Linear Models

Jonathan May

August 27, 2024

1 Nominal task: text classifier

We have some book reviews. We want to know automatically if they're positive or negative.

Positive review:

```
I loved this book. The food was really good and fast (which is good
because I have a very packed schedule). Also great if you're on a
budget. The recipes have variations so I could eat different things but
not have to buy a whole new set of ingrediants.
```

Negative review:

```
I tried reading this book but found it so turgid and poorly written that
I put it down in frustration.  It reads like a translation from another
language by an academic bureacrat. The theme is interesting, the
execution poor.  Cannot recommend
```

2 Evaluation

We should always ask: 'How do we know we are doing well at what we are trying to do?' This time the answer is fairly simple (it won't always be). We collect many reviews along with their labels (Positive/Negative, which we may have to create out of some other labels, like a numerical score).

We'll calculate simple *accuracy*: $\frac{\#\text{ correct}}{\#\text{ total}}$.

3 Data

We will divide our data into *train*, *development* (dev) (also sometimes called 'validation') and *test* corpora. Train is used to build a model and is the largest data set (usually 80% of the data). Dev is not used to train but is frequently consulted during optimization (where relevant) to avoid *overfitting* on training data. Test is usually not evaluated or looked at except for when optimization is done, to ensure even less overfitting. Sometimes an even more super-secret *blind* test set is not even available to you but held off to test your final model.

4 A classifier model

Here's a simple framework for this problem:

```
def evaluate(sentence, option, model):
  # fill me in
  pass

def classify(sentence, options, model):
  scores = {}
  for option in options:
    scores[option] = evaluate(sentence, option, model)
  # key functions as argmax
  return max(scores, key=scores.get)
```

Let's consider methods for the 'evaluate' function:

5 Rule-Based (top-down?) Model

Positive reviews should have positive words, and negative reviews should have negative words. Thankfully, people have compiled such *sentiment lexicons* for English. See `http://www.enchantedlearning.com/wordlist`.

Positive words example: {absolutely, adorable, bountiful, bounty, cheery}

Negative words example: {angry, abysmal, bemoan, callous}

Let's put the intuition and data together:

```
# probably shouldn't structure your code this way...
model = {}
model['good'] = set(['yay', 'love', ...])
model['bad'] = set(['terrible', 'boo', ...])

def evaluate(sentence, option, model):
  score = 0
  for word in sentence.split():
    if word in model[option]:
      score+=1
  return score
```

6 Empirical Model

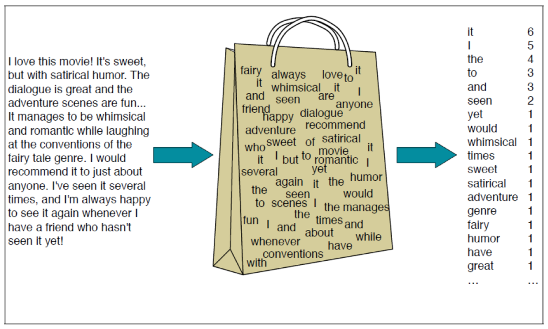
Our intuitions about word sentiment aren't perfect and neither are those of the people who made the word list. But we do have many examples of reviews and their sentiments. So we can make our own list of good and bad words. Instead of hard-coding the model as I did above, we can create a *training* function that takes in sentences *with their labels* and then returns a model:

```
from collections import Counter, defaultdict
def train(labeled_sentences):
  scores = defaultdict(lambda: Counter()) # doubly nested structure
  for sentence, label in labeled_sentences:
    for word in sentence.split():
      scores[word][label]+=1
  model = defaultdict(lambda: set())
  for word, table in scores.items():
    # the most frequent label associated with the word
    # key functions as argmax
    label = max(table, key=table.get)
    model[label].add(word)
  return model
```

We now have our first *supervised* model. We should back up and consider what we are actually trying to model from a probabilistic view. For one thing let's consider the actual probability of each label, rather than our ad-hoc adding method above.

Our classifier should choose $\text{argmax}_y P(y|s)$ for sentence s where y is one of a fixed set of labels. But we didn't consider s as some monotone thing; we considered the occurrence of each word as an event.

So we'll make an assumption called the *bag of words assumption* which is that for sentence $s = w_1 w_2 \dots w_n$, we say $P(y|s) = P(y|w_1, w_2, \dots, w_n)$. Note this now doesn't depend on the order of the words.[1]


I love this movie! It's sweet, but with satirical humor. The dialogue is great and the adventure scenes are fun... It manages to be whimsical and romantic while laughing at the conventions of the fairy tale genre. I would recommend it to just about anyone. I've seen it several times, and I'm always happy to see it again whenever I have a friend who hasn't seen it yet!

it	6
I	5
the	4
to	3
and	3
seen	2
yet	1
would	1
whimsical	1
times	1
sweet	1
satirical	1
adventure	1
genre	1
fairy	1
humor	1
have	1
great	1
...	...

Figure 7.1 Intuition of the multinomial naive Bayes classifier applied to a movie review. The position of the words is ignored (the *bag of words* assumption) and we make use of the frequency of each word.

Figure from J&M 3rd ed. draft, sec 7.1

But this model will only be valid if we can calculate probabilities of a label for the exact multiset of each sentence's words. We need another assumption, the *Naive Bayes assumption* which is that the probability of a label given a word is conditionally independent of the other words.

Note from Bayes' rule:

$$P(y|w_1, w_2, \dots, w_n) = \frac{P(w_1, w_2, \dots, w_n|y)P(y)}{P(w_1, w_2, \dots, w_n)}$$

[1] This assumption isn't really part of Naive Bayes, it's an assumption about the feature set being used. It's probably better to say that for x, y, we calculate $f(x, y) = f_1, \dots, f_n$ and then proceed from there. The above is a bit of a simplification. The reading avoids this simplification.

and since the word sequence itself is constant, we can say

$$\text{argmax}_y P(y|w_1, w_2, \ldots, w_n) = \text{argmax}_y P(w_1, w_2, \ldots, w_n|y)P(y)$$

(This means that we're optimizing the joint probability $P(s, y)$, but it's monotonic with $P(s|y)$.)

We can use the chain rule to break up the first term on the right:

$$P(w_1, w_2, \ldots, w_n|y) = P(w_1|y, w_2, \ldots, w_n)P(w_2|y, w_3, \ldots, w_n) \ldots P(w_n|y)$$

Then we apply the Naive Bayes assumption:

$$P(w_1|y, w_2, \ldots, w_n) \approx P(w_1|y)$$

and so on for all the other terms. That gives us

$$P(w_1, w_2, \ldots, w_n|y)P(y) \approx P(w_1|y)P(w_2|y), \ldots, P(w_n|y)P(y)$$

Is this a good model? George Box, statistician: "All models are wrong, but some models are useful."

What are the problems with the bag of words assumption?

What are the problems with the Naive Bayes assumption?

Does it work? Yes, for many tasks actually. And it's very simple, so it's usually worth trying.

Here's a new trainer:

```
from collections import Counter, defaultdict
wprobdenom = "__ALL__" # assume this token doesn't appear
def train(labeled_sentences):
  wscores = defaultdict(lambda: Counter()) # doubly nested structure
  cscores = Counter()
  for sentence, label in labeled_sentences:
    cscores[label]+=1
    for word in sentence.split():
      wscores[label][word]+=1
      wscores[label][wprobdenom]+=1
  model = {'cprobs':{}, 'wprobs':{}}
  for label in cscores.keys():
    model['cprobs'][label] = cscores[label]/len(labeled_sentences)
    wprob = {}
    for word, score in wscores[label].items():
      wprob[word] =  score/wscores[label][wprobdenom]
    model['wprobs'][label] = wprob
  return model
```

And a new, more appropriate classifier

```
def evaluate(sentence, option, model):
    score = model['cprobs'][option]
    for word in sentence.split():
        score *= model['wprobs'][option][word]
    return score
```

6.1 Practicalities: smoothing

`score *= model['wprobs'][option][word]` is going to be problematic if we have never seen a word with a particular class. Solution: smoothing!

Laplace (add-1) smoothing: assume you've seen every word (even words you haven't seen before) with every class!

Assume you have 10k words in the training set of negative class without Laplace smoothing:

$P(\text{amazing}|\text{negative}) = 0/10,000 = 0$ (seen the word but not with class 'negative')

$P(\text{blargh}|\text{negative}) = 0/10,000 = 0$ (never seen the word)

Now you introduce a new term, 'OOV'. If you haven't seen your test word during training, pretend your word is 'OOV'. Then, since you add 1 for each vocabulary word and the 'OOV' with each class, if your vocabulary size was 500, you now get:

$P(\text{amazing}|\text{negative}) = 1/10,501$

and

$P(\text{blargh}|\text{negative}) = 1/10,501$

You may want to actually *introduce* some 'OOV' into your training set and replace words that appear fewer than some k times with 'OOV'. This is so that your model can learn how to behave with 'OOV's.

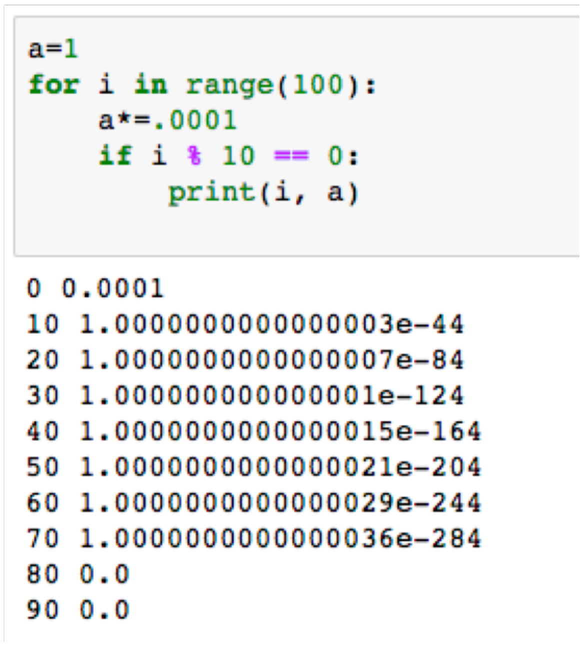
If you use a subword tokenizer like BPE, you can reduce the reliance on 'OOV' and smoothing, but you can't quite eliminate either one. Why not?

6.2 Practicalities: underflow

Recall:

```
    for word in sentence.split():
        score *= model['wprobs'][option][word]
```

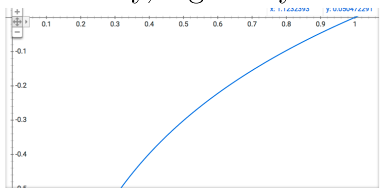
Sentence may be long! Probabilities may be small! It's very easy to run into underflow: try this:



```
a=1
for i in range(100):
    a*=.0001
    if i % 10 == 0:
        print(i, a)
```

```
0 0.0001
10 1.0000000000000003e-44
20 1.0000000000000007e-84
30 1.000000000000001e-124
40 1.0000000000000015e-164
50 1.0000000000000021e-204
60 1.0000000000000029e-244
70 1.0000000000000036e-284
80 0.0
90 0.0
```

Thankfully, logs are your friend, because of the following graph of $\log(x)$:



So instead rewrite the function as

```
from math import log

def evaluate(sentence, option, model):
  score = log(model['cprobs'][option])
  for word in sentence.split():
    score += log(model['wprobs'][option][word])
  return score
```

(Note the operator change)

7 problems with naive bayes assumption

We tolerated the naive bayes assumption: $P(w_1, w_2, \ldots, w_n|y)P(y) = P(w_1|y)P(w_2|y), \ldots, P(w_n|y)P(y)$ because it was useful, but we know that it is incorrect. Let's take a closer look:

So, by chain rule, $P(w_1, w_2, \ldots, w_n|y) = P(w_1|y, w_2, \ldots, w_n)P(w_2|y, w_3, \ldots, w_n) \ldots P(w_n|y)$ but by the Naive Bayes assumption we restrict the conditional to just be y.

Of course, this is wrong! Some words are clearly not conditionally independent of each other, i.e. $P(\text{San}|y) \neq P(\text{San}|y, \text{Francisco})$.

A more intuitive example: Imagine if 9/10 people recommended a movie to you. What if 8 of those 9 didn't actually see the movie but just repeated whatever the 9th person said to you?

Naive Bayes may in fact be a decent assumption for the specific case we've seen it in, and for most other words that occur alongside it (except for nearby words), but this is not the only *feature* we might care about. In particular we may care about overlapping features (e.g. the word, the word AND its predecessors/successors, prefix of the word, if the word is on certain lists of words, etc.).

8 Representation of features and weights for multi-class classification (esp. in Eisenstein)

We're going to talk about features in general but more importantly we're going to talk about feature weights, and especially if you've taken ML before you may be a bit confused by the notation in Eisenstein that I will borrow:

Sentence x: 'it rocks'

Label possibilities: 'Positive (+), Negative (−), Neutral (∅)'

Feature classes: "number of words" (nW), "contains word ending in s" ($*s$), "contains 'happy"' (ch)

Feature function f :

$f(x,+) = [2,1,0,0,0,0,0,0,0]$

$f(x,-) = [0,0,0,2,1,0,0,0,0]$

$f(x,\varnothing) = [0,0,0,0,0,0,2,1,0]$

For every feature we assign a weight. In the previous Naive Bayes discussion the features were all of the form 'contains x' for word x (could actually be 'number of times we see x'), and the weights were $P(x|y)$ (from the Naive Bayes assumption). As we see above, the features can be more arbitrary than that. We can arrange our weights in a weight vector, which by convention we'll call θ:

$\theta = [P(nW|+), P(*s|+), P(ch|+), P(nW|-), P(*s|-), P(ch|-), P(nW|\varnothing), P(*s|\varnothing), P(ch|\varnothing)]$

So given f and θ we can do 'inference' like so:

$\text{argmax}_{y \in \mathcal{Y}}\, \theta \cdot f(x,y)$

Eisenstein uses $\Psi(x,y) = \theta \cdot f(x,y)$; I call that the 'model score' or 'model cost'; it's the model's opinion of y being suitable for x.

What about $P(y)$? Note this is the background probabiliy of the class y. We can include this as a term that is always on for the set of features associated with class y. It's called the 'bias'. So revising the above,

$f(x,+) = [2,1,0,1,0,0,0,0,0,0,0,0]$ and so on...

$$\begin{aligned}\theta = [&P(nW|+), P(*s|+), P(ch|+), P(+),\\ &P(nW|-), P(*s|-), P(ch|-), P(-),\\ &P(nW|\varnothing), P(*s|\varnothing), P(ch|\varnothing), P(\varnothing)]\end{aligned}$$

9 perceptron

I called θ 'weights' for good reason – who says they have to be probabilities? We can adopt a 'trial and error' approach:

```
theta = random weights

for each sentence, label in data:
  if we would choose some other label over the correct one:
    modify theta so we don't do that
return theta
```

Here it is in a bit more gory detail using the framework from before:

```python
import numpy as np

# note that evaluate needs to be rewritten to be more general;
# left as an exercise to the reader, but should include a features() method

def train(labeled_sentences, options, featsize):
  # should be a better way to determine feat size
  # probably want to initialize differently
  model = {'theta': np.random(featsize)} # or zeroes
  for i in range(iterations): # user-determined
    for sentence, label in labeled_sentences:
      hyp = classify(sentence, options, model)
      if hyp != label:
        model['theta'] += features(sentence, label)-features(sentence, hyp)
  return model
```

Quick illustration: Some feature type $f1$ has value 2, $f2$ has value 1, $f3$ has value 0. There are three classes.

feats	$f(x,1)$	$f(x,2)$	$f(x,3)$	θ
$f1_1$	2	0	0	0.3
$f2_1$	1	0	0	0.7
$f3_1$	0	0	0	0.8
$f1_2$	0	2	0	-0.2
$f2_2$	0	1	0	2.2
$f3_2$	0	0	0	-.4
$f1_3$	0	0	2	-4
$f2_3$	0	0	1	-4
$f3_3$	0	0	0	-4

Let's say class 1 is correct. Given the table above, we get the following model costs:

$\Psi(x,1) = 1.3; \Psi(x,2) = 1.8; \Psi(x,3) = -12.$

So update $\theta = \theta + f(x,1) - f(x,2)$:

feats	$f(x,1)$	$f(x,2)$	$f(x,3)$	θ
$f1_1$	2	0	0	2.3
$f2_1$	1	0	0	1.7
$f3_1$	0	0	0	0.8
$f1_2$	0	2	0	-2.2
$f2_2$	0	1	0	1.2
$f3_2$	0	0	0	-.4
$f1_3$	0	0	2	-4
$f2_3$	0	0	1	-4
$f3_3$	0	0	0	-4

Class 2's weights went down (if they affected the outcome) and class 1's went up. Class 3's weights weren't involved at all. Now we get the following model costs:

$\Psi(x,1) = 6.3; \Psi(x,2) = -3.2; \Psi(x,3) = -12.$

So the item is correctly classified.

Some things to discuss in the context of machine learning: averaging all θ at the end, learning rates, and batch sizes.

10 loss function justification for perceptron

This seemed to work in the demo case above but has a very ad-hoc feel to it. (Side note: oftentimes models *are* originally designed in an ad-hoc way and only later did the theory work out. The original paper on perceptrons from 1957 has, AFAICT, nothing regarding the following) Why does this work?

It's worth considering the *loss* (often, though not always, written as ℓ) of the model. Remember, the model conveys an opinion about how to classify[2] that is to some degree untrue. Loss can be thought of as 'how wrong the model is.' For perceptron the loss for item i in a data set is (from Eisenstein):

$$\ell_{\text{perceptron}}(\theta; \mathbf{x}^{(i)}, y^{(i)}) = \max_{y \in \mathcal{Y}} \theta \cdot f(\mathbf{x}^{(i)}, y) - \theta \cdot f(\mathbf{x}^{(i)}, y^{(i)})$$

The first term ($\max_{y \in \mathcal{Y}} \theta \cdot f(\mathbf{x}^{(i)}, y)$) is exactly how we pick a label, and the second term is the model score of the right label. If we picked the right label, the loss is zero. Otherwise, the model score for the wrong label is higher, and the amount higher is how wrong we are.

Why do we have loss[3]? Because there's something wrong with our model, i.e. θ. But we can change that. How much should we change it? To minimize the loss, of course!

ℓ is an equation, and we can take its derivative with respect to θ. By adjusting θ in the negative direction of the gradient we will have a lower loss on our training data.

Let's assume $\hat{y}$ is the hypothesis y and that it's not the true label; what's the derivative? Well what's the equation?

Let's call $\hat{\mathbf{f}}$ $f(\mathbf{x}, \hat{y})$ and $\mathbf{f}$ $f(\mathbf{x}, y)$ (I dropped the superscripts; too hard to type) and make them subscriptable. Then $\ell = \theta_1 \hat{f}_1 - \theta_1 f_1 + \theta_2 \hat{f}_2 - \theta_2 f_2 + \ldots + \theta_n \hat{f}_n - \theta_n f_n$. Then

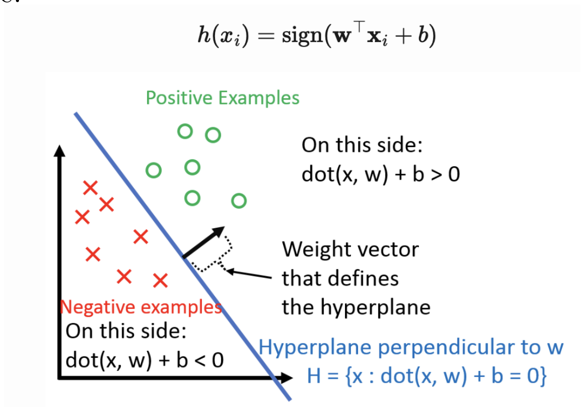
[2]Or for regression models, the value associated with an input

[3]Assuming our data is *separable*, which see below

$\partial\ell/\partial\theta_1 = \hat{f}_1 - f_1$ or to be more vector wise about it, $\partial\ell/\partial\theta = \hat{\mathbf{f}} - \mathbf{f}$. So the negative of that is $\mathbf{f} - \hat{\mathbf{f}}$.

You might wonder 'isn't there a closed form of this?'[4] Yes, there is, in the sense of this being a solution for a general linear model. Let the features be matrix A of dimension $N \times F$ for N training data and F features, and let numeric (integral?) labels be a vector B $(N \times 1)$[5]. Let the weights be vector W $(F \times 1)$. Then a linear model can be expressed $A \times W = B$. Then $W = A^{-1}B$ but usually A^{-1} isn't invertible, so $W = (A^TA)^{-1}A^TB$.[6] But for large N that inverse operation is quite slow, so this isn't usually done.

BTW, another way to look at what the perceptron is doing is finding the *separating hyperplane* between differently labeled samples. This is easiest to visualize in the binary case:



The weights define a line/plane/hyperplane along which the score is zero; we want all the differently labeled examples to be divided by that separator.

11 support vector machine

Very briefly, perceptron updates are as follows:

$$\hat{y} = \text{argmax}_y\, \theta \cdot f(x^{(i)}, y)$$

If $\hat{y} = y^{(i)}$ then no update, otherwise

$$\theta = \theta + f(x^{(i)}, y^{(i)}) - f(x^{(i)}, \hat{y})$$

Introduce cost function for making a mistake $c(y^{(i)}, y)$; (for instance, let's just say cost is 1 if y is wrong, 0 otherwise). Change the updates as follows:

$$\hat{y} = \text{argmax}_y\, \theta \cdot f(x^{(i)}, y) + c(y^{(i)}, y)$$

$$\theta = \lambda\theta + f(x^{(i)}, y^{(i)}) - f(x^{(i)}, \hat{y})^7$$

[4] i.e. a single equation that can be solved to give ideal weights, as in Naive Bayes

[5] I think any unique value per label should do but FYI the literature I'm seeing, which is all over the place on if this is even possible, at best shows examples with labels 1 and -1 only.

[6] $(A^TA)^{-1}A^T$ is a pseudo-inverse of A.

[7] In Eisenstein $(1 - \lambda)$ is used and this fits the derivation better.

This is going to choose $\hat{y}$ that are close to $y^{(i)}$ instead of useless updates with the correct values, and it scales the previous weights to avoid overfitting. This is called *support vector machine* and has a nice derivation that we would spend time on in an ML class, but the upshot is it can work better than perceptron (see the Pang et al. paper in the reading, for instance).

12 logistic regression

One nice thing about the naive bayes model is that it's probabilistic, so if your classifier is one part of a pipeline, you can tell the rest of the pipeline your confidence in your output in a rational way. $\Psi(x, y)$ has range $(-\infty, \infty)$ which is less helpful. Another issue with perceptron/svm is that they are only concerned with making sure the correct answer is chosen/has large margin (in training).

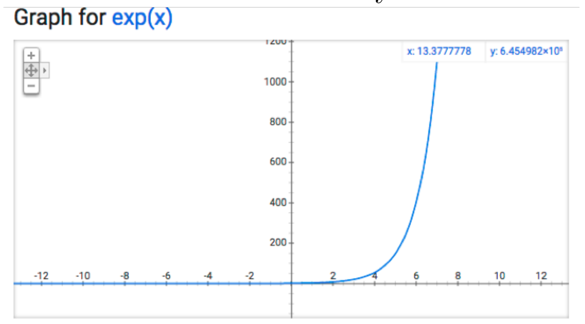
Nobody forced us to keep $\Psi(x, y)$ as the model score. A preferred model score would be $P(y|x)$, the conditional probability of the output given the input. How do we form a probability distribution from a set of scores?

We can't simply normalize:

$$\frac{\Psi(x, y)}{\sum_{y' \in \mathcal{Y}} \Psi(x, y')}$$

will not work. Why?

Instead we can use e^{Ψ}. Why?



So our new model cost will be

$$\frac{e^{\Psi(x,y)}}{\sum_{y' \in \mathcal{Y}} e^{\Psi(x,y')}} \tag{1}$$

This, by the way, is the 'softmax' function that comes up a lot in machine learning. It does exactly the same job we are trying to do here: convert a set of numbers in the range $(-\infty, \infty)$ into a distribution while keeping the ordering the same. In fact, it generally 'peaks' the highest number; that's why this can be thought of as a 'soft' form of the 'max' operator (hence the name).

Now that we've got a distribution we have a natural loss function we can use: cross-entropy! $H(p, q) \triangleq -E_p \log q$, where E_p is the *expected value* w/r/t p. Concretely:

$$H(p, q) = -\sum_{x \in \mathcal{X}} p(x) \log q(x)$$

Where here, $q(x) = q(y|x) =$ Equation 1. From information theory this is the average number of bits[8] needed to identify an item if q is used to identify the item, but p is the true distribution. So intuitively you want as small a cross-entropy as possible, and that makes this a natural loss function. But what is $p(y|x)$, the true distribution? A good choice for supervised learning is to assume that the provided label is the truth and should always occur and that all other labels should never occur. That means that our calculation of cross entropy, which will in practice use training set $\mathcal{X}$ and label space $\mathcal{Y}$:

$$-\frac{\sum_{x,y\in\mathcal{X}}\sum_{y'\in\mathcal{Y}} p(y'|x)\log q(y'|x)}{|\mathcal{X}|}$$

becomes

$$-\frac{\sum_{x,y\in\mathcal{X}}\log q(y|x)}{|\mathcal{X}|}$$

since $p(y|x) = 1$ and $\forall y' \in \mathcal{Y} \setminus y, p(y'|x) = 0$.

Now considering a single item and substituting back in Equation 1:

$$-\log\frac{\exp(\theta\cdot f(\mathbf{x},y))}{\sum_{y'\in\mathcal{Y}}\exp(\theta\cdot f(\mathbf{x},y'))}$$

$$-\theta\cdot f(\mathbf{x},y) + \log\sum_{y'\in\mathcal{Y}}\exp(\theta\cdot f(\mathbf{x},y'))$$

Now, the gradient:

$$\partial\ell/\partial\theta = -f(\mathbf{x},y) + \frac{1}{\sum_{y''\in\mathcal{Y}}\exp(\theta\cdot f(\mathbf{x},y''))}\sum_{y'\in\mathcal{Y}}\exp(\theta\cdot f(\mathbf{x},y'))f(\mathbf{x},y')$$

$$\partial\ell/\partial\theta = -f(\mathbf{x},y) + \sum_{y'\in\mathcal{Y}}\frac{\exp(\theta\cdot f(\mathbf{x},y'))}{\sum_{y''\in\mathcal{Y}}\exp(\theta\cdot f(\mathbf{x},y''))}f(\mathbf{x},y')$$

$$\partial\ell/\partial\theta = -f(\mathbf{x},y) + \sum_{y'\in\mathcal{Y}} q(y'|\mathbf{x};\theta)f(\mathbf{x},y')$$

That second term is the expectation of $Y|X$, which is to say, the probability (according to the model) of a value times that value, summed over all possible values. Naturally, we *reduce* θ by (some fraction of) this value.

Note that we are now considering not only how far away each wrong answer is from the right answer, but how confident we are about each wrong answer. If we have low confidence about an answer it will affect the loss very little.

[8]really, nats