Introduction

Aim: Promote modular software design as a foundation for interactive data-driven apps

Case Study: Inbound airport passenger management in the form of an interactive shiny web app.

Background: Interactive visualisation via apps is an effective way to gain insights from complex data analysis and exploration. However, app complexity typically increases with the number and nature of included features. The shiny R package is a powerful tool for interactive data science communication. It connects the underlying analytical platform to the end users via a responsive web interface and features a built-in modularisation framework to mitigate app development complexity. This helps to manage app behaviour in the reactive code execution framework (Goleumund, 2015) that underpins shiny.

Results: shiny is an excellent deployment tool to present data-driven solutions implemented in R to broad audiences as it facilitates rapid prototyping and development, and modular app architecture without the need to re-implement core analytics components.

Methodology

- **Build an airport simulation model** to predict changes in queuing behaviour in a network of queues that represent the inbound passenger facilitation process in an international terminal.
- **Implement this in R** (R Core Team, 2017) based on the schema in Wu et al. (2014) (Figure 1).
- **Select metrics from model outputs** and design a simple graphical visualisation as the basis of our data visualisation framework. Key features: interpretability & consistency.
- **Construct a dashboard app in R** using shiny and shinydashboard. Leverage the modularisation framework in shiny (Cheng, 2017) to manage app complexity. Create a user input module to enable users to modify key input parameters and submit custom simulation forecast requests and understand the potential impact of operational changes.

Results

**App development**: Modular design that enabled rapid development of an interactive visualisation app based on a complex data modelling engine; robust, reusable and updatable app components; individual components that could be seamlessly integrated into a single, navigable design to systematically communicate results to end users.

**Data visualisation framework**: The app communicates a system state via two metrics, occupancy (passenger counts) and dwell time (area clearance time). For simplicity, SEA processing represents an arbitrary combination of examination by customs or quarantine personnel, after which passengers immediately exit the system.

**Key**

- **System section**: Red (slowest) ➔ Orange (moderate) ➔ Green (fastest)
- **System queue**: Exit ➔ SmartGate ➔ SmartGate ➔ ECP ➔ BH ➔ Gate n ➔ Gate 1 ➔ Gate 2 ➔ …
- **Passenger flow**: AC ➔ SmartGate ➔ M: Australia ➔ M: Foreign ➔ SEA ➔ Exit

Figure 1: Schema of airport system used to construct simulation model. Simulated passengers enter the system on inbound flights at preassigned arrival gates represented as individual queue routing models passing through the Airside Concourse (AC). Transit through the AC to the immigration Entry Control Point (ECP) is defined by a combination of gate queue departure time and simulated mobility demographics randomly assigned to each passenger. Nationality and other routing demographics for Australian and New Zealand citizens are defined by a complex system of unique or sequential processing strategies (M) for Australia (Aus) and foreign passport holders, then are also computed to assign virtual passengers to one of the four immigration queues. Passengers are randomly assigned an origin or more baggage items with associated arrival and screening demographics. The baggage collection process is captured in the Baggage Hall (BH), before passenger screening in the Secondary Examination Area (SEA). This queuing simulation is designed to be easily extended to include additional queue routes, queuing model parameters, and the additional complexity of a dynamic airport environment.

**Discussion**

Benefits of app development through modularisation techniques:

- Improved overall app quality though focused development of robust app subunits that are integrated into a cohesive product. Componentised app logic is easier to develop and debug, which enhances code reliability and focuses development efforts.
- Reduction in overall app development time with faster feature development cycles as modular components are readily reconfigurable for reuse. Further, therefore new features can be developed in isolation with minimal disruption of the existing codebase.
- Improved data communication outcomes as careful design of modular visualisation components results in the seamless establishment of consistent visual semantics throughout the app, improving data comparability.
- Methodology is scalable and generalizable.

**Acknowledgements**

This work was funded through the Australian Research Council (ARC) linkage grant “Improving the Productivity and Efficiency of Australian Airports” (LP140100282).

**References**