UNIVERSIDAD TORCUATO DI TELLA

PREDICCIÓN DE CHURN EN MOBILE GAMING (II)

DATA MINING - MAESTRIA MIM

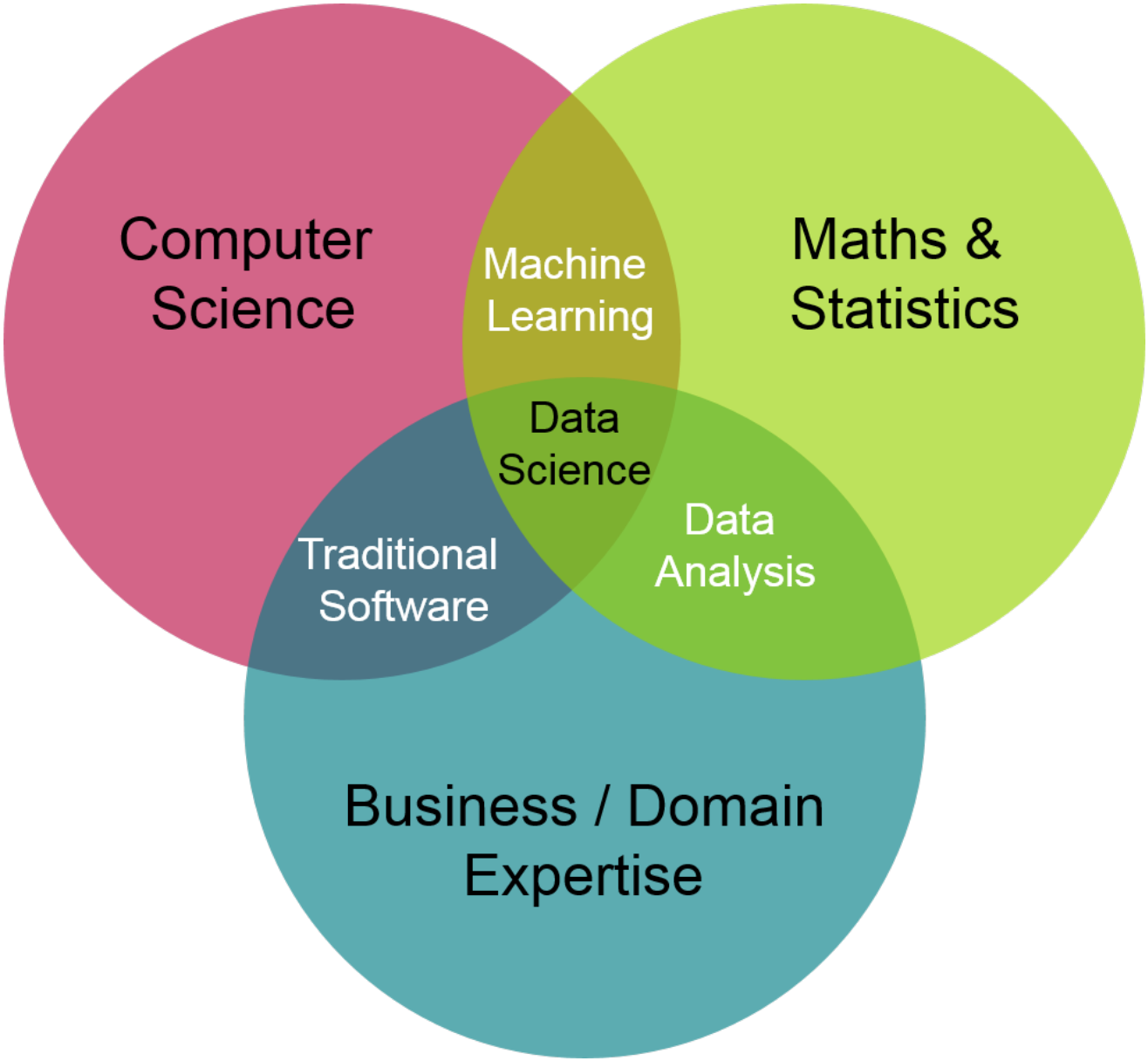
Pedro Ferrari

24 de Abril de 2021

AGENDA

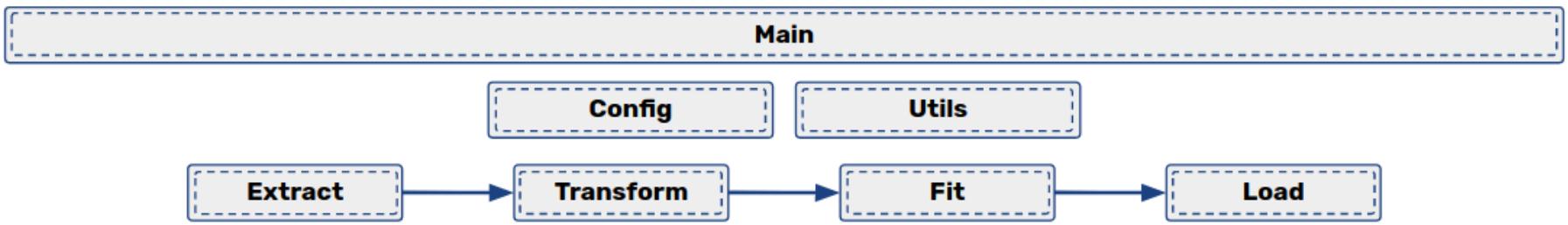
1. **Un Plan de Ataque (General) al Problema**
2. **Consultas y Trabajo en el TP**

¿QUÉ ES DATA SCIENCE?



¿QUÉ ES HACER OPERATIVAMENTE DATA SCIENCE?

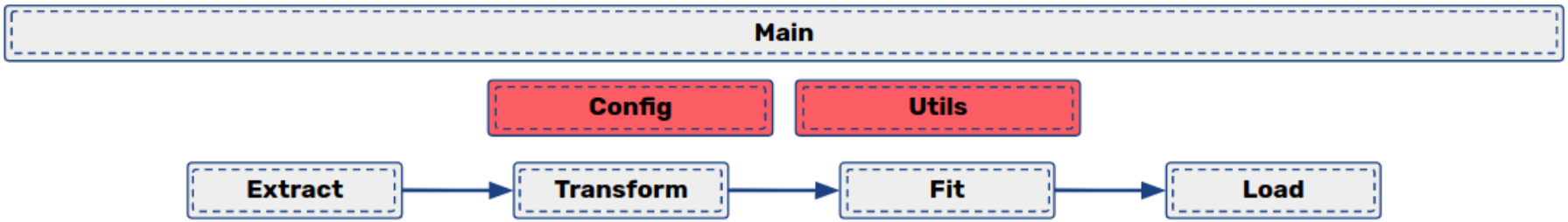
- **ETFL**: **E**xtract **T**ransform **F**it **L**oad
  - Distribución de tiempos: E 30 %, T 50 %, F 10 %, L 10 %
- ¿Podemos **modularizar** el código de esta manera? En general sí...
  - Módulos típicos de un proyecto de DS:
    - `cfg.R`: contiene variables de configuración (constantes) usadas en todo el proyecto (Ej: directorios locales donde se guardan archivos intermedios).
    - `extract.R`: encargado de extraer o cargar datos de una o varias fuentes (si hay una base de datos entonces involucra SQL).
    - `transform.R`: recibe datos y los transforma para que puedan ser usados por el modelo.
    - `fit.R`: estima modelos, genera predicciones y computa métricas de performance.
    - `load.R`: carga las predicciones/estimaciones en algún sitio donde puedan ser fácilmente consumidas.
    - `utils.R`: funciones auxiliares usadas de forma frecuente en el proyecto.
    - `main.R`: contiene la lógica principal del programa construida a partir de llamadas a los otros módulos.



BUENAS PRÁCTICAS (Y MUCHOS ACRÓNIMOS)

- Logging
- DRY (*don't repeat yourself*) y SRP (*single responsibility principle*): facilita el debugging (via breakpoints)
- CLI (*command line interface*)
- VCS (*version control system*): Git!
- TDD (*test driven development*)
- Buenas API (application programming interface)

DRY, SRP Y DEBUGGING: MODULOS CFG.R Y UTILS.R



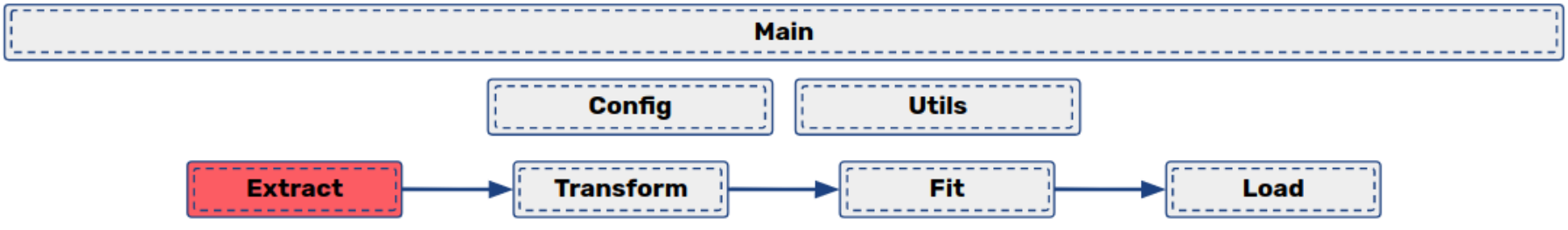
- Estrategia general: buscar repeticiones y abstraer
- Veamos ejemplos de esto en el código....
  - Si buscan la siguiente función verán que es usada más de 10 veces.

```
library(glue)

vec_to_str <- function(cols, sorted = TRUE) {
    if (sorted) {
        cols <- sort(cols)
    }
    return(glue("{cols}", cols = glue_collapse(cols, ", ")))
}

R>> vec_to_str(c("b", "a"))
a, b
```

LOGGING, CLI Y VCS: MODULO EXTRACT.R



- Veamos otra vez el código...
  - CLI:

```
$> Rscript extract.R --help
```

  - CLI y logging:

```
$> Rscript extract.R --in-dir ~/fasdlkefe
```

  - Cargar y guardar 10 % de los datos de entrenamiento (y mirar el log!)

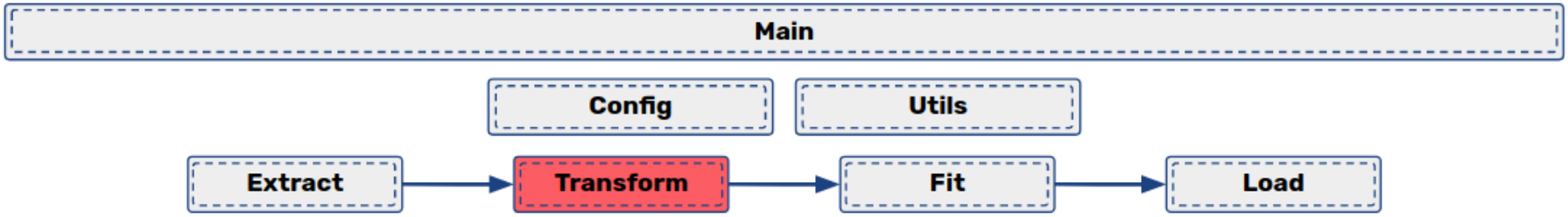
```
$> Rscript extract.R --in-dir dir_con_datos --phase train --sample-ratio 0.1
```

  - Cargar y guardar 30 % de los datos de evaluación

```
$> Rscript extract.R --in-dir dir_con_datos --phase evaluation --sample-ratio 0.3
```

- ¿Cuál es la historia de este archivo? -> VCS y en particular Git.

EL MÓDULO TRANSFORM.R



- Encargado de todo el preprocesamiento de los datos. *Heavy lifting* hecho por la función `transform_pipeline`:
  - (i) `drop_useless_`: Elimina columns que sabemos de antemano que no vamos a usar (alimentado por el EDA previo).
  - (ii) `drop_duplicate_rows`: Elimina filas repetidas.
  - (iii) `drop_nulls_`: Elimina columnas con porcentaje de nulos mayor a un umbral.
  - (iv) `drop_zero_var_`: Elimina columnas constantes (con varianza nula).
  - (v) `make_new_cols_`: Hace feature engineering...
    - Genera atributos a partir conteo de tipos de usuarios, agregación de eventos, etc.
  - (vi) `ensure_dtypes_`: Chequea que las columnas sean numéricas o factores exclusivamente.
  - (vii) `fill_and_scale_numeric_`: Rellena valores nulos de las columnas numéricas con sus mediana y las re-escala ("estandariza").
  - (viii) `encode_categorical_and_split_`: Codifica variables categóricas usando "One Hot" o "Feature Hashing". Además separa el conjunto de entrenamiento del de test usando matrices ralas.

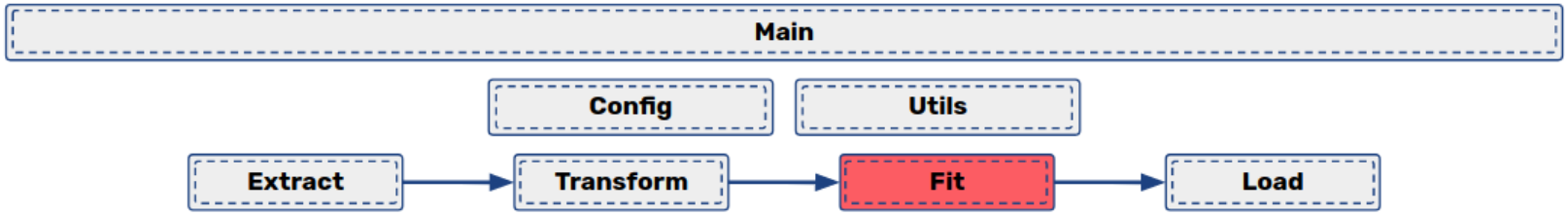
ONE HOT ENCODING VS FEATURE HASHING

- ¿Cómo trabajar con variables categóricas?
  - Usar algoritmos que las acepten per se...
  - *One Hot Encoding* o "Dummyficar": transformar variable de n observaciones y k valores distintos en k variables binarias de n observaciones cada uno.
    - Si hay muchos niveles entonces se pueden eliminar algunas (por ejemplo tomando los de mayor frecuencia) o usar matrices ralas.
    - ¿Sirve OHE en nuestra aplicación? Veamos el código...

```
$> Rscript transform.R --encoder one_hot
```

- *Feature Hashing* o *Hashing Trick* al rescate...
  - Estrategia:
    (i) Prealocar vector de tamaño n
    (ii) Aplicar una función de hash h sobre cada elemento s de cada vector de features categórico.
    (iii) Updatear el vector de salida en el indice $h(s)mod(n)$.
  - Ventajas:
    - No requiere (prácticamente) preprocesamiento de la data
    - Trabaja con direcciones (índices) con lo cual usa mucha menos memoria
    - Elimina el problema de tener mismas variables en train y test (donde potencialmente aparecen nuevas)

EL MÓDULO FIT.R: REGRESIÓN LOGÍSTICA Y REGULARIZACIÓN

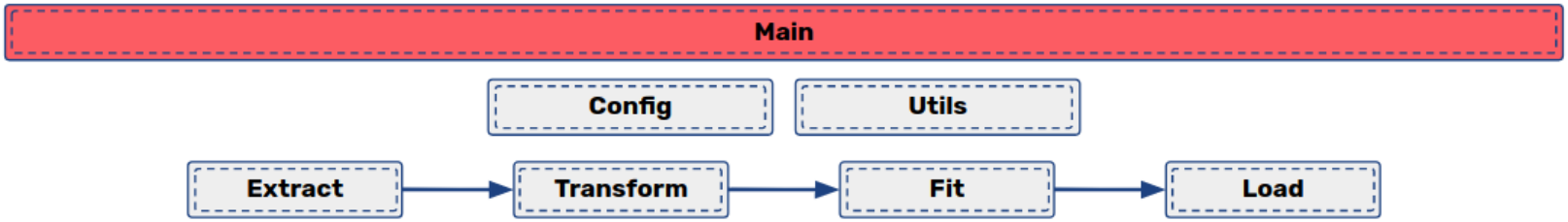


- Contexto de clasificación binaria:
  - Y_t vale 1 (0) si ocurre (no ocurre) determinado comportamiento a tiempo t; X_{t-k} variable explicativa (agregada)
  - Regresión lineal simple: $Y_t = \beta_0 + \beta_1 X_{t-k} + \varepsilon_t$
    - Usar $\hat{Y}$ para clasificar (ej: es clickeador si $\hat{Y} > 0.5$). ¿Problema?
  - Mejor modelar directamente $\Pr(Y = 1|X)$ como $\Pr(Y_t = 1) = F(\beta_0 + \beta_1 X_{t-k})$
    - Donde $F(\cdot)$ es la función logística dada por $F(\cdot) = \frac{1}{1+e^{-x}}$
- ¿Cómo estimamos los coeficientes?
  - Intuitivamente buscamos $\hat{\beta}_i$ tal que que $\Pr(\cdot)$ vale cercano a 1 si hay click y 0 en caso contrario.
    - Implica maximizar numéricamente (el logaritmo) de la función de verosimilitud:

$$\ln \mathcal{L}(Y_t, \beta_0, \beta_1) = \sum_{t=1}^{N} (Y_t \ln(F(\cdot)) + (1 - Y_t) \ln(1 - F(\cdot)))$$

    - No hay forma cerrada, hay que hacerlo numéricamente -> descenso de gradiente
  - ¿Podemos imponer restricciones a los coeficientes?
    - Técnicas como Lasso (L1) o Ridge (L2) o Elastic Net...

EL MÓDULO MAIN.R Y ALGUNOS RESULTADOS



- Probemos en el código lo siguiente:
  1. Regresión logística con 10 % de los datos de entrenamiento.

```
$> Rscript fit.R --train-ratio 0.1 --eval-ratio 0.3 --classifier lr
```

  2. XGBoost con 10 % de los datos.

```
$> Rscript fit.R --train-ratio 0.1 --eval-ratio 0.3 --classifier xgb
```

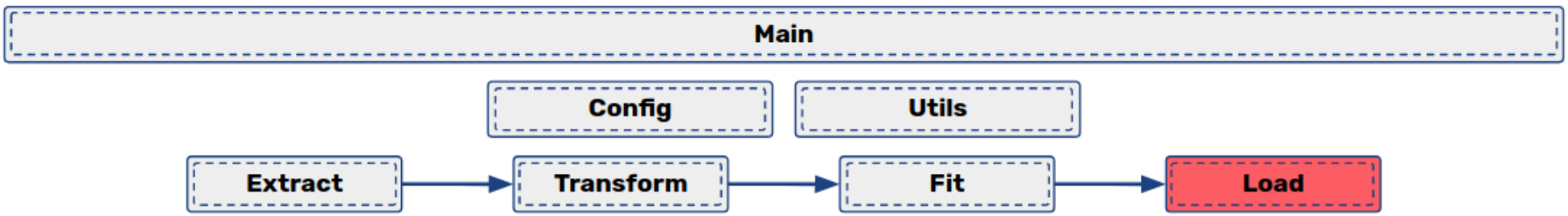
     - ¿Qué sucede si lo corremos con 15 %? ¿Y con 40 %?
- Podemos correr el programa completo desde `main.R`

```
$> Rscript main.R --train-ratio 0.1 --eval-ratio 0.3
```

  - Algunas observaciones para obtener un buen score
    - Dada la existencia de un feature dominante es importante crear otros nuevos features: "Applied Machine Learning is Basically Feature Engineering" (Andrew Ng)
    - Mejoras (decrecientes) con aumentos en el tamaño de muestra.
    - Modelos más complejos parecen hacer la diferencia (árboles profundos y muchas iteraciones son parámetros a tunear).

POSIBLES (NECESARIAS) MEJORAS

- El score es razonablemente bueno....
  - Tengo el dataset de test etiquetado.
- Hay espacios para mejoras....
  - Performance del modelo:
    - Mejorar la ingeniería de atributos: por ejemplo probar interacciones entre los features (en caso de usar la regresión logística)
    - Tunear hiper parámetros
    - Usar técnicas de under/over sampling (por ejemplo tunear parametro `scale_pos_weight` usando ratio de negativos a positivos)
    - Probar otros modelos (como Random Forest) y pensar en ensamble de modelos
  - El código:
    - Agregar una función que detecte y elimine/impute outliers.
    - Loggear parámetros entre distintas corridas de los modelos
    - Escribir el modulo `load.R` que emplee la API de Kaggle para subir resultados.
    - Escribir tests unitarios y de integración



¿Qué (otras) cosas probaron ustedes?

AGENDA

1. Un Plan de Ataque (General) al Problema

2. **Consultas y Trabajo en el TP**