Logistic Regression

With an intro to sigmoid, softmax, and cross-entropy

Richard Corrado

Fat Cat Machine Learning

github.com/richcorrado

Goals

- Apply neural networks to study the MNIST digit classification problem.
- Use TensorFlow to accomplish this: requires low-level definitions of the models we will use.
- NNs use linear models to link layers and to output.
- Need to understand how to implement multiclass classification via linear models at a fairly low-level.
- Understand how to map linear output to class labels: sigmoid and softmax functions
- Understand appropriate cost function: cross-entropy

Ordinary Linear Regression

Design or Feature Matrix:

$$\mathbf{X} = \begin{matrix} \uparrow \\ \text{example} \\ \text{index} \\ \downarrow \end{matrix} \overset{\leftarrow \quad \text{feature} \quad \text{index} \quad \rightarrow}{\left(\begin{matrix} \\ \\ \\ \end{matrix} \right)}$$

Response (Vector):

$$\mathbf{y} = \begin{matrix} \uparrow \\ \text{example} \\ \text{index} \\ \downarrow \end{matrix} \left(\begin{matrix} \\ \\ \\ \end{matrix} \right)$$

We assume that $\mathbf{y}$ takes continuous values.

Linear Parameters:

$$\text{weights}: \quad \mathbf{W} = \begin{matrix} \uparrow \\ \text{feature} \\ \text{index} \\ \downarrow \end{matrix} \begin{pmatrix} & & \\ & & \\ & & \\ & & \end{pmatrix}$$

$$\text{bias}: \quad \mathbf{b} = \begin{matrix} \uparrow \\ \text{example} \\ \text{index} \\ \downarrow \end{matrix} \begin{pmatrix} b \\ \vdots \\ \vdots \\ b \end{pmatrix}$$

Then the output of a linear model

$$\hat{\mathbf{y}}(\mathbf{X}, \mathbf{W}, b) = \mathbf{X}\mathbf{W} + \mathbf{b}$$

is a vector of dimension (# of examples).

Maximum Likelihood Estimate

If $\mathbf{y}$ is a continuous response, it makes sense to assume that the errors between the true and predicted values

$$\epsilon = \mathbf{y} - \hat{\mathbf{y}}$$

are normally distributed, then conditional probability of reproducing $\mathbf{y}$ from the model is

$$p(\mathbf{y}|\mathbf{X}) = \mathcal{N}(\mathbf{y}; \hat{\mathbf{y}}, \sigma^2),$$

$$\mathcal{N}(\mathbf{y}; \hat{\mathbf{y}}, \sigma^2) = \frac{1}{(2\pi\sigma^2)^{n/2}} \exp\left(-\frac{1}{2\sigma^2}|\mathbf{y} - \hat{\mathbf{y}}|^2\right).$$

We want to maximize the probability of obtaining predictions that have a small error compared to the true values.

View $\mathbf{X}$ as fixed, then $p(\mathbf{y}|\mathbf{X}) = L(\mathbf{W}, b|\mathbf{X}, \mathbf{y})$ is the likelihood function for the parameters $\rightarrow$ find $\mathbf{W}, \mathbf{b}$ that maximize.

The natural logarithm is monotonically increasing, so equivalently maximize

$$\ln L(\mathbf{W}, b|\mathbf{X}, \mathbf{y}) = -\frac{1}{2\sigma^2}|\mathbf{y} - \hat{\mathbf{y}}|^2 - \ln\sqrt{2\pi\sigma^2},$$

or **minimize** the **cost function**:

$$J(\mathbf{W}, b) = |\mathbf{y} - \hat{\mathbf{y}}|^2,$$

by choosing appropriate parameters $\mathbf{W}, \mathbf{b}$. We recognize J as the residual sum of squares.

Gradient Descent

Cost function is minimized when

$$\nabla_{\mathbf{W}} J(\mathbf{W}, b) = \nabla_b J(\mathbf{W}, b) = 0.$$

Since

$$J(\mathbf{W}, b) = \mathrm{Tr}(\mathbf{XW} + \mathbf{b} - \mathbf{y})(\mathbf{XW} + \mathbf{b} - \mathbf{y})^T,$$

$$\nabla_{\mathbf{W}} J(\mathbf{W}, b) = 2(\mathbf{XW} + \mathbf{b} - \mathbf{y})^T \mathbf{X}.$$

This is a vector of dimension(# of features).

Consider the shift

$$\mathbf{W}' = \mathbf{W} - \epsilon(\mathbf{XW} + \mathbf{b} - \mathbf{y})^T\mathbf{X},$$

where $\epsilon > 0$. Then we can show that

$$J(\mathbf{W}', b) = J(\mathbf{W}, b) - 2\epsilon \mathrm{Tr}(\mathbf{XW}+\mathbf{b}-\mathbf{y})\mathbf{X}(\mathbf{XW}+\mathbf{b}-\mathbf{y})^T\mathbf{X} + \mathcal{O}(\epsilon^2).$$

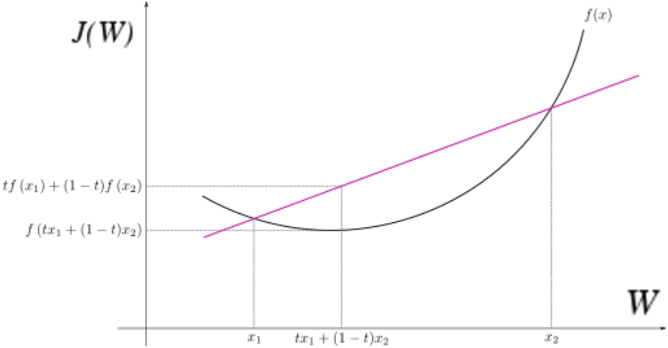
Therefore, for small enough ϵ, we have $J(\mathbf{W}', b) < J(\mathbf{W}, b)$, *i.e.*, we have reduced the cost function by this change of parameters.

Gradient descent algorithm:

while $J(\mathbf{W}, b) > \delta$: # tolerance parameter $\delta > 0$
$\quad \mathbf{W} = \mathbf{W} - \epsilon(\mathbf{XW} + \mathbf{b} - \mathbf{y})^T\mathbf{X}$

ϵ is usually called the **learning rate**.

For the linear model, the cost function is convex:



This implies that gradient descent will converge in a neighborhood of the true global minimum for appropriately small ϵ, δ.

For general optimization problems, gradient descent is not guaranteed to converge, or if it does, it might find a local minimum.

Binary Response

If the response $\mathbf{y}$ is not continuous, but discrete, the previous analysis based on Normal distribution of errors is invalid. Suppose that we have a binary response, taking values $y = 0, 1$. Now we need to specify $p(y = 1|\mathbf{X})$, since

$$p(y = 1|\mathbf{X}) + p(y = 0|\mathbf{X}) = 1.$$

Problem: find $\phi(z)$ so that:

$$p(y = 1|\mathbf{X}) = \phi(z), \quad z = \mathbf{XW} + b,$$

subject to $0 < \phi(z) < 1$, while $-\infty < z < \infty$.

Have:

$$-\infty < z < \infty,$$

$$0 < \phi(z) < 1.$$

Note that

$$-\infty < \ln \phi(z) < 0,$$

$$0 < -\ln(1 - \phi(z)) < \infty,$$

and so

$$-\infty < \ln \phi(z) - \ln(1 - \phi(z)) < \infty.$$

Then

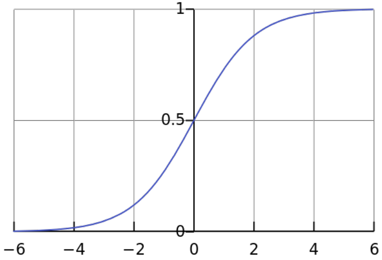
$$\ln\left(\frac{\phi(z)}{1 - \phi(z)}\right) = z,$$

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

is a reasonable choice. This is the **sigmoid function**.

Sigmoid or Logistic Function

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



- Rapidly changing near decision boundary $z = 0$.
- Well-behaved derivatives for gradient descent.

Bernoulli Distribution

$$p(y=1|\mathbf{X}) = \phi(z) = \frac{e^z}{1+e^z},$$

$$p(y=0|\mathbf{X}) = 1-\phi(z) = \frac{1}{1+e^z},$$

$$p(y|\mathbf{X}) = \phi(z)^y(1-\phi(z))^{1-y} = \frac{e^{yz}}{\sum_{y'=0}^{1} e^{y'z}}.$$

- $q_{y=1} = \phi(z)$ is model probability to find $y=1$.
- $q_{y=0} = 1-\phi(z)$ is model probability to find $y=0$.
- $p_{y=1} = y$ is true probability that $y=1$.
- $p_{y=0} = 1-y$ is is true probability that $y=0$.

Cost Function

As before, applying the maximum likelihood principal to $p(y|\mathbf{X})$ leads to minimizing the cost function

$$\begin{aligned} J(\text{one example}) &= -\ln p(y|\mathbf{X}) \\ &= -y \ln \phi(z) - (1-y)\ln(1-\phi(z)) \\ &= -p_{y=1} \ln q_{y=1} - p_{y=0} \ln q_{y=0} \\ &= -\sum_{y=0}^{1} p(y) \ln q(x) \\ &= \mathbb{E}_p \left[-\ln q \right]. \end{aligned}$$

This expectation value is called the **cross-entropy** between the model distribution $q(y)$ and the true distribution $p(y)$.

Multiclass Classification

Suppose now we have C classes, which is equivalent to $y = 0, 1, \ldots, C - 1$.

One vs. All Scheme:
For each class c, have a binary classification between $y = c$ and $y \neq c$.

One Hot Encoding:
Replace class labels with vector representation:

$$\begin{aligned} 0 &\rightarrow (1, 0, \ldots, 0) \\ 1 &\rightarrow (0, 1, \ldots, 0) \\ &\vdots \\ C - 1 &\rightarrow (0, 0, \ldots, 0, 1). \end{aligned}$$

Scikit-Learn LabelBinarizer

```python
import pandas as pd
from sklearn.preprocessing import LabelBinarizer
from sklearn.datasets import load_iris
lb = LabelBinarizer()
iris_data = load_iris()
lb.fit(iris_data.target)
label_vecs = lb.transform(iris_data.target)
labels_df = pd.DataFrame(label_vecs, columns = ['c_0', 'c_1', 'c_2'])
labels_df['label'] = iris_data.target
labels_df = labels_df[['label','c_0', 'c_1', 'c_2']]
labels_df.sample(n=5)
```

	label	c_0	c_1	c_2
41	0	1	0	0
2	0	1	0	0
88	1	0	1	0
70	1	0	1	0
131	2	0	0	1

Argmax Function

Note: Class label maps to index of nonzero element of class vector.

numpy.argmax

numpy.argmax(*a*, *axis=None*, *out=None*)

Returns the indices of the maximum values along an axis.

Parameters:	**a** : *array_like* Input array. **axis** : *int, optional* By default, the index is into the flatte

Map back to class labels:

```
print("np.argmax([1,0,0]) = %d" % np.argmax([1,0,0]))
print("np.argmax([0,1,0]) = %d" % np.argmax([0,1,0]))
print("np.argmax([0,0,1]) = %d" % np.argmax([0,0,1]))
```

```
np.argmax([1,0,0]) = 0
np.argmax([0,1,0]) = 1
np.argmax([0,0,1]) = 2
```

New Linear Model

$$\text{weights}: \quad \mathbf{W} = \begin{matrix} \uparrow \\ \text{feature} \\ \text{index} \\ \downarrow \end{matrix} \overset{\leftarrow \quad \text{class} \quad \text{index} \quad \rightarrow}{\begin{pmatrix} & & & \\ & & & \\ & & & \\ & & & \end{pmatrix}}$$

$$\text{bias}: \quad \mathbf{b} = \begin{matrix} \uparrow \\ \text{example} \\ \text{index} \\ \downarrow \end{matrix} \overset{\leftarrow \quad \text{class} \quad \text{index} \quad \rightarrow}{\begin{pmatrix} b & \cdots & \cdots & b \\ \vdots & & & \vdots \\ \vdots & & & \vdots \\ b & \cdots & \cdots & b \end{pmatrix}}$$

$$\mathbf{z}(\mathbf{X}, \mathbf{W}, b) = \mathbf{XW} + \mathbf{b}$$

is an array of dimension (# of examples) $\times$ (# of classes).

Class Probabilities

Convert $\mathbf{z}$ to class probabilities with **softmax function**:

$$\text{softmax}(\mathbf{z})_c = \frac{\exp(z_c)}{\sum_a \exp(z_a)},$$

$$\text{softmax}(\mathbf{z}) = \frac{1}{\sum_a e^{z_a}} \left(e^{z_0}, e^{z_1}, \ldots, e^{z_{C-1}} \right).$$

- Each element is in $[0, 1]$.
- Sum over elements $= 1$
- Maximum value of $\exp(z_c)$ determines the most probable class $\rightarrow$ can find it with `numpy.argmax`.

Cost Function

the new cost function is sometimes called the **softmax cross-entropy**

$$J(\text{example } i) = \sum_{i=c}^{C} y_{ic} \ln \text{softmax}(z)_{ic},$$

- $y_{ic} = 1$ iff example i is in class c.
- $\text{softmax}(z)_{ic}$ is the model probability that the example i is in class c.

Scikit-Learn LogisticRegression

```
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_iris
log_clf = LogisticRegression(penalty='l2',  n_jobs=-1)
iris_data = load_iris()
log_clf.fit(iris_data.data, iris_data.target)
proba = log_clf.predict_proba(iris_data.data)
pred = log_clf.predict(iris_data.data)
proba_df = pd.DataFrame(proba, columns = ['p_0', 'p_1', 'p_2'])
proba_df['argmax(p_i)'] = proba_df.idxmax(axis=1).str.strip('p_')
proba_df['y_pred'] = pred
proba_df['y_true'] = iris_data.target
proba_df.sample(n=5)
```

	p_0	p_1	p_2	argmax(p_i)	y_pred	y_true
129	0.000678	0.510705	0.488616	1	1	2
28	0.860034	0.139955	0.000010	0	0	0
78	0.013350	0.563206	0.423444	1	1	1
102	0.000278	0.330535	0.669186	2	2	2
59	0.033049	0.528709	0.438242	1	1	1

Review

- We've learned the necessary ingredients to use the output of a neural network to do multiclass classification at a low-level.
- This will be useful when we apply TensorFlow to build neural networks for, e.g., the MNIST digit problem.
- We've learned the role of the sigmoid, softmax and cross-entropy cost function in multiclass classification.
- We've seen some tools from numpy and scikit-learn that help us with one hot encoding and one vs. all classification schemes.