COM4519 DATA MINING
Logistic regression

Lecturer: Begüm MUTLU BİLGE, PhD

begummutlubilge+com4519@gmail.com (recommended)
bmbilge@ankara.edu.tr

Logistic regression

Discovering the link between features or cues and some particular outcome: logistic regression.

Close relationship with neural networks

Logistic regression can be used to classify an observation into one of two classes, or into one of many classes.

Generative and Discriminative Classifiers

Generative and Discriminative Classifiers

A generative model makes use of likelihood term, which expresses how to generate the features of a document if we know it was of class c.

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} \overbrace{P(d|c)}^{\text{likelihood}} \overbrace{P(c)}^{\text{prior}}$$

A discriminative model in the text categorization scenario attempts model to directly compute **P(c|d)**.

- It will learn to assign high weight to document features that directly improve its ability to discriminate between possible classes
  - Even if it couldn't generate an example of one of the classes.

Generative and Discriminative Classifiers

Discriminative (conditional) models are widely used in NLP.

- They give high accuracy performance
- They make it easy to incorporate lots of linguistically important features
- They allow automatic building of language independent NLP modules

In text classification task, we have some data {(d, c)} of paired observations d and hidden classes c.

Generative and Discriminative Classifiers

Generative (joint) models place probabilities over **both observed data and the hidden stuff** (generate the observed data from hidden stuff).

- some generative models: n-gram models, Naive Bayes classifiers, hidden Markov models, probabilistic context-free grammars, …

Discriminative (conditional) models **take the data as given, and put a probability over hidden structure given the data**.

- some discriminative models: logistic regression, maximum entropy models, conditional random fields, SVMs, perceptron, …

Components of a probabilistic machine learning classifier

A machine learning system for classification then has four components:

- A feature representation of the input.
- A classification function that computes $\hat{y}$, the estimated class, via $p(y|x)$
- An objective function for learning
- An algorithm for optimizing the objective function.

Components of a probabilistic machine learning classifier

Logistic regression has two phases:

train: we train the system (specifically the weights w and b) using stochastic gradient descent and the cross-entropy loss.

test: Given a test example x we compute p(y|x) and return the higher probability label y = 1 or y = 0.

Classification: the sigmoid

The goal: to train a classifier that can make a **binary decision** about the class of a new input observation.

Sigmoid (logistic function)

$X = [x1 , x2 , ..., xn]$

$Y = 1/0$

We want to know the probability $P(y = 1|x)$

- that this observation is a member of the class.

Training

We train the system to learn a vector of weights for features and a bias term using stochastic gradient descent and the cross-entropy loss.

- Each weight w_i is a real number, and is associated with one of the input features x_i.
- The weight w_i : how important that input feature is to the classification decision

The bias term (intercept) is another real number that's added to the weighted inputs.

Testing

For given a test example x, we compute P(y|x) and return the higher probability label y=1 (member of class) or y=0 (not member of class).

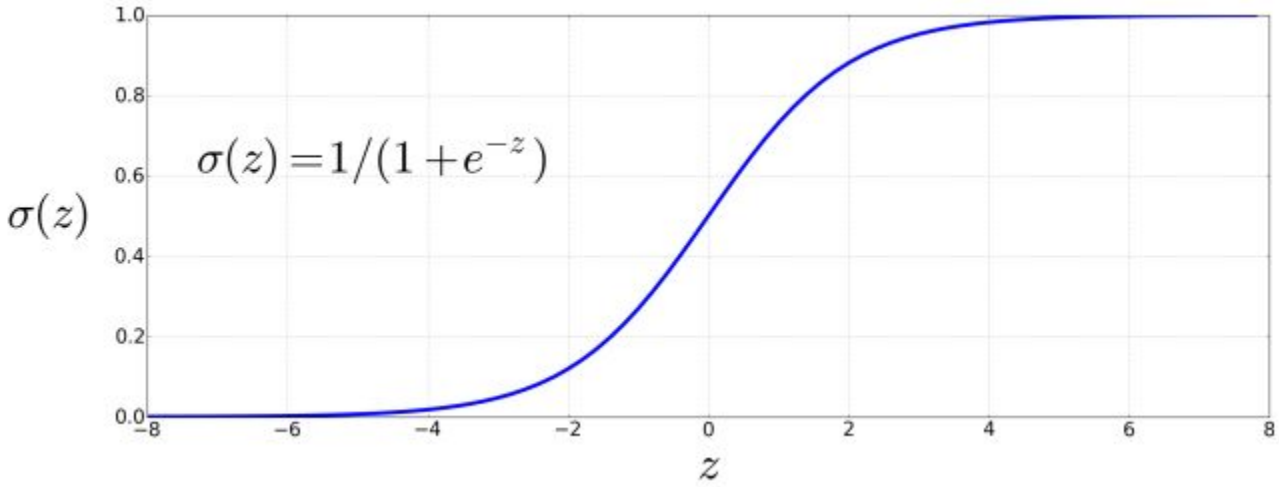
$$z = \left(\sum_{i=1}^{n} w_i x_i\right) + b$$

$$z = w \cdot x + b$$

$$-\infty < z < +\infty$$

To create a probability, we'll pass z through the sigmoid function, σ (z).

Sigmoid function



The sigmoid function σ(z) takes a real value and maps it to the range [0, 1].

It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

Sigmoid function

The sigmoid has a number of advantages;

- It maps a real-valued number into range [0,1].
- It tends to squash outlier values toward 0 or 1.
- It is **differentiable**.
  - This will be handy for learning.

Sigmoid function

If we apply the sigmoid to the sum of the weighted features, we get a number between 0 and 1.

To make it a probability, we just need to make sure that the two cases, p(y = 1) and p(y = 0), **sum to 1**. We can do this as follows:

$$
\begin{aligned}
P(y=1) &= \sigma(w \cdot x + b) \\
&= \frac{1}{1+\exp\left(-(w \cdot x + b)\right)} \\
P(y=0) &= 1 - \sigma(w \cdot x + b) \\
&= 1 - \frac{1}{1+\exp\left(-(w \cdot x + b)\right)} \\
&= \frac{\exp\left(-(w \cdot x + b)\right)}{1+\exp\left(-(w \cdot x + b)\right)}
\end{aligned}
$$

Decision

The sigmoid function has the property

$1 - \sigma(x) = \sigma(-x)$

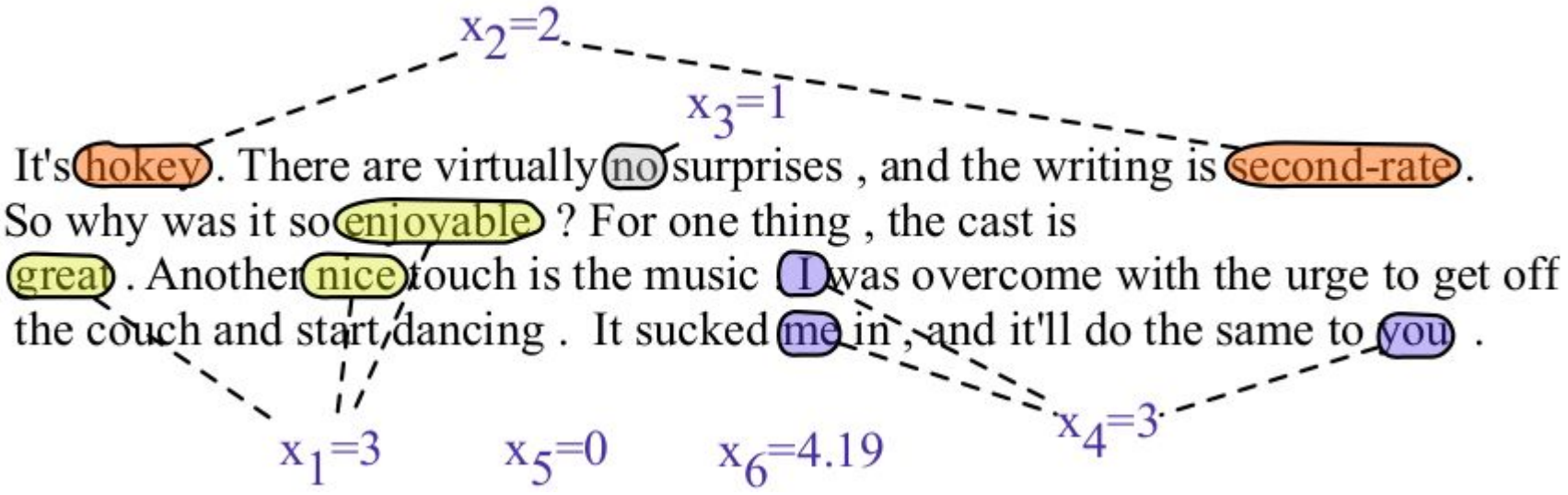
so we could also have expressed

$P(y = 0)$ as $\sigma(-(w \cdot x + b))$

How do we make a decision?

$$\text{decision}(x) = \begin{cases} 1 & \text{if } P(y = 1|x) > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

Example: sentiment classification



Var	Definition	Value in Fig.
x_1	count(positive lexicon words $\in$ doc)	3
x_2	count(negative lexicon words $\in$ doc)	2
x_3	$\begin{cases} 1 & \text{if "no"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	1
x_4	count(1st and 2nd pronouns $\in$ doc)	3
x_5	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	0
x_6	log(word count of doc)	$\ln(66) = 4.19$

Example: sentiment classification

Let's assume we've **already learned** a real-valued **weight for each of these features**,

and that the **6 weights corresponding to the 6 features are [2.5, −5.0, −1.2, 0.5, 2.0, 0.7]**,

while **b = 0.1**.

Given these 6 features and the input review x, P(+|x) and P(−|x) can be computed using

$$
\begin{aligned}
p(+|x) = P(y=1|x) &= \sigma(w \cdot x + b) \\
&= \sigma([2.5, -5.0, -1.2, 0.5, 2.0, 0.7] \cdot [3, 2, 1, 3, 0, 4.19] + 0.1) \\
&= \sigma(.833) \\
&= 0.70 \\
p(-|x) = P(y=0|x) &= 1 - \sigma(w \cdot x + b) \\
&= 0.30
\end{aligned}
$$

Learning in Logistic Regression

How are the parameters of the model, the weights **w** and bias **b**, **learned**?

Logistic regression is an instance of supervised classification in which

- We know the correct label y (either 0 or 1) for each observation x.

What the system produces is **ŷ**, the system's estimate of the **true y**.

We want to learn parameters (meaning w and b) that make ŷ for each training observation as close as possible to the true y.

Learning in Logistic Regression

Two components of learning:

- A metric for how close the current label ($\hat{y}$) is to the true gold label y: **loss function**
- An optimization **algorithm** for iteratively updating the weights so as to minimize this loss function

The cross-entropy loss function

We need a loss function that expresses, for an observation x, how close the classifier output ($\hat{y} = \sigma(w \cdot x + b)$) is to the correct output (y, which is 0 or 1).

We'll call this:

$L(\hat{y}, y)$ = How much $\hat{y}$ differs from the true y

Conditional maximum likelihood estimation

Negative log likelihood loss, generally called the cross-entropy loss.

The cross-entropy loss function

Let's derive this loss function, applied to a single observation x.

We'd like to learn weights that maximize the probability of the correct label p(y|x).

Since there are only two discrete outcomes (1 or 0) and we can express the probability p(y|x) that our classifier produces for one observation as the following:

$$p(y|x) = \hat{y}^y (1-\hat{y})^{1-y}$$

The cross-entropy loss function

Now we take the log of both sides. whatever values maximize a probability will also maximize the log of the probability

$$\begin{aligned}\log p(y|x) &= \log\left[\hat{y}^y(1-\hat{y})^{1-y}\right] \\ &= y\log\hat{y} + (1-y)\log(1-\hat{y})\end{aligned}$$

In order to turn this into loss function (something that we need to minimize), we'll just flip the sign on, and the result is the cross-entropy loss L_{CE}:

$$L_{\text{CE}}(\hat{y}, y) = -\log p(y|x) = -\left[y\log\hat{y} + (1-y)\log(1-\hat{y})\right]$$

The cross-entropy loss function

Finally we can plug in the definition of

- $\hat{y} = \sigma(w \cdot x + b)$

$$L_{CE}(\hat{y}, y) = -[y \log \sigma(w \cdot x + b) + (1 - y) \log (1 - \sigma(w \cdot x + b))]$$

We want the loss to be smaller if the model's estimate is close to correct, and bigger if the model is confused.

The binary cross-entropy loss function by Python

```python
from math import log
from numpy import mean

# calculate cross entropy
def cross_entropy(exp, pred):

    # -( 0.0 * log(0.9) + 1.0 * log(0.1) )
    return -sum([exp[j]*log(pred[j]) for j in range(len(exp))])


# define classification data
p = [1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
q = [0.1, 0.9, 0.9, 0.6, 0.8, 0.1, 0.4, 0.2, 0.1, 0.3, 0.1]
# calculate cross entropy for each example
results = list()
for i in range(len(p)):
    # create the distribution for each event {0, 1}
    expected = [1.0 - p[i], p[i]]   #for i = 0; [0.0, 1.0]
    predicted = [1.0 - q[i], q[i]]  #for i = 0; [0.9, 0.1]
    # calculate cross entropy for the two events
    ce = cross_entropy(expected, predicted)
    print('>[y=%.1f, yhat=%.1f] ce: %.3f' % (p[i], q[i], ce))
    results.append(ce)

# calculate the average cross entropy
mean_ce = mean(results)
print('Average Cross Entropy: %.3f' % mean_ce)
```

```
>[y=1.0, yhat=0.1] ce: 2.303
>[y=1.0, yhat=0.9] ce: 0.105
>[y=1.0, yhat=0.9] ce: 0.105
>[y=1.0, yhat=0.6] ce: 0.511
>[y=1.0, yhat=0.8] ce: 0.223
>[y=0.0, yhat=0.1] ce: 0.105
>[y=0.0, yhat=0.4] ce: 0.511
>[y=0.0, yhat=0.2] ce: 0.223
>[y=0.0, yhat=0.1] ce: 0.105
>[y=0.0, yhat=0.3] ce: 0.357
>[y=0.0, yhat=0.1] ce: 0.105
Average Cross Entropy: 0.423
```

```python
# calculate cross entropy with keras
from numpy import asarray
from keras import backend
from keras.losses import binary_crossentropy
# prepare classification data
p = asarray([1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
q = asarray([0.1, 0.9, 0.9, 0.6, 0.8, 0.1, 0.4, 0.2, 0.1, 0.3, 0.1])
# convert to keras variables
y_true = backend.variable(p)
y_pred = backend.variable(q)
# calculate the average cross-entropy
mean_ce = backend.eval(binary_crossentropy(y_true, y_pred))
print('Average Cross Entropy: %.3f' % mean_ce)
```

```
Average Cross Entropy: 0.423
```

$$L = -\frac{1}{N}\left[\sum_{j=1}^{N}\left[t_j \log(p_j) + (1-t_j)\log(1-p_j)\right]\right]$$

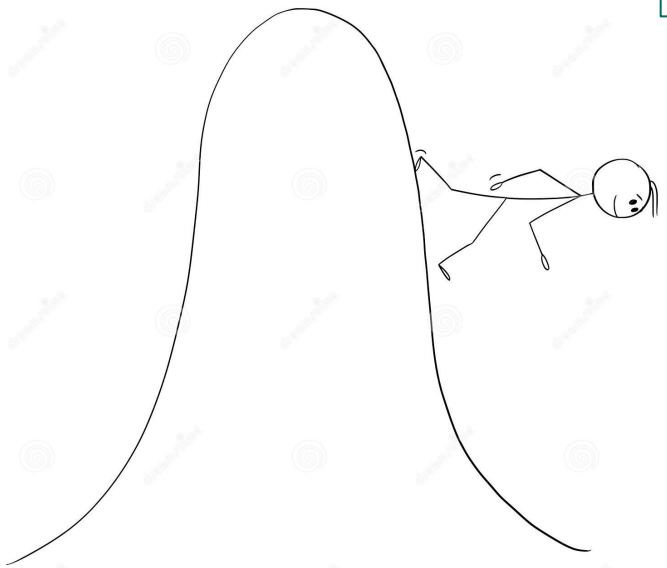
for N data points where t_i is the truth value taking a value 0 or 1 and p_i is the Softmax probability for the i^{th} data point.

Gradient Descent

Our goal with gradient descent is to find the optimal weights: minimize the loss function we've defined for the model.

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} L_{\text{CE}}(f(x^{(i)}; \theta), y^{(i)})$$

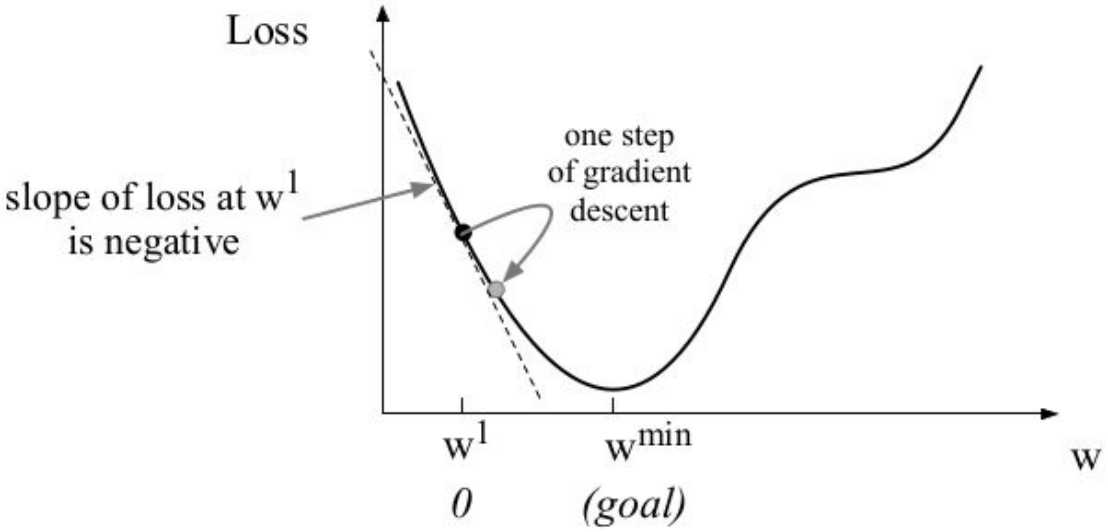
weights



Gradient Descent

How shall we find the minimum of this (or any) loss function?

- Gradient descent



The first step in iteratively finding the minimum of this loss function, by moving w in the reverse direction from the slope of the function. Since the slope is negative, we need to move w in a positive direction, to the right.

Learning rate

The magnitude of the amount to move in gradient descent is the value of the slope

$$\frac{d}{dw} L(f(x;w), y)$$

weighted by a learning rate η

The final equation for updating θ based on the gradient is

$$\theta_{t+1} = \theta_t - \eta \nabla L(f(x;\theta), y)$$

Gradient Descent

In order to update θ , we need a definition for the gradient $\nabla L(f(x;\theta),y)$.

Recall that for logistic regression, the cross-entropy loss function is:

$$L_{\text{CE}}(\hat{y},y) = -[y\log\sigma(w\cdot x+b)+(1-y)\log(1-\sigma(w\cdot x+b))]$$

It turns out that the derivative of this function for one observation vector x:

$$\frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial w_j} = [\sigma(w\cdot x+b)-y]x_j$$

Stochastic gradient descent

An **online** algorithm that minimizes the loss function by

- Computing its gradient after each training example, and
- Nudging θ in the right direction
  - (the opposite direction of the gradient).

Stochastic gradient descent

function STOCHASTIC GRADIENT DESCENT($L()$, $f()$, x, y) **returns** θ

where: L is the loss function
f is a function parameterized by θ
x is the set of training inputs $x^{(1)}$, $x^{(2)}$, ..., $x^{(m)}$
y is the set of training outputs (labels) $y^{(1)}$, $y^{(2)}$, ..., $y^{(m)}$

$\theta \leftarrow 0$

repeat til done *

For each training tuple $(x^{(i)}, y^{(i)})$ (in random order)

1. Optional (for reporting): # How are we doing on this tuple?
   Compute $\hat{y}^{(i)} = f(x^{(i)}; \theta)$ # What is our estimated output $\hat{y}$?
   Compute the loss $L(\hat{y}^{(i)}, y^{(i)})$ # How far off is $\hat{y}^{(i)}$ from the true output $y^{(i)}$?
2. $g \leftarrow \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$ # How should we move θ to maximize loss?
3. $\theta \leftarrow \theta - \eta\, g$ # Go the other way instead

return θ

The stochastic gradient descent algorithm.

*The algorithm can terminate when it converges (or when the gradient norm <), or when progress halts (for example when the loss starts going up on a held-out set).

Stochastic gradient descent

function STOCHASTIC GRADIENT DESCENT($L()$, $f()$, x, y) **returns** θ

where: L is the loss function
f is a function parameterized by θ
x is the set of training inputs $x^{(1)}$, $x^{(2)}$, ..., $x^{(m)}$
y is the set of training outputs (labels) $y^{(1)}$, $y^{(2)}$, ..., $y^{(m)}$

$\theta \leftarrow 0$

repeat til done *

For each training tuple $(x^{(i)}, y^{(i)})$ (in random order)

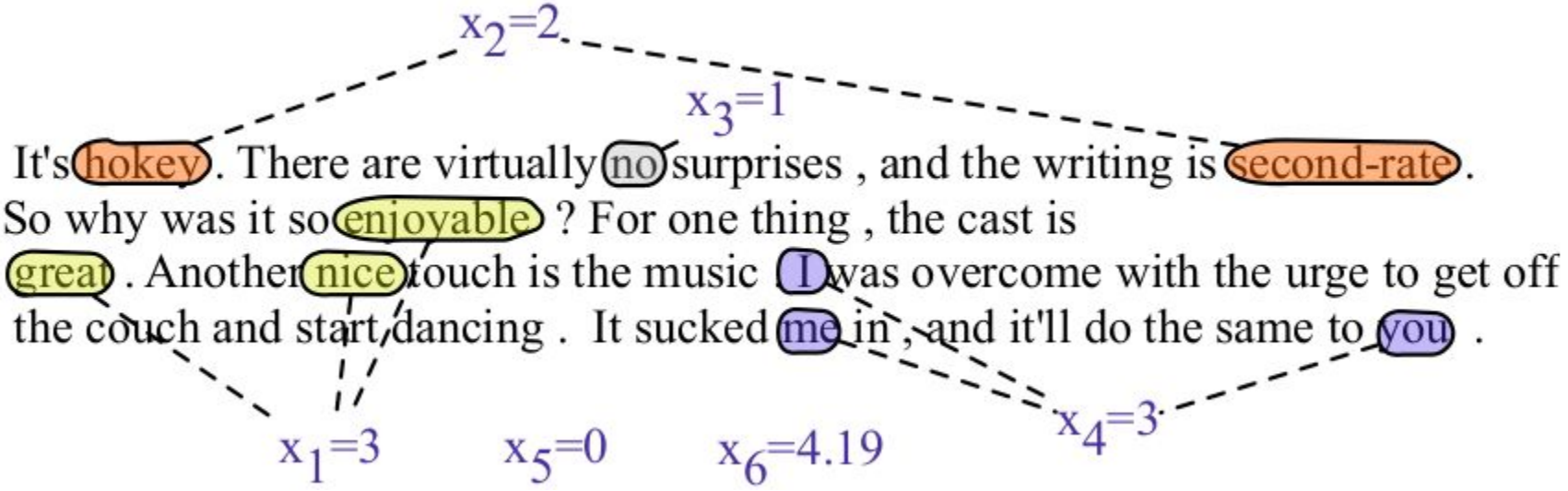
1. Optional (for reporting):
   Compute $\hat{y}^{(i)} = f(x^{(i)}; \theta)$ # What is our estimated output $\hat{y}$?
   Compute the loss $L(\hat{y}^{(i)}, y^{(i)})$ # How far off is $\hat{y}^{(i)}$ from the true output $y^{(i)}$?
2. $g \leftarrow \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$ # How should we move θ to maximize loss?
3. $\theta \leftarrow \theta - \eta\, g$ # Go the other way instead

return θ

The stochastic gradient descent algorithm.

*The algorithm can terminate when it converges (or when the gradient norm <), or when progress halts (for example when the loss starts going up on a held-out set).

Working through an example



We'll use a simplified version of the example in Figure as it sees a single observation x, whose correct value is **y = 1** (this is a positive review), and with only two features:

x1 = 3 (count of positive lexicon words)

x2 = 2 (count of negative lexicon words)

Working through an example

w1 = w2 = b = 0

η = 0.1

The single update step requires that we compute the gradient, multiplied by the learning rate

$$\theta^{t+1} = \theta^t - \eta \nabla_\theta L(f(x^{(i)};\theta), y^{(i)})$$

I here are three parameters, so the gradient vector has 3 dimensions, for w1 , w2 , and b. We can compute the first gradient as follows:

$$\begin{bmatrix} \frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial w_1} \\ \frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial w_2} \\ \frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial b} \end{bmatrix} = \begin{bmatrix} (\sigma(w \cdot x + b) - y)x_1 \\ (\sigma(w \cdot x + b) - y)x_2 \\ \sigma(w \cdot x + b) - y \end{bmatrix} = \begin{bmatrix} (\sigma(0) - 1)x_1 \\ (\sigma(0) - 1)x_2 \\ \sigma(0) - 1 \end{bmatrix} = \begin{bmatrix} -0.5x_1 \\ -0.5x_2 \\ -0.5 \end{bmatrix} = \begin{bmatrix} -1.5 \\ -1.0 \\ -0.5 \end{bmatrix}$$

Working through an example

Now that we have a gradient, we compute the new parameter vector θ 1 by moving θ 0 in the opposite direction from the gradient:

$$\theta^1 = \begin{bmatrix} w_1 \\ w_2 \\ b \end{bmatrix} - \eta \begin{bmatrix} -1.5 \\ -1.0 \\ -0.5 \end{bmatrix} = \begin{bmatrix} .15 \\ .1 \\ .05 \end{bmatrix}$$

So after one step of gradient descent, the weights have shifted to be:

w1 = .15, w2 = .1, and b = .05.

SGD to mini-batch training

Stochastic Gradient Descent (SGD) is called stochastic because

- It chooses a **single random example** at a time,
- Moving the weights so as to improve performance on that single example.

That can result in very choppy movements,

So it's common to compute the gradient over **batches** of **training instances** rather than a single instance.

Batch training

In batch training we compute the gradient over the **entire dataset**.

Batch training offers a superb estimate of which direction to move the weights

- By seeing so many examples,
- At the cost of spending a lot of time processing every single example in the training set to compute this perfect direction.

Mini-batch training

We train on a group of **m** examples (perhaps 512, or 1024) that is less than the whole dataset.

- If m is the size of the dataset, doing **batch gradient descent**;
- if m = 1, doing **stochastic gradient descent**.

Mini-batch training also has the **advantage** of computational **efficiency**.

- The mini-batches can easily be vectorized, choosing the size of the minibatch based on the computational resources.

This allows us to process all the examples in one mini-batch in parallel and then accumulate the loss

- Not possible with individual or batch training.

Mini-batch training

Mini-batch versions of the cross-entropy loss function

$$\begin{aligned}\log p(\text{training labels}) &= \log \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}) \\ &= \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}) \\ &= -\sum_{i=1}^{m} L_{\text{CE}}(\hat{y}^{(i)}, y^{(i)})\end{aligned}$$

$$\begin{aligned}Cost(\hat{y}, y) &= \frac{1}{m}\sum_{i=1}^{m} L_{\text{CE}}(\hat{y}^{(i)}, y^{(i)}) \\ &= -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log \sigma(w \cdot x^{(i)} + b) + (1 - y^{(i)}) \log \left(1 - \sigma(w \cdot x^{(i)} + b)\right)\end{aligned}$$

Mini-batch training

The mini-batch gradient is the average of the individual gradients

$$\frac{\partial Cost(\hat{y},y)}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}\left[\sigma(w\cdot x^{(i)}+b)-y^{(i)}\right]x_j^{(i)}$$

Overfitting

A **good model** should be able to **generalize well** from the training data to the unseen test set, but a **model that overfits** will have **poor generalization**

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} L_{\mathrm{CE}}(f(x^{(i)}; \theta), y^{(i)})$$

The new regularization term R(θ) is used to penalize large weights.

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) - \alpha R(\theta)$$

Regularization

There are two common ways to compute this regularization term R(θ):

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) - \boxed{\alpha R(\theta)}$$

L2:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \left[\sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) \right] - \alpha \sum_{j=1}^{n} \theta_j^2$$

L1:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \left[\sum_{1=i}^{m} \log P(y^{(i)}|x^{(i)}) \right] - \alpha \sum_{j=1}^{n} |\theta_j|$$

Multinomial logistic regression

Also called **softmax regression**

In multinomial logistic regression the target **y** is a variable that ranges over **more than two classes**;

- We want to know the probability of y being in each potential class $c \in C$, $p(y = c|x)$.

Uses a generalization of the sigmoid, called the softmax function

The softmax function takes a vector $z = [z1, z2, ...,zk]$ of k arbitrary values and maps them to a probability distribution, with each value in the range (0,1), and all the values summing to 1.

Multinomial logistic regression

For a vector z of dimensionality k, the softmax is defined as

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{k} \exp(z_j)} \quad 1 \leq i \leq k$$

The softmax of an input vector z = [z1,z2,...,zk] is thus a vector itself:

$$\text{softmax}(z) = \left[\frac{\exp(z_1)}{\sum_{i=1}^{k} \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{k} \exp(z_i)}, \ldots, \frac{\exp(z_k)}{\sum_{i=1}^{k} \exp(z_i)} \right]$$

Multinomial logistic regression

The denominator is used to normalize all the values into probabilities.

Thus for example given a vector:

$z = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$

the resulting (rounded) softmax(z) is

[0.055, 0.090, 0.006, 0.099, 0.74, 0.010]

Again like the sigmoid, the input to the softmax will be the dot product between a weight vector w and an input vector x (plus a bias).

Multinomial logistic regression

But now we'll need separate weight vectors (and bias) for each of the K classes.

$$p(y=c|x) = \frac{\exp(w_c \cdot x + b_c)}{\sum_{j=1}^{K} \exp(w_j \cdot x + b_j)}$$

Like the sigmoid, the softmax has the property of squashing values toward 0 or 1.

Thus if one of the inputs is larger than the others, it will tend to push its probability toward 1, and suppress the probabilities of the smaller inputs.

Logistic Regression by Python

```
from sklearn.linear_model import LogisticRegression
lreg = LogisticRegression(random_state=0,verbose=1, solver='lbfgs',penalty='l2', max_iter=500)
lreg.fit(X_train, y_train)
y_pred= lreg.predict(X_test)
y_pred_proba= lreg.predict_proba(X_test)
print("Number of mislabeled points out of a total %d points : %d"
      % (X_test.shape[0], (y_test != y_pred).sum()))
print("Score: " + str(lreg.score(X_test, y_test)))
#Return the mean accuracy on the given test data and labels.
```

```
Number of mislabeled points out of a total 114 points : 5
Score: 0.956140350877193
```

COM4519 DATA MINING
Logistic regression

Lecturer: Begüm MUTLU BİLGE, PhD

begummutlubilge+com4519@gmail.com (recommended)
bmbilge@ankara.edu.tr