A gentle introduction to maximum entropy, log-linear, exponential, logistic, harmonic, Boltzmann, Markov Random Fields, Conditional Random Fields, etc., models

Mark Johnson

Department of Computing

March, 2013 (updated August 2015)

A gentle introduction to maximum entropy, log-linear, exponential, logistic, harmonic, Boltzmann, Markov Random Field, etc., models

- How can we possibly cover so many kinds of models in a single talk?
- *Because they are all basically the same*
- If an idea is really (really!) good, you can justify it in many different ways!

Outline

Introducing exponential models

Features in exponential models

Learning exponential models

Regularisation

Conditional models

Stochastic gradient descent and error-driven learning

Avoiding the partition function and its derivatives

Weakly labelled training data

Summary

Why probabilistic models?

- Problem setup: given a set $\mathcal{X}$ of possible items
  - e.g., $\mathcal{X}$ is the set of all possible English words (sequences of characters)
  - e.g., $\mathcal{X}$ is the set of all possible sentences (sequences of English words)
  - e.g., $\mathcal{X}$ is the set of all possible images (256×256 pixel arrays)
- Our goal is to learn a probability distribution $\mathsf{P}(X)$ over $\mathcal{X}$
  - $\mathsf{P}(X)$ identifies which items $x \in \mathcal{X}$ are more likely and which ones are less likely
  - e.g., if $\mathcal{X}$ is the set of possible English sentences, $\mathsf{P}(X)$ is called a *language model*
  - language models are very useful in machine translation and speech recognition because they identify plausible sentences (e.g., "recognise speech" vs. "wreck a nice beach")
- In this talk, we are interested in learning $\mathsf{P}(X)$ from data $D = (x_1, \ldots, x_n)$, which is sampled from the (unknown) $\mathsf{P}(X)$

Motivating exponential models

- **Goal:** define a probability distribution $\mathrm{P}(X)$ over the $x \in \mathcal{X}$
- Idea: describe x in terms of *weighted features*
- Let $\mathcal{S}(x) \subseteq \mathcal{S}$ be the *set of x's features*
- Let v_s be the *weight of feature $s \in \mathcal{S}$*
  - if $v_s > 1$ then s makes x more probable
  - if $v_s < 1$ then s makes x less probable
- If $\mathcal{S}(x) = \{s_1, \ldots, s_n\}$, then

$$\begin{aligned} \mathrm{P}(X{=}x) &\propto v_{s_1} \ldots v_{s_n} \\ &= \prod_{s \in \mathcal{S}(x)} v_s \end{aligned}$$

- Generalises many well-known models (e.g., HMMs, PCFGs)
  - what are the features and the feature weights in an HMM or a PCFG?

In generative models defined as a product of conditional distributions as factors, factors cannot be greater than 1

The partition function

- Probability distributions must sum to 1, i.e.,

$$\sum_{x \in \mathcal{X}} \mathsf{P}(X{=}x) \quad = \quad 1$$

- But in general

$$\sum_{x \in \mathcal{X}} \left(\prod_{s \in \mathcal{S}(x)} v_s \right) \quad \neq \quad 1$$

$\Rightarrow$ *Normalise* the weighted feature products

$$Z \quad = \quad \sum_{x \in \mathcal{X}} \left(\prod_{s \in \mathcal{S}(x)} v_s \right)$$

Z is called the *partition function*

- Then define:

$$\mathsf{P}(X{=}x) \quad = \quad \frac{1}{Z} \prod_{s \in \mathcal{S}(x)} v_s$$

Q: Why is Z called a partition *function*? What is it a function of?

Feature functions

- Functions are often notationally easier to deal with than sets
- For each feature $s \in \mathcal{S}$ define a *feature function* $f_s : \mathcal{X} \mapsto 2$

$$f_s(x) \quad = \quad 1 \text{ if } s \in \mathcal{S}(x), \text{ and } 0 \text{ otherwise}$$

- Then we can rewrite

$$\begin{aligned} \mathrm{P}(X{=}x) &= \frac{1}{Z} \prod_{s \in \mathcal{S}(x)} v_s \\ &= \frac{1}{Z} \prod_{s \in \mathcal{S}} {v_s}^{f_s(x)} \end{aligned}$$

- Now we can have *real-valued feature functions*
- From here on assume we have a *vector of m feature functions*

$$\begin{aligned} \boldsymbol{f} &= (f_1, \ldots, f_m), \text{ and} \\ \boldsymbol{f}(x) &= (f_1(x), \ldots, f_m(x)) \end{aligned}$$

Exponential form

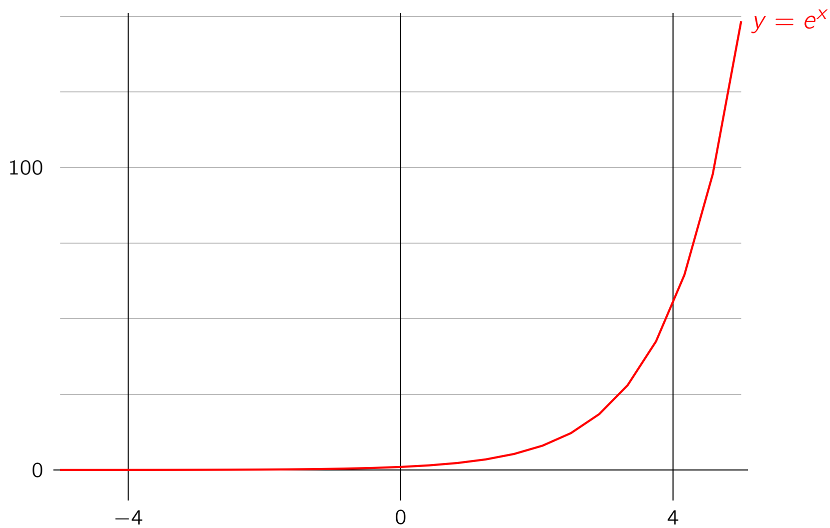
- The feature weights v_j must be non-negative because probabilities are non-negative
- An easy way to ensure that feature weights are positive is to work in log space. Let $w_j = \log(v_j)$ or equivalently $v_j = \exp(w_j)$.
  - If $w_j > 0$ then having feature j makes x more probable
  - If $w_j < 0$ then having feature j makes x less probable

$$
\begin{aligned}
\mathrm{P}(X{=}x) &= \frac{1}{Z}\prod_{j=1}^{m} {v_j}^{f_j(x)} \\
&= \frac{1}{Z}\exp\left(\sum_{j=1}^{m} w_j f_j(x)\right) \\
&= \frac{1}{Z}\exp\left(\boldsymbol{w}\cdot\boldsymbol{f}(x)\right)
\end{aligned}
$$

where:

$$
\begin{aligned}
\boldsymbol{w} &= (w_1, \ldots, w_m) \\
\boldsymbol{f}(x) &= (f_1(x), \ldots, f_m(x)) \\
Z &= \sum_{x' \in \mathcal{X}} \exp\left(\boldsymbol{w}\cdot\boldsymbol{f}(x')\right)
\end{aligned}
$$

The exponential function



Outline

Introducing exponential models

Features in exponential models

Learning exponential models

Regularisation

Conditional models

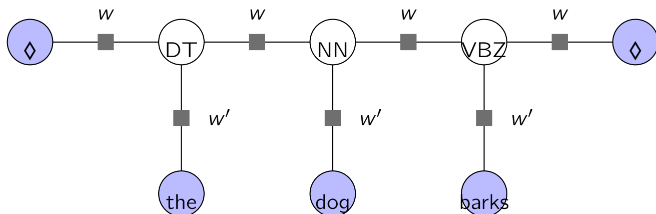
Stochastic gradient descent and error-driven learning

Avoiding the partition function and its derivatives

Weakly labelled training data

Summary

Features in Random Fields



$$\mathsf{P}(x) = \frac{1}{Z} w_{\lozenge,\mathrm{DT}}\, w'_{\mathrm{DT},\mathrm{the}}\, w_{\mathrm{DT},\mathrm{NN}}\, w'_{\mathrm{NN},\mathrm{dog}}\, w_{\mathrm{NN},\mathrm{VBZ}}\, w'_{\mathrm{VBZ},\mathrm{barks}}\, w_{\mathrm{VBZ},\lozenge}$$

- If $\mathcal{V}$ is the set of words and $\mathcal{Y}$ is the set of labels, there is a feature for each combination in $\mathcal{Y} \times \mathcal{V}$ and for each combination in $\mathcal{Y} \times \mathcal{Y}$.
- If $n_{y,y'}$ is the number of times label y precedes label y' in x, and $m_{y,v}$ is the number of times label y appears with word v, then:

$$\mathsf{P}(x) = \frac{1}{Z} \left(\prod_{y,y \in \mathcal{Y} \times \mathcal{Y}} w_{y,y'}^{n_{y,y'}} \right) \left(\prod_{y,v \in \mathcal{Y} \times \mathcal{V}} {w'}_{y,v}^{n_{y,v}} \right)$$

PCFGs and HMMs as exponential models

- Models like PCFGs and HMMs define the probability of a structure (e.g., a parse tree) *as a product of the probabilities of its components*
  - In a PCFG, each rule $A \to \beta$ has a probability $p_{A \to \beta}$
  - The probability of a tree is *the product of the probabilities of the rules used in its derivation*

$$\mathrm{P}(x) = \prod_{A \to \beta \in \mathcal{R}} p_{A \to \beta}^{n_{A \to \beta}(x)}$$

  where $n_{A \to \beta}(x)$ is the number of times rule $A \to \beta$ is used in derivation of tree x

$\Rightarrow$ A PCFG can be expressed as an exponential model where:

  - define a 1-to-1 mapping from PCFG rules to features (i.e., number the rules)
  - define the feature functions: $f_{A \to \beta}(x) = n_{A \to \beta}(x)$, and
  - set the feature values: $v_{A \to \beta} = p_{A \to \beta}$

$$\mathrm{P}(x) = \prod_{A \to \beta \in \mathcal{R}} v_{A \to \beta}^{f_{A \to \beta}(x)}$$

$\Rightarrow$ *A PCFG (and an HMM) is an exponential model where $Z = 1$*

Categorical features

- Suppose $(g_1, \ldots, g_m)$ are *categorical features*, where g_k ranges over $\mathcal{G}_k$
  - E.g., if $\mathcal{X}$ is a set of words, then $\textsf{suffix}(x)$ might be the last letter of x
- *"One-hot" encoding of categorical features:*
  - Define a binary feature $f_{g_k=c}$ for each combination of a categorical feature g_k, $k = 1, \ldots, m$ and a possible value $c \in \mathcal{G}_k$

$$f_{g_k=c}(x) \quad = \quad 1 \quad \text{if } g_k(x) = c, \text{ and 0 otherwise}$$

$\Rightarrow$ Number of binary features grows extremely rapidly

  - reranking parser has about 40 categorical features, but around 2 million binary features
- But you *only need to instantiate feature-value pairs observed in training data*
  - learning procedures in general set $w_{g=c} = 0$ if feature-value pair $g(x) = c$ is not present in training data

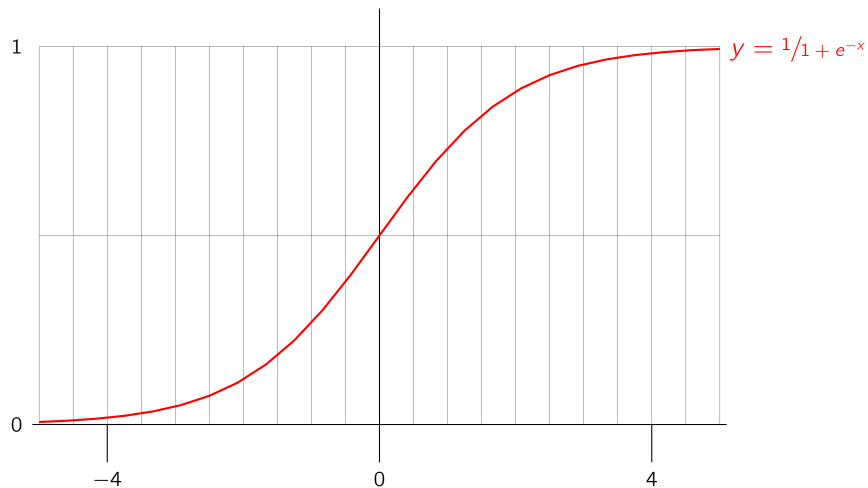
Feature redundancy in binary models

- Consider a situation where there are 2 outcomes: $\mathcal{X} = \{a, b\}$

$$\begin{aligned}
\mathrm{P}(X{=}x) &= \frac{1}{Z} \exp\left(\boldsymbol{w} \cdot \boldsymbol{f}(x)\right), \text{ where:} \\
Z &= \sum_{x' \in \mathcal{X}} \exp\left(\boldsymbol{w} \cdot \boldsymbol{f}(x')\right) = \exp\left(\boldsymbol{w} \cdot \boldsymbol{f}(a)\right) + \exp\left(\boldsymbol{w} \cdot \boldsymbol{f}(b)\right), \text{ so:} \\
\mathrm{P}(X{=}a) &= \frac{\exp\left(\boldsymbol{w} \cdot \boldsymbol{f}(a)\right)}{\exp\left(\boldsymbol{w} \cdot \boldsymbol{f}(a)\right) + \exp\left(\boldsymbol{w} \cdot \boldsymbol{f}(b)\right)} \\
&= \frac{1}{1 + \exp\left(\boldsymbol{w} \cdot (\boldsymbol{f}(b) - \boldsymbol{f}(a))\right)} \\
&= s(\boldsymbol{w} \cdot (\boldsymbol{f}(a) - \boldsymbol{f}(b))), \text{ where:} \\
s(z) &= \frac{1}{1 + \exp(-z)} \text{ is the } \textit{logistic sigmoid function}
\end{aligned}$$

$\Rightarrow$ In binary models *only the difference between feature values matters*

The logistic sigmoid function



Feature redundancy in exponential models

- This result generalises to all exponential models
- Let $\boldsymbol{u} = (u_1, \ldots, u_m)$ be *any vector* of same dimensionality as the features
- Define an exponential model using *new feature functions* $\boldsymbol{f}'(x) = \boldsymbol{f}(x) + \boldsymbol{u}$. Then:

$$
\begin{aligned}
\mathrm{P}(X{=}x) &= \frac{\exp\left(\boldsymbol{w} \cdot \boldsymbol{f}'(x)\right)}{\sum_{x' \in \mathcal{X}} \exp\left(\boldsymbol{w} \cdot \boldsymbol{f}'(x')\right)} \\
&= \frac{\exp\left(\boldsymbol{w} \cdot \boldsymbol{f}(x)\right)\ \exp\left(\boldsymbol{w} \cdot \boldsymbol{u}\right)}{\sum_{x' \in \mathcal{X}} \exp\left(\boldsymbol{w} \cdot \boldsymbol{f}(x')\right)\ \exp\left(\boldsymbol{w} \cdot \boldsymbol{u}\right)} \\
&= \frac{\exp\left(\boldsymbol{w} \cdot \boldsymbol{f}(x)\right)}{\sum_{x' \in \mathcal{X}} \exp\left(\boldsymbol{w} \cdot \boldsymbol{f}(x')\right)}
\end{aligned}
$$

$\Rightarrow$ *Adding or subtracting a constant vector to feature values does not change the distribution defined by an exponential model*

- The feature extractor for the reranking parser subtracts the vector $\boldsymbol{u}$ that *makes the feature vectors as sparse as possible*

Outline

Introducing exponential models

Features in exponential models

Learning exponential models

Regularisation

Conditional models

Stochastic gradient descent and error-driven learning

Avoiding the partition function and its derivatives

Weakly labelled training data

Summary

Methods for learning from data

- Learning or estimating feature weights $\boldsymbol{w}$ from *training data* $D = (x_1, \ldots, x_n)$, where each $x_i \in \mathcal{X}$
- *Maximum likelihood:* choose $\boldsymbol{w}$ to make D as likely as possible

$$\widehat{\boldsymbol{w}} = \underset{\boldsymbol{w}}{\operatorname{argmax}} L_D(\boldsymbol{w}), \text{ where:}$$

$$L_D(\boldsymbol{w}) = \prod_{i=1}^{n} \mathsf{P}_{\boldsymbol{w}}(x_i)$$

- *Minimising negative log likelihood* is mathematically equivalent, and has mathematical and computational advantages
  - negative log likelihood is *convex* (with fully visible training data)
  - ⇒ single optimum that can be found by "following gradient downhill"
  - avoids floating point underflow
- But other learning methods may have advantages
  - with a large number of features, a *regularisation penalty term* (e.g., L1 and/or L2 prior) helps to avoid overfitting
  - *optimising a specialised loss function* (e.g., expected f-score) can improve performance on a specific task

Learning as minimising a loss function

- Goal: find the feature weights $\widehat{\boldsymbol{w}}$ that *minimise the negative log likelihood* ℓ_D of feature weights $\boldsymbol{w}$ given data $D = (x_1, \ldots, x_n)$:

$$
\begin{aligned}
\widehat{\boldsymbol{w}} &= \operatorname*{argmin}_{\boldsymbol{w}} \ell_D(\boldsymbol{w}) \\
\ell_D(\boldsymbol{w}) &= -\log L_D(\boldsymbol{w}) \;=\; -\log \prod_{i=1}^{n} \mathsf{P}_{\boldsymbol{w}}(x_i) \\
&= \sum_{i=1}^{n} -\log \mathsf{P}_{\boldsymbol{w}}(x_i)
\end{aligned}
$$

- The negative log likelihood ℓ_D is a sum of the *losses* $-\log \mathsf{P}_{\boldsymbol{w}}(x_i)$ the model $\boldsymbol{w}$ incurrs on each data item x_i
- The maximum likelihood estimator selects the model $\widehat{\boldsymbol{w}}$ that minimises the loss ℓ_D on data set D
- Many other machine learning algorithms for estimating $\boldsymbol{w}$ from D can be understood as minimising some loss function

Why is learning exponential models hard?

- Exponential models are so flexible because the features can have arbitrary weights

$\Rightarrow$ The partition function Z is required to ensure the distribution $\mathrm{P}(x)$ is normalised

- The partition function *Z varies as a function of $\boldsymbol{w}$*

$$\mathrm{P}(X{=}x) = \frac{1}{Z}\exp\left(\boldsymbol{w}\cdot\boldsymbol{f}(x)\right), \text{ where:}$$

$$Z = \sum_{x'\in\mathcal{X}}\exp\big(\boldsymbol{w}\cdot\boldsymbol{f}(x')\big)$$

$\Rightarrow$ So we can't ignore Z, which makes it hard to optimise the likelihood!

- no closed-form solution for the feature weights w_j
- learning usually involves *numerically optimising* the likelihood function or some other loss function
- calculating Z requires *summing over entire space $\mathcal{X}$*
- many methods for approximating Z and/or its derivatives; typically unclear how the approximations affect the estimates of $\boldsymbol{w}$

The derivative of the negative log likelihood

- Efficient numerical optimisation routines require evaluation of the function to be minimised (negative log likelihood ℓ_D) *and its derivatives*
  - use a standard package; L-BFGS (LMVM), conjugate gradient
- We'll optimise $1/n$ times the negative log likelihood of $\boldsymbol{w}$ given data $D = (x_1, \ldots, x_n)$:

$$\ell_D(\boldsymbol{w}) = -\frac{1}{n}\sum_{i=1}^{n} \log \mathsf{P}_{\boldsymbol{w}}(x_i) = \log Z - \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{w} \cdot \boldsymbol{f}(x_i)$$

- The derivative of ℓ is:

$$\frac{\partial \ell_D}{\partial w_j} = \mathsf{E}_{\boldsymbol{w}}[f_j] - \mathsf{E}_D[f_j], \text{ where:}$$

$$\mathsf{E}_{\boldsymbol{w}}[f_j] = \sum_{x' \in \mathcal{X}} f_j(x')\mathsf{P}_{\boldsymbol{w}}(x') \quad \textit{(expected value of } f_j \textit{ wrt } \mathsf{P}_{\boldsymbol{w}}\textit{)}$$

$$\mathsf{E}_D[f_j] = \frac{1}{n}\sum_{i=1}^{n} f_j(x_i) \quad \textit{(expected value of } f_j \textit{ wrt } D\textit{)}$$

- At optimum $\partial \ell_D / \partial \boldsymbol{w} = \mathbf{0}$
  $\Rightarrow$ *model's expected feature values equals data's feature values*

Exercise: derive the formulae on the previous slide!

- This is a basic result for exponential models that is the basis of many other results
- If you want to generalise exponential models, you'll need to derive similar formulae
- You'll need to know:
  - that derivatives distribute over sums
  - that $\partial \log(x) / \partial x = 1/x$
  - the chain rule, i.e., that $\partial y / \partial x = \partial y / \partial u \; \partial u / \partial x$

Maximum entropy models

- Idea: given training data D and feature functions $\boldsymbol{f}$, find the distribution $\mathsf{P}'(X)$ that:
    1. $\mathsf{E}_{\mathsf{P}'}[f_j] = \mathsf{E}_D[f_j]$ for all features f_j,
       i.e., P' agrees with D on the features
    2. of all distributions satisfying (1), P' has *maximum entropy*
       i.e., P' has the least possible additional information
- Because $\widehat{\boldsymbol{w}} = \operatorname{argmin}_{\boldsymbol{w}} \ell_D(\boldsymbol{w})$ then

$$\frac{\partial \ell_D}{\partial \boldsymbol{w}}(\widehat{\boldsymbol{w}}) \quad = \quad \mathbf{0}$$

$\Rightarrow$ $\mathsf{E}_{\boldsymbol{w}}[f_j] = \mathsf{E}_D[f_j]$ for all features f_j

- Theorem: $\mathsf{P}_{\boldsymbol{w}} = \mathsf{P}'$, i.e., for any data D and feature functions $\boldsymbol{f}$ *the maximum likelihood distribution and the maximum entropy distribution are the same distribution*

Outline

Introducing exponential models

Features in exponential models

Learning exponential models

Regularisation

Conditional models

Stochastic gradient descent and error-driven learning

Avoiding the partition function and its derivatives

Weakly labelled training data

Summary

Why regularise?

- If every $x \in D$ has feature f_j and some $x \in \mathcal{X}$ does not, then $\widehat{w_j} = \infty$
- If no $x \in D$ has feature f_j and some $x \in \mathcal{X}$ does, then $\widehat{w_j} = -\infty$
- Infinities cause problems for numerical routines
- Just because a feature always occurs/doesn't occur in training data doesn't mean this will also occur in test data ("accidental zeros")
- These are extreme examples of *overlearning*
  - overlearning often occurs when the size of the data D is not much greater than the number of features m
- Idea: add a *regulariser* (also called a penalty term or prior) to the negative log likelihood that *penalises large feature weights*
  - Recall that $w_j = 0$ means that feature f_j is ignored

L_2 regularisation

- Instead of minimising the negative log likelihood $\ell_D(\boldsymbol{w})$, we optimise

$$\begin{aligned}
\widehat{\boldsymbol{w}} &= \underset{\boldsymbol{w}}{\text{argmin}}\ \ell_D(\boldsymbol{w}) + c\, R(\boldsymbol{w}), \text{ where:} \\
R(\boldsymbol{w}) &= \|\boldsymbol{w}\|_2^2 \\
&= \boldsymbol{w} \cdot \boldsymbol{w} \\
&= \sum_{j=1}^{m} w_j^2
\end{aligned}$$

- R is a *penalty term* that varies with each feature weight w_j such that:
  - the penalty is zero when $w_j = 0$,
  - the penalty is *greater than zero whenever* $w_j \neq 0$, and
  - the penalty *grows as* w_j *moves further away from* 0
- The *regulariser constant* c is usually set to *optimise performance on held-out data*

Bayesian MAP estimation

- Recall Bayesian belief updating:

$$\underbrace{\mathsf{P}(\text{Hypothesis} \mid \text{Data})}_{\text{Posterior}} \propto \underbrace{\mathsf{P}(\text{Data} \mid \text{Hypothesis})}_{\text{Likelihood}} \underbrace{\mathsf{P}(\text{Hypothesis})}_{\text{Prior}}$$

- In our setting:
  - Data $= D = (x_1, \ldots, x_n)$
  - Hypothesis $= \boldsymbol{w} = (w_1, \ldots, w_m)$
- If we want the MAP (Maximum Aposteriori) estimate for $\boldsymbol{w}$:

$$\begin{aligned}
\widehat{\widehat{\boldsymbol{w}}} &= \operatorname*{argmax}_{\boldsymbol{w}} \underbrace{\mathsf{P}(\boldsymbol{w} \mid D)}_{\text{Posterior}} \\
&= \operatorname*{argmax}_{\boldsymbol{w}} \underbrace{\mathsf{P}(D \mid \boldsymbol{w})}_{\text{Likelihood}} \underbrace{\mathsf{P}(\boldsymbol{w})}_{\text{Prior}} \\
&= \operatorname*{argmax}_{\boldsymbol{w}} \left(\prod_{i=1}^{n} \mathsf{P}(x_i \mid \boldsymbol{w}) \right) \mathsf{P}(\boldsymbol{w})
\end{aligned}$$

Regularisation as Bayesian MAP estimation

- Restate the MAP estimate in terms of negative log likelihood ℓ_D:

$$\begin{aligned}
\widehat{\widehat{\boldsymbol{w}}} &= \underset{\boldsymbol{w}}{\operatorname{argmax}} \underbrace{\left(\prod_{i=1}^{n} \mathsf{P}(x_i \mid \boldsymbol{w})\right)}_{\text{Likelihood}} \underbrace{\mathsf{P}(\boldsymbol{w})}_{\text{Prior}} \\
&= \underset{\boldsymbol{w}}{\operatorname{argmin}} \left(-\sum_{i=1}^{n} \log \mathsf{P}(x_i \mid \boldsymbol{w})\right) - \log \mathsf{P}(\boldsymbol{w}) \\
&= \underset{\boldsymbol{w}}{\operatorname{argmin}}\ \ell_D(\boldsymbol{w}) - \log \mathsf{P}(\boldsymbol{w}), \text{ where:} \\
\ell_D(\boldsymbol{w}) &= -\sum_{i=1}^{n} \log \mathsf{P}(x_i \mid \boldsymbol{w})
\end{aligned}$$

$\Rightarrow$ *MAP estimate $\widehat{\widehat{\boldsymbol{w}}}$ equals regularised MLE $\widehat{\boldsymbol{w}}$*

$$\widehat{\boldsymbol{w}} = \underset{\boldsymbol{w}}{\operatorname{argmin}}\ \ell_D(\boldsymbol{w}) + c\,R(\boldsymbol{w})$$

if $cR(\boldsymbol{w}) = -\log \mathsf{P}(\boldsymbol{w})$, i.e., *if the regulariser is the negative log prior*

L_2 regularisation as a Gaussian prior

- What kind of prior is an L_2 regulariser?
- If $cR(\boldsymbol{w}) = -\log \mathsf{P}(\boldsymbol{w})$ then

$$\mathsf{P}(\boldsymbol{w}) \quad = \quad \exp\left(-cR(\boldsymbol{w})\right)$$

- If $R(\boldsymbol{w}) = \|\boldsymbol{w}\|_2^2 = \sum_{j=1}^{m} w_j^2$, then *the prior is a zero-mean Gaussian*

$$\mathsf{P}(\boldsymbol{w}) \quad \propto \quad \exp\left(-c \sum_{j=1}^{m} w_j^2\right)$$

  The additional factors in the Gaussian become constants in log probability space, and therefore can be ignored when finding $\widehat{\boldsymbol{w}}$
- L_2 regularisation is also known as *ridge regularisation*

L_1 regularisation or Lasso regularisation

- The L_1 norm is the sum of the absolute values

$$\begin{aligned} R(\boldsymbol{w}) &= \|\boldsymbol{w}\|_1 \\ &= \sum_{j=1}^{m} |w_j| \end{aligned}$$

- L_1 regularisation is popular because it produces *sparse feature weights*
  - a feature weight vector $\boldsymbol{w}$ is *sparse* iff most of its values are zero
- But it's difficult to optimise L_1-regularised log-likelihood because *its derivative is discontinuous at the orthant boundaries*

$$\frac{\partial R}{\partial w_j} = \begin{cases} +1 & \text{if } w_j > 0 \\ -1 & \text{if } w_j < 0 \end{cases}$$

- Specialised versions of standard numerical optimisers have been developed to optimise L_1-regularised log-likelihood

What does regularisation do?

- Regularised negative log likelihood

$$\widehat{\boldsymbol{w}} = \underset{\boldsymbol{w}}{\operatorname{argmin}} \; \ell_D(\boldsymbol{w}) + cR(\boldsymbol{w})$$

- At the optimum weights $\widehat{\boldsymbol{w}}$, for each j:

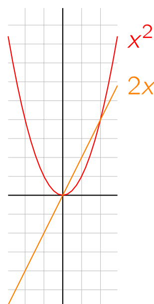
$$\begin{aligned} \frac{\partial \ell_D}{\partial w_j} + c\frac{\partial R}{\partial w_j} &= 0, \text{ or equivalently} \\ \mathsf{E}_D[f_j] - \mathsf{E}_{\boldsymbol{w}}[f_j] &= c\frac{\partial R}{\partial w_j} \end{aligned}$$

I.e., the regulariser gives the model some "slack" in requiring the empirical expected feature values equal the model's predicted expected feature values.

Why does L_1 regularisation produce sparse weights?

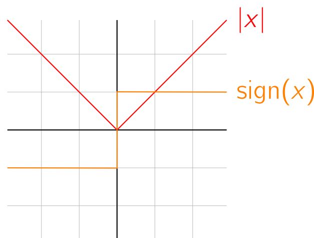
- Regulariser's derivative specifies gap between empirical and model feature expectation

$$\mathsf{E}_D[f_j] - \mathsf{E}_{\boldsymbol{w}}[f_j] \quad = \quad c\frac{\partial R}{\partial w_j}$$



- For L_2 regularisation, $\frac{\partial R}{\partial w_j} \to 0$ as $w_j \to 0$
  - little effect on small w

  $\Rightarrow$ no reason for feature weights to be zero



- For L_1 regularisation, $\frac{\partial R}{\partial w_j} \to \text{sign}(w_j)$ as $w_j \to 0$
  - regulariser has effect whenever $w \neq 0$
  - regulariser drives feature weights to 0 whenever "expectation gap" $< c$

Group sparsity via the Group Lasso

- Sometimes features come in natural groups; e.g., $\boldsymbol{F} = (\boldsymbol{f}_1, \ldots, \boldsymbol{f}_m)$, where each $\boldsymbol{f}_j = (f_{j,1}, \ldots, f_{j,v_j}), j = 1, \ldots, m$
- Corresponding weights $\boldsymbol{W} = (\boldsymbol{w}_1, \ldots, \boldsymbol{w}_m)$, where each $\boldsymbol{w}_j = (w_{j,1}, \ldots, w_{j,v_j})$

$$\mathrm{P}(X{=}x) \quad = \quad \frac{1}{Z} \exp\left(\sum_{j=1}^{m} \sum_{k=1}^{v_j} w_{j,k}\, f_{j,k}(x) \right)$$

- We'd like *group sparsity*, i.e., for "most" $j \in 1, \ldots, m$, $\boldsymbol{w}_j = \mathbf{0}$
- The *group Lasso* regulariser achieves this:

$$\begin{aligned} R(\boldsymbol{W}) \quad &= \quad \sum_{j=1}^{m} c_j \left\| \boldsymbol{w}_j \right\|_2 \\ &= \quad \sum_{j=1}^{m} c_j \left(\sum_{k=1}^{v_j} w_{j,k}^2 \right)^{1/2} \end{aligned}$$

Optimising the regularised log likelihood

- Learning feature weights involves optimising regularised likelihood

$$\begin{aligned}
\widehat{\boldsymbol{w}} &= \underset{\boldsymbol{w}}{\operatorname{argmin}}\ \ell_D(\boldsymbol{w}) + cR(\boldsymbol{w}) \\
\ell_D(\boldsymbol{w}) &= -\frac{1}{n}\sum_{i=1}^{n} \log \mathrm{P}(x_i) \ = \ \log Z - \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{w}\cdot\boldsymbol{f}(x_i) \\
Z &= \sum_{x'\in\mathcal{X}} \exp\big(\boldsymbol{w}\cdot\boldsymbol{f}(x')\big)
\end{aligned}$$

- Challenges in optimisation:
  - If regulariser R is not differentiable (e.g., $R = L_1$), then you need a specialised optimisation algorithm to handle discontinuous derivatives
  - if $\mathcal{X}$ is large (infinite), calculating Z may be difficult because it involves summing over $\mathcal{X}$

  $\Rightarrow$ just evaluate on the subset $\mathcal{X}' \subset \mathcal{X}$ where $\boldsymbol{w}\cdot\boldsymbol{f}$ is largest (assuming you can find it)

Outline

Introducing exponential models

Features in exponential models

Learning exponential models

Regularisation

Conditional models

Stochastic gradient descent and error-driven learning

Avoiding the partition function and its derivatives

Weakly labelled training data

Summary

Why conditional models?

- In a conditional model, each datum is a pair (x, y), where $x \in \mathcal{X}$ and $y \in \mathcal{Y}$
- The goal of a conditional model is to *predict y given x*
- Usually x is an item or an observation and y is a *label* for x
  - e.g., $\mathcal{X}$ is the set of all possible news articles, and $\mathcal{Y}$ is a set of topics, e.g. $\mathcal{Y} = \{\text{finance}, \text{sports}, \text{politics}, \ldots\}$
  - e.g., $\mathcal{X}$ is the set of all possible 256×256 images, and $\mathcal{Y}$ is a set of labels, e.g., $\mathcal{Y} = \{\text{cat}, \text{dog}, \text{person}, \ldots\}$
  - e.g., $\mathcal{X}$ is the set of all possible Tweets, and $\mathcal{Y}$ is a Boolean value indicating whether $x \in \mathcal{X}$ expresses a sentiment
  - e.g., $\mathcal{X}$ is the set of all possible sentiment-expressing Tweets, and $\mathcal{Y}$ is a Boolean value indicating whether $x \in \mathcal{X}$ has positive or negative sentiment
- We will do this by learning a *conditional probability distribution* $\mathrm{P}(Y \mid X)$, which is the probability of Y *given* X
- We estimate $\mathrm{P}(Y \mid X)$ from data $D = ((x_1, y_1), \ldots, (x_n, y_n))$, that consists of pairs of items x_i and their labels y_j (*supervised learning*)
  - in *unsupervised learning* we are only given the data items x_i, but not their labels y_i (clustering)
  - in *semi-supervised learning* we are not given the labels y_i for all data items x_i (we might be given only some labels, or the labels might only be partially identified)

Conditional exponential models

- Data $D = ((x_1, y_1), \ldots, (x_n, y_n))$ consists of (x, y) *pairs*, where $x \in \mathcal{X}$ and $y \in \mathcal{Y}$
- Want to predict y from x, for which we only need *conditional distribution* $\mathrm{P}(Y \mid X)$, not the joint distribution $\mathrm{P}(X, Y)$
- Features are now functions $f(x, y)$ over (x, y) pairs
- Conditional exponential model:

$$\begin{aligned} \mathrm{P}(y \mid x) &= \frac{1}{Z(x)} \exp\left(\boldsymbol{w} \cdot \boldsymbol{f}(x, y)\right), \text{ where:} \\ Z(x) &= \sum_{y' \in \mathcal{Y}} \exp\left(\boldsymbol{w} \cdot \boldsymbol{f}(x, y')\right) \end{aligned}$$

- Big advantage: $Z(x)$ only requires a sum over $\mathcal{Y}$, while "joint" partition function Z requires a sum over all $\mathcal{X} \times \mathcal{Y}$ pairs
  - in many applications label set $\mathcal{Y}$ is small
  - *size of $\mathcal{X}$ doesn't affect computational effort to compute $Z(x)$*

Features in conditional models

- In a conditional model, changing the feature function $\boldsymbol{f}(x, y)$ to $\boldsymbol{f}'(x, y) = \boldsymbol{f}(x, y) + \boldsymbol{u}(x)$ *does not change the distribution* $\mathrm{P}(y \mid x)$
  - ⇒ *adding or subtracting a function that only depends on x does not affect a conditional model*
  - ⇒ *to be useful in a conditional model, a feature must be a non-constant function of y*
- A feature $f(x, y) = f(y)$ that only depends on y behaves like a *bias node* in a neural net
  - it's often a good idea to have a "one-hot" feature for each $c \in \mathcal{Y}$:

$$f_{y=c}(y) \quad = \quad 1 \text{ if } y = c \text{, and } 0 \text{ otherwise}$$

- If $\mathcal{X}$ is a set of discrete categories, it's often useful to have pairwise "one-hot" features for each $c \in \mathcal{X}$ and $c' \in \mathcal{Y}$

$$f_{x=c,y=c'}(x, y) \quad = \quad 1 \text{ if } x = c \text{ and } y = c' \text{, and } 0 \text{ otherwise}$$

Using a conditional model to make predictions

- Labelling problem: we have feature weights $\boldsymbol{w}$ and want to predict label y for some x
- The most probable label $\widehat{y}(x)$ given x is:

$$\begin{aligned}
\widehat{y}(x) &= \operatorname*{argmax}_{y' \in \mathcal{Y}} \mathsf{P}_{\boldsymbol{w}}(Y{=}y' \mid X{=}x) \\
&= \operatorname*{argmax}_{y' \in \mathcal{Y}} \frac{1}{Z(x)} \exp\big(\boldsymbol{w} \cdot \boldsymbol{f}(x, y')\big) \\
&= \operatorname*{argmax}_{y' \in \mathcal{Y}} \boldsymbol{w} \cdot \boldsymbol{f}(x, y')
\end{aligned}$$

- Partition function $Z(x)$ is a constant here, so drops out

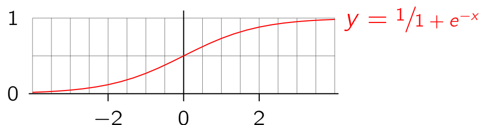
Logistic regression

- Suppose $\mathcal{Y} = \{0, 1\}$, i.e., our labels are Boolean

$$\begin{aligned}
\mathrm{P}(Y{=}1 \mid X{=}x) &= \frac{\exp\left(\boldsymbol{w} \cdot \boldsymbol{f}(x, 1)\right)}{\exp\left(\boldsymbol{w} \cdot \boldsymbol{f}(x, 0)\right) + \exp\left(\boldsymbol{w} \cdot \boldsymbol{f}(x, 1)\right)} \\
&= \frac{1}{1 + \exp\left(\boldsymbol{w} \cdot (\boldsymbol{f}(x, 0) - \boldsymbol{f}(x, 1))\right)} \\
&= \frac{1}{1 + \exp\left(-\boldsymbol{w} \cdot \boldsymbol{g}(x)\right)}, \text{ where:} \\
g_j(x) &= f_j(x, 1) - f_j(x, 0), \text{ for all } j \in 1, \ldots, m
\end{aligned}$$

$\Rightarrow$ Only *relative feature differences matter* in a conditional model

- Logistic sigmoid function:



Estimating conditional exponential models

- Compute *maximum conditional likelihood estimator* by minimizing negative log conditional likelihood

$$
\begin{aligned}
\widehat{\boldsymbol{w}} &= \operatorname*{argmin}_{\boldsymbol{w}} \ \ell_D(\boldsymbol{w}), \text{ where:} \\
\ell_D(\boldsymbol{w}) &= -\sum_{i=1}^{n} \log \mathsf{P}_{\boldsymbol{w}}(y_i \mid x_i) \\
&= \sum_{i=1}^{n} (\log Z(x_i) - \boldsymbol{w} \cdot \boldsymbol{f}(x_i, y_i))
\end{aligned}
$$

- Derivatives are differences of *conditional expectations* and *empirical feature values*

$$
\begin{aligned}
\frac{\partial \ell_D}{\partial w_j} &= \sum_{i=1}^{n} \left(\mathsf{E}_{\boldsymbol{w}}[f_j \mid x_i] - f_j(x_i, y_i) \right), \text{ where:} \\
\mathsf{E}_{\boldsymbol{w}}[f_j \mid x] &= \sum_{y' \in \mathcal{Y}} f_j(x, y') \mathsf{P}_{\boldsymbol{w}}(y \mid x) \quad \textit{(expected value of } f_j \textit{ given } x\textit{)}
\end{aligned}
$$

Regularising conditional exponential models

- Calculating derivatives of conditional likelihood only requires summing over $\mathcal{Y}$, and not $\mathcal{X}$
  - not too expensive if $|\mathcal{Y}|$ is small
  - if $\mathcal{Y}$ has a regular structure (e.g., a sequence), then there may be efficient algorithms for summing over $\mathcal{Y}$
- Regularisation adds a *penalty term* to objective function we seek to optimise
- *Important to regularise* (unless number of features is small)
  - L_1 (Lasso) regularisation produces *sparse feature weights*
  - L_2 (ridge) regularisation produces dense feature weights
  - Group lasso regularisation produces group-level sparsity in feature weights

Outline

Introducing exponential models

Features in exponential models

Learning exponential models

Regularisation

Conditional models

Stochastic gradient descent and error-driven learning

Avoiding the partition function and its derivatives

Weakly labelled training data

Summary

Why stochastic gradient descent?

- For small/medium data sets, “batch” methods using standard numerical optimisation procedures (such as L-BFGS) can work very well
  - these directly *minimise the negative log likelihood* ℓ_D
  - to calculate the negative log likelihood and its derivatives requires a pass through the entire training data
- But for very large data sets (e.g., data sets that don’t fit into memory), or with very large models (such as neural nets), these can be too slow
- *Stochastic gradient descent* calculates a noisy gradient from a small subset of the training data, so it can learn considerably faster
  - but the solution it finds is often less accurate

Gradient descent and mini-batch algorithms

- Idea: to minimise $\ell_D(\boldsymbol{w})$, move in direction of negative gradient ${\partial \ell_D}/{\partial \boldsymbol{w}}$
- If $\widehat{\boldsymbol{w}}^{(t)}$ is current estimate of $\boldsymbol{w}$, update as follows:

$$\begin{aligned}
\widehat{\boldsymbol{w}}^{(t+1)} &= \widehat{\boldsymbol{w}}^{(t)} - \varepsilon \, \frac{\partial \ell_D}{\partial \boldsymbol{w}}(\widehat{\boldsymbol{w}}^{(t)}) \\
&= \widehat{\boldsymbol{w}}^{(t)} - \varepsilon \sum_{i=1}^{n} \left(\mathsf{E}_{\widehat{\boldsymbol{w}}^{(t)}}[\boldsymbol{f} \mid x_i] - \boldsymbol{f}(x_i, y_i) \right)
\end{aligned}$$

- ε is *step size*; can be difficult to find a good value for it!
- This is *not a good optimisation algorithm*, as it zig-zags across valleys
- Update is *difference between expected and empirical feature values*
  - Each update requires a full pass through $D \Rightarrow$ relatively slow
- "Mini-batch algorithms": calculate expectations on *a small sample of D* to determine weight updates

Stochastic gradient descent as mini-batch of size 1

- Stochastic Gradient Descent (SGD) is the mini-batch algorithm with a mini-batch of size 1
- If $\widehat{\boldsymbol{w}}^{(t)}$ is current estimate of $\boldsymbol{w}$, training data $D = ((x_1, y_1), \ldots, (x_n, y_n))$, and r_t is a random number in $1, \ldots, n$ then:

$$\begin{aligned}
\widehat{\boldsymbol{w}}^{(t+1)} &= \widehat{\boldsymbol{w}}^{(t)} - \varepsilon \left(\mathrm{E}_{\widehat{\boldsymbol{w}}^{(t)}}[\boldsymbol{f} \mid x_{r_t}] - \boldsymbol{f}(x_{r_t}, y_{r_t})\right), \text{ where:} \\
\mathrm{E}_{\boldsymbol{w}}[\boldsymbol{f} \mid x] &= \sum_{y' \in \mathcal{Y}} \boldsymbol{f}(x, y') \, \mathrm{P}_{\boldsymbol{w}}(y' \mid x) \\
\mathrm{P}_{\boldsymbol{w}}(y \mid x) &= \frac{1}{Z(x)} \exp\left(\boldsymbol{w} \cdot f(x, y)\right) \\
Z(x) &= \sum_{y' \in \mathcal{Y}} \exp\left(\boldsymbol{w} \cdot f(x, y')\right)
\end{aligned}$$

- Stochastic gradient descent updates estimate of $\boldsymbol{w}$ after seeing each training example

$\Rightarrow$ Learning can be very fast; might not even need a full pass over D

- Perhaps the most widely used learning algorithm today

The Perceptron algorithm as approximate SGD

- Idea: assume $\mathsf{P}_{\boldsymbol{w}}(y \mid x)$ is peaked around $\widehat{y}_{\boldsymbol{w}}(x) = \operatorname{argmax}_{y' \in \mathcal{Y}} \boldsymbol{w} \cdot \boldsymbol{f}(x, y')$. Then:

$$\begin{aligned} \mathsf{E}_{\boldsymbol{w}}[\boldsymbol{f} \mid x] &= \sum_{y' \in \mathcal{Y}} \boldsymbol{f}(x, y') \, \mathsf{P}_{\boldsymbol{w}}(y' \mid x) \\ &\approx \boldsymbol{f}(x, \widehat{y}(x)) \end{aligned}$$

- Plugging this into the SGD algorithm, we get:

$$\begin{aligned} \widehat{\boldsymbol{w}}^{(t+1)} &= \widehat{\boldsymbol{w}}^{(t)} - \varepsilon \left(\mathsf{E}_{\widehat{\boldsymbol{w}}^{(t)}}[\boldsymbol{f} \mid x_{r_t}] - \boldsymbol{f}(x_{r_t}, y_{r_t}) \right) \\ &\approx \widehat{\boldsymbol{w}}^{(t)} - \varepsilon \left(\boldsymbol{f}(x_{r_t}, \widehat{y}_{\widehat{\boldsymbol{w}}^{(t)}}(x)) - \boldsymbol{f}(x_{r_t}, y_{r_t}) \right) \end{aligned}$$

- This is an *error-driven learning rule*, since *no update is made on iteration t if* $\widehat{y}(x_{r_t}) = y_{r_t}$

Regularisation as weight decay in SGD and Perceptron

- Regularisation: minimise a *penalised negative log likelihood*

$$\widehat{\boldsymbol{w}} = \underset{\boldsymbol{w}}{\operatorname{argmin}} \; \ell_D(\boldsymbol{w}) + cR(\boldsymbol{w}), \text{ where:}$$

$$R(\boldsymbol{w}) = \begin{cases} \sum_{j=1}^{m} w_j^2 & \text{with an } L_2 \text{ regulariser} \\ \sum_{j=1}^{m} |w_j| & \text{with an } L_1 \text{ regulariser} \end{cases}$$

- Adding L_2 regularisation in SGD and Perceptron introduces *multiplicative weight decay*:

$$\widehat{\boldsymbol{w}}^{(t+1)} = \widehat{\boldsymbol{w}}^{(t)} - \varepsilon \left(\mathsf{E}_{\widehat{\boldsymbol{w}}^{(t)}}[\boldsymbol{f} \mid x_{r_t}] - \boldsymbol{f}(x_{r_t}, y_{r_t}) + 2c\widehat{\boldsymbol{w}}^{(t)} \right)$$

- Adding L_1 regularisation in SGD and Perceptron introduces *additive weight decay*:

$$\widehat{\boldsymbol{w}}^{(t+1)} = \widehat{\boldsymbol{w}}^{(t)} - \varepsilon \left(\mathsf{E}_{\widehat{\boldsymbol{w}}^{(t)}}[\boldsymbol{f} \mid x_{r_t}] - \boldsymbol{f}(x_{r_t}, y_{r_t}) + c \, \operatorname{sign}(\widehat{\boldsymbol{w}}^{(t)}) \right)$$

Stabilising SGD and the Perceptron

- The Perceptron is guaranteed to converge to a weight vector that correctly classifies all training examples *if the training data is separable*
- Most of our problems are *non-separable*
  - ⇒ SGD and the Perceptron never converge to a weight vector
  - ⇒ final weight vector depends on last examples seen
- Reducing learning rate ε in later iterations can stabilise weight vector $\widehat{\boldsymbol{w}}$
  - if learning rate is too low, SGD takes a long time to converge
  - if learning rate is too high, $\widehat{\boldsymbol{w}}$ can over-shoot
  - selecting appropriate learning rate is almost "black magic"
- *Bagging* can be used to stabilise SGD and perceptron
  - *construct multiple models* by running SGD or perceptron many times on random permutations of training data
  - combine predictions of models at run time by *averaging* or *voting*
- The *averaged perceptron* is a fast approximate version of bagging
  - train a single perceptron as usual
  - at end of training, average the weights *from all iterations*
  - use these averaged weights at run-time

ADAGRAD and ADADELTA

- There are many methods that attempt to automatically set the learning rate ε
- ADAGRAD and ADADELTA are two of the currently most popular methods
- ADAGRAD estimates a separate learning rate ε_j for each feature weight w_j
- If $g_j^{(t)}$ is the derivative of the regularised negative log likelihood ℓ_D w.r.t. feature weight w_j at step t, then the ADADGRAD update rule is:

$$\begin{aligned}\widehat{w}_j^{(t+1)} &= \widehat{w}_j^{(t)} - \frac{\eta}{\sqrt{\sum_{t'=1}^{t} g_j^{(t)}}}\, g_j^{(t)}, \text{ where:} \\ g_j^{(t)} &= \frac{\partial \ell_D(\boldsymbol{w}^{(t)})}{\partial w_j}\end{aligned}$$

- This effectively scales the learning rate so features with large derivatives or with fluctuating signs have a slower learning rate
- If a group of features are known to have the same scale, it may make sense for them to share the same learning rate
- The ADADELTA rule is newer and only slightly more complicated
- Both ADAGRAD and ADADELTA only require you to store the sum of the previous derivatives

Momentum

- Intuition: a ball rolling down a surface will settle at a (local) minimum
- Update should be a mixture of the previous update and derivative of regularised log likelihood ℓ_D

$$\begin{aligned} \widehat{\boldsymbol{w}}^{(t+1)} &= \widehat{\boldsymbol{w}}^{(t)} + \boldsymbol{v}^{(t+1)} \\ \boldsymbol{v}^{(t+1)} &= \alpha \boldsymbol{v}^{(t)} - (1-\alpha)\,\varepsilon\,\frac{\partial \ell_D(\widehat{\boldsymbol{w}}^{(t)})}{\partial \boldsymbol{w}} \end{aligned}$$

- Momentum can smooth statistical fluctuations in SGD derivatives
- If derivatives all point in roughly same direction, updates $\boldsymbol{v}$ can become quite large

  $\Rightarrow$ set learning rate ε to much lower than without momentum
  - typical value for momentum hyper-parameter $\alpha = 0.9$

Perceptron vs. L-BFGS in reranking parser

Averaged perceptron (randomised data)
Exponential model, adjusting regularizer constant c

0.91
0.908
0.906
0.904
0.902
0.9
0.898

f-score on section 24

0.898 0.9 0.902 0.904 0.906 0.908 0.91

f-score on sections 20-21

Comments on SGD and the Perceptron

- Widely used because *easy to implement* and *fast to train*
  - in my experience, not quite as good as numerical optimisation with L-BFGS
- Overlearning can be a problem
  - *regularisation* becomes *weight decay*
  - L_2 regularisation is *multiplicative weight decay*
  - L_1 regularisation is *subtractive weight decay*
  - often more or less ad hoc methods are used instead of or in addition to regularisation
    - averaging (bagging, averaged perceptron, etc.)
    - early stopping
- If you're using either SGD or Perceptron, *try ADAGRAD and ADADELTA learning rules*
  - these methods *automatically change the learning rate ε* during learning
  - they can identify *different learning rates for different features*

  ⇒ much faster learning that with SGD or Perceptron alone

Outline

Introducing exponential models

Features in exponential models

Learning exponential models

Regularisation

Conditional models

Stochastic gradient descent and error-driven learning

Avoiding the partition function and its derivatives

Weakly labelled training data

Summary

Challenges when $\mathcal{Y}$ is large

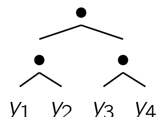
- The SGD update rule:

$$\begin{aligned}
\widehat{\boldsymbol{w}}^{(t+1)} &= \widehat{\boldsymbol{w}}^{(t)} - \varepsilon \left(\mathsf{E}_{\widehat{\boldsymbol{w}}^{(t)}}[\boldsymbol{f} \mid x_{r_t}] - \boldsymbol{f}(x_{r_t}, y_{r_t}) \right), \text{ where:} \\
\mathsf{E}_{\boldsymbol{w}}[\boldsymbol{f} \mid x] &= \sum_{y' \in \mathcal{Y}} \boldsymbol{f}(x, y')\, \mathsf{P}_{\boldsymbol{w}}(y' \mid x) \\
\mathsf{P}_{\boldsymbol{w}}(y \mid x) &= \frac{1}{Z(x)} \exp\left(\boldsymbol{w} \cdot f(x, y)\right) \\
Z(x) &= \sum_{y' \in \mathcal{Y}} \exp\left(\boldsymbol{w} \cdot f(x, y')\right)
\end{aligned}$$

- Each update step requires calculating the *partition function $Z(x)$ and its derivatives* $\mathsf{E}_{\widehat{\boldsymbol{w}}^{(t)}}[\boldsymbol{f} \mid x_{r_t}]$
- These require summing over $\mathcal{Y}$, which can dominate the computation time if $\mathcal{Y}$ is large
  - in modern speech recognition and machine translation systems, $\mathcal{Y}$ is the vocabulary of a natural language, so $|\mathcal{Y}| \approx 10^5$

Factoring $\mathbf{P}(Y \mid X)$

- Produce a hierarchical clustering of $\mathcal{Y}$, which defines a tree over the $\mathcal{Y}$.



- Train a separate model for each internal node in the tree
  - the probability of a leaf (output) node is the *product* of probabilities of each decision on the root to leaf path
- This usually does not produce a very good model
- Conjecture: bagging (e.g., averaging) the output of many such tree models would improve accuracy

Estimating expected feature counts by sampling

- SGD update rule:

$$\begin{aligned}\widehat{\boldsymbol{w}}^{(t+1)} &= \widehat{\boldsymbol{w}}^{(t)} - \varepsilon \left(\mathsf{E}_{\widehat{\boldsymbol{w}}^{(t)}}[\boldsymbol{f} \mid x_{r_t}] - \boldsymbol{f}(x_{r_t}, y_{r_t})\right), \text{ where:} \\ \mathsf{E}_{\boldsymbol{w}}[\boldsymbol{f} \mid x] &= \sum_{y' \in \mathcal{Y}} \boldsymbol{f}(x, y')\, \mathsf{P}_{\boldsymbol{w}}(y' \mid x)\end{aligned}$$

- Idea: use a sampling method to estimate the expected feature counts $\mathsf{E}_{\widehat{\boldsymbol{w}}^{(t)}}[\boldsymbol{f} \mid x_{r_t}]$
- Importance sampling:
  - draw samples from a *proposal distribution* over $\mathcal{Y}$ (e.g., unigram distribution)
  - calculate expectation from samples reweighted according to *importance weights* (which don't require partition function)
- May require a large number of samples to accurately estimate expectations

Noise-contrastive estimation

- Noise-contrastive estimation can be viewed as *importance sampling with only two samples* (and where importance weights are ignored)
- Suppose the training item at iteration t is $(x_{r_t}, y^+_{r_t})$.
- Set $y^-_t \in \mathcal{Y}$ to a random sample from a *proposal distribution* (e.g., unigram distribution over $\mathcal{Y}$)
- We approximate:

$$\mathsf{E}_{\widehat{\boldsymbol{w}}^{(t)}}[\boldsymbol{f} \mid x_{r_t}] \approx \frac{\boldsymbol{f}(x_{r_t}, y^+_{r_t}) \exp(\boldsymbol{w}^{(t)} \cdot \boldsymbol{f}(x_{r_t}, y^+_{r_t})) + \boldsymbol{f}(x_{r_t}, y^-_t) \exp(\boldsymbol{w}^{(t)} \cdot \boldsymbol{f}(x_{r_t}, y^-_t))}{\exp(\boldsymbol{w}^{(t)} \cdot \boldsymbol{f}(x_{r_t}, y^+_{r_t})) + \exp(\boldsymbol{w}^{(t)} \cdot \boldsymbol{f}(x_{r_t}, y^-_t))}$$

- If y^-_t is less probable than $y^+_{r_t}$ the expectation $\mathsf{E}_{\widehat{\boldsymbol{w}}^{(t)}}[\boldsymbol{f} \mid x_{r_t}] \approx \boldsymbol{f}(x_{r_t}, y^+_{r_t})$, so the expectations will cancel, and there won't be a large weight update
- If y^-_t is more probable than $y^+_{r_t}$ the expectation $\mathsf{E}_{\widehat{\boldsymbol{w}}^{(t)}}[\boldsymbol{f} \mid x_{r_t}] \approx \boldsymbol{f}(x_{r_t}, y^-_t)$, so there can be a large weight update
- Widely used in the neural net community today

Outline

Introducing exponential models

Features in exponential models

Learning exponential models

Regularisation

Conditional models

Stochastic gradient descent and error-driven learning

Avoiding the partition function and its derivatives

Weakly labelled training data

Summary

Ambiguous or weakly labelled training data as partial observations

- Suppose our training data doesn't tell us the *true label* y_i for each example x_i, but only provides us with *a set of labels* $\mathcal{Y}_i$ that contains the unknown true label y_i

$$\begin{aligned} D &= ((x_1, \mathcal{Y}_1), \ldots, (x_n, \mathcal{Y}_n)) \text{ where:} \\ y_i &\in \mathcal{Y}_i \subseteq \mathcal{Y} \end{aligned}$$

- Idea: learn a model that maximizes $\prod_{i=1}^{n} \mathrm{P}(\mathcal{Y}_i \mid x_i)$
- Example: in reranking the gold parse might not be in the beam, so train model to select one of the best parses available in beam; we don't care which is chosen
- Example: in arc-eager dependency parsing, several different moves can lead to same gold parse; we don't care which the parser chooses

Partially-observed conditional exponential models

- Data $D = (((x_1, \mathcal{Y}_1), \ldots, (x_n, \mathcal{Y}_n))$, where $\mathcal{Y}_i \subseteq \mathcal{Y}$ and $\mathcal{Y}_i \neq \varnothing$
- Compute *maximum conditional likelihood estimator* by minimizing negative log conditional likelihood

$$
\begin{aligned}
\widehat{\boldsymbol{w}} &= \underset{\boldsymbol{w}}{\operatorname{argmin}}\ \ell_D(\boldsymbol{w}), \text{ where:} \\
\ell_D(\boldsymbol{w}) &= -\sum_{i=1}^{n} \log \mathsf{P}_{\boldsymbol{w}}(\mathcal{Y}_i \mid x_i) \\
&= \sum_{i=1}^{n} \left(\log Z(x_i, \mathcal{Y}) - \log Z(x_i, \mathcal{Y}_i)\right), \text{ where:} \\
Z(x, \mathcal{Y}') &= \sum_{y' \in \mathcal{Y}'} \exp\left(\boldsymbol{w} \cdot \boldsymbol{f}(x, y')\right)
\end{aligned}
$$

- Intuition: $\log Z(x_i, \mathcal{Y}) - \log Z(x_i, \mathcal{Y}_i)$ will be small when most mass is assigned to $\mathcal{Y}_i$
- If $\mathcal{Y}_i = \mathcal{Y}$, then example i *has no information*
- Warning: ℓ_D is usually *not convex* $\Rightarrow$ local minima
  - hidden data problems usually have non-convex log likelihoods

Derivatives for partially-observed conditional models

- Negative log likelihood:

$$\begin{aligned} \ell_D(\boldsymbol{w}) &= \sum_{i=1}^{n} \left(\log Z(x_i, \mathcal{Y}) - \log Z(x_i, \mathcal{Y}_i)\right), \text{ where:} \\ Z(x, \mathcal{Y}') &= \sum_{y' \in \mathcal{Y}'} \exp\left(\boldsymbol{w} \cdot \boldsymbol{f}(x, y')\right) \end{aligned}$$

- Derivatives are differences of *two conditional expectations*

$$\begin{aligned} \frac{\partial \ell_D}{\partial w_j} &= \sum_{i=1}^{n} \left(\mathsf{E}_{\boldsymbol{w}}[f_j \mid x_i, \mathcal{Y}] - \mathsf{E}_{\boldsymbol{w}}[f_j \mid x_i, \mathcal{Y}_i]\right), \text{ where:} \\ \mathsf{E}_{\boldsymbol{w}}[f_j \mid x, \mathcal{Y}'] &= \sum_{y' \in \mathcal{Y}'} f_j(x, y') \mathsf{P}_{\boldsymbol{w}}(y' \mid x) \quad \textit{(expected value given } x \textit{ and } \mathcal{Y}'\textit{)} \end{aligned}$$

- These derivatives are no harder to compute than for the fully-observed case
- SGD and perceptron algorithms generalise straight-forwardly to partially-observed data

Partially-observed data in the reranker

- Training data consists of a sequence of *training data items* (sentences)
- Each data item consists of a sequence of *candidates* (parses)
  - the number of candidates per data item can vary
- Each candidate consists of a sequence of *feature-value pairs*
- Each feature is an integer, and each value is a floating-point number
  - feature value 1 is special-cased because it's so common in "one-hot" representations
- To allow partially-observed training data, each candidate has a *gold weight*
  - for a standard MaxEnt model, the gold candidate in each data item has gold weight 1, all others have gold weight 0
  - with partially-observed data, more than one candidate has weight 1

Other interesting things the reranker can do

- Data items (sentences) and candidates (parses) can be given "costs" so the reranker can calculate f-scores
  - can optimise *expected f-score* instead of log likelihood
  - useful with skewed data (e.g., in disfluency detection, where most words are fluent)
- The reranker uses L1 and/or L2 regularisation
  - can optimise regulariser constants to maximise log likelihood or f-score of heldout data
- Features are organised into *feature classes*
  - each feature class can have its own regulariser constant
  - these feature constants can be optimised can be on heldout data
- Standard optimiser is L-BFGS-OWLQN, but can also use Averaged Perceptron
  - Averaged Perceptron is not quite as good as L-BFGS, but *much faster*
  - Averaged Perceptron can be used to search for subset of feature classes that optimise f-score on heldout data

Outline

Introducing exponential models

Features in exponential models

Learning exponential models

Regularisation

Conditional models

Stochastic gradient descent and error-driven learning

Avoiding the partition function and its derivatives

Weakly labelled training data

Summary

Summary

- Maximum entropy models capture the intuition that *features interact multiplicatively*, i.e., can increase or decrease the probability of an outcome
- Calculating the *partition function* Z and its derivatives is usually the central challenge in MaxEnt modelling
- *Conditional* MaxEnt models, which model $\mathrm{P}(y \mid \boldsymbol{x})$, often have simpler partition functions than *joint* models, which model $\mathrm{P}(y, \boldsymbol{x})$.
- *Regularisation* is often essential to avoid over-learning
  - *L1 regularisation* produces sparse feature weight vectors $\boldsymbol{w}$ at the individual feature level
  - the *group Lasso* produces sparse feature weight vectors $\boldsymbol{w}$ at the feature group level
- Stochastic Gradient Descent (SGD) is an easy and fast way to learn MaxEnt models (but less accurate?)
- The Perceptron is SGD for conditional MaxEnt with a Viterbi approximation

Where we go from here

- *Conditional Random Fields* are conditional MaxEnt models that use dynamic programming to calculate the partition function and their derivatives
  - generally requires $\mathcal{Y}$ to have some kind of regular structure, e.g., a sequence (sequence labelling) or a tree (parsing)
- *Neural networks* use MaxEnt models as components (they are networks of MaxEnt models, but "neural net" sounds better!)
  - *Boltzmann machines* are MaxEnt models where the data items are graphs
  - *feed-forward networks* use conditional MaxEnt models as components