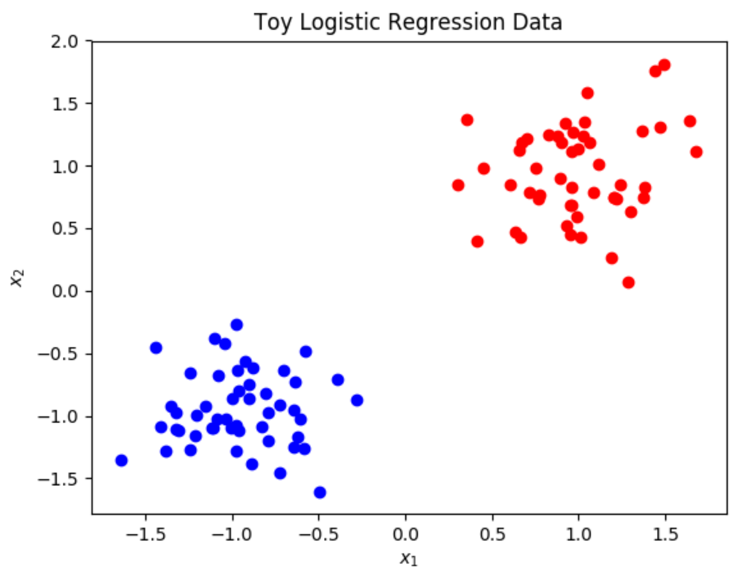

Figure 3-7. Plot of the toy classification data distribution.

New TensorFlow Concepts

Creating simple machine learning systems in TensorFlow will require that you learn some new TensorFlow concepts.

Placeholders

A placeholder is a way to input information into a TensorFlow computation graph. Think of placeholders as the input nodes through which information enters TensorFlow. The key function used to create placeholders is `tf.placeholder` (Example 3-4).

Example 3-4. Create a TensorFlow placeholder

```
>>> tf.placeholder(tf.float32, shape=(2,2))
<tf.Tensor 'Placeholder:0' shape=(2, 2) dtype=float32>
```

We will use placeholders to feed datapoints *x* and labels *y* to our regression and classification algorithms.

Feed dictionaries and Fetches

Recall that we can evaluate tensors in TensorFlow by using `sess.run(var)`. How do we feed in values for placeholders in our TensorFlow computations then? The answer is to construct *feed dictionaries*. Feed dictionaries are Python dictionaries that map TensorFlow tensors to `np.ndarray` objects that contain the concrete values for these placeholders. A feed dictionary is best viewed as an input to a TensorFlow computation graph. What then is an output? TensorFlow calls these outputs *fetches*. You have seen fetches already. We used them extensively in the previous chapter without calling them as such; the fetch is a tensor (or tensors) whose value is retrieved from the computation graph after the computation (using placeholder values from the feed dictionary) is run to completion (Example 3-5).

Example 3-5. Using fetches

```
>>> a = tf.placeholder(tf.float32, shape=(1,))
>>> b = tf.placeholder(tf.float32, shape=(1,))
>>> c = a + b
>>> with tf.Session() as sess:
        c_eval = sess.run(c, {a: [1.], b: [2.]})
        print(c_eval)
[ 3.]
```

Name scopes

In complicated TensorFlow programs, there will be many tensors, variables, and placeholders defined throughout the program. `tf.name_scope(name)` provides a simple scoping mechanism for managing these collections of variables (Example 3-6). All computational graph elements created within the scope of a `tf.name_scope(name)` call will have `name` prepended to their names.

This organizational tool is most useful when combined with TensorBoard, since it aids the visualization system in automatically grouping graph elements within the same name scope. You will learn more about TensorBoard further in the next section.

Example 3-6. Using namescopes to organize placeholders

```
>>> N = 5
>>> with tf.name_scope("placeholders"):
      x = tf.placeholder(tf.float32, (N, 1))
      y = tf.placeholder(tf.float32, (N,))
>>> x
<tf.Tensor 'placeholders/Placeholder:0' shape=(5, 1) dtype=float32>
```

Optimizers

The primitives introduced in the last two sections already hint at how machine learning is done in TensorFlow. You have learned how to add placeholders for datapoints and labels and how to use tensorial operations to define the loss function. The missing piece is that you still don't know how to perform gradient descent using TensorFlow.

While it is in fact possible to define optimization algorithms such as gradient descent directly in Python using TensorFlow primitives, TensorFlow provides a collection of optimization algorithms in the `tf.train` module. These algorithms can be added as nodes to the TensorFlow computation graph.

> **Which optimizer should I use?**
>
> There are many possible optimizers available in `tf.train`. For a short preview, this list includes `tf.train.GradientDescentOptimizer`, `tf.train.MomentumOptimizer`, `tf.train.AdagradOptimizer`, `tf.train.AdamOptimizer`, and many more. What's the difference between these various optimizers?
>
> Almost all of these optimizers are based on the idea of gradient descent. Recall the simple gradient descent rule we previously introduced:
>
> $$W = W - \alpha \nabla W$$
>
> Mathematically, this update rule is primitive. There are a variety of mathematical tricks that researchers have discovered that enable faster optimization without using too much extra computation. In general, `tf.train.AdamOptimizer` is a good default that is relatively robust. (Many optimizer methods are very sensitive to hyperparameter choice. It's better for beginners to avoid trickier methods until they have a good grasp of the behavior of different optimization algorithms.)

Example 3-7 is a short bit of code that adds an optimizer to the computation graph that minimizes a predefined loss `l`.

Example 3-7. Adding an Adam optimizer to TensorFlow computation graph

```
learning_rate = .001
with tf.name_scope("optim"):
  train_op = tf.train.AdamOptimizer(learning_rate).minimize(l)
```

Taking gradients with TensorFlow

We mentioned previously that it is possible to directly implement gradient descent algorithms in TensorFlow. While most use cases don't need to reimplement the contents of `tf.train`, it can be useful to look at gradient values directly for debugging purposes. `tf.gradients` provides a useful tool for doing so (Example 3-8).

Example 3-8. Taking gradients directly

```
>>> W = tf.Variable((3,))
>>> l = tf.reduce_sum(W)
>>> gradW = tf.gradients(l, W)
>>> gradW
[<tf.Tensor 'gradients/Sum_grad/Tile:0' shape=(1,) dtype=int32>]
```

This code snippet symbolically pulls down the gradients of loss `l` with respect to learnable parameter (`tf.Variable`) `W`. `tf.gradients` returns a list of the desired gradients. Note that the gradients are themselves tensors! TensorFlow performs symbolic differentiation, which means that gradients themselves are parts of the computational graph. One neat side effect of TensorFlow's symbolic gradients is that it's possible to stack derivatives in TensorFlow. This can sometimes be useful for more advanced algorithms.

Summaries and file writers for TensorBoard

Gaining a visual understanding of the structure of a tensorial program can be very useful. The TensorFlow team provides the TensorBoard package for this purpose. TensorBoard starts a web server (on localhost by default) that displays various useful visualizations of a TensorFlow program. However, in order for TensorFlow programs to be inspected with TensorBoard, programmers must manually write logging statements. `tf.train.FileWriter()` specifies the logging directory for a TensorBoard program and `tf.summary` writes summaries of various TensorFlow variables to the specified logging directory. In this chapter, we will only use `tf.summary.scalar`, which summarizes a scalar quantity, to track the value of the loss function. `tf.summary.merge_all()` is a useful logging aid that merges multiple summaries into a single summary for convenience.

The code snippet in Example 3-9 adds a summary for the loss and specifies a logging directory.

Example 3-9. Adding a summary for the loss

```
with tf.name_scope("summaries"):
  tf.summary.scalar("loss", l)
  merged = tf.summary.merge_all()

train_writer = tf.summary.FileWriter('/tmp/lr-train', tf.get_default_graph())
```

Training models with TensorFlow

Suppose now that we have specified placeholders for datapoints and labels, and have defined a loss with tensorial operations. We have added an optimizer node `train_op` to the computational graph, which we can use to perform gradient descent steps (while we may actually use a different optimizer, we will refer to updates as gradient descent for convenience). How can we iteratively perform gradient descent to learn on this dataset?

The simple answer is that we use a Python `for`-loop. In each iteration, we use `sess.run()` to fetch the `train_op` along with the merged summary op `merged` and the loss `l` from the graph. We feed all datapoints and labels into `sess.run()` using a feed dictionary.

The code snippet in Example 3-10 demonstrates this simple learning method. Note that we don't make use of minibatches for pedagogical simplicity. Code in following chapters will use minibatches when training on larger datasets.

Example 3-10. A simple example of training a model

```
n_steps = 1000
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  # Train model
  for i in range(n_steps):
    feed_dict = {x: x_np, y: y_np}
    _, summary, loss = sess.run([train_op, merged, l], feed_dict=feed_dict)
    print("step %d, loss: %f" % (i, loss))
    train_writer.add_summary(summary, i)
```

Training Linear and Logistic Models in TensorFlow

This section ties together all the TensorFlow concepts introduced in the previous section to train linear and logistic regression models upon the toy datasets we introduced previously in the chapter.

Linear Regression in TensorFlow

In this section, we will provide code to define a linear regression model in TensorFlow and learn its weights. This task is straightforward and you can do it without TensorFlow easily. Nevertheless, it's a good exercise to do in TensorFlow since it will bring together the new concepts that we have introduced throughout the chapter.

Defining and training linear regression in TensorFlow

The model for a linear regression is straightforward:

$$y = wx + b$$

Here w and b are the weights we wish to learn. We transform these weights into `tf.Variable` objects. We then use tensorial operations to construct the L^2 loss: