lecture4-classification

October 28, 2021

1 Lecture 4: Classification and Logistic Regression

1.0.1 Applied Machine Learning

Volodymyr Kuleshov Cornell Tech

2 Announcements

- Homework 1 is out on Gradescope
- Our Python and Numpy tutorial is this week

3 Part 1: Classification

So far, every supervised learning algorithm that we've seen has been an instance of regression.

We will next look at classification. First, let's define what classification is.

4 Review: Components of a Supervised Machine Learning Problem

To apply supervised learning, we define a dataset and a learning algorithm.

$$\underbrace{\text{Dataset}}_{\text{Features, Attributes, Targets}} \quad + \quad \underbrace{\text{Learning Algorithm}}_{\text{Model Class + Objective + Optimizer}} \quad \rightarrow \text{Predictive Model}$$

The output is a predictive model that maps inputs to targets. For instance, it can predict targets on new inputs.

5 Review: Regression vs. Classification

Consider a training dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(n)}, y^{(n)})\}$.

We distinguish between two types of supervised learning problems depnding on the targets $y^{(i)}$.

1. **Regression**: The target variable $y \in \mathcal{Y}$ is continuous: $\mathcal{Y} \subseteq \mathbb{R}$.
2. **Classification**: The target variable y is discrete and takes on one of K possible values: $\mathcal{Y} = \{y_1, y_2, \ldots y_K\}$. Each discrete value corresponds to a *class* that we want to predict.

6 Binary Classification

An important special case of classification is when the number of classes $K = 2$.

In this case, we have an instance of a *binary classification* problem.

7 Classification Dataset: Iris Flowers

To demonstrate classification algorithms, we are going to use the Iris flower dataset.

It's a classical dataset originally published by R. A. Fisher in 1936. Nowadays, it's widely used for demonstrating machine learning algorithms.

[1]:
```
import numpy as np
import pandas as pd
from sklearn import datasets
import warnings
warnings.filterwarnings('ignore')

# Load the Iris dataset
iris = datasets.load_iris(as_frame=True)

print(iris.DESCR)
```

```
.. _iris_dataset:

Iris plants dataset
--------------------

**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica

    :Summary Statistics:

    ============== ==== ==== ======= ===== ====================
                    Min  Max   Mean    SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length:   4.3  7.9   5.84   0.83    0.7826
```

```
    sepal width:    2.0  4.4   3.05   0.43   -0.4194
    petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)
    petal width:    0.1  2.5   1.20   0.76    0.9565  (high!)
    ============== ==== ==== ======= ===== ====================

    :Missing Attribute Values: None
    :Class Distribution: 33.3% for each of 3 classes.
    :Creator: R.A. Fisher
    :Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
    :Date: July, 1988

The famous Iris database, first used by Sir R.A. Fisher. The dataset is taken
from Fisher's paper. Note that it's the same as in R, but not as in the UCI
Machine Learning Repository, which has two wrong data points.

This is perhaps the best known database to be found in the
pattern recognition literature.  Fisher's paper is a classic in the field and
is referenced frequently to this day.  (See Duda & Hart, for example.)  The
data set contains 3 classes of 50 instances each, where each class refers to a
type of iris plant.  One class is linearly separable from the other 2; the
latter are NOT linearly separable from each other.

.. topic:: References

   - Fisher, R.A. "The use of multiple measurements in taxonomic problems"
     Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to
     Mathematical Statistics" (John Wiley, NY, 1950).
   - Duda, R.O., & Hart, P.E. (1973) Pattern Classification and Scene Analysis.
     (Q327.D83) John Wiley & Sons.  ISBN 0-471-22361-1.  See page 218.
   - Dasarathy, B.V. (1980) "Nosing Around the Neighborhood: A New System
     Structure and Classification Rule for Recognition in Partially Exposed
     Environments".  IEEE Transactions on Pattern Analysis and Machine
     Intelligence, Vol. PAMI-2, No. 1, 67-71.
   - Gates, G.W. (1972) "The Reduced Nearest Neighbor Rule".  IEEE Transactions
     on Information Theory, May 1972, 431-433.
   - See also: 1988 MLC Proceedings, 54-64.  Cheeseman et al"s AUTOCLASS II
     conceptual clustering system finds 3 classes in the data.
   - Many, many more …
```

[2]:
```
# print part of the dataset
iris_X, iris_y = iris.data, iris.target
pd.concat([iris_X, iris_y], axis=1).head()
```

[2]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	\
0	5.1	3.5	1.4	0.2	
1	4.9	3.0	1.4	0.2	
2	4.7	3.2	1.3	0.2	

```
3                4.6               3.1               1.5               0.2
4                5.0               3.6               1.4               0.2
```

```
   target
0       0
1       0
2       0
3       0
4       0
```

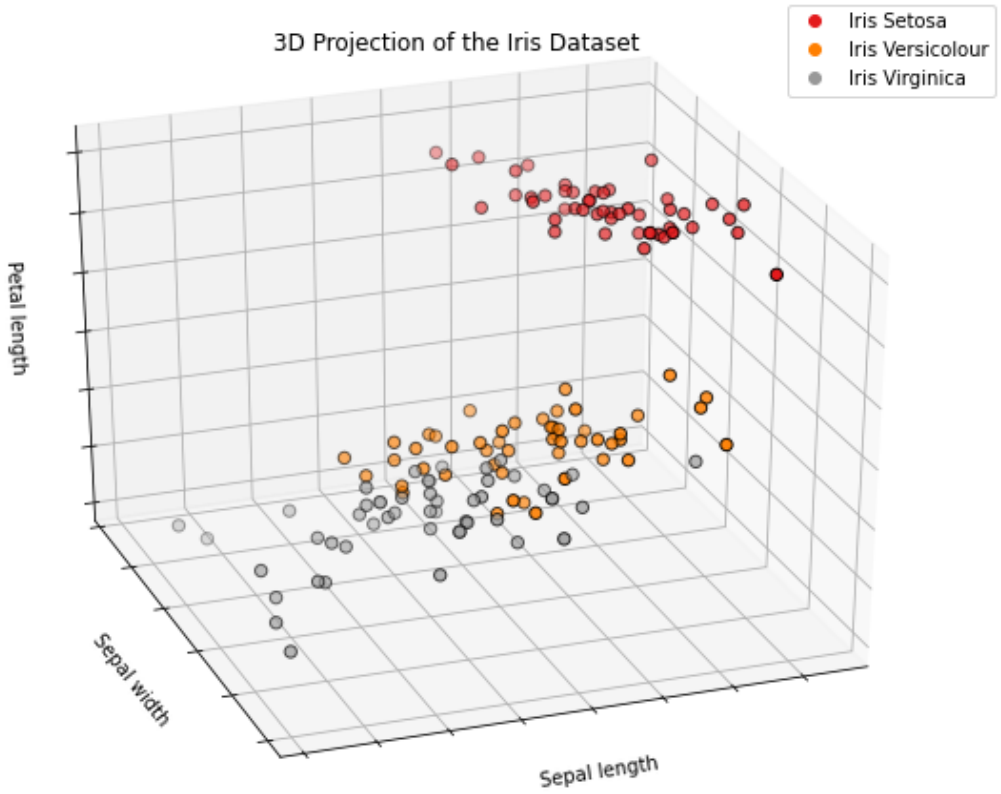
Let's visualize this dataset. We first import `matplotlib` and other plotting tools.

[3]:
```
# Code from: https://scikit-learn.org/stable/auto_examples/datasets/
 ↪plot_iris_dataset.html
%matplotlib inline
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.decomposition import PCA
```

Here is a visualization of this dataset in 3D. Note that we are using the first 3 features (out of 4) in this dateset.

[4]:
```
# let's visualize this dataset
fig = plt.figure(1, figsize=(8, 6))
ax = Axes3D(fig, elev=-150, azim=110)
X_reduced = iris_X.to_numpy()[:,:3]
ax.set_title("3D Projection of the Iris Dataset")
ax.w_xaxis.set_ticklabels([])
ax.w_yaxis.set_ticklabels([])
ax.set_xlabel("Sepal length")
ax.set_ylabel("Sepal width")
ax.set_zlabel("Petal length")
ax.w_zaxis.set_ticklabels([])
p1 = ax.scatter(X_reduced[:, 0], X_reduced[:, 1], X_reduced[:, 2], c=iris_y,
           cmap=plt.cm.Set1, edgecolor='k', s=40)
plt.legend(handles=p1.legend_elements()[0], labels=['Iris Setosa', 'Iris␣
 ↪Versicolour', 'Iris Virginica'])
```

[4]:
```
<matplotlib.legend.Legend at 0x1236419e8>
```



8 Understanding Classification

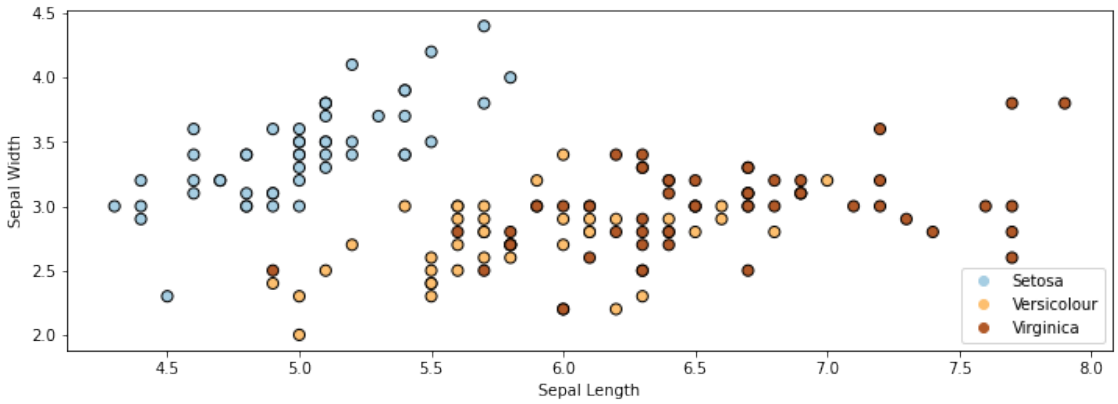
How is clasification different from regression? * In regression, we try to fit a curve through the set of targets $y^{(i)}$.

- In classification, classes define a partition of the feature space, and our goal is to find the boundaries that separate these regions.
- Outputs of classification models often have a simple probabilistic interpretation: they are probabilities that a data point belongs to a given class.

Let's look at our Iris dataset again. Note that we are now only using the first 2 attributes in this dateset.

[5]:
```
plt.rcParams['figure.figsize'] = [12, 4]
p1 = plt.scatter(iris_X.iloc[:, 0], iris_X.iloc[:, 1], c=iris_y,
            edgecolor='k', s=50, cmap=plt.cm.Paired)
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.legend(handles=p1.legend_elements()[0], labels=['Setosa', 'Versicolour',␣
↪'Virginica'], loc='lower right')
```

[5]: <matplotlib.legend.Legend at 0x12376d8d0>



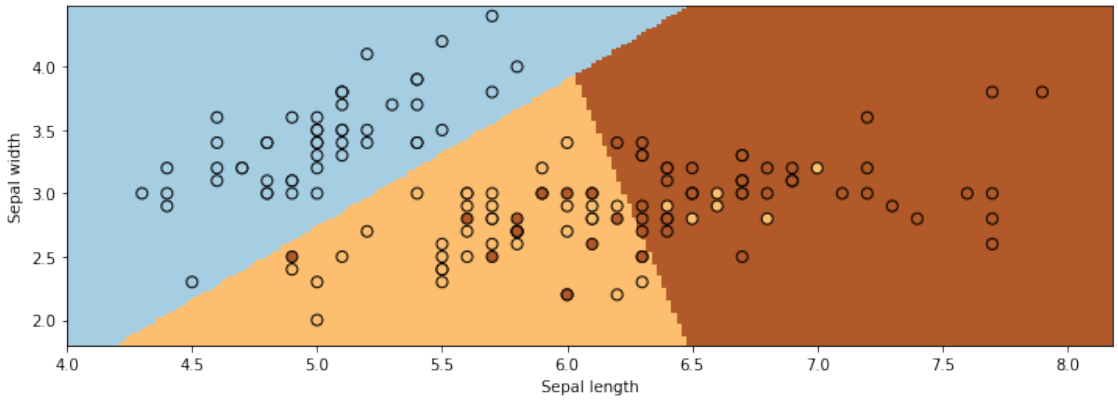
Let's train a classification algorithm on this data.

[6]:
```
from matplotlib import pyplot as plt
plt.rcParams['figure.figsize'] = [12, 4]
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression(C=1e5)

# Create an instance of Logistic Regression Classifier and fit the data.
X = iris_X.to_numpy()[:,:2]
# rename class two to class one
Y = iris_y.copy()
logreg.fit(X, Y)
```

[6]: LogisticRegression(C=100000.0)

Below, we see the regions predicted to be associated with the blue, brown, and yellow classes and the lines between them are the decision boundaries.

[7]:
```
xx, yy = np.meshgrid(np.arange(4, 8.2, .02), np.arange(1.8, 4.5, .02))
Z = logreg.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)

# Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=Y, edgecolors='k', cmap=plt.cm.Paired, s=50)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')

plt.show()
```



9 Part 2: Logistic Regression

Next, we are going to see our first classification algorithm: logistic regression.

10 Review: Classification

Consider a training dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(n)}, y^{(n)})\}$.

We distinguish between two types of supervised learning problems depnding on the targets $y^{(i)}$.

1. **Regression**: The target variable $y \in \mathcal{Y}$ is continuous: $\mathcal{Y} \subseteq \mathbb{R}$.
2. **Classification**: The target variable y is discrete and takes on one of K possible values: $\mathcal{Y} = \{y_1, y_2, \ldots y_K\}$. Each discrete value corresponds to a *class* that we want to predict.

11 Binary Classification and the Iris Dataset

We will to start by looking at binary (two-class) classification.

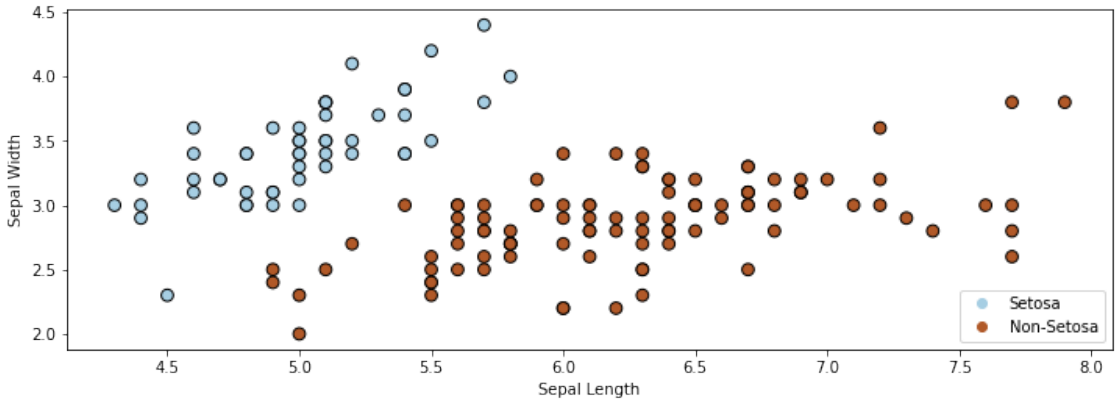
To keep things simple, we will use the Iris dataset. We will attempt to distinguish class 0 (Iris Setosa) from the other two classes.

We only use the first two features in the dataset. Our task is to tell apart Setosa flowers from non-Setosa flowers.

[9]:
```
# rename class two to class one
iris_y2 = iris_y.copy()
iris_y2[iris_y2==2] = 1

# Plot also the training points
p1 = plt.scatter(iris_X.iloc[:, 0], iris_X.iloc[:, 1], c=iris_y2,
            edgecolor='k', s=60, cmap=plt.cm.Paired)
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
```

```
plt.legend(handles=p1.legend_elements()[0], labels=['Setosa', 'Non-Setosa'],␣
↪loc='lower right')
```

[9]:
```
<matplotlib.legend.Legend at 0x123b7add8>
```



12 Review: Least Squares

Recall that the linear regression algorithm fits a linear model of the form

$$f(x) = \sum_{j=0}^{d} \theta_j \cdot x_j = \theta^\top x.$$

It minimizes the mean squared error (MSE)

$$J(\theta) = \frac{1}{2n} \sum_{i=1}^{n} (y^{(i)} - \theta^\top x^{(i)})^2$$

on a dataset $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(n)}, y^{(n)})\}$.

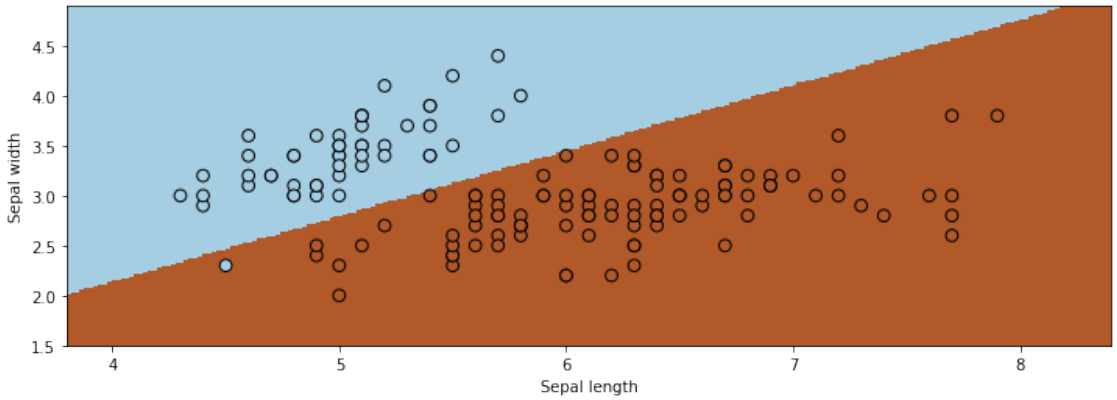
We could use least squares to solve our classification problem, setting $\mathcal{Y} = \{0, 1\}$.

[16]:
```
# https://scikit-learn.org/stable/auto_examples/linear_model/plot_iris_logistic.
↪html
from sklearn.linear_model import LinearRegression
linreg = LinearRegression()

# Fit the data.
X = iris_X.to_numpy()[:,:2]
Y = iris_y2
linreg.fit(X, Y)
```

[16]:
```
LinearRegression()
```

Ordinary least squares returns a decision boundary that is not unreasonable.

[17]:
```
# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, x_max]x[y_min, y_max].
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
h = .02  # step size in the mesh
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
Z = linreg.predict(np.c_[xx.ravel(), yy.ravel()])
Z[Z>0.5] = 1
Z[Z<0.5] = 0

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)

# Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=Y, edgecolors='k', cmap=plt.cm.Paired, s=60)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')

plt.show()
```



However, applying ordinary least squares is problematic for a few reasons. * There is nothing to prevent outputs larger than one or smaller than zero, which is conceptually wrong * We also don't have optimal performance: at least one point is misclassified, and others are too close to the decision boundary.

13 The Logistic Function

To address the fact that the output of linear regression is not in $[0, 1]$, we will instead attempt to *squeeze* it into that range using

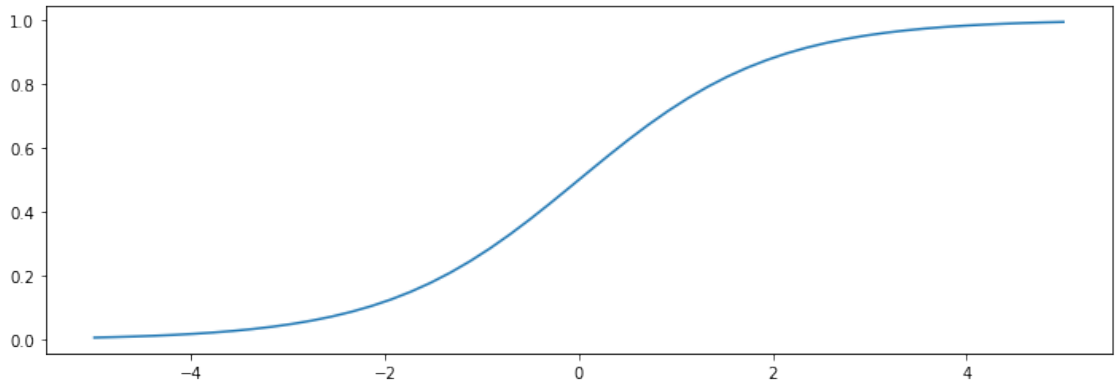
$$\sigma(z) = \frac{1}{1 + \exp(-z)}.$$

This is known as the *sigmoid* or *logistic* function.

The logistic function $\sigma : \mathbb{R} \to [0, 1]$ "squeezes" points from the real line into $[0, 1]$.

[11]:
```
import numpy as np
from matplotlib import pyplot as plt

def sigmoid(z):
    return 1/(1+np.exp(-z))

z = np.linspace(-5, 5)
plt.plot(z, sigmoid(z))
```

[11]:
```
[<matplotlib.lines.Line2D at 0x12c456cc0>]
```



14 The Logistic Function: Properties

The sigmoid function is defined as

$$\sigma(z) = \frac{1}{1 + \exp(-z)}.$$

A few observations:

- The function tends to 1 as $z \to \infty$ and tends to 0 as $z \to -\infty$.
- Thus, models of the form $\sigma(\theta^\top x)$ output values between 0 and 1, which is suitable for binary classification.
- It is easy to show that the derivative of $\sigma(z)$ has a simple form: $\frac{d\sigma}{dz} = \sigma(z)(1 - \sigma(z))$.

15 Logistic Regression: Model Class

Logistic regression is a classification algorithm which uses a model f_θ of the form

$$f_\theta(x) = \sigma(\theta^\top x) = \frac{1}{1 + \exp(-\theta^\top x)},$$

where

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

is the *sigmoid* or *logistic* function.

Note that logistic regression is actually a binary **classification** algorithm.

The term *regression* is an unfortunate historical misnomer.

Let's implement a logistic regression model in `numpy`.

[12]:
```
def f(X, theta):
    """The sigmoid model we are trying to fit.

    Parameters:
    theta (np.array): d-dimensional vector of parameters
    X (np.array): (n,d)-dimensional data matrix

    Returns:
    y_pred (np.array): n-dimensional vector of predicted targets
    """
    return sigmoid(X.dot(theta))
```

16 Probabilistic Interpretations

The logistic model can be interpreted to output a probability, and defines a conditional probability distribution as follows:

$$P_\theta(y = 1|x) = \sigma(\theta^\top x)$$
$$P_\theta(y = 0|x) = 1 - \sigma(\theta^\top x).$$

Recall that a probability over $y \in \{0, 1\}$ is called Bernoulli.

Part 3: Maximum Likelihood

In order to train a logistic regression model, we need to define an objective.

We derive this objective using the principle of maximum likelihood.

17 The Data Distribution

We will assume that our dataset is sampled from a probability distribution $\mathbb{P}$, which we will call the *data distribution*. We will denote this as

$$x, y \sim P_{\text{data}}.$$

The training set $\mathcal{D} = \{(x^{(i)}, y^{(i)}) \mid i = 1, 2, ..., n\}$ consists of *independent and identicaly distributed* (IID) samples from P_{data}.

18 Data Distribution: IID Sampling

The key assumption is that the training examples are *independent and identicaly distributed* (IID). * Each training example is from the same distribution. * This distribution doesn't depend on previous training examples.

Example: Flipping a coin. Each flip has same probability of heads & tails and doesn't depend on previous flips.

Counter-Example: Yearly census data. The population in each year will be close to that of the previous year.

19 Data Distribution: Example

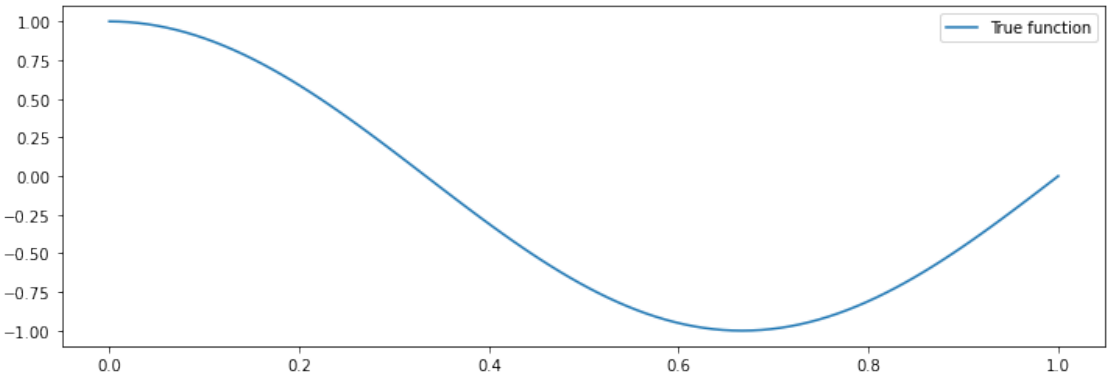
Let's implement an example of a data distribution in numpy.

[3]:
```
import numpy as np
np.random.seed(0)

def true_fn(X):
    return np.cos(1.5 * np.pi * X)
```

Let's visualize it.

[4]:
```
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [12, 4]

X_test = np.linspace(0, 1, 100)
plt.plot(X_test, true_fn(X_test), label="True function")
plt.legend()
```

[4]:
```
<matplotlib.legend.Legend at 0x11bb46b70>
```



Let's now draw samples from a data distribution. We will generate random x, and then generate random y using

$$y = f(x) + \epsilon$$

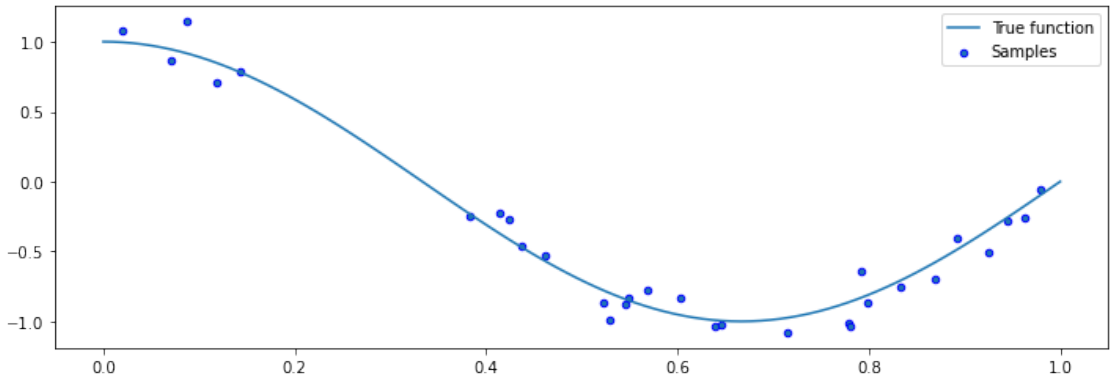
for a random noise variable ϵ.

[5]:
```
n_samples = 30

X = np.sort(np.random.rand(n_samples))
y = true_fn(X) + np.random.randn(n_samples) * 0.1
```

We can visualize the samples.

[6]:
```
plt.plot(X_test, true_fn(X_test), label="True function")
plt.scatter(X, y, edgecolor='b', s=20, label="Samples")
plt.legend()
```

[6]:
```
<matplotlib.legend.Legend at 0x11bd52438>
```



20 Data Distribution: Motivation

Why assume that the dataset is sampled from a distribution?

- The process we model may be effectively random. If y is a stock price, there is randomness in the market that cannot be captured by a deterministic model.
- There may be noise and randomness in the data collection process itself (e.g., collecting readings from an imperfect thermometer).
- We can use probability and statistics to analyze supervised learning algorithms and prove that they work.

21 Recall: Supervised Learning Models

A supervised learning model is a function

$$f_\theta : \mathcal{X} \to \mathcal{Y}$$

that maps inputs $x \in \mathcal{X}$ to targets $y \in \mathcal{Y}$.

Models have *parameters* $\theta \in \Theta$ living in a set Θ.

22 Probabilistic Interpretations

Many supervised learning models have a probabilistic interpretation. Often a model f_θ defines a probability distribution of the form

$$P_\theta(x, y) : \mathcal{X} \times \mathcal{Y} \to [0, 1] \qquad \text{or} \qquad P_\theta(y|x) : \mathcal{X} \times \mathcal{Y} \to [0, 1].$$

We refer to these as *probabilistic models*.

For example, our logistic model defines ("parameterizes") a probability distribution $P_\theta(y|x) : \mathcal{X} \times \mathcal{Y} \to [0, 1]$ as follows:

$$P_\theta(y = 1|x) = \sigma(\theta^\top x)$$
$$P_\theta(y = 0|x) = 1 - \sigma(\theta^\top x).$$

23 Maximum Likelihood

We can train any model that defines a probability distribution $P_\theta(x, y)$ by optimizing the *maximum likelihood* objective

$$L(\theta) = \prod_{i=1}^{n} P_\theta(x^{(i)}, y^{(i)})$$

defined over a dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(n)}, y^{(n)})\}$.

This asks that P_θ assign a high probability to the training instances in the dataset $\mathcal{D}$.

A lot of mathematical derivations and numerical calculations become simpler if we instead maximize the log of the likelihood:

$$\ell(\theta) = \log L(\theta) = \log \prod_{i=1}^{n} P_\theta(x^{(i)}, y^{(i)}) = \sum_{i=1}^{n} \log P_\theta(x^{(i)}, y^{(i)}).$$

Note that since the log is a monotonically increasing function, it doesn't change the optimal θ.

Finally, it's often simpler to take the average, instead of the sum. This gives us the following learning principle, known as *maximum log-likelihood*:

$$\max_\theta \ell(\theta) = \max_\theta \frac{1}{n} \sum_{i=1}^{n} \log P_\theta(x^{(i)}, y^{(i)}).$$

24 Example: Flipping a Random Coin

$ $ Consider a simple example in which we repeatedly toss a biased coin and record the outcomes.

- There are two possible outcomes: heads (H) and tails (T). A training dataset consists of tosses of the biased coin, e.g., $\mathcal{D} = \{H, H, T, H, T\}$
- Assumption: true probability distribution is $P_{\text{data}}(x)$, $x \in \{H, T\}$
- Our task is to determine the probability θ of seeing heads.

25 Example: Flipping a Random Coin

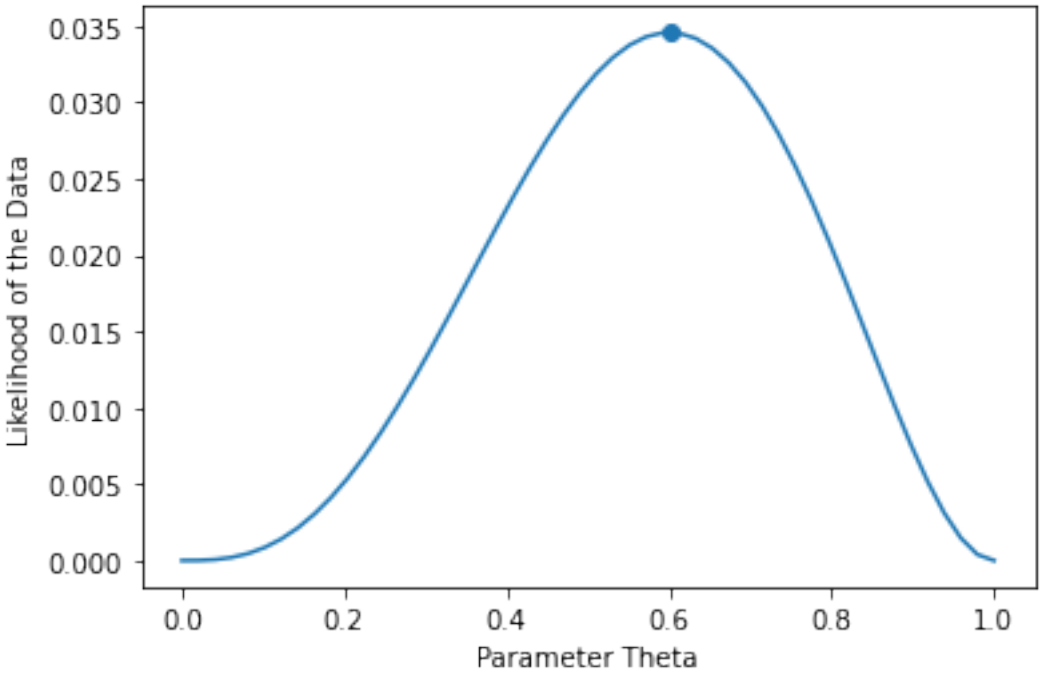
How should we choose θ if 3 out of 5 tosses are heads? Let's apply maximum likelihood learning.

- Our dataset is $\mathcal{D} = \{x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}, x^{(5)}\} = \{H, H, T, H, T\}$
- Our model is $P_\theta(x) = \theta$ if $x = H$ and $P_\theta(x) = 1 - \theta$ if $x = T$, and there is a single parameter $\theta \in [0, 1]$
- The likelihood of the data is $L(\theta) = \prod_{i=1}^{n} P_\theta(x^{(i)}) = \theta \cdot \theta \cdot (1-\theta) \cdot \theta \cdot (1-\theta)$.

We optimize for θ which makes $\mathcal{D}$ most likely. What is the solution in this case?

[12]:
```
%matplotlib inline
import numpy as np
from matplotlib import pyplot as plt

# our dataset is {H, H, T, H, T}; if theta = P(x=H), we get:
coin_likelihood = lambda theta: theta*theta*(1-theta)*theta*(1-theta)

theta_vals = np.linspace(0,1)
plt.ylabel('Likelihood of the Data')
plt.xlabel('Parameter Theta')
plt.scatter([0.6], [coin_likelihood(0.6)])
plt.plot(theta_vals, coin_likelihood(theta_vals))
```

[12]:
```
[<matplotlib.lines.Line2D at 0x11ae41208>]
```



The likelihood $L(\theta)$ is maximized by $\theta = 0.6$, which is also what we expect intuitively since 3/5

tosses are heads.

26 Example: Flipping a Random Coin

Our log-likelihood function is

$$\begin{aligned} L(\theta) &= \theta^{\#\text{ heads}} \cdot (1-\theta)^{\#\text{ tails}} \\ \log L(\theta) &= \log(\theta^{\#\text{ heads}} \cdot (1-\theta)^{\#\text{ tails}}) \\ &= \#\text{ heads} \cdot \log(\theta) + \#\text{ tails} \cdot \log(1-\theta) \end{aligned}$$

The maximum likelihood estimate is the $\theta^* \in [0,1]$ such that $\log L(\theta^*)$ is maximized.

Differentiating the log-likelihood function with respect to θ and setting the derivative to zero, we obtain

$$\theta^* = \frac{\#\text{ heads}}{\#\text{ heads} + \#\text{ tails}}$$

When exact solutions are not available, we can optimize the log likelihood numerically, e.g. using gradient descent.

We will see examples of this later.

27 Conditional Maximum Likelihood

Often, a machine learning model defines a *conditional* model $P_\theta(y|x)$. In this setting, we can maximize the *conditional maximum likelihood*:

$$\max_\theta \frac{1}{n} \sum_{i=1}^{n} \log P_\theta(y^{(i)}|x^{(i)}).$$

This asks that for each input $x^{(i)}$ in the dataset $\mathcal{D}$, P_θ should assign a high probability to the correct target $y^{(i)}$.

28 KL Divergences and Max Likelihood

Maximizing likelihood is closely related to minimizing the Kullback-Leibler (KL) divergence $D(\cdot\|\cdot)$ between the model distribution and the data distribution.

$$D(p\|q) = \sum_{\mathbf{x}} p(\mathbf{x}) \log \frac{p(\mathbf{x})}{q(\mathbf{x})}.$$

The KL divergence is always non-negative, and equals zero when p and q are identical. This makes it a natural measure of similarity that's useful for comparing distributions.

Let $\hat{P}(x,y)$ denote the *empirical* distribution of the data:

$$\hat{P}(x,y) = \begin{cases} \frac{1}{n} & \text{if } (x,y) \in \mathcal{D} \\ 0 & \text{otherwise.} \end{cases}$$

This distribution assigns a probability of $1/n$ to each of the data points in $\mathcal{D}$ (and zero to all the other possible (x, y)); it can be seen as a guess of the true data distribution from which the dataset $\mathcal{D}$ was obtained.

Selecting parameters θ by maximizing the likelihood is equivalent to selecting θ that minimizes the KL divergence between the empirical data distribution and the model distribution:

$$\begin{aligned}\max_\theta \frac{1}{n}\sum_{i=1}^{n} \log P_\theta(x^{(i)}, y^{(i)}) &= \max_\theta \mathbb{E}_{\hat{P}(x,y)}\left[\log P_\theta(x, y)\right] \\ &= \min_\theta KL(\hat{P}(x, y)||P_\theta(x, y)).\end{aligned}$$

Here, $\mathbb{E}_{p(x)}f(x)$ denotes $\sum_{x\in\mathcal{X}} f(x)p(x)$ if x is discrete and $\int_{x\in\mathcal{X}} f(x)p(x)dx$ if x is continuous.

The same is true for conditional log-likelihood (homework problem!):

$$\max_\theta \mathbb{E}_{\hat{P}(x,y)}\left[\log P_\theta(y|x)\right] = \min_\theta \mathbb{E}_{\hat{P}(x)}\left[KL(\hat{P}(y|x)||P_\theta(y|x))\right].$$

29 Recall: Ordinary Least Squares

Recall that in ordinary least squares (OLS), we have a linear model of the form

$$f(x) = \sum_{j=0}^{d} \theta_j \cdot x_j = \theta^\top x.$$

At each training instance (x, y), we seek to minimize the squared error

$$(y - \theta^\top x)^2.$$

30 Maximum Likelihood Interpretation of OLS

Let's make our usual linear regression model probabilistic: assume that the targets and the inputs are related by

$$y = \theta^\top x + \epsilon,$$

where $\epsilon \sim \mathcal{N}(0, \sigma^2)$ is a random noise term that follows a Gaussian (or "Normal") distribution.

The density of $\epsilon \sim \mathcal{N}(0, \sigma^2)$ is a Gaussian distribution:

$$P(\epsilon; \sigma) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{\epsilon^2}{2\sigma^2}\right).$$

This implies that

$$P(y|x; \theta) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(y - \theta^\top x)^2}{2\sigma^2}\right).$$

This is a Gaussian distribution with mean $\mu_\theta(x) = \theta^\top x$ and variance σ^2.

Given an input of x, this model outputs a "mini Bell curve" with width σ around the mean $\mu(x) = \sigma^\top x$.

Let's now learn the parameters θ of

$$P(y|x;\theta) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y-\theta^\top x)^2}{2\sigma^2}\right)$$

using maximum likelihood.

The log-likelihood of this model at a point (x, y) equals

$$\log L(\theta) = \log p(y|x;\theta) = \text{const}_1 \cdot (y-\theta^\top x)^2 + \text{const}_2$$

for some constants $\text{const}_1, \text{const}_2$. But that's just the least squares objective!

Least squares thus amounts to fitting a Gaussian model $\mathcal{N}(y; \mu(x), \sigma)$ with a standard deviation σ of one and a mean of $\mu(x) = \theta^\top x$.

Note in particular that OLS implicitly makes the assumption that the noise is Gaussian around the mean.

When that's not the case, you may want to also experiment with other kinds of models.

31 Part 4: Learning in Logistic Regression

Next, we will use maximum likelihood to learn the parameters of a logistic regression model.

32 Logistic Regression

Recall that a logistic model defines ("parameterizes") a probability distribution $P_\theta(y|x) : \mathcal{X} \times \mathcal{Y} \to [0, 1]$ as follows:

$$P_\theta(y = 1|x) = \sigma(\theta^\top x)$$
$$P_\theta(y = 0|x) = 1 - \sigma(\theta^\top x).$$

When $y \in \{0, 1\}$, can write this more compactly as

$$P_\theta(y|x) = \sigma(\theta^\top x)^y \cdot (1 - \sigma(\theta^\top x))^{1-y}$$

33 Applying Maximum Lilkelihood

Following the principle of maximum likelihood, we want to optimize the following objective defined over a binary classification dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(n)}, y^{(n)})\}$.

$$\begin{aligned}\ell(\theta) &= \frac{1}{n}\sum_{i=1}^{n} \log P_\theta(y^{(i)} \mid x^{(i)}) \\ &= \frac{1}{n}\sum_{i=1}^{n} \log \sigma(\theta^\top x^{(i)})^{y^{(i)}} \cdot (1 - \sigma(\theta^\top x^{(i)}))^{1-y^{(i)}} \\ &= \frac{1}{n}\sum_{i=1}^{n} y^{(i)} \cdot \log \sigma(\theta^\top x^{(i)}) + (1 - y^{(i)}) \cdot \log(1 - \sigma(\theta^\top x^{(i)})).\end{aligned}$$

This objective is also often called the log-loss, or cross-entropy.

Observe that this objective asks the model to ouput a large score $\sigma(\theta^\top x^{(i)})$ (a score that's close to one) if $y^{(i)} = 1$, and a score that's small (close to zero) if $y^{(i)} = 0$.

Let's implement the log-likelihood objective.

[13]:
```
def log_likelihood(theta, X, y):
    """The cost function, J(theta0, theta1) describing the goodness of fit.

    We added the 1e-6 term in order to avoid overflow (inf and -inf).

    Parameters:
    theta (np.array): d-dimensional vector of parameters
    X (np.array): (n,d)-dimensional design matrix
    y (np.array): n-dimensional vector of targets
    """
    return (y*np.log(f(X, theta) + 1e-6) + (1-y)*np.log(1-f(X, theta) + 1e-6)).
↪mean()
```

34 Review: Gradient Descent

If we want to minimize an objective $J(\theta)$, we may start with an initial guess θ_0 for the parameters and repeat the following update:

$$\theta_i := \theta_{i-1} - \alpha \cdot \nabla_\theta J(\theta_{i-1}).$$

35 Gradient of the Log-Likelihood

We want to use gradient descent to maximize the log-likelihood, hence our objective is $J(\theta) = -\ell(\theta)$.

We can show that the gradient of the negative log-likelihood equals:

$$\nabla_\theta J(\theta) = \nabla_\theta \left[-\ell(\theta)\right] = \Big(\sigma(\theta^\top x) - y\Big) \cdot \mathbf{x}.$$

Interestingly, this expression looks similar to the gradient of the mean squared error, which we derived in the previous lecture.

Let's implement the gradient.

[14]:
```
def loglik_gradient(theta, X, y):
    """The cost function, J(theta0, theta1) describing the goodness of fit.

    Parameters:
    theta (np.array): d-dimensional vector of parameters
    X (np.array): (n,d)-dimensional design matrix
    y (np.array): n-dimensional vector of targets

    Returns:
    grad (np.array): d-dimensional gradient of the MSE
    """
    return np.mean((f(X, theta)-y) * X.T, axis=1)
```

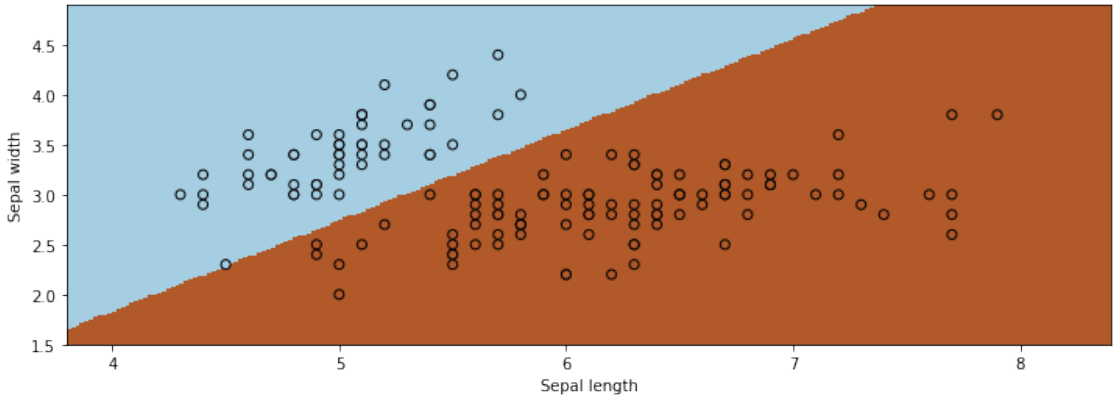
Let's now implement gradient descent.

[15]:
```
threshold = 5e-5
step_size = 1e-1

theta, theta_prev = np.zeros((3,)), np.ones((3,))
opt_pts = [theta]
opt_grads = []
iter = 0
iris_X['one'] = 1
X_train = iris_X.iloc[:,[0,1,-1]].to_numpy()
y_train = iris_y2.to_numpy()

while np.linalg.norm(theta - theta_prev) > threshold:
    if iter % 50000 == 0:
        print('Iteration %d. Log-likelihood: %.6f' % (iter,␣
 ↪log_likelihood(theta, X_train, y_train)))
    theta_prev = theta
    gradient = loglik_gradient(theta, X_train, y_train)
    theta = theta_prev - step_size * gradient
    opt_pts += [theta]
    opt_grads += [gradient]
    iter += 1
```

```
Iteration 0. Log-likelihood: -0.693145
Iteration 50000. Log-likelihood: -0.021506
Iteration 100000. Log-likelihood: -0.015329
Iteration 150000. Log-likelihood: -0.012062
Iteration 200000. Log-likelihood: -0.010076
```

Let's now visualize the result.

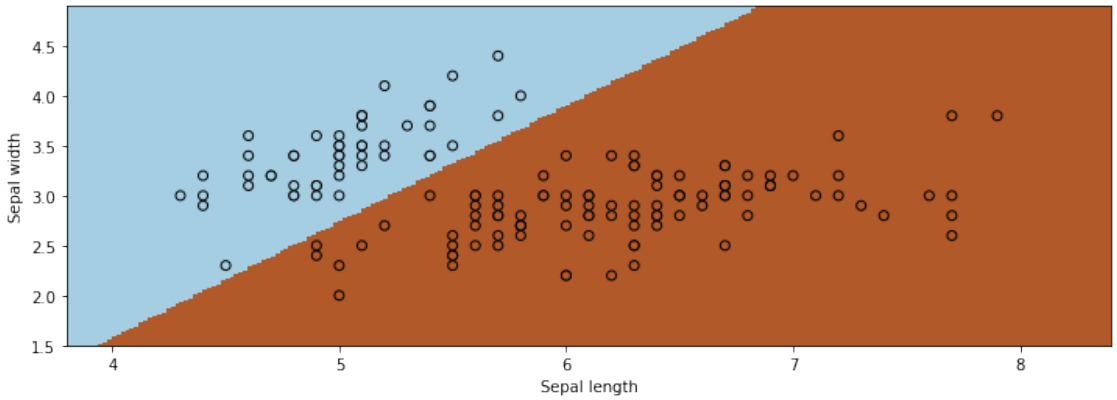
[16]:
```
xx, yy = np.meshgrid(np.arange(x_min, x_max, .02), np.arange(y_min, y_max, .02))
Z = f(np.c_[xx.ravel(), yy.ravel(), np.ones(xx.ravel().shape)], theta)
Z[Z<0.5] = 0
Z[Z>=0.5] = 1

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)

# Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=Y, edgecolors='k', cmap=plt.cm.Paired)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')

plt.show()
```



This is how we would use the algorithm via `sklearn`.

[17]:
```
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression(C=1e5, fit_intercept=True)

# Create an instance of Logistic Regression Classifier and fit the data.
X = iris_X.to_numpy()[:,:2]
Y = iris_y2
logreg.fit(X, Y)

xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, .02))
Z = logreg.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)
```

```
# Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=Y, edgecolors='k', cmap=plt.cm.Paired)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')

plt.show()
```



36 Observations About Logistic Regression

- Logistic regression finds a linear decision boundary. This is the set of points for which $P(y = 1|x) = P(y = 0|x)$, or equivalently:

$$0 = \log \frac{P(y=1|x)}{P(y=0|x)} = \log \frac{\frac{1}{1+\exp(-\theta^\top x)}}{1 - \frac{1}{1+\exp(-\theta^\top x)}} = \theta^\top x$$

  The set of x for which $0 = \theta^\top x$ is a linear surface.

- Unlike least squares, we don't have a closed form solution (a formula) for the optimal θ. We can nonetheless find it numerically via gradient descent.

37 Algorithm: Logistic Regression

- **Type**: Supervised learning (binary classification)
- **Model family**: Linear decision boundaries.
- **Objective function**: Cross-entropy, a special case of log-likelihood.
- **Optimizer**: Gradient descent.
- **Probabilistic interpretation**: Parametrized Bernoulli distribution.

38 Part 5: Multi-Class Classification

Finally, let's look at an extension of logistic regression to an arbitrary number of classes.

39 Multi-Class Classification

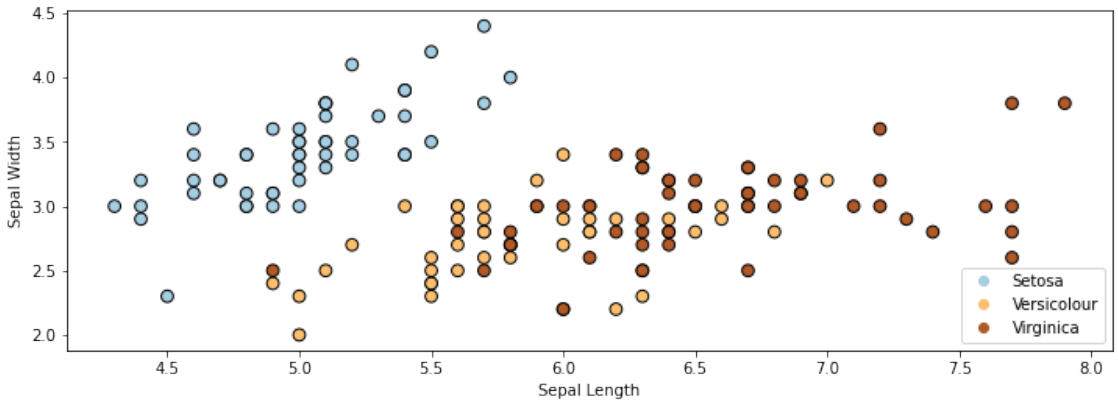
Logistic regression only applies to binary classification problems. What if we have an arbitrary number of classes K?

- The simplest approach that can be used with any machine learning algorithm is the "one vs. all" approach. We train one classifer for each class to distinguish that class from all the others.
- This works, but is not very elegant.
- Alternatively, we may fit a probabilistic model that outputs multi-class probabilities.

Let's load a fully multiclass version of the Iris dataset.

[18]:
```
# https://scikit-learn.org/stable/auto_examples/neighbors/plot_classification.
↪html
# Plot also the training points
p1 = plt.scatter(iris_X.iloc[:, 0], iris_X.iloc[:, 1], c=iris_y,
            edgecolor='k', s=60, cmap=plt.cm.Paired)
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.legend(handles=p1.legend_elements()[0], labels=['Setosa', 'Versicolour',␣
↪'Virginica'], loc='lower right')
```

[18]:
```
<matplotlib.legend.Legend at 0x12fbd4f60>
```



40 The Softmax Function

The logistic function $\sigma : \mathbb{R} \to [0, 1]$ "squeezes" the score $z \in \mathbb{R}$ of a class into a probability in $[0, 1]$.

The *softmax* function $\vec{\sigma} : \mathbb{R}^K \to [0, 1]^K$ is a multi-class version of σ * It takes in a K-dimensional *vector* of class scores $\vec{z} \in \mathbb{R}$ * It "squeezes" $\vec{z}$ into a length K *vector* of probabilities in $[0, 1]^K$

The k-th component of the output of the softmax function $\vec{\sigma}$ is defined as

$$\sigma(\vec{z})_k = \frac{\exp(z_k)}{\sum_{l=1}^{K} \exp(z_l)}.$$

Softmax takes a vector of scores $\vec{z}$, exponentiates each score z_k, and normalizes the exponentiated scores such that they sum to one.

When $K = 2$, this looks as follows:

$$\sigma(\vec{z})_1 = \frac{\exp(z_1)}{\exp(z_1) + \exp(z_2)}.$$

Observe that adding a constant $c \in \mathbb{R}$ to each score z_k doesn't change the output of softmax, e.g.:

$$\frac{\exp(z_1)}{\exp(z_1) + \exp(z_2)} = \frac{\exp(z_1 + c)}{\exp(z_1 + c) + \exp(z_2 + c)}.$$

Without loss of generality, we can assume $z_1 = 0$. For any $\vec{z} = (z_1, z_2)$, we can define $\vec{z}' = (0, z_2') = (0, z_2 - z_1)$ such that $\vec{\sigma}(\vec{z}) = \vec{\sigma}(\vec{z}')$. Assuming $z_1 = 0$ doesn't change the probabilities that $\vec{\sigma}$ can output.

Assuming that $z_1 = 0$ means that $\exp(z_1) = 1$ and softmax becomes

$$\sigma(\vec{z})_1 = \frac{1}{1 + \exp(z_2)}.$$

This is effectively our sigmoid function. Hence softmax generalizes the sigmoid function.

41 Recall: Logistic Regression

Logistic regression is a classification algorithm which uses a model f_θ of the form

$$f_\theta(x) = \sigma(\theta^\top x) = \frac{1}{1 + \exp(-\theta^\top x)},$$

where

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

is the *sigmoid* or *logistic* function. It trains this model using maximum likelihood.

42 Softmax Regression: Model Class

Softmax regression is a multi-class classification algorithm which uses a model $f_\theta : \mathcal{X} \to [0, 1]^K$ that generalizes logistic regression.

Softmax regression works as follows: 1. Given an input x, we compute K scores, one per class. The score

$$z_k = \theta_k^\top x$$

of class k is a linear function of x and parameters θ_k for class k

2. We "squeeze" the vector of scores $\vec{z}$ into $[0,1]^K$ using the softmax function $\vec{\sigma}$ and we output $\vec{\sigma}(\vec{z})$, a vector of K probabilities.

The parameters of this model are $\theta = (\theta_1, \theta_2, ..., \theta_K)$, and the parameter space is $\Theta = \mathbb{R}^{K \times d}$.

The output of the model is a *vector* of class membership probabilities, whose k-th component $f_\theta(x)_k$ is

$$f_\theta(x)_k = \sigma(\theta_k^\top x)_k = \frac{\exp(\theta_k^\top x)}{\sum_{l=1}^K \exp(\theta_l^\top x)},$$

where each $\theta_l \in \mathbb{R}^d$ is the vector of parameters for class ℓ and $\theta = (\theta_1, \theta_2, ..., \theta_K)$.

This model is again over-parametrized: adding a constant $c \in \mathbb{R}$ to every score $\theta_k^\top x$ does not change the output of the model.

As before, we can assume without loss of generality that $z_1 = 0$ (or equivalently that $\theta_1 = 0$). This doesn't change the set of functions $\mathcal{X} \to [0,1]^K$ that our model class can represent.

Note again that softmax regression is actually a **classification** algorithm.

The term *regression* is an unfortunate historical misnomer.

43 Softmax Regression: Probabilistic Interpretation

The softmax model outputs a vector of probabilities, and defines a conditional probability distribution as follows:

$$P_\theta(y = k|x) = \vec{\sigma}(\vec{z})_k = \frac{\exp(\theta_k^\top x)}{\sum_{l=1}^K \exp(\theta_l^\top x)}.$$

Recall that a probability over $y \in \{1, 2, ..., K\}$ is called Categorical.

44 Softmax Regression: Learning Objective

We again maximize likelihood over a dataset $\mathcal{D}$.

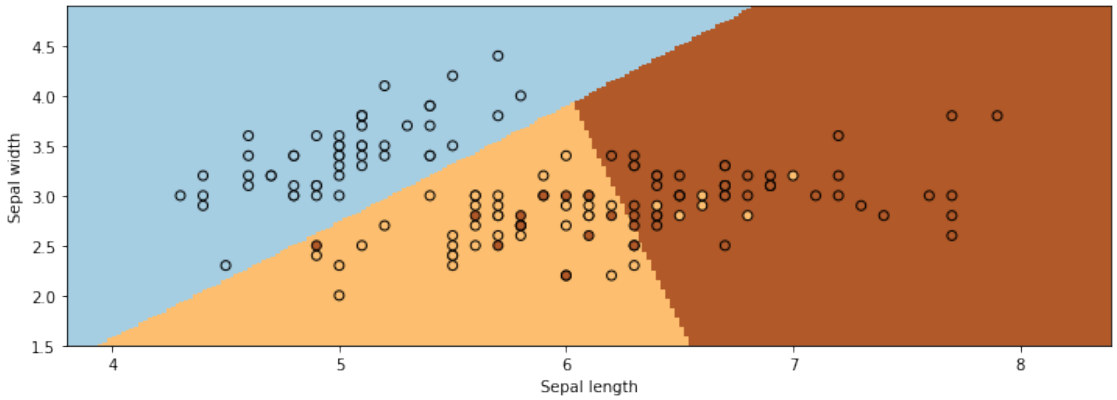
$$\begin{aligned} L(\theta) &= \prod_{i=1}^n P_\theta(y^{(i)} \mid x^{(i)}) = \prod_{i=1}^n \vec{\sigma}(\vec{z}^{(i)})_{y^{(i)}} \\ &= \prod_{i=1}^n \left(\frac{\exp(\theta_{y^{(i)}}^\top x^{(i)})}{\sum_{l=1}^K \exp(\theta_l^\top x^{(i)})} \right). \end{aligned}$$

We optimize this using gradient descent.

Let's now apply softmax regression to the Iris dataset by using the implementation from `sklearn`.

[19]:
```
# https://scikit-learn.org/stable/auto_examples/linear_model/plot_iris_logistic.
↪html
```

```python
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression(C=1e5, multi_class='multinomial')

# Create an instance of Softmax and fit the data.
logreg.fit(X, iris_y)
Z = logreg.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)

# Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=iris_y, edgecolors='k', cmap=plt.cm.Paired)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')

plt.show()
```



45 Algorithm: Softmax Regression

- **Type**: Supervised learning (multi-class classification)
- **Model family**: Linear decision boundaries.
- **Objective function**: Softmax loss, a special case of log-likelihood.
- **Optimizer**: Gradient descent.
- **Probabilistic interpretation**: Parametrized categorical distribution.