# Artificial Neural Networks: Linear Multiclass Classification (Part 3)

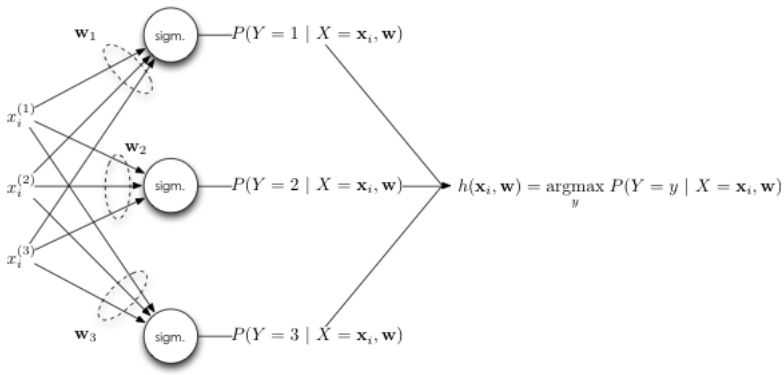
September 27, 2013 in ml primers, neural networks

In the last section, we went over how to use a linear neural network to perform classification. We covered using both the perceptron algorithm and gradient descent with a sigmoid activation function to learn the placement of the decision boundary in our feature space. However, we only covered binary classification. What if we instead want to classify a point belonging to one of $K$ classes?

## Theory

Multiclass classification using a linear neural network is a fairly simple extension of the binary classification setup. You may think that instead of outputting 0/1 from our second layer node, we could output 0, 1, ..., $K-1$. However, this is *not the case* . Our labels are not necessarily linear, and halfway between $\hat{y}=0$ and $\hat{y}=2$ is not necessarily $\hat{y}=1$. They are in fact categorical, and we use $k \in \{0, 1, \ldots, K\}$ out of computational convenience.

Consider instead representing a label using a binary vector of length $K$. Having a 1 in position $k$ corresponds to a label of $k$. Then, we can extend our linear network (with a sigmoid activation at the output) to learn how to output this vector. It would look something like the following:




Note that instead of $|\mathbf{w}| = M$ (where $M$ is the number of features), we instead have $|\mathbf{w}| = MK$. So in this figure, we have $K=3$ classes, and 3 features, giving us 9 weights in total. In fact, this is the neural network view of multinomial logistic regression. Recall the previous likelihood used in binary logistic regression:

$$P(Y=y \mid X=\mathbf{x}_i, \mathbf{w}) = \frac{1}{1+\exp(-y\mathbf{w}^T\mathbf{x}_i)}$$

To extend this to $K$ classes, we use the following likelihood:

$$P(Y=k \mid X=\mathbf{x}_i, \mathbf{w}) = \frac{\exp(\mathbf{w}_k^T\mathbf{x}_i)}{\sum_{k'}^{K}\exp(\mathbf{w}_{k'}^T\mathbf{x}_i)}$$

The key difference is that there are now $K$ sets of $M$ weights, one for each label. These are specified when determining the likelihood for a particular $k$. By using this formalism, we ensure that the values produced by the output nodes forms a valid probability distribution, as we are normalizing the likelihood by summing over all values of $k$.

Our training routine is exactly the same as in Part 2, except that the gradient of the multinomial logistic regression objective is slightly different:

$$\nabla_{\mathbf{w}_k}\ell(\mathbf{w}) = \frac{1}{N}\sum_{i}^{N}\left(\mathbf{x_i}(1-P(Y=y_i \mid X=\mathbf{x}_i, \mathbf{w}))\right)$$

See the Appendix for a full derivation.

## Implementation

The implementation of multiclass linear classification doesn't change much from the binary case, except for the gradient and how we label our data points. For this toy example, we'll be generating 3 clusters of two-dimensional data. To make things more interesting, we won't restrict them to be linearly separable.

Let's jump right into it. This code can be found in `ann_linear_multiclassification.py`. First, we generate three clusters of data, and give each a label of either 0, 1, or 2:

```
# Generate three random clusters of 2D data
N_c = 200
A = 0.6*np.random.randn(N_c, 2)+[1, 1]
B = 0.6*np.random.randn(N_c, 2)+[3, 3]
C = 0.6*np.random.randn(N_c, 2)+[3, 0]
X = np.hstack((np.ones(3*N_c).reshape(3*N_c, 1), np.vstack((A, B, C))))
Y = np.vstack(((np.zeros(N_c)).reshape(N_c, 1),
      np.ones(N_c).reshape(N_c, 1), 2*np.ones(N_c).reshape(N_c, 1)))
K = 3
N = K*N_c
```

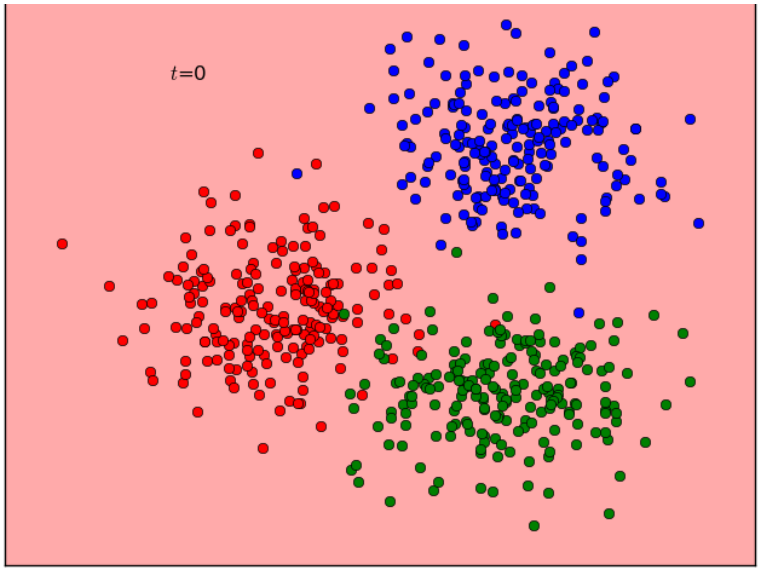
Next we run gradient descent using the multinomial logistic regression gradient:

```
# Run gradient descent
eta = 1E-2
max_iter = 1000
w = np.zeros((3, 3))
grad_thresh = 5
for t in range(0, max_iter):
   grad_t = np.zeros((3, 3))
   for i in range(0, N):
      x_i = X[i, :]
      y_i = Y[i]
      exp_vals = np.exp(w.dot(x_i))
      lik = exp_vals[int(y_i)]/np.sum(exp_vals)
      grad_t[int(y_i), :] += x_i*(1-lik)

   w = w + 1/float(N)*eta*grad_t
   grad_norm = np.linalg.norm(grad_t)

   if grad_norm < grad_thresh:
      print "Converged in ",t+1,"steps."
      break

   if t == max_iter-1:
      print "Warning, did not converge."
```

There are a couple of things to note here. First, our weight vector $\mathbf{w}$ is now actually a 3x3 matrix. We have 3 classes, so we need 3 sets of weights. Recall that each set of weights has a bias weight and a weight for each feature, giving them a length of $M + 1$. Therefore, each row of $\mathbf{w}$ corresponds to the weight for each $k \in \{0, 1, 2\}$. You can see this initialization in line 20.

In line 27, we calculate the *unnormalized* likelihood using numpy's `dot` function. `dot` computes the dot product between the input $\mathbf{x_i}$ and each of the weights in a row-wise fashion. In line 28, we generate the *normalized* likelihood that our datapoint $\mathbf{x}_i$ belongs to class 0, 1, or 2. Finally, in line 29, we use these likelihoods to calculate the gradient. This code follows the math exactly, so there shouldn't be any surprises here.

Running around 1,000 epochs with the given descent parameters will generate classification regions in the following manner:




## Code Download

2D multi-label classification Python code.

## Appendix

### Deriving the gradient for multinomial logistic regression

Our likelihood over the data for MLR is the following:

$$P(D_Y \mid D_X, \mathbf{w}) = \prod_i^N \frac{\exp(\mathbf{w}_{y_i}^T \mathbf{x}_i)}{\sum_{k'}^K \exp\left(\mathbf{w}_{k'}^T \mathbf{x}_i\right)}$$

The log likelihood is then:

$$\ell(\mathbf{w}) = \sum_i^N \left( \mathbf{w}_{y_i}^T \mathbf{x}_i - \log\left( \sum_{k'}^K \exp(\mathbf{w}_k^T \mathbf{x}_i) \right) \right)$$

We can then determine the gradient for a particular set of weights when $y_i = k$:

$$\begin{aligned}
\nabla_{\mathbf{w}_k} \ell(\mathbf{w}) &= \frac{\partial}{\partial \mathbf{w}_k} \sum_i^N \left( \mathbf{w}_k^T \mathbf{x}_i - \log\left( \sum_{k'}^K \exp(\mathbf{w}_k^T \mathbf{x}_i) \right) \right) \\
&= \sum_i^N \left( \frac{\partial}{\partial \mathbf{w}_k} \mathbf{w}_k^T \mathbf{x}_i - \frac{\partial}{\partial \mathbf{w}_k} \log\left( \sum_{k'}^K \exp(\mathbf{w}_k^T \mathbf{x}_i) \right) \right) \\
&= \sum_i^N \left( \mathbf{x}_i - \frac{\mathbf{x}_i \exp(\mathbf{w}_k^T \mathbf{x}_i)}{\sum_{k'}^K \exp(\mathbf{w}_k^T \mathbf{x}_i)} \right)
\end{aligned}$$

Recall that our likelihood is:

$$P(Y = k \mid X = \mathbf{x}_i, \mathbf{w}) = \frac{\exp(\mathbf{w}_k^T \mathbf{x}_i)}{\sum_{k'}^K \exp\left(\mathbf{w}_{k'}^T \mathbf{x}_i\right)}$$

So then our gradient is:

$$\nabla_{\mathbf{w}_k} \ell(\mathbf{w}) = \sum_i^N \left( \mathbf{x_i} (1 - P(Y = y_i \mid X = \mathbf{x}_i, \mathbf{w})) \right)$$

The gradient contribution of a particular example for all $\mathbf{w}_{k \neq y_i}$ is zero, so for each example we only need to update the particular set of weights corresponding to $k = y_i$.

Tags: neural network, multiclass, softmax, tutorial



Prev / Next

Comments (7) Newest First Subscribe via e-mail

Preview Post Comment…

**dehand** 9 months ago · 0 Likes

The theory looks very fine

**Alex Sepnov** 2 years ago · 0 Likes

I tried to run this problem... but it didn't converge. can u check again the code?

**Benjamin Deonovic** 2 years ago · 0 Likes

I ran your code and it seems to converge VERY VERY slowly. Can you comment on that?

**Brian Dolhansky** 2 years ago · 0 Likes

Sorry for the (very) late response, but I didn't do any hyper-parameter tuning in this case. This is not performant code - it's just an example. There are plenty of optimizations that could be added!

**Andrew Phelps Cassidy** 2 years ago · 0 Likes

Does this mean you MUST minimize cross entropy when doing multinomial classification? Is Sum of Squared Errors not appropriate?

**Brian Dolhansky** 2 years ago · 0 Likes

Sum of squared errors is used for regression - in this case we are doing classification. The reason we can't use sum of squared errors is that the class labels don't correspond to any real-world value. In other words, the class "3" doesn't mean it's 3 times "larger" than the class 1, they are just labels. Likewise, if your regression model wanted to indicate that an example was class 1 with probability 0.5 and class 3 with probability 0.5, that doesn't mean the example should be labelled as "2." The multinomial objective will instead assign a probability to each class.

**Jay** 2 years ago · 0 Likes

This is an useful explanation. Thanks.