A THEORY OF THE LEARNABLE
(L.G. VALIANT)

Theory Lunch Presentation
Claire Le Goues
05/20/10
HOW DO YOU KNOW THAT?
bool is_a_duck() {
    return (walks_like_a_duck() && quacks_like_a_duck());
}
HOW DO YOU KNOW THAT?

bool is_a_duck() {
    return (walks_like_a_duck() && quacks_like_a_duck());
}
HOW DID YOU LEARN THAT?
“A program for performing a task [like recognizing ducks – Ed.] has been acquired by learning if it has been acquired by any means other than explicit programming.”
Why This Paper Matters
Why This Paper Matters

- Present a general framework for reasoning about what is learnable as allowed by algorithmic complexity.
Present a general framework for reasoning about what is learnable as allowed by algorithmic complexity.

Introduce the idea of Probably Approximately Learnable (PAL) problems, or problems that are learnable in polynomial time, with high correctness.
Why This Paper Matters

- Present a general framework for reasoning about what is learnable as allowed by algorithmic complexity.
- Introduce the idea of Probably Approximately Learnable (PAL) problems, or problems that are learnable in polynomial time, with high correctness.
- Prove 3 classes of programs to be PAL.
Outline

1. General framework for defining Learning Machines, or programs that can learn/write/produce other programs of a particular type.
   - A Learning Machine for animal recognition, for example, might learn to write a program that recognizes whether a given animal is a duck.

2. Definition of a particular learning protocol.

3. Definition of when a program class is reasonably-learnable.

4. Definition/proofs of reasonably-learnable program classes.
A Restriction
Focus on learning skills that involve recognizing whether a concept (boolean predicate) is true for given (boolean) data.
A Restriction

- Focus on learning skills that involve recognizing whether a concept (boolean predicate) is true for given (boolean) data.
- Learn to answer the question: is this animal a duck?
A Restriction

- Focus on learning skills that involve recognizing whether a concept (boolean predicate) is true for given (boolean) data.

- Learn to answer the question: is this animal a duck?

walks like a duck = true  
purple = false  
fluffy = true  
yellow = true  
beak = true  
big = false  
quacks like a duck = true  
angry = false  
...

Low-level Definitions
Given $t$ boolean variables $p_1, \ldots, p_t$: 
Low-level Definitions

- Given $t$ boolean variables $p_1, \ldots, p_t$:
- A vector is an assignment to each of the $t$ variables one of $\{0, 1, *\}$.
  - * means “undetermined.”
Low-level Definitions

- Given $t$ boolean variables $p_1, \ldots, p_t$:
- A vector is an assignment to each of the $t$ variables one of $\{0, 1, *\}$.
  - * means “undetermined.”
- Function (concept) $F$ maps all vectors to $\{0, 1\}$.
  - Learning machine is learning concepts.
  - The variables used in $F$ are determined in $F$. 
Given \( t \) boolean variables \( p_1, \ldots, p_t \):

- A **vector** is an assignment to each of the \( t \) variables one of \( \{0, 1, *\} \).
  - * means “undetermined.”

- Function (**concept**) \( F \) maps all vectors to \( \{0, 1\} \).
  - Learning machine is learning concepts.
  - The variables used in \( F \) are **determined** in \( F \).
Low-level Definitions

- Given $t$ boolean variables $p_0, \ldots, p_t$:
- A **vector** is an assignment to each of the $t$ variables one of $\{0, 1, *\}$.
  - * means “undetermined.”
- Function (concept) $F$ maps all vectors to $\{0, 1\}$.
  - Learning machine is learning concepts.
  - The variables used in $F$ are determined in $F$. 
Given $t$ boolean variables $p_1, \ldots, p_t$:
- A **vector** is an assignment to each of the $t$ variables, one of $\{0, 1, *\}$.
  - Variables: $\{\text{walks like a duck, beak, purple, \ldots}\}$
  - Vector $v$: $\{\text{walks-like-a-duck}=0, \text{beak}=1, \text{purple}=*, \ldots\}$
  - $F(v) = \text{is\_a\_duck}(v) = \text{false}$
- Learning machine is learning concepts.
- The variables used in $F$ are *determined* in $F$. 


Low-level Definitions

- Given \( t \) boolean variables \( p_1, \ldots, p_t \):
  - A vector is an assignment to each of the \( t \) variables from \( \{0, 1, \ast\} \).
    - Variables: \( \{\text{purple, walks like a duck, beak, ...}\} \)
    - Vector \( v \): \( \{\text{purple} = \ast, \text{walks_like_a_duck} = 0, \text{beak} = 1 \ldots\} \)

Learning machine is learning concepts.
- The variables used in \( \mathcal{F} \) are determined in \( \mathcal{F} \).
Low-level Definitions

- Given \( t \) boolean variables \( p_1, \ldots, p_t \):
- A **vector** is an assignment to each of the \( t \) variables over \( \{0, 1, \ast\} \).

  - Variables: \{purple, walks like a duck, beak, \ldots\}

- Functions \( f \) to \( \{0, 1\} \).
  - Learning machine is learning concepts.
  - The variables used in \( f \) are **determined** in \( f \).
Low-level Definitions

- Given $t$ boolean variables $p_1, \ldots, p_t$:
- A **vector** is an assignment to each of the $t$ variables from one of $\{0, 1, *\}$.
- Functions map vectors to $\{0, 1\}$.
- Learning machine is learning concepts.
- The variables used in $\mathcal{F}$ are determined in $\mathcal{F}$.
• $t$ boolean variables $p_1, \ldots, p_t$:
• **Vectors** assign variables to one of $\{0, 1, *\}$.
• Concept $F$ maps vectors to $\{0, 1\}$. 
- t boolean variables $p_1, \ldots, p_t$:
- Vectors assign variables to one of $\{0, 1, *\}$.
- Concept $F$ maps vectors to $\{0, 1\}$.

Assume $D$, a probability distribution over all vectors $v$ which $F$ evaluates to $1$. 
+ $t$ boolean variables $p_1, \ldots, p_t$:
+ **Vectors** assign variables to one of $\{0, 1, \ast\}$.
+ Concept $F$ maps vectors to $\{0, 1\}$.

+ Assume $D$, a probability distribution over all vectors $v$ which $F$ evaluates to $1$.
+ $D$ is meant to describe the relative natural frequency of positive examples of whatever we’re trying to learn.
• **t** boolean variables $p_1, \ldots, p_t$:

• Vectors assign variables to one of $\{0, 1, *\}$.

• Concept $F$ maps vectors to $\{0, 1\}$.

• Assume $D$, a probability distribution over all vectors $\mathbf{v}$ which $F$ evaluates to 1.

• $D$ is meant to describe the relative natural frequency of positive examples of whatever we’re trying to learn.

• If we have a vector $\mathbf{v}$ that describes a mallard, then $D(\mathbf{v}) =$ relative frequency of mallards in the duck population.
• t boolean variables \( p_1, \ldots, p_t \):
• Vectors assign variables to one of \( \{0, 1, *\} \).
• Concept \( f \) maps vectors to \( \{0, 1\} \).

• Assume \( D \), a probability distribution over all vectors \( v \) which \( f \) evaluates to \( 1 \).
• \( D \) is meant to describe the relative natural frequency of positive examples of whatever we're trying to learn.

• Probability distribution \( D \) over all true vectors \( v \).

• If we have a vector \( v \) that describes a mallard, then \( D(v) = \) relative frequency of mallards in the universe.
• $t$ boolean variables $p_1, \ldots, p_t$:
• **Vectors** assign variables to one of $\{0, 1, *\}$.
• Concept $F$ mapping vectors to $\{0, 1\}$.
• Probability distribution $D$ over all true $v$. 
• t boolean variables $p_1, \ldots, p_t$:
• Vectors assign variables to one of $\{0, 1, *\}$.
• Concept $\mathcal{F}$ mapping vectors to $\{0, 1\}$.
• Probability distribution $\mathcal{D}$ over all true $v$. 
High-level Definitions.
High-level Definitions.

- A learning machine has two components:
A learning machine has two components:
• A learning protocol, or the method by which information is gathered from the world.
A learning machine has two components:

- A learning protocol, or the method by which information is gathered from the world.
- A deduction procedure, or the mechanism for learning new concepts from gathered information.
VALIANT’S LEARNING PROTOCOL
- \( t \) boolean variables \( p_1, \ldots, p_t \):
- **Vectors** assign variables to one of \( \{0, 1, *\} \).
- Concept \( F \) mapping vectors to \( \{0, 1\} \).
- Probability distribution \( D \) over all true \( v \).
- $t$ boolean variables $p_1, \ldots, p_t$:
- **Vectors** assign variables to one of $\{0, 1, *\}$.
- Concept $F$ mapping vectors to $\{0, 1\}$.
- Probability distribution $D$ over all true $v$. 
• t boolean variables $p_1, \ldots, p_t$:

• Vectors assign variables to one of \{0, 1, *\}.

• Concept $F$ mapping vectors to \{0, 1\}.

• Probability distribution $D$ over all true $v$.

• Learner has access to two routines (or teachers):
- **t** boolean variables \( p_1, \ldots, p_t \):
- **Vectors** assign variables to one of \( \{0, 1, *\} \).
- Concept \( F \) mapping vectors to \( \{0, 1\} \).
- Probability distribution \( D \) over all true \( v \).

- Learner has access to two routines (or teachers):
  1. **EXAMPLE**: takes no input, returns a vector \( v \) such that \( F(v) = 1 \).
  - Probability that EXAMPLE returns any particular \( v \) is \( D(v) \).
- $t$ boolean variables $p_1, \ldots, p_t$:
- **Vectors** assign variables to one of $\{\emptyset, 1, *\}$.
- Concept $F$ mapping vectors to $\{\emptyset, 1\}$.
- Probability distribution $D$ over all true $v$.

- Learner has access to two routines (or teachers):
1. **EXAMPLE**: takes no input, returns a vector $v$ such that $F(v) = 1$.
   - Probability that EXAMPLE returns any particular $v$ is $D(v)$.
2. **ORACLE**: takes as input a vector $v$, returns $F(v)$. 
Duck Example of Protocol Functions
Duck Example of Protocol Functions

\[ F(v) = \text{is\_a\_duck}(v) \]
$F(v) = \text{is\_a\_duck}(v)$
Duck Example of Protocol Functions

\[ F(v) = \text{is\_a\_duck}(v) \]

**EXAMPLE()**
$F(v) = \text{is\_a\_duck}(v)$

<table>
<thead>
<tr>
<th>EXAMPLE()</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ORACLE()</td>
<td></td>
</tr>
</tbody>
</table>
Duck Example of Protocol Functions

\[ F(v) = \text{is\_a\_duck}(v) \]

<table>
<thead>
<tr>
<th>EXAMPLE()</th>
<th><img src="example_duck.png" alt="Example Duck" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>ORACLE()</td>
<td><img src="oracle_duck.png" alt="Oracle Duck" /></td>
</tr>
</tbody>
</table>
Duck Example of Protocol Functions

\[ F(v) = \text{is\_a\_duck}(v) \]

| Example() | True
| ORACLE( ) | True
| ORACLE( ) | True

\[ F(v) = \text{is\_a\_duck}(v) \]
Duck Example of Protocol Functions

\[ F(v) = \text{is\_a\_duck}(v) \]

<table>
<thead>
<tr>
<th>EXAMPLE()</th>
<th>[ \text{duck} ]</th>
<th>[ \text{false} ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORACLE()</td>
<td>[ \text{duck} ]</td>
<td>TRUE</td>
</tr>
<tr>
<td>ORACLE()</td>
<td>[ \text{elephant} ]</td>
<td>FALSE</td>
</tr>
</tbody>
</table>
Realistic Example of Protocol Functions
Realistic Example of Protocol Functions

\[ F = (a_1 \lor a_2) \land (a_4 \lor a_1) \]
Realistic Example of Protocol Functions

\[ F = (a_1 \lor a_2) \land (a_4 \lor a_1) \]

EXAMPLE()
Realistic Example of Protocol Functions

\[ F = (a_1 \lor a_2) \land (a_4 \lor a_1) \]

| EXAMPLE() | \{a_1=1, a_2=0, a_3=*\} |
### Realistic Example of Protocol Functions

\[ F = (a_1 \lor a_2) \land (a_4 \lor a_1) \]

<table>
<thead>
<tr>
<th>EXAMPLE()</th>
<th>{a_1=1, a_2=0, a_3=*}</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXAMPLE()</td>
<td></td>
</tr>
</tbody>
</table>
Realistic Example of Protocol Functions

\[ F = (a_1 \lor a_2) \land (a_4 \lor a_1) \]

<table>
<thead>
<tr>
<th>EXAMPLE()</th>
<th>{a_1=1, a_2=0, a_3=*}</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXAMPLE()</td>
<td>{a_1=0, a_2=1, a_3=*, a_4=1}</td>
</tr>
</tbody>
</table>
\[ F = (a_1 \lor a_2) \land (a_4 \lor a_1) \]

<table>
<thead>
<tr>
<th>EXAMPLE()</th>
<th>{a_1=1, a_2=0, a_3=*}</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXAMPLE()</td>
<td>{a_1=0, a_2=1, a_3=*, a_4=1}</td>
</tr>
<tr>
<td>ORACLE({a_1=0,a_2=0,a_3=*,a_4=1})</td>
<td></td>
</tr>
</tbody>
</table>
### Realistic Example of Protocol Functions

\[ F = (a_1 \lor a_2) \land (a_4 \lor a_1) \]

<table>
<thead>
<tr>
<th>EXAMPLE()</th>
<th>{a_1=1, a_2=0, a_3=*}</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXAMPLE()</td>
<td>{a_1=0, a_2=1, a_3=*, a_4=1}</td>
</tr>
<tr>
<td>ORACLE({a_1=0,a_2=0,a_3=*,a_4=1})</td>
<td>FALSE</td>
</tr>
</tbody>
</table>
A class of problems is **Probably Approximately Learnable** if instances of the problem can be learned by a deduction algorithm that:
A class of problems is **Probably Approximately Learnable** if instances of the problem can be learned by a **deduction algorithm** that:

- Uses this protocol.
A class of problems is **Probably Approximately Learnable** if instances of the problem can be learned by a deduction algorithm that:

- Uses this protocol.
- Runs in reasonable time: polynomial by adjustable parameter h, size of learned program, and number of variables determined in the learned formula.
A class of problems is Probably Approximately Learnable if instances of the problem can be learned by a deduction algorithm that:

- Uses this protocol.
- Runs in reasonable time: polynomial by adjustable parameter \( h \), size of learned program, and number of variables determined in the learned formula.
- Produces a program that says something is false when it’s true with probability no greater than \( (1-h^{-1}) \); never says that something is true when it’s false.
A Summary, in English
We are trying to make a program (learning machine) that can learn, in polynomial time, another program (the learned program) that recognizes whether a boolean formula (concept) is true for any set of boolean data.
A Summary, in English

- We are trying to make a program (learning machine) that can learn, in polynomial time, another program (the learned program) that recognizes whether a boolean formula (concept) is true for any set of boolean data.
- The learning program has access to a function that will give it a bunch of examples, as well as a function that will check its work.
A Summary, in English

- We are trying to make a program (learning machine) that can learn, in polynomial time, another program (the learned program) that recognizes whether a boolean formula (concept) is true for any set of boolean data.

- The learning program has access to a function that will give it a bunch of examples, as well as a function that will check its work.

- The learning machine can learn a program that is sometimes wrong, so long as the probability that the learned program is ever wrong is adjustable.
1. General framework for defining Learning Machines, or programs that can learn/write/produce other programs of a particular type.
   - A Learning Machine for animal recognition, for example, might learn to write a program that recognizes whether a given animal is a duck.

2. Definition of a particular learning protocol.

3. Definition of when a program class is reasonably-learnable.

4. Definition/proofs of reasonably-learnable program classes.
Outline

1. General framework for defining Learning Machines, or programs that can learn/write/produce other programs of a particular type.
   - A Learning Machine for animal recognition, for example, might learn to write a program that recognizes whether a given animal is a duck.

2. Definition of a particular learning protocol.

3. Definition of when a program class is reasonably-learnable.

4. Definition/proofs of reasonably-learnable program classes.
The paper proves three different program classes probably-approximately-learnable.

Outline

1. General framework for defining Learning Machines, particularly:
   - The paper proves three different program classes probably-approximately-learnable.

2. Define...

3. Define...

4. Define...
Outline

1. General framework for defining Learning Machines, Ongoing work on deriving

- The paper proves three different program classes probably-approximately-learnable.
- I am not going to walk through the proofs; they are by construction of deduction algorithms that can learn the given programs and proofs of their bounds.

2. Discussion

3. Discussion

4. Discussion

5. Discussion
Outline

1. General framework for defining Learning Machines, conceptual
   notions.
   - The paper proves three different program classes probably-approximately-learnable.
   - I am not going to walk through the proofs; they are by construction of deduction algorithms that can learn the given programs and proofs of their bounds.

2. Definition of a function.

3. Definition of a learning algorithm.

4. Definition of a machine learning a function.
1. General framework for defining Learning Machines, and prove three different program classes probably-approximately-learnable.
   - The paper proves three different program classes probably-approximately-learnable.
   - I am not going to walk through the proofs; they are by construction of deduction algorithms that can learn the given programs and proofs of their bounds.

2. Define the learning algorithms.
   - I am going to give the upper bounds of the algorithms. This requires a definition of a function.

3. Define the learning algorithms.
   - The proof of that function’s upper bound is the major lemma in all three proofs, so I will outline it.
1. General framework for defining Learning Machines, criteria for program classes, probably-approximately-learnable.
   - The paper proves three different program classes probably-approximately-learnable.
   - I am not going to walk through the proofs; they are by construction of deduction algorithms that can learn the given programs and proofs of their bounds.

2. Define bounds for algorithms. This requires a definition of a function.
   - I am going to give the upper bounds of the algorithms. This requires a definition of a function.

3. Define a function to prove its upper bound.
   - The proof of that function’s upper bound is the major lemma in all three proofs, so I will outline it.

4. Define a function’s upper bound.
   - This means the next 3 slides are mathy.
A Combinatorial Bound
A Combinatorial Bound

- $L(h, S)$ is a function defined for all real numbers $h > 1$ and integers $S > 1$. 
A Combinatorial Bound

- \( L(h, S) \) is a function defined for all real numbers \( h > 1 \) and integers \( S > 1 \).
- Returns smallest integer \( n \) such that in \( n \) independent Bernoulli trials, each with probability at least \( h^{-1} \) of success, \( P(< S \text{ successes}) < h^{-1} \)
  - Bernoulli trial: an experiment whose outcomes are either “success” or “failure”; randomly distributed by some probability function.
Upper Bound on $\mathbb{L}(h, S)$
Upper Bound on $\mathbb{L}(h,S)$

$\mathbb{L}(h,S) \leq 2h(S + \log_e h)$
Upper Bound on $\mathcal{L}(h, S)$

\[ \mathcal{L}(h, S) \leq 2h(S + \log_e h) \]

Proof by algebraic substitution of well-known inequalities:
Upper Bound on $L(h, S)$

$\mathbb{L}(h, S) \leq 2h(S + \log_e h)$

Proof by algebraic substitution of well-known inequalities:

1. $\forall x > 0, \ (1 + x^{-1})^x < e$
Upper Bound on $L(h, S)$

$L(h, S) \leq 2h(S + \log_e h)$

Proof by algebraic substitution of well-known inequalities:

1. $\forall x > 0$, $(1 + x^{-1})^x < e$
2. $\forall x > 0$, $(1 - x^{-1})^x < e^{-1}$
Upper Bound on $\mathbb{L}(h, S)$

\[ \mathbb{L}(h, S) \leq 2h(S + \log_e h) \]

Proof by algebraic substitution of well-known inequalities:

1. $\forall x > 0, \quad (1 + x^{-1})^x < e$
2. $\forall x > 0, \quad (1 - x^{-1})^x < e^{-1}$

3. In $m$ independent trials, each with success probability $\geq p$:
   
   $\mathbb{P}(\text{successes at most } k) \leq \left( \frac{m-mp}{m-k} \right)^{m-k} \left( \frac{mp}{k} \right)^k$
So?
\[ L(h, S) \text{ is basically linear in both } h \text{ and } S. \]
So?

- $L(h, S)$ is basically linear in both $h$ and $S$.
- Applies to using EXAMPLEs and ORACLE to determine vectors.
So?

- $L(h, S)$ is basically linear in both $h$ and $S$.
- Applies to using EXAMPLEs and ORACLE to determine vectors.
- An algorithm can approximate the set of determined variables in natural EXAMPLEs of $F$ in runtime independent of total number of variables in the world.
$L(h, S)$ is basically linear in both $h$ and $S$.

- Applies to using EXAMPLEs and ORACLE to determine vectors.
- An algorithm can approximate the set of determined variables in natural EXAMPLEs of $F$ in runtime independent of total number of variables in the world.
  - Dependent only the number of variables that are determined in $F$. 
Given that learning protocol, what classes of tasks are learnable in polynomial time?
Answer: At Least 3 Classes of Programs
1. $k$-CNF expressions
1. $k$-CNF expressions
2. Monotone DNF expressions
Answer: At Least 3 Classes of Programs

1. $k$-CNF expressions
2. Monotone DNF expressions
3. $\mu$-expressions
$k$-CNF Expressions
$k$–CNF Expressions

- Conjunctive Normal form (CNF):

$$(a_1 \lor a_2 \lor a_3) \land (a_4 \lor a_1) \ldots$$
$k-$CNF Expressions

- Conjunctive Normal form (CNF):
  $$(a_1 \lor a_2 \lor a_3) \land (a_4 \lor a_1) \ldots$$

- $k$-CNF expression: a CNF expression where each internal clause is composed of $\leq k$ literals.
$k$–CNF Expressions

- Conjunctive Normal form (CNF):
  \[(a_1 \lor a_2 \lor a_3) \land (a_4 \lor a_1) \ldots\]

- $k$-CNF expression: a CNF expression where each internal clause is composed of $\leq k$ literals.

- Learnable with an algorithm that does not call ORACLE, and calls EXAMPLE $\leq L(h, 2t^{k+1})$ times. ($t$ is the number of variables)
Monotone DNF Expressions

- Disjunctive Normal Form (DNF):

\[(a_1 \land a_2 \land a_3) \lor (a_1 \land a_4) \ldots\]
Monotone DNF Expressions

- Disjunctive Normal Form (DNF):
  \[(a_1 \land a_2 \land a_3) \lor (a_1 \land a_4) \ldots\]

- An expression is monotone if it contains no negated literals.
Monotone DNF Expressions

- Disjunctive Normal Form (DNF):
  \[(a_1 \land a_2 \land a_3) \lor (a_1 \land a_4) \ldots\]
- An expression is monotone if it contains no negated literals.
- Learnable with an algorithm that calls EXAMPLEs \( L = L(h, d) \) times and ORACLEs \( d \times t \) times, where \( d \) is the degree of the expression and \( t \) is the number of variables.
μ-expressions
µ-expressions

- General expression over \( \{p_1, ..., p_t\} \) defined recursively (1 ≤ i ≤ t):

\[
f := p_i \mid \neg p_i \mid f_1 \land f_2 \mid f_1 \lor f_2
\]
µ-expressions

- General expression over \( \{p_1, \ldots, p_t\} \) defined recursively \((1 \leq i \leq t)\):

\[
f := p_i \ | \ \neg p_i \ | \ f_1 \land f_2 \ | \ f_1 \lor f_2
\]

- A µ-expression is an expression in which each \( p \) appears at most once.
µ-expressions

- General expression over \{p_1, \ldots, p_t\} defined recursively (1 ≤ i ≤ t):
  \[ f := p_i \mid \neg p_i \mid f_1 \land f_2 \mid f_1 \lor f_2 \]

- A µ-expression is an expression in which each \( p \) appears at most once.
- Learnable with an exactly correct algorithm that calls two slightly more powerful ORACLE functions \( O(t^3) \) times total.
Executive Summary
Learnability theory is concerned with what programs can be learned automatically.
Learnability theory is concerned with what programs can be learned automatically.

We should reason about what is programmatically learnable in the same way we reason about what is computable.
Executive Summary

- Learnability theory is concerned with what programs can be learned automatically.
- We should reason about what is programmatically learnable in the same way we reason about what is computable.
- A class of programs is Probably Approximately Learnable when, using a particular type of teacher, a given algorithm can learn a program that can recognize instances of that class with a certain probability.
Executive Summary

- Learnability theory is concerned with what programs can be learned automatically.
- We should reason about what is programmatically learnable in the same way we reason about what is computable.
- A class of programs is Probably Approximately Learnable when, using a particular type of teacher, a given algorithm can learn a program that can recognize instances of that class with a certain probability.
- 3 examples of such learnable program types are k-CNF expressions, monotone DNF expressions, and $\mu$-expressions.
Interesting Concluding Questions

- What else is learnable by these definitions?
- Is the definition of “learnable” reasonable?
  - How powerful should the teachers be?
  - What about if we use negative in addition to positive examples?
- How do humans learn?
• Assume we have urn that contains many marbles of $s$ different types. We want to “learn” the different types of marbles by taking a small random sample $x$, of size sufficient that, with high probability, it contains at least 99% of $s$ marble type representatives.
• Assume we have urn that contains many marbles of \( s \) different types. We want to “learn” the different types of marbles by taking a small random sample \( x \), of size sufficient that, with high probability, it contains at least 99% of \( s \) marble type representatives.

• Definition of \( L(h, S) \) implies:
  \[
  |x| = L(100, S) \implies P(\text{succeeded overall}) > 99%
  \]
• Assume we have an urn that contains many marbles of \( s \) different types. We want to “learn” the different types of marbles by taking a small random sample \( x \), of size sufficient that, with high probability, it contains at least 99% of \( s \) marble type representatives.

• Definition of \( L(h,S) \) implies:

\[
|X| = L(100,S) \implies P(\text{succeeded overall}) > 99%
\]

• “Success” for each trial is defined as picking a marble we haven’t picked before. Success clearly depends on previous choices, but the probability of each success will always be at least 1%, \textbf{independent of previous choices}. 