Lecture 8 mutliclass classification

wbg231

December 2022

1 motivation

- up to this point we have only done binary classification
- binary classification works well for a lot of problems (spam vs non spam) or (positive vs negative sentiment)
- but that does not deal with all types of questions, like maybe we want to recognize the objects in a photograph, or face recognition or recognizing letters from handwriting (some of these questions have 1000's or millions of classes)
- what are some potential issues with many classes?
  1. computation cost (may have to train many classification models)
  2. class imbalance (if we are training a classifier to detect dog or not dog there are likely many more examples of not dog than dog)
  3. different cost of errors (want to be sure it is a dog, versus most of the time it is not a dog. that is if we are training a one versus all classifier we want to only predict positive when we are really confident in our guess)

plan for today

- today we are going to explore the following questions
- how to reduce mutliclass classification into binary classification.
- we could train many binary classifiers, (could also do a linear regression and bin your regression outputs to a certain class like if $f(x) \in [0,1)$ say it is a dog if $f(x) \in [1,2)$ say it is a cat. this is kind of an arbitrary way to do it though.)
- how do generalize what we learned in binary classification to mutliclass classification (we can think about loss functions)
- we are also going to talk about structured prediction

2 reduction to binary classification

one vs all and one vs rest

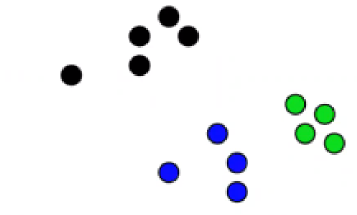
- suppose we have an input space X (general)
- and outputs space is discrete integers $Y = \{1...k\} : \forall y_i \in \mathbb{Z}$
- we are going to train k binary classifiers once for each class $h_1...h_k : X \to \mathbb{R}$ (map from input space to real numbers)
- classifier h_i distinguishes class i from all other classes (that is $h_i(x) = 1 \Rightarrow$ we think x is class i and $h_i(x) = -1 \Rightarrow$ we think x is not class i
- this is what we call the one versus all (or one versus rest)
- we predict by a **majority vote** such that

$$h(x) = argmax_{i \in [1,k]} h_i(x)$$

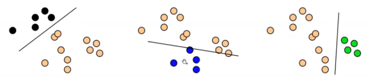
- and we break ties arbitrarily

example

- suppose we have a dataset with 3 classes that looks like this

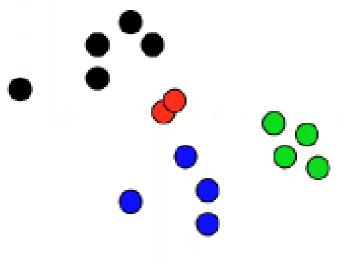


- the output of our model should be 1,2,3 for any given x
- we can train 1 versus all classifiers (here we are assuming only linear classifiers) but in theory it does not matter if they are linear or not.



- so we trained the following three classifiers with the following 3 decision boundaries
- here we are assuming that each set is linearly separable from the rest.
- in the ideal case only one of the classes should have a positive score and the others should have negative values

- there are cases where we can not linearly separate the data.
- and so one versus all (with a linear assumption) will not work here.



- such as here

all versus all, one vs one or all pairs

- as we showed above the one versus all does not work in all cases, such as when as the cases where the classes can not be serrated
- another way to deal with this is the all versus all classifier (also called one vs one or all pairs)
- suppose we have an input space X (general)
- and outputs space is discrete integers $Y = \{1...k\} : \forall y_i \in \mathbb{Z}$
- we are going to train $\binom{k}{2}$ (that is the number of permutations of length 2 ie all pairs of classes) where $h_{i,j} : X \to \mathbb{R}$ where $where i \in [1,k], j[i+1,k]$
- $h_i(x) = 1 \Rightarrow$ we think x is class i and $h_i(x) = -1 \Rightarrow$ we think x is class j
- we again predict using a majority vote such that

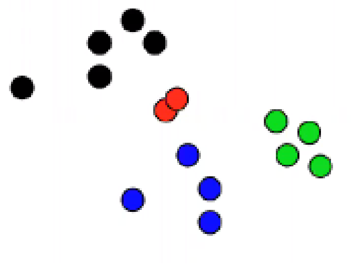
$$h(x) = argmax_{i\in[1,k]} \Sigma_{i\neq j} h_{i,j}(x)\mathbb{I}(i<j) + h_{j,i}(x)\mathbb{I}(j<i)$$

  so each class gets (k-1) votes for the k-1 classifiers that take value positive one if we think it is that class so at most you can get each class gets at most $k-1$

- again ties can be broken in an arbitrary way.
- note that as we are training $\binom{k}{2}$ classifiers our number of classifiers will quadratically increase with the number of classes
- in general the classifiers will not be calibrated so we will just sum up the integer number of votes

Example

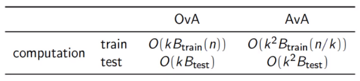
- suppose we again have this data set



- here we are assuming that each pair of classes is linearly separable this is more expressive than one versus all.
- so here we are training $\binom{4}{2} = 6$ classifiers
- this does septate the classes

one vs all vs all vs all

- here is a chart comparing on versus all and all versus all



		OvA	AvA
computation	train	$O(kB_{\text{train}}(n))$	$O(k^2B_{\text{train}}(n/k))$
	test	$O(kB_{\text{test}})$	$O(k^2B_{\text{test}})$

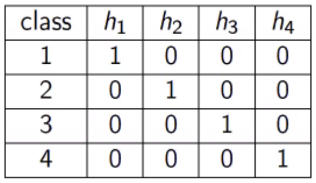
- here we are assuming that each pair of classes is linearly separable this is more expressive than one versus all.
- all vs all requires a quadratic number of classifiers while one vs all only requires a linear number of classifiers
- so when we are training a one vs all classifier we need to train each classifier on our total number of data points (that is all training data marked for every class)
- in contrast when we are training are all vs all classifier each classifier is only concerned with the data that deals with its one of two classes which in big O notation is $\frac{n}{k}$ (the point is you are dealing with a much smaller dataset for each of the small classifiers) (this also assumes that the dataset is balanced)
- in terms of test time there is not really a difference as in either case we are taking one example and running through all classifiers we have
- there are some challenges with this

1. while training a one vs all classifier if the data is imbalanced if you have 10 classes and 100 data points that are evenly distributed across all classes. for each 1 vs all classifier you will have 10 positive examples and 90 negative examples meaning we will have large calibration issues
2. all versus all while training has a small training set for each classifier (if again we have 100 data points that are evenly distributed across all classes then for each classifier we have 10 positive examples and 10 negative examples to train each classifier on)
3. so calibration (ie how we compare the real value output from our classifiers) is an issue (do we scale one class up or down)? is an issue for both models
4. tie breaking is also arbitrary which is not great

- so these are simple models, but they lack theoretical rigor.
- they do work well in practice though

code words for labels

- the basic idea here is we can encode labels as binary codes and predict the code bits directly
- this allows us to train fewer binary classifiers to do the same task.
- so suppose we want to do one versus all classification with 4 classes and 4 classifiers



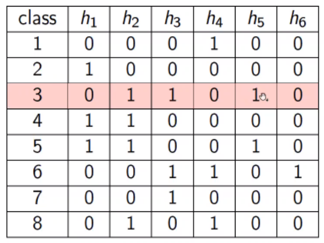
class	h_1	h_2	h_3	h_4
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

-
- here we are one hot encoding our labels, here each class gets one representation
- ideally there are only 4 situations that can happen. each of the class labels
- but in practice if we train our classifiers independently it is possible for us to get results that do not neatly fit into any class (like 1100)
- it is hard to predict using that.
- so this encoding uses k bits for k classes, can we reduce the number of bits we are using? yes given k bits there are 2^k total number of codes

- one versus all uses k bits to represent k classes. there are 2^k possible encodings done with k bits

error correcting output codes (ECOC)

- in general you can use between log(k) and k bits to represent k classes
- here is an example where we use 6 bits to represent 8 classes.



class	h_1	h_2	h_3	h_4	h_5	h_6
1	0	0	0	1	0	0
2	1	0	0	0	0	0
3	0	1	1	0	1	0
4	1	1	0	0	0	0
5	1	1	0	0	1	0
6	0	0	1	1	0	1
7	0	0	1	0	0	0
8	0	1	0	1	0	0

-
- so for instance class 3 is coded as 011011
- so in this context we train one classifiers h_i for each bit (so in the example 6) and classifier h_i takes value 1 when the i-th bit is 1, and -1 when the ith bit is zero
- we then predict classes based on the closest label in terms of hamming distance ie we take the label which would require the minimum bit changes to get. that is in this case basically how many bits different
- also note that all of our classifier's are independent in this case.
- so if we have log(k) bits then every possible combination of bits will be matched to a class exactly so we will not have to worry about hamming distance
- there is a problem of code design we want good binary classifiers, we may want our independent bits to represent something that can go do down a decision tree so that each classifier knows what it it does.
- this type of approach is called error correcting output codes. so even if we make a mistake and predict a class that does not exist we could correct, it using hamming distance
- this approach is computationally efferent we only need to use between log(k) and k bits.
- why not always use the minimal number of bits
    1. if the minimum hamming distance between any pair of codes words is d, then this algorithm can only correct $\lfloor \frac{d-1}{2} \rfloor$ errors.

2. that is if code names are close to one another then we are not very robust to errors ie all classifiers need to get every bit right or else we will mislabel an example

- so we want to find a ballance between using too many bits and not having enough classifiers to correct for errors
- there are some theoretical results that could be cool looking through later on the course website

review

- so far we have discussed approaches that reduce mutliclass problems to binary classifier problems
- the key here is to design a natural binary classification problem without using a lot of computation
- the issues are that this does not generalize to really large numbers of classes
- so there are other methods we can work with

3 mutliclass classification loss functions

review of binary logistic regression

- our goal is given an input x we want to predict an output of either zero or one.
- the output of our model is given by
$$f(x) = sigmoid(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-w^tx-b}}$$
this transforms our prediction to be between zero and one so we can interpret it as a probability
- the given we understand that $f(x) = P(x=1)$ it is natural that
$$P(x=0) = (1-P(x=1)) = 1-f(x) = \frac{e^{-w^tx-b}}{1+e^{-w^tx-b}} = \frac{e^{-w^tx-b}}{1+e^{-w^tx-b}} * \frac{e^{w^txb}}{e^{w^txb}} =$$
$$\frac{1}{1+e^{w^tx-n}} = sigmoid(-z) = f(-x)$$
- another way to think of this is that to train for one class (1) we are Learning parameters w,b and to train for the zero class we are learning parameters -w,-b

- so in binary logistic regression even if we are only training with one set of parameters we are implicitly learning another which is just fully determined by the first set

mutliclass logistic regression

- now what is we have one w_c for each class c
- we can define a more general type of sigmoid function also called the soft max function as
$$f_c(c) = \frac{e^{w_c^t x + b_c}}{\Sigma_c e^{w_c^t x + b_c}}$$
- the idea is that this score sums up to 1, but every classes is represented by a non-normalized score $e^{w_c^t + b_c}$
- loss function $L = \Sigma_i - y_c^i log(f_c(x^i))$
- this is a more generalized form of the binary case.
- they write that $\frac{\partial L}{\partial z} = f - y$ and say z is the sigmoid function but i am not sure what they are trying to say here

comparison to one versus all

- for both one vs all and logistic regression all the following holds
- the hypothesis space for k classes here is
$$\mathcal{F} = \{x \to h_i(x) | h_1...h_k \in \mathcal{H}\}$$
that is we are trinaing k independent classifiers which will all be in the single hypothesis $\mathcal{H} = \{h : X \to \mathbb{R}\}$ that is maps from our input space to real numbers
- $h_i(x)$ scores how likely we think it is that input x is from class i
- they just more or less use different score functions
- for one versus all objective $h_i(x) > 0$ for all x of label i and $h_i(x) < 0$ for all other labels
- we predict at test time to predict (x,i) correctly we just need $h_i(x) > h_j(x) \forall j \neq i$

multiclass perceptron

- for the perceptron model the base linear predictor is given by $h_(x) = w_i^t x : w \in \mathbb{R}^d$
- and the algorithm is

```
1. given a mutliclass dataset $\mathcal{D} = \{x, y\}$ so note that this means for each of the k predictors we are learning d weights so we will in total have a weight matrix $W \in \mathbb{R}^{k \times d}$
2. initialize $w \leftarrow 0$
3. for iter=1,2...T do
   (a) for $(x, y) \in \mathcal{D}$
       i. $\hat{y} = argmax_{y' \in y} w_{y'}^t x$ highest scoring class (ie class we are most confident it is )
       ii. if $\hat{y} \neq y$ then we have made a mistake
           A. $w_y \leftarrow w_y + x$ move wights of true class closer to input x
           B. $w_{\hat{y}} \leftarrow w_{\hat{y}} - x$ move the wrong class wights further from x.
       iii. end
   (b) end
4. end
```

- for each class we have a base linear predictor this is pretty similar to the softmax base in that it is an inner product
- this is very similar to the original perceptron algorithm
- now for each class we have a

rewriting the scoring function

- we want to be able to scale to many classes
- so if we have $W = k \times d$ wights we may have scalability issues
- we want to encode the information form W in a single weight vector $w \in \mathbb{R}^d$
- we do this by instead of having k classifiers have one classifiers with a larger input dimension
$$w_i^t x = w^t \psi(x, i)$$
$$h_i(x) = h(x, i)$$
- so here we are encoding are labels in the feature space using a feature transform $\psi : X \times I \to \mathbb{R}$
- so ψ represents what we call a compatibility score so it takes in pairs of labels and inputs and outputs how compatible that input and label are.

the multi vector construction

- so given we have a weight vector $W \in \mathbb{R}^{k\times d}$ we can flatten it into a vector $w = \begin{pmatrix} w_{1,1} & ... & w_{1,d} & w_{2,1} & ... & ...w_{k,d} \end{pmatrix} \in \mathbb{R}^{1\times(d*k)}$ that is a row vector
- and then we can define $\psi : (X, Y) \rightarrow \mathbb{R}^{1\times(D*k)}$
- then we have $< w, \psi(x, y) >=< wy, x >$
- this allows us to re-write the mutliclass perceptron as follows

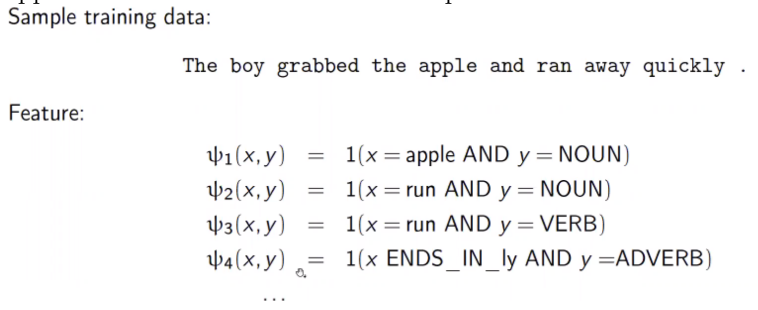
```
1. given a mutliclass dataset $\mathcal{D} = \{x, y\}$ so note that this means for
   each of the k predictors we are learning d weights so we will in total
   have a weight matrix $W \in \mathbb{R}^{k\times d}$
2. initialize $w \leftarrow 0$
3. for iter=1,2...T do
   (a) for $(x, y) \in \mathcal{D}$
       i. $\hat{y} = argmax_{y' \in y} w^t \phi(x, y')$ highest scoring class (ie class we
          are most confident it is )
       ii. if $\hat{y} \neq y$ then we have made a mistake
           A. $w_y \leftarrow w_y + \phi(x, y')$ move wights of true class closer to
              $\phi(x, y')$
           B. $w_{\hat{y}} \leftarrow w_{\hat{y}} - \phi(x, y')$ move the wrong class wights further
              from $\phi(x, y')$.
       iii. end
   (b) end
4. end
```

part of speech tagging

- input space $X = \{\text{all words}\}$
- output space $Y = \{\text{noun, verb, adj..}\}$
- features of $x \in X$: [all possible words]
- how to construct our feature vector?
- could have the multi vector construction $w \in \mathbb{R}^{d\times k}$ but this does not scale well to large feature or output spaces
- we could also directly design features for each class

$$\psi(x, y) = (\psi_1(x, y), \psi_2(x, y), ..\psi_d(x, y))$$

so that is we basically have each ψ_i looking at each element of d, which upper bounds the size of our feature space to be d



Sample training data:

```
The boy grabbed the apple and ran away quickly .
```

Feature:

$$\begin{aligned}
\psi_1(x,y) &= 1(x = \text{apple AND } y = \text{NOUN}) \\
\psi_2(x,y) &= 1(x = \text{run AND } y = \text{NOUN}) \\
\psi_3(x,y) &= 1(x = \text{run AND } y = \text{VERB}) \\
\psi_4(x,y) &= 1(x \text{ ENDS\_IN\_ly AND } y = \text{ADVERB}) \\
&\dots
\end{aligned}$$

- so here we have an example of our feature map capturing more fine detail about each part of the sentence than just using a one hot encoding that checks for the presence of every combination of words and part of speech
- so what is the captured in the feature map, and when we do the dot product what are we doing (we are taking the dot product between w, $\psi(x, y)$) representing a compatibility score.
- so after training our $w_1..w_4$ based on our feature $\psi_i(x, y)$ we want the value $w_i^t\psi_i(x, y)$ to be large when we think having that combination of word and label is correct (so that we want w_1, w_3 to be high and w_2 to be low)
- keep in mind that there is no need to keep in include feature that are not in the training data as we will have no way tot rain them.

feature templates

- maybe could take into account near by words
- could read of features from the training data meaning
- some time features could be spare
- could use a hash function template

review

- ingredients for mutliclass classification
- scoring functions for each class similar to training
- represent labels in the input space and a single weight vector
- not we want to discuss generalized SVM models for multiclass classification

4 svm

why would we like svm over perceptron

- svm allows for non-linearity
- svm prefers a large margin
- perceptron is not a maximal margin classifier it does not care as long as the classes are separated, which may not give us as much robustness

margin for multiclass

- in the binary case the margin for (x^n, y^n) is given by

$$y^n f(x^n) = y^n w^t x^n$$

- we want our margin to be large and positive ie $w^t x^n$ should have the same sign as y^n
- in the multiclass case we need a class specific margin for (x^n, y^n)

$$h(x^n, y^n) - h(x^n, y)$$

  where y^n is true class and y is the predicted label
- that is the difference between scores of teh correct class and all other classes
- want the margin to be large and postive for all $y \neq y^n$

svm separable case

- in the binary classification case our linearly separable problem is

$$min_w \frac{1}{2}||w||^2 : y^n w^t x^n \geq 1, \forall (x^i, y^y) \in \mathbb{D}$$

- in the multiclass case our linearity separable problem is we define our margn as $m_{n,y}(w) :=< w, \psi(x^n, y^n) - < w, \psi < x_n, t >$ so that is the margin of of data point n is the difference between the score of the correct class and the score of the true class

$$min_w \frac{1}{2}||w||^2 : m_{n,y} \geq 1, \forall (x^n, y^n) \in \mathbb{D}, y \neq y^n$$

hingle loss

- recall that hinge loss is $\ell_{hinge}(y, \hat{y}) = max(0, 10yh(x))$
- and that svm can basically be thought of as optimizing hinge loss
- it is a convex upper bound for zero one loss
- what's the zero one loss for mutliclass classification we can write it as

$$\Delta(y, y') = \mathbb{I}(y \neq y')$$

- so the upper bound on $\delta(y, y')$ is
$\hat{y} := argmax_{y' \in y} < e, \psi(x, y') >$
- so by that we know $< w, \psi(x, y) >\leq< w, \psi(w, \hat{y}) >$ and as y is the true class label we know $0 \leq< w, \psi(x, y) >\leq< w, \psi(w, \hat{y}) >$
- this means we can write $\Delta(y, \hat{y}) \neq \Delta(y, \hat{y})- < w, (\psi(x, y) - \psi(x, \hat{y})) >$
- so we can define our geneal hinge loss as

$$\ell_{hinge}(y, x, w) := max_{y' \in y}(\Delta(y, y')- < w, (\psi(w, y) - \psi(x, y') >)$$

where y' is our predicted class and y is the true class. so we are looking for the max violation effectively

- recall in binary svm we can write the hinge loss formulation as

$$min_{w \in \mathbb{R}^d} \frac{1}{2}||w||^2 + C\Sigma_{i=1}^{n} max(0, 1 - y^n w^t x^n)$$

- the multiclass objective

$$min_{w \in \mathbb{R}^d} \frac{1}{2}||w||^2 + C\Sigma n = 1^N max_{y \in y'}(\Delta(y, y')- < x, (\psi(x, y) - \psi(x, y')) >)$$

- so $\Delta(y, y')$ is the target margin for each class
- if the margin meets or exceeds $\Delta(y^n, y')\forall y \in Y$ then there is no on example n, that is there is no loss if the margin of all other classes is bellow 1 distance from the true class score than there is no loss

does one versus all work in practice

- yeah one versus all works pretty well n practice
- but there are a lot of mutliclass frameworks that may be better
- all of the mutliclass frameworks preformed roughly the same

mutliclass svm recap

- the frameworks we have developed for multiclass keeps track of compatibility features, scoring functions, multiclass margins and target margins
- this generalized well where k is large and only versus large is intractable.
- the key idea is that we can generalize across any output y by using features transforms $\psi(x, y)$

5 introduction to structured prediction

example part of speech tagging

- structured prediction is a big field but this is a small introduction
- given a sentence we may want to tag the part of speech of each word
- so call $V = \{\text{all english words}\}$
- call our input space $X = V^n$ where n is any length so that is any sequence of words
- $P = \{\text{part of speech tags}\}$
- $T = P^n$

multiclass hypothesis space

- so we assume that there is a discrete output space y(x), that is very large but has some structure like for instance a noun may be likely to be followed by a very or adjective but not another noun
- our base hypothesis space (meaning the hypothesis space for a single predictor is) $H = \{h : X \times Y \to \mathbb{R}\}$ that is functions that map from an input and potential label h(x,y) to a compatibility score
- mutliclass hypothesis space

$$\mathcal{F} = \{x \to argmax_{y \in y} h(x, y) h \in H\}$$

- our finial prediction function is $f \in \mathcal{F}$ where y is the argmax of all the compatibility scores
- so for each of the f in the hypothesis space there is an underlying compatibility score function

unary features

- a unary feature only depends on the label at a single position y_i and x
- for example $\phi_1(x, y_i) = 1(x_i = \text{runs})1(y_i = \text{verb})$ or $\phi_2(x, y_i) = 1(x_i = \text{runs})1(y_i = \text{noun})$
- we can define features that care only about x and y
- note it can only care about one label but can take into account multiple inputs for example $\phi_3(x, y_i) = 1(x_{i=1} = \text{he})1(x_i = \text{runs})1(y_i = \text{verb})$

markov features

- a markov or pairwise feature only depends on two adjective labels y_i, y_{i-1} and x
- for example $\theta_1(x, y_{i-1}, y_i) = 1(y_{i-1} = \text{pronoun})1(y_i = \text{verb})$
- it is possible to expand this to higher order features but that causes exponential dependencies

local feature vectors and compatibility scores

- at each position i in a sequence define the local feature vector (comprised of unary feature ϕ_i and markov features θ_i) as

$$\psi_i(x, y_{i-1}, y_i) = \begin{pmatrix} \phi_1(x, y_i), \phi_2(x, y_i) \\ \psi_1(x, y_{i-1}, y_i), \psi_2(x, y_{i-1}, y_i) \end{pmatrix}$$

- and the local compatibility score at position i is $< w, \psi_i(x, y_{i-1}, y_i) >$
- the compatibility score for (x,y) is the sum of local compatibility scores $\Sigma_i < w, \psi_i(x, y_{i-1}, y_i) >=< w, \Sigma_i \psi_i(x, y_{i-1}, y_i) >=< w, \psi(x, y) >$
- where we define the sequence feature vector by

$$\psi(x, y) = \Sigma_i \psi_i(x, y_{i-1}, y_i)$$

structured perceptron

1. given a mutliclass dataset $\mathcal{D} = \{x, y\}$ so note that this means for each of the k predictors we are learning d weights so we will in total have a weight matrix $W \in \mathbb{R}^{k \times d}$
2. initialize $w \leftarrow 0$
3. for iter=1,2...T do

```
    (a) for $(x, y) \in \mathcal{D}$
        i. $\hat{y} = argmax_{y' \in y} w^t \psi(x, y')$ highest scoring class (ie class we are most confident it is )
        ii. if $\hat{y} \neq y$ then we have made a mistake
            A. $w_y \leftarrow w_y + \psi(x, y')$ move wights of true class closer to $\psi(x, y')$
            B. $w_{\hat{y}} \leftarrow w_{\hat{y}} - \psi(x, y')$ move the wrong class wights further from $\psi(x, y')$.
        iii. end
    (b) end
4. end
```

- so this identical to the mutliclass case except we are using a different feature map

structured hinge loss

- recall that our geneal hinge loss defention is $\ell_{hinge}(y, \hat{y}) = max_{y' \in y(x)}(\Delta(y, y') + < w, (\psi(x, y') - \psi(x, y)) >)$
- but structured prediction is about sequences so what is the hamming loss between two sequences?
- the hamming loss is given by

$$\Delta(y, y') = \frac{1}{L} 1(y_i \neq y_i^{'})$$

  where l is the length of the sequence

arg max for sequences

- the argmax problem can be difficult for sequences
- to compute predictions we need to find $argmax_{y \in y(x)} < w, \psi(x, y) >$ where $|y(x)|$ is exponentially large
- observation $\psi(x, y)$ decomposes into $\Sigma_i \psi_i(x, y)$
- the solution to this is dynamic programming
- at every time step we have different words each with a potential pART of speech label label, compute the score up tell time step t and then consider the max possible score from all the previous posabilties

- so in the first time step we consider the score at each nodes in teh second time step we consider all 3 possible parents and take the max of those, do this for all the nodes at each step look at all the parents and t teh end you find the max score achievable for the final node.
- need to know what each node's parent was
- called the viterbi algorithm
- there are t time steps at each time step you look at all y nodes, and y possible parents so total run time is $O(ty^2)$
- this is much more efficient than just doing the naive search

conditional random fields

- recall that we can write logistic regression in the general form $P(y|x) = \frac{1}{z(x)}e^{w^t\psi(x,y)}$
- where z is a normalization constant $z(x) = \Sigma_{y\in Y}e^{w^t\psi(x,y)}$
- this is a probabilistic interpretation of the crf so this will output a probability distribution we can sample from
- we can incorporate uniformly and markov features to this $P(y|x) = \frac{1}{z(x)}e^{\Sigma_t w^t\psi(x,y_t,y_{t-1})}$
- here we can learn our weights by minimizing the regularized negative log likelihood $l(w) = \frac{-1}{N}\Sigma_{i-1}^{n}log(p(y^i|x^i))+\frac{1}{2}\lambda||w||^2 = \frac{-1}{N}\Sigma_i\Sigma_t\Sigma_k w_k\phi_k(y_t^i, y_{t-1}^i)+ \frac{1}{n}\Sigma_i Z(X_i) + \frac{1}{2}\Sigma_k\lambda w_k^2$
- we deal with taking the derivative i am going to be honest i straight to not have the life force to that right now
- the gradient update step ends up being the difference between the empirical update of your feture and expceation udner the model distribution
- in CRF we need to compute the expectation under the model distribution $P(y|x)$
- this as well as doing the arg max in the structured svm problem are both np hard problems

crf inference

- we kind of do a similar thing to the verterbi algorithm but for the expectation
- there are forward and backwards way to do this

- at test time we can again use viterbi to infer argmax
- this can be called belief propagation
- there are a lot of potential examples where crf is a good idea like pos tagging or image segmentation