Minimizing the Negative Log-Likelihood, in English

Will Wolf

May 18, 2017

Roughly speaking, my machine learning journey began on Kaggle (http://kaggle.com). "There's data, a model (i.e. estimator) and a loss function to optimize," I learned. "Regression models predict continuous-valued real numbers; classification models predict 'red,' 'green,' 'blue.' Typically, the former employs the mean squared error or mean absolute error; the latter, the cross-entropy loss. Stochastic gradient descent updates the model's parameters to drive these losses down." Furthermore, to fit these models, just `import sklearn`.

A dexterity with the above is often sufficient for — at least from a technical stance — both employment and impact as a data scientist. In industry, commonplace prediction and inference problems — binary churn, credit scoring, product recommendation and A/B testing, for example — are easily matched with an off-the-shelf algorithm plus proficient data scientist for a measurable boost to the company's bottom line. In a vacuum I think this is fine: the winning driver does not *need* to know how to build the car. Surely, I've been this person before.

Once fluid with "scikit-learn fit and predict," I turned to statistics. I was always aware that the two were related, yet figured them ultimately parallel sub-fields of my job. With the former, I build classification models; with the latter, I infer signup counts with the Poisson distribution and MCMC — right?

Before long, I dove deeper into machine learning — reading textbooks, papers and source code and writing this blog. Therein, I began to come across *terms I didn't understand used to describe the things that I did.* "I understand what the categorical cross-entropy loss is, what it does and how it's defined," for example.

Marginally wiser, I now know two truths about the above:

1. Techniques we anoint as "machine learning" - classification and regression models, notably - have their underpinnings almost entirely in statistics. For this reason, terminology often flows between the two.
2. None of this stuff is new.

The goal of this post is to take three models we know, love, and know how to use and explain what's really going on underneath the hood. I will assume the reader is familiar with concepts in both machine learning and statistics, and comes in search of a deeper understanding of the connections therein. There will be math — but only as much as necessary. Most of the derivations can be skipped without consequence.

When deploying a predictive model in a production setting, it is generally in our best interest to `import sklearn`, i.e. use a model that someone else has built. This is something we already know how to do. As such, this post will start and end here: your head is currently above water; we're going to dive into the pool, touch the bottom, then work our way back to the surface. Lemmas will be written in .

Error preparing HTML-CSS output (preProcess)

First, let's meet our three protagonists. We'll define them in Keras (https://keras.io/) for the illustrative purpose of a unified and idiomatic API.

Linear regression (http://ufldl.stanford.edu/tutorial/supervised/LinearRegression/) with mean squared error

```
input = Input(shape=(10,))
output = Dense(1)(input)

model = Model(input, output)
model.compile(optimizer=_, loss='mean_squared_error')
```

Logistic regression (http://ufldl.stanford.edu/tutorial/supervised/LogisticRegression/) with binary cross-entropy loss

```
input = Input(shape=(10,))
output = Dense(1, activation='sigmoid')(input)

model = Model(input, output)
model.compile(optimizer=_, loss='binary_crossentropy')
```

Error preparing HTML-CSS output (preProcess)

Softmax regression (http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression/) with categorical cross-entropy loss

```
input = Input(shape=(10,))
output = Dense(3, activation='softmax')(input)

model = Model(input, output)
model.compile(optimizer=_, loss='categorical_crossentropy')
```

Next, we'll select four components key to each: its response variable, functional form, loss function and loss function plus regularization term. For each model, we'll describe the statistical underpinnings of each component — the steps on the ladder towards the surface of the pool.

Before diving in, we'll need to define a few important concepts.

Random variable

I define a random variable as "a thing that can take on a bunch of different values."

- "The tenure of despotic rulers in Central Africa" is a random variable. It could take on values of 25.73 years, 14.12 years, 8.99 years, ad infinitum; it could not take on values of 1.12 million years, nor -5 years.
- "The height of the next person to leave the supermarket" is a random variable.
- "The color of shirt I wear on Mondays" is a random variable. (Incidentally, this one only has ~3 unique values.)

Probability distribution

A probability distribution is a lookup table for the likelihood of observing each unique value of a random variable. Assuming a given variable can take on values in *[Math Processing Error]*, the following is a valid probability distribution:

```
p = {'rain': .14, 'snow': .37, 'sleet': .03, 'hail': .46}
```

Trivially, these values must sum to 1.

- A *probability mass function* is a probability distribution for a discrete-valued random variable.
- A *probability density function* a probability distribution for a continuous-valued random variable.
  - *Gives*, because this function itself is not a lookup table. Given a random variable that takes on values in *[Math Processing Error]*, we do not and cannot define *[Math Processing Error]*, *[Math Processing Error]*, *[Math Processing Error]*, etc.
  - Instead, we define a function that tells us the probability of observing a value within a certain *range*, i.e. *[Math Processing Error]*.
  - This is the probability density function, where *[Math Processing Error]*.

Entropy

Entropy quantifies the number of ways we can reach a given outcome. Imagine 8 friends are splitting into 2 taxis en route to a Broadway show. Consider the following two scenarios:

- *Four friends climb into each taxi.* We could accomplish this with the following assignments:

```
# fill the first, then the second
assignment_1 = [1, 1, 1, 1, 2, 2, 2, 2]

# alternate assignments
assignment_2 = [1, 2, 1, 2, 1, 2, 1, 2]

# alternate assignments in batches of two
assignment_3 = [1, 1, 2, 2, 1, 1, 2, 2]

# etc.
```

- *All friends climb into the first taxi.* There is only one possible assignment.

Error preparing HTML-CSS output (preProcess)

```
assignment_1 = [1, 1, 1, 1, 1, 1, 1, 1]
```

(In this case, the Broadway show is probably in West Africa (http://willtravellife.com/2013/04/how-does-a-west-african-bush-taxi-work/) or a similar part of the world.)

Since there are more ways to reach the first outcome than there are the second, the first outcome has a higher entropy.

More explicitly

We compute entropy for probability distributions. This computation is given as:

[Math Processing Error]
where:

- There are *[Math Processing Error]* unique events.
- Each event *[Math Processing Error]* has probability *[Math Processing Error]*.

Entropy is the *weighted-average log probability* over possible events, which measures the *uncertainty inherent in their probability distribution.* The higher the entropy, the less certain we are about the value we're going to get.

Let's calculate the entropy of our distribution above.

```
p = {'rain': .14, 'snow': .37, 'sleet': .03, 'hail': .46}

def entropy(prob_dist):
    return -sum([ p*log(p) for p in prob_dist.values() ])

In [1]: entropy(p)
Out[1]: 1.1055291211185652
```

For comparison, let's assume two more distributions and calculate their respective entropies.

```
p_2 = {'rain': .01, 'snow': .37, 'sleet': .03, 'hail': .59}

p_3 = {'rain': .01, 'snow': .01, 'sleet': .03, 'hail': .95}

In [2]: entropy(p_2)
Out[2]: 0.8304250977453105

In [3]: entropy(p_3)
Out[3]: 0.2460287703075343
```

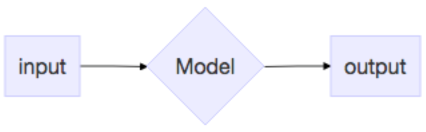
In the first distribution, we are least certain as to what tomorrow's weather will bring. As such, this has the highest entropy. In the third distribution, we are almost certain it's going to hail. As such, this has the lowest entropy.

Finally, it is a probability distribution that dictates the different taxi assignments just above. A distribution for a random variable that has many possible outcomes has a higher entropy than a distribution that gives only one.

Now, let's dive into the pool. We'll start at the bottom and work our way back to the top.

Response variable

Roughly speaking, each model looks as follows. It is a diamond that receives an input and produces an output.



The models differ in the type of response variable they predict, i.e. the *[Math Processing Error]*.

- Linear regression predicts a continuous-valued real number. Let's call it `temperature`.
- Logistic regression predicts a binary label. Let's call it `cat or dog`
- Softmax regression predicts a multi-class label. Let's call it `red or green or blue`.

Error preparing HTML-CSS output (preProcess)

In each model, the response variable can take on a bunch of different values. In other words, they are *random variables.* Which probability distributions are associated with each?

Unfortunately, we don't know. All we do know, in fact, is the following:

- `temperature` has an underlying true mean *[Math Processing Error]* and variance *[Math Processing Error]*.
- `cat or dog` takes on the value `cat` or `dog`. The likelihood of observing each outcome does not change over time, in the same way that *[Math Processing Error]* for a fair coin is always *[Math Processing Error]*.
- `red or green or blue` takes on the value `red` or `green` or `blue`. The likelihood of observing each outcome does not change over time, in the same way that the probability of rolling a given number on fair die is always *[Math Processing Error]*.

For clarity, each one of these assumptions is utterly banal — "so lacking in originality as to be obvious and boring." *Can we use them nonetheless to select probability distributions for our random variables?*

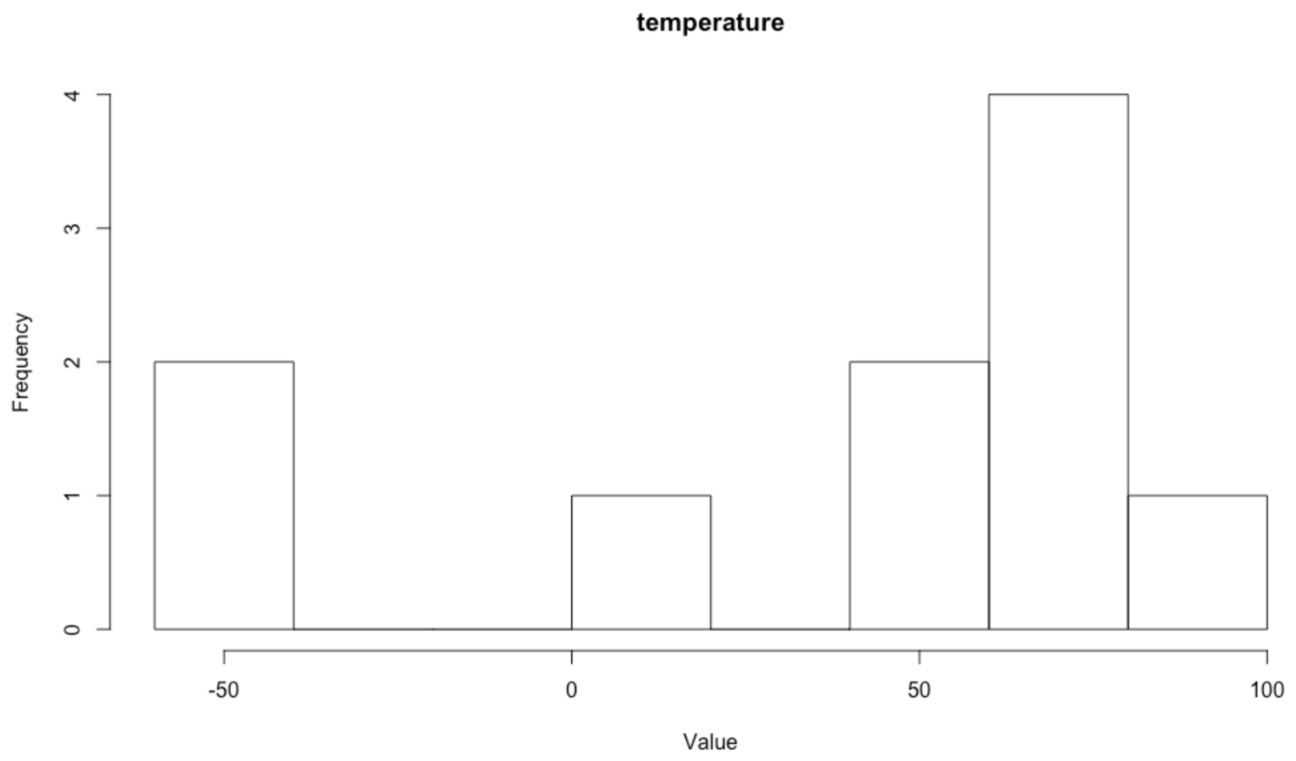
Maximum entropy distributions

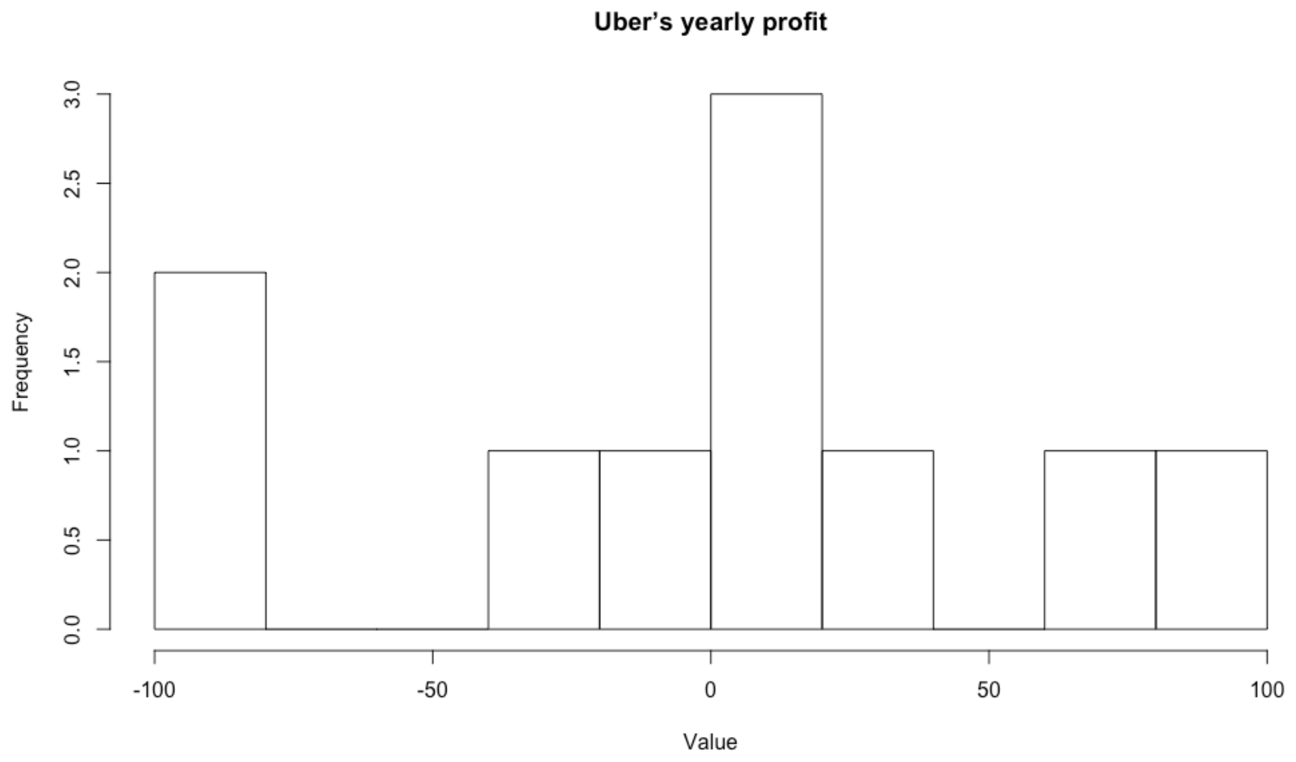
Consider another continuous-valued random variable: "Uber's yearly profit." Like `temperature`, it also has an underlying true mean *[Math Processing Error]* and variance *[Math Processing Error]*. Trivially, the respective means and variances will be different. Assume we observe 10 (fictional) values of each that look as follows:

-100	-50
-80	5
-20	56
5	65
15	62
-10	63
22	60
12	78
70	100
100	-43

Plotting, we get:

Error preparing HTML-CSS output (preProcess)





We are not given the true underlying probability distribution associated with each random variable — not its general "shape," nor the parameters that control this shape. We will *never* be given these things, in fact: the point of statistics is to infer what they are.

To make an initial choice we keep two things in mind:

- *We'd like to be conservative*. We've only seen 10 values of "Uber's yearly profit;" we don't want to discount the fact that the next 20 could fall into *[Math Processing Error]* just because they haven't yet been observed.
- *We need to choose the same probability distribution "shape" for both random variables, as we've made identical assumptions for each*.

As such, we'd like the most conservative distribution that obeys its constraints. This is the *maximum entropy distribution* (https://en.wikipedia.org/wiki/Maximum_entropy_probability_distribution).

Error preparing HTML-CSS output (preProcess)

For `temperature`, the maximum entropy distribution is the Gaussian distribution (https://en.wikipedia.org/wiki/Normal_distribution). Its probability density function is given as:

[Math Processing Error]
For `cat or dog`, it is the binomial distribution (https://en.wikipedia.org/wiki/Binomial_distribution). Its probability mass function (for a single observation) is given as:

[Math Processing Error]
(I've written the probability of the positive event as *[Math Processing Error]*, e.g. *[Math Processing Error]* for a fair coin.)

For `red or green or blue`, it is the multinomial distribution (https://en.wikipedia.org/wiki/Multinomial_distribution). Its probability mass function (for a single observation) is given as:

[Math Processing Error]
While it may seem like we've "waved our hands" over the connection between the stated equality constraints for the response variable of each model and the respective distributions we've selected, it is Lagrange multipliers (https://en.wikipedia.org/wiki/Lagrange_multiplier) that succinctly and algebraically bridge this gap. This post (https://www.dsprelated.com/freebooks/sasp/Maximum_Entropy_Property_Gaussian.html) gives a terrific example of this derivation. I've chosen to omit it as I did not feel it would contribute to the clarity nor direction of this post.

Finally, while we do assume that a Gaussian dictates the true distribution of values of both "Uber's yearly profit" and `temperature`, it is, trivially, a different Gaussian for each. This is because each random variable has its own true underlying mean and variance. These values make the respective Gaussians taller or wider — shifted left or shifted right.

Functional form

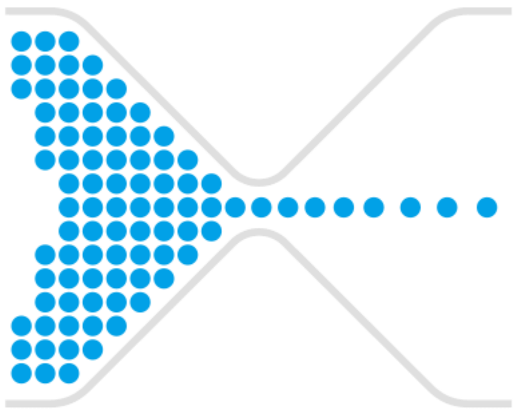
Our three protagonists generate predictions via distinct functions: the identity function (https://en.wikipedia.org/wiki/Identity_function), the sigmoid function (https://en.wikipedia.org/wiki/Sigmoid_function) and the softmax function (https://en.wikipedia.org/wiki/Softmax_function), respectively. The respective Keras output layers make this clear:

```
output = Dense(1)(input)
output = Dense(1, activation='sigmoid')(input)
output = Dense(3, activation='softmax')(input)
```

In this section, I'd like to:

- Show how each of the Gaussian, binomial and multinomial distributions can be reduced to the same functional form.
- Show how this functional form allows us to naturally derive the output functions for our three protagonist models.

Graphically, this looks as follows, with three distributions going in and three output functions coming out.



The conceptual bottleneck is the "exponential family" (https://en.wikipedia.org/wiki/Exponential_family) of probability distributions.

Error preparing HTML-CSS output (preProcess)

Exponential family distributions

> In probability and statistics, an exponential family is a set of probability distributions of a certain form, specified below. This special form is chosen for mathematical convenience, on account of some useful algebraic properties, as well as for generality, as exponential families are in a sense very natural sets of distributions to consider.

— Wikipedia

I don't relish quoting this paragraph — and especially one so deliriously general. This said, the reality is that exponential functions provide, at a minimum, a unifying framework for deriving the canonical activation and loss functions we've come to know and love. To move forward, we simply have to cede that the "mathematical conveniences, on account of some useful algebraic properties, etc." that motivate this "certain form" are not totally heinous nor misguided.

A distribution belongs to the exponential family if it can be written in the following form:

[Math Processing Error]
where:

- *[Math Processing Error]* is the *canonical parameter* of the distribution. (We will hereby work with the single-canonical-parameter exponential family form.)
- *[Math Processing Error]* is the *sufficient statistic*. It is often the case that *[Math Processing Error]*.
- *[Math Processing Error]* is the *log partition function*, which normalizes the distribution. (A more in-depth discussion of this normalizing constant can be found in a previous post of mine: Deriving the Softmax from First Principles (https://cavaunpeu.github.io/2017/04/19/deriving-the-softmax-from-first-principles/).)

"A fixed choice of *[Math Processing Error]*, *[Math Processing Error]* and *[Math Processing Error]* defines a family (or set) of distributions that is parameterized by *[Math Processing Error]*; as we vary *[Math Processing Error]*, we then get different distributions within this family."[1] This simply means that a coin with *[Math Processing Error]* gives a different distribution over outcomes than one with *[Math Processing Error]*. Easy.

Gaussian distribution

Since we're working with the single-parameter form, we'll assume that *[Math Processing Error]* is known and equals *[Math Processing Error]*. Therefore, *[Math Processing Error]* will equal *[Math Processing Error]*, i.e. the thing we pass into the Gaussian (that moves it left or right).

[Math Processing Error]
where:

- *[Math Processing Error]*
- *[Math Processing Error]*
- *[Math Processing Error]*
- *[Math Processing Error]*

Finally, we'll express *[Math Processing Error]* in terms of *[Math Processing Error]* itself:

[Math Processing Error]

Binomial distribution

We previously defined the binomial distribution (for a single observation) in a crude, peacewise form. We'll now define it in a more compact form which will make it easier to show that it is a member of the exponential family. Again, *[Math Processing Error]* gives the probability of observing the true class, i.e. *[Math Processing Error]*.

[Math Processing Error]
where:

- *[Math Processing Error]* Error preparing HTML-CSS output (preProcess)

- *[Math Processing Error]*
- *[Math Processing Error]*
- *[Math Processing Error]*

Finally, we'll express *[Math Processing Error]* in terms of *[Math Processing Error]*, i.e. the parameter that this distribution accepts:

[Math Processing Error]
[Math Processing Error]
You will recognize our expression for *[Math Processing Error]* as the sigmoid function.

Multinomial distribution

Like the binomial distribution, we'll first rewrite the multinomial (for single observation) in a more compact form. *[Math Processing Error]* gives a vector of class probabilities for the *[Math Processing Error]* classes; *[Math Processing Error]* denotes one of these classes.

[Math Processing Error]
This is almost pedantic: it says that *[Math Processing Error]* equals the probability of observing class *[Math Processing Error]*. For example, given

```
p = {'rain': .14, 'snow': .37, 'sleet': .03, 'hail': .46}
```

we would compute:

[Math Processing Error]
Expanding into the exponential family form gives:

[Math Processing Error]
where:

- *[Math Processing Error]*
- *[Math Processing Error]*
- *[Math Processing Error]*
- *[Math Processing Error]*

Finally, we'll express *[Math Processing Error]* in terms of *[Math Processing Error]*, i.e. the parameter that this distribution accepts:

[Math Processing Error]
Plugging back into the second line we get:

[Math Processing Error]
This you will recognize as the softmax function. (For a probabilistically-motivated derivation, please see a previous post (https://cavaunpeu.github.io/2017/04/19/deriving-the-softmax-from-first-principles/).)

Finally:

[Math Processing Error]
[Math Processing Error]

Generalized linear models

Each protagonist model outputs a response variable that is distributed according to some (exponential family) distribution. However, the *canonical parameter* of this distribution, i.e. the thing we pass in, will *vary per observation*.

Consider the logistic regression model that's predicting `cat or dog`. If we input a picture of a cat, we'll output "cat" according to the stated distribution.

Error preparing HTML-CSS output (preProcess)

[Math Processing Error]
If we input a picture of a dog, we'll output "dog" according the same distribution.

[Math Processing Error]
Trivially, the *[Math Processing Error]* value must be different in each case. In the former, *[Math Processing Error]* should be small, such that we output "cat" with probability *[Math Processing Error]*. In the latter, *[Math Processing Error]* should be large, such that we output "dog" with probability *[Math Processing Error]*.

So, what dictates the following? - *[Math Processing Error]* in the case of linear regression, in which *[Math Processing Error]* - *[Math Processing Error]* in the case of logistic regression, in which *[Math Processing Error]* - *[Math Processing Error]* in the case of softmax regression, in which *[Math Processing Error]*

Here, I've introduced the subscript *[Math Processing Error]*. This makes explicit the `cat or dog` dynamic from above: each input to a given model will result in its *own* canonical parameter being passed to the distribution on the response variable. That logistic regression better make *[Math Processing Error]* when looking at a picture of cat.

How do we go from a 10-feature input *[Math Processing Error]* to this canonical parameter? We take a linear combination:

[Math Processing Error]

Linear regression

[Math Processing Error]. This is what we need for the normal distribution.

Logistic regression

[Math Processing Error]. To solve for *[Math Processing Error]*, we solve for *[Math Processing Error]*.

As you'll remember we did this above: *[Math Processing Error]*.

It does this in the same way that our weather distribution dictates tomorrow's forecast.

```
p = {'rain': .14, 'snow': .37, 'sleet': .03, 'hail': .46}
```

Softmax regression

[Math Processing Error]. To solve for *[Math Processing Error]*, i.e. the full vector of probabilities for observation *[Math Processing Error]*, we solve for each individual probability *[Math Processing Error]* then put them in a list.

We did this above as well: *[Math Processing Error]*. This is the softmax function.

Finally, why a linear model, i.e. why *[Math Processing Error]*? Andrew Ng calls it a "design choice."[1] I've motivated this formulation a bit in the softmax post (https://cavaunpeu.github.io/2017/04/19/deriving-the-softmax-from-first-principles/). mathematicalmonk[2] would probably have a more principled explanation than us both. For now, we'll make do with the following: - A linear combination is perhaps the simplest way to consider the impact of each feature on the canonical parameter. - A linear combination commands that either *[Math Processing Error]*, or a *function of [Math Processing Error]*, vary linearly with *[Math Processing Error]*. As such, we could write our model as *[Math Processing Error]*, where *[Math Processing Error]* applies some complex transformation to our features. This makes the "simplicity" of the linear combination less simple.

Error preparing HTML-CSS output (preProcess)

Loss function

We've now discussed how each response variable is generated, and how we compute the parameters for those distributions on a per-observation basis. Now, how do we quantify how good these parameters are?

To get us started, let's go back to predicting `cat or dog`. If we input a picture of a cat, we should compute *[Math Processing Error]* given our binomial distribution.

[Math Processing Error]
A perfect computation gives *[Math Processing Error]*. The loss function quantifies how close we got.

Maximum likelihood estimation

Each of our three distributions receives a parameter — *[Math Processing Error]* and *[Math Processing Error]* respectively. We then pass in a *[Math Processing Error]* and the distribution tells us the probability of observing that value. (In the case of continuous-valued random variables, i.e. our distribution is a probability density function, it tells us a value *proportional* to this probability.)

If we instead *fix [Math Processing Error]* and pass in varying *parameter values*, our function becomes a *likelihood function*. It will tell us the likelihood of a given parameter having produced the now-fixed *[Math Processing Error]*.

If this is not clear, consider the following example:

> A Moroccan walks into a bar. He's wearing a football jersey that's missing a sleeve. He has a black eye, and blood on his jeans. How did he most likely spend his day?
> 1. At home, reading a book.
> 2. Training for a bicycle race.
> 3. At the soccer game drinking beers with his friends - all of whom are MMA fighters and despise the other team.

We'd like to pick the parameter that most likely gave rise to our data. This is the *maximum likelihood estimate*. Mathematically, we define it as:

[Math Processing Error]
As we've now seen (ad nauseum), *[Math Processing Error]* depends on the parameter its distribution receives. Additionally, this parameter — *[Math Processing Error]* or *[Math Processing Error]* — is defined in terms of *[Math Processing Error]*. Further, *[Math Processing Error]*. As such, *[Math Processing Error]* is a function of *[Math Processing Error]* and the observed data *[Math Processing Error]*. This is perhaps *the* elementary truism of machine learning — you've known this since Day 1.

Since our observed data are fixed, *[Math Processing Error]* is the only thing that we can vary. Let's rewrite our argmax in these terms:

[Math Processing Error]
Finally, this expression gives the argmax over a single data point, i.e. training observation, *[Math Processing Error]*. To give the likelihood over all observations (assuming they are independent of one another, i.e. the outcome of the first observation should not be expected to impact that of the third), we take the product.

[Math Processing Error]
The product of numbers in *[Math Processing Error]* gets very small, very quickly. Let's maximize the log likelihood instead so we can work with sums.

Linear regression

Maximize the log-likelihood of the Gaussian distribution. Remember, *[Math Processing Error]* and *[Math Processing Error]* assemble to give *[Math Processing Error]*, where *[Math Processing Error]*.

[Math Processing Error]
Maximizing the log-likelihood of our data with respect to *[Math Processing Error]* is equivalent to maximizing the negative mean squared error between the observed *[Math Processing Error]* and our prediction thereof.

Error preparing HTML-CSS output (preProcess)

Notwithstanding, most optimization routines *minimize*. So, for practical purposes, we go the other way.

Logistic regression

Same thing.

Negative log-likelihood:

[Math Processing Error]

Multinomial distribution

Negative log-likelihood:

[Math Processing Error]

Cross entropy (https://en.wikipedia.org/wiki/Cross_entropy)

We previously defined entropy as a way to quantify the uncertainty inherent in a probability distribution. Next, we'll use this same notion to quantify the uncertainty inherent in using the probabilities *in one distribution to predict events in another.*

```
p = {'red': .25, 'green': .45, 'blue':, .3}
q = {'red': .35, 'green': .4, 'blue':, .25}
```

[Math Processing Error]
This is the definition of cross entropy.

KL-Divergence (https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence)

Similarly, the Kullback-Leibler divergence quantifies the *additional uncertainty in [Math Processing Error]* introduced by using *[Math Processing Error]* to approximate *[Math Processing Error]*. It is given, almost trivially, as:

[Math Processing Error]
Why don't we use this in machine learning models instead of the cross entropy? The KL-divergence between distributions requires us to know the true, underlying probabilities of both the actual distribution *[Math Processing Error]* and our prediction thereof. Unfortunately, we never have the former: that's why we build the model.

Maximum a posteriori estimation

When estimating *[Math Processing Error]* via the MLE, we put no constraints on the permissible values thereof. More explicitly, we allow *[Math Processing Error]* to be *equally likely to assume any real number* — be it *[Math Processing Error]*, or *[Math Processing Error]*, or *[Math Processing Error]*, or *[Math Processing Error]*.

In practice, this assumption is both unrealistic and superfluous: typically, we do wish to constrain *[Math Processing Error]* (our weights) to a non-infinite range of values. We do this by putting a *prior* on *[Math Processing Error]*. Whereas the MLE computes *[Math Processing Error]*, the maximum a posteriori estimate, or MAP, computes *[Math Processing Error]*.

As before, we start by taking the log. Our joint likelihood with prior now reads:

[Math Processing Error]
Error preparing HTML-CSS output (preProcess)
We dealt with the left term in the previous section. Now, we'll simply tack on the log-prior to the respective log-likelihoods.

As every element of *[Math Processing Error]* is a continuous-valued real number, let's assign it a Gaussian distribution with mean 0 and variance *[Math Processing Error]*.

[Math Processing Error]
[Math Processing Error]
Our goal is to maximize this term plus the log likelihood — or equivalently, minimize their opposite — with respect to *[Math Processing Error]*. For a final step, let's discard the parts that don't include *[Math Processing Error]* itself.

[Math Processing Error]
This is L2 regularization. Furthermore, placing different prior distributions on *[Math Processing Error]* yields different regularization terms; most notably, a Laplace prior (https://en.wikipedia.org/wiki/Laplace_distribution) gives the L1.

Linear regression

[Math Processing Error]

Logistic regression

[Math Processing Error]

Softmax regression

[Math Processing Error]

Finally, in machine learning, we say that regularizing our weights ensures that "no weight becomes too large," i.e. too "influential" in predicting *[Math Processing Error]*. In statistical terms, we can equivalently say that this term *restricts the permissible values of these weights to a given interval. Furthermore, this interval is dictated by the scaling constant [Math Processing Error], which intrinsically parameterizes the prior distribution itself.* In L2 regularization, this scaling constant gives the variance of the Gaussian.

Going fully Bayesian

The key goal of a predictive model is to compute the following distribution:

[Math Processing Error]
By term, this reads:

- *[Math Processing Error]*: given historical data *[Math Processing Error]*, i.e. some training data, and a new observation *[Math Processing Error]*, compute the distribution of the possible values of the response *[Math Processing Error]*.
  - In machine learning, we typically select the *expected* value of that distribution, i.e. a single value, or point estimate.
- *[Math Processing Error]*: given historical data *[Math Processing Error]*, a new observation *[Math Processing Error]* and *any plausible value of [Math Processing Error]*, i.e. perhaps not the optimal value, compute *[Math Processing Error]*.
  - This is given by the functional form of the model in question, i.e. *[Math Processing Error]* in the case of linear regression.
- *[Math Processing Error]*: given historical data *[Math Processing Error]* and a new observation *[Math Processing Error]*, compute the distribution of the values of *[Math Processing Error]* that plausibly gave rise to our data.
  - The *[Math Processing Error]* plays no part; it's simply there such that the expression under the integral factors correctly.
  - In machine learning, we typically select the MLE or MAP estimate of that distribution, i.e. a single value, or point estimate.

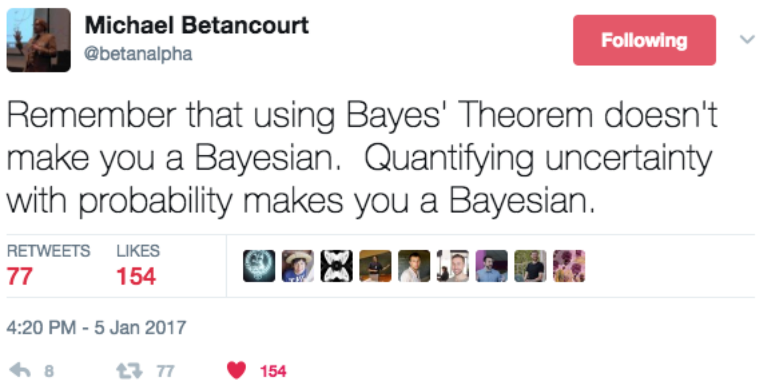
Error preparing HTML-CSS output (preProcess)

In a perfect world, we'd do the following:

- Compute the *full distribution* over *[Math Processing Error]*.
- With each value in this distribution and a new observation *[Math Processing Error]*, compute *[Math Processing Error]*.
  - NB: *[Math Processing Error]* is an object which contains all of our weights. In 10-feature linear regression, it will have 10 elements. In a neural network, it could have millions.
- We now have a *full distribution* over the possible values of the response *[Math Processing Error]*.

.

Unfortunately, in complex systems with a non-trivial functional form and number of weights, this computation becomes intractably large. As such, in fully Bayesian modeling, we approximate these distributions. In classic machine learning, we assign them a single value (point estimate). It's a bit lazy, really.


Michael Betancourt
@betanalpha
Following

Remember that using Bayes' Theorem doesn't make you a Bayesian. Quantifying uncertainty with probability makes you a Bayesian.

RETWEETS 77 LIKES 154

4:20 PM - 5 Jan 2017

8 77 154

Summary

I hope this post serves as useful context for the machine learning models we know and love. A deeper understanding of these algorithms offers humility — the knowledge that none of these concepts are particularly new — as well as a vision for how to extend these algorithms in the direction of robustness and increased expressivity.

Thanks so much for reading this far. Now, climb out of the pool, grab a towel and `import sklearn`.

Error preparing HTML-CSS output (preProcess)

Resources

I recently gave a talk on this topic at Facebook Developer Circle: Casablanca (https://www.facebook.com/groups/265793323822652). Voilà the:

- Slides (https://www.slideshare.net/WilliamWolfDataScien/youve-been-doing-statistics-all-along)
- Video (https://www.facebook.com/aboullaite.mohammed/videos/1959648697600819/)

---

1. CS229 Machine Learning Course Materials, Lecture Notes 1 (http://cs229.stanford.edu/materials.html) ↩↩
2. mathematical monk - Machine Learning (https://www.youtube.com/playlist?list=PLD0F06AA0D2E8FFBA) ↩

Error preparing HTML-CSS output (preProcess)