Programming Assignment

Naive Bayes and logistic regression

Instructions

In this notebook, you will write code to develop a Naive Bayes classifier model to the Iris dataset using Distribution objects from TensorFlow Probability. You will also explore the connection between the Naive Bayes classifier and logistic regression.

Some code cells are provided you in the notebook. You should avoid editing provided code, and make sure to execute the cells in order to avoid unexpected errors. Some cells begin with the line:

`#### GRADED CELL ####`

Don't move or edit this first line - this is what the automatic grader looks for to recognise graded cells. These cells require you to write your own code to complete them, and are automatically graded when you submit the notebook. Don't edit the function name or signature provided in these cells, otherwise the automatic grader might not function properly.

How to submit

Complete all the tasks you are asked for in the worksheet. When you have finished and are happy with your code, press the **Submit Assignment** button at the top of this notebook.

Let's get started!

We'll start running some imports, and loading the dataset. Do not edit the existing imports in the following cell. If you would like to make further Tensorflow imports, you should add them here.

In [1]:
```
#### PACKAGE IMPORTS ####

# Run this cell first to import all required packages. Do not make any imports elsewhere in the notebook
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
from sklearn import datasets, model_selection
%matplotlib inline

# If you would like to make further imports from TensorFlow or TensorFlow Probability, add them here
```

The Iris dataset

In this assignment, you will use the Iris dataset. It consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). Four features were measured from each sample: the length and the width of the sepals and petals, in centimeters. For a reference, see the following papers:

- R. A. Fisher. "The use of multiple measurements in taxonomic problems". Annals of Eugenics. 7 (2): 179–188, 1936.

Your goal is to construct a Naive Bayes classifier model that predicts the correct class from the sepal length and sepal width features. Under certain assumptions about this classifier model, you will explore the relation to logistic regression.

Load and prepare the data

We will first read in the Iris dataset, and split the dataset into training and test sets.

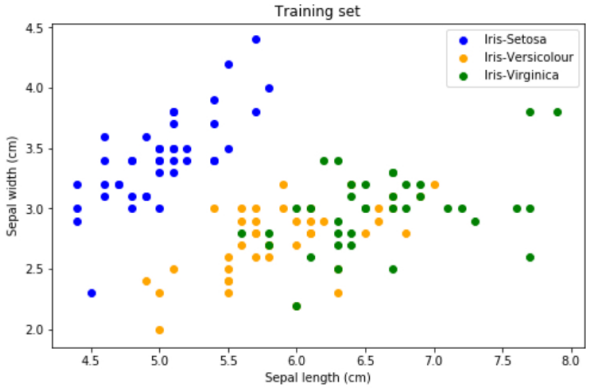
In [2]:
```
# Load the dataset

iris = datasets.load_iris()
```

In [3]:
```
# Use only the first two features: sepal length and width

data = iris.data[:, :2]
targets = iris.target
```

In [4]:
```
# Randomly shuffle the data and make train and test splits

x_train, x_test, y_train, y_test = model_selection.train_test_split(data, targets, test_size=0.2)
```

In [5]:
```
# Plot the training data

labels = {0: 'Iris-Setosa', 1: 'Iris-Versicolour', 2: 'Iris-Virginica'}
label_colours = ['blue', 'orange', 'green']

def plot_data(x, y, labels, colours):
```

```
    for c in np.unique(y):
        inx = np.where(y == c)
        plt.scatter(x[inx, 0], x[inx, 1], label=labels[c], c=colours[c])
    plt.title("Training set")
    plt.xlabel("Sepal length (cm)")
    plt.ylabel("Sepal width (cm)")
    plt.legend()

plt.figure(figsize=(8, 5))
plot_data(x_train, y_train, labels, label_colours)
plt.show()
```



Naive Bayes classifier

We will briefly review the Naive Bayes classifier model. The fundamental equation for this classifier is Bayes' rule:

$$P(Y = y_k|X_1, \dots, X_d) = \frac{P(X_1, \dots, X_d|Y = y_k)P(Y = y_k)}{\sum_{k=1}^{K} P(X_1, \dots, X_d|Y = y_k)P(Y = y_k)}$$

In the above, d is the number of features or dimensions in the inputs X (in our case $d = 2$), and K is the number of classes (in our case $K = 3$). The distribution $P(Y)$ is the class prior distribution, which is a discrete distribution over K classes. The distribution $P(X|Y)$ is the class-conditional distribution over inputs.

The Naive Bayes classifier makes the assumption that the data features X_i are conditionally independent give the class Y (the 'naive' assumption). In this case, the class-conditional distribution decomposes as

$$\begin{aligned} P(X|Y = y_k) &= P(X_1, \dots, X_d|Y = y_k) \\ &= \prod_{i=1}^{d} P(X_i|Y = y_k) \end{aligned}$$

This simplifying assumption means that we typically need to estimate far fewer parameters for each of the distributions $P(X_i|Y = y_k)$ instead of the full joint distribution $P(X|Y = y_k)$.

Once the class prior distribution and class-conditional densities are estimated, the Naive Bayes classifier model can then make a class prediction $\hat{Y}$ for a new data input $\tilde{X} := (\tilde{X}_1, \dots, \tilde{X}_d)$ according to

$$\begin{aligned} \hat{Y} &= \text{argmax}_{y_k} P(Y = y_k|\tilde{X}_1, \dots, \tilde{X}_d) \\ &= \text{argmax}_{y_k} \frac{P(\tilde{X}_1, \dots, \tilde{X}_d|Y = y_k)P(Y = y_k)}{\sum_{k=1}^{K} P(\tilde{X}_1, \dots, \tilde{X}_d|Y = y_k)P(Y = y_k)} \\ &= \text{argmax}_{y_k} P(\tilde{X}_1, \dots, \tilde{X}_d|Y = y_k)P(Y = y_k) \end{aligned}$$

Define the class prior distribution

We will begin by defining the class prior distribution. To do this we will simply take the maximum likelihood estimate, given by

$$P(Y = y_k) = \frac{\sum_{n=1}^{N} \delta(Y^{(n)} = y_k)}{N},$$

where the superscript (n) indicates the n-th dataset example, $\delta(Y^{(n)} = y_k) = 1$ if $Y^{(n)} = y_k$ and 0 otherwise, and N is the total number of examples in the dataset. The above is simply the proportion of data examples belonging to class k.

You should now write a function that builds the prior distribution from the training data, and returns it as a `Categorical` Distribution object.

- The input to your function `y` will be a numpy array of shape `(num_samples,)`
- The entries in `y` will be integer labels $k = 0, 1, \dots, K - 1$
- Your function should build and return the prior distribution as a `Categorical` distribution object
  - The probabilities for this distribution will be a length-K vector, with entries corresponding to $P(Y = y_k)$ for $k = 0, 1, \dots, K - 1$
  - Your function should work for any value of $K \geq 1$
  - This Distribution will have an empty batch shape and empty event shape

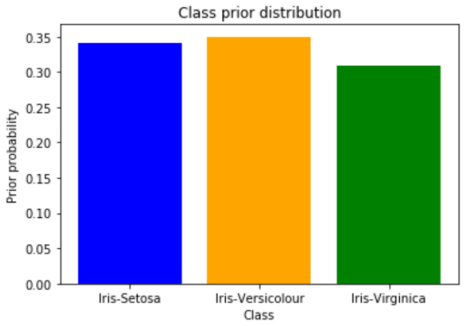
In [6]:

```
#### GRADED CELL ####

# Complete the following function.
# Make sure to not change the function name or arguments.

def get_prior(y):
    """
    This function takes training labels as a numpy array y of shape (num_samples,) as an input.
    Your function should
```

```
    Your function should
    This function should build a Categorical Distribution object with empty batch shape
    and event shape, with the probability of each class given as above.
    Your function should return the Distribution object.
    """

```

In [7]:
```
# Run your function to get the prior

prior = get_prior(y_train)
# prior.prob([1]) + prior.prob([0]) + prior.prob([2])
```

In [8]:
```
# Plot the prior distribution

labels = ['Iris-Setosa', 'Iris-Versicolour', 'Iris-Virginica']
plt.bar([0, 1, 2], prior.probs.numpy(), color=label_colours)
plt.xlabel("Class")
plt.ylabel("Prior probability")
plt.title("Class prior distribution")
plt.xticks([0, 1, 2], labels)
plt.show()
```



Define the class-conditional densities

We now turn to the definition of the class-conditional distributions $P(X_i|Y = y_k)$ for $i = 0, 1$ and $k = 0, 1, 2$. In our model, we will assume these distributions to be univariate Gaussian:

$$P(X_i|Y = y_k) = N(X_i|\mu_{ik}, \sigma_{ik}) = \frac{1}{\sqrt{2\pi\sigma_{ik}^2}} \exp\left\{ -\frac{1}{2}\left(\frac{x - \mu_{ik}}{\sigma_{ik}} \right)^2 \right\}$$

with mean parameters μ_{ik} and standard deviation parameters σ_{ik}, twelve parameters in all. We will again estimate these parameters using maximum likelihood. In this case, the estimates are given by

$$\hat{\mu}_{ik} = \frac{\sum_n X_i^{(n)} \delta(Y^{(n)} = y_k)}{\sum_n \delta(Y^{(n)} = y_k)}$$

$$\hat{\sigma}_{ik}^2 = \frac{\sum_n (X_i^{(n)} - \hat{\mu}_{ik})^2 \delta(Y^{(n)} = y_k)}{\sum_n \delta(Y^{(n)} = y_k)}$$

Note that the above are just the means and variances of the sample data points for each class.

You should now write a function the computes the class-conditional Gaussian densities, using the maximum likelihood parameter estimates given above, and returns them in a single, batched `MultivariateNormalDiag` Distribution object.

- The inputs to the function are
  - a numpy array `x` of shape `(num_samples, num_features)` for the data inputs
  - a numpy array `y` of shape `(num_samples,)` for the target labels
- Your function should work for any number of classes $K \geq 1$ and any number of features $d \geq 1$

In [9]:
```
#### GRADED CELL ####

# Complete the following function.
# Make sure to not change the function name or arguments.

def get_class_conditionals(x, y):
    """
    This function takes training data samples x and labels y as inputs.
    This function should build the class-conditional Gaussian distributions above.
    It should construct a batch of distributions for each feature and each class, using the
    parameter estimates above for the means and standard deviations.
    The batch shape of this distribution should be rank 2, where the first dimension corresponds
    to the number of classes and the second corresponds to the number of features.
    Your function should then return the Distribution object.
    """

```

In [10]:
```
# Run your function to get the class-conditional distributions

class_conditionals = get_class_conditionals(x_train, y_train)
class_conditionals
```

Out[10]:
```
<tfp.distributions.MultivariateNormalDiag 'MultivariateNormalDiag' batch_shape=[3] event_shape=[2] dtype=float32>
```

In [11]:
```
# Random cell to verify some outputs
```

```
n_classes = np.unique(y_train)
n_features = x_train.shape[1]
print(n_classes)
means = np.zeros((n_classes.shape[0],n_features))
print(means)
variances = np.zeros((n_classes.shape[0],n_features))
for i, n_class in enumerate(n_classes):
    means[i] = x_train[y_train == n_class].mean(axis = 0, keepdims = True)
    variances[i] = x_train[y_train == n_class].var(axis = 0, keepdims = True)

print(np.sqrt(variances))
dist = tfd.MultivariateNormalDiag(loc = means.T,
                          scale_diag = variances.T)
dist
```

```
[0 1 2]
[[0. 0.]
 [0. 0.]
 [0. 0.]]
[[0.35776423 0.3668604 ]
 [0.51815126 0.29051522]
 [0.56943066 0.32364794]]
```

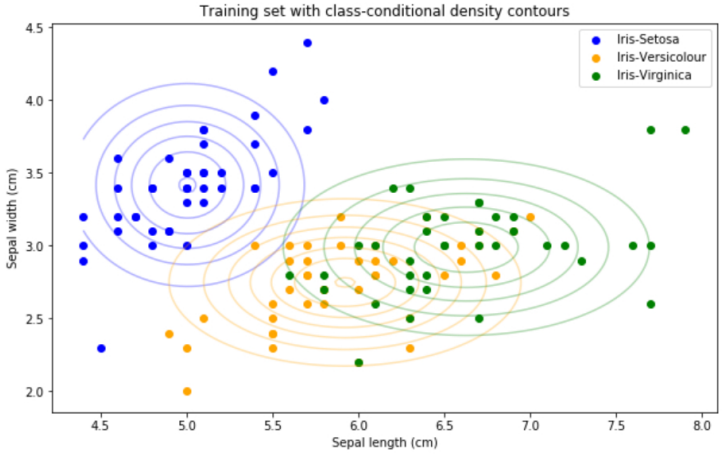
Out[11]: `<tfp.distributions.MultivariateNormalDiag 'MultivariateNormalDiag' batch_shape=[2] event_shape=[3] dtype=float64>`

In [12]:
```
## Random cell to verify some outputs

locs = np.array([
    [[2],[3]],
    [[3],[4]],
    [[5],[6]]
], dtype = 'float32')

scale_diags = np.array([
    [[1],[2]],
    [[3],[4]],
    [[5],[6]]
], dtype = 'float32')

MVND = tfd.MultivariateNormalDiag(loc = locs,
                          scale_diag = scale_diags)

MVND
```

Out[12]: `<tfp.distributions.MultivariateNormalDiag 'MultivariateNormalDiag' batch_shape=[3, 2] event_shape=[1] dtype=float32>`

We can visualise the class-conditional densities with contour plots by running the cell below. Notice how the contours of each distribution correspond to a Gaussian distribution with diagonal covariance matrix, since the model assumes that each feature is independent given the class.

In [13]:
```
# Plot the training data with the class-conditional density contours

def get_meshgrid(x0_range, x1_range, num_points=100):
    x0 = np.linspace(x0_range[0], x0_range[1], num_points)
    x1 = np.linspace(x1_range[0], x1_range[1], num_points)
    return np.meshgrid(x0, x1)

def contour_plot(x0_range, x1_range, prob_fn, batch_shape, colours, levels=None, num_points=100):
    X0, X1 = get_meshgrid(x0_range, x1_range, num_points=num_points)
    Z = prob_fn(np.expand_dims(np.array([X0.ravel(), X1.ravel()]).T, 1))
    Z = np.array(Z).T.reshape(batch_shape, *X0.shape)
    for batch in np.arange(batch_shape):
        if levels:
            plt.contourf(X0, X1, Z[batch], alpha=0.2, colors=colours, levels=levels)
        else:
            plt.contour(X0, X1, Z[batch], colors=colours[batch], alpha=0.3)

plt.figure(figsize=(10, 6))
plot_data(x_train, y_train, labels, label_colours)
x0_min, x0_max = x_train[:, 0].min(), x_train[:, 0].max()
x1_min, x1_max = x_train[:, 1].min(), x_train[:, 1].max()
contour_plot((x0_min, x0_max), (x1_min, x1_max), class_conditionals.prob, 3, label_colours)
plt.title("Training set with class-conditional density contours")
plt.show()
```



Make predictions from the model

Now the prior and class-conditional distributions are defined, you can use them to compute the model's class probability predictions for an unknown test input $\tilde{X} = (\tilde{X}_1, \ldots, \tilde{X}_d)$, according to

$$P(Y = y_k | \tilde{X}_1, \ldots, \tilde{X}_d) = \frac{P(\tilde{X}_1, \ldots, \tilde{X}_d | Y = y_k) P(Y = y_k)}{\sum_{k=1}^{K} P(\tilde{X}_1, \ldots, \tilde{X}_d | Y = y_k) P(Y = y_k)}$$

The class prediction can then be taken as the class with the maximum probability:

$$\hat{Y} = \text{argmax}_{y_k} P(Y = y_k | \tilde{X}_1, \ldots, \tilde{X}_d)$$

You should now write a function to return the model's class probabilities for a given batch of test inputs of shape `(batch_shape, 2)`, where the `batch_shape` has rank at least one.

- The inputs to the function are the `prior` and `class_conditionals` distributions, and the inputs `x`
- Your function should use these distributions to compute the probabilities for each class k as above
  - As before, your function should work for any number of classes $K \geq 1$
- It should then compute the prediction by taking the class with the highest probability
- The predictions should be returned in a numpy array of shape `(batch_shape)`

In [14]:
```
#### GRADED CELL ####

# Complete the following function.
# Make sure to not change the function name or arguments.

def predict_class(prior, class_conditionals, x):
    """
    This function takes the prior distribution, class-conditional distribution, and
    a batch of inputs in a numpy array of shape (batch_shape, 2).
    This function should compute the class probabilities for each input in the batch, using
    the prior and class-conditional distributions, according to the above equation.
    Note that the batch_shape of x could have rank higher than one!
    Your function should then return the class predictions by taking the class with the
    maximum probability in a numpy array of shape (batch_shape,).
    """

```

In [15]:
```
## Coding cell to test the codes

prior = get_prior(y_train)
prior_logs = np.log(prior.probs)
cond_probs = class_conditionals.log_prob(x_test[:,None,:])
joint_likelihood = tf.add(prior_logs, cond_probs)
norm_factor = tf.math.reduce_logsumexp(joint_likelihood, axis = -1, keepdims = True)
log_prob = joint_likelihood - norm_factor
tf.argmax(log_prob, axis = -1).numpy()
```

Out[15]:
```
array([1, 2, 0, 1, 2, 2, 0, 2, 2, 0, 0, 1, 1, 1, 0, 1, 0, 1, 2, 1, 0, 1,
       0, 1, 2, 2, 2, 2, 2, 0])
```

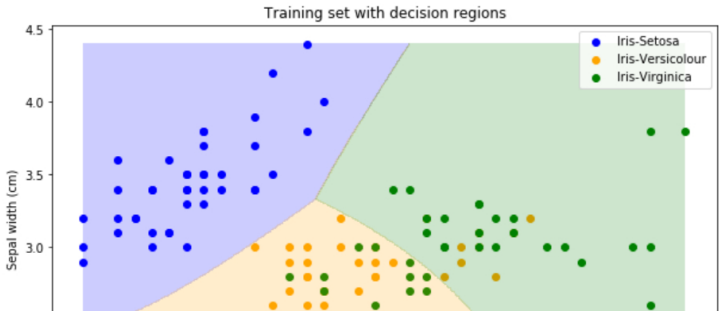
In [16]:
```
# Get the class predictions

predictions = predict_class(prior, class_conditionals, x_test)
predictions
```

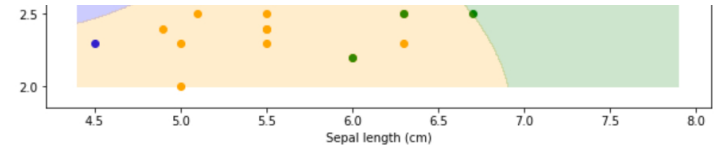
Out[16]:
```
array([1, 2, 0, 1, 2, 2, 0, 2, 2, 0, 0, 1, 1, 1, 0, 1, 0, 1, 2, 1, 0, 1,
       0, 1, 2, 2, 2, 2, 2, 0])
```

In [17]:
```
# Evaluate the model accuracy on the test set

accuracy = accuracy_score(y_test, predictions)
print("Test accuracy: {:.4f}".format(accuracy))
```

```
Test accuracy: 0.6667
```

In [18]:
```
# Plot the model's decision regions

plt.figure(figsize=(10, 6))
plot_data(x_train, y_train, labels, label_colours)
x0_min, x0_max = x_train[:, 0].min(), x_train[:, 0].max()
x1_min, x1_max = x_train[:, 1].min(), x_train[:, 1].max()
contour_plot((x0_min, x0_max), (x1_min, x1_max),
             lambda x: predict_class(prior, class_conditionals, x),
             1, label_colours, levels=[-0.5, 0.5, 1.5, 2.5],
             num_points=500)
plt.title("Training set with decision regions")
plt.show()
```





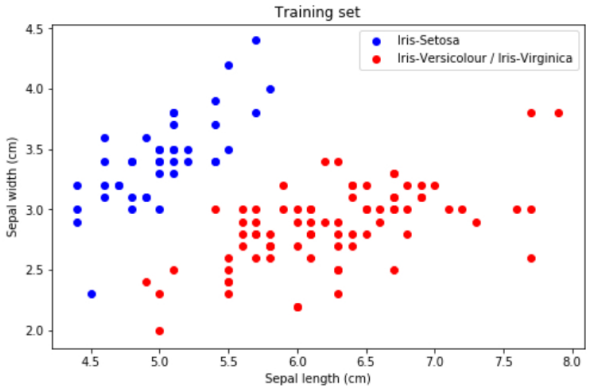
Binary classifier

We will now draw a connection between the Naive Bayes classifier and logistic regression.

First, we will update our model to be a binary classifier. In particular, the model will output the probability that a given input data sample belongs to the 'Iris-Setosa' class: $P(Y = y_0 | \tilde{X}_1, \ldots, \tilde{X}_d)$. The remaining two classes will be pooled together with the label y_1 .

In [19]:
```
# Redefine the dataset to have binary labels

y_train_binary = np.array(y_train)
y_train_binary[np.where(y_train_binary == 2)] = 1

y_test_binary = np.array(y_test)
y_test_binary[np.where(y_test_binary == 2)] = 1
```

In [20]:
```
# Plot the training data

labels_binary = {0: 'Iris-Setosa', 1: 'Iris-Versicolour / Iris-Virginica'}
label_colours_binary = ['blue', 'red']

plt.figure(figsize=(8, 5))
plot_data(x_train, y_train_binary, labels_binary, label_colours_binary)
plt.show()
```



We will also make an extra modelling assumption that for each class k, the class-conditional distribution $P(X_i | Y = y_k)$ for each feature $i = 0, 1$, has standard deviation σ_i, which is the same for each class k.

This means there are now six parameters in total: four for the means μ_{ik} and two for the standard deviations σ_i $(i, k = 0, 1)$.

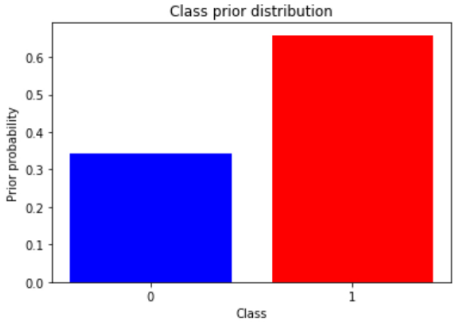
We will again use maximum likelihood to estimate these parameters. The prior distribution will be as before, with the class prior probabilities given by

$$P(Y = y_k) = \frac{\sum_{n=1}^{N} \delta(Y^{(n)} = y_k)}{N},$$

We will use your previous function `get_prior` to redefine the prior distribution.

In [21]:
```
# Redefine the prior

prior_binary = get_prior(y_train_binary)
```

In [22]:
```
# Plot the prior distribution

plt.bar([0, 1], prior_binary.probs.numpy(), color=label_colours_binary)
plt.xlabel("Class")
plt.ylabel("Prior probability")
plt.title("Class prior distribution")
plt.xticks([0, 1], labels_binary)
plt.show()
```



For the class-conditional densities, the maximum likelihood estimate for the means are again given by

$$\hat{\mu}_{ik} = \frac{\sum_n X_i^{(n)} \delta(Y^{(n)} = y_k)}{\sum_n \delta(Y^{(n)} = y_k)}$$

However, the estimate for the standard deviations σ_i is updated. There is also a closed-form solution for the shared standard deviations, but we will instead learn these from the data.

You should now write a function that takes the training inputs and target labels as input, as well as an optimizer object, number of epochs and a TensorFlow Variable. This function should be written according to the following spec:

- The inputs to the function are:
  - a numpy array `x` of shape `(num_samples, num_features)` for the data inputs
  - a numpy array `y` of shape `(num_samples,)` for the target labels
  - a `tf.Variable` object `scales` of length 2 for the standard deviations σ_i
  - `optimiser` : an optimiser object
  - `epochs` : the number of epochs to run the training for
- The function should first compute the means μ_{ik} of the class-conditional Gaussians according to the above equation
- Then create a batched multivariate Gaussian distribution object using `MultivariateNormalDiag` with the means set to μ_{ik} and the scales set to `scales`
- Run a custom training loop for `epochs` number of epochs, in which:
  - the average per-example negative log likelihood for the whole dataset is computed as the loss
  - the gradient of the loss with respect to the `scales` variables is computed
  - the `scales` variables are updated by the `optimiser` object
- At each iteration, save the values of the `scales` variable and the loss
- The function should return a tuple of three objects:
  - a numpy array of shape `(epochs,)` of loss values
  - a numpy array of shape `(epochs, 2)` of values for the `scales` variable at each iteration
  - the final learned batched `MultivariateNormalDiag` distribution object

NB: ideally, we would like to constrain the `scales` variable to have positive values. We are not doing that here, but in later weeks of the course you will learn how this can be implemented.

In [23]:
```
#### GRADED CELL ####

# Complete the following function.
# Make sure to not change the function name or arguments.

def learn_stdevs(x, y, scales, optimiser, epochs):
    """
    This function takes the data inputs, targets, scales variable, optimiser and number of
    epochs as inputs.
    This function should set up and run a custom training loop according to the above
    specifications, by setting up the class conditional distributions as a MultivariateNormalDiag
    object, and updating the trainable variables (the scales) in a custom training loop.
    Your function should then return the a tuple of three elements: a numpy array of loss values
    during training, a numpy array of scales variables during training, and the final learned
    MultivariateNormalDiag distribution object.
    """


    def nll(x, y, distribution):

    @tf.function
    def get_loss_and_grads(x,y, distribution):

```

In [24]:
```
## Coding cell to see if code works

def nll(x, y, distribution):


@tf.function
def get_loss_and_grads(x,y, distribution):



for i in range(500):


```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-24-e5f1f43e3691> in <module>
     30 for i in range(500):
     31     loss, grads = get_loss_and_grads(x,y_train_binary, distribution)
---> 32     opt.apply_gradients(zip(grads, distribution.trainable_variables))
     33
     34     scales_value = scales.value().numpy()

NameError: name 'opt' is not defined
```

In [25]:
```
# Define the inputs to your function

scales = tf.Variable([1., 1.])
opt = tf.keras.optimizers.Adam(learning_rate=0.01)
epochs = 500
```

In [26]:
```
# Run your function to learn the class-conditional standard deviations
```

```
nlls, scales_arr, class_conditionals_binary = learn_stdevs(x_train, y_train_binary, scales, opt, epochs)
```

```
Step 000: Loss: 246.750: Scales: [0.99 0.99].
Step 001: Loss: 244.870: Scales: [0.9799999 0.9799982].
Step 002: Loss: 242.982: Scales: [0.9699997  0.96999323].
Step 003: Loss: 241.085: Scales: [0.9599995 0.959984 ].
Step 004: Loss: 239.180: Scales: [0.9499995 0.9499693].
Step 005: Loss: 237.268: Scales: [0.94      0.9399478].
Step 006: Loss: 235.347: Scales: [0.93000144 0.92991835].
Step 007: Loss: 233.418: Scales: [0.9200043 0.9198798].
Step 008: Loss: 231.481: Scales: [0.9100094 0.909831 ].
Step 009: Loss: 229.537: Scales: [0.9000174 0.8997708].
Step 010: Loss: 227.586: Scales: [0.8900294 0.889698 ].
Step 011: Loss: 225.628: Scales: [0.8800465 0.8796117].
Step 012: Loss: 223.663: Scales: [0.87007016 0.86951065].
Step 013: Loss: 221.693: Scales: [0.8601019  0.85939395].
Step 014: Loss: 219.716: Scales: [0.8501435 0.8492606].
Step 015: Loss: 217.735: Scales: [0.840197  0.8391097].
Step 016: Loss: 215.749: Scales: [0.83026475 0.82894033].
Step 017: Loss: 213.759: Scales: [0.82034934 0.8187516 ].
Step 018: Loss: 211.765: Scales: [0.8104538 0.8085427].
Step 019: Loss: 209.770: Scales: [0.8005813 0.798313 ].
```

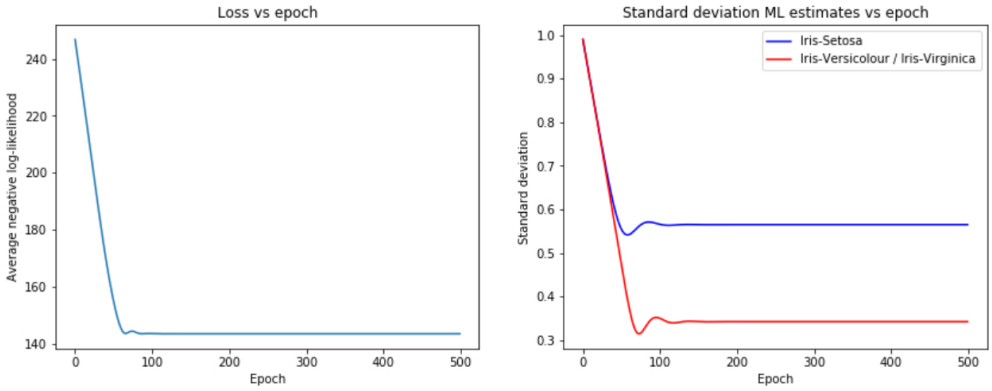
In [27]:

```
# View the distribution parameters

print("Class conditional means:")
print(class_conditionals_binary.loc.numpy())
print("\nClass conditional standard deviations:")
print(class_conditionals_binary.stddev().numpy())
```

```
Class conditional means:
[[5.007317  3.4170732]
 [6.2544303 2.8607595]]

Class conditional standard deviations:
[[0.5651235  0.34261736]
 [0.5651235  0.34261736]]
```

In [28]:

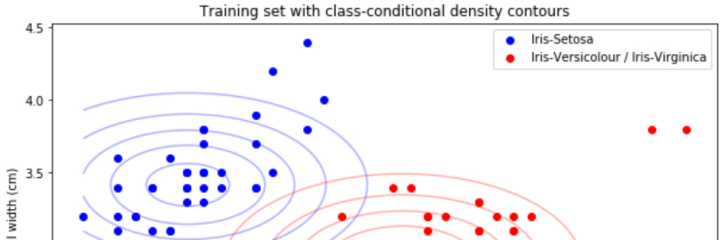
```
# Plot the loss and convergence of the standard deviation parameters

fig, ax = plt.subplots(1, 2, figsize=(14, 5))
ax[0].plot(nlls)
ax[0].set_title("Loss vs epoch")
ax[0].set_xlabel("Epoch")
ax[0].set_ylabel("Average negative log-likelihood")
for k in [0, 1]:
    ax[1].plot(scales_arr[:, k], color=label_colours_binary[k], label=labels_binary[k])
ax[1].set_title("Standard deviation ML estimates vs epoch")
ax[1].set_xlabel("Epoch")
ax[1].set_ylabel("Standard deviation")
plt.legend()
plt.show()
```

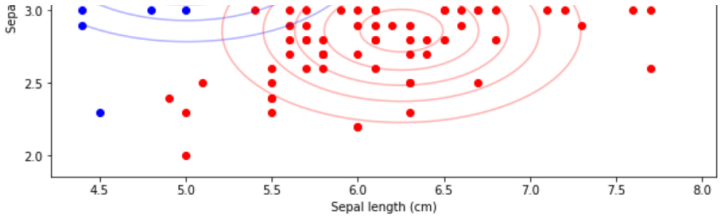


We can also plot the contours of the class-conditional Gaussian distributions as before, this time with just binary labelled data. Notice the contours are the same for each class, just with a different centre location.

In [29]:

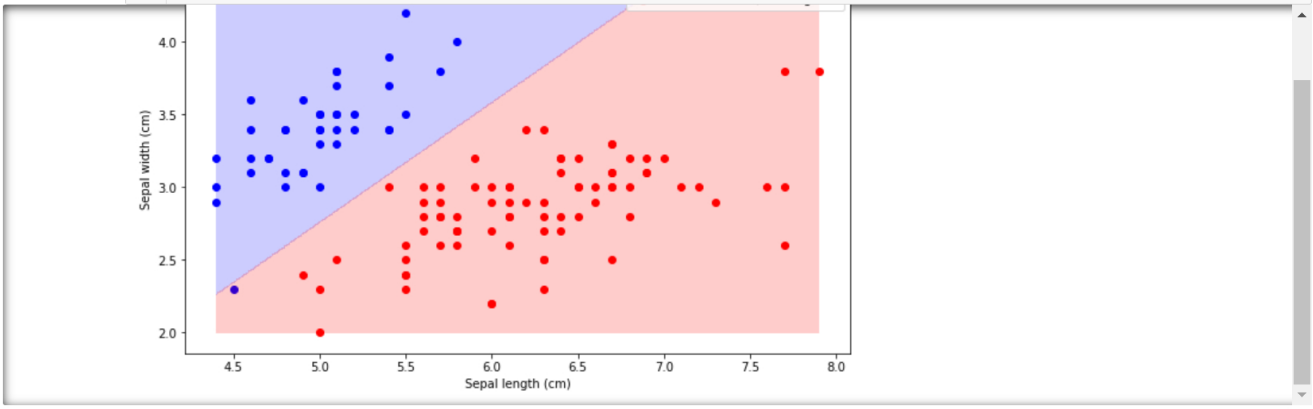
```
# Plot the training data with the class-conditional density contours

plt.figure(figsize=(10, 6))
plot_data(x_train, y_train_binary, labels_binary, label_colours_binary)
x0_min, x0_max = x_train[:, 0].min(), x_train[:, 0].max()
x1_min, x1_max = x_train[:, 1].min(), x_train[:, 1].max()
contour_plot((x0_min, x0_max), (x1_min, x1_max), class_conditionals_binary.prob, 2, label_colours_binary)
plt.title("Training set with class-conditional density contours")
plt.show()
```





We can also plot the decision regions for this binary classifier model, notice that the decision boundary is now linear.

In [30]:
```
# Plot the model's decision regions

plt.figure(figsize=(10, 6))
plot_data(x_train, y_train_binary, labels_binary, label_colours_binary)
x0_min, x0_max = x_train[:, 0].min(), x_train[:, 0].max()
x1_min, x1_max = x_train[:, 1].min(), x_train[:, 1].max()
contour_plot((x0_min, x0_max), (x1_min, x1_max),
             lambda x: predict_class(prior_binary, class_conditionals_binary, x),
             1, label_colours_binary, levels=[-0.5, 0.5, 1.5],
             num_points=500)
plt.title("Training set with decision regions")
plt.show()
```



Link to logistic regression

In fact, we can see that our predictive distribution $P(Y = y_0|X)$ can be written as follows:

$$\begin{aligned} P(Y = y_0|X) &= \frac{P(X|Y = y_0)P(Y = y_0)}{P(X|Y = y_0)P(Y = y_0) + P(X|Y = y_1)P(Y = y_1)} \\ &= \frac{1}{1 + \frac{P(X|Y=y_1)P(Y=y_1)}{P(X|Y=y_0)P(Y=y_0)}} \\ &= \sigma(a) \end{aligned}$$

where $\sigma(a) = \frac{1}{1+e^{-a}}$ is the sigmoid function, and $a = \log \frac{P(X|Y=y_0)P(Y=y_0)}{P(X|Y=y_1)P(Y=y_1)}$ is the *log-odds*.

With our additional modelling assumption of a shared covariance matrix Σ, it can be shown (using the Gaussian pdf) that a is in fact a linear function of X:

$$a = w^T X + w_0$$

where

$$\begin{aligned} w &= \Sigma^{-1}(\mu_0 - \mu_1) \\ w_0 &= -\frac{1}{2}\mu_0^T\Sigma^{-1}\mu_0 + \frac{1}{2}\mu_1^T\Sigma^{-1}\mu_1 + \log\frac{P(Y = y_0)}{P(Y = y_1)} \end{aligned}$$

The model therefore takes the form $P(Y = y_0|X) = \sigma(w^T X + w_0)$, with weights $w \in \mathbb{R}^2$ and bias $w_0 \in \mathbb{R}$. This is the form used by logistic regression, and explains why the decision boundary above is linear.

In the above we have outlined the derivation of the generative logistic regression model. The parameters are typically estimated with maximum likelihood, as we have done.

Finally, we will use the above equations to directly parameterise the output Bernoulli distribution of the generative logistic regression model.

You should now write the following function, according to the following specification:

- The inputs to the function are:
  - the prior distribution `prior` over the two classes
  - the (batched) class-conditional distribution `class_conditionals`
- The function should use the parameters of the above distributions to compute the weights and bias terms w and w_0 as above
- The function should then return a tuple of two numpy arrays for w and w_0

In [31]:
```
#### GRADED CELL ####

# Complete the following function.
# Make sure to not change the function name or arguments.

def get_logistic_regression_params(prior, class_conditionals):
    """
```

```
    This function takes the prior distribution and class-conditional distribution as inputs.
    This function should compute the weights and bias terms of the generative logistic
    regression model as above, and return them in a 2-tuple of numpy arrays of shapes
    (2,) and () respectively.
    """

```

In [32]:
```
sigma_inv = np.linalg.inv(class_conditionals_binary.covariance())
print(class_conditionals_binary.loc)
mu_0 = class_conditionals_binary.loc.numpy()[:,0]
mu_1 = class_conditionals_binary.loc.numpy()[:,1]
print(mu_0, mu_1)
print(prior.probs[0])
w = np.dot(sigma_inv,(mu_0-mu_1))
w_0 = -0.5*mu_0.T @ sigma_inv @ mu_0 + 0.5*mu_1.T @ sigma_inv @ mu_1 + np.log(prior.probs[0]/prior.probs[1])
w_0
```

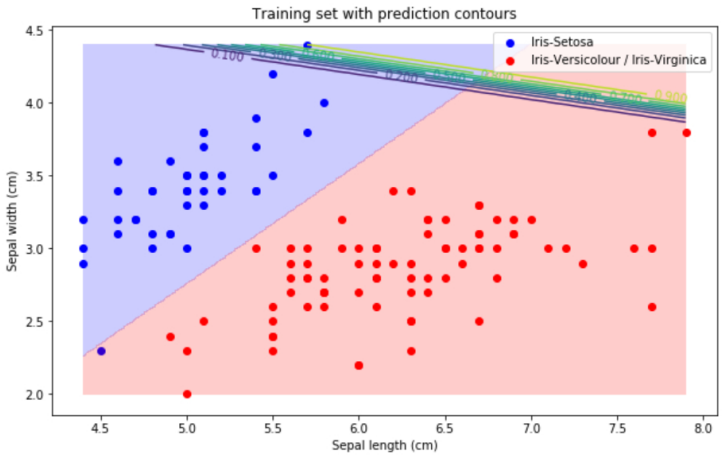
```
tf.Tensor(
[[5.007317  3.4170732]
 [6.2544303 2.8607595]], shape=(2, 2), dtype=float32)
[5.007317  6.2544303] [3.4170732 2.8607595]
tf.Tensor(0.34166667, shape=(), dtype=float32)
```

Out[32]:
```
-152.75925
```

In [33]:
```
# Run your function to get the logistic regression parameters

w, w0 = get_logistic_regression_params(prior_binary, class_conditionals_binary)
```

We can now use these parameters to make a contour plot to display the predictive distribution of our logistic regression model.

In [34]:
```
# Plot the training data with the logistic regression prediction contours

fig, ax = plt.subplots(1, 1, figsize=(10, 6))
plot_data(x_train, y_train_binary, labels_binary, label_colours_binary)
x0_min, x0_max = x_train[:, 0].min(), x_train[:, 0].max()
x1_min, x1_max = x_train[:, 1].min(), x_train[:, 1].max()
X0, X1 = get_meshgrid((x0_min, x0_max), (x1_min, x1_max))

logits = np.dot(np.array([X0.ravel(), X1.ravel()]).T, w) + w0
Z = tf.math.sigmoid(logits)
lr_contour = ax.contour(X0, X1, np.array(Z).T.reshape(*X0.shape), levels=10)
ax.clabel(lr_contour, inline=True, fontsize=10)
contour_plot((x0_min, x0_max), (x1_min, x1_max),
             lambda x: predict_class(prior_binary, class_conditionals_binary, x),
             1, label_colours_binary, levels=[-0.5, 0.5, 1.5],
             num_points=300)
plt.title("Training set with prediction contours")
plt.show()
```



Congratulations on completing this programming assignment! In the next week of the course we will look at Bayesian neural networks and uncertainty quantification.