Lecture topics: model selection criteria

- Feature subset selection (cont'd)
  - information criterion
  - conditional log-likelihood and description length
  - regularization
- Combination of classifiers, boosting

Feature subset selection (cont'd)

We have already discussed how to carry out feature selection with the Naive Bayes model. This model is easy to specify and estimate when the input vectors $\mathbf{x}$ are discrete (here vectors with binary $\{-1, 1\}$ components). When we elect to use only a subset $\mathcal{J}$ of features for classification, our model can be written as

$$P(\mathbf{x}, y) = \left[\prod_{i \in J} P(x_i|y)\right] \left[\prod_{i \notin J} P(x_i)\right] P(y) \tag{1}$$

Note that the features not used for classification, indexed by $j \notin \mathcal{J}$, are assumed independent of the label. This way we still have a distribution over the original input vectors $\mathbf{x}$ and labels but assert that only some of the components of $\mathbf{x}$ are relevant for classification. The probabilities involved such as $P(x_i|y)$ are obtained directly from empirical counts involving x_i and y (see previous lecture).

The selection criterion we arrived at indicated that, as far as the log-likelihood of all the data is concerned, we should include features (replace $P(x_i)$ with $P(x_i|y)$ in the above model) in the decreasing order of

$$\hat{I}(X_i; Y) = \hat{H}(X_i) - \hat{H}(X_i|Y) \tag{2}$$

where the entropies $\hat{H}(X_i)$ and $\hat{H}(X_i|Y)$ are evaluated on the basis of the estimated probabilities for the Naive Bayes model. The disadvantage of this criterion is that the features are selected individually, i.e., without regard to how effective they might be in specific combinations with each other. This is a direct result of the Naive Bayes model as well as the fact that we opted to find features that maximally help increase the log-likelihood of all the data. This is clearly slightly different from trying to classify examples accurately.

For example, from the point of view of classification, it is not necessary to model the distribution over the feature vectors $\mathbf{x}$. All we care about is the conditional probability $P(y|\mathbf{x})$. Why should our estimation and the feature selection criterion then aim to increase the log-likelihood of generating the feature vectors as well? We will now turn to improving the feature selection part with MDL while still estimating the probabilities $P(x_i|y)$ and $P(y)$ in closed form as before.

Feature subset selection with MDL

Let's assume we have estimated the parameters in the Naive Bayes model, i.e., $\hat{P}(\mathbf{x}_i|y)$ and $\hat{P}(y)$, as before by maximizing the log-likelihood of all the data. Let's see what this model implies in terms of classification:

$$\hat{P}(y=1|\mathbf{x}) = \frac{\hat{P}(\mathbf{x}, y=1)}{\hat{P}(\mathbf{x}, y=1) + \hat{P}(\mathbf{x}, y=-1)} = \frac{1}{1 + \frac{\hat{P}(\mathbf{x}, y=-1)}{\hat{P}(\mathbf{x}, y=1)}} \quad (3)$$

$$= \frac{1}{1 + \exp(-\log \frac{\hat{P}(\mathbf{x}, y=1)}{\hat{P}(\mathbf{x}, y=-1)})} = \frac{1}{1 + \exp(-\hat{f}(\mathbf{x}))} \quad (4)$$

where

$$\hat{f}(\mathbf{x}) = \log \frac{\hat{P}(\mathbf{x}, y=1)}{\hat{P}(\mathbf{x}, y=-1)} \quad (5)$$

is the discriminant function arising from the Naive Bayes model. So, for example, when $\hat{f}(\mathbf{x}) > 0$, the logistic function implies that $\hat{P}(y=1|\mathbf{x}) > 0.5$ and we would classify the example as positive. If $\hat{f}(\mathbf{x})$ is a linear function of the inputs $\mathbf{x}$, then the form of the conditional probability from the Naive Bayes model would be exactly as in a logistic regression model. Let's see if this is indeed so.

$$\hat{f}(\mathbf{x}) = \log \frac{\hat{P}(\mathbf{x}, y=1)}{\hat{P}(\mathbf{x}, y=-1)} = \sum_{i \in \mathcal{J}} \log \frac{\hat{P}(x_i|y=1)}{\hat{P}(x_i|y=-1)} + \log \frac{\hat{P}(y=1)}{\hat{P}(y=-1)} \quad (6)$$

Note that only terms tried to the labels remain. Is this a linear function of the binary features x_i? Yes, it is. We can write each term as a linear function of x_i as follows:

$$\log \frac{\hat{P}(x_i|y=1)}{\hat{P}(x_i|y=-1)} = \delta(x_i, 1) \log \frac{\hat{P}(x_i=1|y=1)}{\hat{P}(x_i=1|y=-1)} + \delta(x_i, -1) \log \frac{\hat{P}(x_i=-1|y=1)}{\hat{P}(x_i=-1|y=-1)} \quad (7)$$

$$= \frac{x_i+1}{2} \log \frac{\hat{P}(x_i=1|y=1)}{\hat{P}(x_i=1|y=-1)} + \frac{1-x_i}{2} \log \frac{\hat{P}(x_i=-1|y=1)}{\hat{P}(x_i=-1|y=-1)} \quad (8)$$

By combining these terms we can write the discriminant function in the usual linear form

$$\hat{f}(\mathbf{x}) = \sum_{i \in \mathcal{J}} \hat{w}_i x_i + \hat{w}_0 \tag{9}$$

where the parameters $\hat{w}_i$ and $\hat{w}_0$ are functions of the Naive Bayes conditional probabilities. Note that while we ended up with a logistic regression model, the parameters of this conditional distribution, $\hat{w}_i$ and $\hat{w}_0$, have been estimated quite differently from the logistic model.

We can now evaluate the conditional probability $\hat{P}(y|\mathbf{x}, \mathcal{J})$ from the Naive Bayes model corresponding to any subset $\mathcal{J}$ of relevant features. Let's go back to the feature selection problem. The subset $\mathcal{J}$ should be optimized to maximize the conditional log-likelihood of labels given examples. Equivalently, we can *minimize* the description length

$$\text{DL-data}(\mathcal{J}) = \sum_{t=1}^{n} -\log \hat{P}(y_t|\mathbf{x}_t, \mathcal{J}) \tag{10}$$

with respect to $\mathcal{J}$. This is a criterion that evaluates how useful the features are for classification and it can be no longer reduced to evaluating features independently of others. This also means that the optimization problem for finding $\mathcal{J}$ is a difficult one. The simplest way of approximately minimizing this criterion would be to start with no features, then include the single best feature, followed by the second feature that works best with the first one, and so on. Note that in this simple setting the classifier parameters associated with different subsets of features are fixed by the Naive Bayes model; we only optimize over the subset of features.

The above sequential optimization of the criterion would yield features that are more useful for classification than ranking them by mutual information (the first feature to include would be the same, however). But we do not yet have a criterion for deciding how many features to include. In the MDL terminology, we need to describe the model as well. The more features we include the more bits we need to describe both the set and the parameters associated with using that set (the Naive Bayes conditional probabilities).

So, first we need to communicate the set (size and the elements). The (any) integer $|\mathcal{J}|$ can be communicated with the cost of

$$\log^* |\mathcal{J}| = \log |\mathcal{J}| + \log \log |\mathcal{J}| + \ldots \tag{11}$$

nats. The elements in the set, assuming (a priori) that they are drawn uniformly at random from d possible features, require

$$\log \binom{d}{|\mathcal{J}|} \tag{12}$$

nats. Finally, we need to communicate the Naive Bayes parameters associated with the elements in the feature set. $\hat{P}(y)$ can only take $n+1$ possible values and thus requires $\log(n+1)$ nats. Once we have sent this distribution, the receiver will have access to the counts $n_y(1)$ and $n_y(-1)$ in addition to n that they already knew. $\hat{P}(x_i|y=1)$ can only take $n_y(1)+1$ possible values with the cost of $\log(n_y(1)+1)$, and similarly for $\hat{P}(x_i|y=-1)$. Given that there are $|\mathcal{J}|$ of these, the total communication cost for the model is then

$$\begin{aligned} \text{DL-model}(|\mathcal{J}|) &= \log^* |\mathcal{J}| + \log \binom{d}{|\mathcal{J}|} + \qquad (13) \\ &\quad \log(n+1) + |\mathcal{J}| \Big(\log(n_y(1)+1) + \log(n_y(-1)+1) \Big) \qquad (14) \end{aligned}$$

Note that the model cost only depends on the size of the feature set, not the elements in the set. We would find the subset of features by minimizing

$$\text{DL}(\mathcal{J}) = \text{DL-data}(\mathcal{J}) + \text{DL-model}(|\mathcal{J}|) \qquad (15)$$

sequentially or otherwise.

Feature selection via regularization

An alternative to explicitly searching for the right feature subset is to try to formulate the selection problem as a regularization problem. We will no longer use parameters from a Naive Bayes model but instead work directly with a logistic regression model

$$P(y|\mathbf{x},\theta) = g\Big(\sum_{i=1}^{d} \theta_i x_i + \theta_0 \Big) \qquad (16)$$

The typical regularization problem for estimating the parameters θ would be penalized conditional log-likelihood with the squared norm regularization

$$J(\theta; S_n) = \sum_{t=1}^{n} \log P(y_t|\mathbf{x}_t, \theta) - \lambda \sum_{i=1}^{d} \theta_i^2 \qquad (17)$$

The squared norm regularization won't work for our purposes, however. The reason is that none of the parameters would be set exactly to zero as a result of solving the optimization problem (none of the features would be clearly selected or excluded). We will instead use a *one-norm* regularization penalty:

$$J(\theta; S_n) = \overbrace{\sum_{t=1}^{n} \log P(y_t|\mathbf{x}_t, \theta)}^{l(\theta;S_n)} - \lambda \sum_{i=1}^{d} |\theta_i| \tag{18}$$

If we let $\lambda \rightarrow \infty$, then all θ_i (except for θ_0) would go to zero. On the other hand, when $\lambda = 0$, none of θ_i are zero. These are identical to the squared penalty. However, the intermediate values of λ produce more interesting results: some of θ_i will be set exactly to zero while this would never happen with the squared penalty. Let's try to understand why.

Suppose we fix all but a single parameter θ_k to their optimized values (zero or not) and view $l(\theta; S_n)$ as a function of this parameter. Suppose the optimal value of $\theta_k > 0$, then

$$\frac{d}{d\theta_k} l(\theta; S_n) - \lambda = 0 \tag{19}$$

at the optimal θ_k. In other words, the slope of the conditional log-likelihood function relative to this parameter has to be exactly λ. If parameter θ_k is irrelevant then the conditional log-likelihood is unlikely to be affected much by the parameter, keeping the slope very small. In this case, θ_k would be set exactly to zero. The larger the value of λ, the more of the parameters would end up going to zero. Another nice property is that if the optimum value of θ_k is zero, then we would get the same regression model whether or not the corresponding feature were included in the model to begin with. A good value of λ, and therefore a good number of features, can be found via cross-validation.

So why don't we get the same "zeroing" of the parameters with the squared penalty? Consider the same situation, fix all but θ_k. Then at the optimum

$$\frac{d}{d\theta_k} l(\theta; S_n) - 2\lambda\theta_k = 0 \tag{20}$$

However small the slope of the log-likelihood is at $\theta_k = 0$, we can move the parameter just above zero so as to satisfy the above optimality condition. There's no fixed lower bound on how useful the parameter has to be for it to take non-zero values.

Combining classifiers and boosting

We have so far discussed feature selection as a problem of finding a subset of features out of d possible ones. In many cases the possible feature set available to us may not even be enumerable. As an example, consider real valued input vectors $\mathbf{x} = [x_1, \ldots, x_d]^d$, still $d-$dimensional. We can turn this real valued vector into a feature vector of binary $\{-1, 1\}$ features in a number of different ways. One possibility is to run simple classifiers on the input vector and collect the predicted class labels into a feature vector. One of the simplest classifiers is the *decision stump*:

$$h(\mathbf{x}; \theta) = \text{sign}\Big(s(x_k - \theta_0) \Big) \tag{21}$$

where $\theta = \{s, k, \theta_0\}$ and $s \in \{-1, 1\}$ specifies the label to predict on the positive side of $x_k - \theta_0$. In other words, we simply select a specific component and threshold its value. There are still an uncountable number of such classifiers, even based on the same input component since the threshold θ_0 is real valued. Our "feature selection" problem here is to find a finite set of such stumps.

In order to determine which stumps ("features") would be useful we have to decide how they will be exploited. There are many ways to do this. For example, we could run a linear classifier based on the resulting binary feature vectors

$$\phi(\mathbf{x}; \theta) = [h(\mathbf{x}; \theta_1), \ldots, h(\mathbf{x}; \theta_m)]^T \tag{22}$$

We will instead collect the outputs of the simple stumps into an *ensemble*:

$$h_m(\mathbf{x}) = \sum_{j=1}^{m} \alpha_j h(\mathbf{x}; \theta_j) \tag{23}$$

where $\alpha_j \geq 0$ and $\sum_{j=1}^{m} \alpha_j = 1$. We can view the ensemble as a voting combination. Given $\mathbf{x}$, each stump votes for a label and it has α_j votes. The ensemble then classifies the example according to which label received the most votes. Note that $h_m(\mathbf{x}) \in [-1, 1]$. $h_m(\mathbf{x}) = 1$ only if all the stumps agree that the label should be $y = 1$. The ensemble is a linear classifier based on $\phi(\mathbf{x}; \theta)$, for a fixed θ, but with constrained parameters. However, our goal to learn both which features to include, i.e., $h(\mathbf{x}; \theta_j)$'s, as well as how they are combined (the α's). This is a difficult problem solve.

We can combine any set of classifiers into an ensemble, not just stumps. For this reason, we will refer to the simple classifiers we are combining as *base learners* or *base classifiers* (also called weak learners or component classifiers). Note that the process of combining simple "weak" classifiers into one "strong" classifier is analogous to the use of kernels to go from a simple linear classifier to a non-linear classifier. The difference is that here we are *learning* a small number of highly *non-linear features* from the inputs rather than using a fixed process of generating a large number of features from the inputs as in the polynomial kernel.

The ensembles can be useful even if generated through randomization. For example, we can generate random subsets of smaller training sets from the original one, and train a classifier, e.g., an SVM, based on each such set. The outputs of the SVM classifiers, trained with slightly different training sets, can be combined into an ensemble with uniform α's. This process, known as *bagging*, is a method of reducing the *variance* of the resulting classifier. The uniform weighting will not help with the *bias*.

Now, let's figure out how to train ensembles. We will need a loss function and there many possibilities, including the logistic loss. For simplicity, we will use the *exponential loss*:

$$\text{Loss}(y, h(\mathbf{x})) = \exp(-yh(\mathbf{x})) \tag{24}$$

The loss is small if the ensemble classifier agrees with the label y (the smaller the stronger the agreement). It is large if the ensemble strongly disagrees with the label. This is the basic loss function is in an ensemble method called *Boosting*.

The simplest way to optimize the ensemble is to do it in stages. In other words, we will first find a single stump (an ensemble of one), then add another while keeping the first one fixed, and so on, never retraining those already added into the ensemble. To facilitate this type of estimation, we will assume that $\alpha_j \geq 0$ but won't require that they will sum to one (we can always renormalize the votes after having trained the ensemble).

Suppose now that we have already added $m-1$ base learners into the ensemble and call this ensemble $h_{m-1}(\mathbf{x})$. This part will be fixed for the purpose of adding $\alpha_m h(\mathbf{x};\theta_m)$. We can then try to minimize the training loss corresponding to the ensemble

$$\begin{aligned} h_m(\mathbf{x}) &= \sum_{j=1}^{m-1} \hat{\alpha}_j h(\mathbf{x};\hat{\theta}_j) + \alpha_m h(\mathbf{x};\theta_m) && (25) \\ &= h_{m-1}(\mathbf{x}) + \alpha_m h(\mathbf{x};\theta_m) && (26) \end{aligned}$$

To this end,

$$\begin{aligned}
J(\alpha_m, \theta_m) &= \sum_{t=1}^{n} \text{Loss}(y_t, h_m(\mathbf{x}_t)) && (27)\\
&= \sum_{t=1}^{n} \exp\Big(- y_t h_{m-1}(\mathbf{x}_t) - y_t \alpha_m h(\mathbf{x}_t; \theta_m)\Big) && (28)\\
&= \sum_{t=1}^{n} \overbrace{\exp\Big(- y_t h_{m-1}(\mathbf{x}_t)\Big)}^{W_{m-1}(t)} \exp\Big(- y_t \alpha_m h(\mathbf{x}_t; \theta_m)\Big) && (29)\\
&= \sum_{t=1}^{n} W_{m-1}(t) \exp\Big(- y_t \alpha_m h(\mathbf{x}_t; \theta_m)\Big) && (30)
\end{aligned}$$

In other words, for the purpose of estimating the new base learner, all we need to know from the previous ensemble are the weights $W_{m-1}(t)$ associated with the training examples. These weights are exactly the losses of the $m-1$ ensemble on each of the training example. Thus, the new base learner will be "directed" towards examples that were misclassified by the $m-1$ ensemble $h_{m-1}(\mathbf{x})$.

The estimation problem that couples α_m and θ_m is still a bit difficult. We will simplify this further by figuring out how to estimate θ_m first and then decide the votes α_m that we should assign to the new base learner (i.e., how much we should rely on its predictions). But what is the criterion for θ_m independent of α_m? Consider as a thought experiment that we calculate for all possible θ_m the derivative

$$\frac{d}{d\alpha_m} J(\alpha_m, \theta_m)\Big|_{\alpha_m = 0} = -\sum_{t=1}^{m} W_{m-1}(t) y_t h(\mathbf{x}_t; \theta_m) \qquad (31)$$

This derivative tells us how much we can reduce the overall loss by increasing the vote (from zero) of the new base learner with parameters θ_m. This derivative is expected to be negative so that the training loss is decreased by adding the new base learner. It makes sense then to find a base learner $h(\mathbf{x}; \theta_m)$, parameters θ_m, so as to minimize this derivative (making it as negative as possible). Such a base learner would be expected to lead to a large reduction of the training loss. Once we have this $\hat{\theta}_m$ we can subsequently optimize the training loss $J(\alpha_m, \hat{\theta}_m)$ with respect to α_m for a fixed $\hat{\theta}_m$.

We have now essentially all the components to define the *Adaboost algorithm*. We will make one modification which is that the weights will be normalized to sum to one. This is advantageous as they can become rather small in the course of adding the base learners. The normalization won't change the optimization of θ_m nor α_m. We denote the normalized weights with $\tilde{W}_{m-1}(t)$. The boosting algorithm is defined as

(0) Set $W_0(t) = 1/n$ for $t = 1, \ldots, n$.

(1) At stage m, find a base learner $h(\mathbf{x}; \hat{\theta}_m)$ that approximately minimizes

$$-\sum_{t=1}^{m} \tilde{W}_{m-1}(t) y_t h(\mathbf{x}_t; \theta_m) = 2\epsilon_m - 1 \tag{32}$$

where ϵ_m is the weighted classification error on the training examples, weighted by the normalized weights $\tilde{W}_{m-1}(t)$.

(2) Set

$$\hat{\alpha}_m = 0.5 \log\left(\frac{1 - \hat{\epsilon}_m}{\hat{\epsilon}_m}\right) \tag{33}$$

where $\hat{\epsilon}_m$ is the weighted error corresponding to $\hat{\theta}_m$ chosen in step (1). The value $\hat{\alpha}_m$ is the closed form solution for α_m that minimizes $J(\alpha_m, \hat{\theta}_m)$ for fixed $\hat{\theta}_m$.

(3) Update the weights on the training examples

$$\tilde{W}_m(t) = c_m \cdot \tilde{W}_{m-1}(t) \exp\left(- y_t \hat{\alpha}_m h(\mathbf{x}_t; \hat{\theta}_m)\right) \tag{34}$$

where c_m is the normalization constant to ensure that $\tilde{W}_m(t)$ sum to one. The new weights can be interpreted as normalized losses for the new ensemble $h_m(\mathbf{x}_t) = h_{m-1}(\mathbf{x}) + \hat{\alpha}_m h(\mathbf{x}; \hat{\theta}_m)$.

The Adaboost algorithm sequentially adds base learners to the ensemble so as to decrease the training loss.