Homework

- WSD results on development set:

always assign `01`	88.40%
most frequent sense	92.61%
most frequent possible sense	92.73%

```
bulba% paste out props.dev | distr feel
{'01': 11}
bulba% paste senses.dev props.dev | distr feel
{'02': 3, 'XX': 1, '01': 7}
```

Homework

[A1 A&W Brands] [V lost] [A2 14] to [A4 27] .

```
<roleset id="lose.01" name="decrease, fall">
<roles>
  <role descr="logical subject, patient, thing falling" n="1"/>
  <role descr="EXT, amount fallen" n="2"/>
  <role descr="start point" n="3"/>
  <role descr="end point" n="4"/>
  <role descr="medium" f="LOC" n="M"/>
</roles>
</roleset>

<roleset id="lose.02" name="lose, no longer have">
<roles>
  <role descr="entity losing something" n="0"/>
  <role descr="thing lost" n="1"/>
  <role descr="benefactive, entity gaining thing lost" n="2"/>
</roles>
</roleset>
```

Homework

- Next steps for project
- Cascade approach
  - ⋆ find argument boundaries
  - ⋆ label arguments with roles
- Presentations in two weeks

Support Vector Machines

- *Empirical Risk Minimization* finds a function $f \in \mathcal{F}$ which minimizes the training error
- *Structural Risk Minimization* finds a function $f \in \mathcal{F}$ which minimizes a bound on the generalization error (which depends on the training error and model capacity)
- For a separating hyperplane, increasing the margin decreases the capacity
- This, plus SRM, yields the *optimal margin classifier* (Vapnik 1982): find the canonical hyperplane which separates the training data and maximizes the margin

Support Vector Machines

- Finding the optimal separating hyperplane is a quadratic programming problem (optimizing a quadratic function with linear constraints on the variables)
- Similar problems come up in operations research and finance, and methods for solving quadratic programs are well established
- In case the training data is not separable (or even if it is), we can use a *soft margin* algorithm which allows a certain amount of ‘slack’
- This is a big improvement over the perceptron, but is still limited to linear decision boundaries

Support Vector Machines

- This optimization problem can be stated in either a *primal* form (in terms of the decision boundary) or a *dual* form (in terms of the training examples)

- The dual form has several advantages:
  - ⋆ it's easier to solve than the primal form
  - ⋆ only the training examples right on the margin or inside it (the *support vectors*) are needed to represent the boundary
  - ⋆ the dot product can be replaced by an arbitrary *kernel function,* allowing certain kinds of non-linearities to be captured

- From this, we get Support Vector Machines, a leading contender for best all-around learning algorithm

Support Vector Machines

- SVMs are much harder to understand and to implement than other learning methods
- But, the ‘leading ideas’ aren’t so bad, and there are a number of high quality implementations around for us to use
- Besides the obvious, work on SVMs has had two significant contributions to machine learning in general:
  - ⋆ validated results of statistical learning theory (especially SRM)
  - ⋆ showed the benefit of using simple learners + kernel functions to learn complex concepts
- demo, demo, demo

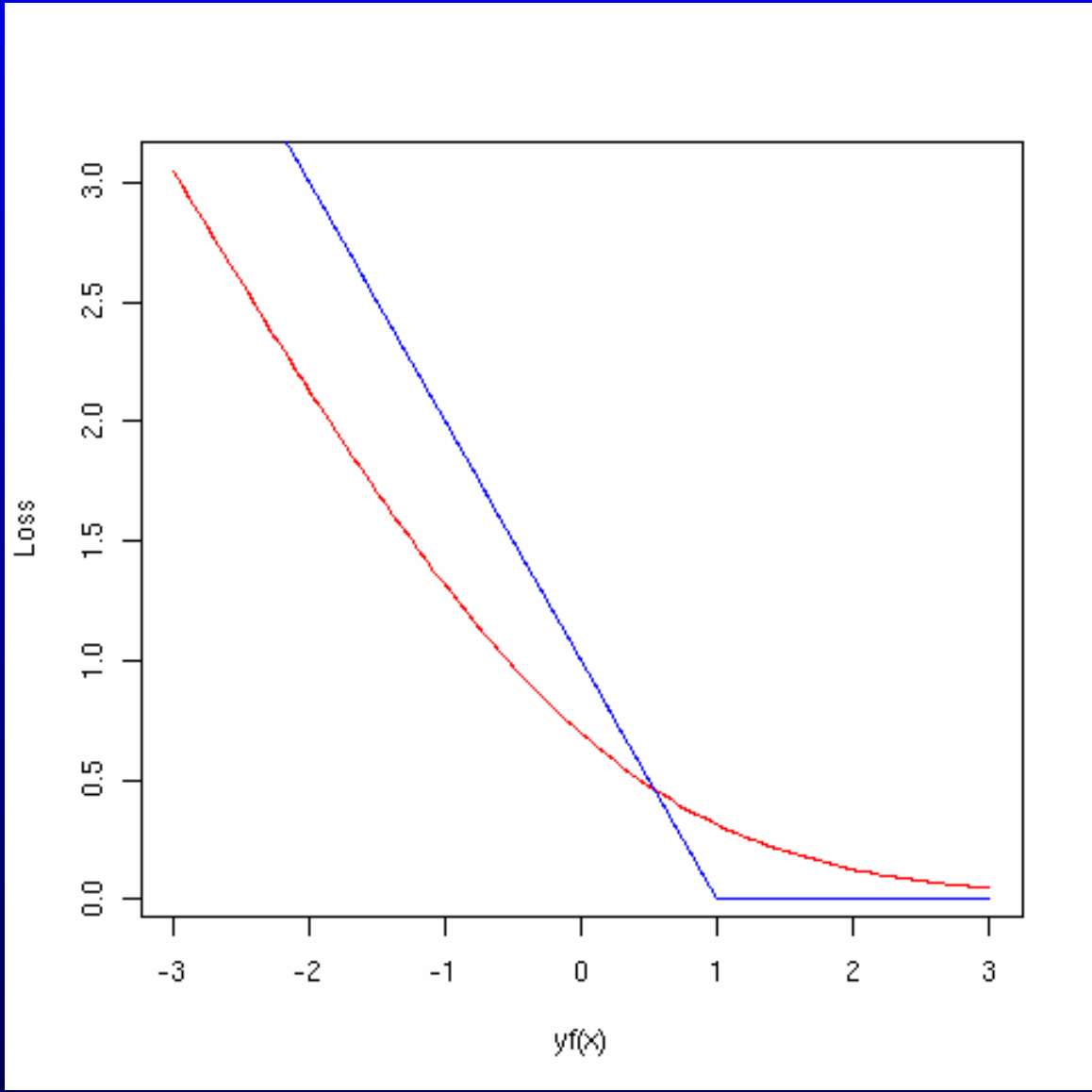
Loss functions

- What are SVMs really doing?
- Recall that boosting turned out to be an algorithm to minimize the exponential loss
- This is similar to maximum entropy methods, which minimize the negative logistic log-likelihood
- SVMs can also be fit into the same framework. They find the solution to:

$$\min_{\beta} \sum_{i} [1 - y_i \, f(x_i; \beta)]_+ + \lambda \, ||\beta||^2$$

- This is the *hinge loss* $[1 - y_i \, f(x_i; \beta)]_+$ plus a quadratic *penalty* or *regularization* term

Support vector machines



Loss functions

- The hinge loss is an upper bound on the *zero-one loss,* a natural for classification, and the logistic log-likelihood of MaxEnt models is very similar
- This gives us another perspective on Gaussian prior smoothing
- This also gives a unified framework for linking SVMs, MaxEnt, boosting, and many other types of models:

$$\min_{\beta} L(y, f) + \lambda J(\beta)$$

- New methods can be developed by tinkering with the loss and penalty functions

Implementation

- While we know how to solve quadratic programs in general, SVMs are particularly challenging
- Many generic QP codes need the entire $n \times n$ Gram matrix
- Others return extremely small values instead of zeros ($10^{-17} \approx 0$)
- Most training points are irrelevant, so we can use active set methods
- Memoization can speed up calculation of $K(x_i, x_j)$
- *Chunking* starts with a small training set, and gradually adds items that get misclassified

Implementation

- Joachims' (1997) *SVM*^{light} decomposes the problem into smaller, simpler problems, and eliminates examples which are unlikely to be support vectors early on
- Platt's (1998) Sequential Minimal Optimization (SMO) decomposes the problem into subproblems that are small enough to solve analytically
- Large scale problems can be approached by constructing a low-rank approximation to the Gram matrix
- Computational cost is still a problem, but becoming less and less so for medium-sized problems (10,000 training examples/100 features)

Kernel functions

- Much of the power of SVMs derives from the kernel trick
- Designing an appropriate kernel can make a huge difference (there's No Free Lunch!)
- Linear, polynomial, and RBF kernels are a good place to start
- If K_1 and K_2 are kernels, and $\alpha_1, \alpha_2 \geq 0$, then:

$$\begin{aligned} K(x_i, x_j) &= \alpha_1 K_1(x_i, x_j) + \alpha_2 K_2(x_i, x_j) \\ K(x_i, x_j) &= K_1(x_i, x_j) K_2(x_i, x_j) \end{aligned}$$

  are also kernels

String kernels

- Linear kernels can be computed efficiently for sparse bag-of-words models
- If $f(w, x)$ is the freqeuency of for w in document x, then computing the dot product:

$$K(x_i, x_j) = \sum_w f(w, x_i)\, f(w, x_j)$$

  has cost that depends on the length of the documents, *not* the size of the vocabulary

- But, why use a bag of words?
  - ⋆ efficiency
  - ⋆ independence

String kernels

- The bag of words model misses out on a lot
- It depends on having a complete language/domain dependent vocabulary
- It can’t represent a partial match between related (but not identical words)
- It completely ignores multi-word units and syntactic relations
- Really, its only strength is that it’s easy to use

String kernels

- A linear kernel is comparable to a mapping like:

	c	a	t	r	b
$\phi(\text{cat})$	1	1	1	0	0
$\phi(\text{car})$	1	1	0	1	0
$\phi(\text{bat})$	0	1	1	0	1
$\phi(\text{bar})$	0	1	0	1	1

- Similarity between words depends only on the number of letters they have in common, not their order or proximity

String kernels

- An alternative would represent a word as a set of possibly discontinuous n grams
- The trigram **c-r-d** characterizes the words **car**d and **c**ust**ar**d, but the former more than the latter
- We use a decay factor λ ($0 < \lambda < 1$), so that if a match that spans n characters, it gets a weight of λ^n
- Using bigrams, this mapping would give us:

	c-a	c-t	a-t	b-a	b-t	c-r	a-r	b-r
ϕ(cat)	λ^2	λ^3	λ^2	0	0	0	0	0
ϕ(car)	λ^2	0	0	0	0	λ^3	λ^2	0
ϕ(bat)	0	0	λ^2	λ^2	λ^3	0	0	0
ϕ(bar)	0	0	0	λ^2	0	0	λ^2	λ^3

String kernels

- The value of the kernel function is the dot product of the feature vectors:

$$\begin{aligned} K(\text{car}, \text{cat}) &= \phi(\text{car}) \cdot \phi(\text{cat}) \\ &= \langle \lambda^2, \lambda^3, \lambda^2, 0, 0, 0, 0, 0 \rangle \cdot \langle \lambda^2, 0, 0, 0, 0, \lambda^3, \lambda^2, 0 \rangle \\ &= \lambda^4 \end{aligned}$$

- The normalized distance between the words is:

$$\begin{aligned} \hat{K}(\text{car}, \text{cat}) &= \frac{K(\text{car}, \text{cat})}{\sqrt{K(\text{car}, \text{car})\, K(\text{cat}, \text{cat})}} \\ &= \frac{\lambda^4}{2\,\lambda^4 + \lambda^6} \\ &= \frac{1}{2 + \lambda^2} \end{aligned}$$

String kernels

- This *string subsequence kernel* (SSK) can be extended to entire documents
- For interesting subsequence sizes and document lengths, explicit computation of all of the features would be impractical
- A very similar problem arises in bioinformatics: comparing DNA sequences
- We can use dynamic programming to efficiently evaluate the kernel function without constructing the feature vectors
- The can also be computed using *suffix trees,* a compact representation of the substrings in a text¡

String kernels

- Let Σ^n be the set of all strings of length n. To construct the feature mapping ϕ for a string s, we define the u coordinate for each $u \in \Sigma^n$:

$$\phi_u(s) = \sum_{\mathbf{i}:u=s[\mathbf{i}]} \lambda^{l(\mathbf{i})}$$

- The kernel function is given by:

$$\begin{aligned} K_n(s,t) &= \sum_{u \in \Sigma^n} \phi_u(s) \cdot \phi_u(t) \\ &= \sum_{u \in \Sigma^n} \sum_{\mathbf{i}:u=s[\mathbf{i}]} \sum_{\mathbf{j}:u=t[\mathbf{j}]} \lambda^{l(\mathbf{i})+l(\mathbf{j})} \end{aligned}$$

String kernels

- Lohdi et al. (2002) compare SSK to standard word kernel (WK) and n-gram kernels (NGK) for text classification
- NGK and SSK show very similar performance overall
- Best results for $n = 3$ or $n = 4$, but higher than that reduces performance
- Increasing λ for SSK increases precision but decreases recall
- Summing SSKs with multiples values of n can improve things very slightly
- As the amount of training data increases, benefits of SSK are reduced