Classification with linear models

Preparing

- Copy the Default data (about credit card default) into your bootcamp workspace folder
- Install **scikit-learn**

Classification

- In a regression problem, the outputs are continuous numerical variables
- In a classification problem, the outputs are discrete classes
- How might we use a linear model to do classification?
- One approach is to use **Fisher's linear discriminant analysis (LDA)** and its variants, which are described in section 4.4 of the ISL book, and which I won't go into here
- The other is to do a **logistic regression**, or its generalization as a **sigmoid regression**, so named for reasons we will see below
- While LDA is still popular, and handles some situations that logistic regression is bad at, it also makes a few assumptions about the distribution of the X's that logistic regression enables us to drop; usually people go for logistic regression first
- Logistic regression also has a closer relationship with typical neural networks
- Let's start with a bad idea of how to build a linear classifier: let's just assign each class a number and then fit a linear regression
- Why is this a bad idea? First, let's make reference to the following example from the book, diagnosing a patient on the basis of their symptoms (whatever they may be: they would be the X's):

$$Y = \begin{cases} 1 & \text{if } \texttt{stroke}; \\ 2 & \text{if } \texttt{drug overdose}; \\ 3 & \text{if } \texttt{epileptic seizure}. \end{cases} \qquad Y = \begin{cases} 1 & \text{if } \texttt{epileptic seizure}; \\ 2 & \text{if } \texttt{stroke}; \\ 3 & \text{if } \texttt{drug overdose}. \end{cases}$$

Figure 1: image

- With the help of some pictures you can convince yourself that it **doesn't make sense to assert there is an ordering when there isn't**
- For this reason we almost always prefer to break down multiple classes into **multiple binary variables** in some way, so that we can construct **multiple separate** (though not necessarily independent) **classifiers**
- For example, for the example we could construct these variables as follows:

$$Y_1 = \begin{cases} 1 \text{ if epileptic seizure} \\ 0 \text{ otherwise} \end{cases} \qquad Y_2 = \begin{cases} 1 \text{ if stroke} \\ 0 \text{ otherwise} \end{cases}$$

- We could even throw in the third for good measure, even though it isn't strictly speaking necessary - This is an improvement, but there is still a reason not to do this - To understand this, consider what the regression is going to do: it will want to bring the line down toward zero whenever (i.e., for whatever values of X) there are **fewer** examples of whichever class is coded as **1**; vice versa when there are proportionally **more** - It can be easily shown that the line will try and fit to the **probabilities** of the two classes - But probabilities are bounded: they can't be below 0 or above 1; the linear regression doesn't know this - It will do okay as long as the probabilities don't get too extreme; but as they approach 1 (conversely, 0, for the other class), it will be less and less accurate - Consider the following:

- This behaviour is pathological: the classifier will always predict the first (0) class, but it seems obvious that there are at least some cases where it should predict the second (1) class
- The reason is that the high concentration of examples of the first class on the left drags the line very, very far down - but there should be a limit to how far the predictions can go down: zero!
- The traditional solution to this is simple: rather than building a linear model to predict probabilities directly, use a linear regression model to predict a number that, at a fuzzy level, "indicates how close to zero or to one the probability is"
- That is, we could think of some function that maps real numbers to numbers between zero and one, and get its **output** to match the probabilities

$$\begin{aligned} \hat{p} &= f(\hat{z}) \\ \hat{z} &= \beta_0 + \beta_1 X_1 + \cdots + \beta_k X_k \end{aligned}$$

- The fact that a linear regression model (i.e., a line) has no upper or lower bound, indicates therefore that this function will necessarily never actually **reach** zero or one (because what would it do afterward?)
- If the function is symmetrical and monotonic (i.e., always keeps going in the same direction), then it will necessarily look something like an **s**, or, if you prefer Greek, a sigma, thus be **sigmoid**, because at some point it has to asymptotically approach zero and one on either end
- The choice doesn't matter much for simple regression problems (it does in more complicated models like neural nets), but the usual choice is the

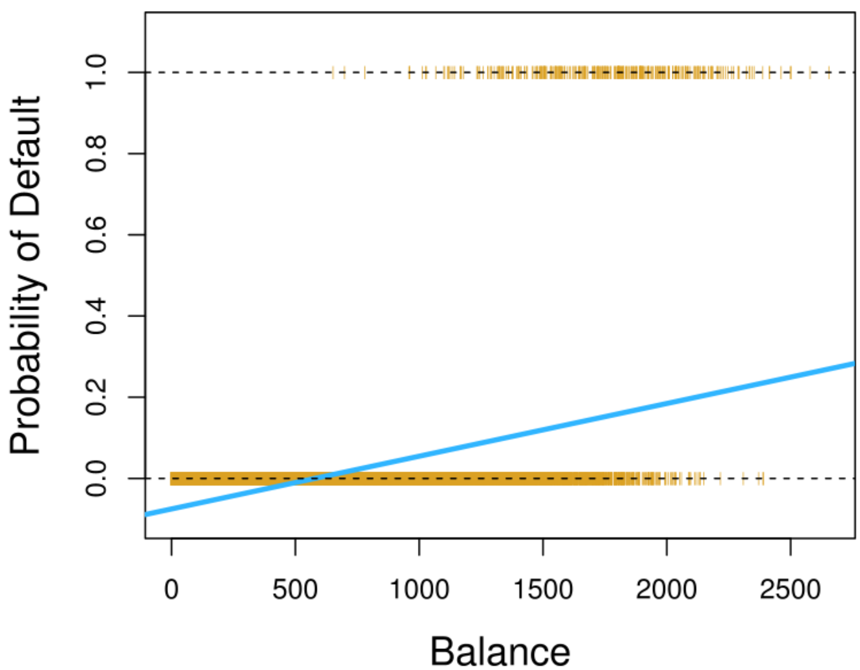

Figure 2: image

inverse logit or **logistic** function:

$$f(z) = \frac{e^z}{1 + e^z}$$

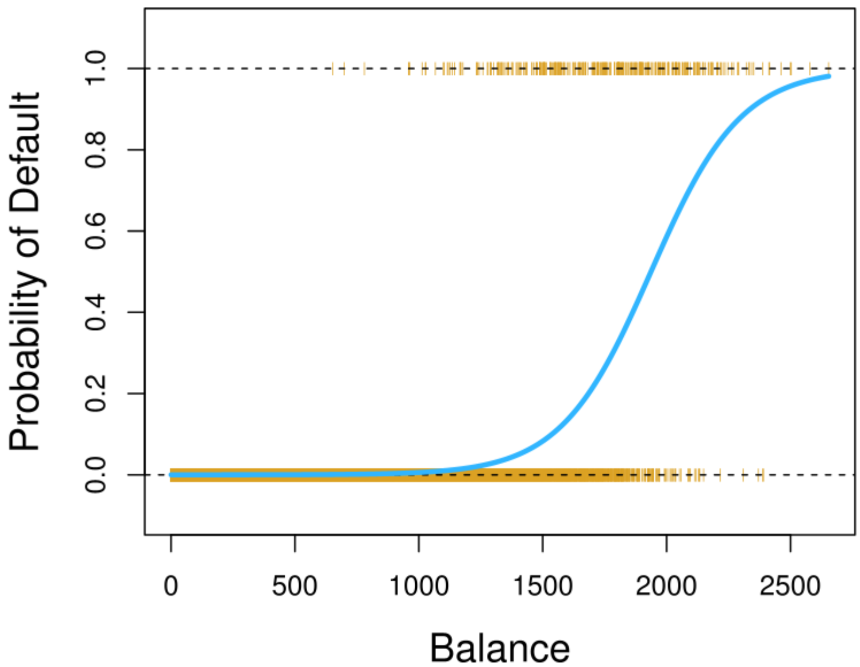

Figure 3: image

- Why this? Well, it is motivated by the notion of **odds**: if something has probability of 0.75, we say that it has **odds** of 3:1 - because that's the ratio of the two probabilities

$$\frac{p}{1-p}$$

- With a bit of algebra, we determine that

$$z = \log\left(\frac{e^z}{1+e^z}\right)$$

- What this means is, as z increases (decreases), the **log odds** of the category coded as 1 will increase (decreases)
- And what do we get if $z = 0$? (Think)

- And what do we expect the model to do if this prediction **isn't** true (i.e., how is that situation modelled)? (Think)
- The typical loss function is a little different than the least-squares loss - but you can look this up as it's not important for now
- It is perhaps worth it to know that, regardless of the loss function, there is no analytic solution

An example

- Load the Default data in an interactive Python session as **default** and examine the data frame
- Here is a wonky 3D plot:

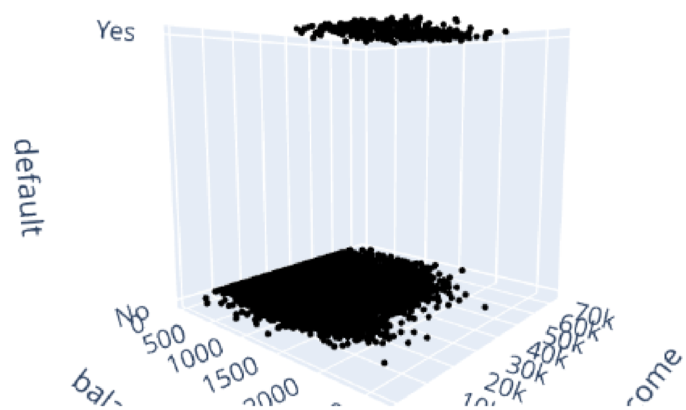

Figure 4: image

- We are going to fit a sigmoid regression using scikit-learn; I haven't told you how it's going to be fitted yet, but you might have guessed that the class variable (default) is going to need be converted into a 0-1 indicator
- You don't actually need to do this explicitly for what we are about to do (scikit-learn will pick a mapping for you), but sometimes you will need to, so for completeness, here is how I would do it if I had to:

```
default['default_code'] = [{'Yes': 1, 'No': 0}[x] for x in default['default']]
```

- Here is how we fit the model using scikit-learn and examine the coefficients

```
import sklearn.linear_model as slm
m = slm.LogisticRegression(penalty=None).fit(default[["balance", "income"]], default["defaul
print(m.intercept_)
```

```
print(m.coef_)
```

- Let's have a look at what it predicts:

```
predictions = m.predict_proba(default[["balance", "income"]])
print(predictions)
```

- Here is another wonky 3D graph:

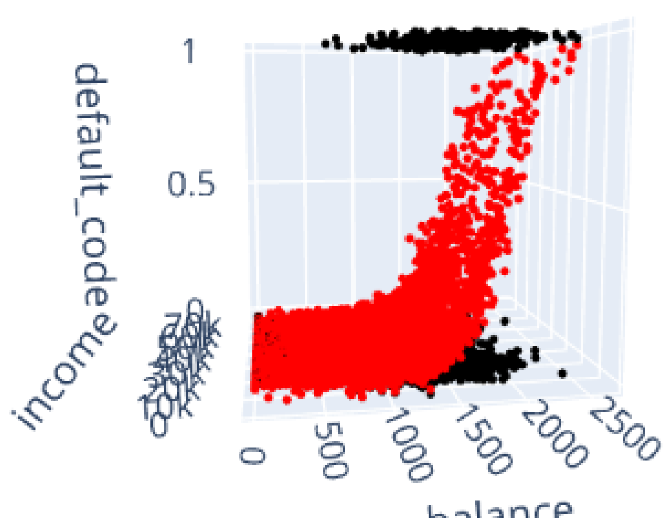

Figure 5: image

Discriminant functions

- We introduced the notion of linear classification **by rounding** - we predicted, in logistic regression, a **p** between 0 and 1, and rounded either up or down accordingly to get the predicted class
- We also saw that for logistic regression that this is equivalent to determining **whether z is positive or negative** (above or below zero)
- We call **z**, which is a linear function of the inputs, the **discriminant function**
- What I told you, but didn't show you visually, is that this means that, in the space of the input variables, there is a line or a plane which separates the two classes

$$
\begin{aligned}
0 &= \beta_0 + \beta_1 x_1 + \beta_2 x_2 \\
\Leftrightarrow \quad -\beta_2 x_2 &= \beta_0 + \beta_1 x_1 \\
\Leftrightarrow \quad x_2 &= -\frac{\beta_0}{\beta_2} - \frac{\beta_1}{\beta_2} x_1
\end{aligned}
$$

- I wasn't successfully able to get plotly to display or to save plots on the cluster (and I didn't have time to get to the bottom of the issue); this is the sort of thing I would not usually do remotely anyway, so feel free to install plotly locally and do the following, which is what I used to generate the plot below:

```
import plotly.express as px
import plotly.graph_objects as go

b02 = m.intercept_/m.coef_[0,1]
b12 = m.coef_[0,0]/m.coef_[0,1]
fig = px.scatter(default, x='balance', y='income', color="default")
fig.add_trace(go.Scatter(x=default['balance'], y=(-b02 - b12*default['balance']), name="slop
fig.update_yaxes(range=[0, 80000])
fig.show()
```

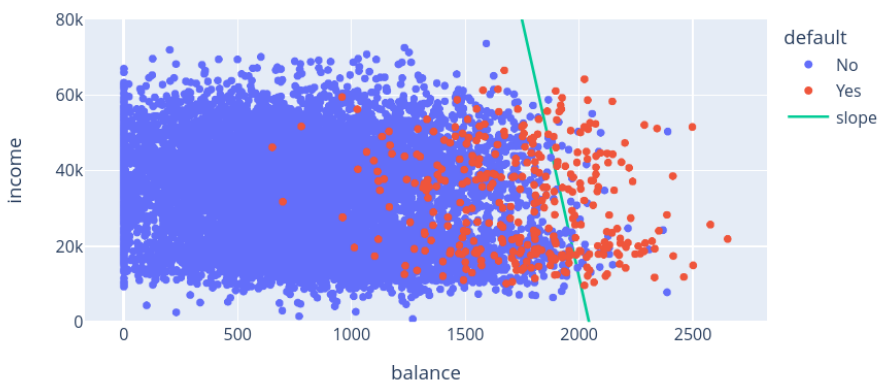

Figure 6: image