Comp 311
Functional Programming

Eric Allen, PhD
Vice President, Engineering
Two Sigma Investments, LLC
Machine Learning With Spark

• Given a collection of examples with various attributes and a label, we wish to predict the labels for new examples:

\(<\text{height, weight, age, systolic bp, diastolic bp}>: \text{medicine?}\>

\(<170 \text{ cm, 72 kg, 52, 120, 80}>: \text{YES}\>

\(<150 \text{ cm, 60 kg, 34 years, 130, 70}> : \text{NO}\>

...
Machine Learning With Spark

• We can view the examples as vectors in a high-dimensional vector space

• The problem of labeling yes/no can be solved by finding the best hyperplane that divides the given examples according to their labels

• This new hyperplane can be used to predict labels for new examples
Machine Learning With Spark

• We can view the examples as vectors in a high-dimensional vector space

• The problem of labeling yes/no can be solved by finding the best hyperplane that divides the given examples according to their labels

• This new hyperplane can be used to predict labels for new examples
Logistic Regression With Spark

```scala
val points = spark.textFile(...).map(parsePoint).cache()

var w = Vector.random(D) // current separating plane

for (i <- 1 to ITERATIONS) {
  val gradient = points.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)

  w -= gradient
}

println("Final separating plane: " + w)
```
Homework 6
def tactic(state: ProofState => (PartialProof, ProofState)) =
  StateAction[ProofState, PartialProof](state)
def orElimTactic(f: Formula) = tactic {
    (proofState: ProofState) => {
        (f, proofState) match {
            case (p \or q, ((gamma :- r) :: goals)) =>
                def partialProof(proofs: List[Sequent]) = {
                    proofs match {
                        case (proofA :: proofB :: proofC :: Nil) =>
                            orElim(proofA, proofB, proofC)
                        case _ => throw new ProofError("orElim applied to " + proofs)
                    }
                }
                (partialProof,
                    (gamma :- p \or q) :: (gamma + p :- r) :: (gamma + q :- r) :: goals)
            case _ => throw TacticError("orElimTactic applied to " + proofState)
        }
    }
}
def impliesElimTactic(p: Formula) = tactic {
  (proofState: ProofState) => {
    proofState match {
      case ((gamma :- q) :: goals) =>
        def partialProof(proofs: List[Sequent]) = {
          proofs match {
            case (proofA :: proofB :: Nil) =>
              impliesElim(proofA, proofB)
            case _ => throw new ProofError("impliesElim applied to " + proofs)
          }
        }
        (partialProof, ((gamma :- (p -> q)) :: (gamma :- p) :: goals))
      case _ => throw TacticError("impliesElimTactic applied to " + proofState)
    }
  }
}
Summary and Conclusion
Functional Programming

- A style of programming involving no mutation
- A style of programming in which computations are represented by passing functions as arguments and returning them as results
- A style of programming in which the entirety of a computation is determined by explicit input and output values
The Substitution Model

- The behavior of purely functional programs can be understood via the Substitution Model of Evaluation.
- Having a rigorous model allows us to reason more precisely about the behavior of programs.
Static Types and Values

• There is a deep connection between computation of static types and computation of values

• We can think of a computation as having both a static and dynamic evaluation
Design Recipes

- Categorizing solution based on templates
- Test-driven development
Abstract and Recursively Defined Datatypes

• Immutable data is often well characterized by this framework

• Language syntax is defined by the same framework

• Every abstract datatype can be thought of as characterizing its own language
Pattern Matching

• Dramatically improves the conciseness of computations over abstract datatypes
Variance in Types

- Immutable datatypes naturally lead to more expressive type relationships
  - Arrow types
  - Immutable collections
Programming Principles

• Keep It Simple

• Don’t Repeat Yourself
First-Class Functions

• Enables significant reduction in code repetition

• Leads to new ways of thinking about and organizing computations
  • map
  • reduce
  • filter
  • flatMap
Monads

- An overarching pattern of organization for many computations can be found via the concept of monads
- For-expressions are the syntax of monadic computation
Monads

- Monads express many computing constructs
  - Collections
  - Options
  - Purely functional state
Continuations

• A unifying construct for control in computation
  • Exceptions
  • Concurrency
  • Pre-emption
The Environment Model

- Facilitates reasoning about mutual recursion
- Facilitates reasoning about state
- Facilitates reasoning about types
- Facilitates reasoning about logics
Alternative Approaches to Computation

• Lexical vs Dynamic Scoping
  • Dynamic scoping is incompatible with static types

• Call-by-Value vs Call-by-Name
  • Call-by-Name is in often a better fit for functional programming

• Traits
  • Composable units of computation
Generative Recursion

• Not all computations can be expressed as structural traversals of abstract datatypes
  
  • But many can!

• Sometimes either domain knowledge or deep algorithmic insights is needed

  • Quicksort, Heaps, Red-Black Trees
Accumulators and Tail Recursion

• There is far more to accumulators than just simple tail recursion

  • Determine the *meaning* of your accumulator variables, and respect that meaning

  • Distinct approaches to accumulator meaning lead to significantly different results

  • Tree traversal, peg solitaire
State and Functional Programming

• Streams
  • Call-By-Name-Style lists
  • Often an effective alternative to stateful computation
• Memoization
  • Using state to improve performance of stateless computation
• The State Monad
  • Encapsulate and replay stateful computation, providing initial state as input
The Curry-Howard Isomorphism

• There is a deep connection between types and logical propositions

• There is a deep connection between programs and logical proofs
Some Scala Libraries of Note

- Parser combinators
- Actors for concurrency
Tactical Theorem Proving

• One of the driving problems driving the development of functional programming

• Term-rewriting systems
  • A unifying concept for computer science
  • Appear in types, models of computation, logic

• Tactics and the state monad
Functional Distributed Computing

- MapReduce
- Apache Spark and RDDs
- Distributed machine learning
A Functional Programming Renaissance

- Multiple trends are driving software design toward functional programming
  - Multicore processors
  - Big data
Take Advantage of the Work of the Past

• The Substitution Model
• Lexical scoping
• Closures
• Streams
• Monads
• Memoization
• Continuations
Consider Comp 411

• Application of functional programming to defining programming languages

• Abstract datatypes to define syntax

• Type checkers to define static semantics

• Interpreters to define dynamic semantics (via a meta-language)

• Incrementally enhance understanding by removing available features from the meta-language
Adopt a Sense of Coding
Craftsmanship

• Keep It Simple
• Don’t Repeat Yourself
• Test from the start!
Teach a Class

• There’s no better way to learn a subject