Day 1

Classification

This day will serve as an introduction to machine learning. We recall some fundamental concepts about decision theory and classification. We also present some widely used models and algorithms and try to provide the main motivation behind them. There are several textbooks that provide a thorough description of some of the concepts introduced here: for example, Mitchell (1997),Duda et al. (2001), Schölkopf and Smola (2002), Joachims (2002), Bishop (2006), Manning et al. (2008), to name just a few. The concepts that we introduce in this chapter will be revisited in later chapters, where the same algorithms and models will be adapted to structured inputs and outputs. For now, we concern only with multi-class classification (with just a few classes).

1.1 Notation

In what follows, we denote by $\mathcal{X}$ our *input set* (also called *observation set*), and by $\mathcal{Y}$ our *output set*. We will make no assumptions about the set $\mathcal{X}$, which can be continuous or discrete. In this lecture, we consider *classification* problems, where $\mathcal{Y} = \{c_1, \ldots, c_K\}$ is a finite set, consisting of K *classes* (also called *labels*). For example, $\mathcal{X}$ can be a set of documents in natural language, and $\mathcal{Y}$ a set of topics, the goal being to assign a topic to each document.

We use upper-case letters for denoting random variables, and lower-case letters for value assignments to those variables: for example,

- X is a random variable taking values on $\mathcal{X}$,
- Y is a random variable taking values on $\mathcal{Y}$,
- $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ are particular values for X and Y.

We consider *events* such as $X = x$, $Y = y$, etc. Throughout, we use modified notation and let $P(y)$ denote the *probability* associated with the event $Y = y$

(instead of writing $P_Y(Y = y)$). *Joint* and *conditional* probabilities are denoted respectively as $P(x, y) \triangleq P_{X,Y}(X = x \wedge Y = y)$ and $P(x|y) \triangleq P_{X|Y}(X = x \mid Y = y)$. From the laws of probabilities:

$$P(x, y) = P(y|x)P(x), \tag{1.1}$$

for all $x \in \mathcal{X}$ and $y \in \mathcal{Y}$.

Quantities that are predicted or estimated from the data will be appended a hat-symbol: for example, estimations of the probabilities above are denoted as $\hat{P}(y)$, $\hat{P}(x, y)$ and $\hat{P}(y|x)$; and a prediction of an output will be denoted $\hat{y}$.

We assume that a *training dataset* $\mathcal{D}$ is provided which consists of input-output pairs (called *examples* or *instances*):

$$\mathcal{D} = \{(x^1, y^1), \ldots, (x^M, y^M)\} \subseteq \mathcal{X} \times \mathcal{Y}. \tag{1.2}$$

The goal of (supervised) machine learning is to use $\mathcal{D}$ to learn a function h (called a *classifier*) that maps from $\mathcal{X}$ to $\mathcal{Y}$: this way, given a new instance $x \in \mathcal{X}$ (test example), the machine makes a prediction $\hat{y}$ by evaluating h on x, i.e., $\hat{y} = h(x)$.

1.2 Generative Classifiers: Naïve Bayes

If we knew the *true* distribution $P(X, Y)$, the best possible classifier (Bayes optimal) would be one which predicts according to

$$\begin{aligned} \hat{y} &= \arg\max_{y \in \mathcal{Y}} P(y|x) = \arg\max_{y \in \mathcal{Y}} \frac{P(x, y)}{P(x)} \\ &=^{\dagger} \arg\max_{y \in \mathcal{Y}} P(x, y) \\ &= \arg\max_{y \in \mathcal{Y}} P(y)P(x|y), \end{aligned} \tag{1.3}$$

where in † we used the fact that $P(x)$ is constant with respect to y. The probability distributions $P(Y)$ and $P(X|Y)$ are respectively called the *class prior* and the *class conditionals*. Figure 1.2 shows an example of the Bayes optimal decision boundary for a toy example. Generative models assume data are generated according to the following generative story (independently for each $m = 1, \ldots, M$):

1. A class $y_m \sim P(Y)$ is drawn from the class prior distribution;
2. An input $x_m \sim P(X|Y = y_m)$ is drawn from the corresponding class conditional.

Training a generative model amounts to *estimating* these probabilities using the dataset $\mathcal{D}$, yielding estimates $\hat{P}(y)$ and $\hat{P}(x|y)$.

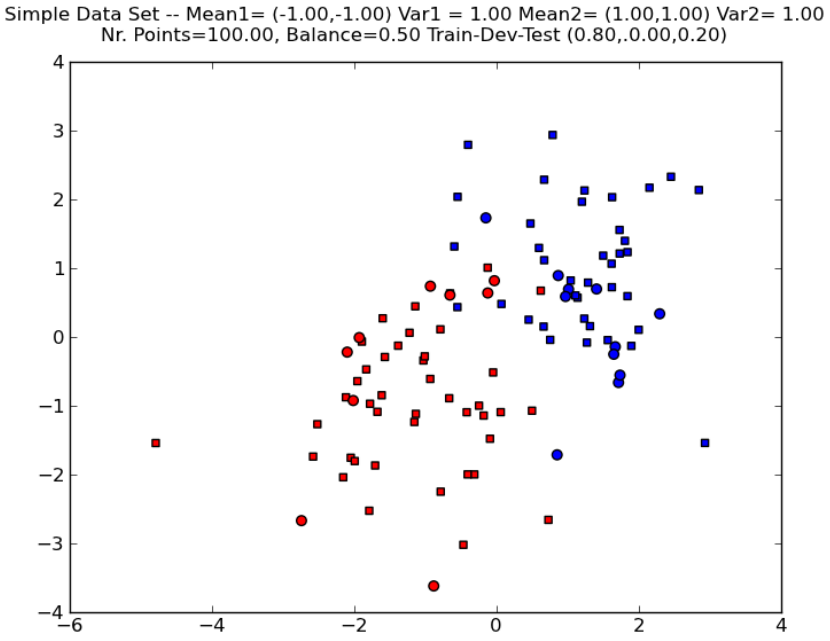

Figure 1.1: Example of a dataset. The input set consists in points in the real plane, $\mathcal{X} = \mathbb{R}^2$, and the output set consists of two classes (Red and Blue). Training points are represented as squares, while test points are represented as circles.

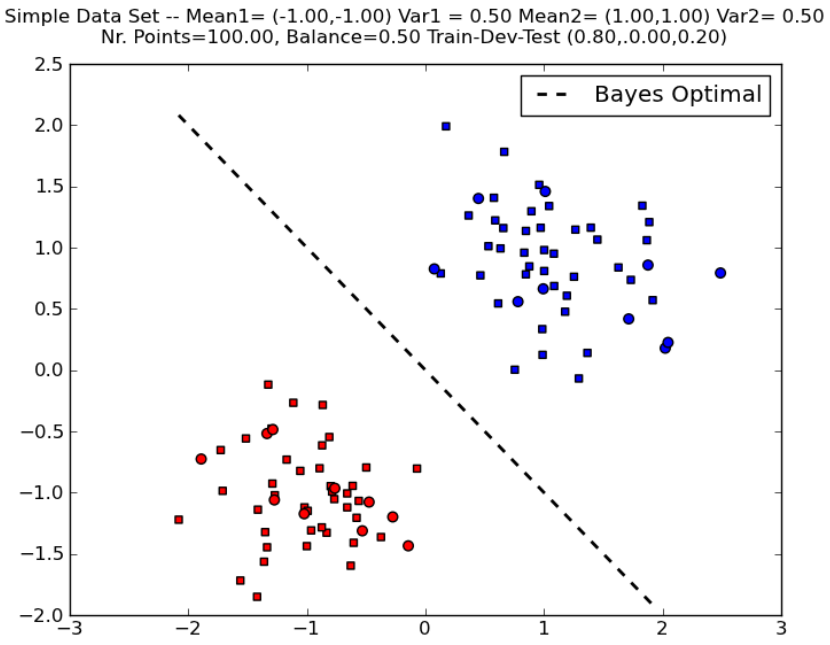

Figure 1.2: Example of a dataset together with the corresponding Bayes optimal decision boundary. The input set consists in points in the real plane, $\mathcal{X} = \mathcal{R}$, and the output set consists of two classes (Red and Blue). Training points are represented as squares, while test points are represented as circles.

At test time, given a new input $x \in \mathcal{X}$, a prediction is made according to

$$\hat{y} \quad = \quad \arg\max_{y \in \mathcal{Y}} \hat{P}(y)\hat{P}(x|y). \tag{1.4}$$

We are left with two important problems:

1. How should the distributions $\hat{P}(Y)$ and $\hat{P}(X|Y)$ be "defined" (i.e., what kind of independence assumptions should they state, or how should they factor?)

2. How should parameters be estimated from the training data $\mathcal{D}$?

The first problem strongly depends on the application at hand. Quite often, there is a natural decomposition of the input variable X into J components,

$$X = (X_1, \ldots, X_J). \tag{1.5}$$

The naïve Bayes method makes the following assumption: $X_1, \ldots, X_J$ *are conditionally independent given the class*. Mathematically, this means that

$$P(X|Y) \quad = \quad \prod_{j=1}^{J} P(X_j|Y). \tag{1.6}$$

Note that this independence assumption greatly reduces the number of parameters to be estimated (degrees of freedom) from $O(\exp(J))$ to $O(J)$, hence estimation of $\hat{P}(Y)$ and $\hat{P}(X|Y)$ becomes much simpler, as we shall see. It also makes the overall computation much more efficient (in particular for large J) and it decreases the risk of overfitting the data. On the other hand, if the assumption is over-simplistic it may increase the risk of under-fitting.

The *maximum likelihood criterion* aims to maximize the probability of the training sample, assuming it was generated iid. This probability (call it $P(\mathcal{D})$) factorizes as

$$\begin{aligned} P(\mathcal{D}) \quad &= \quad \prod_{m=1}^{M} P(x^m, y^m) \\ &= \quad \prod_{m=1}^{M} P(y^m) \prod_{j=1}^{J} P(x_j^m|y^m). \end{aligned} \tag{1.7}$$

1.2.1 Example: 2-D Gaussians

We first illustrate the naïve Bayes assumption with a toy example. Suppose that $\mathcal{X} = \mathbb{R}^2$ and $\mathcal{Y} = \{1, 2\}$. Assume that each class-conditional is a two-dimensional Gaussian distribution with fixed covariance, i.e., $P(X_1, X_2|Y = y) = \mathcal{N}(\boldsymbol{\mu}_y, \boldsymbol{\Sigma}_y)$.

According to the naïve Bayes assumption, $\hat{P}(X_1, X_2|Y) = \hat{P}(X_1|Y)\hat{P}(X_2|Y)$ (remark: this is equivalent to assuming that the $\mathbf{\Sigma}_y$ are diagonal!). For simplicity, we also assume that the two classes have unit variance. Then, we have $\hat{P}(X_1|Y = y) = \mathcal{N}(\mu_{y1}, 1.0)$ and $\hat{P}(X_2|Y = y) = \mathcal{N}(\mu_{y2}, 1.0)$ (Figure 1.1 shows an example a dataset of two gaussians with unit variance. Where $\mu_{y1} = [-1, -1]$ and $\mu_{y1} = [1, 1]$. Figure 1.2 shows the same example but where both gaussian have $\Sigma = 0.5$, together with the Bayes optimal decision boundary). The parameters that need to be estimated are the class-conditional means $\mu_{11}, \mu_{12}, \mu_{21}, \mu_{22}$ and the class priors $\hat{P}(Y = 1)$ and $\hat{P}(Y = 2)$. Given a training sample $\mathcal{D} = \{(x^1, y^1), \ldots, (x^M, y^M)\}$, denote by $\mathcal{I}_1 \subseteq \{1, \ldots, M\}$ the indices of those instances belonging to class 1, and by $\mathcal{I}_2 \subseteq \{1, \ldots, M\}$ the indices of the ones that belong to class 2. The maximum likelihood estimates of the quantities above are:

$$\begin{aligned} \hat{P}(Y = 1) &= \frac{|\mathcal{I}_1|}{M}, \quad \hat{P}(Y = 2) = \frac{|\mathcal{I}_2|}{M} \\ \mu_{11} &= \frac{1}{|\mathcal{I}_1|} \sum_{m \in \mathcal{I}_1} x_1^m, \quad \mu_{12} = \frac{1}{|\mathcal{I}_1|} \sum_{m \in \mathcal{I}_1} x_2^m \\ \mu_{21} &= \frac{1}{|\mathcal{I}_2|} \sum_{m \in \mathcal{I}_2} x_1^m, \quad \mu_{22} = \frac{1}{|\mathcal{I}_2|} \sum_{m \in \mathcal{I}_2} x_2^m. \end{aligned} \tag{1.8}$$

In words: the class priors' estimates are their relative frequencies, and the class-conditional means' estimates are the sample means.

Exercise 1.1 *Start by importing all the libraries necessary for this lab through the following preamble:*

```
import sys
sys.path.append("readers/" )
sys.path.append("classifiers/" )

import simple_data_set as sds
import linear_classifier as lcc
import gaussian_naive_bayes as gnbc
import naive_bayes as nb
```

Now, generate a training and a test dataset like in the previous example, each with $M = 100$ points, 50 of each class. Assume the following class-conditionals: $P(X|Y = 1) \sim N((-1, -1), \sigma^2 \boldsymbol{I})$ and $P(X|Y = 2) \sim N((1, 1), \boldsymbol{I})$, for $\sigma = 1.0$. To do this, run the following command from the `code` *directory:*

```
sd = sds.SimpleDataSet(nr_examples=100, g1 = [[-1,-1],1],
    g2 = [[1,1],1], balance=0.5, split=[0.5, 0, 0.5])
```

You can visualize your data and see the Bayes optimal surface boundary by typing:

```
fig,axis = sd.plot_data()
```

Now, run naïve Bayes on this dataset. To do that, use the class `GaussianNaiveBayes`*, which is defined in the file* `GaussianNaiveBayes.py` *under the classification directory. Report your estimates, as well as training set and testing set accuracies:*

```
gnb = gnbc.GaussianNaiveBayes()
params_nb_sd = gnb.train(sd.train_X, sd.train_y)

print "Estimated Means"
print gnb.means
print "Estimated Priors"
print gnb.prior
y_pred_train = gnb.test(sd.train_X,params_nb_sd)
acc_train = gnb.evaluate(sd.train_y, y_pred_train)
y_pred_test = gnb.test(sd.test_X,params_nb_sd)
acc_test = gnb.evaluate(sd.test_y, y_pred_test)
print "Gaussian Naive Bayes Simple Dataset Accuracy train:
    %f test: %f"%(acc_train,acc_test)
```

To visualize the surface boundary estimated by naïve Bayes, type:

```
fig,axis = sd.add_line(fig,axis,params_nb_sd,"Naive Bayes",
    "red")
```

Do not worry for now about why the surface boundaries look the way they look. This is going to be the subject of §1.3.

Repeat the exercise above for different values of σ^2, different balances, and different sample sizes. What do you observe?

1.2.2 Example: Multinomial Model for Document Classification

We now consider a more realistic scenario where the naïve Bayes classifier may be applied. Suppose that the task is *document classification*: $\mathcal{X}$ is the set of all possible documents, and $\mathcal{Y} = \{c_1, \ldots, c_K\}$ is a set of *topics* for those documents. Let $\mathcal{V} = \{w_1, \ldots, w_J\}$ be the vocabulary, i.e., the set of words that occur in some document.

A very popular document representation is through a "bag-of-words": each document is seen as a multiset of words along with their frequencies; word ordering is ignored. We are going to see that this is equivalent to a naïve Bayes assumption with the *multinomial model*.[1] We associate to each class a multinomial distribution, which ignores word ordering, but takes into consideration the frequency with which each word appears in a document. For simplicity, we assume that all documents have the same length L.[2] Each document x is assumed to have been generated as follows. First, a class y is generated according to $P(y)$. Then, x is generated by sequentially picking words from $\mathcal{V}$ with replacement. Each word w_j is picked with probability $P(w_j|y)$. For example, the probability of generating a document $x = w_{j_1} \dots w_{j_L}$ (*i.e.*, a sequence of L words—*tokens*—$w_{j_1}, \dots, w_{j_L}$) is

$$P(x|y) = \prod_{l=1}^{L} P(w_{j_l}|y) = \prod_{j=1}^{J} P(w_j|y)^{n_j(x)}, \tag{1.9}$$

where $n_j(x)$ is the number of occurrences of word w_j in document x.

Hence, the assumption is that word occurrences (*tokens*) are independent given the class. The parameters that need to be estimated are $\hat{P}(c_1), \dots, \hat{P}(c_K)$, and $\hat{P}(w_j|c_k)$ for $j = 1, \dots, J$ and $k = 1, \dots, K$. Given a training sample $\mathcal{D} = \{(x^1, y^1), \dots, (x^M, y^M)\}$, denote by $\mathcal{I}_k$ the indices of those instances belonging to the kth class. The maximum likelihood estimates of the quantities above are:

$$\hat{P}(c_k) = \frac{|\mathcal{I}_k|}{M}, \qquad \hat{P}(w_j|c_k) = \frac{\sum_{m \in \mathcal{I}_k} n_j(x^m)}{|\mathcal{I}_k|}. \tag{1.10}$$

In words: the class priors' estimates are their relative frequencies (as before), and the class-conditional word probabilities are the relative frequencies of those words across documents with that class.

Exercise 1.2 *In this exercise we will use the the Amazon sentiment analysis data (Blitzer et al., 2007), where the goal is to classify text documents as expressing a* positive *or* negative *sentiment (i.e., a classification problem with two labels). We are going to focus on book reviews. To load the data, type:*

```
import sentiment_reader as srs
import naive_bayes as nb

scr = srs.SentimentCorpus("books")
```

[1] Another popular model for documents is the Bernoulli model, which only looks at the presence/absence of a word in a document, rather than word frequency. See Manning et al. (2008); McCallum and Nigam (1998) for further information.

[2] We can get rid of this assumption by defining a distribution on the document length. Everything stays the same if that distribution is uniform up to a maximum document length.

This will load the data in a bag-of-words representation where rare words (occurring less than 5 *times in the training data) are removed.*

1. *Create a file* `MultinomialNaiveBayes.py` *and implement the naïve Bayes with the multinomial model in a new class* `MultinomialNaiveBayes`*. (Hint: look at the implementation of* `GaussianNaiveBayes` *for inspiration).*

2. *Run naïve Bayes with the multinomial model on the Amazon dataset (sentiment classification) and report results both for training and testing:*

```
import multinomial_naive_bayes as mnb

mnb = mnb.MultinomialNaiveBayes()
params_nb_sc = mnb.train(scr.train_X,scr.train_y)
y_pred_train = mnb.test(scr.train_X,params_nb_sc)
acc_train = mnb.evaluate(scr.train_y, y_pred_train)
y_pred_test = mnb.test(scr.test_X,params_nb_sc)
acc_test = mnb.evaluate(scr.test_y, y_pred_test)
print "Multinomial Naive Bayes Amazon Sentiment Accuracy
    train: %f test: %f"%(acc_train,acc_test)
```

3. *Observe that words that were not observed at training time cause problems at test time. Why? To solve this problem, apply a simple* add-one *smoothing technique: replace the expression in Eq. 1.10 for the estimation of the conditional probabilities by*

$$\hat{P}(w_j|c_k) = \frac{1 + \sum_{m \in \mathcal{I}_k} n_j(x^m)}{J + |\mathcal{I}_k|}.$$

where J is the number of distinct words.

This is a widely used smoothing strategy which has a Bayesian interpretation: it corresponds to choosing a uniform prior for the word distribution on both classes, and to replace the maximum likelihood criterion by a maximum a posteriori *approach. This is a form of* regularization*, preventing the model from* overfitting *on the training data. See* e.g. *Manning and Schütze (1999); Manning et al. (2008) for more information. Report the new accuracies.*

1.3 Features and Linear Classifiers

In the previous section, we assumed a particular representation for the input objects $x \in \mathcal{X}$: points in a 2D Euclidean space (in the Gaussian example) and bag-of-words representations of text documents (in the sentiment data example).

The methods discussed in this lecture are also applicable to a wide range of problems, regardless of the intricacies of our input objects. It is useful to think

about each $x \in \mathcal{X}$ as an abstract object, which is subject to a set of descriptions or measurements, which are called *features*. A feature is simply a real number that describes the value of some property of x. For instance in the toy examples described above you can thing of X as a set of points and the features to be its 2D coordinates. Let $g_1(x), \ldots, g_J(x)$ be J features of x. We call the vector

$$\boldsymbol{g}(x) = (g_1(x), \ldots, g_J(x)) \tag{1.11}$$

a *feature vector representation* of x. The map $\boldsymbol{g} : \mathcal{X} \rightarrow \mathbb{R}^J$ is called a *feature mapping*.

In NLP applications, features are often binary-valued and result from evaluating propositions such as:

$$g_1(x) \triangleq \begin{cases} 1, & \text{if sentence } x \text{ contains the word } \textit{Ronaldo} \\ 0, & \text{otherwise.} \end{cases} \tag{1.12}$$

$$g_2(x) \triangleq \begin{cases} 1, & \text{if all words in sentence } x \text{ are capitalized} \\ 0, & \text{otherwise.} \end{cases} \tag{1.13}$$

$$g_3(x) \triangleq \begin{cases} 1, & \text{if } x \text{ contains any of the words } \textit{amazing}, \textit{excellent} \text{ or } \textit{:-)} \\ 0, & \text{otherwise.} \end{cases} \tag{1.14}$$

In this example, the feature vector representation of the sentence x

Ronaldo kicked the ball and scored an amazing goal!

would be $\boldsymbol{g}(x) = (1, 0, 1)$.

In multi-class learning problems, rather than associating features only with the input objects, it is useful to consider *joint feature mappings* $\boldsymbol{f} : \mathcal{X} \times \mathcal{Y} \rightarrow \mathbb{R}^D$. In that case, the *joint feature vector* $\boldsymbol{f}(x, y)$ can be seen as a collection of joint input-output measurements. For example:

$$f_1(x, y) \triangleq \begin{cases} 1, & \text{if } x \text{ contains } \textit{Ronaldo}, \text{ and topic } y \text{ is } \texttt{sport} \\ 0, & \text{otherwise.} \end{cases} \tag{1.15}$$

$$f_2(x, y) \triangleq \begin{cases} 1, & \text{if } x \text{ contains } \textit{Ronaldo}, \text{ and topic } y \text{ is } \texttt{politics} \\ 0, & \text{otherwise.} \end{cases} \tag{1.16}$$

A very simple form of defining a joint feature mapping which is often employed is via:

$$\begin{aligned} \boldsymbol{f}(x, y) &\triangleq \boldsymbol{g}(x) \otimes \boldsymbol{e}_y \\ &= (0, \ldots, 0, \underbrace{\boldsymbol{g}(x)}_{y\text{th slot}}, 0, \ldots, 0) \end{aligned} \tag{1.17}$$

where $\boldsymbol{g}(x) \in \mathbb{R}^J$ is a input feature vector, $\otimes$ is the Kronecker product ($[\boldsymbol{a} \otimes \boldsymbol{b}]_{ij} = a_i b_j$) and $\boldsymbol{e}_y \in \mathbb{R}^K$, with $[\boldsymbol{e}_y]_c = 1$ iff $y = c$, and 0 otherwise. Hence

$\boldsymbol{f}(x, y) \in \mathbb{R}^D$ with $D = JK$.

Linear classifiers are very popular in natural language processing applications. They make their decision based on the rule:

$$\hat{y} = \arg\max_{y \in \mathcal{Y}} \boldsymbol{w} \cdot \boldsymbol{f}(x, y). \tag{1.18}$$

where

- $\boldsymbol{w} \in \mathbb{R}^D$ is a *weight vector*;
- $\boldsymbol{f}(x, y) \in \mathbb{R}^D$ is a *feature vector*;
- $\boldsymbol{w} \cdot \boldsymbol{f}(x, y) = \sum_{d=1}^{D} w_d f_d(x, y)$ is the inner product between $\boldsymbol{w}$ and $\boldsymbol{f}(x, y)$.

Hence, each feature $f_d(x, y)$ has a weight w_d and, for each class $y \in \mathcal{Y}$, a score is computed by linearly combining all the weighted features. All these scores are compared, and a prediction is made by choosing the class with the largest score.

Remark 1.1 *With the design above (Eq. 1.17), and decomposing the weight vector as* $\boldsymbol{w} = (\boldsymbol{w}_{c_1}, \ldots, \boldsymbol{w}_{c_K})$*, we have that*

$$\boldsymbol{w} \cdot \boldsymbol{f}(x, y) = \boldsymbol{w}_y \cdot \boldsymbol{g}(x). \tag{1.19}$$

In words: each class $y \in \mathcal{Y}$ gets its own weight vector $\boldsymbol{w}_y$, and one defines a input feature vector $\boldsymbol{g}(x)$ that only looks at the input $x \in \mathcal{X}$. This representation is very useful when features only depend on input x since it allows a more compact representation. Note that the number of features is normally very large.

Remark 1.2 *The multinomial naïve Bayes classifier described in the previous section is an instance of a linear classifier (in fact, so is the 2-D Gaussian Bayes classifier—try to show this). Recall that the naïve Bayes classifier predicts according to* $\hat{y} = \arg\max_{y \in \mathcal{Y}} \hat{P}(y)\hat{P}(x|y)$*. Taking logs, in the multinomial model for document classification this is equivalent to:*

$$\begin{aligned}
\hat{y} &= \arg\max_{y \in \mathcal{Y}} \log \hat{P}(y) + \log \hat{P}(x|y) \\
&= \arg\max_{y \in \mathcal{Y}} \log \hat{P}(y) + \sum_{j=1}^{J} n_j(x) \log \hat{P}(w_j|y) \\
&= \arg\max_{y \in \mathcal{Y}} \boldsymbol{w}_y \cdot \boldsymbol{g}(x),
\end{aligned} \tag{1.20}$$

where

$$\begin{aligned}
\boldsymbol{w}_y &= \left(b_y, \log \hat{P}(w_1|y), \ldots, \log \hat{P}(w_J|y)\right) \\
b_y &= \log \hat{P}(y) \\
\boldsymbol{g}(x) &= (1, n_1(x), \ldots, n_J(x)).
\end{aligned} \tag{1.21}$$

Algorithm 2 Averaged perceptron

input: dataset $\mathcal{D}$, number of rounds R
initialize $t = 0, \boldsymbol{w}^t = \mathbf{0}$
for $r = 1$ **to** R **do**
  $\mathcal{D}_s = \text{shuffle}(\mathcal{D})$
  for $i = 1$ **to** M **do**
    $m = \mathcal{D}_s(i)$
    $t = t + 1$
    take training pair (x^m, y^m) and predict using the current model:

$$\hat{y} \leftarrow \arg\max_{y' \in \mathcal{Y}} \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y')$$

    update the model: $\boldsymbol{w}^{t+1} \leftarrow \boldsymbol{w}^t + \boldsymbol{f}(x^m, y^m) - \boldsymbol{f}(x^m, \hat{y})$
  end for
end for
output: the averaged model $\hat{\boldsymbol{w}} \leftarrow \frac{1}{t} \sum_{i=1}^{t} \boldsymbol{w}^i$

Hence, the multinomial model yields a prediction rule of the form

$$\hat{y} \quad = \quad \arg\max_{y \in \mathcal{Y}} \boldsymbol{w}_y \cdot \boldsymbol{g}(x). \tag{1.22}$$

Exercise 1.3 *Show that the Gaussian naïve Bayes classifier with shared and given variance is also a linear classifier, and derive the formulas for $\boldsymbol{w}_y, b_y$. You should obtain the formulas that are implemented in the* `train` *method of* `GaussianNaiveBayes`.

Look again at the decision boundary that you have found in Exercise 1.1 and compare it with the Bayes optimal classifier.

1.4 Online Algorithms: Perceptron and MIRA

1.4.1 Perceptron

Perhaps the oldest algorithm to train a linear classifier is the *perceptron* (Rosenblatt, 1958), which we depict as Alg. 2.[3]

The perceptron algorithm works as follows: at each round, it takes an input datum, and uses the current model to make a prediction. If the prediction is correct, nothing happens. Otherwise, the model is corrected by adding the feature vector w.r.t. the correct output and subtracting the feature vector w.r.t. the predicted (wrong) output. Then, we proceed to the next round. Alg. 2 is remarkably simple; yet it often reaches a very good performance, often bet-

[3] Actually, we are showing a more robust variant of the perceptron, which averages the weight vector as a post-processing step.

ter than the Naïve Bayes model, and usually not much worse than maximum entropy models or SVMs (which will be described in the next section).

A weight vector $\boldsymbol{w}$ defines a *separating hyperplane* if it classifies all the training data correctly, *i.e.*, if $y^m = \arg\max_{y \in \mathcal{Y}} \boldsymbol{w} \cdot \boldsymbol{f}(x^m, y)$ hold for $m = 1, \ldots, M$. A dataset $\mathcal{D}$ is *separable* if such a weight vector exists (in general, $\boldsymbol{w}$ is not unique). A very important property of the perceptron algorithm is the following: if $\mathcal{D}$ is separable, then the number of mistakes made by the perceptron algorithm is *finite*. This means that under this assumption, the perceptron will eventually reach a separating hyperplane $\boldsymbol{w}$.

There are other variants of the perceptron (e.g., with regularization) which we omit for brevity.

Exercise 1.4 *We provide an implementation of the perceptron algorithm in the class* `Perceptron` *(file* `Perceptron.py`*).*

1. *Run the perceptron algorithm on the simple dataset previously generated and report its train and test set accuracy:*

```
import perceptron as percc

perc = percc.Perceptron()
params_perc_sd = perc.train(sd.train_X,sd.train_y)
y_pred_train = perc.test(sd.train_X,params_perc_sd)
acc_train = perc.evaluate(sd.train_y, y_pred_train)
y_pred_test = perc.test(sd.test_X,params_perc_sd)
acc_test = perc.evaluate(sd.test_y, y_pred_test)
print "Perceptron Simple Dataset Accuracy train: %f test: %
    f"%(acc_train,acc_test)
```

2. *Plot the decision boundary found:*

```
fig,axis = sd.add_line(fig,axis,params_perc_sd,"Perceptron"
    ,"blue")
```

   Change the code to save the intermediate weight vectors, and plot them every five iterations. What do you observe?

3. *Run the perceptron algorithm on the Amazon dataset.*

1.4.2 Margin Infused Relaxed Algorithm (MIRA)

The MIRA algorithm (Crammer and Singer, 2002; Crammer et al., 2006) has achieved very good performance in NLP problems. At each round t, MIRA updates the weight vector by solving the following optimization problem:

$$\boldsymbol{w}^{t+1} \leftarrow \arg\min_{\boldsymbol{w},\xi} \quad \xi + \tfrac{\lambda}{2}\|\boldsymbol{w} - \boldsymbol{w}^t\|^2 \tag{1.23}$$

$$\text{s.t.} \quad \boldsymbol{w} \cdot \boldsymbol{f}(x^m, y^m) \geq \boldsymbol{w} \cdot \boldsymbol{f}(x^m, \hat{y}) + 1 - \xi \tag{1.24}$$

$$\xi \geq 0, \tag{1.25}$$

where $\hat{y} = \arg\max_{y' \in \mathcal{Y}} \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y')$ is the prediction using the model with weight vector $\boldsymbol{w}^t$. By inspecting Eq. 1.23 we see that MIRA attempts to achieve a tradeoff between *conservativeness* (penalizing large changes from the previous weight vector via the term $\frac{\lambda}{2}\|\boldsymbol{w} - \boldsymbol{w}^t\|^2$) and *correctness* (by requiring, through the constraints, that the new model $\boldsymbol{w}^{t+1}$ "separates" the true output from the prediction with a margin (although slack $\xi \geq 0$ is allowed).[4] Note that, if the prediction is correct ($\hat{y} = y^m$) the solution of the problem Eq. 1.23 leaves the weight vector unchanged ($\boldsymbol{w}^{t+1} = \boldsymbol{w}^t$). This quadratic programming problem has a closed form solution:[5]

$$\boldsymbol{w}^{t+1} \leftarrow \boldsymbol{w}^t + \eta^t(\boldsymbol{f}(x^m, y^m) - \boldsymbol{f}(x^m, \hat{y})),$$

with

$$\eta^t = \min\left\{\lambda^{-1}, \frac{\boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, \hat{y}) - \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y^m) + \rho(\hat{y}, y^m)}{\|\boldsymbol{f}(x^m, y^m) - \boldsymbol{f}(x^m, \hat{y})\|^2}\right\},$$

where $\rho : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$ is a non-negative cost function, such that $\rho(\hat{y}, y)$ is the cost incurred by predicting $\hat{y}$ when the true output is y; we assume $\rho(y, y) = 0$ for all $y \in \mathcal{Y}$. For simplicity, we focus here on the 0/1-cost (but keep in mind that other cost functions are possible):

$$\rho(\hat{y}, y) = \begin{cases} 1 & \text{if } \hat{y} \neq y \\ 0 & \text{otherwise.} \end{cases} \tag{1.26}$$

MIRA is depicted in Alg. 3. For other variants of MIRA, see Crammer et al. (2006).

Exercise 1.5 *Implement the MIRA algorithm (Hint: use the perceptron algorithm as a starting point and modify it as necessary). Do this by creating a file* `Mira.py` *and implement class* `Mira`*. Then, repeat the perceptron exercise now using MIRA, for several values of* λ*:*

```
import mira as mirac

mira = mirac.Mira()
mira.regularizer = 1.0 # This is lambda
```

[4] The intuition for this large margin separation is the same for support vector machines, which will be discussed in §1.5.2.

[5] Note that the perceptron updates are identical, except that we always have $\eta_t = 1$.

Algorithm 3 MIRA

input: dataset $\mathcal{D}$, parameter λ, number of rounds R
initialize $t = 0, \boldsymbol{w}^t = \mathbf{0}$
for $r = 1$ **to** R **do**
  $\mathcal{D}_s = \text{shuffle}(\mathcal{D})$
  for $i = 1$ **to** M **do**
    $m = \mathcal{D}_s(i)$
    $t = t + 1$
    take training pair (x^m, y^m) and predict using the current model:

$$\hat{y} \leftarrow \arg\max_{y' \in \mathcal{Y}} \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y')$$

    compute loss: $\ell^t = \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, \hat{y}) - \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y^m) + \rho(\hat{y}, y^m)$
    compute stepsize: $\eta^t = \min\left\{\lambda^{-1}, \frac{\ell^t}{\|\boldsymbol{f}(x^m, y^m) - \boldsymbol{f}(x^m, \hat{y})\|^2}\right\}$
    update the model: $\boldsymbol{w}^{t+1} \leftarrow \boldsymbol{w}^t + \eta^t(\boldsymbol{f}(x^m, y^m) - \boldsymbol{f}(x^m, \hat{y}))$
  end for
end for
output: the averaged model $\hat{\boldsymbol{w}} \leftarrow \frac{1}{t}\sum_{i=1}^{t} \boldsymbol{w}^i$

```
params_mira_sd = mira.train(sd.train_X,sd.train_y)
y_pred_train = mira.test(sd.train_X,params_mira_sd)
acc_train = mira.evaluate(sd.train_y, y_pred_train)
y_pred_test = mira.test(sd.test_X,params_mira_sd)
acc_test = mira.evaluate(sd.test_y, y_pred_test)
print "Mira Simple Dataset Accuracy train: %f test: %f"%(
    acc_train,acc_test)
fig,axis = sd.add_line(fig,axis,params_mira_sd,"Mira","green")

params_mira_sc = mira.train(scr.train_X,scr.train_y)
y_pred_train = mira.test(scr.train_X,params_mira_sc)
acc_train = mira.evaluate(scr.train_y, y_pred_train)
y_pred_test = mira.test(scr.test_X,params_mira_sc)
acc_test = mira.evaluate(scr.test_y, y_pred_test)
print "Mira Amazon Sentiment Accuracy train: %f test: %f"%(
    acc_train,acc_test)
```

Compare the results achieved and separating hiperplanes found.

1.5 Discriminative Classifiers: Maximum Entropy and Support Vector Machines

Unlike the naïve Bayes classifier, the algorithms described in the last section (perceptron and MIRA) directly focus on finding a separating hyperplane to discriminate among the classes, rather than attempting to model the probability $P(X, Y)$ that generates the data. This kind of methods are called *discriminative* (by opposition to the *generative* ones). This section presents two important discriminative classifiers, with widespread use in NLP applications: maximum entropy and support vector machines.

1.5.1 Maximum Entropy Classifiers

The notion of *entropy* in the context of Information Theory (Shannon, 1948) is one of the most significant advances in mathematics in the twentieth century. The principle of *maximum entropy* (which appears under different names, such as "maximum mutual information" or "minimum Kullback-Leibler divergence") plays a fundamental role in many methods in statistics and machine learning (Jaynes, 1982). For an excellent textbook on Information Theory, we recommend Cover et al. (1991). The basic rationale is that choosing the model with the highest entropy (subject to constraints that depend on the observed data) corresponds to making the fewest possible assumptions regarding what was unobserved, trying to making uncertainty about the model as large as possible. For example, if we have a dice and want to estimate the probability of its outcomes, the distribution with the highest entropy would be the uniform distribution (each outcome having of probability a $1/6$). Now suppose that we partition the set of possible outcomes in two groups and are only told about how many times outcomes on each of the groups have occurred. If we know that outcomes $\{1, 2, 3\}$ occurred 10 times in total, and $\{4, 5, 6\}$ occurred 30 times in total, then the principle of maximum entropy would lead us to estimate $P(1) = P(2) = P(3) = 1/12$ and $P(1) = P(2) = P(3) = 1/4$ (i.e., outcomes would be uniform within each of the two groups).

For an introduction of maximum entropy models, along with pointers to the literature, see `http://www.cs.cmu.edu/~aberger/maxent.html`. A fundamental result is that the maximum entropy distribution $P_{\boldsymbol{w}}(Y|X)$ under *first moment matching constraints* (which mean that feature expectations under that distribution $\frac{1}{M}\sum_m E_{Y\sim P_{\boldsymbol{w}}}[\boldsymbol{f}(x_m, Y)]$ must match the observed relative frequencies $\frac{1}{M}\sum_m \boldsymbol{f}(x_m, y_m)$) is a *log-linear model*. The dual of that optimization problem is that of maximizing likelihood in a log-linear model (in the binary case, called *logistic regression* model).

The maximum entropy distribution[6] has the following parametric form:

$$P_{\boldsymbol{w}}(y|x) = \frac{\exp(\boldsymbol{w} \cdot \boldsymbol{f}(x,y))}{Z(\boldsymbol{w},x)} \tag{1.27}$$

The denominator in Eq. 1.27 is called the *partition function*:

$$Z(\boldsymbol{w},x) = \sum_{y' \in \mathcal{Y}} \exp(\boldsymbol{w} \cdot \boldsymbol{f}(x,y')). \tag{1.28}$$

An important property of the partition function is that the gradient of its logarithm equals the feature expectations:

$$\begin{aligned} \nabla_{\boldsymbol{w}} \log Z(\boldsymbol{w},x) &= E_{\boldsymbol{w}}[\boldsymbol{f}(x,Y)] \\ &= \sum_{y' \in \mathcal{Y}} P_{\boldsymbol{w}}(y'|x)\boldsymbol{f}(x,y'). \end{aligned} \tag{1.29}$$

Maximum entropy models are trained *discriminatively*: this means that, instead of maximizing the *joint* likelihood $P_{\boldsymbol{w}}(x^1,\ldots,x^M,y^1,\ldots,y^M)$ (like generative approaches, such as naïve Bayes, do), one maximizes directly the *conditional* likelihood $P_{\boldsymbol{w}}(y^1,\ldots,y^M|x^1,\ldots,x^M)$. The rationale is that one does not need to worry about modeling the input variables if all we want is an accurate estimate of $P(Y|X)$, which is what matters for prediction. The average conditional log-likelihood is:

$$\begin{aligned} \mathcal{L}(\boldsymbol{w};\mathcal{D}) &= \frac{1}{M} \log P_{\boldsymbol{w}}(y^1,\ldots,y^M|x^1,\ldots,x^M) \\ &= \frac{1}{M} \log \prod_{m=1}^{M} P_{\boldsymbol{w}}(y^m|x^m) \\ &= \frac{1}{M} \sum_{m=1}^{M} \log P_{\boldsymbol{w}}(y^m|x^m) \\ &= \frac{1}{M} \sum_{m=1}^{M} \left(\boldsymbol{w} \cdot \boldsymbol{f}(x^m,y^m) - \log Z(\boldsymbol{w},x^m)\right). \end{aligned} \tag{1.30}$$

We try to find the parameters $\boldsymbol{w}$ that maximize the log-likelihood $\mathcal{L}(\boldsymbol{w};\mathcal{D})$; to avoid overfitting, we add a regularization term that penalizes values of $\boldsymbol{w}$ that have a high magnitude. The optimization problem becomes:

$$\begin{aligned} \hat{\boldsymbol{w}} &= \arg\max_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w};\mathcal{D}) - \frac{\lambda}{2}\|\boldsymbol{w}\|^2 \\ &= \arg\min_{\boldsymbol{w}} -\mathcal{L}(\boldsymbol{w};\mathcal{D}) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2. \end{aligned} \tag{1.31}$$

[6] Also called a log-linear model, a Boltzmann distribution, or an exponential family of distributions.

Here we use the squared L_2-norm as the regularizer,[7] but other norms are possible. The scalar $\lambda \geq 0$ controls the amount of regularization. Unlike the naïve Bayes examples, this optimization problem does not have a closed form solution in general; hence we need to resort to numerical optimization (see section **??**). Let $F_\lambda(\boldsymbol{w}; \mathcal{D}) = -\mathcal{L}(\boldsymbol{w}; \mathcal{D}) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2$ be the objective function in Eq. 1.31. This function is convex, which implies that a local optimum of Eq. 1.31 is also a global optimum. $F_\lambda(\boldsymbol{w}; \mathcal{D})$ is also differentiable: its gradient is

$$\begin{aligned} \nabla_{\boldsymbol{w}} F_\lambda(\boldsymbol{w}; \mathcal{D}) &= \frac{1}{M} \sum_{m=1}^{M} (-\boldsymbol{f}(x^m, y^m) + \nabla_{\boldsymbol{w}} \log Z(\boldsymbol{w}, x^m)) + \lambda \boldsymbol{w} \\ &= \frac{1}{M} \sum_{m=1}^{M} (-\boldsymbol{f}(x^m, y^m) + E_{\boldsymbol{w}}[\boldsymbol{f}(x^m, Y)]) + \lambda \boldsymbol{w}. \end{aligned} \tag{1.32}$$

A batch gradient method to optimize Eq. 1.31 is shown in Alg. 4. Essentially, Alg. 4 iterates through the following updates until convergence:

$$\begin{aligned} \boldsymbol{w}^{t+1} &\leftarrow \boldsymbol{w}^t - \eta_t \nabla_{\boldsymbol{w}} F_\lambda(\boldsymbol{w}^t; \mathcal{D}) \\ &= (1 - \lambda\eta_t)\boldsymbol{w}^t + \eta_t \frac{1}{M} \sum_{m=1}^{M} (\boldsymbol{f}(x^m, y^m) - E_{\boldsymbol{w}}[\boldsymbol{f}(x^m, Y)]). \end{aligned} \tag{1.33}$$

Convergence is ensured for suitable stepsizes η_t. Monotonic decrease of the objective value can also be ensured if η_t is chosen with a suitable line search method, such as Armijo's rule (Nocedal and Wright, 1999). In practice, more sophisticated methods exist for optimizing Eq. 1.31, such as conjugate gradient or L-BFGS. The latter is an example of a quasi-Newton method, which only requires gradient information, but use past gradients to try to construct second order (Hessian) approximations.

In large-scale problems (very large M) batch methods are slow, *online* or *stochastic* optimization make attractive alternative methods. Stochastic gradient methods make "noisy" gradient updates by considering only a single instance at the time. The resulting algorithm is shown as Alg. 5. At each round t, an instance $m(t)$ is chosen, either randomly (stochastic variant) or by cycling through the dataset (online variant). The stepsize sequence must decrease with t: typically, $\eta_t = \eta_0 t^{-\alpha}$ for some $\eta_0 > 0$ and $\alpha \in [1, 2]$, tuned in a development partition or with cross-validation.

Exercise 1.6 *We provide an implementation of the L-BFGS algorithm for training maximum entropy models in the class* `MaxEnt_batch`*, as well as an implementation of the SGD algorithm in the class* `MaxEnt_online`*.*

1. *Train a maximum entropy model using L-BFGS on the Simple data set (try*

[7] In a Bayesian perspective, this corresponds to choosing independent Gaussian priors $p(w_d) \sim \mathcal{N}(0; 1/\lambda^2)$ for each dimension of the weight vector.

Algorithm 4 Batch Gradient Descent for Maximum Entropy

input: $\mathcal{D}$, λ, number of rounds T,
learning rate sequence $(\eta_t)_{t=1,\ldots,T}$
initialize $\boldsymbol{w}^1 = \mathbf{0}$
for $t = 1$ **to** T **do**
  for $m = 1$ **to** M **do**
    take training pair (x^m, y^m) and compute conditional probabilities using the current model, for each $y' \in \mathcal{Y}$:

$$P_{\boldsymbol{w}^t}(y'|x^m) = \frac{\exp(\boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y'))}{Z(\boldsymbol{w}, x)}$$

    compute the feature vector expectation:

$$E_{\boldsymbol{w}}[\boldsymbol{f}(x^m, Y)] = \sum_{y' \in \mathcal{Y}} P_{\boldsymbol{w}^t}(y'|x^m)\boldsymbol{f}(x^m, y')$$

  end for
  choose the stepsize η_t using, *e.g.*, Armijo's rule
  update the model:

$$\boldsymbol{w}^{t+1} \leftarrow (1 - \lambda\eta_t)\boldsymbol{w}^t + \eta_t M^{-1} \sum_{m=1}^{M} (\boldsymbol{f}(x^m, y^m) - E_{\boldsymbol{w}}[\boldsymbol{f}(x^m, Y)])$$

end for
output: $\hat{w} \leftarrow \boldsymbol{w}^{T+1}$

different values of λ). Compare the results with the previous methods. Plot the decision boundary.

```
import max_ent_batch as mebc

me_lbfgs = mebc.MaxEnt_batch()
me_lbfgs.regularizer = 1.0
params_meb_sd = me_lbfgs.train(sd.train_X,sd.train_y)
y_pred_train = me_lbfgs.test(sd.train_X,params_meb_sd)
acc_train = me_lbfgs.evaluate(sd.train_y, y_pred_train)
y_pred_test = me_lbfgs.test(sd.test_X,params_meb_sd)
acc_test = me_lbfgs.evaluate(sd.test_y, y_pred_test)
print "Max-Ent batch Simple Dataset Accuracy train: %f test
    : %f"%(acc_train,acc_test)

fig,axis = sd.add_line(fig,axis,params_meb_sd,"Max-Ent-
    Batch","orange")
```

Algorithm 5 SGD for Maximum Entropy

input: $\mathcal{D}$, λ, number of rounds T,
learning rate sequence $(\eta_t)_{t=1,\ldots,T}$
initialize $\boldsymbol{w}^1 = \mathbf{0}$
for $t = 1$ **to** T **do**
  choose $m = m(t)$ randomly
  take training pair (x^m, y^m) and compute conditional probabilities using the current model, for each $y' \in \mathcal{Y}$:

$$P_{\boldsymbol{w}^t}(y'|x^m) = \frac{\exp(\boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y'))}{Z(\boldsymbol{w}, x)}$$

  compute the feature vector expectation:

$$E_{\boldsymbol{w}}[\boldsymbol{f}(x^m, Y)] = \sum_{y' \in \mathcal{Y}} P_{\boldsymbol{w}^t}(y'|x^m)\boldsymbol{f}(x^m, y')$$

  update the model:

$$\boldsymbol{w}^{t+1} \leftarrow (1 - \lambda\eta_t)\boldsymbol{w}^t + \eta_t \left(\boldsymbol{f}(x^m, y^m) - E_{\boldsymbol{w}}[\boldsymbol{f}(x^m, Y)]\right)$$

end for
output: $\hat{\boldsymbol{w}} \leftarrow \boldsymbol{w}^{T+1}$

2. *Train a maximum entropy model using L-BFGS, on the Amazon dataset (try different values of λ) and report training and test set accuracy. What do you observe?*

```
params_meb_sc = me_lbfgs.train(scr.train_X,scr.train_y)
y_pred_train = me_lbfgs.test(scr.train_X,params_meb_sc)
acc_train = me_lbfgs.evaluate(scr.train_y, y_pred_train)
y_pred_test = me_lbfgs.test(scr.test_X,params_meb_sc)
acc_test = me_lbfgs.evaluate(scr.test_y, y_pred_test)
print "Max-Ent Batch Amazon Sentiment Accuracy train: %f
    test: %f"%(acc_train,acc_test)
```

3. *Now, fix $\lambda = 1.0$ and train with SGD (you might try to adjust the initial step). Compare the objective values obtained during training with those obtained with L-BFGS. What do you observe?*

```
import max_ent_online as meoc

me_sgd = meoc.MaxEnt_online()
me_sgd.regularizer = 1.0
```

```
params_meo_sc = me_sgd.train(scr.train_X,scr.train_y)
y_pred_train = me_sgd.test(scr.train_X,params_meo_sc)
acc_train = me_sgd.evaluate(scr.train_y, y_pred_train)
y_pred_test = me_sgd.test(scr.test_X,params_meo_sc)
acc_test = me_sgd.evaluate(scr.test_y, y_pred_test)
print "Max-Ent Online Amazon Sentiment Accuracy train: %f
    test: %f"%(acc_train,acc_test)
```

1.5.2 Support Vector Machines

Support vector machines are also a discriminative approach, but they are not a probabilistic model at all. The basic idea is that, if the goal is to accurately predict outputs (according to some cost function), we should focus on that goal in the first place, rather than trying to estimate a probability distribution ($P(Y|X)$ or $P(X,Y)$), which is a more difficult problem. As Vapnik (1995) puts it, "do not solve an estimation problem of interest by solving a more general (harder) problem as an intermediate step."

We next describe the *primal* problem associated with multi-class support vector machines (Crammer and Singer, 2002), which is of primary interest in natural language processing. There is a significant amount of literature about Kernel Methods (Schölkopf and Smola, 2002; Shawe-Taylor and Cristianini, 2004) mostly focused on the *dual* formulation. We will not discuss non-linear kernels or this dual formulation here.[8]

Consider $\rho(y', y)$ as a non-negative cost function. For simplicity, we focus here on the $0/1$-cost defined by Equation 1.26 (but keep in mind that other cost functions are possible). The *hinge loss*[9] is the function

$$\ell(\boldsymbol{w}; x, y) = \max_{y' \in \mathcal{Y}} \boldsymbol{w} \cdot \boldsymbol{f}(x, y') - \boldsymbol{w} \cdot \boldsymbol{f}(x, y) + \rho(y', y). \tag{1.34}$$

Note that the objective of Eq. 1.34 becomes zero when $y' = y$. Hence, we always have $\ell(\boldsymbol{w}; x, y) \geq 0$. Moreover, if ρ is the $0/1$ cost, we have $\ell(\boldsymbol{w}; x, y) = 0$ if and only if the weight vector is such that the model makes a correct prediction with a *margin* greater than 1: *i.e.*, $\boldsymbol{w} \cdot \boldsymbol{f}(x, y) \geq \boldsymbol{w} \cdot \boldsymbol{f}(x, y') + 1$ for all $y' \neq y$. Otherwise, a positive loss is incurred.

[8] The main reason why we prefer to discuss the primal formulation with linear kernels is that the resulting algorithms run in linear time (or less), while known kernel-based methods are quadratic with respect to M. In large-scale problems (large M) the former are thus more appealing.

[9] The hinge loss for the $0/1$ cost is sometimes defined as $\ell(\boldsymbol{w}; x, y) = \max\{0, \max_{y' \neq y} \boldsymbol{w} \cdot \boldsymbol{f}(x, y') - \boldsymbol{w} \cdot \boldsymbol{f}(x, y) + 1\}$. Given our definition of $\rho(\hat{y}, y)$, note that the two definitons are equivalent.

Support vector machines (SVM) tackle the following optimization problem:

$$\hat{\boldsymbol{w}} \quad = \quad \arg\min_{\boldsymbol{w}} \sum_{m=1}^{M} \ell(\boldsymbol{w}; x^m, y^m) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2, \tag{1.35}$$

where we also use the squared L_2-norm as the regularizer. For the 0/1-cost, the problem in Eq. 1.35 is equivalent to:

$$\arg\min_{\boldsymbol{w},\boldsymbol{\xi}} \quad \sum_{m=1}^{M} \xi_m + \frac{\lambda}{2}\|\boldsymbol{w}\|^2 \tag{1.36}$$

$$\text{s.t.} \quad \boldsymbol{w} \cdot \boldsymbol{f}(x^m, y^m) \geq \boldsymbol{w} \cdot \boldsymbol{f}(x^m, \tilde{y}^m) + 1 - \xi_m, \quad \forall m, \tilde{y}^m \in \mathcal{Y} \setminus \{y^m\} \tag{1.37}$$

Geometrically, we are trying to choose the linear classifier that yields the largest possible separation margin, while we allow some violations, penalizing the amount of slack via extra variables $\xi_1, \ldots, \xi_m$.

Problem 1.35 does not have a closed form solution. Moreover, unlike maximum entropy models, here the objective function is non-differentiable, hence smooth optimization is not possible. However, it is still convex, which ensures that any local optimum is the global optimum. Despite not being differentiable, we can still define a *subgradient* of the objective function (which generalizes the concept of gradient), which enables us to apply subgradient-based methods. A stochastic subgradient algorithm for solving Eq. 1.35 is illustrated as Alg. 6. The similarity with maximum entropy models (Alg. 5) is striking: the only difference is that, instead of computing the feature vector expectation using the current model, we compute the feature vector associated with the cost-augmented prediction using the current model.

A variant of this algorithm was proposed by Shalev-Shwartz et al. (2007) under the name *Pegasos*, with excellent properties in large-scale settings. Other algorithms and software packages for training SVMs that have become popular are SVMLight (`http://svmlight.joachims.org`) and LIBSVM (`http://www.csie.ntu.edu.tw/~cjlin/libsvm/`), which allow non-linear kernels. These will generally be more suitable for smaller datasets, where high accuracy optimization can be obtained without much computational effort.

Remark 1.3 *Note the similarity between the stochastic (sub-)gradient algorithms (Algs. 5–6) and the online algorithms seen above (perceptron and MIRA).*

Exercise 1.7 *Implement the SVM primal algorithm (Hint: look at the models implemented earlier, you should only need to change a few lines of code). Do this by creating a file* `SVM.py` *and implement class* `SVM`*. Then, repeat the MaxEnt exercise now using SVMs, for several values of* λ*:*

```
import svm as svmc

svm = svmc.SVM()
```

Algorithm 6 Stochastic Subgradient Descent for SVMs

1: **input:** $\mathcal{D}$, λ, number of rounds T,
learning rate sequence $(\eta_t)_{t=1,\ldots,T}$
2: initialize $\boldsymbol{w}^1 = \mathbf{0}$
3: **for** $t = 1$ **to** T **do**
4: choose $m = m(t)$ randomly
5: take training pair (x^m, y^m) and compute the "cost-augmented prediction" under the current model:

$$\tilde{y} = \arg\max_{y' \in \mathcal{Y}} \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y') - \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y^m) + \rho(y', y)$$

6: update the model:

$$\boldsymbol{w}^{t+1} \leftarrow (1 - \lambda\eta_t)\boldsymbol{w}^t + \eta_t \left(\boldsymbol{f}(x^m, y^m) - \boldsymbol{f}(x^m, \tilde{y})\right)$$

7: **end for**
8: **output:** $\hat{\boldsymbol{w}} \leftarrow \boldsymbol{w}^{T+1}$

```
svm.regularizer = 1.0 # This is lambda
params_svm_sd = svm.train(sd.train_X,sd.train_y)
y_pred_train = svm.test(sd.train_X,params_svm_sd)
acc_train = svm.evaluate(sd.train_y, y_pred_train)
y_pred_test = svm.test(sd.test_X,params_svm_sd)
acc_test = svm.evaluate(sd.test_y, y_pred_test)
print "SVM Online Simple Dataset Accuracy train: %f test: %f"%(
    acc_train,acc_test)

fig,axis = sd.add_line(fig,axis,params_svm_sd,"SVM","orange")

params_svm_sc = svm.train(scr.train_X,scr.train_y)
y_pred_train = svm.test(scr.train_X,params_svm_sc)
acc_train = svm.evaluate(scr.train_y, y_pred_train)
y_pred_test = svm.test(scr.test_X,params_svm_sc)
acc_test = svm.evaluate(scr.test_y, y_pred_test)
print "SVM Online Amazon Sentiment Accuracy train: %f test: %f"
    %(acc_train,acc_test)
```

Compare the results achieved and separating hiperplanes found.

1.6 Comparison

Table 1.6 provides a high-level comparison among the different models discussed in this chapter.

	Naive Bayes	Perceptron	MIRA	MaxEnt	SVMs
Generative/Discriminative	G	D	D	D	D
Performance if true model not in the hipothesis class	Bad	Fair (may not converge)	Good	Good	Good
Performance if features overlap	Fair	Good	Good	Good	Good
Training	Closed Form	Easy	Easy	Fair	Fair
Hyperparameters to tune	1 (smoothing)	0	1	1	1

Table 1.1: Comparison among different models.

Exercise 1.8 • *Using the simple data set run the different models varying some characteristics of the data, number of points, variance (hence separability), class balance. Use function XX which receives a dataset and plots all decisions boundaries and accuracies. What can you say about the methods when the amount of data increases? What about when the classes become too unbalanced.*

1.7 Final remarks

Some implementations of the discussed algorithms are available on the Web:

- SVMLight: `http://svmlight.joachims.org`
- LIBSVM: `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`
- Maximum Entropy: `http://homepages.inf.ed.ac.uk/lzhang10/maxent_toolkit.html`
- MALLET: `http://mallet.cs.umass.edu/`.