Idris for Haskell developers

Kostiantyn Rybnikov

April 22, 2018

Intro

Why another language? Fewer bugs in Haskell code?

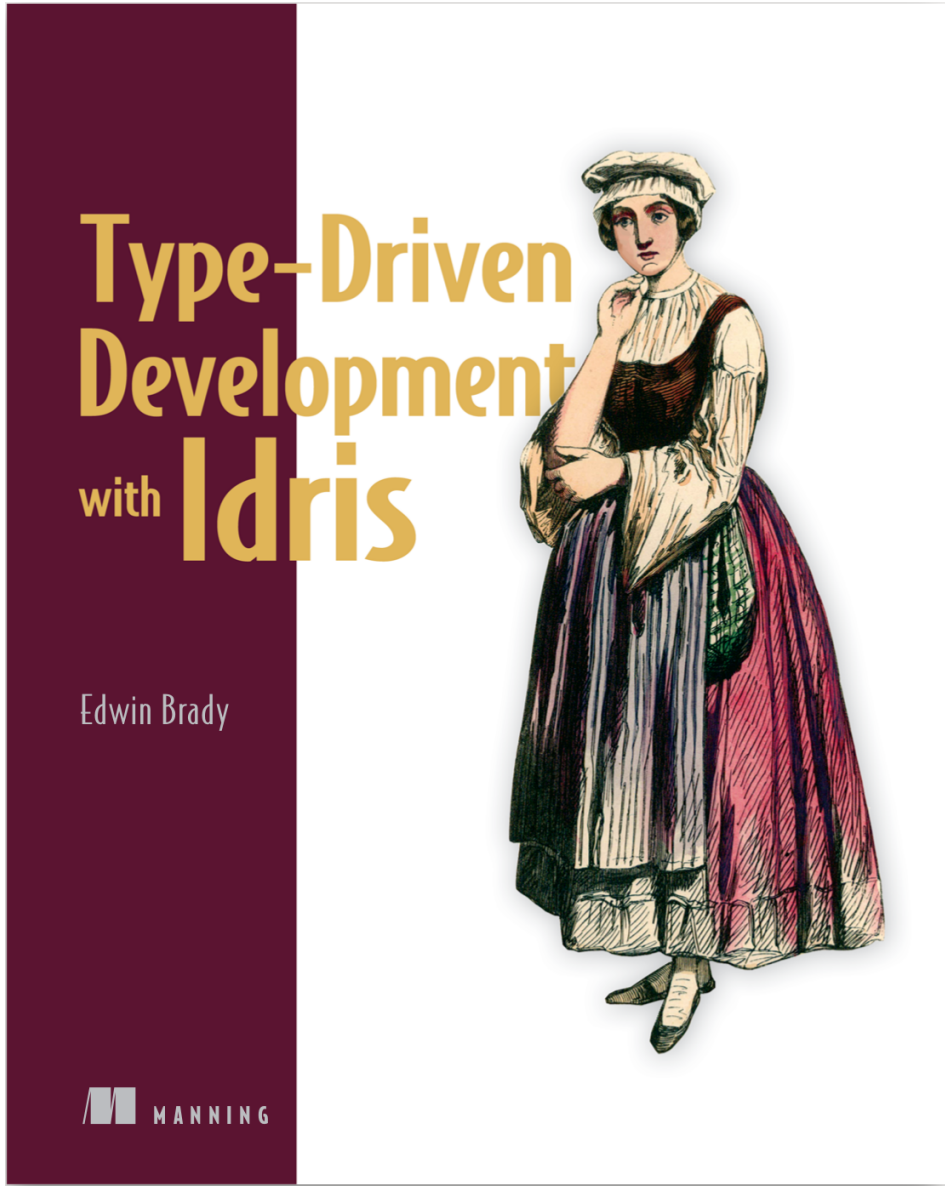


Figure 1: The Idris Book

Linear Types in Haskell

The result won't be as good as Rust for what Rust does: it's a different trade-off where we assume that such precision is only needed in small key areas of a program that otherwise freely uses functional programming abstractions as we know them today.

Linear Types in Haskell

```
data S a
data R a

newCaps :: (R a ⊸ S a ⊸ IO ()) ⊸ IO ()
send :: S a ⊸ a ⊸ IO ()
receive :: R a ⊸ (a ⊸ IO ()) ⊸ IO ()
```

Figure 2: Linear Types Example

Linear Types in Haskell

```
split :: [a] ->. ([a], [a])
split []       = ([], [])
split [x]      = ([x], [])
split (x:y:z) = go (x,y) (split z) where
  go :: (a,a) ->. ([a], [a]) ->. ([a], [a])
  go (a,b) (c,d) = (a:c, b:d)
```

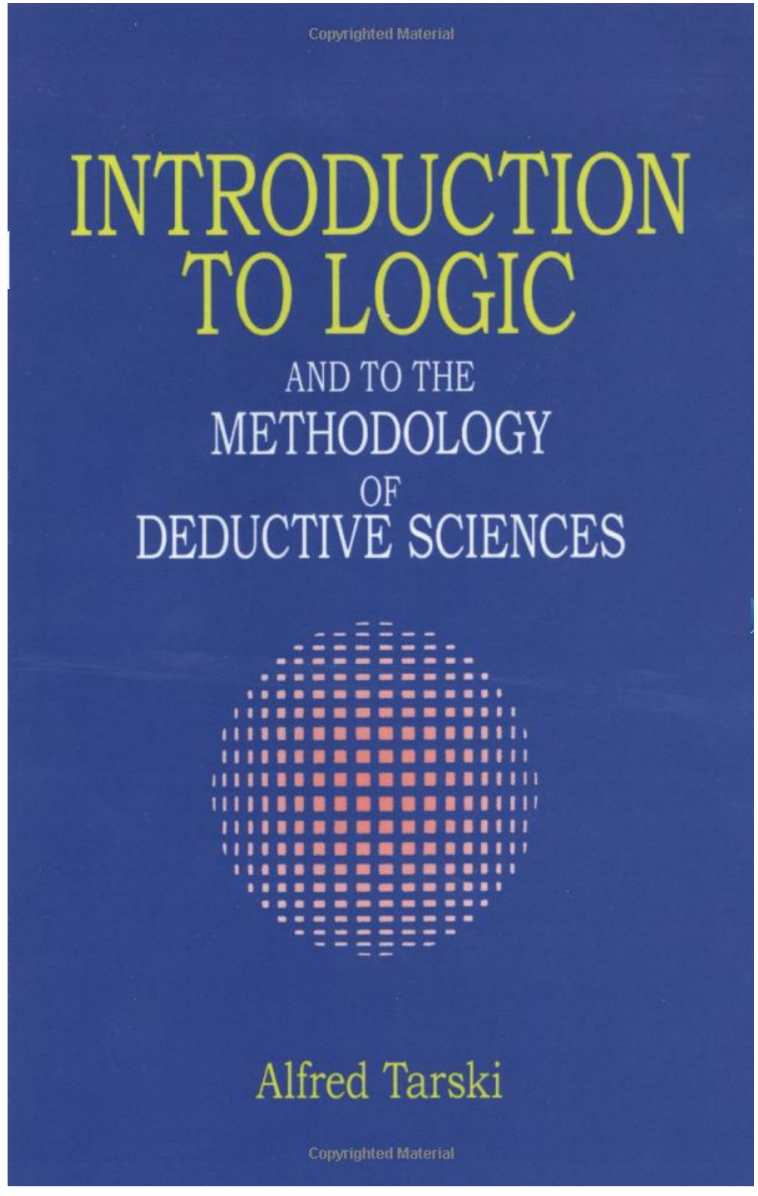


Figure 3: Tarski Logic

• VII •

CONSTRUCTION OF A MATHEMATICAL THEORY:

LAWS OF ORDER FOR NUMBERS

43. Primitive terms of the theory under construction; axioms concerning fundamental relations among numbers

With a certain amount of knowledge of the fields of logic and methodology at our disposal, we shall now undertake to lay the foundations of a particular and, incidentally, very elementary mathematical theory. This will be a good opportunity for us to assimilate better our previously acquired knowledge, and even to expand it to some extent.

The theory with which we shall concern ourselves constitutes a fragment of the arithmetic of real numbers. It contains fundamental theorems concerning the basic relations *less than* and *greater than* among numbers, as well as the basic operations on numbers, namely of addition and subtraction. It presupposes nothing but logic.

The primitive terms which we shall adopt in this theory are the following:

real number,
is less than,
is greater than,
sum.

Figure 4: Numbers Laws

Logic

- AXIOM 1. For any numbers x and y (e.g., for arbitrary elements of the set N) we have: $x = y$ or $x < y$ or $x > y$
- AXIOM 2. If $x < y$, then $y \not< x$
- AXIOM 3. If $x > y$, then $y \not> x$
- AXIOM 4. If $x < y$ and $y < z$, then $x < z$
- AXIOM 5. If $x > y$ and $y > z$, then $x > z$

Our next task consists in the derivation of a number of theorems from the axioms adopted by us:

- THEOREM 1. No number is smaller than itself $x \not< x$
- THEOREM 2. No number is greater than itself: $x \not> x$

Figure 5: The StackExchange Question

```
postulate N : Type
postulate lt : N -> N -> Type
postulate gt : N -> N -> Type
postulate add : N -> N -> N

postulate axiom1 : (x, y : N)
  -> Either (x = y) $ Either (x `lt` y) (x `gt` y)
postulate axiom2 : x `lt` y -> Not (y `lt` x)
postulate axiom3 : x `gt` y -> Not (y `gt` x)
postulate axiom4 : x `lt` y -> y `lt` z -> x `lt` z
postulate axiom5 : x `gt` y -> y `gt` z -> x `gt` z

theorem1 : Not (x `lt` x)
theorem1 contra = axiom2 contra contra
```

NLP

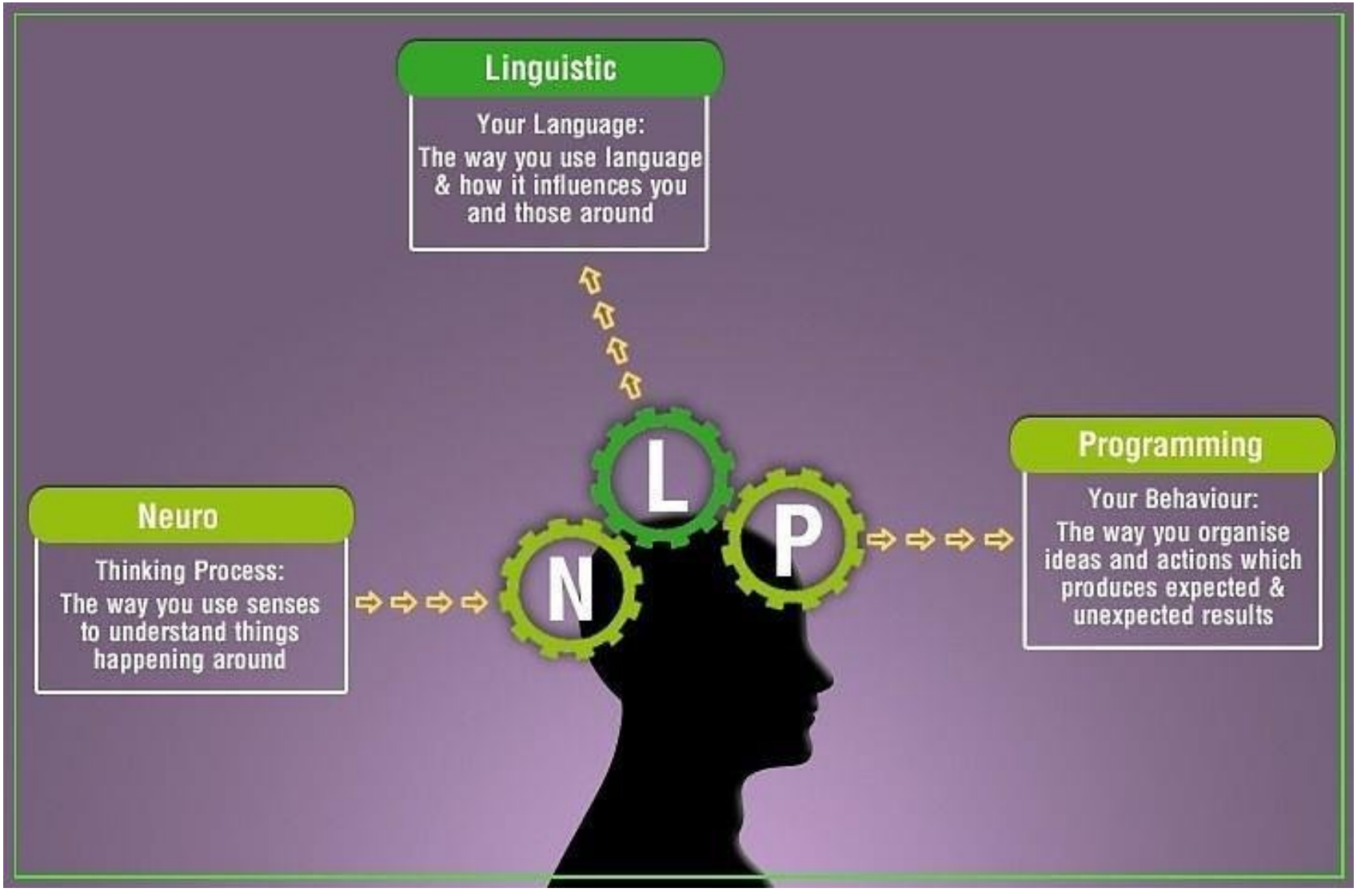

Linguistic
Your Language:
The way you use language & how it influences you and those around
Neuro
Thinking Process:
The way you use senses to understand things happening around
N
L
P
Programming
Your Behaviour:
The way you organise ideas and actions which produces expected & unexpected results

Figure 6: NLP

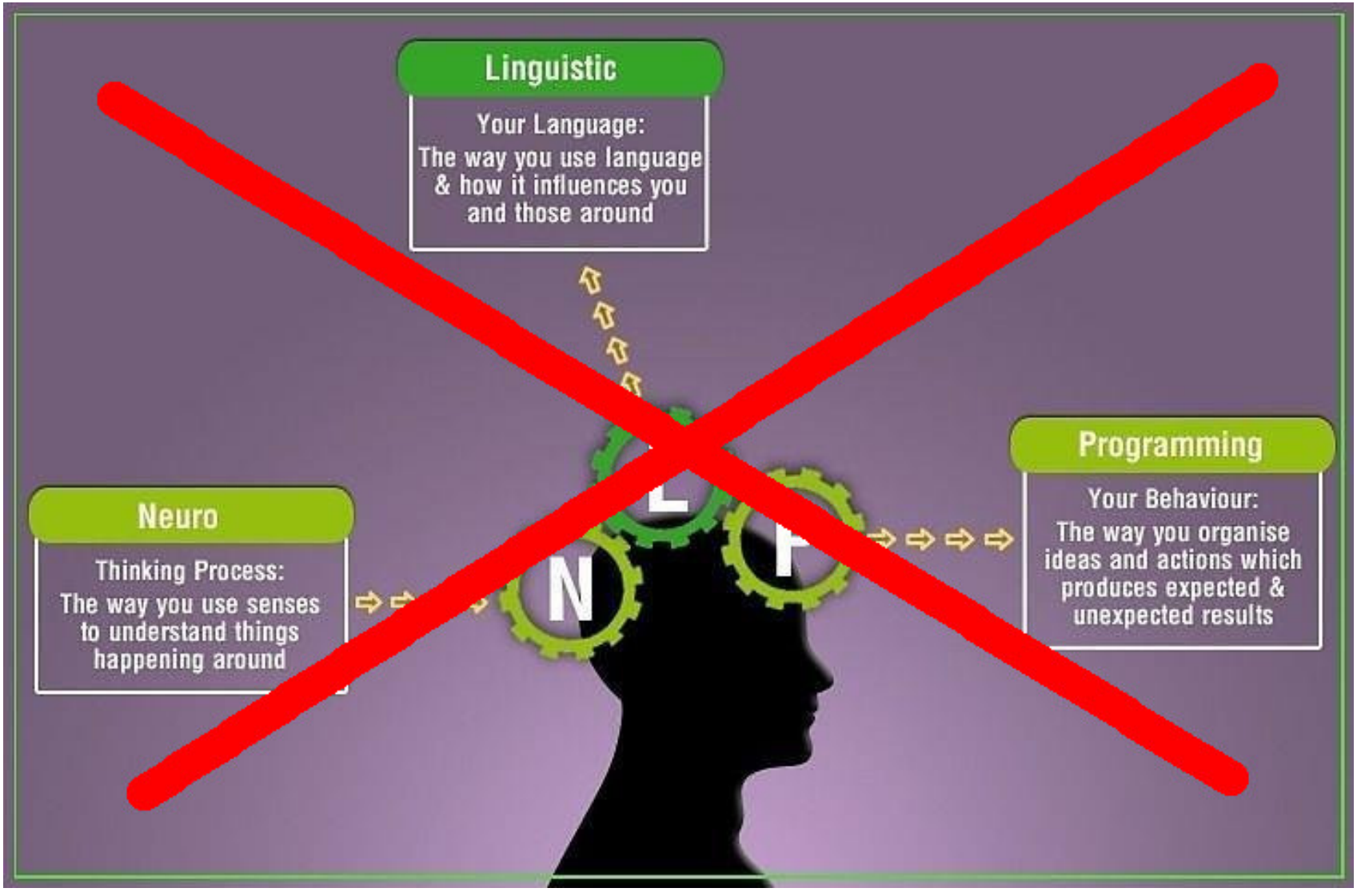
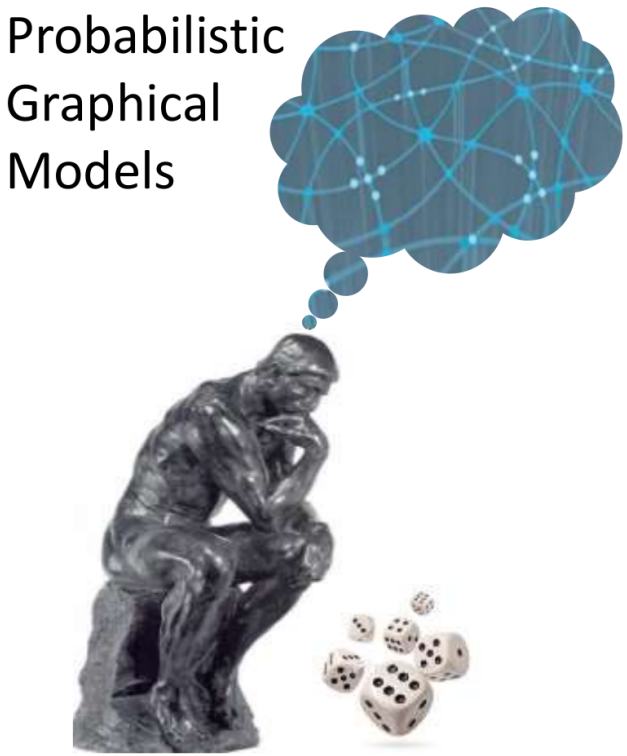


Figure 7: Not Really

Figure 8: Tom Waits meme

Figure 9: Meme Went Wild



Representation

Bayesian Networks

Naïve Bayes

Figure 10: Naive Bayes

Multinomial Naive Bayes for Text Categorization Revisited

Ashraf M. Kibriya, Eibe Frank, Bernhard Pfahringer, and Geoffrey Holmes

Department of Computer Science
University of Waikato
Hamilton, New Zealand
{amk14, eibe, bernhard, geoff}@cs.waikato.ac.nz

Abstract. This paper presents empirical results for several versions of the multinomial naive Bayes classifier on four text categorization problems, and a way of improving it using locally weighted learning. More specifically, it compares standard multinomial naive Bayes to the recently proposed transformed weight-normalized complement naive Bayes classifier (TWCNB) [1], and shows that some of the modifications included in TWCNB may not be necessary to achieve optimum performance on some datasets. However, it does show that TFIDF conversion and document length normalization are important. It also shows that support vector machines can, in fact, sometimes very significantly outperform both methods. Finally, it shows how the performance of multinomial naive Bayes can be improved using locally weighted learning. However, the overall conclusion of our paper is that support vector machines are still the method of choice if the aim is to maximize accuracy.

1 Introduction

Automatic text classification or text categorization, a subtopic in machine learning, is becoming increasingly important with the ever-growing amount of textual information stored in electronic form. It is a supervised learning technique, in which every new document is classified by assigning one or more class labels from a fixed set of pre-defined classes. For this purpose a learning algorithm is employed that is trained with correctly labeled training documents. The documents are generally represented using a "bag-of-words" approach, where the order of the words is ignored and the individual words present in the document constitute its features. The features present in all the documents make up the feature space. Since the number of words can be very large, the resulting learn-

Figure 11: Multinomial Naive Bayes

2.3 Multinomial Naive Bayes

Let us now discuss how multinomial naive Bayes computes class probabilities for a given document. Let the set of classes be denoted by C. Let N be the size of our vocabulary. Then MNB assigns a test document t_i to the class that has the highest probability $\Pr(c|t_i)$, which, using Bayes' rule, is given by:

$$\Pr(c|t_i) = \frac{\Pr(c)\Pr(t_i|c)}{\Pr(t_i)}, \qquad c \in C \tag{1}$$

The class prior $\Pr(c)$ can be estimated by dividing the number of documents belonging to class c by the total number of documents. $\Pr(t_i|c)$ is the probability of obtaining a document like t_i in class c and is calculated as:

$$\Pr(t_i|c) = (\sum_n f_{ni})! \prod_n \frac{\Pr(w_n|c)^{f_{ni}}}{f_{ni}!}, \tag{2}$$

where f_{ni} is the count of word n in our test document t_i and $\Pr(w_n|c)$ the probability of word n given class c. The latter probability is estimated from the training documents as:

$$\widehat{\Pr}(w_n|c) = \frac{1 + F_{nc}}{N + \sum_{x=1}^{N} F_{xc}}, \tag{3}$$

where F_{xc} is the count of word x in all the training documents belonging to class c, and the Laplace estimator is used to prime each word's count with one to avoid the zero-frequency problem [2]. The normalization factor $\Pr(t_i)$ in Equation 1 can be computed using

$$\Pr(t_i) = \sum_{k=1}^{|C|} \Pr(k)\Pr(t_i|k). \tag{4}$$

Note that that the computationally expensive terms $(\sum_n f_{ni})!$ and $\prod_n f_{ni}!$ in Equation 2 can be deleted without any change in the results, because neither depends on the class c, and Equation 2 can be written as:

$$\Pr(t_i|c) = \alpha \prod_n \Pr(w_n|c)^{f_{ni}}, \tag{5}$$

where α is a constant that drops out because of the normalization step.

Figure 12: Paper Formulas

```haskell
-- | Pr(c|t_i). Main working horse.
multinomialNaiveBayes ::
     (Eq word, Eq cls, Hashable cls) =>
     Env cls word -> [word] -> cls
multinomialNaiveBayes env wrds = getMax (map f (envClasses env))
  where
    d = Document (error "classes not known yet") wrds
    f c =
      ( c
      , classPrior env c * (documentProbability env d c) /
        normalizationFactor env d)
getMax = fst . head . sortBy (flip compare `on` snd)
```

```haskell
-- | Globals used in formulas
data Env cls word = Env
  { envClasses :: [cls]
  , envDocs :: [Document cls word]
  , envVocabulary :: [word]
  , envEfficiencyCache :: EfficiencyCache cls word
    -- ^ added purely for lookup efficiency
  } deriving (Show, Eq, Generic)
```

```haskell
docsInClass ::
     (Eq cls, Hashable cls) =>
     Env cls word -> cls -> [Document cls word]
docsInClass env c =
  fromMaybe []
    (H.lookup c (docsPerClass (envEfficiencyCache env)))
  where
_noCacheImp = filter (\d -> c `elem` docClasses d) (envDocs env)
```

Math Institute

Figure 13: Math Institute

Figure 14: Informatics Class

Haskell Saves Lives

Jump to: navigation, search

```
From: hudak@EDU.YALE.CS
Subject: A Good Story
Date: Thu, 1 Apr 93 10:26:31 -0500
To: haskell@EDU.YALE.CS
```

```
For every bad story there is a good one.  Recently Haskell was used
in an experiment here at Yale in the Medical School.  It was used to
replace a C program that controlled a heart-lung machine.  In the six
months that it was in operation, the hospital estimates that probably
a dozen lives were saved because the program was far more robust than
the C program, which often crashed and killed the patients.
```

```
-Paul
```

Figure 15: Haskell Saves Lives

The Talk

```
StringOrInt : Bool -> Type
StringOrInt x = case x of
  True => Int
  False => String
```

```
StringOrInt : Bool -> Type
StringOrInt x = case x of
  True => Int
  False => String

getStringOrInt : (x : Bool) -> StringOrInt x
getStringOrInt x = case x of
  True => 94
  False => "Ninety four"

valToString : (x : Bool) -> StringOrInt x -> String
valToString x val = case x of
  True => cast val
  False => val
```

Smaller things

Colors in the REPL

```
allLength : List String -> List Nat
allLength [] = []
allLength (word :: words) = length word :: allLength words

xor : Bool -> Bool -> Bool
xor False y = y
xor True y = not y

isEven : Nat -> Bool
isEven Z = True
isEven (S k) = not (isEven k)
```

Figure 16: before-colorized-code

Colors in the REPL

```
allLength : List String -> List Nat
allLength [] = []
allLength (word :: words) = length word :: allLength words

xor : Bool -> Bool -> Bool
xor False y = y
xor True y = not y

isEven : Nat -> Bool
isEven Z = True
isEven (S k) = not (isEven k)
```

Figure 17: after-colorized-code

Definition ordering matters

```
mutual
  even : Nat -> Bool
  even Z = True
  even (S k) = odd k

  odd : Nat -> Bool
  odd Z = False
  odd (S k) = even k
```

Documentation comments

```
Idris> :doc List
Data type Prelude.List.List : Type -> Type
Generic lists
Constructors:
Nil : List elem
The empty list
(::) : elem -> List elem -> List elem
A non-empty list, consisting of a head element and the rest of
the list.
infixr 7
```

Pattern-matching with alternatives:

```
readNumbers : IO (Maybe (Nat, Nat))
readNumbers =
  do Just num1_ok <- readNumber | Nothing => pure Nothing
     Just num2_ok <- readNumber | Nothing => pure Nothing
     pure (Just (num1_ok, num2_ok))
```

Lighter update syntax

```
update_billing_address : Customer -> String -> Customer
update_billing_address customer address =
  record { account->address = address } customer
```

Apply function over a field

```
concat_billing_address : Customer -> String -> Customer
concat_billing_address customer complement =
  record { account->address $= (++ complement) } customer
```

Lossy Cast instead of Read

```
Idris> :doc Cast
Interface Cast
    Interface for transforming an instance of a data type to anot

Parameters:
    from, to

Methods:
    cast : Cast from to => (orig : from) -> to

the Int (cast "abc")
=> 0 : Int

the Int (cast "123")
=> 123 : Int
```

Records and projections have only one constructor

```haskell
data Foo = Foo { getFoo :: Int }
         | Bar { getBar :: Int }

getFoo (Bar 10)  -- BOOM!
```

- strings are not lists
- the %name directive lets you choose which names are generated upon code-generation
- shared base editor integration
- etc.?

Bigger things

Can't type-check this without a type-hint

```
import Data.Vect

total insSortKr : Ord elem => Vect n elem -> Vect n elem
insSortKr [] = []
insSortKr (x :: []) = [x]
insSortKr (x :: xs) = insertKr x (insSortKr xs)
  where
    -- insertKr : elem -> Vect n elem -> Vect (S n) elem
    insertKr el [] = [el]
    insertKr el (y :: ys) =
      case el > y of
        False => el :: y :: ys
        True => y :: (insertKr el ys)
```

Can't type-check this without a type-hint

```
- + Errors (1)
 `-- no_typehint_check.idr line 9 col 4:
     When checking left hand side of Main.insSortKr, insertKr:
     Type mismatch between
             Vect 0 elem (Type of [])
     and
             Vect len elem (Expected type)

     Specifically:
             Type mismatch between
                     0
             and
                     len
```

Type-hints via “the”

```
Idris> :doc the
Prelude.Basics.the : (a : Type) -> (value : a) -> a
    Manually assign a type to an expression.
    Arguments:
        a : Type  -- the type to assign

        value : a  -- the element to get the type

    The function is Total
Idris> the Int (cast "20")
20 : Int
```

Bound and unbound implicits

```
reverse : List elem -> List elem
append : Vect n elem -> Vect m elem -> Vect (n + m) elem

reverse : {elem : Type} -> List elem -> List elem
append : {elem : Type} -> {n : Nat} -> {m : Nat} ->
         Vect n elem -> Vect m elem -> Vect (n + m) elem

-- while internally it's:
append : {elem : _} -> {n : _} -> {m : _} ->
         Vect n elem -> Vect m elem -> Vect (n + m) elem
-- and inferred as:
append : {elem : Type} -> {n : Nat} -> {m : Nat} ->
         Vect n elem -> Vect m elem -> Vect (n + m) elem
```

Overloading names

```
Idris> the (List _) ["Hello", "There"]
["Hello", "There"] : List String
Idris> the (Vect _ _) ["Hello", "There"]
["Hello", "There"] : Vect 2 String
```

Record fields are namespaced

```
record Account where
  constructor MkAccount
  accound_id : String
  address : String

record Customer where
  constructor MkCustomer
  name : String
  address : String
  account : Account

custom_billing_account : Customer -> Bool
custom_billing_account customer =
  address customer /= address (account customer)
```

Separation of evaluation and execution

```
double_it : IO ()
double_it = do
  input <- getLine
  let n = the Int (cast input)
  printLn (n * 2)
```

Separation of evaluation and execution

```
[*src/InOut> double_it
io_bind (io_bind prim_read
                 (\x =>
                    io_pure (prim__strRev (with block in Prelude.

          (\input =>
             io_bind (prim_write (prim__concat (prim__toStrInt (pri
                     (\__bindx => io_pure ())) : IO ()
```

Separation of evaluation and execution

```
[*src/InOut> :exec double_it
123 -- My input
246 -- Output
```

Using implicit arguments in functions

```
-- instead of
length : Vect n elem -> Nat
length [] = Z
length (x :: xs) = 1 + length xs

-- you can do when you need to:
length : Vect n elem -> Nat
length {n} xs = n
```

Holes are OK to leave upon evaluation

```
allLengths : List String -> List Nat
allLengths [] = []
allLengths (word :: words) = length word :: ?rest

*WordLength> allLengths ["Hello", "Interactive", "Editors"]
5 :: ?rest : List Nat
```

Do notation without Monads

```
namespace CommandDo
  (>>=) : Command a -> (a -> Command b) -> Command b
  (>>=) = Bind
namespace ConsoleDo
  (>>=) : Command a -> (a -> Inf (ConsoleIO b)) -> ConsoleIO b
  (>>=) = Do
```

Laziness is opt-in, not opt-out

TODO

Type synonyms are just a special case of function application

```
import Data.Vect
Position : Type
Position = (Double, Double)

Polygon : Nat -> Type
Polygon n = Vect n Position

tri : Polygon 3
tri = [(0.0, 0.0), (3.0, 0.0), (0.0, 4.0)]
```

Big things

Totality checking

TODO

Implicit values

```
λΠ> :doc append
Main.append : (elem : Type) ->
    (n : Nat) -> (m : Nat) -> Vect n elem -> Vect m elem -> Vect
*Append> append _ _ _ ['a','b'] ['c','d']
['a', 'b', 'c', 'd'] : Vect 4 Char
*Append> append _ _ _ _ ['c','d']
(input):Can't infer argument n to append,
Can't infer explicit argument to append
```

Dependent tuples

```
readVect : IO (len ** Vect len String)
readVect = do x <- getLine
              if (x == "")
              then pure (_ ** [])
              else do (_ ** xs) <- readVect
                      pure (_ ** x :: xs)
```

ViewPatterns

Haskell in 1987, in his paper "Views: a way for pattern matching to cohabit with data abstraction."

Views as a programming idiom, using dependent types and a notation similar to the with notation in Idris, was later proposed by Conor McBride and James McKinna in their 2004 paper, "The view from the left."

Using views to eliminate the division by zero problem

```
import Data.Primitives.Views

arithInputs : Int -> Stream Int
arithInputs seed = map bound (randoms seed)
  where
    bound : Int -> Int
    bound num with (divides num 12)
      bound ((12 * div) + rem) | (DivBy prf) = rem + 1
```

Using views to eliminate the division by zero problem

```
λΠ> :doc divides
Data.Primitives.Views.Int.divides : (val : Int) -> (d : Int) -> D
    Covering function for the Divides view

    The function is Total
λΠ> :doc Divides
Data type Data.Primitives.Views.Int.Divides : Int -> (d : Int) ->
    View for expressing a number as a multiplication + a remainde

Constructors:
    DivByZero : Divides x (fromInteger 0)


    DivBy : (prf : rem >= fromInteger 0 && Delay (rem < d) = True
        Divides (d * div + rem) d
```

Demo

Outro

Good news

```
item :: pi (b :: Bool) -> Cond b Int [Int]
item True  = 42
item False = [1,2,3]
```

Acknowledgements and links

- Edwin Brady, for the book and the language
- https://deque.blog/2017/06/14/10-things-idris-improved-over-haskell/
- https://www.tweag.io/posts/2017-03-13-linear-types.html
- https://github.com/k-bx/nlp/blob/master/naiveml/src/NaiveML/MultinomialNaiveBayes.h
- https://stackoverflow.com/questions/37362342/difference-between-haskell-and-idris-reflection-of-runtime-compiletime-in-the-t

KyivHaskell and Haskell Study Group

Figure 18: kyivhaskell

After reading this book, TDD took on a new meaning for me

– Giovanni Ruggiero, Eligotech