Lecture 6: Classification Algorithms (Supervised learning)

Adapted from Applied Machine Learning Lecture Notes of Volodymyr Kuleshov, Cornel Tech

Instructor Tan Bui

Part 1: Classification

So far, every supervised learning algorithm that we've seen has been an instance of regression. Next, let's look at some classification algorithms. First, we will define what classification is.

Review: Components of A Supervised Machine Learning Problem

At a high level, a supervised machine learning problem has the following structure:

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

Regression vs. Classification

Consider a training dataset $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, (\mathbf{x}^{(n)}, y^{(n)})\}$.
We distinguish between two types of supervised learning problems depending 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.

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.

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.

In [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 ...

In [2]:

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

Out[2]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

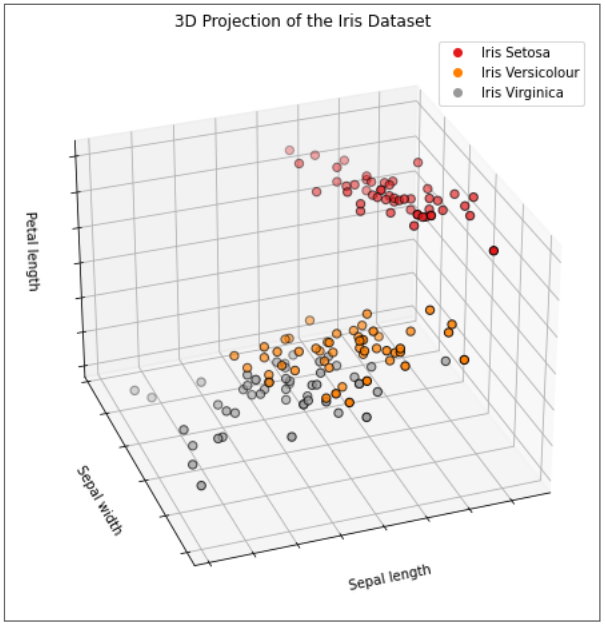
Here is a visualization of this dataset in 3D. Note that we are using the first 3 features (out of 4) in this dateset.

In [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

# let's visualize this dataset

fig = plt.figure(1, figsize=(8, 6))
ax = Axes3D(fig, elev=-150, azim=110)
# X_reduced = PCA(n_components=3).fit_transform(iris_X)
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'])
```

Out[3]:

```
<matplotlib.legend.Legend at 0x129076cc0>
```


3D Projection of the Iris Dataset
Iris Setosa
Iris Versicolour
Iris Virginica
Petal length
Sepal width
Sepal length

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 have a simple probabilistic interpretation: they are probabilities that a data point belongs to a given class.

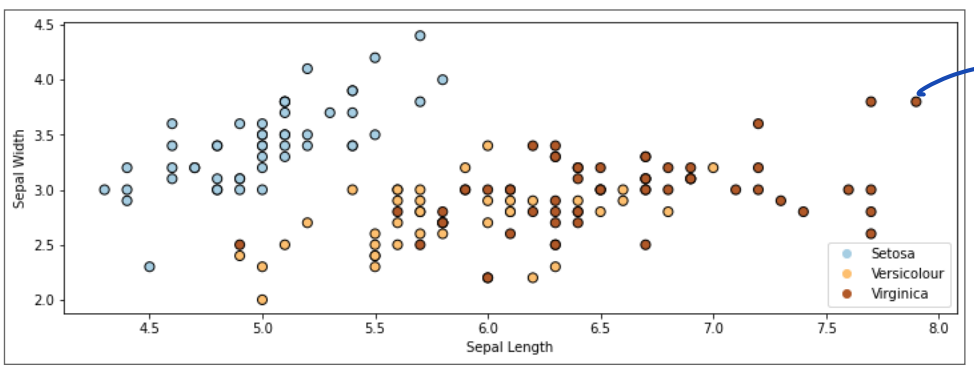
Let's visualize our Iris dataset to see this. Note that we are using the first 2 features in this dateset.

In [4]:

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

# Plot also the training points
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')
```

Out[4]:

```
<matplotlib.legend.Legend at 0x1292f7a20>
```

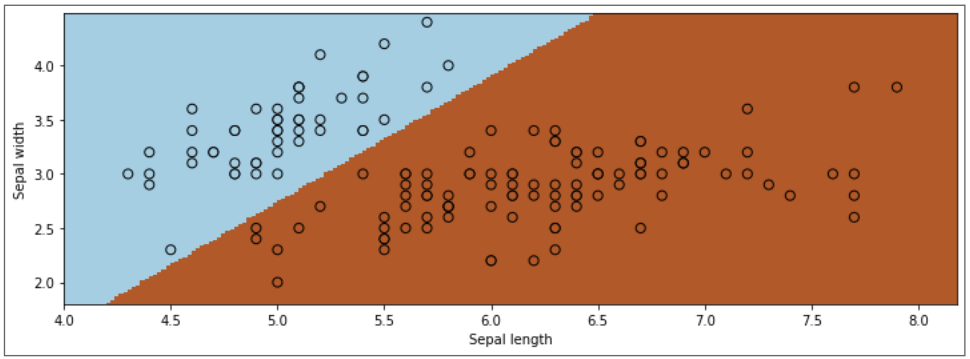


Virginica

Let's train a classification algorithm on this data.
Below, we see the regions predicted to be associated with the blue and non-blue classes and the line between them in the decision boundary.

In [5]:

```
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
iris_y2 = iris_y.copy()
iris_y2[iris_y2==2] = 1
Y = iris_y2
logreg.fit(X, Y)

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()
```


Sepal width
Sepal length
4.0
3.5
3.0
2.5
2.0
4.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0

Part 2: Nearest Neighbors (Non-Parametric and Lazy Learning Algorithm)

Previously, we have seen what defines a classification problem. Let's now look at our first classification algorithm.

A Simple Classification Algorithm: Nearest Neighbors

Suppose we are given a training dataset $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, (\mathbf{x}^{(n)}, y^{(n)})\}$. At inference time, we receive a query point $\mathbf{x}'$ and we want to predict its label y'.

A really simple but suprisingly effective way of returning y' is the *nearest neighbors* approach.

- Given a query datapoint $\mathbf{x}'$, find the training example $(\mathbf{x}, y)$ in $\mathcal{D}$ that's closest (**in which sense**?) to $\mathbf{x}'$, in the sense that $\mathbf{x}$ is "nearest" to x'
- Return y, the label of the "nearest neighbor" $\mathbf{x}$.

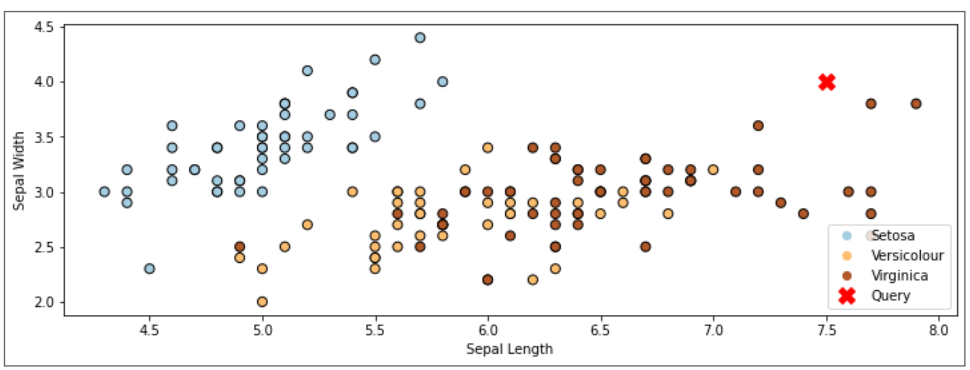
In the example below on the Iris dataset, the red cross denotes the query $\mathbf{x}'$. The closest class to it is "Virginica". (We're only using the first two features in the dataset for simplicity.)

In [6]:

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

# Plot also the training points
p1 = plt.scatter(iris_X.iloc[:, 0], iris_X.iloc[:, 1], c=iris_y,
            edgecolor='k', s=50, cmap=plt.cm.Paired)
p2 = plt.plot([7.5], [4], 'rx', ms=10, mew=5)
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.legend(['Query Point', 'Training Data'], loc='lower right')
plt.legend(handles=p1.legend_elements()[0]+p2, labels=['Setosa', 'Versicolour', 'Virginica', 'Query'], loc='lower right'
```

Out[6]:

```
<matplotlib.legend.Legend at 0x12982b4e0>
```



Choosing a Distance Function (sense/measure of nearest)

How do we select the point $\mathbf{x}$ that is the closest to the query point $\mathbf{x}'$? There are many options:

- The Euclidean distance $||\mathbf{x} - \mathbf{x}'||_2 = \sqrt{\sum_{j=1}^{d} |x_j - x'_j|^2)}$ is a popular choice.
- The Minkowski distance $||\mathbf{x} - \mathbf{x}'||_p = (\sum_{j=1}^{d} |x_j - x'_j|^p)^{1/p}$ generalizes the Euclidean, L1 and other distances.
- The Mahalanobis distance $\sqrt{\mathbf{x}^\top \mathbf{V} \mathbf{x}}$ for a positive semidefinite matrix $\mathbf{V} \in \mathbb{R}^{d \times d}$ also generalizes the Euclidean distnace.
- Discrete-valued inputs can be examined via the Hamming distance $|\{j : x_j \neq x'_j\}|$ and other distances.

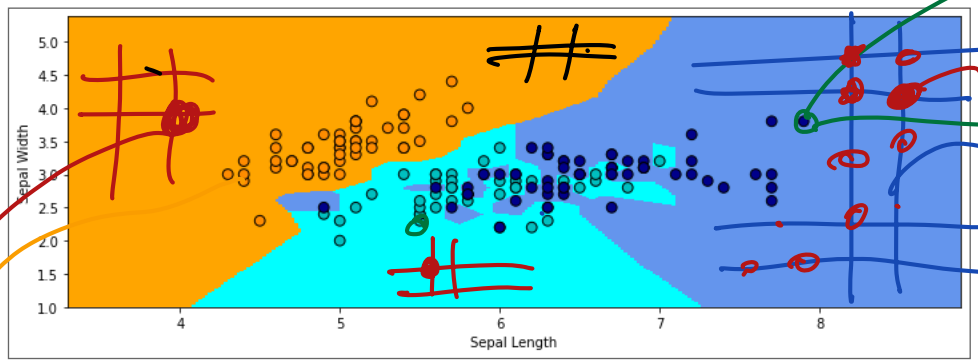
Let's apply Nearest Neighbors to the above dataset using the Euclidean distance (or equiavalently, Minkowski with $p = 2$)

In [23]:

```
# https://scikit-learn.org/stable/auto_examples/neighbors/plot_classification.html
from sklearn import neighbors
from matplotlib.colors import ListedColormap

# Train a Nearest Neighbors Model
clf = neighbors.KNeighborsClassifier(n_neighbors=1, metric='minkowski', p=2)
clf.fit(iris_X.iloc[:,:2], iris_y)

# Create color maps
cmap_light = ListedColormap(['orange', 'cyan', 'cornflowerblue'])
cmap_bold = ListedColormap(['darkorange', 'c', 'darkblue'])

# 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 = iris_X.iloc[:, 0].min() - 1, iris_X.iloc[:, 0].max() + 1
y_min, y_max = iris_X.iloc[:, 1].min() - 1, iris_X.iloc[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                     np.arange(y_min, y_max, 0.02))
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

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

# Plot also the training points
plt.scatter(iris_X.iloc[:, 0], iris_X.iloc[:, 1], c=iris_y, cmap=cmap_bold,
            edgecolor='k', s=60)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())

plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
```

Out[23]:

```
Text(0, 0.5, 'Sepal Width')
```



$x_{closest}$

x'

nearest to x'

$y' = y_{closest}$

||

Virginica

0 : are data D

Setosa

$\rightarrow x' \Rightarrow y' =$ Setosa

In the above example, the regions of the 2D space that are assigned to each class are highly irregular. In areas where the two classes overlap, the decision of the boundary flips between the classes, depending on which point is closest to it.

typical for non-separable cases

K-Nearest Neighbors

$K=1 \Rightarrow$ nearest neighbor

Intuitively, we expect the true decision boundary to be smooth. Therefore, we average K nearest neighbors at a query point.

- Given a query datapoint $\mathbf{x}'$, find the K training examples $\mathcal{N} = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, (\mathbf{x}^{(K)}, y^{(K)})\} \subseteq D$ that are closest to $\mathbf{x}'$.
- Return $y_{\mathcal{N}}$, the consensus label of the neighborhood $\mathcal{N}$.

The consesus $y_{\mathcal{N}}$ can be determined by voting (take the label of the label with the highest counts), weighted average (for regression), etc.

$K = 5$: $\mathcal{N}(x') = \{B, S, B, S, B\}$

$\Downarrow$ voting

$y' = B$

$\Rightarrow$ to avoid tie cases: if we have c classes, pick K that is <u>not</u> a multiple of c

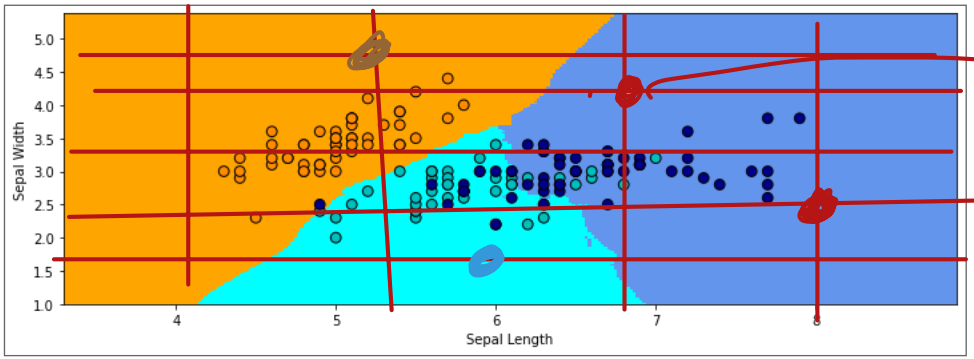
Let's look at Nearest Neighbors with a neighborhood of 30. The decision boundary is much smoother than before.

In [8]:

```
# https://scikit-learn.org/stable/auto_examples/neighbors/plot_classification.html
# Train a Nearest Neighbors Model
clf = neighbors.KNeighborsClassifier(n_neighbors=30, metric='minkowski', p=2)
clf.fit(iris_X.iloc[:,:2], iris_y)
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

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

# Plot also the training points
plt.scatter(iris_X.iloc[:, 0], iris_X.iloc[:, 1], c=iris_y, cmap=cmap_bold,
            edgecolor='k', s=60)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())

plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
```

Out[8]:

```
Text(0, 0.5, 'Sepal Width')
```


Sepal Width
Sepal Length
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
4
5
6
7
x'

What is the best K?

1. Small K: fast but unstable with rough boundaries. Typically does not generalize well
2. Large K: slow, but smooth boundaries. Very large K leads to large generalization error again.
3. "Best K": expensive as we have to carry out the predictions for different K and pick K with smallest prediction errors.
4. Pick odd K to have tiebreaker for binary

Review: Data Distribution

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

$$\mathbf{x}, y \sim \mathbb{P}.$$

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

An interpretation of KNN Estimates with average prediction

Suppose that the output y' of KNN is the average target in the neighborhood $\mathcal{N}(\mathbf{x}')$ around the query $\mathbf{x}'$. Observe that we can write:

$$y' = \boxed{\frac{1}{K} \sum_{(x,y) \in \mathcal{N}(\mathbf{x}')} y} \approx \mathbb{E}[y \mid \mathbf{x}'].$$

- When $\mathbf{x} \approx \mathbf{x}'$ and when $\mathbb{P}$ is reasonably smooth, each y for $(\mathbf{x}, y) \in \mathcal{N}(\mathbf{x}')$ is approximately a sample from $\mathbb{P}(y \mid \mathbf{x}')$ (since $\mathbb{P}$ doesn't change much around $\mathbf{x}'$, $\mathbb{P}(y \mid \mathbf{x}') \approx \mathbb{P}(y \mid \mathbf{x})$).
- Thus y' is essentially a Monte Carlo estimate of $\mathbb{E}[y \mid \mathbf{x}']$ (the average of K samples from $\mathbb{P}(y \mid \mathbf{x}')$).

B = 3
S = 2
U = 1

2.1 = y' ∈ {1, 2, 3}

Set to 2

can be understood as a MC approximation of $\mathbb{E}[y' \mid x']$

Algorithm: K-Nearest Neighbors

- **Type**: Supervised learning (regression and classification)
- **Model family**: Consensus over K training instances.
- **Objective function**: Euclidean, Minkowski, Hamming, etc.
- **Optimizer**: Non at training. Nearest neighbor search at inference using specialized search algorithms (Hashing, KD-trees).
- **Probabilistic interpretation**: Directly approximating the density $P_{\text{data}}(y|\mathbf{x})$.

Pros and Cons of KNN

Pros:

- Can approximate any data distribution arbtrarily well.

Cons:

- Need to store entire dataset to make queries, which is computationally prohibitive.
- Number of data needed scale exponentially with dimension ("curse of dimensionality").

previously: $f_{\vec{\theta}}(\vec{x}) = \vec{\theta}^T \vec{x}$

↓ parametric model

Part 3: Non-Parametric Models

Nearest neighbors is the first example of an important type of machine learning algorithm called a non-parametric model.

Review: Supervised Learning Model

We'll say that a model is a function

$$f : \mathcal{X} \to \mathcal{Y}$$

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

Often, models have *parameters* $\boldsymbol{\theta} \in \Theta$ living in a set Θ. We will then write the model as

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

to denote that it's parametrized by $\boldsymbol{\theta}$.

Review: K-Nearest Neighbors

Suppose we are given a training dataset $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, (\mathbf{x}^{(n)}, y^{(n)})\}$. At inference time, we receive a query point $\mathbf{x}'$ and we want to predict its label y'.

- Given a query datapoint $\mathbf{x}'$, find the K training examples $\mathcal{N} = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, (\mathbf{x}^{(K)}, y^{(K)})\} \subseteq D$ that are closest to $\mathbf{x}'$.
- Return $y_{\mathcal{N}}$, the consensus label of the neighborhood $\mathcal{N}$.

The consesus $y_{\mathcal{N}}$ can be determined by voting, weighted average, etc.

Non-Parametric Models

Nearest neighbors is an example of a *non-parametric* model. Parametric vs. non-parametric are is a key distinguishing characteristic for machine learning models.

A parametric model $f_\theta(x) : \mathcal{X} \times \Theta \to \mathcal{Y}$ is defined by a finite set of parameters $\boldsymbol{\theta} \in \Theta$ whose dimensionality is constant with respect to the dataset. Linear models of the form

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}^\top \mathbf{x}$$

are an example of a parametric model. **The prediction does not use the training dataset**.

In a non-parametric model, **the function f uses the entire training dataset** (or a post-proccessed version of it) to make predictions, as in K-Nearest Neighbors. In other words, the complexity of the model increases with dataset size.

Non-parametric models have the advantage of not loosing any information at training time. However, they are also computationally less tractable and may easily overfit the training set.

Part 4: Logistic Regression

Next, we are going to see a simple parametric classification algorithm that addresses many of these limitations of Nearest Neighbors.

Review: Classification

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

Binary Classification and the Iris Dataset

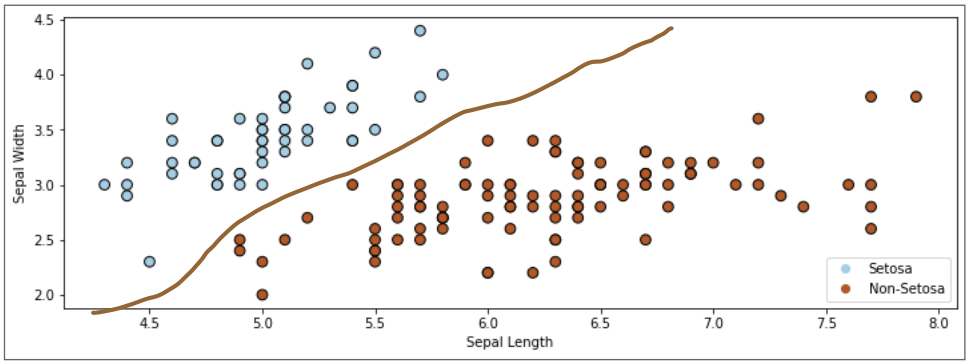
We are going to start by looking at binary (two-class) classification.
To keep things simple, we will use the Iris dataset. We will be predicting the difference between class 0 (Iris Setosa) and the other two classes.

In [9]:

```
# https://scikit-learn.org/stable/auto_examples/neighbors/plot_classification.html

# 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')
```

Out[9]:

```
<matplotlib.legend.Legend at 0x10ac3f278>
```


Sepal Width
Sepal Length
Setosa
Non-Setosa
4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0

Review: Least Squares

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

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

$y = \{0, 1\}$

It minimizes the mean squared error (MSE)

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

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

after trained, $y' = f(x') = \vec{\theta}^{*\top} \vec{x}'$

$0.5 < y'$ → 1

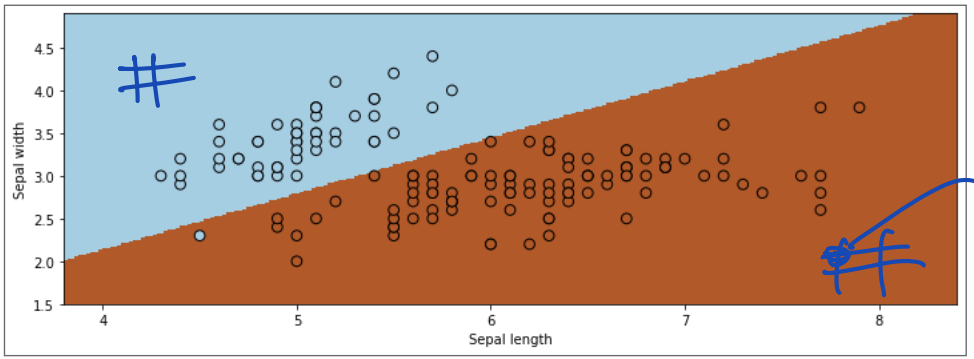
$0.5 \geq y$ → 0

$\} = y' \neq \{0, 1\}$

We could also use the above model for classification problem for which $\mathcal{Y} = \{0, 1\}$.

In [10]:

```
# https://scikit-learn.org/stable/auto_examples/linear_model/plot_iris_logistic.html
import warnings
warnings.filterwarnings("ignore")
from sklearn.linear_model import LinearRegression
logreg = LinearRegression()

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

# 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 = logreg.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()
```



$x' \Rightarrow y' = 0$

Least squares returns an acceptable decision boundary on this dataset. However, it 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.

The Logistic Function

To address this problem, we will look at a different hypothesis class. We will choose models of the form:

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

where

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

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

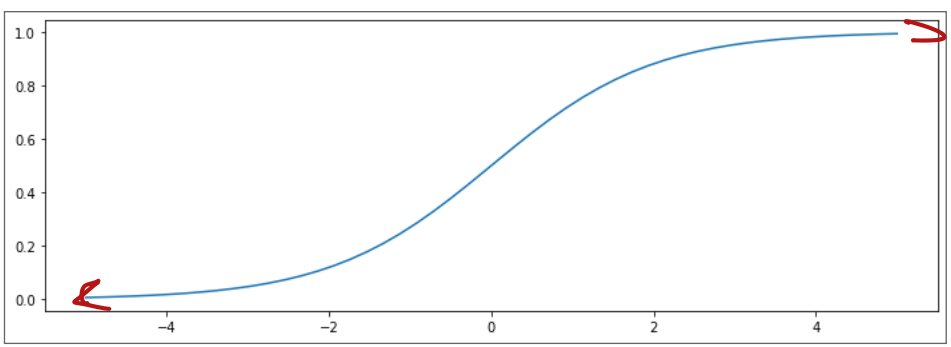
The logistic function $\sigma : \mathbb{R} \to [0, 1]$ "squeezes" points from the real line into $[0, 1]$.

In [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))
```

Out[11]:

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



The Logistic Function: Properties

The sigmoid function is defined as

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

A few observations:

- The tends to 1 as $z \to \infty$ and tends to 0 as $z \to -\infty$.
- Thus, models of the form $\sigma(\boldsymbol{\theta}^\top \mathbf{x})$ 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: $\frac{d\sigma}{dz} = \sigma(z)(1 - \sigma(z))$.

HW

Let's implement our model using the sigmoid function.

In [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))
```

Review: Probabilistic Least Squares

Recall that least squares can be interpreted as fitting a Gaussian probabilistic model

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

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

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

for some constants $\text{const}_1, \text{const}_2$.
Least squares thus amounts to fitting a Gaussian $\mathcal{N}(y; \mu(\mathbf{x}), \sigma)$ with a standard deviation σ of one and a mean of $\mu(\mathbf{x}) = \boldsymbol{\theta}^\top\mathbf{x}$.

A Probabilistic Approach to Classification

We can take this probabilistic perspective to derive a new algorithm for binary classification.
We will start by using our logistic model $\mathcal{M}$ to parametrize a probability distribution as follows:

$$p(y=1|\mathbf{x};\boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}^\top \mathbf{x})$$
$$p(y=0|\mathbf{x};\boldsymbol{\theta}) = 1 - \sigma(\boldsymbol{\theta}^\top \mathbf{x}).$$

defined

$P(y=0|x,\theta) = \sigma(\theta^T x)$

$P(y=1|x,\theta) = 1 - \sigma(\theta^T x)$

A probability over $y \in \{0, 1\}$ of the form $P(y = 1) = p$ is called Bernoulli. Thus, we postulate a conditional Bernoulli distribution for our model.

Note that we can write this more compactly as

$$p(y|\mathbf{x};\boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}^\top \mathbf{x})^y \cdot (1 - \sigma(\boldsymbol{\theta}^\top \mathbf{x}))^{1-y}$$

Review: Conditional Maximum Likelihood

A general approach of optimizing conditional models of the form $P_{\boldsymbol{\theta}}(y|\mathbf{x})$ is by minimizing expected KL divergence with respect to the data distribution:

$$\min_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x}\sim\mathbb{P}_{\text{data}}} \left[D(P_{\text{data}}(y|\mathbf{x}) \;||\; P_{\theta}(y|\mathbf{x})) \right].$$

With a bit of math, we can show that the maximum likelihood objective becomes

$$\max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x},y\sim\mathbb{P}_{\text{data}}} \log P_{\boldsymbol{\theta}}(y|\mathbf{x}).$$

This is the principle of *conditional maximum likelihood*.

Applying Maximum Lilkelihood

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

$$
\begin{aligned}
L(\boldsymbol{\theta}) &= \prod_{i=1}^{n} p(y^{(i)} \mid \mathbf{x}^{(i)}; \boldsymbol{\theta}) \\
&= \prod_{i=1}^{n} \sigma(\boldsymbol{\theta}^\top \mathbf{x}^{(i)})^{y^{(i)}} \cdot (1 - \sigma(\boldsymbol{\theta}^\top \mathbf{x}^{(i)}))^{1-y^{(i)}}.
\end{aligned}
$$

We have derive the log-loss, or cross-entropy loss:

$$
\log(L(\boldsymbol{\theta})) = \sum_{i=1}^{n} y^{(i)} \log(\sigma(\boldsymbol{\theta}^\top \mathbf{x}^{(i)})) + (1 - y^{(i)}) \log(1 - \sigma(\boldsymbol{\theta}^\top \mathbf{x}^{(i)})).
$$

This model and objective function define *logistic regression*, one of the most widely used classification algorithms (the name "regression" is an unfortunate misnomer!).
Let's implement the likelihood objective.

In [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()
```

Review: Gradient Descent

If we want to optimize $J(\boldsymbol{\theta})$, we start with an initial guess $\boldsymbol{\theta}_0$ for the parameters and repeat the following update:

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

As code, this method may look as follows:

```
theta, theta_prev = random_initialization()
while norm(theta - theta_prev) > convergence_threshold:
    theta_prev = theta
    theta = theta_prev - step_size * gradient(theta_prev)
```

Derivatives of the Log-Likelihood

Let's work out the gradient for our log likelihood objective for an arbitrary pair $(\mathbf{x}, y)$

$$
\begin{aligned}
\frac{\partial \log L(\boldsymbol{\theta})}{\partial \theta_j} &= \frac{\partial}{\partial \theta_j} \log\left(\sigma(\boldsymbol{\theta}^\top \mathbf{x})^y \cdot (1 - \sigma(\boldsymbol{\theta}^\top \mathbf{x}))^{1-y}\right) \\
&= \left(y \cdot \frac{1}{\sigma(\boldsymbol{\theta}^\top \mathbf{x})} - (1-y) \frac{1}{1 - \sigma(\boldsymbol{\theta}^\top \mathbf{x})}\right) \frac{\partial}{\partial \theta_j} \sigma(\boldsymbol{\theta}^\top \mathbf{x}) \\
&= \left(y \cdot \frac{1}{\sigma(\boldsymbol{\theta}^\top \mathbf{x})} - (1-y) \frac{1}{1 - \sigma(\boldsymbol{\theta}^\top \mathbf{x})}\right) \sigma(\boldsymbol{\theta}^\top \mathbf{x})(1 - \sigma(\boldsymbol{\theta}^\top \mathbf{x})) \frac{\partial}{\partial \theta_j} \boldsymbol{\theta}^\top \mathbf{x} \\
&= \left(y \cdot (1 - \sigma(\boldsymbol{\theta}^\top \mathbf{x})) - (1-y)\sigma(\boldsymbol{\theta}^\top \mathbf{x})\right) x_j \\
&= \left(y - \sigma(\boldsymbol{\theta}^\top \mathbf{x})\right) x_j.
\end{aligned}
$$

$[y - \sigma(\vec{\theta}^T \vec{x})] \vec{x}$

$\nabla_{\vec{\theta}} \log L(\theta) =$

$$
\begin{bmatrix} \frac{\partial \log L(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial \log L(\theta)}{\partial \theta_d} \end{bmatrix} = \begin{bmatrix} (y - \sigma(\vec{\theta}^T \vec{x}))\, x_1 \\ \vdots \\ y - \sigma(\vec{\theta}^T \vec{x})\, x_d \end{bmatrix}
$$

a scalar

Gradient of the Log-Likelihood

$J(\theta) = \log L(\theta, x, y)$

Using the above expression, we obtain the following gradient:

$$\nabla_\theta J(\boldsymbol{\theta}) = \left(y - \sigma(\boldsymbol{\theta}^\top \mathbf{x})\right) \cdot \mathbf{x}$$

- Note that this expression looks analogous to the gradient of the mean squared error.

$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \log L(\theta, x^i, y^i)$ taking into account all n data

$\nabla_{\vec{\theta}} J(\theta) = \frac{1}{n} \sum \left(y^i - \sigma(\vec{\theta}^T \vec{x}^i)\right) \vec{x}^i$

Let's implement the gradient.

In [14]:

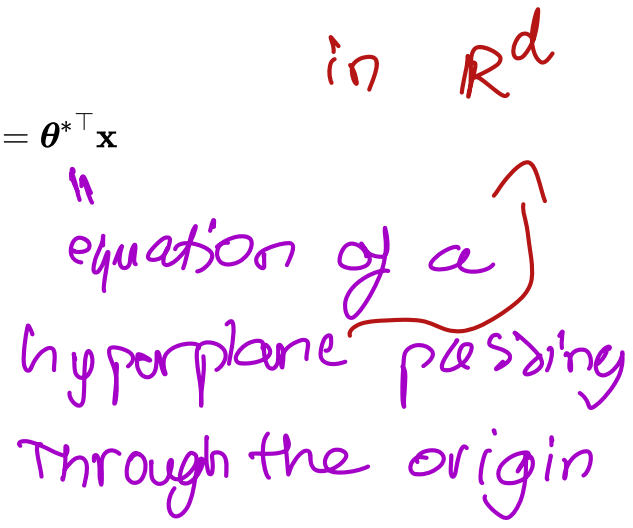
```
def loglik_gradient(theta, X, y):
    """The cost function, J(theta0, theta1) describing the goodness of fit.

    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((y - f(X, theta)) * X.T, axis=1)
```

How to make decision and the decision boundary

- If $p(y=1|\mathbf{x};\boldsymbol{\theta}^*) = \sigma(\boldsymbol{\theta}^{*\top}\mathbf{x}) \gtrless 0.5$, the predicted label is 1, otherwise 0.
- The decision boundary is thus the set of $\mathbf{x}$ such that $P(y=1|\mathbf{x},\boldsymbol{\theta}^*) = P(y=0|\mathbf{x},\boldsymbol{\theta}^*) = 0.5$
- We have, on the decision boundary,

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

- *The decision boundary is thus $0={|boldsymbol|theta^}^\top \mathbf x , which is a hyperplane (and hence linear in \mathbf x$)*



Gradient Descent for Logistic Regression

- Unlike least squares, we don't have a closed form solution for $\boldsymbol{\theta}$, but we can still apply gradient descent.

Putting all of the above (entropy loss and gradient) together, we obtain a complete learning algorithm, logistic regression.

```
theta, theta_prev = random_initialization()
while abs(J(theta) - J(theta_prev)) > conv_threshold:
    theta_prev = theta
    theta = theta_prev - step_size * (f(x, theta)-y) * x
```

Let's implement this algorithm.

In [15]:

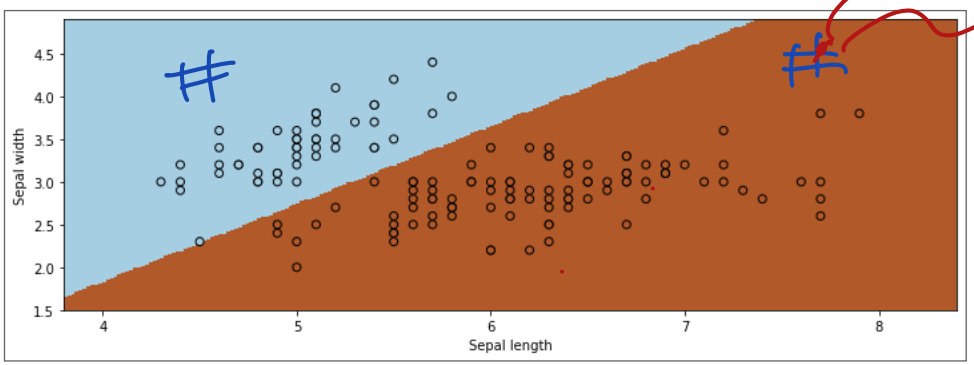
```
threshold = 5e-5
step_size = 1e-1

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:
# while True:
    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.

In [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()
```



x'

use the rule

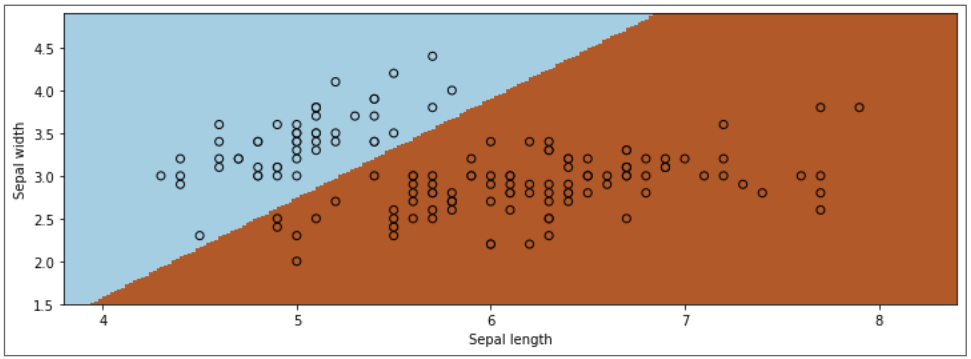
$p(y=1 \mid \theta^*, x') > 0.5$

then $y' = 1$

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

In [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()
```


Sepal width
Sepal length
4.5
4.0
3.5
3.0
2.5
2.0
1.5
4
5
6
7
8

Algorithm: Logistic Regression (classification)

- **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.

Part 5: Multi-Class Classification

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

Multi-Class Classification

Linear 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 one class from all the others.
- This works, but loses a valid probabilistic interpretation and is not very elegant.
- Alternatively, we may fit a probabilistic model that outputs multi-class probabilities.

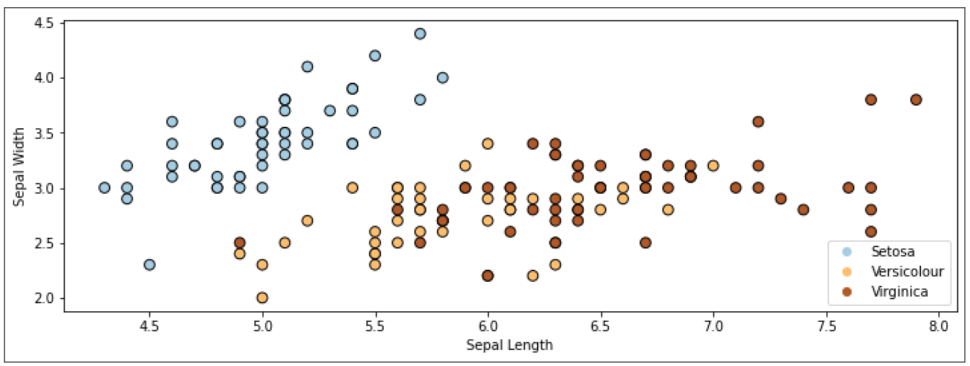
Let's load a fully multiclass version of the Iris dataset.

In [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')
```

Out[18]:

```
<matplotlib.legend.Legend at 0x12fbd4f60>
```



The Softmax Function

The logistic function $\sigma : \mathbb{R} \to [0, 1]$ be seen as mapping input an input $\mathbf{z} \in \mathbb{R}$ to a probability.

Its multi-class extension $\boldsymbol{\sigma} : \mathbb{R}^K \to [0, 1]^K$: maps a K-dimensional input $\mathbf{z} \in \mathbb{R}^K$ to a K-dimensional vector of probabilities.

$\vec{\sigma}(\vec{z})$ $\quad \vec{z} \in \mathbb{R}^K$ and $\vec{\sigma} : \mathbb{R}^K \to [0,1]^K$

Each componnent of $\boldsymbol{\sigma}(\vec{z})$ is defined as

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

kth component of $\vec{\sigma}(\vec{z})$

$\sum_{k=1}^{K} \sigma(z)_k = 1$

We call this the *softmax* function.

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

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

$\sigma(z)_2 = \dfrac{\exp(z_2)}{\exp(z_1) + \exp(z_2)}$

$= \dfrac{1}{1 + \exp(z_2 - z_1)}$ $\quad = \dfrac{\exp(z_2 - z_1)}{1 + \exp(z_2 - z_1)}$

Since

$$\sigma(\mathbf{z})_1 + \sigma(\mathbf{z})_2 = 1,$$

that is, if we know one, we know the other, we can set $\exp(z_1) = 1$. We can think this as normalization or using class 1 as the pivot. This amounts to dividing both numerator and denominator by $\exp(z_1)$ and redefine z_2. Thus we obtain:

Same as setting $z_1 = 0$

$z_2 \leftarrow z_2 - z_1$

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

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

The Softmax Model

We can use the softmax function to define a K-class classification model.
Recall in the binary classification setting, we mapped weights $\boldsymbol{\theta}$ and features $\mathbf{x}$ into a probability as follows:

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

In the multi-class setting, we define a model $f : \mathcal{X} \to [0,1]^K$ that outputs the probability of class k based on the features $\mathbf{x}$ and class-specific weights $\boldsymbol{\theta}^k$:

$$\sigma(\boldsymbol{\theta}^k \mathbf{x})_k = \frac{\exp(\boldsymbol{\theta}^{k\top} \mathbf{x})}{\sum_{l=1}^{K} \exp(\boldsymbol{\theta}^{l\top} \mathbf{x})}.$$

$\sigma(\Theta^\top \vec{x})_k$

kth component of softmax

Its parameter space $\boldsymbol{\Theta} = [\vec{\theta^1}, \ldots, \vec{\theta^K}]$ lies in K-fold Cartesian product $\mathbb{R}^d \times \cdots \times \mathbb{R}^d$, where is the parameter space of logistic regression.

$\vec{\theta}^k \in \mathbb{R}^d \Rightarrow d \times K$ is total

Similar to the binary classification above, we notice that this model is slightly over-parametrized: multiplying every θ^k by a constant results in an equivalent model. For this reason, it is often assumed that one of the class weights $\theta^l = 0$.

clarke rowy

$$\sum_{k=1}^{K} \sigma(\Theta^\top \vec{x})_k = 1$$

Softmax Regression

We again take a probabilistic perspective to derive a K-class classification algorithm based on this model.

We will start by using our softmax model to parametrize a probability distribution model $\mathcal{M}$ as follows:

$$p(y = k|\mathbf{x}; \mathbf{\Theta}) = \boldsymbol{\sigma}(\mathbf{\Theta}^\top \mathbf{x})_k$$

This is called a categorial distribution, and it generalizes the Bernoulli.

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

$$L(\mathbf{\Theta}) = \prod_{i=1}^{n} p(y^{(i)} \mid \mathbf{x}^{(i)}; \mathbf{\Theta}) = \prod_{i=1}^{n} \boldsymbol{\sigma}(\mathbf{\Theta}^\top \mathbf{x}^{(i)})_{y^{(i)}}$$

This model and objective function define *softmax regression*. (The term "regression" here is again a misnomer.)

$K = 4$

$$\vec{\sigma}(\Theta^T \vec{x}') = \begin{bmatrix} 0.2 \\ 0.3 \\ 0.4 \\ 0.1 \end{bmatrix}$$

$$\sum_{k=1}^{K} \sigma(\Theta^T \vec{x}')_k$$

$$\max_{1 \le k \le K} \sigma(\Theta^T x')_k = 0.4$$

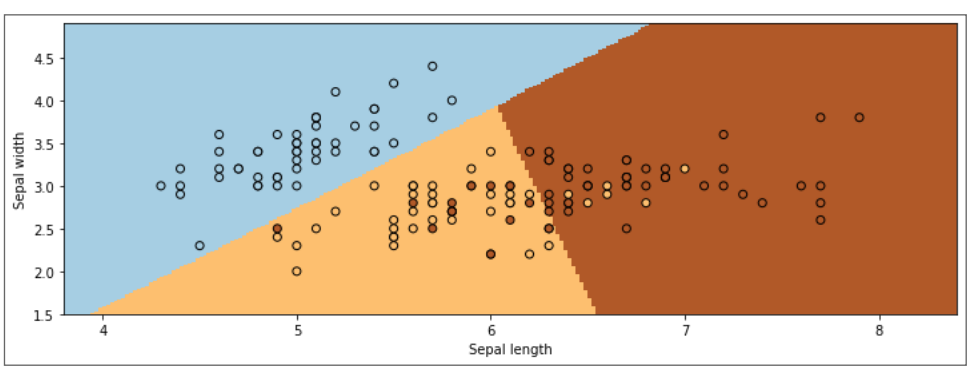
$$\Downarrow$$

$$y' = 3$$

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

In [19]:

```
# https://scikit-learn.org/stable/auto_examples/linear_model/plot_iris_logistic.html
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()
```



Algorithm: Softmax Regression

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