

Multinomial Classification: from binary to many classes

What if our targets come from a class C with more than two discrete values?

$$C = \{c_1, \dots, c_{\#C}\}$$

where $\|C\| > 2$

This is called *Multinomial Classification*

Some models (e.g. Decision Trees) can handle Multinomial classification directly.

For those that don't, we adapt the approach used for Binary Classification.

Since the target for Classification is Categorical

- we turned the Binary Classification task
- into the task of *predicting the probability* that the example's target is Positive

When the number of classes is greater than 2

- we turn the Multinomial Classification task
- into the task of computing a *probability vector* $\mathbf{p}^{(i)}$ of length $\#C$
 $\mathbf{p}_j^{(i)} = \text{Probability that example } i \text{ is in class } C_j$

Aside

In Logistic Regression

- we used a Linear Model for the score (logit)
$$s = \Theta^T \mathbf{x}$$
- and used the sigmoid to convert it into a probability
$$\hat{p} = \sigma(\Theta^T \mathbf{x})$$

Our approach for Multinomial Classification will be similar

- predict a *vector* of scores/logits
 - s_j is the score for class c_j
- convert the vector of scores into a *probability vector* $\hat{\mathbf{p}}$

$$\sum_{j=1}^{\#C} \hat{\mathbf{p}}_j = 1$$

Note that *true* target/label $\mathbf{p}^{(i)}$

- has all the probability mass concentrated at a single class

But *predicted* probability vector $\hat{\mathbf{p}}^{(i)}$

- may have non-zero probabilities at more than one class

Classification asks us to output a choice in \mathcal{C} rather than a probability distribution over \mathcal{C}

So the final step of Multinomial Classification

- usually converts the predicted probability vector $\hat{\mathbf{p}}$
- into a single choice in C
- choice is often either of
 - c_k where $\hat{\mathbf{p}}_k$ is largest
$$\operatorname{argmax}_k \hat{\mathbf{p}}_k$$
 - sample k from distribution $\hat{\mathbf{p}}$

Multinomial classification using multiple binary classifiers

One versus all

The One versus All (OvA) method creates $|C|$ binary classifiers

- One for each $c \in C$
- The classifier for class c identifies
 - Positive examples as those having target c
 - Negative examples as those having targets other than c

For the binary classifier for class c , let

- $\hat{p}^c(\mathbf{x})$ denote the prediction of example \mathbf{x} being Positive (i.e., class c) made by this binary classifier

We can combine the individual binary predictions into a single probability vector $\hat{\mathbf{y}}$

Note that the sum of probabilities across independent binary classifiers may not equal 1.

We need to normalize the individual probabilities to create the OvA prediction vector

$$\hat{\mathbf{p}}_c(\mathbf{x}) = \frac{\hat{p}^c(\mathbf{x})}{\sum_{c' \in C} \hat{p}^{c'}(\mathbf{x})}$$

That is: it normalizes the probabilities so that they sum to 1 for each example.

Note

We have abused notation by using class c as a subscript of $\hat{\mathbf{y}}, \hat{p}$ rather than the integer j , where c is the j^{th} class in C .

Note that the **binary classifier for each class c** has it's own parameters Θ_c .

- So the number of parameters in the Θ for the OvA classifier is $\|C\|$ times as big as the number of parameters for a single classifier.

Let's be clear on the number of coefficients estimated in One versus All:

For the digit classification problem where there are $C = 10$ classes the number of parameters is *10 times* that of a binary classifier.

Fortunately, `sklearn` hides all of this from you.

What you *should* realize is that $\|C\|$ models are being fit, each with it's own parameters.

One versus one

The One versus One (OvO) method creates $\frac{||C||*(||C||-1)}{2}$ binary classifiers

- one for each pair c, c' of distinct values in C
- the classifier for pair c, c' identifies
 - Positive examples as those having target c
 - Negative examples as those having targets c'

Essentially, OvO creates a "competition" between pairs of classes for a given example \mathbf{x}

- the class that "wins" most often is chosen as the predicted class for the OvO classifier on example \mathbf{x}

Softmax

As an alternative to normalizing the individual probabilities of the per-class Binary Classifiers

- we can produce a vector of scores/logits
- normalize the score vector

The Multinomial generalization of Sigmoid is the *Softmax* function

$$\hat{\mathbf{p}}_c(\mathbf{x}) = \frac{\exp(s^c(\mathbf{x}))}{\sum_{c \in C} \exp(s^c(\mathbf{x}))}$$

where $s^c(\mathbf{x})$ is the score predicted by the Binary classifier for class c

By exponentiating the score, the softmax magnifies small differences in scores into larger difference in probability.

To illustrate: suppose we have two relatively close scores $s^c, s^{c'}$ such that

$$\frac{s^c}{s^{c'}} = M \approx 1$$

- If we normalize scores by dividing a score by the sum (across all scores)
 - $\frac{\hat{\mathbf{p}}_c(\mathbf{x})}{\hat{\mathbf{p}}_{c'}(\mathbf{x})} = M$
- If we normalize by softmax
 - $\frac{\hat{\mathbf{p}}_c(\mathbf{x})}{\hat{\mathbf{p}}_{c'}(\mathbf{x})} = \frac{\exp(M\hat{s}_{c'})}{\exp(\hat{s}_{c'})} = \exp(\hat{s}_{c'}(M - 1))$

Multinomial classification by generalizing the loss function

We will deal with the loss functions, both for Binary and Multinomial Classification in a separate module.

- For Binary Classification: the loss function is called Binary Cross Entropy
- The generalization of the loss function to Multinomial Classification is called *Cross Entropy*

Count the parameters !

It is surprisingly easy to wind up with a lot of parameters in Multinomial Classification.

Always count the number of parameters

- to avoid over-fitting

How many parameters did we fit (i.e., what is the size of Θ) ?

```
In [9]: print("The classifier non-intercept parameters shape: {nc}; intercept parameter  
s shape: {ni}".format(  
    nc=mnh.clf.coef_.shape,  
    ni=mnh.clf.intercept_.shape  
)  
    )
```

The classifier non-intercept parameters shape: (10, 784); intercept parameter
s shape: (10,)

sklearn separately stores

- the intercept (`clf.intercept_`): the parameter associated with the const column in \mathbf{X}')
- all other parameters (`clf.coef_`)

As you can see from the leading dimension (10) there are essentially $||C||$ binary classifiers

- One parameter per element of the feature vector
- Plus one intercept/constant parameter

In total Θ has $10 * (784 + 1) = 7850$ parameters.

More precisely

- The target vector \mathbf{y} is of length $\|C\| = 10$, i.e., OHE target
 - We have previously only seen scalar targets
- `LogisticRegression` is performing One versus All (OvA) classification
- Because $\|\mathbf{y}^{(i)}\| > 1$, it is using a Cross Entropy Loss in the Loss function

Compare this to the KNN classifier from the first lecture

- one template per example, at $(28 \times 28) = 784$ parameters per example
- times $m = 5000$ examples

So the Logistic Classifier uses about $m = 5000$ times fewer parameters.

Regularization

Too many parameters in a model makes it susceptible to over-fitting

- "memorizing training examples"
- resulting in poor generalization (out of sample)

In an earlier module (perhaps left for Recitation) we discussed

- [Regularization \(Bias and Variance.ipynb#Regularization:-reducing-overfitting\)](#)
- as a way of forcing parameters towards 0
 - combats over-fitting

Notice that our `LogisticRegression` used an L2 penalty (Ridge Regression)

- That caused about 16% of the parameters to become 0.

Pattern matching: visualizing the parameters

What do the 784 non-intercept parameters look like ?

That is: what is the "template" for each class (digit) ?

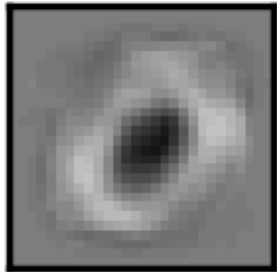
Since there is one parameter per pixel, ordered in the same way as the input image pixels

- We can display the 784 parameters as a (28×28) image.

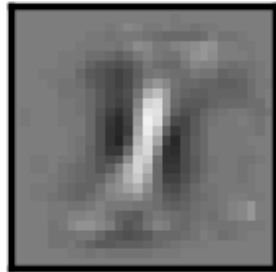
Remember: there is one parameter vector (template) for each of the $||C|| = 10$ classes.

```
In [10]: mnist_fig, mnist_ax = mnh.plot_coeff()
```

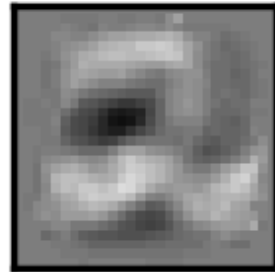
Parameters for...



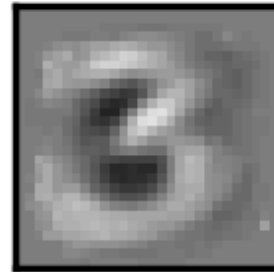
Class 0



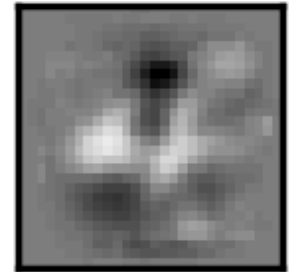
Class 1



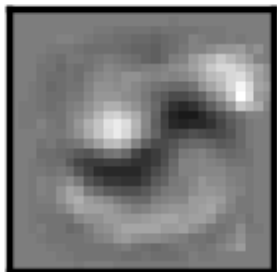
Class 2



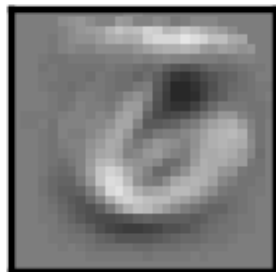
Class 3



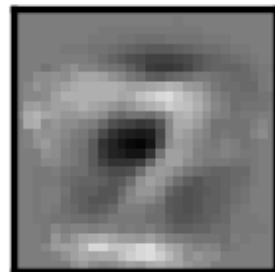
Class 4



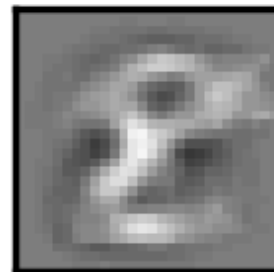
Class 5



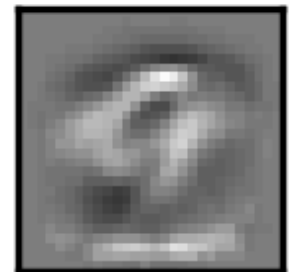
Class 6



Class 7



Class 8



Class 9

Our model learned a template, per digit, which hopefully captures the "essence" of the digit

- Fuzzy, since it needs to match many possible examples of the digit, each written differently

We will "interpret" these coefficients in a subsequent lecture but, for now:

- Dark colored parameters indicate the template for the pixel best matches dark input pixels
- Bright colored parameters indicate the template for the pixel best matches bright input pixels

So the "essence" of an image representing the "1" digit is a vertical band of bright pixels.

TIP The `fetch_mnist_784` routine in the module takes a **long** time to execute. Caching results makes you more productive.

```
In [11]: print("Done")
```

Done

