CIS 419/519: Applied Machine Learning
Lecture 4: On-line Learning

Professor: Dan Roth

This lecture note accompanies 2 Google Colab notebooks. Please click here and here to access them.

1 Quantifying Performance

We now discuss ways to quantify the performance of various learning algorithms so that we can rigorously analyze their performance in a given task. This section discusses a number of examples needed to claim that our learned hypothesis is good.

1.1 Learning Conjunction

Assume that you want to learn the following hidden monotone conjunction [1]:

$$f = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$$

Without going into any details think about these questions: How many examples are needed in order to learn the function? How is it learned from these examples? There are multiple possible learning protocols, we will cover the following three:

- **Protocol I:** The learner proposes instances as queries to the teacher[2].
- **Protocol II:** The teacher provides training examples. Since the teacher knows the hidden function f, it provides good training sets, that allow the learner to learn quickly.
- **Protocol III:** Some random source (e.g. Nature) provides training examples; the teacher (Nature) provides the labels ($f(x)$)

[2]Monotone conjunctions, or conjunctions of non-negated variables, can be understood as functions $f(x)$ that do not decrease as x increases; in our case $f(x)$ goes from 0 to 1 and then stays at 1

[2]The teacher knows the hidden conjunction function

1.1.1 Protocol I

Since we know that this is a monotone conjunction, if the label for one training example is positive, then the variables in the conjunction function must be positive. Considering that the teacher knows what variables are active in the hidden function, we, as the learner, can query each variable at a time by setting it to 0 and the rest of the variables to 1. If the label for the query example is negative (0), we would know that the zeroed variable is necessary for the conjunction. Thus this straightforward algorithm requires $n = 100$ queries, and it will produce the hidden conjunction exactly as a result. In a general case, this protocol takes a minimum of n examples to learn the hidden conjunction where n is the number of possible variables in the hidden function.

Table 1: Queries for Protocol I

Query	Example	Valuation	Conclusion
Is x_{100} in?	$< (1, 1, 1, \ldots, 1, 0), ? >$	$f(x) = 0$	Yes
Is x_{99} in?	$< (1, 1, \ldots, 1, 0, 1), ? >$	$f(x) = 1$	No
$\ldots$			
Is x_1 in?	$< (0, 1, \ldots, 1, 1, 1), ? >$	$f(x) = 1$	No

The final result of the queries is that $h = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$. This protocol uncovers the hidden function for the data, but it requires a query for each variable.

1.1.2 Protocol II

First, we have to assume that the teacher has good intentions meaning that the teacher gives us the best examples possible for us to learn. We however assume the teacher does not collude with the learner and simply provide the function. This means that the teacher could give the learner a superset of the positive examples from which implications can be made. This way each variable that is in the hidden function is shown to be important. For example, the learner has the hypothesis $< (0, 1, 1, 1, 1, 0, \ldots, 0, 1), 1 >$, which is a superset of the good variables. But we are not done yet. The teacher still needs to show the learner that each and only the variables in the hidden conjunction are needed. Thus the teacher would provide negative examples from the superset of the good variables:

Table 2: Examples Given by the Teacher for Protocol II

Example	Conclusion
$< (0, 1, 1, 1, 1, 0, \ldots, 0, 1), 1 >$	$h \supset \bigwedge_{x \in A} A = \{x_2, x_3, x_4, x_5, x_{100}\}$
$< (0, 0, 1, 1, 1, 0, \ldots, 0, 1), 0 >$	x_2 is needed
$< (0, 1, 0, 1, 1, 0, \ldots, 0, 1), 0 >$	x_3 is needed
$\ldots$	
$< (0, 1, 1, 1, 1, 0, \ldots, 0, 0), 0 >$	x_{100} is needed

Model Teaching is tricky because it involves choosing the right examples needed to get the learner to make accurate inferences. A straightforward algorithm requires $k = 6$ examples to produce the hidden conjunction exactly (one for finding superset of the positive examples and 5 for indicating

that each variable is important). $h = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$. In a general case, this protocol takes a minimum of k+1 examples to learn the hidden conjunction where k is the number of variables in the hidden conjunction.

1.1.3 Protocol III

This protocol is most often studied in machine learning, partially because it's more natural and is easier to analyze. In the this protocol, some random source provides training examples and a teacher provides the labels ($f(x)$). Some of the given examples can be as follows:

$$
\begin{gathered}
<(1,1,1,1,1,1,\ldots,1,1),1> \\
<(1,1,1,0,0,0,\ldots,0,0),0> \\
<(1,1,1,1,1,0,\ldots,0,1,1),1> \\
<(1,0,1,1,1,0,\ldots,0,1,1),0> \\
<(1,1,1,1,1,0,\ldots,0,0,1),1> \\
<(1,0,1,0,0,0,\ldots,0,1,1),0> \\
<(1,1,1,1,1,1,\ldots,0,1),1> \\
<(0,1,0,1,0,0,\ldots,0,1,1),0>
\end{gathered}
$$

One algorithm we can use in this protocol is elimination. Start with the set of all literals as candidates, if we see a positive example, and some of the literals are zeros in that example, we can conclude that those 0 literals are unimportant (given that our conjunction is monotone). These variables with value 0 can thus be eliminated. But we are not able to conclude anything when the example is negative.

Table 3: Queries for Protocol III

Example	Conclusion
$<(1,1,1,1,1,1,\ldots,1,1),1>$	$f = x_1 \wedge x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge \ldots \wedge x_{100}$
$<(1,1,1,0,0,0,\ldots,0,0),0>$	learned nothing
$<(1,1,1,1,1,0,\ldots,0,1,1),1>$	$f = x_1 \wedge x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{99} \wedge x_{100}$
$<(1,0,1,1,0,0,\ldots,0,0,1),0>$	learned nothing
$<(1,1,1,1,1,0,\ldots,0,0,1),1>$	$f = x_1 \wedge x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$

In this learning approach, each example, either modifies the hypothesis (by eliminating a variable in some positive examples) or doesn't change learner's hypothesis (negative examples). The key difference between this protocol and the previous two protocols is that the input in this protocol in random (naturally generated). This means that it is not guaranteed to learn the hidden conjunction exactly. Given that our target function is a conjunction, however, we can still say something meaningful about the behavior of the output. We cannot simply use the number of examples to determine the amount of time it takes before learning a satisfying function since there would be 2^{100} possible examples. We can analyze the time to learn something useful using two following approaches:

- **Probabilistic Intuition:** We consider the probability of one variable in the hidden conjunction never appearing in the example set. This probability is very small, so we can argue that

the learned concepts perform well on future data that is distributed similarly to our training data. This approach is a key idea of the Probably Approximately Correct (PAC) framework that we will go over later in the course[3].

- **Mistake Driven Learning:** We can say something important about the performance of our learned hypothesis whenever making a mistake on a given example. Intuitively, if we correct the hypothesis for the mistakenly classified example, we can assume that the modified hypothesis performs better on future data. Using this approach, we can focus on the number of mistakes we are going to make until we are satisfied with our hypothesis as a measure of performance. In this lecture, We will discuss the approach in an online setting.

It is important to note that not all online algorithms are mistake-driven. Some will update their hypothesis without necessarily waiting for a mistake. In this lecture, we will solely focus on online mistake driven algorithms.

1.2 On-line Learning

We want to develop an algorithm that weakly depend on the space dimentionality and strongly rely on small number of relevant attributes. This algorithm is useful in cases which we have a learning problem with a very high dimentionality space but relatively sparse function space (functions of interest depends on a small number of attributes). An example of such a domain is context sensitive spelling. For example, if all of the words of a sentence are considered to be features, the space is very large. In this space, only one word per sentence will represent the target concept and only a few of the words in the sentence will be relevant for disambiguation.

The question then becomes, how should the hypothesis be represented? We want to find a simple and intuitive model that makes the smallest number of mistakes in the long run.

The basic model for On-line Learning is below:

Model:

Instance space: X (dimensionality $- n$)

Target: $f : X \to \{0, 1\}, f \in C$ (where C is a concept class parameterized by n)

Protocol:

learner is given $x \in X$

learner predicts $h(x)$, and is then given $f(x)$ as feedback

Performance:

learner makes a mistake when $h(x) \neq f(x)$

[3] Lecture 5

Measure of number of mistakes algorithm A makes on sequence S of examples, for the target function f is defined as:

$$M_A(C) = max_{f \in C, S} M_A(f, S)$$

Where $M_A(f, S)$ is the number of mistakes algorithm A makes on sequence S of examples, for the target function f.

A is a *mistake bound algorithm* for the concept class C, if $M_A(C)$ is polynomial in n, the complexity parameter of the target concept.

1.2.1 Mistake Bound Learning

We want to know how many mistakes to get to $\epsilon - \delta$ (PAC) behavior; that is, how many mistakes before we can say there is a high possibility that the error rate is less than a small number. We can also look for exact learning: how many mistakes are made before the function stops making mistakes. This is easier to analyze. This view has the notable drawback of being too simple, as it's not clear when mistakes will be made, but the simplicity can also be advantageous.

1.2.2 Generic Mistake Bound Algorithms

It can be reasoned that we can bound the number of mistakes. Ideally, if we make a mistake, we update the hypothesis so that it will not make the same mistake again. However, most learning algorithms will not have this property, unless you do something special.

In the general case, let C be a finite concept class and we want to learn $f \in C$. Consider the following algorithm: consistency (CON):

The CON Algorithm

Let C be a finite concept class. Learn $f \in C$
for all i (steps of the algorithm) **do**
  C_i: all concepts in C consistent with all $i - 1$ previously seen examples
  Choose randomly $f_i \in C_i$ and use it to predict the next example
  If a mistake is made, remove f from C_i to obtain C_{i+1}
end for

It is clear that $C_{i+1} \subseteq C_i$, so that if a mistake is made on the ith example, then $|C_{i+1}| < |C_i|$. In this way, each time we make a mistake we remove a hypothesis and thus make progress toward learning the correct function. As defined above, CON makes at most $|C| - 1$ mistakes.

1.2.3 The Halving Algorithm

Generic Mistake Bound Algorithms makes at most $|C| - 1$ mistakes. The Halving Algorithm attempts to improve upon the Generic Mistake Bound Algorithms when it comes to bound number of mistakes.

In the general case, let C be a finite concept class and we want to learn $f \in C$. Consider the following algorithm:

The Halving Algorithm

Let C be a finite concept class. Learn $f \in C$
$C_0 \leftarrow C$
for all i (steps of the algorithm) **do**
  C_i: all concepts in C consistent with all $i-1$ previously seen examples
  Given an example e_t consider the value $f_j(e_t)$ for all $f_j \in C_i$ and predict by majority.
  if $|\{f_j \in C_i; f_j(e_t) = 0\}| < |\{f_j \in C_i; f_j(e_t) = 1\}|$ **then**
    Prediction = 1
  else
    Prediction = 0
  end if
  if Prediction $\neq f(e_t)$ **then**
    $C_{i+1} \leftarrow$ all elements of C_i that agrees with $f(x)$
  end if
  if C_i has one element left **then**
    Break
  end if
end for

Thus, since $C_{i+1} \in C_i$, when a mistake is made we can eliminate at least half of the concepts, $|C_{i+1}| < \frac{1}{2}|C_i|$. The Halving algorithm makes at most $log(|C|)$ mistakes.

The problem with the Halving algorithm is that it is hard to compute. When the class C is the class of all Boolean functions, then Halving is optimal. But, in general, to be optimal, instead of guessing in accordance with the majority of the valid concepts, we should guess according to the concept group that gives the least number of expected mistakes. This is, however, even harder to compute.

1.2.4 Learning Conjunctions

Remember the hidden conjunction from earlier:

$$f = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$$

The number of all conjunction is 3^n (since each variable can be positive, negative, or absent in the conjunction). As defined above, if we use the halving algorithm we expect to make at most $\log(|C|) = \log(3^n) = n$ mistakes when learning the target hidden function.

Now let's consider k-conjunctions. Assume that $k << n$ where n is the number of attributes and could be a large. The total number of k-conjunctions is $2^k C(n,k) \approx 2^k n^k$. In this case, $\log(|C|) = k \log n$.

1.3 Representation

Representation is a key concept in learning. Assume, for example, that we are interested in learning conjunctions. Should our hypothesis space be the set of conjunctions?

This representation will make the learning task really hard. Theorem: Given a sample on n attributes that is consistent with a conjunctive concept, it is NP-hard to find a pure conjunctive hypothesis that is both consistent with the sample and has the minimum number of attributes [4]. The same hardness holds true for disjunctions. In effect, if we want to learn conjunctions, we may not want to use the set of conjunctions as our hypothesis space.

Instead, we can efficiently learn the concept if we define our hypothesis space as a set of Linear Threshold Functions. In a more expressive class, the search for a good hypothesis sometimes becomes combinatorially easier.

1.3.1 Linear Threshold Functions

Functions that are in the form $f(x) = \text{sign}(w^T \cdot x - \theta) = \text{sign}\{\sum_{i=0}^{n} w_i x_i - \theta\}$. Many functions can be represented as linear threshold functions. Here are two examples with Boolean variables:

- Conjunctions: $y = x_1 \wedge x_3 \wedge x_5$ can be transformed into $y = \text{sign}\{x_1 + x_3 + x_5 - 3\} w = (1, 0, 1, 0, 1)\theta = 3$
- At least m of n: at least 2 of $\{x_1, x_3, x_5\}$ can be transformed into $y = \text{sign}\{x_1 + x_3 + x_5 - 2\} w = (1, 0, 1, 0, 1)\theta = 2$

Although many functions are linear, there are also many that are not (Some can be made into a linear function):

- Xor: $y = (x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2)$ is not linear but can be linearized
- Non trivial DNF: $y = (x_1 \wedge x_2) \vee (x_3 \wedge x_4)$ is not linear but can be linearized

1.3.2 Canonical Representation

As we showed we can learn many hidden functions using Linear Threshold Functions efficiently. After achieving a linear function in the form $f(x) = \text{sign}(w^T \cdot x - \theta)$, we can do the following transformation:

$$\text{sign}(w^T \cdot x - \theta) = \text{sign}(w'^T \cdot x')$$

Where $x' = (x, -1)$ and $w' = (w, \theta)$. This way we move from an n dimensional representation to an $(n + 1)$ dimensional representation with the hyperplane crossing trough the origin. We are now able to learn both w and θ.

[4] [David Haussler, AIJ'88: "Quantifying Inductive Bias: AI Learning Algorithms and Valiant's Learning Framework"]

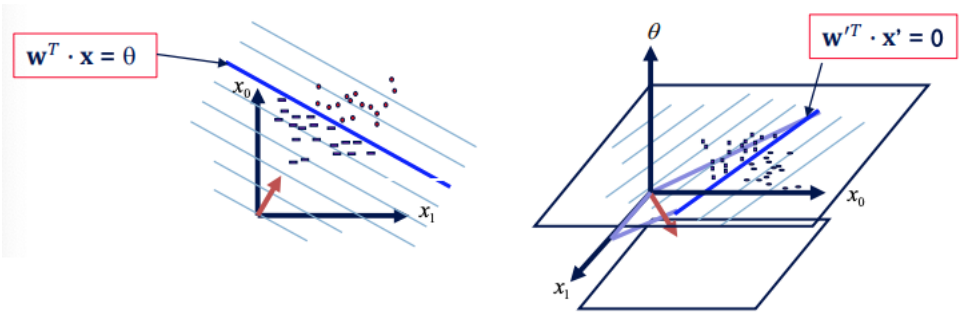


Figure 1: Conical Representation of linear threshold function

2 Perceptron

The perceptron is an on-line and mistake driven algorithm first proposed by the American psychologist Frank Rosenblatt. Almost all the machine learning algorithms we learn in this lecture are a variation of the perceptron. Given an example, the perceptron tries to predict the label using the current hypothesis. If the prediction is correct, then the algorithm does nothing; otherwise, it corrects the hypothesis.

In the literature, perceptron and linear threshold unit are often conflated. A linear threshold unit, as shown in Figure 2, takes inputs X , assigns weights W , and applies a threshold to produce label, y. In this class we'll treat perceptron as one algorithm among many that learns this architecture.

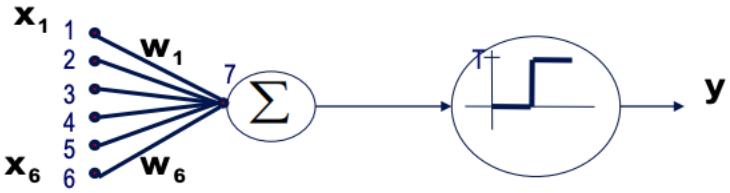


Figure 2: Perceptron as a Linear Threshold Unit

2.1 Learning Rule/Algorithm

Given labeled examples in form below:

$$\{\{x_1, y_1\}, \{x_2, y_2\}, ..., \{x_m, y_m\}\}$$

We are seeking to learn the hidden function f such that $f : X \rightarrow Y = \{-1, +1\}$ where f is represented as $f = \text{sign}\{w^T \cdot x\}$ in canonical form. Perceptron learning rule follows the following algorithm:

Perceptron Algorithm

Initialize $w \in R^n$
for all s (steps of the algorithm) **do**
  for all i (examples) **do**
    Predict the label of instance x_i to be $y = \text{sign}\{w^T \cdot x\}$ (either 1 or -1)
    if $y' \neq y$ **then**
      Update the weight vector: $w = w + ryx$ where r is a constant pre-decided learning rate
    end if
  end for
end for

The intuition in the perceptron's updating rule is that if the true label of an example is positive and we predicted negative for that example, then we are adding a proportional value to the weight vector. Similarly, if the true label is negative, we are subtracting a proportional value from the weight vector.

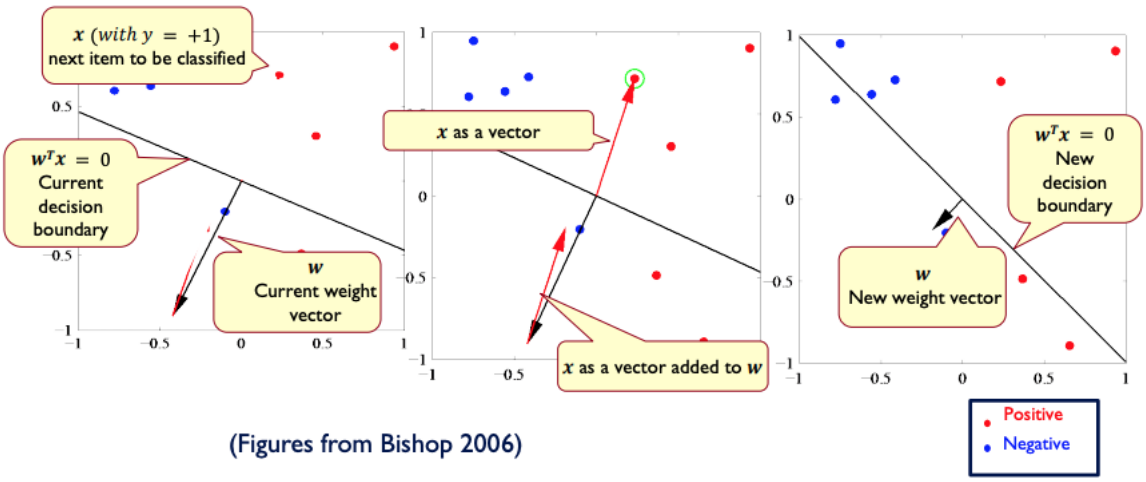


Figure 3: Perceptron's updating rule in action from left to right

Figure 3 gives us some intuition on how the updating rule works in action. In the figure, the direction of the arrow representing w indicates the positive classification of the algorithm. The red arrow in the middle sub-figure indicates the error vector ryx for the shown misclassified red example. The error vector thus updates the weight vector so that we improve the performance for the misclassified example. It is important to note that even after the update we are not classifying the red example correctly. However, if our classification task is linearly separable we will eventually converge to classify all of the examples correctly.;

If it is given that x is Boolean, the update rule will only update the weights of the active features. This is important since it brings computational efficiency when the dimensionality of the hypothesis space is really large. This allows us to only update the weights of the active features.

It is important to notice that for our binary classification $w \cdot x > 0$ which leads to a positive prediction has the equivalent form:

$$\frac{1}{1 + e^{(-w^T \cdot x)}} > \frac{1}{2}$$

The form above is using a logistic function that has values between 0 and 1. This allows us to have a nice probabilistic interpretation of how we select a specific class when it comes to predicting. You can see the a plot of logistic function in figure 4.

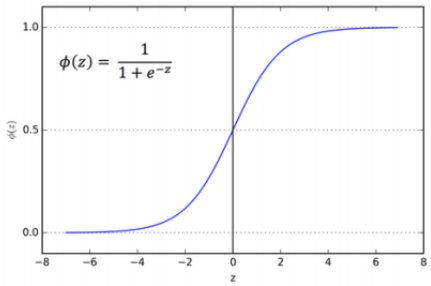


Figure 4: Logistic function

2.2 Learnability

Perceptron can only learn linearly separable functions. Minsky and Papert (1969) wrote an influential book demonstrating perceptron's representational limitations (ie. parity functions like XOR can't be learned; in vision – if patterns are represented with local features – perceptron can't represent symmetry and connectivity).
In 1959, Rosenblatt himself asked

> "What pattern recognition[5] problems can be transformed so as to become linearly separable?"

Rosenblatt had the perspective that if we have a very complex function, and the process for learning it is unknown, we can transform the feature space such that the data becomes linearly separable, as shown in Figure 5.

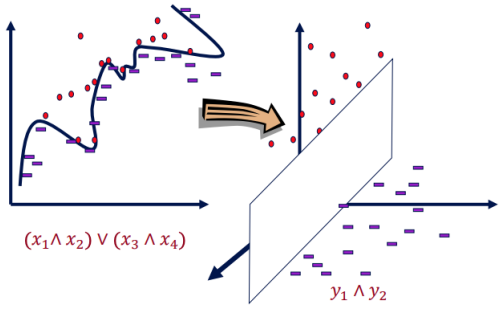


Figure 5: Transformed feature space

[5]Machine learning used to be called pattern recognition

There are two important considerations when thinking about perceptron learnability, as given by the following theorems.

- **Perceptron Convergence Theorem:** If there exist a set of weights that are consistent with the data (ie. the data is linearly separable), the perceptron learning algorithm will converge
- **Perceptron Cycling Theorem:** If the training data is not linearly separable, the perceptron learning algorithm will eventually repeat the same set of weights and therefore enter an infinite loop.

2.3 Mistake Bound

Assume we have a weight vector $w \in R^n$ where $w_0 = (0, ..., 0)$. Upon receiving an example $x \in R^n$, we predict according to a linear threshold function ($w^T \cdot x \geq 0$).

Using Novikoff's[6] proof we have the following theorem:

> "Let $(x_1, y_1), ..., (x_t, y_t)$ be a sequence of labeled examples with $x_i \in R^n$, $||x_i|| \leq R$ and $y_i \in \{-1, 1\}$ for all i. Let $u \in R^n$, $\gamma > 0$ be such that, $||u|| = 1$ and $y_i u^T \cdot x_i \geq \gamma$ for all i. Then Perceptron makes at most $\frac{R^2}{\gamma^2}$ mistakes on this example sequence"

This theorem assumes that all examples are bounded by some value R. R would be at least as large as the largest x_i. The theorem further assumes that there exists some u that separates the data (that is, the data is linearly separable). $||u||$ requires to be a constant that could be arbitrarily scaled.

Finally, the theorem assumes that there exists some γ such that the inequality $y_i u^T \cdot x_i \geq \gamma$ is satisfied for all examples. If the data is linearly separable, the closest point to the hyperplane is γ, and thus this assumption always holds for linearly separable data.

We refer to γ as the complexity parameter, which indicates the difficulty of a learning problem. If γ is small, positive and negative examples are very close and it is difficult to define a hyperplane. A large γ on the other hand, indicates finding a hyperplane is much easier. Thus, it makes sense to measure the difficulty (i.e. the number of mistakes) using the complexity parameter. Please notice that the bound does not depend on the dimensionality nor on the number of examples.

Proof
Let v_k be the be the hypothesis before the k^{th} mistake. Assume that the k^{th} mistake occurs on the input example (x_i, y_i).

$$\therefore y_i(v_k \cdot x_i) \leq 0$$

Then we have:

[6] 1962

$$
\begin{aligned}
v_{k+1} &= v_k + y_i x_i \\
v_{k+1} \cdot u &= v_k \cdot u + y_i (u \cdot x_i) \\
&\geq v_k \cdot u + \gamma \\
\therefore v_{k+1} \cdot u &\geq k\gamma
\end{aligned} \tag{1}
$$

$$
\begin{aligned}
||v_{k+1}||^2 &= ||v_k||^2 + 2y_i(v_k \cdot x_i) + ||x_i||^2 \\
&\leq ||v_k||^2 + R^2 \\
||v_{k+1}||^2 &\leq kR^2
\end{aligned} \tag{2}
$$

Therefore from (1) and (2) we have:

$$
\begin{aligned}
\sqrt{k}R &\geq ||v_{k+1}|| \geq v_{k+1} \cdot u \geq k\gamma \\
\frac{R^2}{\gamma^2} &> k
\end{aligned}
$$

2.4 Robustness to Noise

One important relaxation modification to the perceptron algorithm allows us to be more robust to noise. Consider the case of non-linearly separable data, as in Figure 7. Here, we do not have a margin of γ that separates the data.

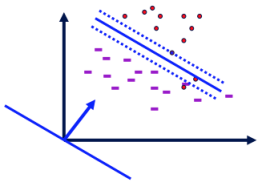


Figure 6: Non-linearly separable case

To solve this, we can use a slack variable, ξ defined as

$$\xi_i = \max(0, \gamma - y_i w \cdot x_i)$$

Intuitively, when $\xi_i = 0$, example x_i is on the correct side of the hyperplane with at least γ distance to the plane, otherwise, the cost grows linearly with $-y_i w \cdot x_i$. Perceptron is expected to have some robustness to noise.

Now if we denote D_2 as following

$$D_2 = [\sum \{\xi_i^2\}]^{\frac{1}{2}}$$

We can refine the mistake bound theorem for Perceptron. In the case that the data is not linearly seprable, Perceptron is guaranteed to make no more than $(\frac{R+D_2}{\gamma})^2$ mistakes on any sequence of examples satisfying $||x_i||^2 < R$.

3 Winnow

Winnow is another on-line and mistake-driven learning algorithm similar to perceptron. The Winnow algorithm uses the same classification scheme as the Perceptron but the Winnow algorithm differs from the Perceptron in its update scheme. Winnow uses a multiplicative approach to updating the weights at each learning iteration.

3.1 Learning Rule/Algorithm

Given labeled examples in form below:

$$\{\{x_1, y_1\}, \{x_2, y_2\}, ..., \{x_m, y_m\}\}$$

We are seeking to learn the hidden function f such that $f : X \rightarrow Y = \{-1, +1\}$ where f is represented as $f = \text{sign}\{w^T \cdot x\}$ in canonical form. The Winnow's learning rule follows the following algorithm:

Winnow Algorithm

Initialize $w = \vec{1}$
for all s (steps of the algorithm) **do**
  for all i (examples) **do**
    Predict the label of instance: **if** $\{w^T \cdot x > 0.5\}$ $y = 1$ **else** $y = -1$
    if $y = 1$ but $y' = -1$ **then**
      for all j (features) **do**
        if $x_j = 1$ **then**
          Promote the weight in the weight vector according to $w_j = 2w_j$
        end if
      end for
    end if
    if $y = -1$ but $y' = 1$ **then**
      for all j (features) **do**
        if $x_j = 1$ **then**
          Demote the weight in the weight vector according to $w_j = \frac{w_j}{2}$
        end if
      end for
    end if
  end for
end for

In principle Winnow is similar to perceptron – increase the weights on positive mistakes, decrease

on negative mistakes – but winnow does this so multiplicatively. Additionally, when learning disjunctions winnow can use elimination rather than demotion; instead of dividing w_i by 2, it can just be set to 0.

3.2 Mistake Bound

In certain circumstances, it can be shown that the Winnow algorithm mistake bound is bounded independently of the number of instances with which it is presented. If the target function that we are after is a k-disjunction on n attributes shown below:

$$f(x) = x_1 \vee x_2 \vee ... \vee x_n$$

Then Nick Littlestone[7] showed that the Winnow algorithm's mistake bound for a sequence of examples is upper bounded by $O(k \log n)$. This is an improvement over the Perceptron algorithm that makes at most $O(n)$ mistakes on the same sequence of examples.

4 Perceptron Practical Issues and Extensions

There are many extensions that can be applied to these basic algorithms(Perceptron and Winnow) to improve their performance. Some improvements are necessary (eg. regularization), and some are for improving the ease of use and tuning. For example, we can convert the output of a Perceptron/Winnow to a conditional probability $P(y = 1|x) = \frac{1}{1+e^{(-Awx)}}$ where A is a parameter that can be tuned on the development set.

There are some other issues such as multiclass classification and infinite attribute domain that we will talk about later in the course.

4.1 Regularization Via Averaged Perceptron

In on-line mistake driven algorithms, we generate a new hypothesis every time a mistake is made. It is reasonable to choose the last hypothesis as the learned output (final hypothesis). However, consider a case where a mistake was made on the penultimate example. That hypothesis survived all prior examples before making a mistake (on the penultimate example), and is thus more reliable than the final hypothesis, which has only seen one example. This intuition doesn't come naturally in mistake-bound algorithms, but it is what drives the PAC model that we'll discuss later in the course. In the PAC model, the performance of the output depends on the number examples rather than the number of mistakes, which can yield global guarantees on performance.

Averaged Perceptron Algorithm tries to address this issue. In general Averaged Perceptron Algorithm is motivated by the following considerations:

- In real life, we want [more] guarantees from our learning algorithm

[7] 1988

- In the mistake bound model we don't know when we will make a mistake. We also don't know what hypothesis we land on in a sequential/on-line scenario
- Ideally, we want to quantify the expected performance as a function of the number of examples seen and not number of mistakes. ("a global guarantee"; the PAC model does that)
- Every Mistake-Bound Algorithm can be converted efficiently to a PAC algorithm – to yield global guarantees on performance

In general To convert a given mistake bound algorithm (into a global guarantee algorithm) we need to wait for a long stretch without mistakes (there must be one). when found, use the hypothesis at the end of this stretch. Such algorithm's PAC behavior is relative to the length of the stretch.

Averaged Perceptron returns a weighted average of earlier hypotheses; the weights are a function of the length of no-mistakes stretch thus the algorithm offers global guarantees when it comes to the performance.

Let m be the number of examples, k the number of mistakes, c_i be the consistency count for hypothesis v_i; on how many examples did v_i make no mistakes. Given a labeled traning set $(x_1, y_1), ..., (x_m, y_m)$, we want to produce a list of weighted perceptrons $(v_1, c_1), ..., (v_k, c_k)$. We have the following for Averaged Perceptron algorithm:

Averaged Perceptron Algorithm

Initialize $k = 0, v_1 = 0, c_1 = 0$
for all s (steps of the algorithm) **do**
  for all i (examples) **do**
    Predict the label of instance x_i to be $y' = \text{sign}\{v_k^T \cdot x_i\}$ (either 1 or -1)
    if $y' \neq y$ **then**
      $c_{k+1} = c_k + 1$
      $v_{k+1} = v_k + y_i x_i$
      $k = k + 1$
    else
      $c_k = c_k + 1$
    end if
  end for
end for

Having produced this collection of weighted perceptrons – $(v_1, c_1), ..., (v_k, c_k)$, we can now predict the label of new example x_n by calculating:

$$y'(x) = \text{sign}[\sum_{i=1}^{k} c_i(v_i \cdot x_n)]$$

4.2 Perceptron with Margin

Perceptron with Margin is also known as Thick Separator. It is an extension to both Perceptron and Winnow algorithms. It is a method to improve generality of our learned hypothesis on unseen data.

When there is a clear margin between positive and negative examples, we are able to choose any hyperplane within the range of the margin to correctly classify all examples correctly. But intuitively we want our hyperplane to be as further away from the closest two postive and negative examples so that we have more room to be wrong for prediction of future unseen examples (we have higher tolerance to noise). We will cover why this is the case more rigorously later in the course. Figure 7 indicates such case where there is some margin between positive and negative examples.

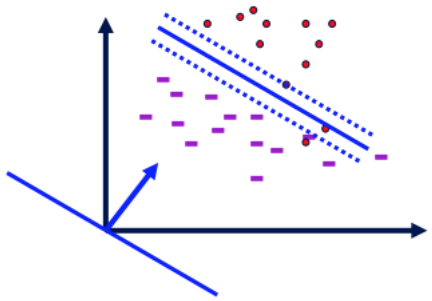


Figure 7: Non-linearly separable case

To achieve the hyperplane described above, we enforce some margin γ for all examples x_i such that we (in Canonical Representation):

- Promote the weights if $w^T.x_i < \gamma$
- Demote the weights if $w^T.x_i > \gamma$

Note that γ here is a functional margin. Its effect could disappear as w grows. Nevertheless, this has been shown to be a very effective algorithmic addition.

4.3 Other Extensions

There are many more variations to basic Perceptron and Winnow algorithm.

4.3.1 Aggressive Perceptron

Aggressive Perceptron is a variation of Perceptron algorithm that updates the weight vector relatively when the algorithm makes a mistake. Assume we've run Perceptron and we've made a mistake on an example, as shown in the left image of Figure 8. Consider that, at some point in the future, we see this example again. We are not guaranteed to classify this example correctly because our step size may be too small to have changed the weight vector adequately.

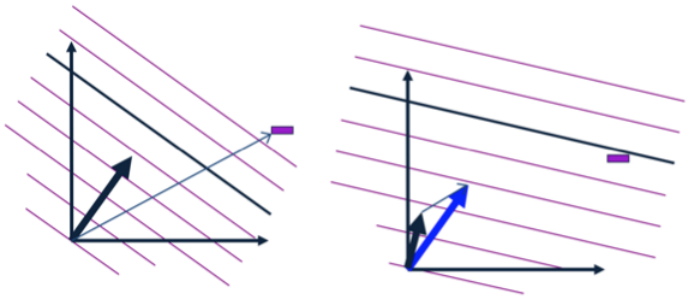


Figure 8: Threshold relative updating

In order to avoid making a mistake on the same example, then, we could either feed an example enough number of times such that the weight vector is updated appropriately enough, or we can change the update step size (learning rate) on an example on which we made a mistake (x_m), according to

$$r = \frac{\theta - w^T x_m}{||x_m||^2}$$

This approach is equivalent to updating the weights on the same example multiple times.

4.3.2 LBJava

Several of mentioned extensions (and a couple more) are implemented in the LBJava learning architecture found in LBJava that supports several linear update rules (Winnow, Perceptron, Nave Bayes).

Download from: http://cogcomp.cs.illinois.edu/page/software

5 Stochastic Gradient Descent

Given examples $\{z = (x, y)\}_{1,m}$ from a distribution over $X \times Y$, we are trying to learn a linear function, parameterized by a weight vector w such that we minimize the expected risk function below:

$$J(w) = \mathbb{E}_z Q(z, w) \approx \frac{1}{m} \sum_{i=1}^{m} Q(z_i, w_i)$$

In Stochastic Gradient Descent(SGD) algorithms we approximate this minimization by incrementally updating the weight vector w as follows:

$$w_{t+1} = w_t - r_t g_w Q(z_t, w_t) = w_t - r_t g_t$$

Where $g_t = g_w Q(z_t, w_t)$ is the gradient of loss function Q with respect to w at time t.

With the use of loss functions in the update rule of the SGD algorithm, we can see that different SGD algorithms only vary in the choice of various loss functions $Q(z, w)$. The choice of the loss function can be dependent on the learning task in hand.

5.1 Loss Functions

5.1.1 Least Mean Squares (LMS)

Least Mean Squares (LMS) loss function $Q((x, y), w)$ is defined as:

$$Q((x, y), w) = \frac{1}{2}(y - w^T x)^2$$

Then we will have the following for the LMS loss update rule (also called Windrow's Adaline):

$$w_{t+1} = w_t + r(y_i - w_t^T x_i)x_i$$

Note that even though we make binary predictions based on $\text{sign}(w^T x)$, we do not take the actual sign of the dot product into account in the loss function.

5.1.2 Hinge

Hinge loss function $Q((x, y), w)$ is defined as:

$$Q((x, y), w) = \max(0, 1 - yw^T x)$$

Then we will have the following for the hinge loss update rule for example (x_i, y_i) at time-step t:

SGD Hinge Loss Update rule

if $y_i w_i^T x_i \leq 1$ **then**
  $w_{t+1} = w_t + r y_i x_i$
end if

5.1.3 Logistic

Logistic loss function $Q((x, y), w)$ is defined as:

$$Q((x, y), w) = \log(1 + e^{-yw^T x})$$

Then we will have the following for the hinge loss update rule for example (x_i, y_i) at time-step t:

$$w_{t+1} = w_t + r\frac{e^{-y_i w_t^T x_i} y_i}{1 + e^{-y_i w_t^T x_i}} x_i$$

5.1.4 AdaGrad

Thus far all of our update rules focused on fixed learning rates. Imagine an algorithm that adapts its update based on historical information (previous examples); frequently occurring features get

small learning rates and infrequent features get higher ones. Intuitively, this algorithm would learn slowly from features that change a lot, but faster on those features that don't make frequent changes.
Adagrad is one such algorithm. It assigns a per-feature learning rate for feature j at time t, defined as:

$$r_{t,j} = \frac{r}{\sqrt{G_{t,j}}}$$

Where $G_{t,j} = \sum_{k=1}^{t} g_{k,j}^2$, or the sum of squares of gradients at feature j until time t. The update rule for Adagrad is then given by:

$$w_{t+1,j} = w_{t,j} - \frac{r g_{t,j}}{\sqrt{G_{t,j}}}$$

In practice this algorithm is supposed to update weights faster than Perceptron or SGD with LMS loss.

5.2 Regularization

One problem in theoretic machine learning is the need for regularization. In addition to the loss function, we add R, a *regularization term*, that is used to prevent our learned function from overfitting to the training data. Overfitting the model generally takes the form of making an overly complex model to explain idiosyncrasies in the data under study. Incorporating R, we now seek to minimize:

$$\frac{1}{m} \sum_{i=1}^{m} Q(z_i, w_i) + \lambda R_i(w_i)$$

Applying regularization to the above loss functions we have:

- LMS Loss:
  - Ridge Regression: $R(w) = ||w||_2^2$
  - LASSO problem: $R(w) = ||w||_1$
- Hinge Loss:
  - Support Vector Machines: $R(w) = ||w||_2^2$
- Logistic Loss:
  - Logistic Regression: $R(w) = ||w||_2^2$

It is important to understand why we enforce regularization through the size of w. In later lectures we will discuss why smaller weight vectors are preferable when it comes to generalization performance. As can be seen above, we use various norms to measure the size of vector w:

- $||w||_1$: The L1-norm calculated as $\sum_i |w_i|$, the sum of absolute values of entries of w.
- $||w||_2$: The L2-norm calculated as $\sqrt{\sum_i w_i^2}$, the square root of the sum of squares of entries of w.
- In the general case, the L-p norm is given by $||w||_p = (\sum_{i=1}^{n} |w_i|^p)^{\frac{1}{p}}$

6 Comparing Algorithms

All of the discussed algorithms do eventually converge to the similar solution for linearly separable data. It is still desirable to determine a measure of comparison; generalization (how many examples do we need to reach a certain performance), efficiency (how long does it take to learn the hypothesis), robustness to noise, adaptation to new domains, etc. Consider the following context-sensitive spelling example:

I don't know {whether, weather} *to laugh or cry*

To learn which whether,weather to use in the sentence, we must first define a feature space (using properties of the sentence) and then map the sentence to that space. Importantly, there are two steps in creating a feature representations, which can be framed using the following two questions:

1. What are available information sources? (sensors)
2. What kinds of features can be constructed from these sources? (functions)

In the context-sensitive spelling example, the sensors include the words themselves, their order, and their properties (part-of-speech, for example). These sensors must be combined with functions since they may not be sufficiently expressive. In our example, words on their own may not tell us enough to determine which of {whether, weather} to use, but conjunctions of pairs or triples of words may be sufficiently expressive (transformed to a linearly separable feature space that can be learned by the discussed algorithms). Compare the left plot in Figure 9 – representing words on their own – with the right plot, representing the feature space of functions over those words.

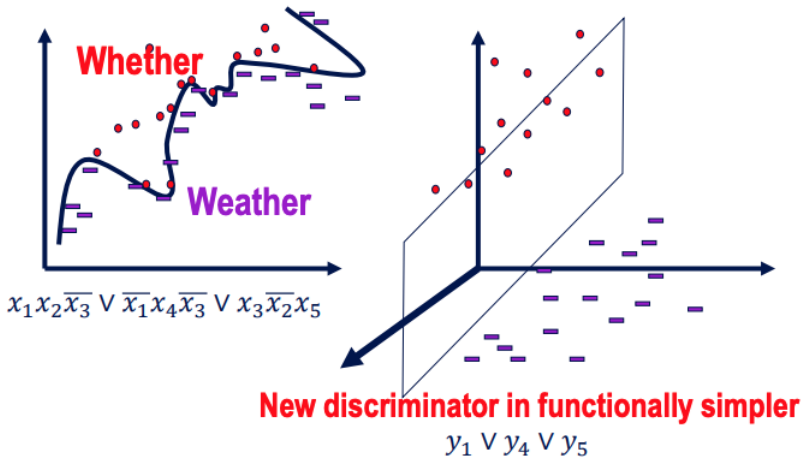


Figure 9: Combining sensors with proper functions for sufficient expressivity

6.1 Generalization

In mistake driven algorithms generalization is measured in terms of bounds of number of mistakes made during the learning process. In many learning scenarios, the number of potential features of the hypothesis space is quite large, but the instance space – the combination of features we observe – is sparse. Furthermore, decision boundaries usually depend on a small set of features in the instance space, making the function space sparse as well. In certain domains – like natural language processing – both the instance space and the function space is sparse, and it is, therefore, necessary to consider the impact of such sparsity in various learning tasks.

6.1.1 Effect of Sparsity

Generalization bounds are driven by sparsity. For multiplicative algorithms, like Winnow, the number of required examples1 largely depends on the number of relevant features, or the size of the target hyperplane: $||u||$). For additive learning algorithms like Perceptron, the number of required examples [8] largely depends on the number of relevant features of input examples $||x||$. This means that Perceptron is heavily dependent on the sparseness of feature space.

- Multiplicative Algorithms:

  Mistake bounds depend on the size of the separating hyperplane $||u||$. Given n, the number of features, and i the index of a given example. Mistake bound (number of mistaken examples), M_w (referring to multiplicative algorithms like Winnow) is given by:

  $$M_w = 2\ln(n)||u||_1^2 \frac{\max_i ||x^{(i)}||_\infty^2}{\min_i (u \cdot x^{(i)})^2}$$

  where $||x||_\infty = \max_i |x_i|$. Since $||x||_\infty$ is just the size of the largest example point, Multiplicative learning algorithms, then, do not care much about the data and rely instead on the L1-norm of the hyperplane, $||u||1$.

- Additive Algorithms:

  Additive algorithms, by contrast, care a lot about the data, as their mistake bounds, M_p (referring to additive algorithms like Perceptron) are given by:

  $$M_p = ||u||_2^2 \frac{\max_i ||x^{(i)}||_2^2}{\min_i (u \cdot x^{(i)})^2}$$

  additive algorithms thus rely on the L2-norm of X, making sparsity of feature space important for measuring their performance.

6.1.2 Extreme Examples

The distinction between multiplicative and additive algorithms is best seen through the extreme examples where their relative strengths and weaknesses are most prominent.

[8] Though we discuss this in later lectures, 'required examples' can be thought of as the number of examples the algorithm needs to see in order to produce a good hypothesis

- Extreme Scenario 1

  Assume the u has exactly k active features (all Boolean), and the other $n - k$ are zero. Only k input features are relevant to the prediction. We thus have:

$$\begin{aligned} ||u||_2 &= k^{\frac{1}{2}} \\ ||u||_1 &= k \\ \max ||x||_2 &= n^{\frac{1}{2}} \\ \max ||x||_\infty &= 1 \end{aligned} \tag{3}$$

  We can now compute the mistake bound for Perceptron as $M_p = kn$, while the bound for Winnow is given by $M_w = 2k^2 \ln(2n)$. Therefore, in cases where the number of active features is much smaller than the total number of features ($k << n$), Winnow requires far fewer examples to find the right hypothesis (logarithmic in n versus linear in n).

- Extreme Scenario 2

  Now assume that all the features are important ($u = (1, 1, 1, ..., 1)$), but the instances are very sparse (only one feature on). In this case the size (L1-norm) of u is n, but the size of the examples is 1:

$$\begin{aligned} ||u||_2 &= n^{\frac{1}{2}} \\ ||u||_1 &= n \\ \max ||x||_2 &= 1 \\ \max ||x||_\infty &= 1 \end{aligned} \tag{4}$$

  In this scenario, Perceptron is expected to have a much lower mistake bound than Winnow, given by $M_p = n$; $M_w = 2n^2 \ln 2n$.

- l of m of n

  Assume an example is labeled positive when l out of m features are on (out of n total features). These l of m of n functions are good representatives of linear threshold functions in general. A comparison between the Perceptron and Winnow mistake bounds for such functions is shown in Figure 10.

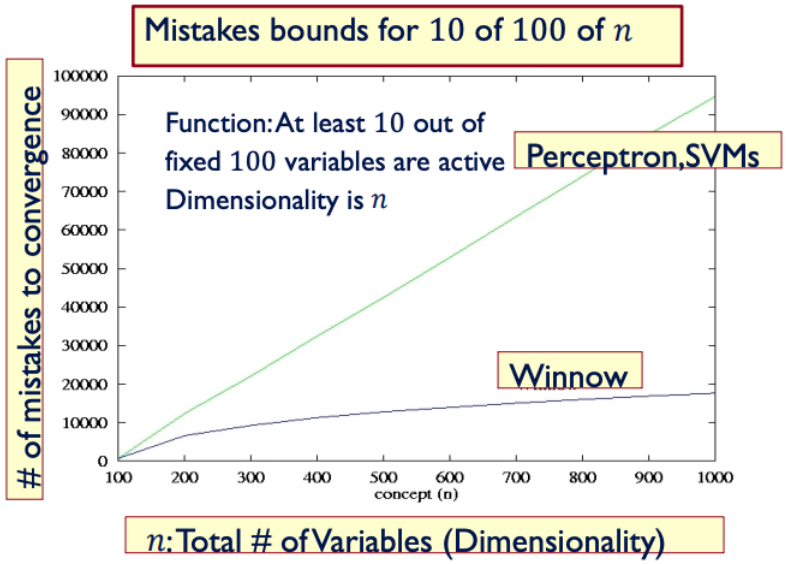
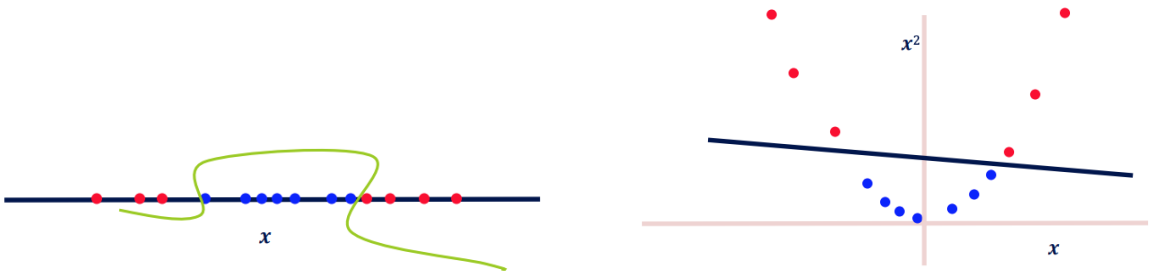


Figure 10: Comparison of Perceptron and Winnow's mistake bounds for l of m of n functions with increasing n

In l out of m out of n functions, the Perceptron mistake bound grows linearly, while the Winnow's bound grows with $log(n)$ as feature space becomes more and more sparse. In the limit, all algorithms behave in the same way. But the realistic scenario – that is, the one with a limited number of examples – requires that we consider which algorithms generalize better.

6.2 Efficiency

Efficiency of a learning algorithm is dependent on the size of the feature space. It is often the case that we don't use simple attributes, and instead treat functions over attributes (for example we use conjunction of feature 1 and 2 as one feature) as our features, making the learning task more difficult to learn efficiently. Consider the case shown in Figure 11(a), which is not linearly separable until the feature space is blown up, shown in Figure 11(b).

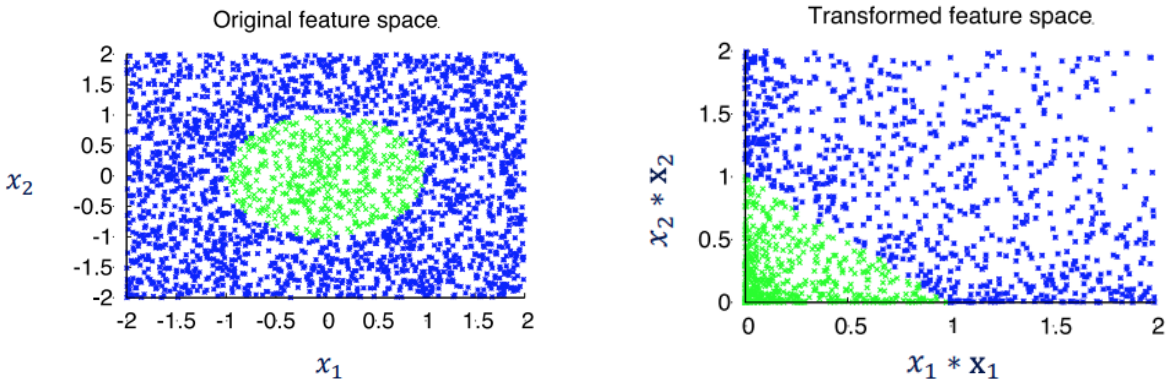


(a) Not Linearly separable in one dimension

(b) Linearly Separable by blow up dimension

Figure 11: Using functions over features to blow up the feature space

Consider that we would like to learn the function $f(x) = 1$ iff $x_1^2 + x_2^2 \leq 1$, shown in Figure 12(a)



(a) Not Linearly separable in original space

(b) Linearly separable in transformed space

Figure 12: Transforming feature space to allow for linear learnability

This data cannot be separated in the original two dimensional space. But if we transform the data to be $x_1' = x_1^2$ and $x_2' = x_2^2$, the data becomes linearly separable. By introducing these transformations, we made it possible to learn from linearly unlearnable feature space. During this process we also made the learning task less computationally efficient. We must consider how to learn efficiently, given our higher dimensional space. There is a solution to the computational efficiency issue that we will discuss now.

7 Kernel Trick

In additive algorithms we can behave as if we've generated complex features while still computing in the original feature space. This is known as the **Kernel Trick**.

7.1 Dual Representation

Many learning problems can be expressed using a dual formulation. Consider the Perceptron algorithm: given examples $x \in \{0,1\}^n$, hypothesis $w \in R^n$ and function $f(x) = \text{sign}(\sum_{i=1}^n w_i x_i(x))$ then we have the update rule below (assuming $r = 1$):

- If $y = 1$ but $w^T x^i \leq 0$ then $w \leftarrow w + x^i$ (Promotion)
- If $y = -1$ but $w^T x^i \geq 0$ then $w \leftarrow w - x^i$ (Demotion)

Assume we run the Perceptron algorithm with an initial w and we encounter the following examples: $(x^1, +1), (x^2, +1), (x^3, -1), (x^4, -1)$. Further assume that mistakes are made on x^1,x^2 and x^4. The resulting weight vector is given by $w = w + x^1 + x^2 x^4$; we made mistakes on positive examples x^1 and x^2 and negative example x^4. This is the heart of the dual representation. Because they share the same space, w can be expressed as a sum of examples on which we made mistakes, given by (α_i is the number of mistakes made on x^i):

$$w = \sum_{i=1}^{m} r\alpha_i y_i x^i$$

Where m is total number of unique examples we make mistakes on. Thus then we can rewrite $f(x_{new})$ over all examples as the following (x_{new} is the example we are trying to predict for):

$$\begin{aligned} f(x_{new}) = \text{sign}(w^T x_{new}) &= (\sum_{i=1}^{m} r\alpha_i y_i x^i)^T x_{new} \\ &= \sum_{i=1}^{m} r\alpha_i y_i ((x^i)^T x_{new}) \end{aligned} \tag{5}$$

7.2 Kernel Based Methods

Being able to represent a model in the dual space allows us to use Kernels. Kernels were popularized by SVMs [9], but many other algorithms can make use of them (to run in the dual space). Kernel based methods allow us to run Perceptron on a very large feature space, without incurring the cost of keeping the values of such a large weight vector. The idea is that we can compute the dot product $(w^T x)$ in the original feature space instead of the blown up feature space. It is important to note that this method pertains only to efficiency. The resulting classifier should be identical to the one you compute in the blown up feature space. In addition, generalization is still relative to that of the original dimensions.

Consider a setting in which we're interested in using the set of all conjunctions between features, The new space is the set of all monomials in this space, or 3^n (possibly $x_i, \neg x_i, 0$ in each position). We can refer to these monomials as $t_i(x)$, or the i^{th} monomial for x. Thus the new linear function is:

$$f(x) = \text{sign}(\sum_{i \in I} w_i t_i(x))$$

We can represent any Boolean function in this defined space. In this space, We can run Perceptron or Winow, but the convergence bound will suffer exponential growth. Consider that each mistake will make an additive contribution to w (either $+1$ or -1) iff $t(z) = 1$. Therefore, the value of w is actually determined by the number of mistakes on which $t()$ was satisfied.

To show this more formally, we now denote P as the set of examples on which we promoted, D to be the set of examples on which we demoted, and M as the set of all mistakes a.k.a. $P \cup D$. We have:

[9] We will talk about them in Lecture 6

$$
\begin{aligned}
f(x) &= \text{sign}(\sum_{i \in I}[\sum_{z \in P, t_i(z)=1} 1 - \sum_{z \in D, t_i(z)=1} 1]t_i(x)) \\
&= \text{sign}(\sum_{i \in I}[\sum_{z \in M} S(z)t_i(z)t_i(x)]) \\
&= \text{sign}(\sum_{i \in I} S(z) \sum_{z \in M} t_i(z)t_i(x))
\end{aligned}
$$

where $S(z) = 1$ if $z \in P$ and $S(z) = -1$ if $z \in D$. In the end, we only care about the sum of $t_i(z)t_i(x)$. The total contribution of z to the sum is equal to the number of monomials that satisfy both x and z. We define this new dot product as:

$$K(x, z) = \sum_{i \in I} t_i(z)t_i(x)$$

We call this the **kernel function** of x and z. Given this new dot product, we can transform the function into a standard notation below:

$$f(x) = \text{sign}(\sum_{z \in M} S(z)K(x, z))$$

We can view the kernel $K(x, z)$ as the distance between x, z in the t-space. I in the definition of our kernel is quite large. However, the kernel can be calculated in the original space, without explicitly writing the t-representation of x and z. Follow the example below:

- Monomial Example:

  Consider the space of all 3^n monominals (allowing both positive and negative literals), then:

  $$K(x, z) = \sum_{i \in I} t_i(z)t_i(x) = 2^{same(x,z)}$$

  where $same(x, z)$ calculates the number of features that have the same value for both x and z. Using this we can compute the dot product of two size 3^n vectors by looking at two vectors of size n. This is where the computational gain comes in (Kernel Trick).

  Assume, for example, that $n = 3$, where $x = (001)$, $z = (011)$. There are $3^3 = 27$ features in this blown up feature space (I). Here we know $same(x, z) = 2$ and in fact, only $\neg x_1, x_3, \neg x_1 \wedge x_3$ and $\emptyset$ are satisfying conjunction that $t_i(x)t_i(z) = 1$.

  We can state a more formal proof. Let $k = same(x, z)$. A monomial can only survive in two ways: choosing to include one of the k literals with the right polarity in the monomial (negate or not) or choosing to not include it at all. Monomials with literals outside this set disappear. Which means we would have 2^k conjunctions.

7.2.1 Implementation Example

- Dual Perception:

  Using the kernel based method, we now have an algorithm to run in the dual space. We run the standard Perceptron algorithms while keeping track of the set of mistakes M, which allows us to compute $S(z)$ at any step:

  $$f(x) = \text{sign}(\sum_{z \in M} S(z)K(x,z))$$

  where $K(x,z) = \sum_{i \in I} t_i(z)t_i(x)$. This way rather than remembering the weight vector w, we remember the set M (P and D as described earlier) – all those examples on which we made mistakes.

- Adding Polynomial Kernel:

  Using our knowledge of the kernel based methods we can separate some nonlinearly separable data by using a polynomial kernel. We predict with respect to a separating hyper planes w (produced by Perceptron, SVM) that can be computed instead, as a function of dot products of feature based representation of (some of) the examples. Given two examples $t = (t_1, t_2, ...t_n)$ and $r = (r_1, r_2, ...r_n)$, we want to map them to a high dimensional space. For example:

  $$\Phi(t_1, t_2, ...t_n) = (1, t_1, ..., t_n, t_1^2, ..., t_n^2, t_1 t_2 ...t_n)$$
  $$\Phi(r_1, r_2, ...r_n) = (1, r_1, ..., r_n, r_1^2, ..., r_n^2, r_1 r_2 ...r_n)$$

  computing the dot product $A = \Phi(t)^T \Phi(r)$ is going to be computationally inefficient since we blew up the feature space. Instead we can compute $B = K(t,r) = [1+(t_1, t_2, ...t_n)^T (r_1, r_2, ...r_n)]^2$. We can claim that $A = B$ in the learning process; though coefficients differ, the learning algorithm will adjust the coefficients accordingly.

7.2.2 General Conditions

A function $K(x,z)$ is a valid kernel **if** it corresponds to an inner product in some (perhaps infinite dimensional) feature space.

$$K(x,z) = \sum_{i \in I} t_i(x)t_i(z)$$

For example consider the following quadratic kernel:

$$\begin{aligned} K(x,z) &= (x_1z_1 + x_2z_2)^2 \\ &= x_1^2z_1^2 + 2x_1z_1x_2z_2 + x_2^2z_2^2 \\ &= (x_1^2, \sqrt{2}x_1x_2, x_2^2)(z_1^2, \sqrt{2}z_1z_2, z_2^2) \\ &= \Phi(x)^T\Phi(z) \end{aligned}$$

This shows that $K(x,z)$ is a valid kernel. It is not always necessary to explicitly show feature function Φ.

- Kernel Matrix:

  The Kernel Matrix is the Gramian matrix [10] of $\{\Phi(x_1), ..., \Phi(x_n)\}$. The size of the kernel matrix depends on the number of examples, not the dimensionality.

In general, we can show $K(x,z)$ is a valid kernel by simply showing that the Kernel matrix as described above is PSD (Positive Semi Definite).[11]

Here are some example kernels:

- Linear Kernel: $K(x,z) = xz$
- Polynomial Kernel of degree d: $K(x,z) = (xz)^d$
- Polynomial Kernel up to degree d: $K(x,z) = (xz + c)^d, (c > 0)$

7.2.3 Constructing New Kernels

By abiding to the following allowed operations you can construct new kernel $K'(x,z)$ from existing valid ones:

- Multiplying kernels by constants:

$$K'(x,z) = cK(x,z)$$

- Multiplying kernel $K(x,z)$ by a function f applied to x and z:

$$K'(x,z) = f(x)K(x,z)f(z)$$

- Applying a polynomial (with non-negative coefficients to $K(x,z)$):

$$K'(x,z) = P(K(x,z))$$

  with $P(z) = \sum_i a_i z^i, (a_i \geq 0)$

[10] The Gramian matrix of a set of n vectors $S = \{x_1, ..., x_n\}$ is the $n \times n$ matrix G with $G_{ij} = x_i x_j$

[11] In linear algebra, a symmetric $n \times n$ matrix M is said to be positive semi definite if the scalar $z^T Mz$ is non-negative for every non-zero column vector $z \in R^n$

- Exponentiating kernels:

$$K'(x,z) = e^{K(x,z)}$$

- Adding two kernels:

$$K'(x,z) = K_1(x,z) + K_2(x,z)$$

- Multiplying two kernels:

$$K'(x,z) = K_1(x,z)K_2(x,z)$$

- If $\Phi(x) \in R^m$ and $K(x,z)$ is a valid kernel in R^m then $K'(x,z) = K(\Phi(x), \Phi(z))$ is also a valid kernel.
- If A is a symmetric positive semi-definite matrix, $K'(x,z) = xAz$ is also a valid kernel.

7.2.4 Gaussian Kernel

Consider the Gaussian Kernel, given by:

$$K(x,z) = e^{\frac{-(x-z)^2}{c}}$$

Where $(x-z)^2$ is the squared Euclidean distance between x and z and $c = 2\sigma^2$ is a free parameter.

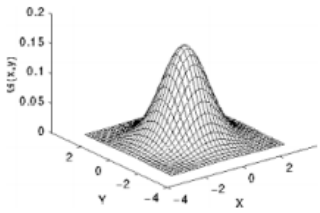


Figure 13: Gaussian Kernel

$K(x,z)$ can be thought of in terms of the distance between x and z; if x and z are very close, the value of the kernel is 1, and if they are very fall apart, the value is 0. We can also consider the property of c; a very small c means $k \approx I$ (identity matrix) (every item is different), and a very large c means all items are the same.

The Gaussian Kernel is a valid kernel because:

$$
\begin{aligned}
K(x,z) &= e^{\frac{-(x-z)^2}{2\sigma^2}} \\
&= e^{\frac{(-xx-zz+2xz)}{2\sigma^2}} \\
&= e^{\frac{-xx}{2\sigma^2}} e^{\frac{xz}{\sigma^2}} e^{\frac{-zz}{2\sigma^2}} \\
&= f(x) e^{\frac{xz}{\sigma^2}} f(z)
\end{aligned}
$$

$e^{\frac{xz}{\sigma^2}}$ is a valid kernel because xz is the linear kernel and we can multiply it by constant $\frac{1}{\sigma^2}$ and then exponetiaite it.

Unlike the discrete kernels discussed earlier, here you cannot easily explicitly blow up the feature space to get an identical representation (infinite dimensional kernel since it is not desecrate)

7.3 Generalization/Efficiency Trade-offs

There is a trade-off between the computational efficiency with which these kernels can be computed and the generalization ability of the classifier.

7.3.1 Computational Efficiency

For Perceptron, for example, consider using a polynomial kernel when you're unsure if the original space is expressive enough. If it turns out that the original space was expressive enough, however, the generalization will suffer because we're now unnecessarily working in an exponential space.

This example therefore shows us that we need to be careful when choosing to whether to use the dual or primal space. This decision depends on whether you have more examples or more features.

In general we know that:

- Dual space has $t_1 m^2$ computation time. t_1 is the size of dual representation feature space.
- Dual space has $t_2 m$ computation time. t_2 is the size of primal representation feature space.

Where m is the number of examples in both. Typically $t_1 << t_2$ because t_2 is the blown up space, so we need to compare the number of examples in proportion with the growth in dimensionality to determine which one is computationally more efficient. As a rough general rule of thumb, if we have a lot of examples, we prefer to stay in the primal space. In fact, most applications today use explicit kernels; that is, they blow up the feature space and work directly in that new space.

7.3.2 Generalization

Consider the case in which we want to move to the space of all combinations of three features. In many cases, most of these combinations will be irrelevant; you may only care about certain combinations. In this case, the most expressive kernel – a polynomial kernel of degree 3 – will

lead to overfitting.

Assume a linearly separable set of points $S = x_1, ..., x_n \in R^n$ with separator $w \in R^n$. We want to embed S into a higher dimensional space $n' > n$ by adding zero-mean random noise e to the additional dimensions. Then $w'\Delta x = (w, 0)\Delta(x, e) = w\Delta x$ which means $w' \in R^{n'}$ still separates S. Now we will look at $\frac{\gamma}{||x||}$ which we have shown to be inversely proportional to generalization (mistake bound):

$$\begin{aligned}\frac{\gamma(S, w')}{||x'||} &= \frac{\min_s(w'^T x')}{||w'||||x'||} \\ &= \frac{\min_s(w^T x)}{||w||||x'||} \\ &< \frac{\gamma(S, w)}{||x||}\end{aligned}$$

Since

$$||x'|| = ||(x, e)|| > ||x||$$

We can see we have a larger ratio, which means generalization suffers. In essence, adding a lot of noisy/irrelevant features cannot help.

8 Multi-Layer Neural Network

Multi-layer network were designed to overcome the computational (expressivity) limitation of a single threshold element. The idea is to stack several layers (the middle layers are called hidden layers) of threshold elements, each layer uses the output of the previous layer as input. Each layer has two parts:

- Linear Unit (Similar to Perceptron)
- Activation Function

Multi-layer networks can represent many arbitrary functions, but building effective learning methods for such networks was [thought to be] difficult. Figure 14 shows a three layer Neural Network.

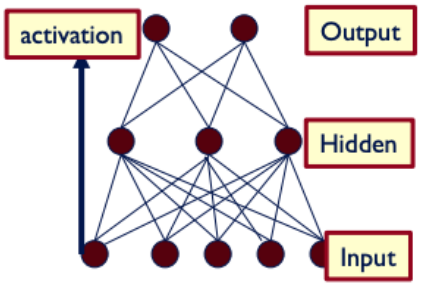


Figure 14: Three Layer Neural Network

8.1 Activation Function/ Model Neuron (Logistic)

Non linearity needs to be introduced into our stacking of multiple linear layer approach so that we can express and learn nonlinear functions. The nonlinearity needs to facilitate learning. One of the best way to approach learning is through gradient updating of each layer's weights (as we saw earlier with SGD approach). This means the non-linearity that we define between each layer needs to be differentiable. For example, $\text{sign}()$ would not be suitable since it is not a differentiable function.

Instead we use a non-linear, differentiable output function such as the sigmoid or logistic function after each linear unit. This means if:

$$net_j = \sum_i w_{ij} \cdot x_i$$

is the linear output of one layer, $o_j = \frac{1}{1+e^{-(net_j - bias_j)}}$ is going to be the input to the next layer of the multiple layer network.