CS 6350 Midterm Review

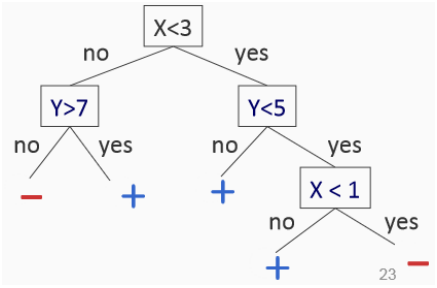
General Supervised Learning

- Supervised learning, instance spaces, label spaces, concept and hypothesis spaces
  - **Supervised learning:**
    - Given some training examples in the form $\langle x, f(x) \rangle$, with f being unknown
    - Typically, the input x is represented in the *feature space*, for example $x \in \{0,1\}^n$ or $x \in \mathbb{R}^n$
    - For a training example x, the value of $f(x)$ is called its *label*
    - Goal: Find a good approximation for f. Aimes to decide: an instance space, determine the instance space's label space, the necessary hypothesis space to do so, and evaluate uncertainty.
  - **Instance spaces:** The set of the examples and features that are going to be looked at. Often elements of the instance space are **feature vectors**.
  - **Label spaces:** The total set of possible labels that each instance can have
  - **Concept and Hypothesis spaces:**
    - **Hypothesis space:** Set of functions that the learning algorithm is going to be searching over. Categorises a certain class with specific constraints and attributes.
    - **Concept space:** Set of functions from which the classifier originates. Concept space is a set of functions from which the true classifier (also known as oracle) originates. The concept space contains the target classifier that is hidden from us.
- Understanding why we need to restrict hypothesis spaces
  - Choose a hypothesis space that is smaller than the space of all functions. The functions that are choosen are done by either prior knowledge or by guessing. It also needs to be flexible enough to work with the data and not too small that nothing agrees with it.
  - For example, in the case of boolean functions, can do only *simple conjunctions*, pick *m-of-n rules* where you pick a set of n variables, of which at least m need to be true, linear functions, etc.
  - At times we are able to "count functions" within our restricted hypothesis space. In the case of booleans we have a total of 2^{2^n} functions. We can place tighter bounds on the order of our hypothesis space when considering only simple conjunctions or m-of-n rules.
- General issues in supervised learning: hypothesis spaces, representation (i.e. features), learning algorithms
  - **Hypothesis spaces:** If the hypothesis space is too large, then learning can not be done because there are too many functions to search over. Therefore, to be able to learn, the hypothesis space must be restricted to a smaller subset so that it is possible to learn.
  - **Representation/features:** Need features that represent the data well. Features that aren't present in a lot of the data *or* have too common of a value are not good features.
  - **Learning algorithms:** The right learning algorithm needs to be choosen such that it can learn from the data well and not be excessive in resource usage. Also you want an algorithm that doesn't overfit the data and has a high success rate.

Decision Trees

- What is a decision tree? What can they represent?
  - **Decision tree:** A *hierarchical data structure* that represents data using a divide-and-conquor strategy. It can be used as a hypothesis class for non-parametric classification or regression.
    General Idea: Given a collection of examples, learn a decision tree that represents it.
  - Decision trees are a family of classifiers for instances that are represented by feature vectors (i.e. vectors of attributes)
  - Decision trees are built based on a *greedy hueristic*
  - *Nodes* are tests for feature vectors
  - There is one *branch* for every value that the feature can take
  - *Leaves* of the tree specify the class labels
  - Decision trees *can represent* all boolean functions. Decision trees are often used with discretely labeled instances.

- How to predict with a decision tree
  - At the root node, you're given a test, you then follow the branch for the correct answer for that one node. Decision trees need not be binary!
- Expressivity, counting the number of decision trees
  - **Expressivity:**
- Dealing with continuous features
  - You would have ranges of values that fall in to each node. For example:



  - If features are continuous, say irrational numbers, we must apply a discrete feature representation for our data.
- Learning Algorithm: The ID3 algorithm entropy information gain
  - The ID3 Algorithm is based on the *entropy* of each attribute
    ID3(S, `Attributes`, `Label`):
    1. **If** all examples have the same label:
       Return a single node tree with the label
    2. **Else:**
       (a) Create a `Root Node` for the tree
       (b) `A` = attribute in `Attributes` that *best* classifies S
       (c) **for each** possible value ν that `A` can take:
          i. Add a new tree branch corresponding to `A` $= \nu$
          ii. Let S_ν be the subset of examples in S with `A` $= \nu$
          iii. **If** $S_\nu \in \varnothing$:
             Add leaf node with the common value of `Label` in S
             Else:
             Below this branch add the subtree `ID3`($S\nu$,`Attributes`$-\{$`A`$\}$,`Label`)
       (d) **Return** `Root Node`
  - **Entropy and Information Gain:**
    * *Entropy* is the set of examples S with respect to binary classification is

$$Entropy(S) = H(S) = -p_+ \log_2(p_+) - p_- \log_2(p_-) \qquad \begin{cases} p_+ \text{ is the porportion of positive examples} \\ p_- \text{ is the proportion of negative examples} \end{cases}$$

$$Gain(S, A) = Entropy(S) - \sum_{\nu \in \text{Values}} \frac{|S_\nu|}{|S|} Entropy(S_\nu)$$

S_ν : The subset of examples where the value of attribute A is set to value ν

  - The root attribute that should be choosen is the attribute with the highest information gain
  - Information gain allows us to split over given examples so they are *relatively pure in one label*. By splitting such that there is a reduction in entropy there is less uncertainty when labeling and a more structured partitioning of labels.
- Overfitting (applicable not just to decision trees) and how to deal with it when training decision trees
  - Consider some arbitrary function in the hypothesis space h'. With data coming from proability distribution D a classifier h can be considered overfit if :

    - $error_{train}(h) < error_{train}(h')$
    - $error_D(h) > error_D(h')$
  - The learning algorithm fits the noise in the data. Irrelevant attribuytes or noisy examples influence the choice of the hypothesis
  - May lead to poor performance on future examples
  - Decision trees are notorous for overfitting, so the solution to this is to favor simpler (shorter) hypotheses as fewer shorter trees are less likely to fit better by coicidence
  - *Held-out-set* method is means to avoid over fitting by first setting a random sample of the training data aside then at every layer when building the decision tree test the current tree's performance on the held out set, if the performance drops stop growing the tree. The restricted height in theory should aide to avoid over fitting. This can be considered as pruning the tree greedily with a bottom up approach.

- Dealing with missing features
  - Using the most common value oif the attribute in the data
  - Using the most common value of the attribute among all examples with the same output
  - Using fractional counts of all the attributes and stochastically choosing a labeling when deciding with probaility equal to the fractional count.
  - *Test time*: Use the same method

- When to use decision trees
  - Binary classifications? Small hypothesis spaces?

Nearest Neighbors

- Instance based learning. How to predict? Importance of representation
  - Training examples are vectors $\mathbf{x}_i$ associated with a label y_i
  - Since instance based, or "memory based" nearest neighbor is an expensive approach.
  - *Learning*: Just store all the training examples
  - *Prediction*: For a new example $\mathbf{x}$, find the training example $\mathbf{x}_i$ that is *closest* to $\mathbf{x}$ and predict the label of $\mathbf{x}$ with the label y_i associated with $\mathbf{x}_i$.
    - *Classification*: Every neighbor votes on the label. Predict the most frequent label among the neighbors.
    - *Regression*: Predict the mean value

- Different definitions of distance
  - Euclidean Distance

$$||\mathbf{x}_1 - \mathbf{x}_2||_2 = \sqrt{\sum_{i=1}^{n} (\mathbf{x}_{1,i} - \mathbf{x}_{2,i})^2}$$

  - Manhattan Distance

$$||\mathbf{x}_1 - \mathbf{x}_2||_1 = \sum_{i=1}^{n} |\mathbf{x}_{1,i} - \mathbf{x}_{2,i}|$$

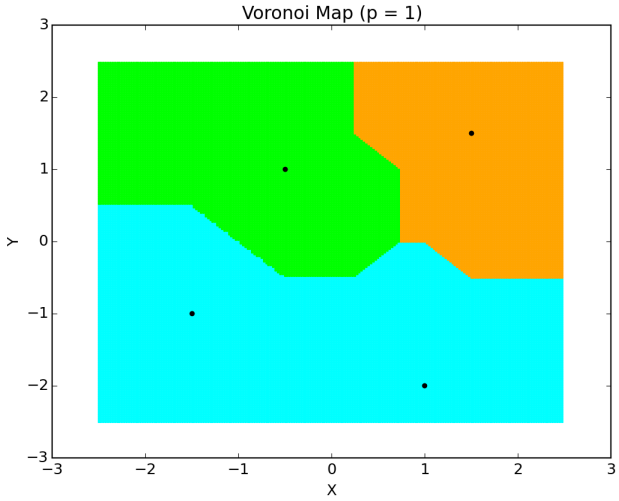
  - L_p-Norm

$$||\mathbf{x}_1 - \mathbf{x}_2||_p = \left(\sum_{i=1}^{n} |\mathbf{x}_{1,i} - \mathbf{x}_{2,i}|^p \right)^{\frac{1}{p}}$$

- Dealing with symbolic features
  - words. If the data under question is not real valued a common approach is to use *the hamming distance*

- Choosing k for k-NN
  - k must be odd to break ties

- Practical aspects: Feature normalization could be important
  - Often, good idea to center the features to make them zero mean and unit standard deviation
  - Because different features could have different scales (weight, height, etc); but the distance weights them equally
- Advantages and disadvantages
  - **Advantages:**
    - Training is *very fast* since it's just adding labeled instances to a list
    - Can learn very *complex functions*
    - Always have the training data, so something can be done later if wanted
  - **Disadvantages:**
    - Needs a lot of storage
    - Prediction can be slow as it has to check every instance, natively $\mathcal{O}(dN)$ for N training examples and d dimensions.
    - Nearest neighbors are fooled by irrelevant attributes
- Voronoi diagrams
  - Voronoi diagrams map the region in question into colors for those regions that are closest to certain labels. This makes it easier for testing the test set as it can simply be put on the map and the color that it is in can be checked against the label.



- Curse of dimensionality (applicable beyond nearest neighbor algorithms)
  - Methods that work with low dimensional spaces may fail in high dimensions
  - What might be intuitive for 2 or 3 dimensions does not always apply to higher dimensional spaces
  - *For Example:* If there is 1000 dimensional feature vectors, but only 10 are relevant, than the distances will be dominated by the large number of irrelevant features.
  - For higher dimensions the sense of distance becomes abscured. For example when considering the amount of empty space between a sphere encased in a cube for higher dimensions the amount of space becomes unintuitively large. Mose volume is found on the boarder of the cube and as a result the sphere seems to approach zero and the amount of empty space grows rapidly.

Linear Classifiers

- What are they? Why are they interesting?
  - Input is an n dimensional vector $\mathbf{x}$, with the output being a label $y \in \{-1, 1\}$
  - *Linear Threshold Units* classify an example $\mathbf{x}$ using parameters $\mathbf{w}$ and b according to the following classification rule
    - Output $= \text{sign}(\mathbf{w}^T\mathbf{x} + b)$

    * $\mathbf{w}^T\mathbf{x} + b \geq 0 \Rightarrow$ predict $y = 1$
    * $\mathbf{w}^T\mathbf{x} + b < 0 \Rightarrow$ predict $y = -1$

- What can they express? What can they not express?
  - Expressive hypothesis class
    * Many functions are linear
    * Often a good guess for a hypothesis space
    * Some functions are not linear (i.e. `XOR`, non-trivial boolean functions)
    * However there are ways tof making them linear in a higher dimensional feature space

- Geometry
  - We can view the linear classifier as defining a *hyperplane* separating instances in the answer space.
  - *Bias term* is needed because if b is zero, then restricting the learner only to hyperplanes that go through the origin which may not be expressive enough

- Feature expansion to predict a broader set of functions
  - *Forced Linearity* allows us to linearly classify non-linear data. For example, we can take data to a higher dimension and perform linear classification in the raised dimensions e.g. our instance space x can be brought baired with a polynomial, making our instance space (x, x^2)

- Gradient Descent
  - Goal is to predict a real valued output using a feature representation of the input. We assume the output is a linear function of the inputs.
  - Learning is done by minimizing the total cost or loss function. Many algorithms in machine learning (perceptron, etc.) follow this paradigm with different loss functions and different hypothesis spaces.
  - Gradient descent uses the following loss function:

$$J(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{m}(y_i - \mathbf{w}^T\mathbf{x}_i)^2$$

Mistake Bound Learning

- One way of asking how good is your classifier
  - **Protocol I**: The learner proposes instances as queries to the teacher
  - **Protocol II**: The teacher (who knows f) provides training examples
  - **Protocol III**: Some random source (e.g. Nature) provides training examples; the Teacher (Nature) provides the labels ($f(x)$)
  - There are 100 boolean variables, but it is not known that only five are relevant: $f = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$. Want to know how many examples are needed to learn it.
    * Since we know that it's a monotone conjunction, it would take $n = 100$ queries for the straightforward approach to produce the hidden conjunction exactly.
    * Teacher gives the answer $\langle(0, 1, 1, 1, 1, 0, \cdots, 0, 1), 1\rangle$
    * Then the teacher proves that these variables are required by turning each one off and showing that the function produces 0
    * This straightforward algorithm has the function learned in $k = 6$ examples to produce the hidden conjunction exactly
  - If given a list of examples that are right and wrong from nature, you can just do a bitwise and operation on them to get the variables that are required. If this returns more functions than are needed, it doesn't matter since those variables are not deterministic for the function.

- The general structure of an online learning algorithm
  - Mistake Bound Alogrithms:
    * Setting:
      · Instance space: $\mathbf{X}$ (dimensionality n)

        - Target f: $\mathbf{X} \to \{0, 1\}$, $f \in C$, the concept class (parameterized by n)
    - Learning Protocol:
        - Learner is given $\mathbf{x} \in \mathbf{X}$, randomly chosen
        - Learner predicts $h(\mathbf{x})$, and is then given $f(\mathbf{x})$ (feedback)
    - Performance: Learner makes a mistake when $h(\mathbf{x}) \neq f(\mathbf{x})$
        - $M_A(f, S)$: Number of mistakes algorithm A makes on sequence S of examples for the target function f
        - $M_A(C) = \max_{f \in C, S} M_A(f, S)$: The maximum possible number of mistakes made by A for any target function C and any sequence S of examples
    - Algorithm A is a *mistake bound algorithm* for the concept class C if $M_A(C)$ is a polynomial in n (ie $\mathcal{O}(n)$)
- No assumptions are made about the distribution of examples
- Examples are presented teo the learning algorithm in a sequence. *Could be adversarial!*

  For each example:

  1. Learner gets an unlabeled example
  2. Learner makes a prediction
  3. Then, the true label is revealed
- Count the number of mistakes
- A concept class is learnable in the *mistake bound model* if there exists an algorithm that makes a polynomial number of mistakes for any sequence of examples
    - Polynomial in the size of examples
- Important in the case of very large data sets, when the data cannot fit in memory

- Goal: Counting Mistakes. What is a mistake bound algorithm
    - Under the mistake bound model, we are not concerned about the number of examples needed to learn a function, only about not making mistakes. Only update when a mistake is made. After a reasonable number of corrections due to miostakes, it should stop making mistakes.
    - The Perceptron Convergence Theorem states that, if there exists a set of weights that are amenable to treatment with Perceptron (i.e., the data is linearly seperable), then the Perceptron learning algorithm will converge.
    - Can be given the *same example* over and over, which under the mistake bound model is ok, since won't be able to learn the function **but** won't be making mistakes either.
    - General Mistake Bound Algorithm:
        - Let C be a finite concept class
        - Goal: Learn $f \in C$
        - Algorithm `CON`:
            - In the i^{th} stage of the algorithm:
            - C_i all concepts in C consistent with all $i-1$ previously seen examples
            - Choose randomly $f \in C_i$ and use to predict the next example
        - Clearly, $C_{i+1} \subseteq C_i$
        - If a mistake is made n the i^{th} example, then $|C_{i+1}| < |C_i|$ so progress is made
        - The `CON` algorithm mades at most $|C| - 1$ mistakes

- Halving algorithm
    - Let C be a finite concept class
    - Goal: To learn a diffen function $f \in C$
        - Initialize $C_0 = C$, the set of all possible functions
        - When an example $\mathbf{x}$ arives:
            - Predict the label for $\mathbf{x}$ as 1 if a *majority* of the functions in C_i predict 1. Otherwise 0. That is, output =1 if
            $$|\{h(\mathbf{x}) = 1 : h \in C_i\}| > |\{h(\mathbf{x}) = 0 : h \in C_i\}|$$
        - If prediction $\neq f(\mathbf{x})$:
            - Update C_{i+1} = all elements of C_i that agree with $f(\mathbf{x})$
        - Learning ends when there is only one element in C_i

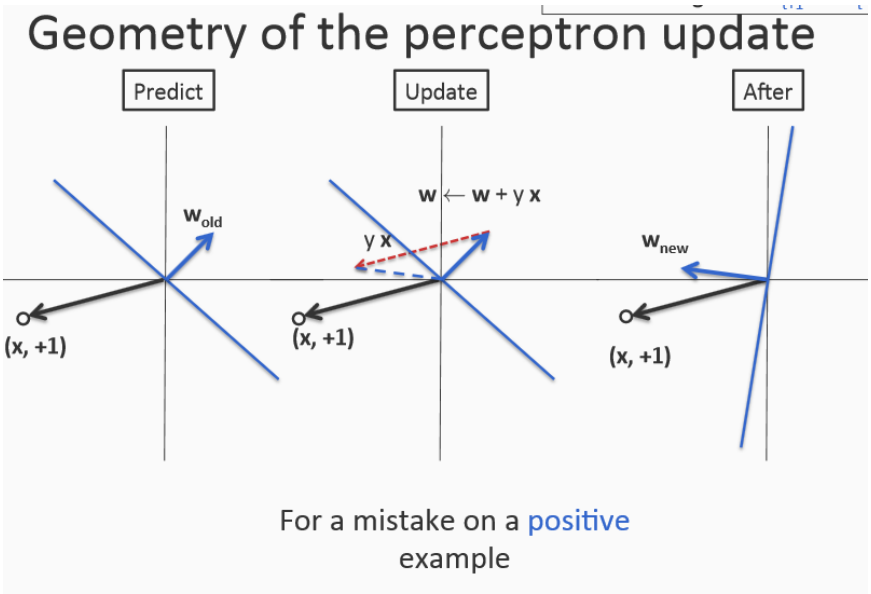
– Proof:

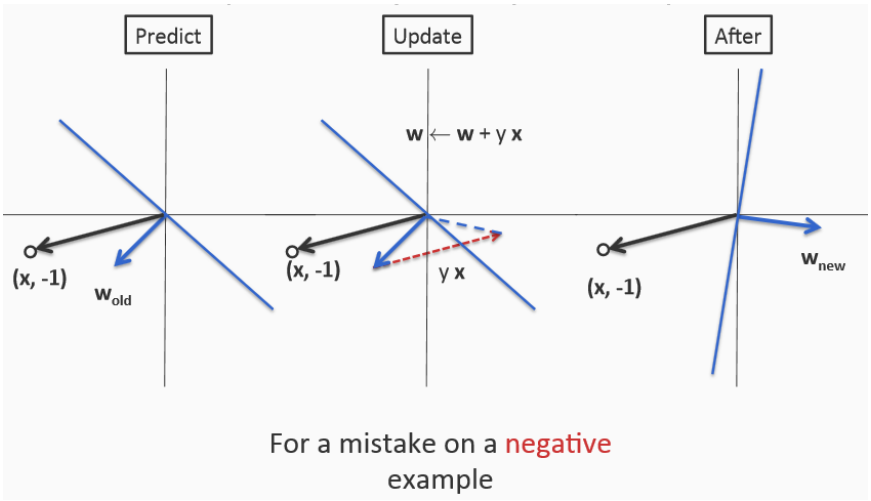
Suppose it makes n mistakes. Finally, we will have the final set of concepts C_n with one element

C_n was created when a majority of the functions in C_{n-1} were incorrect, therefore

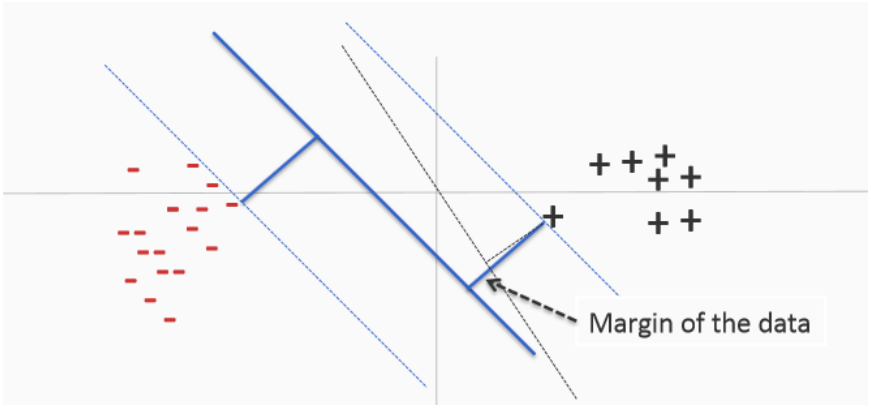
$$\begin{aligned} 1 = |C_n| &< \frac{1}{2}|C_{n-1}| \\ &< \frac{1}{2}\cdot\frac{1}{2}|C_{n-2}| \\ &< \vdots \\ &< \frac{1}{2^n}|C_0| = \frac{1}{2^n}|C| \\ &\Rightarrow \mathcal{O}\left(\log_2 |C|\right) \end{aligned}$$

- Perceptron algorithm, geometry of the update, margin, Novikoff's theorem, variants
  - Perceptron Algorithm
    1. **Input**: A sequence of training examples $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \cdots$ where all $\mathbf{x}_i \in \mathbb{R}^n, y_i \in \{-1, 1\}$
    2. Initialize $\mathbf{w}_0 = 0 \in \mathbb{R}^n$
    3. For each training example $(\mathbf{x}_i, y_i)$:
       (a) Predict $y' = \text{sign}(\mathbf{w}_t^T \mathbf{x}_i)$
       (b) If $y_i \neq y'$:
           i. Update $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + r(y_i \mathbf{x}_i)$
    4. Return final weight vector
    5. Mistake can be written as $y_i \mathbf{w}_t^T \mathbf{x}_i \leq 0$
  - Geometry





– Margin: The margin of a hyperplane for a dataset is the distance between the hyperplane and the data point nearest to it. The *margin of a data set* (γ) is the maximum margin possible for that dataset using any weight vector



– Novikoff's Theorem

∗ The number of mistakes made by the Perceptron algorithm is bound by the dimensionality R and the margin γ. γ defines the separability of the data and is defined as the distance to the two nearest points in the positive and negative groupings of the instance space. The total number of mistakes is defined by $\left(\frac{R}{\gamma}\right)^2$. For booleans, $R^2 = n$ since the L_2 norm of an n dimensional vector is $\sqrt{n}$

- Winnow algorithm, mistake bound, balanced winnow
  - Mistake bound of the Winnow algorithm for k-disjunctions is $\mathcal{O}(k \log(n))$
  - To describe `OR` of r variables where $r||n$ takes $\mathcal{O}(r \log(n))$ bits
  - Winnow *mistake bound* is $\mathcal{O}(r \log(n))$
  - Winnow learns the class of disjunctions in at most $2 + 3r(1 + \log(n))$ mistakes
  - The margin is $\gamma = \frac{\alpha}{L_1(w^*)L_\infty(X)}$ with bound $\mathcal{O}\left(\frac{1}{\gamma^2 \log(n)}\right)$
- Perceptron vs. Winnow
  - The Perceptron algorithm does addative updates while the Winnow does multiplicative. The Perceptron mistake bound for k-disjunctions is $\mathcal{O}(n)$. The Winnow for k-disjunctions is $\mathcal{O}(k \log(n))$
  - Use Winnow for multiplicative algorithms: If you believe that the hidden target function is sparse. Use Perceptron for addative functions: If you believe the hidden target function is dense.
  - **Voted Perceptron** One way of using the Perceptron is to award classifiers if they are successful for a prolonged period of time before an update. To do this we must add a weight to successful weight vectors. We therefore add a count to each success of a given classifier. If one classifier has 100 successful classifications, we add a weight of 100, incrementing this weight during each training example.

$$y = \text{sign}\left(\sum_{i=1}^{m} C^{(i)} \mathbf{w}^i \mathbf{x} + \sum_{i=1}^{m} C^{(i)} C^i b^i\right)$$

Batch Learning

- Assumption that train and test examples are drawn from the same distribution
  - Goal of batch learning: To devise a good learning algorithm that avoids overfitting, namely, find a hypothesis that has a low chance of makign a mistake on a new example.
  - Examples are drawn from fixed and maybe unknown probability distribution D.
  - Learning uses a traiing set S, which is a subset of D
- How it is different from mistake bound learning
  - Online learning has no assumption about the distribution of examples. Batch assumes there exists some probability distribution.
  - Online learning is done over a sequence of trails: learner sees an example, makes a prediction, and updates the hypothesis based on the true label. Batch learning is done over subset of the probability distribution.
  - Goal of online learnign is to bound the number of mistakes whereas batch hopes to lower the probability of making a mistake