and 1's) as well as an $(n \times d)$ array X of data points. Write a `fit()` method that uses equation 6.2 to find and save the optimal $\widehat{\boldsymbol{\beta}}$.

Once the maximum likelihood estimate $\widehat{\boldsymbol{\beta}}$ is found, we have an estimate for the probability

$$P(Y = 1 \,|\, \mathbf{x}) \approx \mathrm{sigm}(\mathbf{x}^{\mathsf{T}}\widehat{\boldsymbol{\beta}}).$$

From this, we can construct a classifier $\widehat{f}$ by setting $\widehat{f}(x) = 1$ if $P(Y = 1 \,|\, \mathbf{x}) \geq \frac{1}{2}$ and $\widehat{f}(x) = 0$ otherwise.

Problem 2. Write a method called `predict_prob()` for your classifier that accepts an $(n \times d)$ array x_test and returns $P(Y = 1 \,|\, x\_test)$. Also write a method called `predict()` that calls `predict_prob()` and returns an array of predicted labels (0's or 1's) for the given array x_test.

Problem 3. To test your classifier, create training arrays X and y as well as testing array X_test. The code to generate X, y, and X_test is provided below. Both X and X_test have 100 random draws from a 2-dimensional multivariate normal distribution centered at $(1, 2)$, and another 100 draws from one centered at $(4, 3)$.

Train your classifier on X and y. Then generate a list of predicted labels using your trained classifier and X_test, and use it to plot X_test with a different color for each predicted label. Your plot should look similar to Figure 6.2.

```
>>> import numpy as np

>>> data = np.column_stack((
        # draw from 2 2-dim. multivariate normal dists.
        np.concatenate((
        np.random.multivariate_normal(np.array([1,2]), np.eye(2), 100),
        np.random.multivariate_normal(np.array([4,3]), np.eye(2), 100)
        )),
        # labels corresonding to each distribution
        np.concatenate(( np.zeros(100), np.ones(100) )) ))
>>> np.random.shuffle(data)
>>> # extract X and y from the shuffled data
>>> X = data[:,:2]
>>> y = data[:,2].astype(int)

>>> X_test = np.concatenate((
        # draw from 2 identical 2-dim. multivariate normal dists.
        np.random.multivariate_normal(np.array([1,2]), np.eye(2), 100),
        np.random.multivariate_normal(np.array([4,3]), np.eye(2), 100)
        ))
>>> np.random.shuffle(X_test)
```

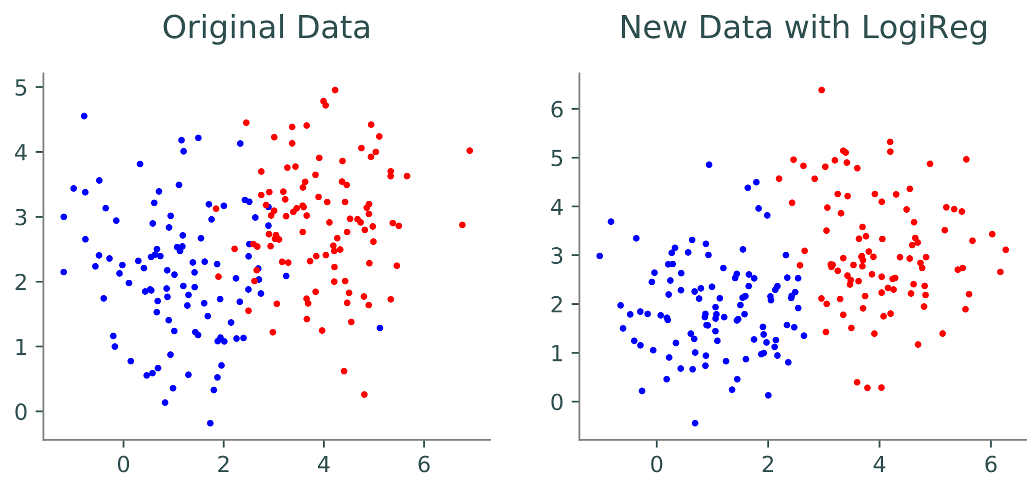

Figure 6.2: In reality, these two distributions overlap a little, but the logistic regression model makes a clean divide between the two.

Statsmodels and Sklearn

The module `statsmodels` contains a package that includes a logistic regression class called `Logit`. A simple example of this class being implemented is as follows.

```python
>>> import statsmodels.api as sm

>>> model = sm.Logit(y, X).fit(disp=0)  # setting disp=0 turns off printed info
>>> probs = model.predict(X_test)  # list of probabilities, not labels
```

`Logit` does *not* add a constant feature (column of 1's) to X by default, so in order to do so, you must apply the function `sm.add_constant()` to both X and X_test. In addition, the `.fit()` method does not regularize the problem by default, which may lead to some errors involving singular matrices. To fix this, you can use the `.fit_regularized()` method instead of `.fit()`.

The module `sklearn` also has a package for logistic regression called `LogisticRegression`, which can be implemented as follows.

```python
>>> from sklearn.linear_model import LogisticRegression

>>> model = LogisticRegression(fit_intercept=True).fit(X, y)  # X before y
>>> labels = model.predict(X_test)  # predicted labels of X_test
```

`LogisticRegression` already regularizes the problem by default. The parameter `fit_intercept` (which defaults to `False`) indicates whether you want to add a constant feature (column of 1's) to X and X_test.

You can also use `sklearn` to score a logistic regression model. After fitting an `sklearn` model, you can call `<model>.score(X_test, y_test)` to find the percentage of accuracy of the model's prediction for X_test, given the true labels in y_test. Alternatively, you can use `sklearn.metrics.accuracy_score` to find the percentage of accuracy between a list of predicted labels and the list of true labels.

```
>>> from sklearn.metrics import accuracy_score

>>> true_labels = [0, 1, 2, 3, 4]
>>> pred_labels = [0, 2, 2, 2, 4]  # predicted labels from logistic regression
>>> accuracy_score(true_labels, predicted_labels)
0.6
```

Problem 4. The code to generate arrays X, y, X_test, and y_test is provided below. X and X_test are each composed of 200 draws from two 20-dimensional multivariate normal distributions, one centered at $\mathbf{0}$, and the other centered at $\mathbf{2}$.

Using each of `LogiReg`, `statsmodels`, and `sklearn`, train a logistic regression classifier on X and y to generate a list of predicted labels for X_test. Then, using y_test, print the accuracy scores for each trained model. Compare the accuracies and training/testing time for all three classifiers. Be sure to add a constant feature with each model.

```
    >>> # predefine the true beta
    >>> beta = np.random.normal(0, 7, 20)

    >>> # X is generated from 2 20-dim. multivariate normal dists.
    >>> X = np.concatenate((
            np.random.multivariate_normal(np.zeros(20), np.eye(20), 100),
            np.random.multivariate_normal(np.ones(20)*2, np.eye(20), 100)
            ))
    >>> np.random.shuffle(X)
    >>> # create y based on the true beta
    >>> pred = 1. / (1. + np.exp(-X @ beta))
    >>> y = np.array( [1 if pred[i] >= 1/2 else 0
                for i in range(pred.shape[0])] )

    >>> # X_test and y_test are generated similar to X and y
    >>> X_test = np.concatenate((
            np.random.multivariate_normal(np.zeros(20), np.eye(20), 100),
            np.random.multivariate_normal(np.ones(20), np.eye(20), 100)
            ))
    >>> np.random.shuffle(X_test)
    >>> pred = 1. / (1. + np.exp(-X_test @ beta))
    >>> y_test = np.array( [1 if pred[i] >= 1/2 else 0
                for i in range(pred.shape[0])] )
```

Multiclass Logistic Regression

Sometimes we may want to classify data into more than two categories, but so far we've only used logistic regression as a binary classifier. The good news is that we can extend logistic regression to classify more than just two categories.

The more popular method for doing this is to generalize the logistic regression model to a multiclass setting. This method is called *multinomial logistic regression* or sometimes *softmax regression.* While standard logistic regression was based on the sigmoid function, multinomial logistic regression is based on the *softmax function* $\mathscr{S} : \mathbb{R}^k \to (0,1)^k$, which is a multivariate version of the sigmoid function, given by

$$\mathscr{S}(t_1, \ldots, t_k) = \left(\frac{e^{t_1}}{\sum_{j=1}^k e^{t_j}}, \ldots, \frac{e^{t_k}}{\sum_{j=1}^k e^{t_j}} \right). \tag{6.3}$$

We will assume that $Y \,|\, X$ is categorically distributed as

$$\text{Cat}(p_1(X), \ldots, p_k(X)) = \text{Cat}(\mathscr{S}(X^\mathsf{T}\boldsymbol{\beta}_1, \ldots, X^\mathsf{T}\boldsymbol{\beta}_k))$$

for some choice of vectors $\boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_k$, which we will estimate from the data. Here

$$p_i(X) = P(Y = i \,|\, X) = \frac{e^{X^\mathsf{T}\boldsymbol{\beta}_i}}{\sum_{j=1}^k e^{X^\mathsf{T}\boldsymbol{\beta}_j}} = \frac{\text{sigm}(X^\mathsf{T}\boldsymbol{\beta}_i)}{\sum_{j=1}^k \text{sigm}(X^\mathsf{T}\boldsymbol{\beta}_j)}.$$

Given a draw of length n of the form $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, we wish to compute $\boldsymbol{\theta} = (\boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_k)$ where, without loss of generality, we may assume $\boldsymbol{\beta}_k = \mathbf{0}$. The maximum likelihood estimate of $\boldsymbol{\theta}$ is computed in a manner similar to the way it was for standard logistic regression. A bit of math shows that

$$\begin{aligned}\ell(\boldsymbol{\theta} \,|\, D) &= -\sum_{i=1}^n \sum_{j=1}^k \delta_{c_j}(y_i) \log(p_j(\mathbf{x}_i)) \\ &= -\sum_{i=1}^n \sum_{j=1}^k \delta_{c_j}(y_i) \log\left(\frac{e^{\mathbf{x}_i^\mathsf{T}\boldsymbol{\beta}_j}}{\sum_{m=1}^k e^{\mathbf{x}_i^\mathsf{T}\boldsymbol{\beta}_m}} \right)\end{aligned}$$

where

$$\delta_{c_j}(y_i) = \begin{cases} 1 & \text{if } y_i = c_j \text{, the jth class} \\ 0 & \text{otherwise.} \end{cases}$$

This is a convex minimization problem with unique minimizer $\widehat{\boldsymbol{\theta}}$. Once $\widehat{\boldsymbol{\theta}} = (\widehat{\boldsymbol{\beta}_1}, \ldots, \widehat{\boldsymbol{\beta}_k})$ is found, we have an estimate for the probability

$$P(Y = y \,|\, \mathbf{x}) \approx \frac{e^{\mathbf{x}^\mathsf{T}\widehat{\boldsymbol{\beta}}_y}}{\sum_{j=1}^k e^{\mathbf{x}^\mathsf{T}\widehat{\boldsymbol{\beta}}_j}}.$$

From this, we can construct a classifier $\widehat{f}$ by setting $\widehat{f}(\mathbf{x}) = \text{argmax}_j P(Y = c_j \,|\, \mathbf{x})$.

Conveniently, `sklearn` has a very simple implementation of multinomial logistic regression that simply requires the argument `multi_class='multinomial'` when initiating a `LogisticRegression` model.

```python
>>> from sklearn.linear_model import LogisticRegression

>>> model = LogisticRegression(
                multi_class='multinomial',
                fit_intercept=True).fit(X, y) # add constant feature
```

Problem 5. The Iris Dataset contains information taken from 150 samples of 3 different types of iris flowers (Setosa, Versicolor, and Virginica). The columns contain measurements for sepal length, sepal width, pedal length, and pedal width. Import the Iris Dataset and perform a train-test split on only the first two columns of the data with `test_size=0.4`. Train a multinomial logistic regression model using the training data with an added constant feature, and generate prediction labels for the test data. Plot the test data by color using your prediction labels.

Your plot should reflect Figure 6.3

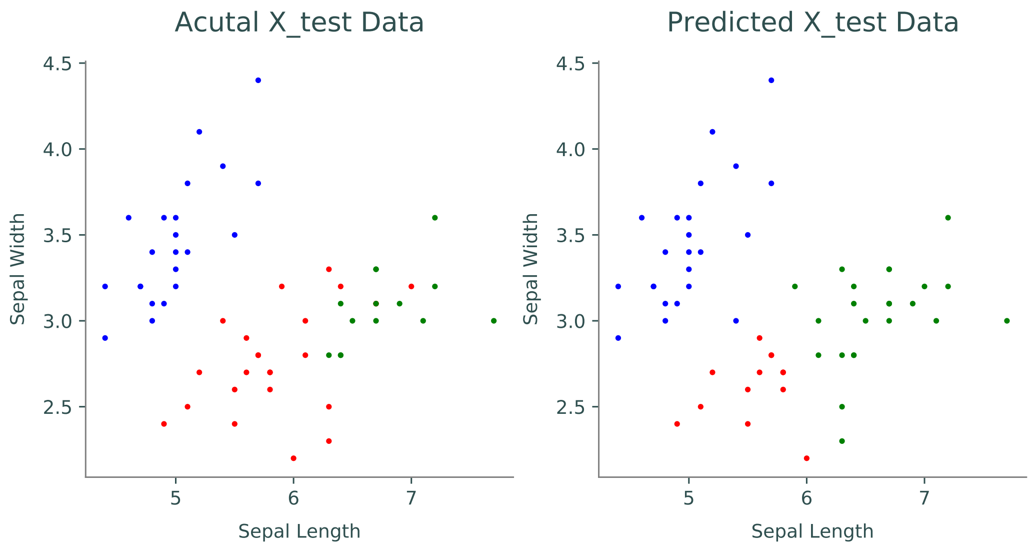

Figure 6.3: Multinomial logistic regression attempt to categorize the Iris Dataset.