Probabilistic Programming

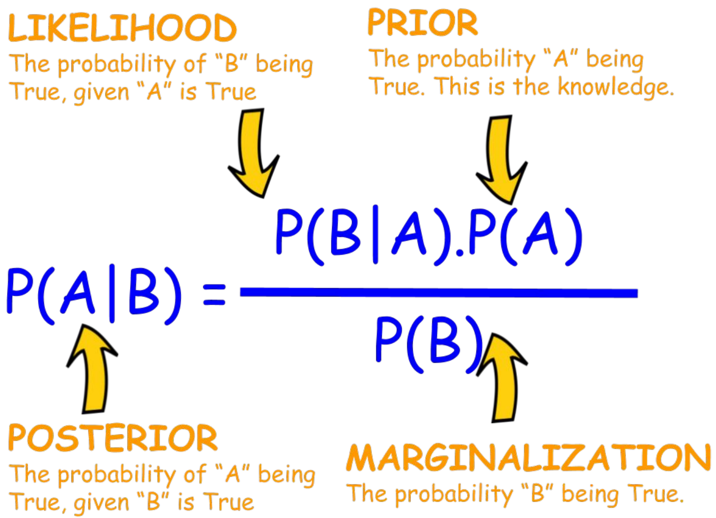
Cassandra Overney & Vivien Chen

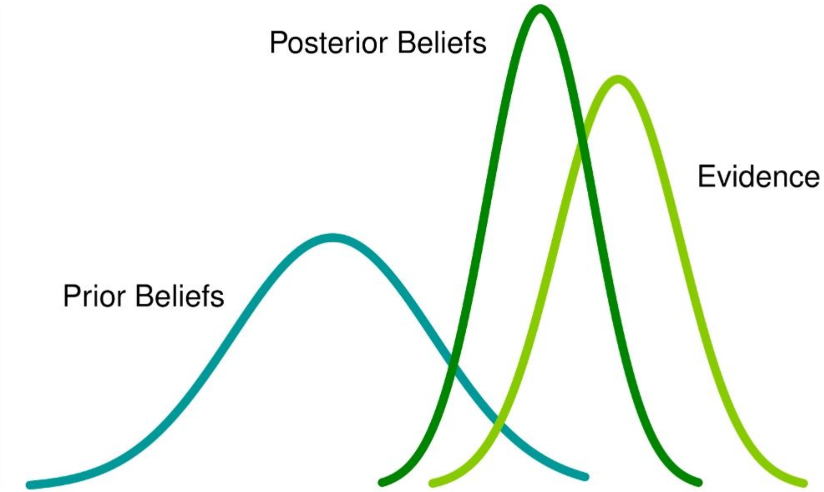
Goals of Project

- Learn about probabilistic programming
- Implement some features of probabilistic programming in Python
  - Primitives (some new math functions)
  - Primitive Distributions (sampling, defining new distributions, operations on distributions)
- *Reimplement FUNC in Python

Bayesian Inference

- Update the probability for a hypothesis as more evidence or data becomes available



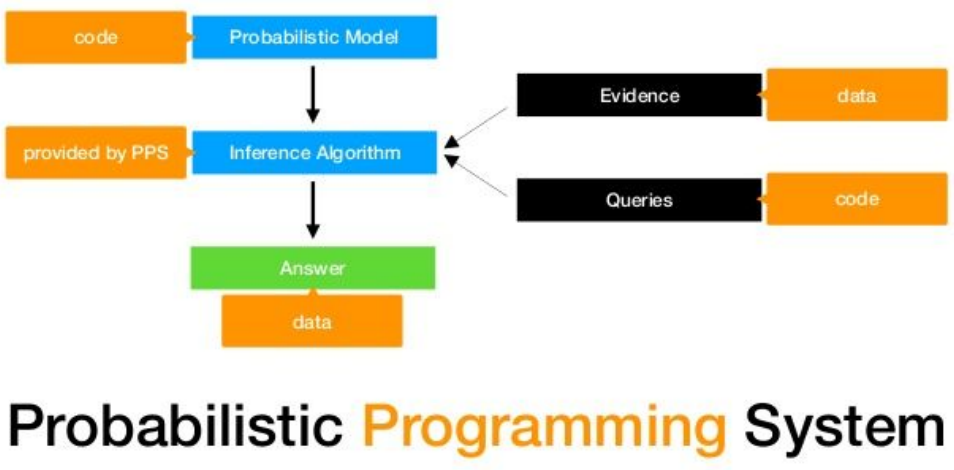


https://www.analyticsvidhya.com/blog/2016/06/bayesian-statistics-beginners-simple-english/

https://towardsdatascience.com/bayes-rule-with-a-simple-and-practical-example-2bce3d0f4ad0

What Is Probabilistic Programming?

- A tool for statistical modeling



https://www.slideshare.net/nblumoeste/intro-to-probabilistic-programming-and-clojures-anglican

Anglican

- Turing-complete probabilistic research programming language
- Allows intuitive modeling in a stochastic environment
- Language easily integrates with Clojure (similar to Lisp and thus FUNC!)
- Easily **scalable**
- Attempts to infer distributions based on observed data

```
(sample* (beta 1 1)) ;; => 0.7185479773538508
```

Our Solution: Fake Anglican (.py)

- Re-implemented FUNC in Python with a bunch of extra primitive operations and some new classes to represent distributions
  - EDistribution
  - VDistribution

```
expr_dist = LP >> lit('defdist') >> LP >> id & reg(r'[ ]*') >> params << RP & expr
            << RP > (lambda x: EDistribution(x[0], x[1], x[2]))
```

```
class EDistribution(Exp):

    def __init__(self, name, params, body):
        self.name = name
        self.params = params
        self.body = body

    ...

    def eval(self, env):
        return VDistribution(self.name, self.params, self.body, env)
```

```python
class VDistribution(Value):

    def __init__(self, name, params, body, env):
        self.name = name
        self.params = params
        self.body = body
        self.env = env

    ...

    def isDistribution(self):
        return True

    def apply(self, args):
        if len(self.params) != len(args):
            runtimeError("wrong number of arguments\n  Function " + str(self))
        new_env = self.env
        for (p,v) in zip(self.params, args):
            new_env = new_env.push(p,v)
        new_env = new_env.push(self.name, self)
        return self.body.eval(new_env)
```

Primitive Operations

- **Tests:** even?, odd?, empty?
- **Relational:** =, ~=, >, >=, <, <=, not
- **Sequences:** vector, first, second, nth, rest, count, cons, map, filter, empty
- **Arithmetic:** +, -, *, /
- **Math:** log, log10, exp, ^ (power), sqrt, cbrt, floor, ceil, round, rint, abs, sign, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, inc, dec, % (mod), sum, cumsum, mean, norm
- **Distribution:** sample, beta, bernoulli, exponential, normal, poisson, uniform, randelm
- **String:** concat, lower, upper, substring
- **Other:** ref, get, put, print

Primitive Distributions

- (bernoulli p): creates a Bernoulli VDistribution with success probability p. When sampled, returns 0 or 1.
- (beta a, b): creates a Beta VDistribution with pseudocounts a and b. When sampled, returns a float on interval [0, 1).
- (exponential l): creates an exponential VDistribution with rate parameter l. When sampled, returns a float in the domain [0, inf).
- (normal m, v): creates a normal VDistribution with mean m and variance v. When sampled, returns a float from (-inf, inf).
- (poisson l): creates a Poisson VDistribution with rate l.
- (uniform min, max): creates a uniform continuous VDistribution. When sampled, returns a float in the domain [min, max).
- (randelm V): creates a VDistribution that returns a random element from a vector V when sampled.

```python
def operNormal(vs):

    checkNumberArgs(vs, 2)
    v1 = vs[0]
    v2 = vs[1]
    rational_float_v1 = convertFloat(v1)
    rational_float_v2 = convertFloat(v2)

    python_func = lambda x : VFloat(np.random.normal(rational_float_v1,
                     rational_float_v2))

    return VDistribution("", [], EPrimitive(python_func), Env())
```

Operations with Distributions

- (sample d): samples and returns a value from VDistribution d
- (- d1): returns a new VDistribution that is the negative of d1
- (+ d1, d2): returns a new VDistribution that is the sum of distributions d1 and d2
- (* d1, d2): returns a new VDistribution that is the product of distributions d1 and d2
- (/ d1, d2): returns a new VDistribution that is the quotient of distributions d1 and d2

```
def operSample(vs):

    if len(vs) > 0:
        v1 = vs[0]
        checkDistribution(v1)
        result = v1.apply(vs[1:])

        if result.isProcedure():
            return result.apply([])
        else:
            return result

    else:
        runtimeError("0 arguments applied to sample")
```

```
class EMultiple(Exp):

    def __init__(self, bodies, oper):
        self.bodies = bodies
        self.oper = oper

    ...

    def eval(self, env):
        results = []
        for body in self.bodies:
            res = body.eval(env)
            # if the result is a procedure call apply to get a more refined
value
            if res.isProcedure():
                res = res.apply([])
            results.append(res)
        return self.oper(results)
```

```
def operPlus(vs):
    ...

    elif v1.isDistribution() and v2.isDistribution():
        new_env = Env(v1.env.content+v2.env.content)
        body = EMultiple([v1.body, v2.body], operPlus)
        return VDistribution("", v1.params+v2.params, body, new_env)

    elif v1.isDistribution() and (v2.isRational() or v2.isFloat()):
        v2_float = convertFloat(v2)
        body = EMultiple([v1.body, EFloat(v2_float)], operPlus)
        return VDistribution("", v1.params, body, v1.env)

    elif v2.isDistribution() and (v1.isRational() or v1.isFloat()):
        v1_float = convertFloat(v1)
        body = EMultiple([v2.body, EFloat(v1_float)], operPlus)
        return VDistribution("", v2.params, body, v2.env)

    ...
```

Parser: Parsita

- Parsers are hard to deal with!!
- We implemented some parser transformations from HW4
- Had to add a bunch of random spaces
- Also args need to be separated by a “,_”

```
LP = reg(r'([ ]*)\(([ ]*)')
RP = reg(r'([ ]*)\)([ ]*)')
```

On the bright side...

```
def params : Parser[List[String]] =
    ( params_many | params_one ) ^^ {
        ps => ps }

def params_many : Parser[List[String]]
=
    ID ~ params ^^ { case p ~ ps =>
        p::ps }

def params_one : Parser[List[String]] =
    ID ^^ { p => List(p) }
```

```
 params_one = reg(r'[ ]*') >> id >
 str
     params = repsep(params_one, ',
 ')
```

DEMO TIME!!!

Questions?