Parking in Westminster: An Analysis in Haskell

OCTOBER 23, 2013OCTOBER 25, 2013 / DOMINIC STEINITZ

I had a fun weekend analysing car parking data in Westminster (http://www.westminster.gov.uk) at the Future Cities Hackathon (http://futurecitieshackathon.com) along with

- Amit Nandi
- Bart Baddeley
- Jackie Steinitz
- Ian Ozsvald (https://twitter.com/ianozsvald)
- Mateusz Łapsa-Malawski

Apparently in the world of car parking where Westminster leads the rest of UK follows. For example Westminster is rolling out individual parking bay monitors (http://www.bbc.co.uk/news/uk-england-london-19732371).

Our analysis gained an honourable mention. Ian has produced a great write-up (http://ianozsvald.com/2013/10/07/future-cities-hackathon-ds_ldn-oct-2013-on-parking-usage-inefficiencies) of our analysis with fine watercolour maps and Bart's time-lapse video of parking behaviour.

We mainly used Python, Pandas (http://pandas.pydata.org) and Excel for the actual analysis and QGIS (http://qgis.org/en/site) for the maps.

I thought it would be an interesting exercise to recreate some of the analysis in Haskell.

A Haskell Implementation

First some pragmas and imports.

```
> {-# OPTIONS_GHC -Wall                     #-}
> {-# OPTIONS_GHC -fno-warn-name-shadowing #-}
> {-# OPTIONS_GHC -fno-warn-orphans        #-}
>
> {-# LANGUAGE ScopedTypeVariables         #-}
> {-# LANGUAGE OverloadedStrings           #-}
> {-# LANGUAGE ViewPatterns                 #-}
> {-# LANGUAGE DeriveTraversable           #-}
> {-# LANGUAGE DeriveFoldable              #-}
> {-# LANGUAGE DeriveFunctor               #-}
>
> module WardsOfLondon ( parkingDia ) where
>
> import Database.Shapefile
>
> import Data.Binary.Get
> import qualified Data.ByteString.Lazy as BL
> import qualified Data.ByteString as B
> import Data.Binary.IEEE754
> import Data.Csv hiding ( decode, lookup )
> import Data.Csv.Streaming
> import qualified Data.Vector as V
> import Data.Time
> import qualified Data.Text as T
> import Data.Char
> import qualified Data.Map.Strict as Map
> import Data.Int( Int64 )
> import Data.Maybe ( fromJust, isJust )
> import Data.List ( unfoldr )
>
> import Control.Applicative
> import Control.Monad
>
> import Diagrams.Prelude
> import Diagrams.Backend.Cairo.CmdLine
>
> import System.FilePath
> import System.Directory
> import System.Locale
> import System.IO.Unsafe ( unsafePerformIO )
>
> import Data.Traversable ( Traversable )
> import qualified Data.Traversable as Tr
> import Data.Foldable ( Foldable )
```

A type synonym to make typing some of our functions a bit more readable (and easier to modify e.g. if we want to use Cairo).

```
> type Diag = Diagram Cairo R2
```

The paths to all our data.

- SHP files are shape files (http://en.wikipedia.org/wiki/Shapefile), a fairly old but widespread map data format that was originally produced by a company called ESRI.
- The polygons (http://data.london.gov.uk/datastore/package/i-trees-canopy-ward-data) for the outline of the wards in Westminster. Surely there is a better place to get this rather than using tree canopy data.
- The polyline data (http://download.geofabrik.de/europe/great-britain.html) for all the roads (and other stuff) in the UK. We selected out all the roads in a bounding box for London. Even so plotting these takes about a minute.
- The parking data were provided by Westminster Council. The set we consider below was about 4 million lines of cashless parking meter payments (about 1.3G).

```
> prefix :: FilePath
> prefix = "/Users/dom"
>
> dataDir :: FilePath
> dataDir = "Downloadable/DataScienceLondon"
>
> borough :: FilePath
> borough = "WestMinster"
>
> parkingBorough :: FilePath
> parkingBorough = "ParkingWestminster"
>
> flGL :: FilePath
> flGL = prefix </> dataDir </> "GreaterLondon
>
> flParkingCashless :: FilePath
> flParkingCashless = "ParkingCashlessDenorm.c
```

The data for payments are contained in a CSV file so we create a record in which to keep the various fields contained therein.

```
> data Payment = Payment
>                { _amountPaid       :: LaxDou
>                ,  paidDurationMins :: Int
>                ,  startDate        :: UTCTim
>                , _startDay         :: DayOfT
>                , _endDate          :: UTCTim
>                , _endDay           :: DayOfT
>                , _startTime        :: TimeOf
>                , _endTime          :: TimeOf
>                , _designationType  :: T.Text
>                ,  hoursOfControl   :: T.Text
>                , _tariff           :: T.Text
>                , _maxStay          :: T.Text
>                , spaces            :: Maybe
>                , _street           :: T.Text
>                , _xCoordinate      :: Maybe
>                , _yCoordinate      :: Maybe
>                , latitude          :: Maybe
>                , longitude         :: Maybe
>                }
>   deriving Show
>
> data DayOfTheWeek = Monday
>                   | Tuesday
>                   | Wednesday
>                   | Thursday
>                   | Friday
>                   | Saturday
>                   | Sunday
>   deriving (Read, Show, Enum)
```

We need to be able to parse the day of the week.

```
> instance FromField DayOfTheWeek where
>   parseField s = read <$> parseField s
```

The field containing the longitude has values of the form -.1. The CSV parser for *Double* will reject this so we create our own datatype with a more relaxed parser.

```
> newtype LaxDouble = LaxDouble { laxDouble ::
>   deriving Show
>
> instance FromField LaxDouble where
>   parseField = fmap LaxDouble . parseField .
>
>     where
>
>       addLeading :: B.ByteString -> B.ByteSt
>       addLeading bytes =
>             case B.uncons bytes of
>               Just (c -> '.', _)    -> B.con
>               Just (c -> '-', rest) -> B.con
>               _ -> bytes
>
>       c = chr . fromIntegral
>       o = fromIntegral . ord
```

We need to be able to parse dates and times.

```
> instance FromField UTCTime where
>   parseField s = do
>     f <- parseField s
>     case parseTime defaultTimeLocale "%F %X"
>       Nothing -> fail "Unable to parse UTC t
>       Just g  -> return g
>
> instance FromField TimeOfDay where
>   parseField s = do
>     f <- parseField s
>     case parseTime defaultTimeLocale "%R" f
>       Nothing -> fail "Unable to parse time
>       Just g  -> return g
```

Finally we can write a parser for our record.

```
> instance FromRecord Payment where
>   parseRecord v
>          | V.length v == 18
>          = Payment <$>
>            v .!  0 <*>
>            v .!  1 <*>
>            v .!  2 <*>
>            v .!  3 <*>
>            v .!  4 <*>
>            v .!  5 <*>
>            v .!  6 <*>
>            v .!  7 <*>
>            v .!  8 <*>
>            v .!  9 <*>
>            v .! 10 <*>
>            v .! 11 <*>
>            v .! 12 <*>
>            v .! 13 <*>
>            v .! 14 <*>
>            v .! 15 <*>
>            v .! 16 <*>
>            v .! 17
>          | otherwise = mzero
```

To make the analysis simpler, we only look at what might be a typical day, a Thursday in February.

```
> selectedDay :: UTCTime
> selectedDay = case parseTime defaultTimeLoca
>   Nothing -> error "Unable to parse UTC time
>   Just t  -> t
```

It turns out that there are a very limited number of different sorts of hours of control so rather than parse this and calculate the number of control minutes per week, we can just create a simple look up table by hand.

```
> hoursOfControlTable :: [(T.Text, [Int])]
> hoursOfControlTable = [
>     ("Mon - Fri 8.30am - 6.30pm"
>   , ("Mon-Fri 10am - 4pm"
>   , ("Mon - Fri 8.30-6.30 Sat 8.30 - 1.30"
>   , ("Mon - Sat 8.30am - 6.30pm"
>   , ("Mon-Sat 11am-6.30pm "
>   , ("Mon - Fri 8.00pm - 8.00am"
>   , ("Mon - Fri 8.30am - 6.30pm "
>   , ("Mon - Fri 10.00am - 6.30pm\nSat 8.30am
>   , ("Mon-Sun 10.00am-4.00pm & 7.00pm - Midn
>   ]
```

Now we create a record in which to record the statistics in which we are interested:

- Number of times a lot is used.
- Number of usage minutes. In reality this is the amount of minutes purchased and people often leave a bay before their ticket expires so this is just a proxy.
- The hours of control for the lot.
- The number of bays in the lot.

N.B. The !'s are *really* important otherwise we get a space leak. In more detail, these are strictness annotations which force the record to be evaluated rather than be carried around unevaluated (taking up unnecessary space) until needed.

```
> data LotStats = LotStats { usageCount
>                          , usageMins
>                          , usageControlTxt
>                          , usageSpaces
>                          }
>   deriving Show
```

As we work our way through the data we need to update our statistics.

```
> updateStats :: LotStats -> LotStats -> LotSt
> updateStats s1 s2 = LotStats { usageCount =
>                              , usageMins  =
>                              , usageControlT
>                              , usageSpaces =
>                              }
>
> initBayCountMap :: Map.Map (Pair Double) Lot
> initBayCountMap = Map.empty
```

We are going to be working with co-ordinates which are pairs of numbers so we need a data type in which to keep them. Possibly overkill.

```
> data Pair a = Pair { xPair :: !a, yPair :: !
>   deriving (Show, Eq, Ord, Functor, Foldable
```

Functions to get bounding boxes.

```
> getPair :: Get a -> Get (a,a)
> getPair getPart = do
>     x <- getPart
>     y <- getPart
>     return (x,y)
>
> getBBox :: Get a -> Get (BBox a)
> getBBox getPoint = do
>     bbMin <- getPoint
>     bbMax <- getPoint
>     return (BBox bbMin bbMax)
>
> bbox :: Get (BBox (Double, Double))
> bbox = do
>   shpFileBBox <- getBBox (getPair getFloat64
>   return shpFileBBox
>
> getBBs :: BL.ByteString -> BBox (Double, Dou
> getBBs = runGet $ do
>   _ <- getShapeType32le
>   bbox
>
> isInBB :: (Ord a, Ord b) => BBox (a, b) -> B
> isInBB bbx bby = ea >= eb && wa <= wb &&
>                  sa >= sb && na <= nb
>   where
>     (ea, sa) = bbMin bbx
>     (wa, na) = bbMax bbx
>     (eb, sb) = bbMin bby
>     (wb, nb) = bbMax bby
>
> combineBBs :: (Ord a, Ord b) => BBox (a, b)
> combineBBs bbx bby = BBox { bbMin = (min ea
>                           , bbMax = (max wa
>                           }
>   where
>     (ea, sa) = bbMin bbx
>     (wa, na) = bbMax bbx
>     (eb, sb) = bbMin bby
>     (wb, nb) = bbMax bby
```

A function to get plotting information from the shape file.

```
> getRecs :: BL.ByteString ->
>             [[(Double, Double)]]
> getRecs = runGet $ do
>   _ <- getShapeType32le
>   _ <- bbox
>   nParts  <- getWord32le
>   nPoints <- getWord32le
>   parts   <- replicateM (fromIntegral nParts
>   points  <- replicateM (fromIntegral nPoint
>   return (getParts (map fromIntegral parts)
>
> getParts :: [Int] -> [a] -> [[a]]
> getParts offsets ps = unfoldr g (gaps, ps)
>   where
>     gaps = zipWith (-) (tail offsets) offset
>     g (  [],   []) = Nothing
>     g (  [],   xs) = Just (xs, ([], []))
>     g (n:ns,   xs) = Just (take n xs, (ns, d
```

We need to be able to filter out e.g. roads that are not in a given bounding box.

```
> recsOfInterest :: BBox (Double, Double) -> [
> recsOfInterest bb = filter (flip isInBB bb .
```

A function to process each ward in Westminster.

```
> processWard :: [ShpRec] -> FilePath ->
>                IO ([ShpRec], ([[(Double, Dou
> processWard recDB fileName = do
>   input <- BL.readFile $ prefix </> dataDir
>   let (hdr, recs) = runGet getShpFile input
>       bb          = shpFileBBox hdr
>   let ps  = head $ map getRecs (map shpRecDa
>   return $ (recsOfInterest bb recDB, (ps, bb
```

We want to draw roads and ward boundaries.

```
> colouredLine :: Double -> Colour Double -> [
> colouredLine thickness lineColour xs = (from
>                                        lw th
>                                        lc li
```

And we want to draw parking lots with the hue varying according to how heavily they are utilised.

```
> bayDots :: [Pair Double] -> [Double] -> Diag
> bayDots xs bs = position (zip (map p2 $ map
>   where dots     = map (\b -> circle 0.0005
>         toPair p = (xPair p, yPair p)
>         c1       = darkgreen  `withOpacity`
>         c2       = lightgreen `withOpacity`
```

Update the statistics until we run out of data.

```
> processCsv :: Map.Map (Pair Double) LotStats
>               Records Payment ->
>               Map.Map (Pair Double) LotStats
> processCsv m rs = case rs of
>   Cons u rest -> case u of
>     Left err ->  error err
>     Right val -> case Tr.sequence $ Pair (la
>       Nothing -> processCsv m rest
>       Just v  -> if startDate val == selecte
>                  then processCsv (Map.insert
>                  else processCsv m rest
>         where
>           delta = LotStats { usageCount = 1
>                            , usageMins  = fr
>                            , usageControlTxt
>                            , usageSpaces
>                            }
>   Nil mErr x  -> if BL.null x
>                  then m
>                  else error $ "Nil: " ++ sho
```

```
> availableMinsThu :: LotStats -> Maybe Double
> availableMinsThu val =
>   fmap fromIntegral $
>   fmap (!!(fromEnum Thursday)) $
>   flip lookup hoursOfControlTable $
>   usageControlTxt val
```

Now for the main function.

```
> parkingDiaM :: IO Diag
> parkingDiaM = do
```

Read in the 4 million records lazily.

```
>   parkingCashlessCsv <- BL.readFile $
>                         prefix </>
>                         dataDir </>
>                         parkingBorough </>
>                         flParkingCashless
```

Create our statistics.

```
>   let bayCountMap = processCsv initBayCountM
>
>       vals = Map.elems bayCountMap
```

Calculate the available minutes for each bay.

```
>       availableMinsThus :: [Maybe Double]
>       availableMinsThus = zipWith f (map ava
>                                      (map (fm
>         where
>           f x y = (*) <$> x <*> y
```

Calculate the actual minutes used for each lot and the usage which determine the hue of the colour of the dot representing the lot on the map.

```
>       actualMinsThu :: [Double]
>       actualMinsThu =
>         map fromIntegral $
>         map usageMins vals
>
>       usage :: [Maybe Double]
>       usage = zipWith f actualMinsThu availa
>         where
>           f x y = (/) <$> pure x <*> y
```

We will need to the co-ordinates of each lot in order to be able to plot it.

```
>   let parkBayCoords :: [Pair Double]
>       parkBayCoords = Map.keys bayCountMap
```

Get the ward shape files.

```
>   fs <- getDirectoryContents $ prefix </> da
>   let wardShpFiles = map (uncurry addExtensi
>            filter ((==".shp"). snd) $
>            map splitExtension fs
```

Get the London roads shape file.

```
>   inputGL <- BL.readFile flGL
>   let recsGL = snd $ runGet getShpFile input
```

Get the data we wish to plot from each ward shape file.

```
>   rps <- mapM (processWard recsGL) wardShpFi
```

Get the roads inside the wards.

```
>   let zs = map (getRecs . shpRecData) $ conc
```

And create blue diagram elements for each road.

```
>       ps :: [[Diag]]
>       ps = map (map (colouredLine 0.0001 blu
```

Create diagram elements for each ward boundary.

```
>       qs :: [[Diag]]
>       qs = map (map (colouredLine 0.0003 nav
```

Westminster is located at about 51 degrees North. We want to put a background colour on the map so either we need to move Westminster to be at the origin or create a background rectangle centred on Westminster. We do the former. We create a rectangle which is slightly bigger than the bounding box of Westminster. And we translate everything so that the South West corner of the bounding box of Westminster is the origin.

```
>   let bbWestminster = foldr combineBBs (BBox
>                       map (snd . snd) rps
>         where
>           inf     = read "Infinity"
>           negInf  = read "-Infinity"
>
>   let (ea, sa) = bbMin bbWestminster
>       (wa, na) = bbMax bbWestminster
>       wmHeight = na - sa
>       wmWidth  = wa - ea
```

Create the background.

```
>       wmBackground = translateX (wmWidth / 2
>                      translateY (wmHeight /
>                      scaleX 1.1 $
>                      scaleY 1.1 $
>                      rect wmWidth wmHeight #
```

Plot the streets.

```
>       wmStreets =  translateX (negate ea) $
>                    translateY (negate sa) $
>                    mconcat (mconcat ps)
```

Plot the parking lots.

```
>       wmParking = translateX (negate ea) $
>                   translateY (negate sa) $
>                   uncurry bayDots $
>                   unzip $
>                   map (\(x, y) -> (x, fromJu
>                   filter (isJust . snd) $
>                   zip parkBayCoords usage
```

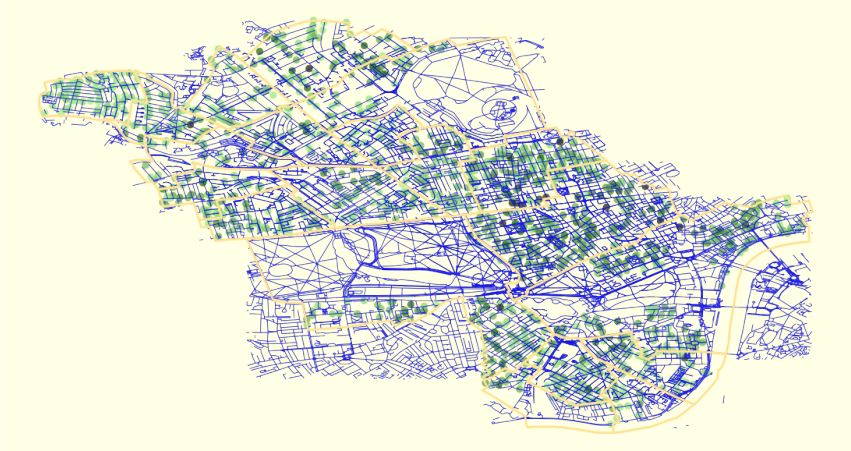
Plot the ward boundaries.

```
>       wmWards = translateX (negate ea) $
>                 translateY (negate sa) $
>                 mconcat (mconcat qs)
>
>   return $ wmBackground <>
>            wmWards <>
>            wmStreets <>
>            wmParking
```

Sadly we have to use *unsafePerformIO* in order to be able to create the post using BlogLiteratelyD.

```
> parkingDia :: Diag
> parkingDia = unsafePerformIO parkingDiaM
```

And now we can see all the parking lots in Westminster as green dots. The darkness represents how heavily utilised they are. The thick gold lines delineate the wards in Westminster. In case it isn't obvious the blue lines are the roads. The Thames, Hyde Park and Regent's Park are fairly easy to spot. Less easy to spot but still fairly visible are Buckingham Palace and Green Park.



Observations

- We appear to need to use *ghc -O2* otherwise we get a spaceleak.
- We didn't explicitly need the equivalent of pandas. It would be interesting to go through the Haskell and Python code and see where we used pandas and what the equivalent was in Haskell.
- Python and R seem more forgiving about data formats e.g. they handle -.1 where Haskell doesn't. Perhaps this should be in the Haskell equivalent of pandas.

Haskell, Statistics

4 thoughts on "Parking in Westminster: An Analysis in Haskell"

1. Pingback: Future Cities Hackathon (@ds_ldn) Oct 2013 on Parking Usage Inefficiencies | Entrepreneurial Geekiness
2. Pingback: Diagrams 1.0 | blog :: Brent -> [String]
3. Pingback: Cartography in Haskell | Maths, Stats & Functional Programming
4. Pingback: Cartography in Haskell – essay studess