UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES ACATLAN

ANÁLISIS ESTADÍSTICO

Simulación Estocástica aplicada a un negocio de comida

TRABAJO FINAL

SIMULACIÓN ESTOCÁSTICA

P R E S E N T A :

Tapia Peñaloza Jorge Luis
Roldán Sanchez Ulises

GUSTAVO FUENTES

Introducción

Como clientes de un negocio de comida, además de buscar satisfacer nuestro paladar, buscamos opciones que nos ofrezcan un servicio rápido y eficiente. Resulta molesto tener que esperar por una mesa, esperar a que te tomen la orden y sobre todo esperar a que los platillos estén listos para servirse. La demanda que tenga un negocio de comida dependerá de diversos factores como la calidad de sus productos, el precio, la ubicación, la hora, la disposición del lugar y la rapidez del servicio.

Nuestro proyecto tiene como principal objetivo el análisis de servicio a comensales a un restaurante. Para ello tomamos en cuenta el tiempo transcurrido entre la llegada de un grupo de comensales y cuando el mesero toma la orden; el tiempo transcurrido entre la toma de orden y la llegada de los platillos a la mesa; y el tiempo que pasa hasta que los comensales terminan sus alimentos, pagan y se van del establecimiento, dejando así la mesa libre para otros comensales. Con estos datos buscamos dar opciones que optimicen la calidad de servicio en un restaurante.

La Simulación Estocástica nos permite analizar diversos problemas mediante la simulación artificial de eventos, dicho con otras palabras, con ayuda de la computación recreamos eventos aleatorios para analizar fenómenos deterministas. Esta poderosa herramienta tiene la facultad de ser baja en costos, altamente precisa en predicciones y es aplicable a prácticamente cualquier problemática relacionada a eventos estocásticos. La simulación fue el principal motor de este proyecto. Pandas, NumPy y Fitter son algunas de las librerías y paquetes de Python que nos permitieron realizar dicha simulación.

El restaurante que tomamos para desarrollar nuestro proyecto lleva por nombre ”El Asadero”, se localiza en Plaza San Mateo, en Naucalpan de Juárez, Estado de México. Se trata de un establecimiento que vende diferentes antojitos mexicanos como sopes, flautas, quesadillas, tortas, gorditas, etc; pero su especialidad son los pozoles y los tacos al pastor. Tiene capacidad para aproximadamente setenta y dos comensales distribuidos en mesas con capacidad para seis personas, atendidos por seis meseros, con cinco personas en la cocina y una fungiendo como cajero. El restaurante se divide en dos pisos, pero el piso de arriba está inhabilitado para dar servicio.

Nos dimos a la tarea de colectar la información en dicho restaurante por cinco horas continuas, registrando el número de comensales que ocupaban una mesa contándolos por sexo, su hora de llegada, la hora en que les tomaban la orden, la hora en que sus platillos llegaban a la mesa y la hora en que salían del establecimiento. Posteriormente realizamos la simulación que nos permitió el análisis de los eventos estocásticos. En este trabajo presentamos los resultados de dicho análisis.

Objetivo

Dar recomendaciones que optimicen la administración en un negocio de comida, definir si el espacio es suficiente para alojar a sus clientes; con la ayuda de la Simulación Estocástica realizada a partir de datos reales recolectados por nosotros mismos.

Analizar si se debe o no habilitar el segundo piso del establecimiento para dar servicio tomando en cuenta diferentes escenarios y la cantidad de comensales que esperan por una mesa.

Python

Python es un lenguaje de programación de alto nivel, interpretado, de propósito general, cuya expansión y popularidad es relativamente reciente. Python es un lenguaje de scripting independiente de plataforma y orientado a objetos, preparado para realizar cualquier tipo de programa, desde aplicaciones Windows a servidores de red o incluso, páginas web. Es un lenguaje interpretado, lo que significa que no se necesita compilar el código fuente para poder ejecutarlo, lo que ofrece ventajas como la rapidez de desarrollo e inconvenientes como una menor velocidad.

El lenguaje fue creado por el holandés, Guido Van Rossum a finales de los ochenta ayudado y motivado por otro lenguaje llamado ABC, el cual fue inventado por Leo Geurts y Lambert Meertens a principios de los años 80 como lenguaje que sustituyera al BASIC. El objetivo de Guido era cubrir la necesidad de un lenguaje orientado a objetos de sencillo uso que sirviese para tratar diversas tareas dentro de la programación que habitualmente se hacía en Unix usando C.

El desarrollo de Python duró varios años, durante los que trabajó en diversas compañías de Estados Unidos. Para el año 2000 ya disponía de un producto bastante completo y un equipo de desarrollo con el que se había asociado incluso en proyectos empresariales. Actualmente trabaja en Zope, una plataforma de gestión de contenidos y servidor de aplicaciones para el web, programada completamente en Python.

Python tiene las siguientes características:

Propósito general

Se pueden crear todo tipo de programas. No es un lenguaje creado específicamente para la web, aunque entre sus posibilidades sí se encuentra el desarrollo de páginas.

Multiplataforma

Hay versiones disponibles de Python en muchos sistemas informáticos distintos. Originalmente se desarrolló para Unix, aunque cualquier sistema es compatible con el lenguaje siempre y cuando exista un intérprete programado para él. Python puede ser utilizado en diversas plataformas y sistemas operativos, entre los que podemos destacar los más populares, cómo Windows, Mac OS X y Linux. Pero, además, Python también puede funcionar en smartphones mediante un intérprete de este lenguaje para el sistema operativo Symbian.

Interpretado

Quiere decir que no se debe compilar el código antes de su ejecución. En realidad, sí que se realiza una compilación, pero esta se realiza de manera transparente para el programador. En ciertos casos, cuando se ejecuta por primera vez un código, se producen unos bytecodes que se guardan en el sistema y que sirven para acelerar la compilación implícita que realiza el intérprete cada vez que se ejecuta el mismo código.

Interactivo

Python dispone de un intérprete por línea de comandos en el que se pueden introducir sentencias. Cada sentencia se ejecuta y produce un resultado visible, que puede ayudarnos a entender mejor el lenguaje y probar los resultados de la ejecución de porciones de código rápidamente.

Orientado a Objetos

La programación orientada a objetos está soportada en Python y ofrece en muchos casos una manera sencilla de crear programas con componentes reutilizables.

Funciones y librerías

Dispone de muchas funciones incorporadas en el propio lenguaje, para el tratamiento de strings, números, archivos, etc. Además, existen muchas librerías que podemos importar en los programas para tratar temas específicos como la programación de ventanas o sistemas en red o análisis estadístico.

Sintaxis clara

Python tiene una sintaxis muy visual, gracias a una notación identada (con márgenes) de obligado cumplimiento. En muchos lenguajes, para separar porciones de código, se utilizan elementos como las llaves o las palabras clave begin y end. Para separar las porciones de código en Python se debe tabular hacia dentro, colocando un margen al código que iría dentro de una función o un bucle. Esto ayuda a que todos los programadores adopten unas mismas notaciones y que los programas de cualquier persona tengan un aspecto muy similar.

Python ofrece un listado de ventajas para sus usuarios entre las cuales se destacan:

- La cantidad de librerías que contiene, tipos de datos y funciones incorporadas en el propio lenguaje, que ayudan a realizar muchas tareas habituales sin necesidad de tener que programarlas desde cero.
- La sencillez y velocidad con la que se crean los programas. Un programa en Python puede tener de 3 a 5 líneas de código menos que su equivalente en Java o C. No requiere dedicar tiempo a su compilación debido a que es interpretado.
- La cantidad de plataformas en las que podemos desarrollar, como Unix, Windows, OS/2, Mac, Amiga y otros.
- Además, Python es gratuito, incluso para propósitos empresariales, no es necesario pagar ninguna licencia para distribuir software desarrollado con esté lenguaje. Incluso su intérprete se distribuye de forma gratuita para diferentes plataformas.

Simulación Estocástica

Un sistema o proceso estocástico es el cual su comportamiento es no determinístico. Esto se traduce en que el estado subsecuente del sistema se determina tanto por las acciones predecibles del proceso, como por un elemento aleatorio. Básicamente los sistemas de la vida real son estocásticos. Su comportamiento puede ser medido y aproximado a distribuciones y probabilidades, pero rara vez pueden ser determinados por un solo valor.

El inicio de la simulación estocástica se remonta a 1940, cuando John Von Neumann y Stanis law Marcin, trabajando en problemas matemáticos relativos a la física nuclear, cuya solución analítica era intratable y la solución de manera experimental resultaba muy costosa, acuñaron el término “Análisis de Monte Carlo”. Este concepto, llamado Simulación Estocástica en la actualidad, consiste en resolver un problema determinista simulando computacionalmente a algún proceso estocástico cuyas características probabilistas satisfacen las condiciones matemáticas del problema original. Con la ayuda de una computadora, se recrean condiciones aleatorias para analizar a una cierta dinámica o fenómeno determinista, y se analizan los resultados computacionales obtenidos. Los modelos predictivos, como una regresión lineal, requieren un conjunto de entradas conocidas para predecir un resultado o valor de destino. En muchas aplicaciones del mundo real, sin embargo, los valores de las entradas son inciertos. La simulación permite explicar la incertidumbre de las entradas en modelos predictivos y evaluar la posibilidad de varios resultados del modelo en presencia de esa incertidumbre.

Una definición formal propuesta por C. West Churchman es la siguiente:

X simula a Y si:

1. X e Y son sistemas formales.

2. Y se considera un sistema real.

3. X es una aproximación de tal sistema real.

4. La validez de X no está exenta de error.

Shubik dice: “La simulación de un sistema es la operación de un modelo (simulador), el cuál es una representación del sistema. Este modelo puede someterse a manipulaciones que serían imposibles de realizar, demasiado costosas o poco prácticas para el sistema.”

Por otra parte, es necesario garantizar que los resultados obtenidos a través de la simulación coincidan razonablemente a la dinámica estudiada. Gracias a diferentes resultados matemáticos de la teoría de probabilidad y estadística, dos disciplinas que permiten el estudio de fenómenos estocásticos.

La simulación estocástica consiste de dos etapas: el diseño del modelo representativo del fenómeno en estudio y la prueba matemática de que dicha aproximación funcione. Sin embargo, desde el punto de vista aplicado, sólo es necesario enfocarse en la primera etapa. En los últimos años ha sido posible el planteamiento de modelos más complejos y su solución a través de la simulación estocástica, usando como herramienta la computación.

Para planificar y hacer uso de un modelo de simulación se sugiere tener en cuenta el siguiente procedimiento:

- Planteamiento del problema.
- Recolección y organización de la información.
- Diseño del modelo matemático.
- Implementación computacional.
- Validación del programa computacional.
- Análisis de resultados.
- Validación de la simulación.

Es importante evitar errores en la implementación de la simulación para el análisis de un problema; a continuación, se mencionan algunos de los errores más comunes que pueden cometerse:

- El abuso en la especificidad del modelo diseñado. Esto tiene como consecuencia un costo en el tiempo de ejecución que convierte a la simulación en una herramienta poco práctica. Una solución es plantear un modelo simple para el problema, e ir aumentando paulatinamente los detalles necesarios hasta llegar a un modelo que represente adecuadamente las características esenciales del problema.
- Presentar una simulación sin calibrar, es decir, realizar una implementación sin tener la certeza de que la simulación realmente representa al problema que se está estudiando.
- Agregar supuestos que no representan al problema, aun cuando la simulación produzca los resultados esperados bajo tales supuestos. Esta práctica está alejada de un procedimiento científico y puede traer malinterpretaciones conceptuales del problema.
- Realizar iteraciones computacionales que no son suficientes para garantizar la cercanía de la solución computacional a la solución real. Siempre que sea posible, se debe estimar el error en términos del número de iteraciones y realizar las iteraciones que sean necesarias para garantizar que nuestro error es menor que una constante dada.
- Ejecutar un programa que utiliza siempre un conjunto de sucesiones particulares de números aleatorios; esto provoca que la simulación no satisface las las hipótesis estadísticas necesarias para garantizar un error pequeño cuando se tienen muchas iteraciones.
- Por último, que nuestro generador de números aleatorios no esté trabajando de manera adecuada.

Además de las ventajas intrínsecas de la simulación podemos enunciar las siguientes:

- Es posible estudiar el efecto de cambios internos y externos del sistema, haciendo alteraciones en el modelo del sistema y observar los efectos de esas alteraciones en el comportamiento del sistema.
- Puede conducir a un mejor entendimiento del sistema y por consiguiente a sugerir estrategias que mejoren la operación y eficiencia del sistema.
- La técnica de simulación puede ser utilizada para experimentar con nuevas situaciones, sobre las cuales se tiene poca o ninguna información.
- El modelo es siempre perfectible, lo que permite mejoras en el tiempo computacional requerido para su accionar.
- Es posible realizar diseño de experimentos para identificar causalidad: puede elegirse un subconjunto de las variables originales de estudio y mantenerse constante, para estudiar la influencia del resto de las variables en el fenómeno de estudio.
- Es posible reproducir un experimento aleatorio en condiciones idénticas tantas veces como sea necesario; esto se logra con el uso de números aleatorios independientes.
- Uno de los beneficios de los modelos de simulación por eventos discretos comparado a otras técnicas de modelos determinísticos –como programación lineal o PEM; así como comparados a otras técnicas de modelos estocásticos –como cadenas de Márkov o programación dinámica; reside en que el problema no requiere de relajación (una técnica utilizada para lidiar con restricciones estrictas de modelos NP-hard para que estos puedan ser resueltos en tiempo polinomial) y por tanto permite mantenerlo más ‘real’. Adicionalmente, permite una representación gráfica y visual del sistema, lo cual facilita el entendimiento y el proceso de validación. Una vez que el modelo es programado en algún software comercial (como ProModel o Arena), un proceso de verificación y validación es realizado para asegurar que el modelo se comporta como es esperado y se compara cercanamente al sistema real. Esto brinda seguridad de que el modelo representa con precisión el proceso siendo analizado.

En contraparte, es importante puntualizar algunas limitaciones que se tienen en el uso de la simulación:

- En la mayoría de las situaciones no se producen resultados exactos.
- A pesar de permitir exploración del fenómeno en muy diversas situaciones, su desempeño como herramienta de optimización suele ser muy pobre.
- Existen costos inherentes para su uso. Además del tiempo requerido para el diseño teórico del modelo, se requiere tiempo para su traducción (humana) en lenguaje computacional, y finalmente, el costo en tiempo-máquina para obtener los resultados deseados.

- Abuso de la simulación. Puede elegirse erróneamente simular en situaciones en donde obtener resultados teóricos exactos son posibles y menos costosos que simular; la decisión de simular un sistema debe ser un último recurso tras evaluar las distintas maneras de resolver el problema de estudio o bien como un complemento exploratorio en el estudio del mismo. Es importante señalar que, al reportar los resultados obtenidos por simulación, se debe ser muy específico en el modelo utilizado y las condiciones computacionales específicas. Con ello el lector tendrá una idea de que tan verosímil es que la realidad coincida con los resultados obtenidos y también la posibilidad de replicar la simulación.

Simulación de números aleatorios

La simulación estocástica está basada en la generación de números aleatorios que provienen de diferentes leyes de probabilidad. Sin embargo, como veremos más adelante, basta generar números aleatorios uniformemente distribuidos en el intervalo (0,1) para incorporar la aleatoriedad en esquemas de simulación estocástica para distribuciones arbitrarias. En este sentido, las variables uniformes en (0,1) conforman la base para simular sistemas estocásticos generales.

En el inicio de la simulación, la aleatoriedad se generaba por medio de técnicas manuales, tales como volados, dados, cartas barajadas, ruletas y urnas con bolas. En aquel entonces se creía que únicamente a través de artefactos mecánicos y/o electrónicos se podía producir verdaderas sucesiones de números aleatorios. Estas sucesiones eran preservadas en tablas de números aleatorios que podían ser consultadas cada vez que se querían utilizar tales números.

Actualmente la gran mayoría de los generadores de números aleatorios no dependen de dispositivos físicos, sino que están basados en simples algoritmos que pueden ser fácilmente implementados en un ordenador, lo cual ha disminuido considerablemente el costo en la generación de números aleatorios. La discusión, en parte filosófica, de si realmente existe o no el azar ha motivado el desarrollo de la generación de números aleatorios basados en sistemas físicos microscópicos reales. Por ejemplo, utilizando el ruido atmosférico o bien la división de un haz de luz. Tales números pueden ser utilizados también en la simulación estocástica, siempre que satisfagan las hipótesis estadísticas necesarias.

Un buen generador de números aleatorios presenta todas las características estadísticas importantes de las verdaderas sucesiones de números aleatorios, aún si tales sucesiones son generadas mediante un algoritmo determinista.

Las cantidades aleatorias generadas mediante algoritmos deterministas se conocen con el nombre de pseudoaleatorias. Los métodos más comunes para generar sucesiones de números aleatorios utilizan los llamados generadores lineales congruencia les que han sido propuestos como versiones del generador propuesto por Lehmer en 1949 que genera una sucesión determinística de números por medio de la fórmula recursiva

$$X_{i+1} = (aX_i + c) \pmod{m}$$

donde el valor inicial es X0, también conocido como semilla, y a, c y m, todos enteros positivos, son el multiplicador, el incremento y el módulo respectivamente.

Para el caso especial donde c = 0 la fórmula anterior se reduce a

$$X_{i+1} = aX_i \pmod{m};$$

y a tal generador se le conoce como generador congruencial multiplicativo.

Note que cada Xi puede asumir valores únicamente en el conjunto {0, 1, . . . , m − 1}, y las cantidades

$$U_i = \frac{X_i}{m}$$

llamadas números pseudoaleatorios (debido a que son generadas por un método determinista) son aproximaciones a una verdadera sucesión de variables aleatorias uniformes.

Sea a un natural. Un generador tiene periodo m si y solo si se cumple que

1. El máximo común divisor entre c y m es 1.

2. a ≡ 1 (mod p) para todo primo p factor de m.

3. a ≡ 1 (mod 4) en el caso en que m es un múltiplo de 4.

Para las implementaciones en computadora, se elige m como un número primo suficientemente grande que pueda ser soportado por la longitud de una palabra en la computadora. Por ejemplo, en una computadora con longitud de palabra de 32-bits, se obtienen generadores estadísticamente aceptables eligiendo m = 231 − 1 y a = 75 , suponiendo que el primer bit es usado para el signo. Un ordenador con longitud de palabra de 64-bits o 128-bits naturalmente producirá mejores resultados estadísticos. Una elección típica de valores para los parámetros multiplicativos también son m = 231 − 1, a = 75 y c = 0. A tal elección se le conoce como el “generador estándar mínimo”, que satisface propiedades deseables. Siguiendo la misma idea, se ha propuesto alternativamente encontrar una sucesión de números pseudoaleatorios realizando operaciones similares al generador congruencial antes visto pero suponiendo que es posible tomar la semilla libremente. Este supuesto permite realizar iteración directa y el uso de números aleatorios en sistemas más sencillos como las hojas de cálculo (y planillas electrónicas en general).

El Generador Congruencial Semilla-Controlado Simple (Lewis-Goodman-Miller 1969) empieza con un valor Z llamado Zin que debe estar en el intervalo (0, 1) y representa el número pseudoaleatorio inicial de la sucesión. De esta manera se produce iterativamente la sucesión:

$$U_{i+1} = \text{red}(m * a * U_i , 0) \pmod{m}/m$$

donde m y a son enteros positivos y red(x, y) es la función redondear x a y decimales. Aquí se refleja la utilidad que representa ingresar una fórmula de este estilo en una hoja de cálculo al designar Ui como el valor de una celda en una cierta posición. La función

Gen.Cong.SemCont() aplica el método anterior. Distintas versiones se han propuesto en virtud de mejorar los resultados estadísticos obtenidos cuando se analiza el comportamiento de los números obtenidos, pensados como observaciones de variables aleatorias independientes y con distribución uniforme en el intervalo (0,1). En 1982, Wichmann y Hill propusieron un modelo más particular que presenta un sistema que utiliza más de una semilla para un número pseudoaleatorio. En 1988, Park y Miller propusieron una versión del Generador Congruencial Lineal en el que m es una potencia de un número primo, a ser directamente un entero de módulo m, c = 0 y la semilla un primo relativo de m. De ahí surgen los parámetros típicos antes vistos, que están pensados en máquinas de 32-bits de forma estándar.

Simulación de Variables aleatorias

Método de la transformada Inversa

Supongamos que queremos simular a una variable aleatoria X, con función de distribución F. Utilizando la definición de la función de distribución inversa, tenemos el siguiente método para generar números provenientes de la distribución F:

1. Generar un número aleatorio U ∼ U(0, 1).

2. Tomar X = F −(U).

En las próximas secciones haremos una especialización a los casos en que la variable aleatoria es continua, es decir cuando su función de distribución es continua; o discreta, donde ocurre que la variable tiene soporte finito o numerable.

Variables Aleatorias Continuas.

En este caso podemos utilizar el algoritmo de manera muy simple, debido a que en este caso la inversa generalizada se reduce a la inversa de la función de distribución F.

Variables Aleatorias Discretas.

Nos enfocaremos ahora en variables aleatorias que toman un conjunto a lo más numerable de valores posibles.

Sea X una variable aleatoria discreta con función de densidad P(X = xi) = pi , para i = 1, 2, . . ., donde ocurre que Σpi = 1 y x1 < x2 < . . .. Si definimos X˜ = F −(U), donde F es la distribución de X y U una variable aleatoria uniforme en (0, 1), ocurre que

$$P(\tilde{X} = x_i) = P\left(\sum_{k=1}^{i-1} p_k \leq U < \sum_{k=1}^{i} p_k\right) = p_i,$$

ya que para 0 < a < b < 1 ocurre que P(a ≤ U < b) = b − a. Utilizando esta observación, el algoritmo de la inversa generalizada se reduce en este caso a lo siguiente:

1. Generar U ∼ U(0, 1).

2. Tomar

$$X = \begin{cases} x_1 & si\ U < p_1 \\ x_2 & si\ p_1 \leq U < p_1 + p_2 \\ \vdots & \\ x_i & si\ \sum_{k=1}^{i-1} p_k \leq U < \sum_{k=1}^{i} p_k \\ \vdots & \end{cases}$$

Variables Discretas con probabilidades iterativas

Un característica interesante de las siguientes distribuciones es la posibilidad de representar y calcular la probabilidad de tomar el valor i, denotado por pi , en terminos de pi−1. Este resultado es utilizado para calcular las fórmulas de De Pril y Panger en Teoría del Riesgo. Puede demostrarse que las únicas distribuciones que cumplen tal propiedad son la Binomial, Poisson y Binomial Negativa.

Distribuciones discretas con esperanza alta.

Respecto a la eficiencia del algoritmo de Transformada Inversa en el caso de variables aleatorias discretas tenemos las siguientes observaciones:

1. En el caso en que X = xi , fue necesario para la computadora realizar i comparaciones. Entonces, el número esperado de iteraciones del algoritmo es

$$\sum_{i=1}^{\infty} i p_i,$$

lo que puede ser muy ineficiente si la cardinalidad del soporte es muy grande o infinito.

2. El algoritmo comienza a hacer tales comparaciones empezando siempre por el menor valor que toma la variable aleatoria, sigue con el segundo valor más pequeño, etc. Esto puede ser muy ineficiente para algunas distribuciones particulares.

En el caso en que conocemos algunas de las características de la función distribución (esperanza, simetría, sesgo, etc.), es natural pensar que podemos utilizar esa información para modificar el algoritmo que utiliza a la inversa generalizada de una distribución y obtener un algoritmo más eficiente. En esta sección nos focalizaremos en el caso de variables aleatorias discretas con valor esperado muy alto.

La idea general de nuestra modificación es empezar a hacer comparaciones a partir del valor de la media μ := E[X] y decidir si continuar hacia valores menores o bien continuar hacia valores mayores que μ, dependiendo del resultado de la primera comparación. Más específicamente: nuestra primera comparación será entre la probabilidad acumulada hasta el valor μ y una variable aleatoria uniforme en (0, 1). Debido a que el valor esperado podría ser no entero, utilizamos el mayor de los enteros que están antes del valor μ, denotado por [μ].

Transformada inversa caso discreto con esperanza alta

1. Generar U ∼ U(0, 1).

2. Si U < F([μ]), tomar

$$X = \begin{cases} x_1 & si\ U < p_1 \\ x_2 & si\ p_1 \leq U < p_1 + p_2 \\ \vdots & \\ x_{\lfloor \mu \rfloor} & si\ \sum_{k=1}^{\lfloor \mu \rfloor - 1} p_k \leq U < \sum_{k=1}^{\lfloor \mu \rfloor} p_k \end{cases}$$

en otro caso, tomar

$$X = \begin{cases} x_{\lfloor \mu \rfloor + 1} & si\ \sum_{k=1}^{\lfloor \mu \rfloor} p_k \leq U < \sum_{k=1}^{\lfloor \mu \rfloor + 1} p_k \\ x_{\lfloor \mu \rfloor + 2} & si\ \sum_{k=1}^{\lfloor \mu \rfloor + 1} p_k \leq U < \sum_{k=1}^{\lfloor \mu \rfloor + 2} p_k \\ \vdots & \end{cases}$$

Método de aceptación y rechazo.

El método de simulación descrito en esta sección sirve para obtener muestras de variables aleatorias de cualquier distribución. Fue propuesto por Stan Ulam y John von Neumann, aunque existe el antecedente de Buffon y su famoso problema de la aguja. Dicho método consiste en muestrear variables aleatorias de una distribución apropiada y someterlas a una prueba para ser o no aceptadas como muestra proveniente de otra distribución.

Supongamos que tenemos una variable aleatoria con función de densidad f que toma valores en el intervalo [a, b] y vale cero en cualquier otro punto. Si definimos

$$c := \sup \{f(x) : x \in [a, b]\} ,$$

entonces, para generar una variable aleatoria Z ∼ f, podemos usar los siguientes pasos de aceptación y rechazo:

1. Generamos X ∼ U(a, b).

2. Generamos Y ∼ U(0, c) independientemente de X.

3. Si Y ≤ f(X), hacemos Z = X. En otro caso, regresamos al paso 1.

En este algoritmo, como el vector (X, Y) está uniformemente distribuido sobre [a, b]×[0, c], ocurre que una pareja aceptada (X, Y), está uniformemente distribuida bajo la curva de f. Y eso implica lo siguiente: si nos fijamos en tiras de tamaño ε en el eje x delimitadas por arriba por la curva f, entonces la probabilidad de que (X, Y) caiga dentro de la tira que contiene a un punto x0 en su base es aproximadamente ε f(x0).

La idea anterior funciona en un contexto más general. Supongamos que f es una densidad cuyo soporte está contenido en el soporte de otra densidad g (que es fácil de muestrear) y que existe c tal que f(x) ≤ cg(x), para cualquier x en el soporte de f. Entonces el siguiente algoritmo nos proporciona un número aleatorio proveniente de la densidad f.

Algoritmo Aceptación-Rechazo

1. Generar $X \sim g$.

2. Generar $Y \sim U(0, 1)$.

3. Aceptamos, $Z = X$, si $Y \leq f(X)\, cg(X)$. En otro caso, regresar al paso 1.

Antes de justificar el hecho de que el algoritmo realmente genera una variable aleatoria de la densidad deseada, mencionaremos algunas virtudes del mismo:

1. No es necesario tener definida explícitamente a la función de densidad.

2. Debido a que el método no utiliza directamente una expresión analítica de la densidad, se puede utilizar cuando el método de la función inversa no es posible o es computacionalmente ineficiente.

3. En estadística Bayesiana, es común conocer a las densidades de probabilidad sin conocer las constantes de normalización asociadas. En estos casos este método permite conocer tales constantes.

Simulación de un proceso Poisson

Simulación de un proceso de Poisson homogéneo

Suponga que se quieren generar los primeros n eventos de un proceso Poisson de intensidad λ. Para ello nos apoyaremos del resultado de que el tiempo transcurrido entre dos eventos sucesivos cualesquiera son una sucesión de variables aleatorias independientes exponenciales de parámetro λ. De manera que para generar un proceso Poisson basta con generar esos tiempos de interarribo.

Si generamos n números aleatorios U1, U2, . . . , Un y hacemos $Xi = -1\ \lambda \log Ui$, entonces las Xi pueden considerarse como los tiempos entre el (i−1)-´esimo y el i-´esimo evento de un proceso Poisson. Supongamos que queremos generar las una trayectoria de un proceso de Poisson en el intervalo finito [0, T] , podemos seguir el procedimiento.

Generación de trayectorias de un proceso de Poisson

En este algoritmo t se refiere al tiempo, I es el número de eventos que han ocurrido al tiempo t y S(I) es el tiempo acumulado hasta el último evento:

1. Hacer $t = 0$, $I = 0$

2. Generar $U \sim U(0, 1)$

3. Hacer $t = t - 1\ \lambda \log U$. Si $t > T$, detenerse

4. En caso contrario, hacer $I = I + 1$ y $S(I) = t$

5. Ir al paso 2.

Simulación de Proceso de Poisson no homogéneo

El proceso de conteo extremadamente importante para propósitos de modelación es el proceso Poisson no homogéneo, el cual relaja la suposición de incrementos estacionarios del proceso Poisson. Esto abre la posibilidad a que la tasa de arribo no necesariamente sea constante, sino que pueda variar en el tiempo, en esta sección se presenta un algoritmo para simular trayectorias de este proceso.

Suponga que deseamos simular una trayectoria de un proceso de Poisson no homogéneo en un intervalo de tiempo [0, T] con función de intensidad λ(t). El método que revisaremos a continuación se conoce con el nombre de muestreo aleatorio.

Este comienza eligiendo un valor λ el cual es tal que λ(t) ≤ λ para toda t ≤ T.

Tal proceso Poisson no homogéneo puede generarse seleccionando aleatoriamente el evento tiempos de un proceso Poisson con tasa λ. Es decir, si un evento de un proceso Poisson con tasa λ que ocurre al tiempo t es contabilizado con probabilidad λ(t)/λ, entonces el proceso de los eventos contabilizados es un proceso Poisson no homogéneo con función de intensidad λ(t) para 0 ≤ t ≤ T.

Por lo tanto, simulando un proceso Poisson y contabilizando aleatoriamente sus eventos, podemos generar un proceso Poisson no homogéneo. De esta forma podemos describir el algoritmo deseado como sigue:

1. Hacer t = 0, I = 0

2. Generar un número aleatorio U ∼ U(0, 1)

3. Hacer t = t − 1 λ log U. Si t > T, detenerse

4. En caso contrario, generar otro número aleatorio U ∼ U(0, 1)

5. Si U ≤ λ(t)/λ, hacer I = I + 1 y S(I) = t

6. Regresar al paso 2.

Simulación de Proceso de Poisson compuesto

En esta sección formalizamos de manera algorítmica la simulación de un proceso de Poisson compuesto X = {Xt}t ≥0 donde

$$X_t = \sum_{i=1}^{N_t} Y_i,$$

con N = {Nt}t≥0 un proceso de Poisson y {Yi}i≥1 una sucesión de variables aleatorias independientes e identicamente distribuidas con distribución F. El siguiente algoritmo describe la manera de simular a X en el intervalo de tiempo [0, T].

1. Hacer t = 0, I = 0.

2. Generar U ∼ U(0, 1)

3. Hacer t = t − 1 λ log U. Si t > T, detenerse

4. En caso contrario, generar hacer I = I + 1 ,y S(I) = t

5. Generar YI ∼ F.

6. Hacer XT = PI i=1 Yi

7. Ir al paso 2.

NUMPY

Uno de los módulos más importantes de Python es Numpy. El origen de Numpy se debe principalmente al diseñador de software Jim Hugunin quien diseñó el módulo Numeric para dotar a Python de capacidades de cálculo similares a las de otros softwares como MATLAB. Posteriormente, mejoró Numeric incorporando nuevas funcionalidades naciendo lo que hoy conocemos como Numpy.

Numpy es el encargado de añadir toda la capacidad matemática y vectorial a Python haciendo posible operar con cualquier dato numérico o array (posteriormente veremos qué es un array). Incorpora operaciones tan básicas como la suma o la multiplicación u otras mucho más complejas como la transformada de Fourier o el álgebra lineal. Además incorpora herramientas que nos permiten incorporar código fuente de otros lenguajes de programación como C/C++ o Fortran lo que incrementa notablemente su compatibilidad e implementación.

Arrays

Con toda probabilidad, el lector que haya realizado un acercamiento a cualquier lenguaje de programación habrá oído hablar de arrays. Un array es el termino que traslada el concepto matemático de vector o matriz a la programación añadiéndole la noción de almacenamiento en memoria. Los arrays disponen en su interior de una serie de elementos dispuestos en filas y/o columnas dependiendo de la dimensión.

El desarrollo y la principal finalidad del módulo Numpy es la creación y modificación de arrays multidimensionales. Para este fin utilizaremos las clase ndarray del ingles N-dimensional array o usando su alias simplemente array (no confundir con la clase array.array que ofrece menos funcionalidad). En Python cada clase puede tener atributos que se pueden llamar con el método visto anteriormente o simplemente escribiendo a continuación de la clase un punto y el atributo. En la mayoría de los IDEs al cargar la clase y escribir el punto aparecen todos los atributos disponibles en orden alfabético por lo que en caso de dudar

siempre podemos utilizar este método para escribir el comando. En el caso de ndarray los principales atributos son los siguientes:

ndarray.ndim –> Proporciona el número de dimensiones de nuestro array. El array identidad es un array cuadrado con una diagonal principal unitaria.

ndarray.shape –> Devuelve la dimensión del array, es decir, una tupla de enteros indicando el tamaño del array en cada dimensión. Para una matriz de n filas y m columnas obtendremos (n,m).

ndarray.size –> Es el número total de elementos del array.

ndarray.dtype –> Es un objeto que describe el tipo de elementos del array.

ndarray.itemsize –> devuelve el tamaño del array en bytes.

ndarray.data –> El buffer contiene los elementos actuales del array.

Los arrays son unos de los elementos más utilizados por los programadores bajo cualquier lenguaje. Este gran uso hace que dispongamos de una gran variedad de comandos para la creación de arrays: arrays unidimensionales, multidimensionales, nulos, unitarios, secuenciales, de números aleatorios, etc. Dependiendo de las necesidades de nuestro problema escogeremos la opción más adecuada.

identity(n,dtype). Devuelve la matriz identidad, es decir, uma matriz cuadrada nula excepto en su diagonal principal que es unitaria. n es el número de filas (y columnas) que tendrá la matriz y dtype es el tipo de dato. Este argumento es opcional. Si no se establece, se toma por defecto como flotante.

ones(shape,dtype). Crea un array de unos compuesto de shape elementos.

zeros(shape, dtype). Crea un array de ceros compuesto de “shape” elementos”.

empty(shape, dtype). Crea un array de ceros compuesto de “shape” elementos” sin entradas.

eye(N, M, k, dtype). Crea un array bidimensional con unos en la diagonal k y ceros en el resto. Es similar a identity. Todos los argumentos son opcionales. N es el número de filas, M el de columnas y k es el índice de la diagonal. Cuando k=0 nos referimos a la diagonal principal y por tanto eye es similar a identity.

arange([start,]stop[,step,],dtype=None). Crea un array con valores distanciados step entre el valor inicial star y el valor final stop. Si no se establece step python establecerá uno por defecto.

linspace(start,stop,num,endpoint=True,retstep=False). Crea un array con valor inicial start, valor final stop y num elementos.

meshgrid(x,y). Genera una matriz de coordenadas a partir de dos los arrays x, y.

SCIPY

Scipy es una librería de herramientas numéricas para Python que se distribuye libremente. El módulo scipy confiere al lenguaje general Python capacidades de cálculo numérico de gran capacidad. Scipy además de poseer en su núcleo a Numpy, posee módulos para optimización de funciones, integración, funciones especiales, resolución de ecuaciones diferenciales ordinarias y otros muchos aspectos.

Su organización se estructura en subpaquetes, que se pueden considerar especializados en dominios científicos determinados. Podemos encontrar estos paquetes, según la ayuda de scipy:

- linalg – Algebra lineal
- signal – Procesamiento de señales
- stats – Funciones estadísticas
- special – Funciones especiales
- integrate – Integración
- interpolate – Herramientas de interpolación
- optimize – Herramientas de optimización
- fftpack – Algortimos de transformada de Fourier
- io – Entrada y salida de datos
- lib.lapack – Wrappers a la librería LAPACK
- lib.blas – Wrappers a la librería BLAS
- lib – Wrappers a librerías externas
- sparse – Matrices sparse
- misc – otras utilidades
- cluster – Vector Quantization / Kmeans
- maxentropy – Ajuste a modelos con máxima entropía

En scipy las funciones universales (suma, resta, division, etc.) se han alterado para no producir errores de coma flotante cuando se encuentran errores; por ejemplo se devuelve

NaN e Inf en los arrays en lugar de errores. Para ayudar a la detección de estos eventos, hay disponibles varias funciones como sp.isnan, sp.isinfinite, sp.isinf. Además se han modificado algunas funciones (log, sqrt, funciones trigonometricas inversas) para devolver valores complejos en lugar de NaN (por ejemplo sp.sqrt(-1) devuelve 1j).

FITTER

El paquete Fitter proporciona una clase simple para identificar la distribución desde la cual se generan las muestras de datos. Utiliza 80 distribuciones de Scipy y le permite trazar los resultados para verificar cuál es la distribución más probable y los mejores parámetros.

Scipy tiene 80 distribuciones y la clase Fitter las escaneará todas, llamará a la función de ajuste, ignorará aquellas que fallan o se ejecutarán para siempre y finalmente dará un resumen de las mejores distribuciones en el sentido de suma de los errores cuadrados.

PANDAS

Pandas es un paquete de Python que proporciona estructuras de datos similares a los dataframes de R. Pandas depende de Numpy, la librería que añade un potente tipo matricial a Python. Los principales tipos de datos que pueden representarse con pandas son:

- Datos tabulares con columnas de tipo heterogéneo con etiquetas en columnas y filas.
- Series temporales.

Pandas proporciona herramientas que permiten:

- Leer y escribir datos en diferentes formatos: CSV, Microsoft Excel, bases SQL y formato HDF5.
- Seleccionar y filtrar de manera sencilla tablas de datos en función de posición, valor o etiquetas.
- Fusionar y unir datos
- Transformar datos aplicando funciones tanto en global como por ventanas
- Manipulación de series temporales
- Hacer gráficas

En pandas existen tres tipos básicos de objetos todos ellos basados a su vez en Numpy:

- Series (listas, 1D),
- DataFrame (tablas, 2D) y
- Panels (tablas 3D).

Series: Son arrays unidimensionales con indexación (arrays con índice o etiquetados), similar a los diccionarios. Las Series se pueden crear tanto a partir de listas como de diccionarios. De manera opcional podemos especificar una lista con las etiquetas de las filas.

DataFrame: Son estructuras de datos similares a las tablas de bases de datos relacionales como SQL. Los DataFrame se pueden crear de diferentes maneras, una forma común de crearlos es partir de listas o diccionarios de listas, de diccionarios o de Series. En los DataFrame tenemos la opción de especificar tanto el index (el nombre de las filas) como columns (el nombre de las columnas).

Para seleccionar datos usamos los métodos loc, iloc e ix. loc permite seleccionar dato usando las etiquetas de filas y columnas, iloc basándose en posición e ix basándose tanto en etiquetas como posición. En el caso de una Serie, devuelve un único valor y en el caso de los DataFrame puede devolver tanto una Serie si sólo se indica la posición de fila, o un valor único si se indican fila y columna. *conca*t permite concatenar Series y DataFrame. Mediante la opción axis, podemos controlar si la unión se debe hacer por filas o por columnas. *join* es un método para combinar DataFrame que puedan tener diferentes etiquetas de filas en un único DataFrame. *groupby* permite agrupar los datos en función de un criterio dado. Devuelve un objeto GroupBy, y uno de sus atributos (groups) es un diccionario dónde las claves son los grupos y los valores son las etiquetas de las filas que pertenecen a dicho grupo. *map* y *applymap* sirven para aplicar una función a cada uno de los elementos de una Serie y un DataFrame respectivamente. i*teritems* y i*terrows* permiten iterar sobre un DataFrame, el primero devuelve el DataFrame columna por columna y el segundo por filas.

Pandas tiene además implementados diversos métodos estadísticos para operar en Series y DataFrames. En el caso de las Series se aplican directamente, mientras que en DataFrame hay que especificar si se tiene que aplicar por filas (axis=1) o por columnas (axis=0, valor por defecto). Además, existen métodos para el cálculo de estadísticos mediante el uso de ventanas deslizantes a lo largo de series de datos. Algunos ejemplos de estos métodos son:

- count Número de observaciones no nulas
- sum Suma de valores
- mean Media
- median Mediana
- min Mínimo
- max Máximo
- mode Moda
- abs Valor absoluto
- std Desviacion estándar
- var Varianza

- quantile Quartil
- corr Cálculo de correlaciones
- cov Cálculo de covarianza

Los objetos de pandas permiten tanto leer datos en diversos formatos (read_csv, read_excel, read_json, read_html, read_pickle....) como escribir en ellos (to_csv, to_excel, to_json, to_html, to_pickle....). Permite incluso leer y escribir en el portapapeles (read_clipboard, to_clipboard). Veamos un par de ejemplos sencillo de cómo leer y escribir un archivo CSV (comma separated value).

Alguno de los argumentos, además del nombre del archivo, que se le pueden pasar a la función read_csv son:

- sep: el delimitador que divide los campos del csv
- header: el número de fila que contiene los nombres de las columnas. Por defecto es la primera línea (linea 0), si no existe cabecera header=None
- index_col: el número de columna que contiene los nombre para usar como indexa. Por defecto no considera ninguna

Distribución Cauchy Plegada

La función densidad de probabilidad de la distribución Cauchy Plegada se puede expresar en términos de la distribución estándar de Cauchy como:

$$f(x, \mu, \sigma) = \frac{1}{\sigma}\left(\text{CAUPDF}\left(\frac{x-\mu}{\sigma}\right) + \text{CAUPDF}\left(\frac{x+\mu}{\sigma}\right)\right) \qquad x \geq 0$$

donde CAUPDF es la función de densidad de probabilidad de una distribución estándar de Cauchy y u y s son los parámetros de ubicación y escala de la distribución primaria de Cauchy. Estos parámetros son parámetros de forma para la distribución plegada de Cauchy. Si u es cero, la distribución plegada de Cauchy se reduce a la mitad de la distribución de Cauchy.

La fórmula para la función de distribución acumulada de la distribución doblada de Cauchy se puede expresar en términos de la distribución acumulada de la distribución estándar de Cauchy de la siguiente manera:

$$F(x, \mu, \sigma) = \text{CAUCDF}\left(\frac{x-\mu}{\sigma}\right) - \text{CAUCDF}\left(\frac{-x-\mu}{\sigma}\right) \qquad x \geq 0$$

donde CAUCDF es la función de distribución acumulativa estándar de Cauchy.

Distribución Mielke

La forma general de la distribución beta-kappa de Mielke tiene la siguiente función de densidad de probabilidad:

$$f(x; k, \theta, u, \beta) = \frac{\frac{k}{\beta}\left(\frac{x-u}{\beta}\right)^{k-1}}{\left(1+\left(\frac{x-u}{\beta}\right)^{\theta}\right)^{1+k/\theta}} \qquad x > u; k, \theta, \beta > 0$$

con k y theta que denotan parámetros de forma y u y beta que denotan la ubicación y los parámetros de escala, respectivamente.

La forma estándar de la distribución es:

$$f(x; k, \theta) = \frac{kx^{k-1}}{\left(1+x^{\theta}\right)^{1+k/\theta}} \qquad x > 0; k, \theta > 0$$

La distribución beta-kappa de Mielke es un caso especial de una distribución F generalizada parametrizada de la forma a * (F (v_1, v_2)^b). Los detalles de la reparametrización se dan en Johnson, Kotz y Balakrishnan. Esta referencia también discute varias formas de distribuciones F generalizadas.

Esta distribución también está estrechamente relacionada con la distribución Kappa

Distribución Triangular

El nombre de esta distribución viene dado por la forma de su función de densidad. Este modelo proporciona una primera aproximación cuando hay poca información disponible, de forma que sólo se necesita conocer el mínimo (valor pesimista), el máximo (valor optimista) y la moda (valor más probable). Estos tres valores son los parámetros que caracterizan a la distribución triangular y se denotan por a, b y c, respectivamente.

Su distribución densidad de probabilidad es:

$$f(x|a,b,c) = \begin{cases} \frac{2(x-a)}{(b-a)(c-a)} & \text{para } a \leq x < c, \\ \frac{2}{b-a} & \text{para } x = c, \\ \frac{2(b-x)}{(b-a)(b-c)} & \text{para } c < x \leq b, \\ 0 & \text{para otros casos} \end{cases}$$

Campo de variación:

$a \leq x \leq b$

Parámetros:

a: mínimo, $-\infty < a < \infty$

c: moda, $-\infty < c < \infty$ con $a \leq c \leq b$

b: máximo, $-\infty < b < \infty$ con $a < b$

Un ejemplo del uso de esta distribución se encuentra en el análisis del riesgo, donde la distribución más apropiada es la beta pero dada su complejidad, tanto en la su comprensión

como en la estimación de sus parámetros, se utiliza la distribución triangular como proxy para la beta.

Distribución F No central

De forma similar a la distribución X^2 no central, la distribución F no central es una suma ponderada de las funciones beta incompletas utilizando las probabilidades de Poisson como ponderaciones.

$$F(x|\nu_1,\nu_2,\delta) = \sum_{j=0}^{\infty}\left(\frac{\left(\frac{1}{2}\delta\right)^j}{j!}e^{\frac{-\delta}{2}}\right) I\left(\frac{\nu_1\cdot x}{\nu_2+\nu_1\cdot x}\middle|\frac{\nu_1}{2}+j,\frac{\nu_2}{2}\right)$$

I (x | a, b) es la función beta incompleta con los parámetros a y b, y δ es el parámetro de no centralidad.
La distribución F es el resultado de tomar la razón de χ2 variables aleatorias, cada una dividida por sus grados de libertad.

Si el numerador de la relación es una variable aleatoria chi-cuadrado no central dividida por sus grados de libertad, la distribución resultante es la distribución F no central.

La aplicación principal de la distribución F no central es calcular el poder de una prueba de hipótesis relativa a una alternativa particular.

Sistema de familias de distribuciones de Johnson

Cuando el supuesto de normalidad sobre la característica de calidad bajo estudio no se cumple, se presentan problemas para la aplicación de algunas técnicas en Control Estadístico de Procesos (Statistical Process Control), SPC. Sin embargo, cuando esto sucede es posible transformar los datos no-normales a datos normales, a través de técnicas como el Sistema de Familias de Distribuciones de Johnson. Johnson en el año de 1949 define tres familias de distribuciones para una variable aleatoria X continua a saber:

- SB: Se refiere a X acotada.
- SL: Se refiere a X acotada por debajo o lognormal.
- SU : Se refiere a X no-acotada.

Para ajustar un conjunto de datos no-normales, es necesario establecer criterios que permitan determinar la pertenencia del mismo a una de las tres familias. Cada una de ellas tiene asociada una transformación de X a una variable normal estándar Z, así como condiciones especiales para los parámetros estimados y el rango de la variable X, que deben tenerse en cuenta cuando se va a escoger la familia con la que se quiere trabajar. A continuación se definen las transformaciones y condiciones para la familia SB de Johnson, tomadas de Chou, Polansky & Mason (1998).

Transformaciones y condiciones para la familia SB de Johnson
Cuando los datos pertenecen a la familia SB la transformación aplicada es:

$$Z = \gamma + \eta \ln\left(\frac{X-\epsilon}{\lambda+\epsilon-X}\right)$$

Sujeta a:
Condiciones de los parámetros: $\eta, \lambda > 0$, $-\infty < \gamma < \infty$, $-\infty < \varepsilon < \infty$.
Condiciones de la variable X: $\varepsilon < X < \varepsilon + \lambda$.

Datos

	N_Comensales	N_H	N_M	Llegada	Pedido	Orden	Salida
0	5	2	3	13:43:14	13:46:54	13:52:04	14:43:16
1	2	1	1	13:43:28	13:46:03	13:49:17	14:42:00
2	3	0	3	13:47:07	13:50:27	13:56:13	14:42:39
3	2	1	1	13:47:56	13:49:51	14:01:11	14:27:25
4	1	0	1	13:54:17	13:55:56	14:02:19	14:38:55
5	3	2	1	13:57:02	14:00:28	14:08:16	14:46:18
6	5	1	4	14:01:53	14:09:12	14:14:25	15:05:26
7	2	1	1	14:06:41	14:09:30	14:15:34	14:40:53
8	3	2	1	14:09:59	14:14:30	14:20:39	14:47:12
9	2	0	2	14:25:06	14:37:48	14:39:10	14:47:19
10	2	2	0	14:28:37	14:37:56	14:44:20	15:23:06
11	2	1	1	14:43:09	14:44:20	14:43:59	15:16:59
12	2	2	0	14:44:33	14:50:13	14:53:16	15:09:56
13	4	3	1	14:55:23	14:58:50	15:09:09	15:41:37
14	1	1	0	15:00:45	15:05:05	15:09:31	15:37:39
15	2	1	1	15:01:40	15:05:08	15:09:44	15:37:46
16	3	2	1	15:10:20	15:15:41	15:27:16	16:03:05
17	1	0	1	15:15:49	15:16:20	15:27:35	16:02:32
18	2	1	1	15:16:50	15:27:54	15:38:24	16:16:51
19	1	1	0	15:28:10	15:32:16	15:38:26	16:19:02
20	2	2	0	11:44:01	11:44:10	11:48:40	14:02:32
21	2	1	1	11:56:12	12:01:11	12:04:36	12:33:08
22	4	1	3	12:05:43	12:09:26	12:16:33	12:50:24
23	1	1	0	12:10:59	12:15:01	12:20:07	12:46:16
24	2	1	2	12:18:09	12:26:05	12:31:01	13:03:37
25	2	1	1	12:22:39	12:25:50	12:31:14	13:03:40
26	2	1	1	12:32:56	12:39:25	12:41:11	13:31:06
27	2	2	0	12:57:11	13:16:46	13:20:44	13:24:50
28	2	0	2	12:57:17	13:02:59	13:03:11	13:38:41

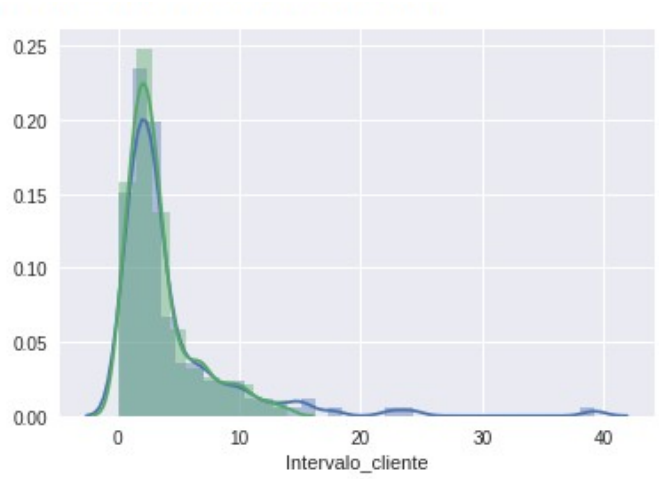
Desarrollo

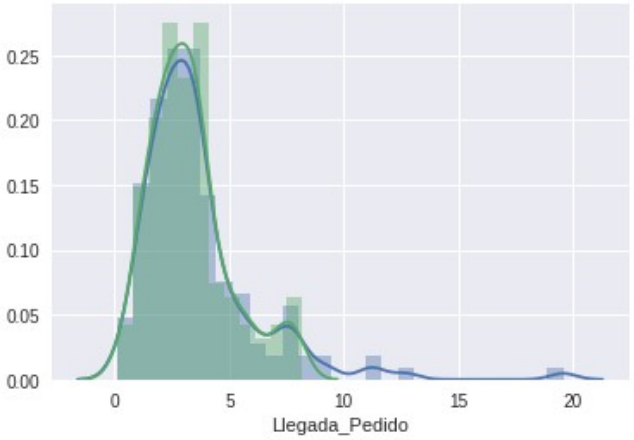
La recolección de datos la hicimos observando a los clientes en el restaurante “El Asadero” de 11:00 am a las 4:00 pm, los datos mostrados anteriormente se fueron registrando y almacenando de manera automatica gracias a un programa hecho en Python, razón por la cual las horas entán registradas de manera exacta.

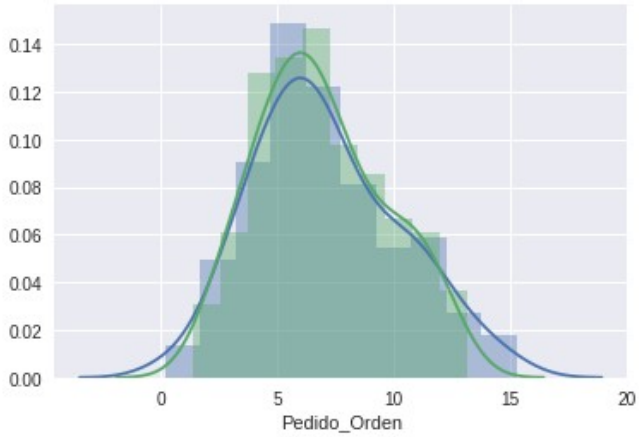
En primer lugar decidimos hacer las diferencias de tiempo entre la llegada de un grupo de comensales y otro, el tiempo que tardaba el mesero en tomar el pedido del grupo de comensales, el tiempo transcurrido hasta que los platillos eran entregados al comensal y el tiempo que pasaba hasta que los comensales dejaban desocupada una mesa.

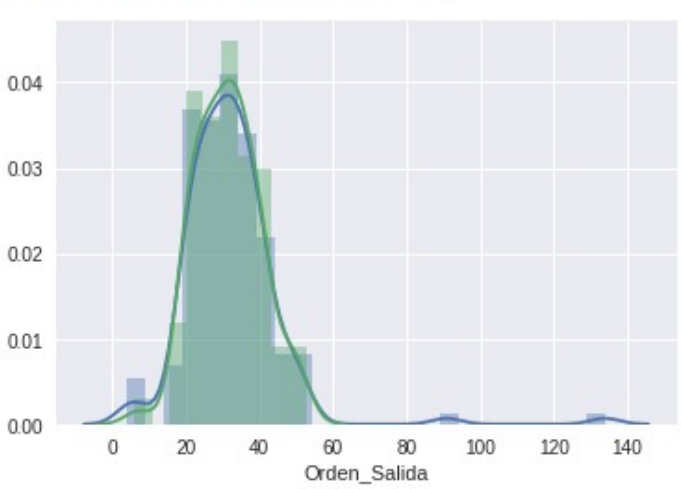
La recolección de datos se hizo en tres días diferentes, en horarios donde habitualmente las personas toman su comida. Entonces el siguiente punto de nuestro experimento fue juntar las tablas de cada día en una sola tabla con la que posteriormente trabajariamos.

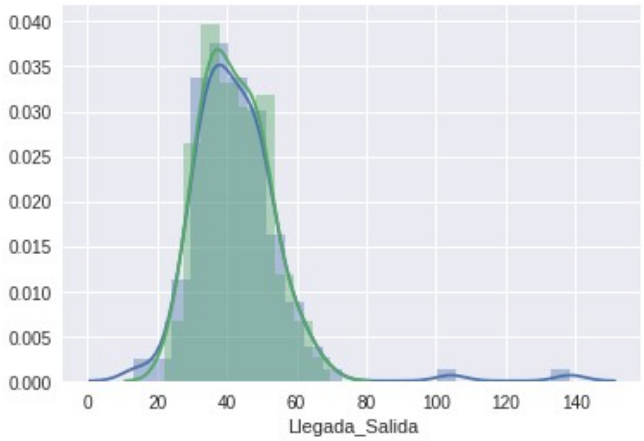
Luego hicimos la eliminación de Outliers para los campos: Intervalo_cliente, Llegada_Pedido, Pedido_Orden, Orden_Salida y Llegada_Salida. Hicimos gráficas que muestran la diferencia entre los campos con Outliers (color azul) y los campos posterior a la eliminación(color verde):





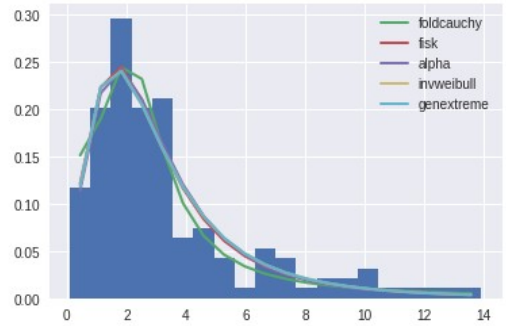






Una vez eliminados los outliers pasamos a observar la distribución que mejor ajusta a los intervalos a estudiar. Estos fueron los resultados:

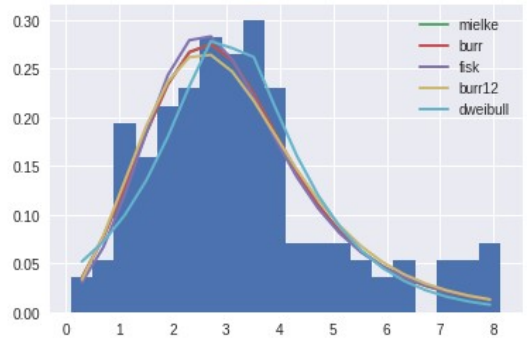
	sumsquare_error
foldcauchy	0.011667
fisk	0.011774
alpha	0.012195
invweibull	0.012710
genextreme	0.012710



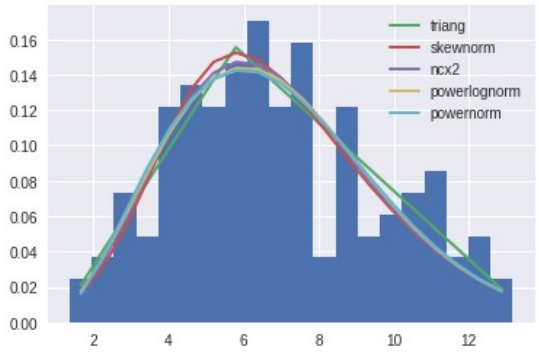
Para el **intervalo de llegada entre grupos de clientes**, la distribución que mejor ajustó fue la **Distribución Plegada de Cauchy**

	sumsquare_error
mielke	0.029408
burr	0.029408
fisk	0.031707
burr12	0.031957
dweibull	0.032181

La distribución que mejor ajustó para el **intervalo de tiempo entre la llegada de comensales y la toma de pedidos** fue la **Distribución Mielke**



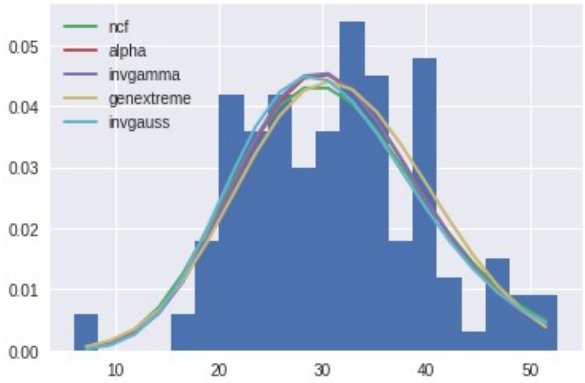
	sumsquare_error
triang	0.013611
skewnorm	0.014382
ncx2	0.014529
powerlognorm	0.014561
powernorm	0.014624



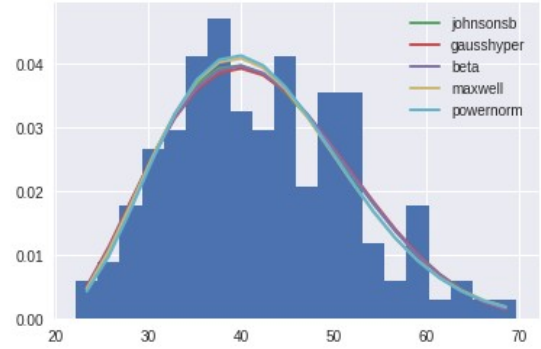
La **Distribución Triangular** fue la que mejor ajustó al **intervalo entre la toma de orden y la llegada de los platillos a la mesa.**

El **intervalo entre la toma de orden y cuando los comensales abandonaban el establecimiento**, ajustó a un distribución **F no central**

	sumsquare_error
ncf	0.001755
alpha	0.001765
invgamma	0.001767
genextreme	0.001773
invgauss	0.001774



	sumsquare_error
johnsonsb	0.000790
gausshyper	0.000795
beta	0.000796
maxwell	0.000818
powernorm	0.000824



El intervalo entre la llegada y la salida de comensales ajustó a una **Distribución Johnson SB**

Una vez teniendo las distribuciones de cada intervalo a usarse hicimos una simulación usando la librería Simpy. Nos dimos cuenta que al establecimiento llegan entre 20 y 40 grupos de comensales de 1:00 pm a 3:00 pm, un intervalo de tiempo que consideramos que en general es el que tiene más demanda lo que desde inicio fue el objeto de estudio.

También notamos que aparentemente el número de mesas del restaurante es adecuado, pues, como en promedio llegan entre 20 y 40 grupos de clientes, cuando llegan en el intervalo de 20 a 34 grupos no hay escases de mesas, a partir de 34 comienza a haber un pequeño deficit, pero es hasta >40 donde el deficit es más grande. El hecho de que los comensales no encuentren mesa implica que después de cierto tiempo decidiran ya no consumir en el lugar, lo que representa pérdidas para el restaurante.

Conclusiones:

Podemos observar gracias a la utilización de simpy, ya que como en general el número de comensales en hora de comida ronda entre 20 y 38 grupos de personas, supone un problema en des abasto de mesas. Dado que si llegan 34 clientes en promedio un cliente no es atendido, a partir de 36 incrementa a 2 y de 40 en a delante el des abasto es mucho mayor.

Entonces proponemos por lo menos agregar dos mesas en la parte de arriba para así contrarrestar ese problema.

Otra alternativa es hacer la mesas más pequeña, debido a que por lo general llegan en pareja y no ocupan en su totalidad todo el espacio de una mesa.

Este proyecto ayudo a darnos cuenta la facilidad de modelar eventos del día a día usando las correctas herramientas.

Bibliografía

Alvarez, M. (2003). *Qué es Python.* noviembre 20, 2017, de DesarrolladorWeb Sitio web: https://desarrolloweb.com/articulos/1325.php

Santana, C. (2013). *¿Qué es Python?*. noviembre 20, 2017, de Codejobs Sitio web: https://www.codejobs.biz/es/blog/2013/03/02/que-es-python

Sánchez, O. (2010). *Simulación de Sistemas Estocásticos.* noviembre 20, 2017, de TIS Consulting Sitio web: http://tisconsulting.org/es/blog/simulating-stochastic-systems/

Baltazar-Larios, F. & López S. (2017). Simulación Estocástica. noviembre 20, 2017, de UNAM Sitio web: http://sistemas.fciencias.unam.mx/~silo/Cursos/sim/notas.pdf

Herrera, J. & Sánchez, L. (2013). Numpy. noviembre 21, 2017, de Computación científica con Python para módulos de evaluación continua en asignaturas de ciencias aplicadas Sitio web: http://webs.ucm.es/info/aocg/python/modulos_cientificos/numpy/index.html

Herrera, J. & Sánchez, L. (2013). Scipy. noviembre 21, 2017, de Computación científica con Python para módulos de evaluación continua en asignaturas de ciencias aplicadas Sitio web: http://webs.ucm.es/info/aocg/python/modulos_cientificos/scipy/index.html

Bioinformatics. (2015). pandas. noviembre 21, 2017, de COMAV Sitio web: https://bioinf.comav.upv.es/courses/linux/python/pandas.html

Moya, R. (2015). Pandas en Python. noviembre 21, 2017, de jarroba Sitio web: https://jarroba.com/pandas-python-ejemplos-parte-i-introduccion/

Heckert, A. (2009). *MIEPDF*. Noviembre 26, 2017, de NIST Sitio web: http://www.itl.nist.gov/div898/software/dataplot/refman2/auxillar/miepdf.htm

Epidat. (2014). *Distribuciones de Probabilidad*. Noviembre 26, 2017, de Epidat Sitio web: https://www.sergas.es/Saude-publica/Documents/1899/Ayuda_Epidat_4_Distribuciones_de_probabilidad_Octubre2014.pdf

MathWorks. (2017). *Noncentral F Distribution*. Noviembre 26, 2017, de MathWorks Sitio web: https://es.mathworks.com/help/stats/noncentral-f-distribution.html?s_tid=gn_loc_drop

NIST. (1997). *FCACDF.* Noviembre 26, 2017, de NIST Sitio web: http://www.itl.nist.gov/div898/software/dataplot/refman2/auxillar/fcacdf.pdf

Lagos, I. (2003). S*istema de familias de distribuciones de Johnson, una alternativa para el manejo de datos no normales en cartas de control.* Noviembre 26, 2017, de Revista

Colombiana de Estadística Sitio web: https://www.emis.de/journals/RCE/V26/V26_1_25LagosVargas.pdf