



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

 etsinf

Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Simulación de datos de sensores industriales

TRABAJO DE FIN DE MÁSTER

Máster en Big Data Analytics

Autor: Pedro Henrique Mano Figueiredo Fernandes

Tutores: Francisco Sánchez Cid, José Ramón Navarro Cerdan

Curso 2015-2016

Abstract

Los entornos industriales hacen uso de sensores para obtener mediciones de varios factores en su cadena de producción. En las fases iniciales, la cantidad de datos generados es muy reducida, por lo que no es significativa para permitir la aplicación de técnicas de Big Data. Estas técnicas pueden descubrir muchos patrones en los datos, útiles para detección de anomalías o predicción futura. El proyecto descrito en este documento busca simular nuevos datos a través de un pequeña cantidad de datos generados por sensores industriales. La aplicación de estas técnicas se extiende a todo el contexto de proyectos IoT (*Internet of Things*), donde los sensores representan la principal fuente de datos.

Palabras clave: Machine learning, Big Data, PCA

Abstract

The industrial environments make use of sensors for acquiring metrics on several variables in their production pipeline. In the initial phases, the generated data has very low volume, making it hard to apply Big Data techniques. These techniques can bring much insight over the data, useful for anomaly detection or future prediction. The project described in this document seeks the simulation of new data using a small amount of data generated by industrial sensors. The application of such techniques can be extended to the entire context of IoT (*Internet of Things*), where sensors represent the main source of data.

Key words: Machine learning, Big Data, PCA

Índice general

Índice general	v
Índice de figuras	vii
Índice de tablas	viii
<hr/>	
1 Introducción	1
1.1 Estructura del documento	1
2 Descripción del problema	3
3 Estado del Arte	5
4 Solución propuesta	7
4.1 Reducción de dimensionalidad - PCA	8
4.1.1 Singular Value Decomposition	10
4.1.2 El método NIPALS	11
4.2 Series temporales - ARIMA	11
4.2.1 Estacionariedad	12
4.2.2 Parámetros del modelo y predicción	14
5 Experimentación y resultados	17
5.1 Simulación con datos controlados	17
5.2 Dataset	23
5.2.1 Estructura	23
5.2.2 Transformación	23
5.2.3 Distribución de los datos	24
5.3 PCA	24
5.3.1 PCA iterativo - NIPALS	25
5.3.2 Test de hipótesis - Estadístico T-cuadrado de Hotelling	28
5.3.3 Simulación con distribución Gaussiana	29
5.4 ARIMA	31
5.4.1 Preparación de la serie temporal	31
5.4.2 Estacionariedad de la serie temporal	32
5.4.3 Predicción ARIMA con PCA de todo el <i>dataset</i>	32
5.4.4 Predicción ARIMA con PCA de un día	33
5.4.5 Simulación con búsqueda de puntos cercanos de la Gaussiana	34
5.5 Big Data	37
5.5.1 MongoDB	38
5.5.2 Spark	41
6 Conclusiones	43
7 Trabajos futuros	45
Bibliografía	47

Índice de figuras

4.1	Datos con distribución Gaussiana y las dos primeras componentes PCA.	8
4.2	Proyección de puntos a una línea.	9
4.3	Ejemplo de serie estacionaria.	12
4.4	Ejemplo de serie no estacionaria.	12
4.5	Ejemplo de serie para aplicar el test de Dickey-Fuller.	13
4.6	Ejemplo de serie con diferenciación.	14
4.7	ARIMA iterativo: AR(1).	15
4.8	ARIMA iterativo: MA(1).	15
4.9	ARIMA iterativo: AR(1) y MA(1).	15
4.10	ARIMA iterativo: AR(2) y MA(2).	16
4.11	Predicciones del modelo ARIMA(2, 1, 2).	16
5.1	$\sin - 1$ con ruido 0,05.	18
5.2	Inversión de $\sin - 1$ con ruido 0,05, usando 5 componentes.	18
5.3	Inversión de $\sin - 1$ con ruido 0,05 según el numero de componentes.	19
5.4	Simulación de la primera componente, usando distribución Gaussiana.	20
5.5	Inversión de $\sin - 1$ con ruido 0,05, usando distribución Gaussiana.	20
5.6	Simulación de la componente 1 con datos controlados.	21
5.7	Simulación de las componentes 2, 3, 4 y 5 con datos controlados.	22
5.8	Inversión de $\sin - 1$ con ruido 0,05 con búsqueda en la simulación Gaussiana.	22
5.9	Datos originales de la variable APVs.	25
5.10	Datos invertidos de la variable APVs.	25
5.11	Datos originales de la variable H6x.	26
5.12	Datos invertidos de la variable H6x.	26
5.13	Datos originales de la variable ACPx.	27
5.14	Datos invertidos de la variable ACPx.	27
5.15	Datos invertidos de la variable ACPx con 14 componentes.	27
5.16	Datos originales y simulados de la componente 1.	29
5.17	Datos originales y simulados de las componentes.	30
5.18	Datos originales y simulados de la variable APHu.	30
5.19	ARIMA para la componente 1 calculada con PCA sobre todo el <i>dataset</i> .	33
5.20	Predicción ARIMA <i>out-of-sample</i> para la componente 1 calculada con PCA de un día.	33
5.21	Simulación para la componente 1, usando SARIMAX con re-alimentación.	35
5.22	Simulación con búsqueda en los datos Gaussianos para la componente 1.	36
5.23	Simulación con búsqueda en los datos Gaussianos.	36
5.24	Gráficos de simulación en tiempo real.	40

Índice de tablas

5.1	Test T^2 de Hotelling para los varios números de componentes.	29
5.2	Consumo de memoria de una matriz numpy según la cantidad de datos. . .	37

CAPÍTULO 1

Introducción

Este proyecto tiene como fuente de datos uno o varios sensores de un mecanizado industrial de inyección de plástico. Los sensores efectúan mediciones de varios factores con una regularidad temporal, generando así muestras de datos en determinados intervalos de tiempo. Cuando se trata de sensores, es normal que los datos no tengan la calidad necesaria para aplicar técnicas matemáticas. Hay que tener en cuenta que la frecuencia de las mediciones no es necesariamente constante, que pueden existir interrupciones con ruido y redundancias. Además, en el ámbito de IoT (*Internet of Things*), se añade la inherente conectividad de los aparatos como factor de complejidad.

Las técnicas de Big Data, en particular *Machine Learning*, necesitan gran cantidad de datos para poder inducir modelos matemáticos que expliquen esos datos. Cuanto más datos mejor, para un aprendizaje más robusto y fiable. Sin embargo, al principio de los proyectos, la cantidad de datos recolectados suele ser pequeña y insuficiente para poder extraer conocimiento significativo de los mismos. El objetivo de este proyecto es aprender y simular el comportamiento de uno o varios sensores a través de una cantidad reducida de datos recolectados en los mismos.

Una vez preparado el *dataset*, es necesario aprender un modelo matemático que lo describa. Las técnicas usadas para ese efecto son del ámbito de *Machine Learning*.

1.1 Estructura del documento

Este documento se estructura de la siguiente forma: la descripción del problema; el estado del arte; un análisis sobre el *dataset* y respectivas transformaciones para adecuar los datos; las soluciones propuestas y respectiva base teórica; la experimentación, resultados y discusión; conclusiones del trabajo realizado y posibles mejoras; y termina con las fuentes bibliográficas que han servido de base del estudio.

CAPÍTULO 2

Descripción del problema

El paradigma de IoT (*Internet of Things*) tiene mucha presencia en la industria. Se utilizan sensores para controlar determinados factores en las cadenas de producción. Los datos generados son de gran importancia. Con el debido tratamiento y análisis, estos datos pueden ayudar descubrir conocimiento nuevo. En el entorno industrial, ese conocimiento se puede traducir en la anticipación de anomalías o mejoras de rendimiento de las máquinas, con todo el beneficio que eso conlleva.

La extracción de conocimiento de los datos se hace con técnicas estadísticas de *Machine Learning*. Estas técnicas están relacionadas con Big Data por la simple regla de que cuantos más datos mejor. Los modelos matemáticos que explican los datos son más robustos si hay mucho volumen de datos. Sin embargo, en las etapas iniciales de implantación de IoT no hay mucha cantidad de datos, lo que dificulta la aplicación de *Machine Learning*.

Hay que tener en cuenta que los datos de sensores suelen tener problemas de calidad, sobre todo relacionados con ruido, redundancias o frecuencia de muestreo. Por ejemplo, si un sensor mide la temperatura de una máquina, esa medición sufre interferencias del ambiente, por lo que la medición no es completamente objetiva. Ese efecto se conoce por ruido. Cuanto a las redundancias, pueden suceder en casos de pérdida de conectividad, en que el sensor vuelve a dar una métrica aunque la haya dado ya anteriormente. Esto sería, también, un caso de cambio de la frecuencia de muestreo, que se traduce en irregularidades de la serie temporal asociada a los datos.

El objetivo de este proyecto es aprender y simular datos de sensores industriales. Los nuevos datos, de mucho mayor volumen, podrán ser usados para probar y depurar los algoritmos de detección de averías y de predicción desarrollados para entornos Big Data.

CAPÍTULO 3

Estado del Arte

PCA (*Principal Component Analysis*) es la técnica más usual de reducción de dimensionalidad, en que los datos son reducidos a lo esencial. PCA se usa para descomponer un conjunto de datos multivariante en el grupo de componentes ortogonales que mejor explican la varianza. El ajuste a los datos es de tipo lineal. Las observaciones originales son proyectadas a un espacio más reducido, a lo largo de las componentes principales. Si se generan nuevos valores para esas proyecciones, al transformar de nuevo al espacio original, se obtienen observaciones nuevas, sintéticas, y sin afectar a la estructura de varianza-covarianza original. De este modo, PCA se puede utilizar como un modelo de simulación bastante directo, como explicado en el artículo *The Truth about Principal Components and Factor Analysis* [2].

Los problemas de ruido y redundancias en los datos son comunes y están bastante estudiados. Para solucionar ese problema, las técnicas de reducción de dimensionalidad, suelen tener buenos resultados. Las primeras componentes son las que mayor varianza explican y por eso se llaman principales. El orden de las componentes es relevante, la primera es la más explicativa, la segunda la que más explica quitando la primera, y así sucesivamente. Así que es de esperar que lo que están explicando las últimas sea, realmente, el ruido de la señal. El artículo *A Tutorial on Principal Component Analysis* [1] deja patente esa evidencia en el objetivo de filtrar el ruido y redundancias para obtener la dinámica importante de los datos.

Las observaciones de los sensores tienen una implícita ordenación cronológica (y así lo tendrán también las componentes principales). El estudio de esa relación se denomina análisis de series temporales. Con este análisis se puede llegar a explicar mejor la serie y hacer predicciones sobre el futuro de la misma.

Los modelos *Exponential smoothing* y *AutoRegressive Integrated Moving Average* (ARIMA) son los más usados para predicción de series temporales. *Exponential smoothing* se basan en la descripción de tendencia y de variación estacional. Pero, en general, observaciones sucesivas suelen tener dependencia entre si, como se describe en el libro *Introduction to Time Series Analysis and Forecasting* [4]. Para incorporar esta dependencia se usan los modelos ARIMA. Los modelos ARIMA son un flexible y poderoso método para análisis de series temporales y predicción. A lo largo de los años, se vienen usando con éxito en varios problemas de investigación y en la práctica.

Cuando la cantidad de datos es muy grande, empiezan a surgir problemas de procesamiento con las técnicas de almacenamiento y programación convencionales.

Las bases de datos relacionales se sustentan en la integridad y consistencia de relaciones entre los datos que almacenan, lo que supone una exigencia computacional que no es adecuada para un volumen muy grande de datos. Por otro lado, las bases de datos

NoSQL, como *MongoDB*, eliminan las restricciones de integridad y relaciones entre los datos, en favor de una capacidad de respuesta más efectiva para volúmenes grandes.

En lo que respecta al procesamiento de esos datos, la programación convencional tiene dificultades, sobre todo relacionadas con la cantidad de memoria necesaria para trabajar con ellos. Así, han surgido paradigmas como *MapReduce*, que permiten procesar los datos de forma distribuida, en paralelo en un *cluster*. La implementación más popular de *MapReduce* es *Apache Hadoop*, que se sustenta en el almacenamiento para la carga de datos. Recientemente ha surgido un nuevo *framework*, llamado *Apache Spark*, con la misma filosofía de *Apache Hadoop*, pero que tiene la evolución de cargar los datos en memoria, lo que se traduce en prestaciones bastante mejores que las de su antecesor.

CAPÍTULO 4

Solución propuesta

La solución propuesta para la simulación de datos empieza con la proyección de los datos del espacio multivariante original a un espacio ortogonal más reducido. Esta técnica de *Machine Learning* se conoce por PCA (*Principal Component Analysis*). El nuevo espacio (con menos dimensiones) contiene las componentes que mejor explican los datos y las respectivas proyecciones de las características originales en esas componentes. La generación de nuevos valores para esas proyecciones permite, al transformar al espacio original, obtener nuevos datos sintéticos, manteniendo la estructura de varianza-covarianza original. Esa es la estrategia usada para la primera simulación, más directa, usando valores aleatorios con distribución normal para las proyecciones. Aparte de esa capacidad de simulación de PCA, aprovecharemos la implícita capacidad de reducción de ruido y de redundancia en los datos.

Para las proyecciones de cada componente, se calcula la media y desviación típica y se generan nuevos valores con distribución Gaussiana. Esta simulación puede ser de millones o miles de millones de nuevos puntos. La simulación Gaussiana servirá como base del estudio, una vez que, al ser invertida, genera una población que es idéntica a la original. La comprobación de dicha similitud de poblaciones se hace a través del estadístico T-cuadrado de Hotelling. Y ésta será nuestra hipótesis nula.

Los datos originales tienen un determinado sentido temporal. Las componentes lo tendrán también, así que se procede al análisis de la serie temporal de esas componentes. El modelo propuesto es ARIMA (*Autoregressive Integrated Moving Average*). Los modelos de series temporales se ajustan a los datos para mejor comprensión de estos o para predicción de nuevos puntos en la serie. El objetivo es dotar la simulación de un patrón temporal aprendido previamente.

Teniendo en cuenta que el estudio se sustenta en la hipótesis nula de la simulación Gaussiana, para cada predicción de las varias componentes con marca temporal, se buscará el punto multi-dimensional más cercano en la simulación Gaussiana. El punto elegido se usará para re-alimentar el modelo ARIMA aprendido. Llegados aquí, tenemos nuevos valores simulados de las componentes y además con carácter temporal. Al re-proyectar este espacio al espacio original obtenemos nuevos valores (simulados) para las características originales.

La simulación Gaussiana necesita tener una gran cantidad de puntos para que la simulación final sea efectiva. Cuantos más mejor, para que la búsqueda encuentre puntos más cercanos. La cantidad mencionada debe ser de la orden de millones o miles de millones de puntos. Con un volumen de datos muy grande, los paradigmas de programación convencionales no tienen capacidad de respuesta, sea por capacidad de almacenamiento o por memoria. Aquí es donde las herramientas Big Data asumen un papel fundamental para la viabilidad de este estudio. En particular, se adoptan *Spark* y *MongoDB* para

permitir escalar la parte de simulación Gaussiana, su almacenamiento y la búsqueda de puntos cercanos.

4.1 Reducción de dimensionalidad - PCA

PCA (*Principal Components Analysis*) pertenece a la familia de aprendizaje no supervisado (*unsupervised learning*), de *Machine Learning*, donde no existen etiquetas o clases a predecir. Los datos de entrenamiento son simplemente observaciones, sin estar clasificados. Las técnicas de *unsupervised learning* se usan para descubrir grupos de similitud (*clusters*) en los datos; o para determinar la distribución de los datos en el espacio original; o también, que interesa a este estudio, para proyectar los datos en espacios de menos dimensiones.

Pocas dimensiones suelen ofrecer mejor *insight* sobre los datos. Como primera ventaja, permiten la aplicación de técnicas de visualización - resulta muy difícil visualizar datos en más de 3 dimensiones (incluso en 3 dimensiones puede ser difícil). Por otro lado, las componentes que no son principales (menos explicativas) suelen ser ruido o redundancia - con esta técnica se pueden atenuar los efectos de estos. La simulación de datos puede usar la ventaja de la eliminación de ruido y redundancia, generando así datos con mayor entidad. Teniendo en cuenta que, a través de PCA, se obtienen no solo las componentes principales como las respectivas proyecciones de los datos en esas componentes, se pueden generar valores nuevos para las proyecciones y así simular nuevos datos sintéticos al re-proyectar al espacio original. Ésta es la suposición de la simulación propuesta.

PCA descompone un *dataset* multivariante en un determinado número de componentes ortogonales que son los que más explican la varianza de ese *dataset*. El número de componentes es menor o igual al número de características originales. Por ejemplo, si dos variables son directamente proporcionales, basta con una sola componente para explicar el comportamiento de las dos - esa correlación quedará representada en la matriz calculada por PCA. Si se convierte esa componente al espacio original de dos características, se obtienen los valores originales.

La dirección de la primera componente principal es aquella a lo largo de la cual las observaciones varían más.

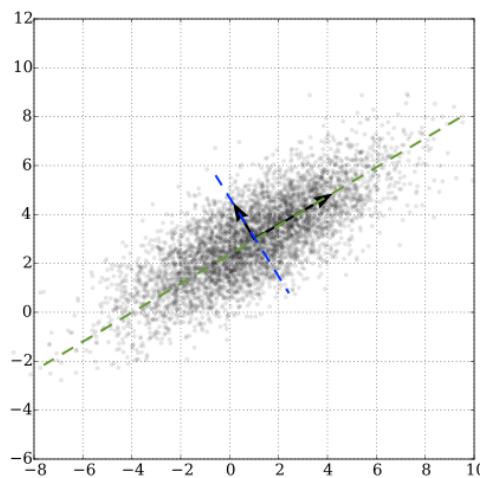


Figura 4.1: Datos con distribución Gaussiana y las dos primeras componentes PCA.

Por ejemplo, la Figura 4.1 representa un conjunto de datos de dos dimensiones con distribución Gaussiana. La flecha más grande representa la dirección de la primera componente principal de los datos. Se puede verificar a simple vista que esta es la dirección con mayor variabilidad en los datos. Si trazamos una línea imaginaria a lo largo de esa flecha (como la línea verde) y proyectamos en ella los datos, tendremos la proyección con mayor variabilidad.

La dirección de segunda componente está representada por la otra flecha. Al proyectar los datos en la línea imaginaria azul, podemos imaginar que el espectro de variabilidad es inferior al la primera.

La proyección de un punto a una línea es muy sencilla: basta con buscar el punto más cercano en esa línea. Los puntos azules en la Figura 4.2 representan la proyección de los puntos originales en la línea.

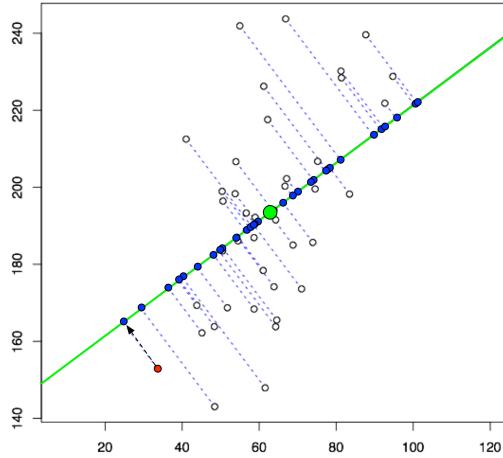


Figura 4.2: Proyección de puntos a una línea.

En términos matemáticos, PCA es una transformación lineal ortogonal. Consideremos la matriz X , constituida por n líneas de observaciones y m características. El objetivo es proyectar los datos a un espacio con dimensionalidad $d < m$, de forma que se maximice la varianza de los datos proyectados. La transformación de PCA se define por la ecuación:

$$Y = X \cdot w \quad (4.1)$$

Que se interpreta de la siguiente forma: una matriz $m \times k$ de pesos (*loadings*) w transforma la matriz $n \times m$ X en la matriz $n \times k$ de componentes principales (*scores*) Y .

Para obtener la primera componente, el primer vector de *loadings* w debe maximizar la varianza. Siguiendo el razonamiento del libro *Pattern Recognition and Machine Learning* [3], consideremos el vector w_1 , que, por conveniencia (y sin pérdida de generalización) debe ser un vector unitario de modo que $w_1^T w_1 = 1$. Cada punto de X , x_i , es proyectado para el escalar $w_1^T x_i$. La media de los datos proyectados es $w_1^T \bar{x}$, donde \bar{x} es la media dada por:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i \quad (4.2)$$

Y la varianza de los datos proyectados:

$$\frac{1}{n} \sum_{i=1}^n x_i \{ w_1^T \cdot x_i - w_1^T \cdot \bar{x} \} = w_1^T \cdot S \cdot w_1 \quad (4.3)$$

Donde S es la matriz de varianzas-covarianzas definida por:

$$S = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x}) \cdot (x_i - \bar{x})^T \quad (4.4)$$

Ahora maximizamos la varianza de los datos proyectados $w_1^T \cdot S \cdot w_1$ con respecto a w_1 . Hay que restringir la maximización para prevenir que tienda para infinito. La restricción viene de la normalización $w_1^T w_1 = 1$. Para forzar la restricción introducimos un multiplicador de Lagrange λ_1 :

$$w_1^T \cdot S \cdot w_1 + \lambda_1 (1 - w_1^T \cdot w_1) \quad (4.5)$$

Que tiene un punto estacionario cuando:

$$S \cdot w_1 = \lambda_1 \cdot w_1 \quad (4.6)$$

Esto define que w_1 es un vector propio de S . Si multiplicamos las parte izquierda por w_1^T , teniendo en cuenta que $w_1^T w_1 = 1$, vemos que la varianza es dada por:

$$w_1^T \cdot S \cdot w_1 = \lambda_1 \quad (4.7)$$

Así que la varianza será máxima cuando se defina w_1 igual al vector propio con el máximo valor propio λ_1 .

4.1.1. Singular Value Decomposition

PCA está muy relacionado con una técnica matemática llamada *Singular Value Decomposition* (SVD), tanto que muchas veces los nombres se usan intercambiados. De hecho, el algoritmo de PCA de scikit-learn usa la descomposición SVD de numpy. SVD es un método más general de entender el cambio de base.

La representación de la descomposición en valores singulares es:

$$X = U \Sigma W^T \quad (4.8)$$

Donde:

- U es una matriz $n \times n$, en que las columnas son vectores unitarios ortogonales de tamaño n .
- Σ es una matriz diagonal $n \times m$ de números positivos σ_i , llamados valores singulares de X .
- W es una matriz $m \times m$, cuyas columnas son vectores unitarios ortogonales de tamaño p .

La ecuación 4.2 indica que una matriz X puede ser convertida en una matriz ortogonal, una matriz diagonal y otra matriz ortogonal. O, dicho de otra forma, corresponde a una rotación, un estiramiento y otra rotación.

4.1.2. El método NIPALS

SVD es el método general para cálculo de PCA. Pero si lo aplicamos a un *dataset* multidimensional muy grande puede tener problemas, porque calcula todas las componentes de una vez (calcula toda la matriz de varianzas-covarianzas de X). Esto implica un gran consumo de memoria y CPU. NIPALS significa *Non-linear Iterative Partial Least Squares* y es un algoritmo iterativo que solo calcula las primeras componentes principales.

Para cada componente, el algoritmo NIPALS itera hasta que el *score* y_1 tenga convergencia. Antes de la iteración, se atribuye un valor aleatorio al *score* y_1 . La convergencia se detecta cuando la variación del *score* y_1 respecto a la iteración anterior es muy pequeña.

En cada iteración, para encontrar el *loading* w_1 se proyecta X hacia y_1 - regresión del *score* respecto a cada columna de X - y se guarda el coeficiente de regresión en el *loading*. Luego se proyecta X hacia w_1 para encontrar el *score* y_1 - regresión del *loading* respecto a cada línea de X - y se guarda el coeficiente de regresión en el *score*.

Una vez convergido, para calcular la siguiente componente se resta el producto $y_1 \cdot w_1$ a X . Esto corresponde a restar a X la parte de los datos explicada por y_1 y w_1 .

4.2 Series temporales - ARIMA

Las series temporales surgen frecuentemente en la monitorización de procesos industriales. Cada observación de un sensor tiene una marca temporal asociada. El conjunto de las observaciones ordenadas de forma cronológica se denomina serie temporal. En este capítulo se demostrará el importante valor que tiene esa marca temporal en el análisis de las señales.

El análisis de series temporales asienta en la suposición de que los datos tomados en determinados puntos en el tiempo tienen una estructura interna (correlación, tendencia o variación estacional) que hay que tener en cuenta. El objetivo del análisis de series temporales es extrapolar datos futuros a través de los datos pasados.

ARIMA significa (*Autoregressive Integrated Moving Average*) y, junto con (*Exponential Smoothing*), es uno de los métodos más populares de análisis de series temporales. ARIMA incorpora formalmente la dependencia entre observaciones sucesivas en el modelo, mientras *Exponential Smoothing* no lo hace de forma tan eficiente.

La componente AR (*Autoregressive* - auto-regresiva) del modelo ARIMA se refiere al uso de observaciones anteriores en una ecuación regresiva para predecir valores futuros. A esta componente se asocia un parámetro p , que corresponde al numero de puntos anteriores a incluir en el modelo. La componente MA (*Moving Average* - media móvil) representa el error del modelo como combinación de términos de error anteriores. El parámetro asociado es q y corresponde al numero de términos de error anteriores a usar. La componente I (*Integrated*) indica la diferenciación de los datos, que corresponde a calcular la diferencia entre los valores y sus antecesores. El parámetro d es el numero de diferenciaciones a aplicar en el modelo. De esta forma, los modelos ARIMA son generalmente definidos por $ARIMA(p, d, q)$.

Es importante resaltar que el modelo $ARIMA(p, d, q)$ asume que la serie no tiene variación estacional y tendencia. Potencialmente es necesario eliminar esas características antes de crear el modelo. Esto nos lleva al concepto de *stationarity* (estacionaridad).

4.2.1. Estacionaridad

Las series temporales son diferentes de un problema de regresión normal porque son dependientes del tiempo. Así que el supuesto básico de la regresión lineal de que las observaciones son independientes no se verifica en este caso. Para poder aplicar el modelo ARIMA es necesario que la serie sea estacionaria. Una serie es estacionaria cuando la media y varianza, si existen, no cambian en función del tiempo. Como ARIMA usa los períodos anteriores para modelar el comportamiento de la serie, es importante que esta sea estable para tener más acierto. Muchas veces la estacionaridad se puede detectar visualmente, como se presenta en los ejemplos siguientes.

La Figura 4.3 es un ejemplo de una serie estacionaria, donde los valores oscilan con una varianza constante alrededor de una media de aproximadamente 69,0.

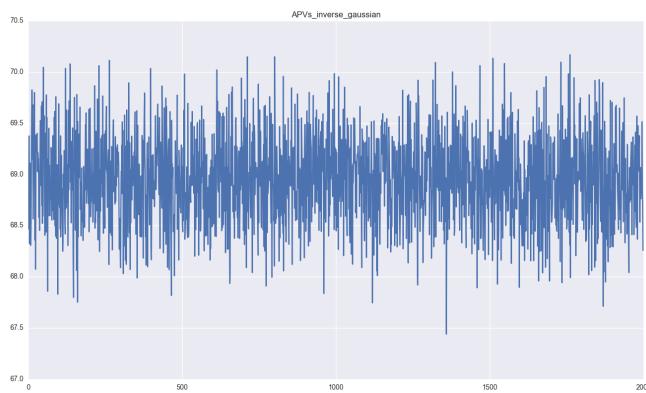


Figura 4.3: Ejemplo de serie estacionaria.

La Figura 4.4, sin embargo, no presenta señales de media y varianza constantes. De hecho se detecta una ligera tendencia decreciente, lo que la hace no estacionaria.

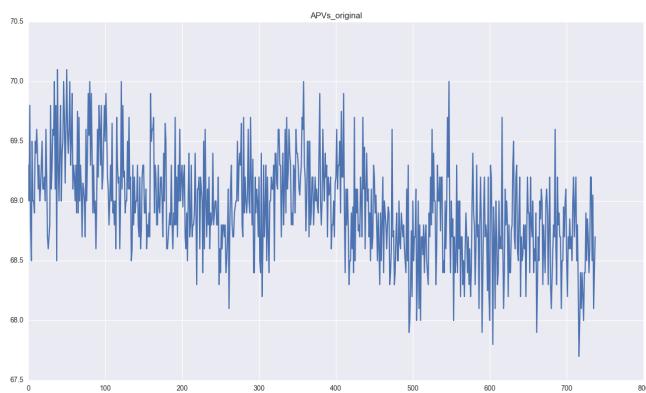


Figura 4.4: Ejemplo de serie no estacionaria.

Otra propiedad importante para que ARIMA sea efectivo es que la serie tenga períodos fijos. Cada observación debe ser dada a una frecuencia constante. Una vez los puntos sean equidistantes entre sí, la variable tiempo, o índice, ya no necesita existir explícitamente. Muchas veces se usa el tiempo para la representación gráfica, sin embargo esa variable no se usa en el modelo en sí. En los gráficos anteriores no se usa la marca de tiempo en el eje x , simplemente un índice numérico.

Hay casos donde la estacionaridad es más difícil de determinar a simple vista. Además el análisis necesita una prueba formal de la estacionaridad. Para eso se usa el test estadístico de Dickey-Fuller, conocido por *augmented Dickey-Fuller test* (ADF). La hipótesis nula asume que la serie es no estacionaria. Valores altos del p-valor indican que el test no rechaza la hipótesis nula y, por lo tanto, la serie es no estacionaria. Por otro lado, valores bajos de p-valor indican que la serie es estacionaria. En general, tomando el 5 % de confianza, si el p-valor es mayor que 0.05 la serie es no estacionaria. Se demuestra en el ejemplo siguiente.

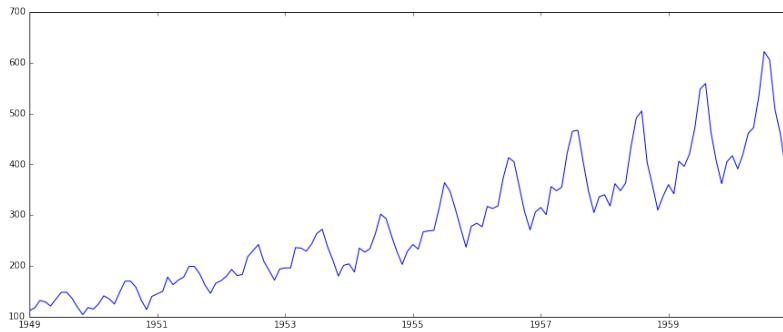


Figura 4.5: Ejemplo de serie para aplicar el test de Dickey-Fuller.

La serie de la Figura 4.5 tiene una tendencia creciente, por lo que no debería ser estacionaria. Apliquemos el test de Dickey-Fuller, presente en módulo `statsmodels` de Python:

```
from statsmodels.tsa.stattools import adfuller
dftest = adfuller(ts)
print('Test Statistic', dftest[0])
print('p-value', dftest[1])
print('Confidence levels', dftest[4])
```

Listing 4.1: Test de Dickey-Fuller de `statsmodels` de Python.

```
Test Statistic 0.815368879206
p-value 0.991880243438
Confidence levels {'1%': -3.4816817173418295, '5%':
-2.8840418343195267, '10%': -2.5787700591715979}
```

Listing 4.2: Resultados del test de Dickey-Fuller de `statsmodels` de Python.

Los resultados del Listing 4.1 demuestran que la hipótesis nula no puede ser descartada, una vez que *p-value* es cercano a 1. El resultado del estadístico lo confirma también: para que se descarte la hipótesis nula, el valor del estadístico debe ser inferior a alguno de los valores de los intervalos de confianza.

La serie tendrá que ser transformada hasta ser estacionaria. Una de las técnicas más comunes para conseguirlo es la diferenciación, que consiste en calcular las diferencias entre observaciones consecutivas. La diferenciación ayuda a estabilizar la media de la serie, eliminando la tendencia y la estacionalidad. El primer nivel de diferenciación de la serie y se define por:

$$y_t - y_{t-1} \quad (4.9)$$

Siendo t un determinado punto en el tiempo. Se pueden aplicar más niveles de diferenciación, pero con un solo nivel suele ser suficiente.

Aplicando diferenciación a la serie anterior obtenemos la serie transformada representada en la Figura 4.6 y respectivos resultados en Listing 4.3.

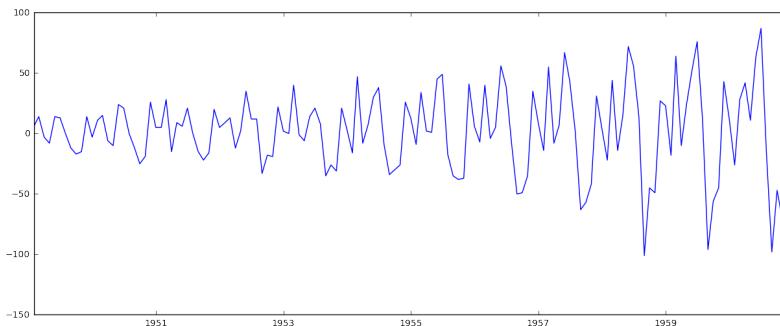


Figura 4.6: Ejemplo de serie con diferenciación.

```
Test Statistic -2.82926682417
p-value 0.0542132902838
Confidence levels {'1%': -3.4816817173418295, '5%': 
-2.8840418343195267, '10%': -2.5787700591715979}
```

Listing 4.3: Resultados del test de Dickey-Fuller de statsmodels de Python.

El test de Dickey-Fuller demuestra que el p-valor ha bajado considerablemente, para valores casi inferiores a 0,05, por lo que podemos decir con un 95 % de confianza que la serie es estacionaria ahora. El análisis visual también permite verificar que la tendencia se ha eliminado.

4.2.2. Parámetros del modelo y predicción

Como mencionado anteriormente, el modelo ARIMA es una ecuación lineal que depende de los parámetros p, d, q , correspondientes a las partes autoregresiva (AR), integración (I) y de medias móviles (MA) del modelo, respectivamente.

El paso de diferenciación es aplicado internamente por el modelo, a través del parámetro d , que corresponde a la parte I (*Integrated*) de ARIMA. No hace falta pasar los datos diferenciados al modelo. Para la determinación de los parámetros p y q óptimos se usan formalmente los gráficos de correlación ACF (*Auto-Correlation Function*) y PACF (*Partial Auto-Correlation Function*). Estos gráficos se han usado en fases iniciales de este estudio, pero salen del ámbito de investigación.

Otra solución es la minimización del error, o valor residual, de la predicción. La diferencia entre la observación y_t y el valor obtenido a través del ajuste del modelo a los datos (o valor ajustado y'_t) se llama valor residual, y es definido por:

$$e_t = y_t - y'_t \quad (4.10)$$

A través de los residuales se puede calcular el *root mean squared error* (RMSE). El modelo con menor RMSE es el mejor. Así que la solución consiste en iterar, probando todas las combinaciones de los parámetros (p, d, q) de ARIMA, para obtener el modelo con menor RMSE. Se ejemplifica el proceso a continuación.

Se ha comprobado antes que la serie necesita un nivel de diferenciación para ser estacionaria: $d = 1$. La componente I de ARIMA será tendrá, así, valor 1.

Empezamos la iteración con un nivel para la componente auto-regresiva: $p = 1$, que corresponde a $AR(1)$. El resultado del ajuste del modelo se puede ver en la Figura 4.7.

La línea azul representa los datos de la serie diferenciada y la línea roja es el ajuste del modelo a esos datos. El RMSE es de $\approx 32,04$.

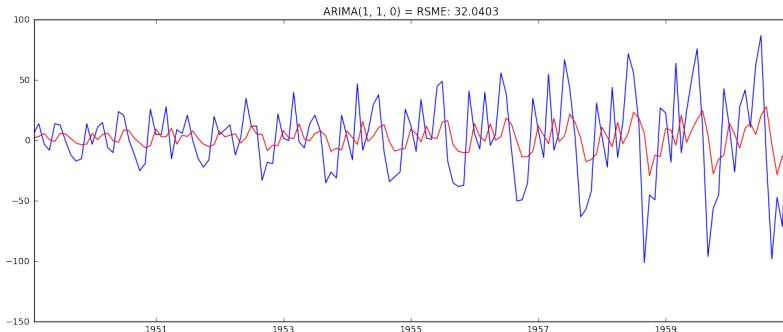


Figura 4.7: ARIMA iterativo: AR(1).

Pasamos a probar la componente de media móvil: $q = 1$, o $MA(1)$. El RMSE ha bajado un poco, ahora es $\approx 31,52$ (Figura 4.8).

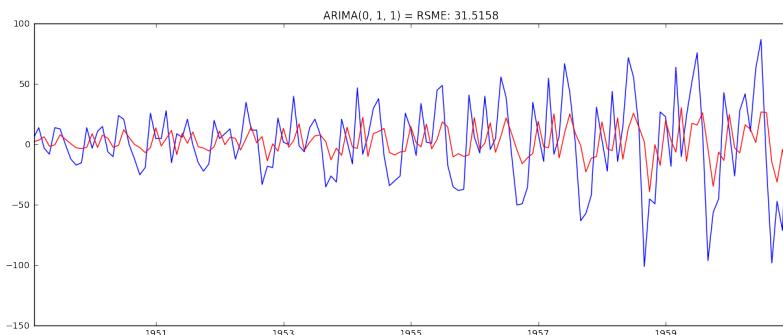


Figura 4.8: ARIMA iterativo: MA(1).

Combinando los dos modelos anteriores es cuando tenemos realmente un modelo ARIMA completo, con las tres componentes: $ARIMA(p, d, q) = ARIMA(1, 1, 1)$. Se puede verificar que el RMSE ha bajado mínimamente respecto al modelo anterior (Figura 4.9).

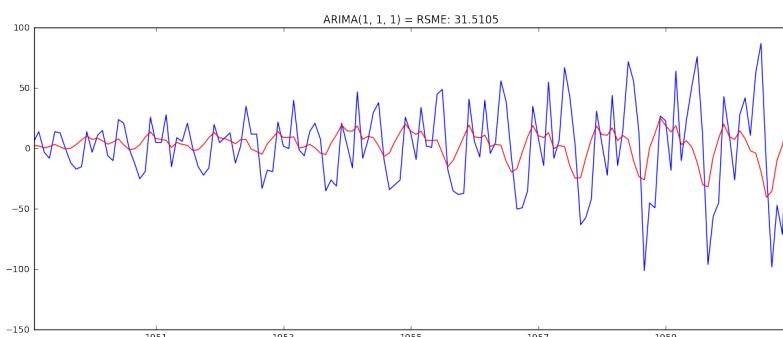


Figura 4.9: ARIMA iterativo: AR(1) y MA(1).

Obviando algunas iteraciones intermedias, pasamos al modelo con dos niveles de AR y de MA : $ARIMA(p, d, q) = ARIMA(2, 1, 2)$. El RMSE obtenido es de $\approx 25,06$, que corresponde al menor valor. Si paramos la iteración, este sería el modelo con mejor ajuste. En la Figura 4.10 podemos verificar ese ajuste visualmente: la línea roja se ajusta mejor a los datos que en los gráficos anteriores.

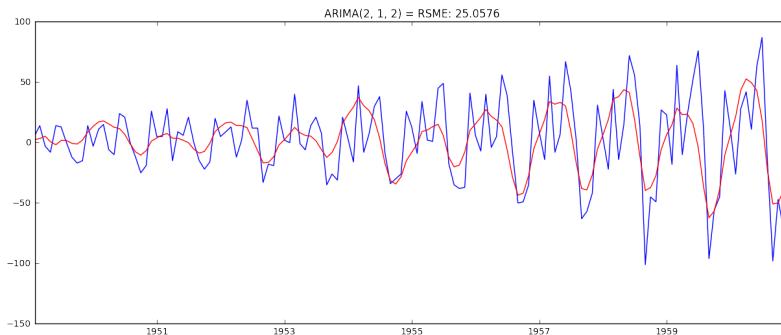


Figura 4.10: ARIMA iterativo: AR(2) y MA(2).

estadísticas, incluyendo modelos de series temporales. Particularmente útil es la funcionalidad de generar gráficas de predicción automáticamente, sin necesidad de volver los datos a la escala original (sin diferenciación), usando `plot_predict`.

Una vez ajustado el modelo a los datos, se pueden hacer predicciones futuras. Se define el periodo de predicción y el modelo aprendido generará esos pronósticos. La Figura 4.11 da un ejemplo de los resultados que se pueden obtener.

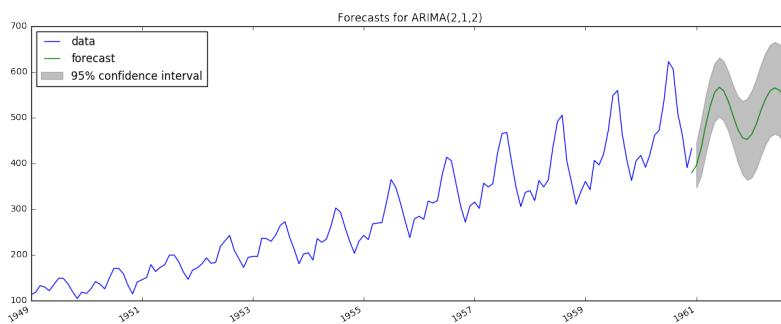


Figura 4.11: Predicciones del modelo ARIMA(2, 1, 2).

La línea azul de la Figura 4.11 representa los datos, ahora sin diferenciación. La línea verde representa la predicción después del final de la serie (llamada predicción *out-of-sample*). El área sombreada representa el intervalo de confianza de 5 % de la predicción. Se puede comprobar visualmente que la predicción sigue aproximadamente la tendencia de la serie. Predicciones con mayor horizonte futuro tendrán mayor incertidumbre, una vez que el modelo hará regresión de los valores futuros con los valores predichos.

El modulo `statsmodels` de Python tiene un conjunto de herramientas estadísticas, incluyendo modelos de series temporales. Particularmente útil es la funcionalidad de generar gráficas de predicción automáticamente, sin necesidad de volver los datos a la escala original (sin diferenciación en este caso), usando `plot_predict`. El código 4.4 es el código que genera el gráfico de la Figura 4.11. El periodo de predicción está aquí definido en los parámetros `start` y `end`, en un total de 20 nuevos puntos desde el final de la serie (`len(ts)`).

```
from statsmodels.tsa.arima_model import ARIMA
model = ARIMA(ts, order=(2, 1, 2))
results = model.fit(disp=-1)
fig, ax = plt.subplots()
plt.plot(ts)
fig = results.plot_predict(start=len(ts)-1, end=len(ts)+20, dynamic
    =True, ax=ax, plot_insample=False)
```

Listing 4.4: `plot_predict` de `statsmodels` de Python.

CAPÍTULO 5

Experimentación y resultados

En este capítulo se exponen las pruebas realizadas y respectivos resultados.

5.1 Simulación con datos controlados

Para dar una visión general e ilustrar la solución propuesta en su totalidad, empezaremos con un ejemplo sencillo. Para esto, se ha creado un *dataset* ficticio y controlado, constituido por las siguientes funciones de senos y cosenos con ruido:

- \sin con ruido 0,05
- $2 \cdot \sin + 3$ con ruido 0,01
- \cos con ruido 0,02
- $4 \cdot \cos - 1$ con ruido 0,09
- $\sin - 1$ con ruido 0,05
- $5 \cdot \cos$ con ruido 0,01
- \cos con ruido 0,02
- $\cos - 1$ con ruido 0,09

El ruido proporciona un toque realista a los datos sintéticos (que podrían ser provenientes de sensores) y nos servirá para comprobar la técnica de PCA. Las observaciones de, por ejemplo, $\sin - 1$ con ruido 0,05 serían las representadas en el gráfico de la Figura 5.1. Se puede ver que la curva sinusoidal no es perfecta por el ruido introducido.

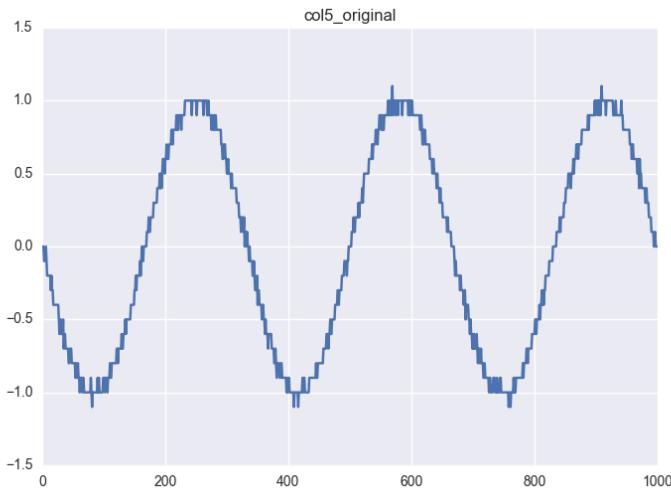


Figura 5.1: $\sin - 1$ con ruido 0,05.

Para comprobar la viabilidad de PCA, se procede al cálculo de un determinado número de componentes - en el ejemplo empezamos con 5 componentes. Posteriormente se procede a la inmediata inversión (sin generar datos nuevos) al espacio original. Usando el mismo ejemplo de $\sin - 1$ con ruido 0,05, el gráfico 5.2 muestra la re-proyección de la misma función sinusoidal al espacio original, después de haber sido proyectada en el espacio reducido de 5 componentes. Se demuestra que no solo la inversión se hace correctamente como también se reduce el ruido de la señal.

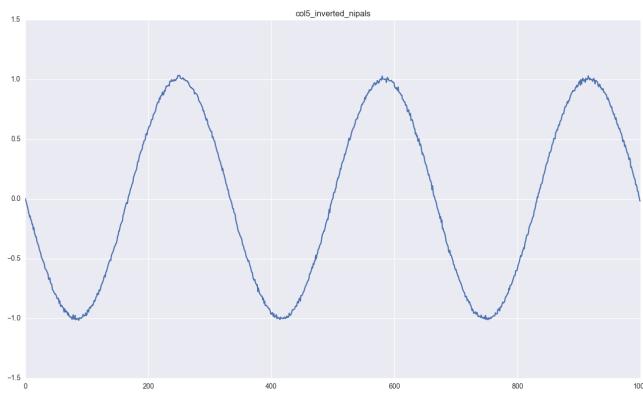


Figura 5.2: Inversión de $\sin - 1$ con ruido 0,05, usando 5 componentes.

Veamos ahora como cambia la señal invertida conforme se van introduciendo las componentes de forma ordenada, para el mismo ejemplo de función sinusoidal. La Figura 5.3 representa el cambio secuencialmente.



Figura 5.3: Inversión de $\sin - 1$ con ruido 0,05 según el numero de componentes.

En la secuencia podemos verificar que, según se van añadiendo componentes, la señal va ganando forma y ruido más parecidos a la señal real. Con pocas componentes, la señal es menos parecida a la real, pero mantiene la forma, media y los límites aceptablemente. Cuando el numero de componentes es más cercano al máximo (gráficos 5.3g y 5.3h), el modelo PCA es capaz de explicar incluso el ruido, siendo la inversión muy similar a la señal original.

Estando sustentada la viabilidad de PCA sobre los datos controlados, la idea es ahora producir datos simulados con distribución Gaussiana sobre las proyecciones y comprobar que la población invertida sigue manteniendo la similitud con la población original. A través de la media y desviación típica de las proyecciones de cada componente, se producen datos aleatorios con distribución Gaussiana. En el gráfico ?? podemos ver, en verde, los valores de proyección de la primera componente; y, en azul, datos simulados con distribución normal para esa componente. Además, en gris se representan los límites mínimo y maximo, media y algunas desviaciones típicas de los valores de la Gaussiana.

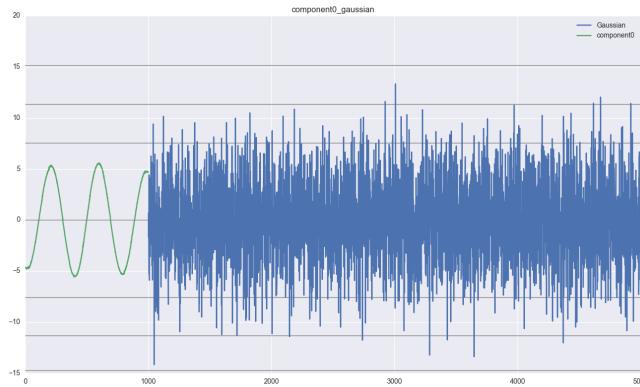


Figura 5.4: Simulación de la primera componente, usando distribución Gaussiana.

Al re-proyectar los datos sintéticos generados con distribución normal al espacio original, obtenemos la simulación representada en el gráfico 5.5. La línea verde representa los datos originales de $\sin - 1$ con ruido 0,05 y la línea azul es la simulación en si.

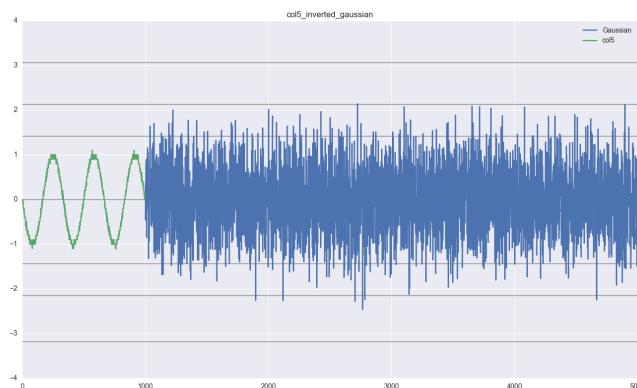


Figura 5.5: Inversión de $\sin - 1$ con ruido 0,05, usando distribución Gaussiana.

El test T^2 de Hotelling corrobora que la población generada es similar a la original, como indica el código R 5.1. El estadístico asume que las poblaciones son similares (hipótesis nula) - en este caso su valor es poco significativo, así que podemos rechazar su hipótesis nula y afirmar que las poblaciones son similares. Además, el p-valor (P-value) indica un grado de confianza alto en esa evaluación.

```
> res <- hotelling.test(x = data, y = inverted_data)
> print(res)
Test stat: 0.0024086
Numerator df: 8
Denominator df: 100991
P-value: 1
```

Listing 5.1: Test T^2 de Hotelling en R.

De esta forma, queda demostrada la validez de la simulación con distribución Gaussiana sobre las proyecciones en las componentes principales.

Llegados a este punto, pasamos a dar un sentido temporal a la predicción. Para esto, se usarán modelos ARIMA, que aprenderán la evolución de los datos de las proyeccio-

nes a lo largo del tiempo. Una vez hecho el aprendizaje, el modelo ARIMA puede hacer predicciones futuras sobre los datos, basándose en los datos pasados. Esta predicción se hará de forma iterativa, punto por punto: en cada iteración, teniendo la predicción ARIMA de un punto nuevo para cada componente, se añade ruido residual y se busca el punto multi-dimensional más cercano en los datos simulados con distribución Gaussiana. El punto más cercano se usará para substituir los puntos predichos por ARIMA y así re-alimentar el modelo con nuevos datos. Los datos simulados serán un sub-conjunto de los datos simulados Gaussianos, que demostramos que eran válidos, por lo que la población generada por inversión al espacio original debe ser válida también.

La idea es ahora ilustrar el proceso completo: cálculo de componentes principales, simulación de datos con distribución Gaussiana y predicción con ARIMA. Para la primera componente, se obtienen los resultados representados en la Figura 5.6. En el gráfico se pueden ver todas las partes del proceso, con superposición para comparar más fácilmente:

- rojo - datos de la componente.
- azul - simulación con distribución normal.
- verde - predicción.

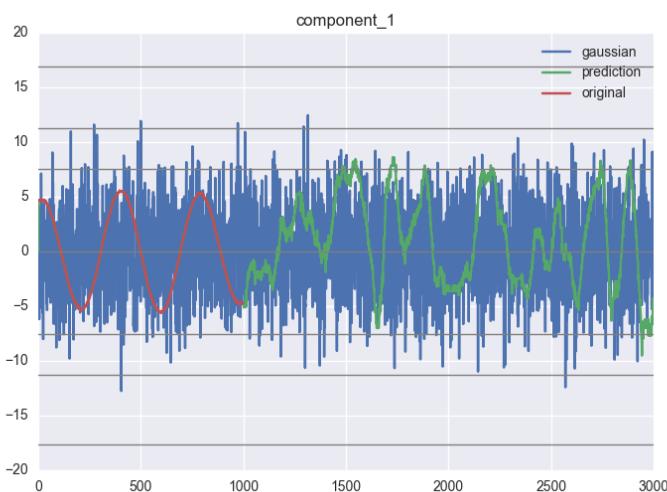


Figura 5.6: Simulación de la componente 1 con datos controlados.

Los gráficos de simulación de las demás componentes se representan en la Figura 5.7. Con datos controlados, el modelo ARIMA capta bien la serie y eso se traduce en una predicción estable y coherente. Además, siempre dentro de los límites de la simulación Gaussiana, como se pretendía. Las restantes componentes tienen un comportamiento idéntico.

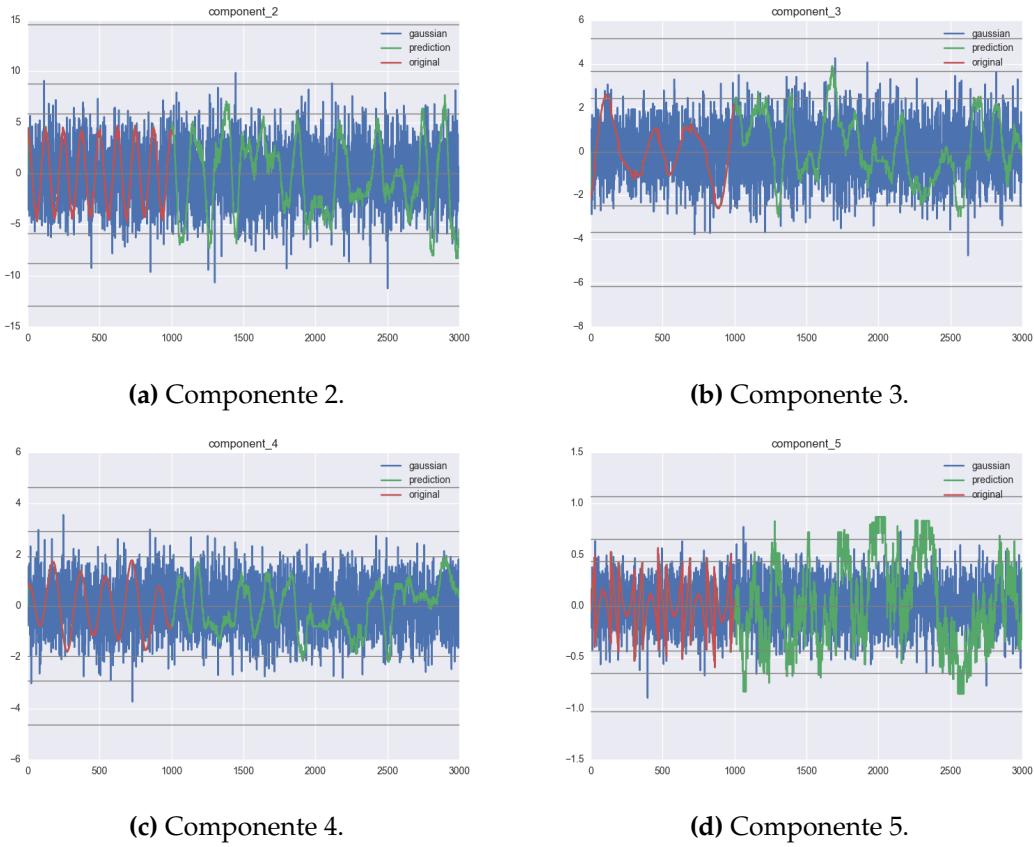


Figura 5.7: Simulación de las componentes 2, 3, 4 y 5 con datos controlados.

Al invertir la simulación al espacio multi-dimensional, se obtienen datos también coherentes para las características originales. El ejemplo de inversión de la serie de $\sin - 1$ con ruido 0,05 sería el representado en la Figura 5.8. La señal invertida conserva aceptablemente la forma de la función sinusoidal original a lo largo del tiempo. Además se mantiene dentro de los límites de la inversión de la respectiva simulación Gaussiana para la misma característica (Figura 5.8b). De todas formas, la calidad de la simulación es configurable, a través del ajuste de numero de componentes, como quedó claro en el estudio de la Figura 5.3.

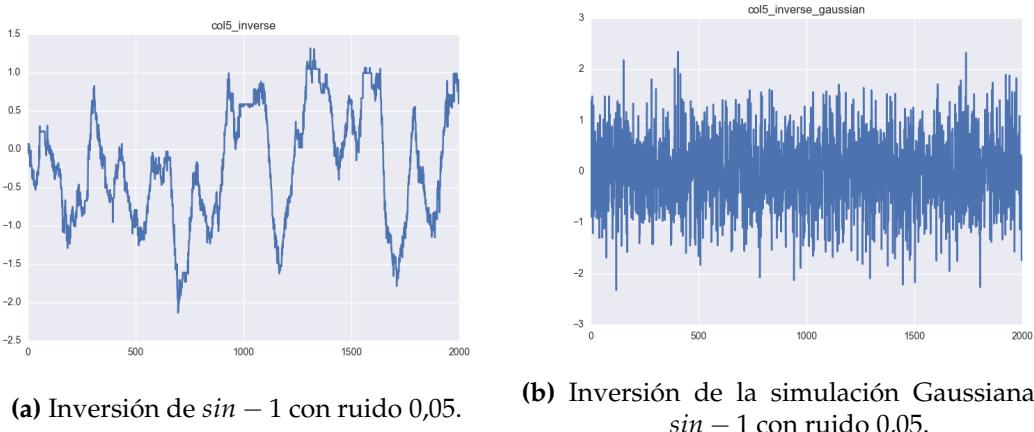


Figura 5.8: Inversión de $\sin - 1$ con ruido 0,05 con búsqueda en la simulación Gaussiana.

Se verifica que, con datos controlados, el proceso completo con PCA y ARIMA tiene resultados coherentes. La calidad de los datos es un factor muy importante para el análisis. Las señales originales de los sensores contienen muchas veces datos anómalos, como *outliers*, que afectan en gran medida el resultado de la aplicación de técnicas de *Machine Learning*. Lo interesante en este estudio es que aporta algunos ajustes que permiten minimizar esos efectos, por ejemplo, cambiar el numero de componentes, aumentar el numero de puntos de la simulación Gaussiana o introducir factores de atenuación de la predicción.

5.2 Dataset

El dataset proporcionado es de reducida dimensión, tiene tan solo 5MB, por lo que la tarea de ingestión no es intensiva. Sin embargo, hay que tratar los datos en bruto antes de empezar a usarlos. El fichero de datos es de texto, así que hay que hacer determinadas conversiones para poder usar tipos más específicos, como sean fechas y números. El primer problema tiene que ver con los formatos de fecha, que no son correctos para el *locale* España en *Windows*, lo que exige una adaptación de los algoritmos de *parsing*, como se describe en el apartado 'Transformación'.

5.2.1. Estructura

Un breve análisis del *dataset* en un editor de texto muestra que tiene un formato de campos separados por espacios y tabulaciones. La cantidad de espacios es variable:

```
Tiempoinicio APHu APVs ...
...
06-oct-2015 21:57:03 44.6 69.3 ...
06-oct-2015 21:57:12 45.1 69.0 ...
06-oct-2015 21:57:21 44.8 69.8 ...
...

```

Listing 5.2: Ejemplo del *dataset*.

La primera línea contiene un *header* (cabecera), con 15 nombres:

```
Tiempoinicio APHu APVs ACPv ZSx ZUs H7x H1x H2x H6x H3x H4x H5x ACPx Svo
```

Listing 5.3: *Header* del *dataset*.

Se puede verificar también que hay una línea vacía después del *header*. El primer campo tiene un formato de fecha/hora y los demás campos tienen formato decimal.

5.2.2. Transformación

Para la ingestión y transformación de los datos se han usado librerías muy útiles y con muchas funcionalidades que facilitan bastante esas tareas: en los scripts Python se ha usado el paquete *Pandas* y en R la función *read.csv2* del paquete *utils*.

Las 14 variables decimales no ofrecen problemas en la ingestión del *dataset*. Para asegurar el formato decimal, es conveniente definir el separador decimal como punto ('.'). Con eso es suficiente para un em parsing correcto.

Las fechas son más complejas de procesar. El formato de mes da indicios de estar escrito en castellano: oct, dic, mar, abr, may, jun. Sin embargo, en *Windows*, el *locale* España usa un punto ('.') como *standard* en la abreviación de mes, por ejemplo 'oct.'. Así que el *parsing* de fechas tiene que ser ajustado si el sistema operativo es *Windows*. La estrategia ha sido el uso de una expresión regular para añadir el punto ('.') necesario en las abreviaciones de meses, como se puede ver en los siguientes *snippets* Python y R:

```
def parse_date(date_string):
    locale_date_string = re.sub("(+-)(.+)(-+)", "\\\1\\\2.\\\3", date_string)
    return datetime.strptime(locale_date_string, "%d-%b-%Y %H:%M:%S")
```

Listing 5.4: *Parsing* de fechas en Python.

```
data$Tiempoinicio <- sub("(\\d+)(\\w+)(-\\d+\\s\\d+:\\d+:\\d+)", "\\\1\\\2.\\\3",
  data$Tiempoinicio)
data$Tiempoinicio <- as.POSIXct(data$Tiempoinicio, format="%d-%b-%Y %H:%M:%S")
```

Listing 5.5: *Parsing* de fechas en R.

En este caso particular de las fechas, el problema no está en el *dataset* realmente. Los *standards* de fechas varian en cada sistema operativo. La idea de exponer este caso no es más que transmitir la necesidad de ajustar y normalizar los datos cuando provienen de sensores.

El resultado final es un *dataset* estructurado con una marca temporal como índice y 14 características: *APHu*, *APVs*, *ACPv*, *ZSx*, *ZUs*, *H7x*, *H1x*, *H2x*, *H6x*, *H3x*, *H4x*, *H5x*, *ACPx* y *Svo*.

5.2.3. Distribución de los datos

Las mediciones de los sensores se distribuyen por varios días, no siempre consecutivos. Los días con mediciones son algunos

- 6-9, 12 de octubre de 2015
- 14 de diciembre de 2015
- 7, 8, 11, 14 de marzo de 2016
- 6, 29 de abril de 2016
- 2, 26 de mayo de 2016
- 8, 17, 20 de junio de 2016

En cada día las muestras son dispares y en momentos distintos del día, probablemente por corresponder a distintas pruebas. Así, también las frecuencias de muestreo son variables - en algunos días parece indicar una frecuencia de 10 segundos pero hay otros donde son de 1 minuto.

5.3 PCA

A continuación se presentan las pruebas realizadas para el cálculo de las componentes principales (PCA) del *dataset* y respectiva inversión al espacio original. Los primeros

análisis son informales, sobre todo con recurso a gráficos. Luego se formalizan estos resultados por medio de un test estadístico. El capítulo termina con una propuesta de simulación de datos, que, respaldada por el test estadístico, servirá de base para las fases siguientes.

5.3.1. PCA iterativo - NIPALS

Para *datasets* grandes, el algoritmo iterativo NIPALS ofrece mejor rendimiento, al determinar solamente el numero de componentes definido. SVD calcularía la matriz de varianzas-covarianzas para todos los componentes, lo que exigiría mucha memoria y CPU. Aunque en este caso el *dataset* es pequeño, usaremos el algoritmo iterativo NIPALS.

Empezamos por analizar visualmente los datos de sensores para un día determinado. Observemos el comportamiento de la variable *APVs* para ese día, representado en la Figura 5.9.

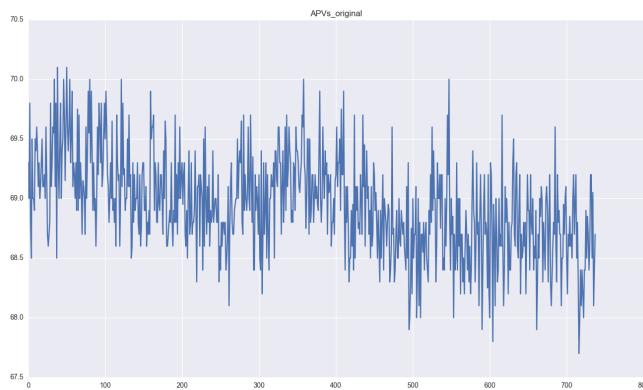


Figura 5.9: Datos originales de la variable *APVs*.

Aplicamos el algoritmo iterativo NIPALS para calcular las 5 componentes principales, proyectando de esta forma el espacio original de 14 dimensiones (características) en un espacio más reducido de solamente 5 dimensiones. Al invertir este espacio de vuelta al espacio original se obtienen datos muy similares a los originales, como se puede comprobar informalmente en la Figura 5.10a, que representa los datos invertidos para la variable *APVs*.

Obviamente se podría usar la descomposición SVD para el cálculo de las componentes principales. Usando SVD de numpy se obtienen los mismos resultados, que se pueden analizar informalmente en la Figura 5.10b.

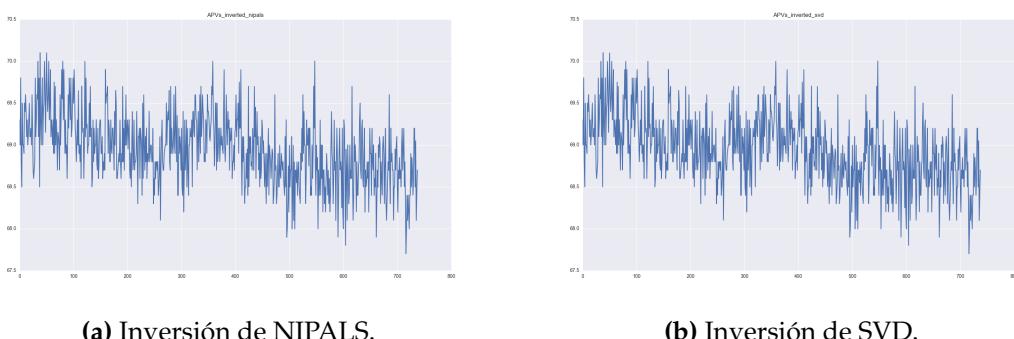


Figura 5.10: Datos invertidos de la variable *APVs*.

Para algunas de las características la eficiencia de la inversión no es tan evidente. Por ejemplo, la serie de $H6x$ empieza con una secuencia de valores alrededor de 39,45 y luego baja para valores alrededor de 39,30, como representado en la Figura 5.11.

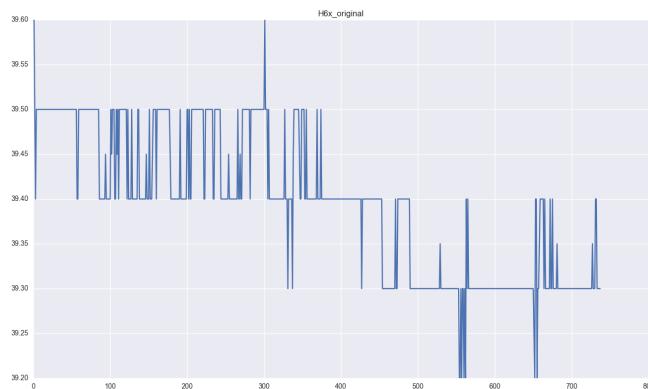
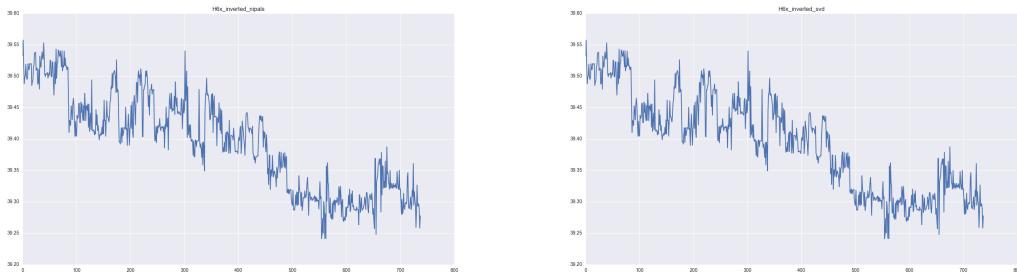


Figura 5.11: Datos originales de la variable $H6x$.

La inversión, sea con NIPALS (Figura 5.12a) o con SVD (Figura 5.12b), aunque no tan evidente, también demuestra esa tendencia. Además se puede verificar que los datos invertidos se mantienen dentro del rango de los límites de los valores originales (39,20 – 39,60).



(a) Inversión de NIPALS.

(b) Inversión de SVD.

Figura 5.12: Datos invertidos de la variable $H6x$.

Otras características son todavía menos evidentes. Con 5 componentes, los resultados de la inversión de $ACPx$ son muy poco claros. En un análisis visual, da incluso la sensación de que no corresponde a la misma característica. Véase el gráfico de los datos originales en la Figura 5.13 y respectiva re-proyección al espacio original en la Figura 5.14.

Aunque parecen datos diferentes, hay algunas propiedades que sí son coherentes, como los límites mínimo y máximo. Los datos originales oscilan principalmente dentro del rango 3,25 – 3,40 y los datos invertidos también. Además, los datos invertidos no contemplan los dos puntos originales que parecen *outliers*, con valores de $\approx 3,14$, probablemente ruido.

En este caso, es importante destacar la elección del numero de componentes. Esa decisión se formalizará más adelante, con el test estadístico. De momento, podemos concluir que, con 5 componentes, los datos invertidos de $ACPx$ que vemos en la Figura 5.14a son probablemente la información relevante de la señal. Al final la variabilidad que modelan las últimas componentes es el ruido y quizás en $ACPx$ hay más ruido que información importante.

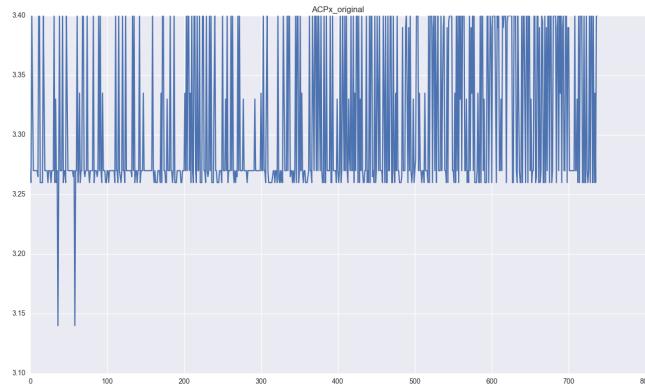
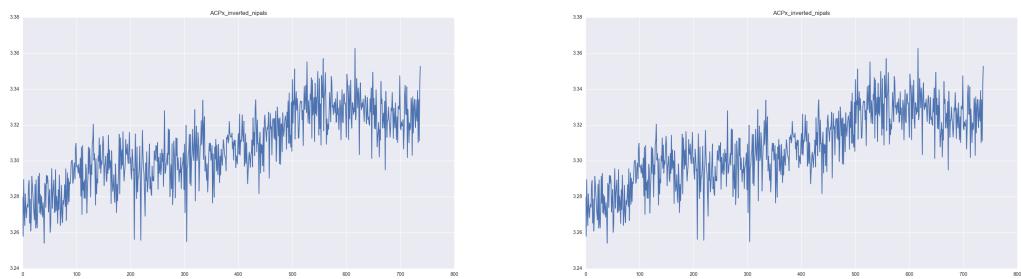


Figura 5.13: Datos originales de la variable $ACPx$.



(a) Inversión de NIPALS.

(b) Inversión de SVD.

Figura 5.14: Datos invertidos de la variable $ACPx$.

Si se calculan más componentes, los resultados se aproximarán más a la realidad, pues se añade a los datos el posible ruido de la señal, que en principio estará contenido en las últimas componentes de la proyección. Sin embargo, aunque el ajuste sea mejor, se pierde generalización del modelo y también la eliminación de ruido. Los resultados de la proyección con 14 componentes se pueden comprobar en la Figura 5.15.

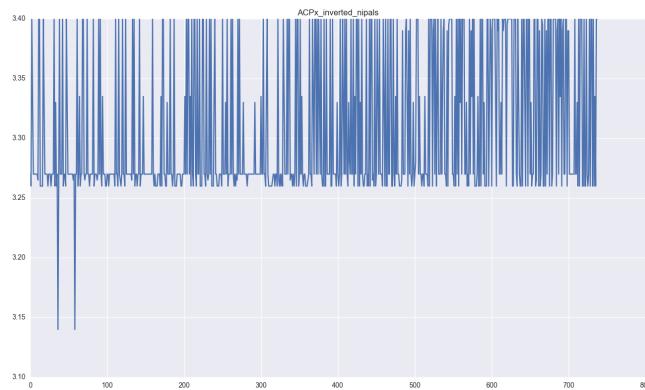


Figura 5.15: Datos invertidos de la variable $ACPx$ con 14 componentes.

5.3.2. Test de hipótesis - Estadístico T-cuadrado de Hotelling

Para validar que la inversión de las matrices se está haciendo correctamente necesitamos una medida objetiva que permita comparar estadísticamente las poblaciones original y invertida. Para ello usaremos el estadístico multidimensional T^2 (T-cuadrado) de Hotelling.

El test T^2 de Hotelling es una generalización del test t de Student. Para dos muestras (poblaciones), el test t evalúa las diferencias en las medias de esas poblaciones. El test T^2 se usa cuando las variables son dos o más.

Las suposiciones del test T^2 de Hotelling para dos poblaciones son:

- cada población sigue una distribución multivariante normal.
- las dos poblaciones son independientes.
- las dos matrices de varianzas-covarianzas son iguales.

La hipótesis nula es que las medias para todas las variables son iguales.

Respecto a la implementación, se han usado los algoritmos de test Hotelling T^2 R, con la librería Hotelling. El código 5.6 es un ejemplo de uso de la librería.

```
> res = hotelling.test(x = data, y = inverted_data)
> print(res)
Test stat: 0.072321
Numerator df: 14
Denominator df: 119782
P-value: 1
```

Listing 5.6: Test T^2 de Hotelling en R.

Si el valor del estadístico es significativo (valor alto) significa que ha encontrado bastantes diferencias entre las poblaciones, por lo que podemos rechazar la hipótesis nula. En el caso demostrado en el Código 5.6, el resultado del estadístico es poco significativo (cercano a cero), así que podemos aceptar la hipótesis nula. La interpretación es que las poblaciones data y inverted_data son iguales, con un grado de confianza alta ($p\text{-value}=1$).

Se ha aplicado el test T^2 de Hotelling exhaustivamente para los varios números de componentes posibles. Los resultados se encuentran en la tabla 5.1.

Se ha usado un pequeño factor aleatorio con distribución normal para ocupar valores vacíos en determinadas variables. Por esa razón los resultados pueden variar y ser algo incoherentes (por ejemplo, con 9 componentes tenemos mayor valor del estadístico que con 1 componente). De todas formas, el objetivo de este análisis exhaustivo es verificar que la inversión de NIPALS es correcta - podemos afirmar que sí, pues el estadístico presenta valores poco significativos y los p-valor son altos.

Aunque los p-valor sean altos (cercanos a 1), se detecta una tendencia para ser inferior a 1 a partir de un determinado numero de componentes (en la tabla 5.1 sería a partir de 9 componentes que el p-valor empieza a dar señales de bajada). Probablemente, esa variación se debe a la introducción del ruido a partir de ese numero de componentes. En base a este estudio, se ha decidido usar 5 componentes para el cálculo de PCA.

Componentes	Hotelling T^2	p-valor
1	0.020809	1
2	0.063641	1
3	0.012115	1
4	0.013919	1
5	0.020096	1
6	0.077881	1
7	0.031675	1
8	0.00012993	1
9	0.20037	0.9994
10	0.029146	1
11	0.16149	0.9998
12	0.08343	1
13	0.0024538	1
14	0.13956	0.9999

Tabla 5.1: Test T^2 de Hotelling para los varios números de componentes.

5.3.3. Simulación con distribución Gaussiana

El primer acercamiento de simulación de datos se hizo usando la distribución normal o Gaussiana. El algoritmo es: para cada componente, se calcula la media y variación típica y, a partir de estos, se genera una nueva serie con distribución gaussiana (ver código 5.8).

```
mus = np.mean(nipals_T, axis=0)
sigmas = np.std(nipals_T, axis=0)

generated_gaussian = np.zeros((GAUSSIAN_SIZE, N_COMPONENTS))
for i in range(N_COMPONENTS):
    # calculate normal distribution by component and store it in column i
    generated_gaussian[:, i] = np.random.normal(mus[i], sigmas[i], GAUSSIAN_SIZE)
```

Listing 5.7: Simulación con distribución Gaussiana.

La Figura 5.16 da una idea gráfica de la simulación con distribución Gaussiana para la componente 1. En verde vemos los datos de la componente y en azul los datos generados. Las líneas gris representan la media, mínimo, máximo y algunas desviaciones típicas. Los datos generados se mantienen dentro de los límites de la serie de la componente.

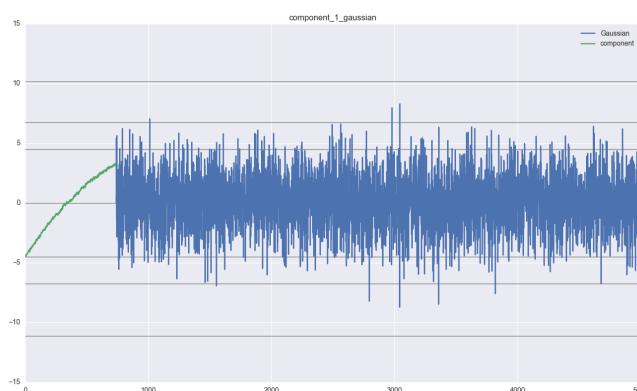


Figura 5.16: Datos originales y simulados de la componente 1.

Las simulaciones para las demás componentes se representan en la secuencia de gráficos de la Figura 5.17.

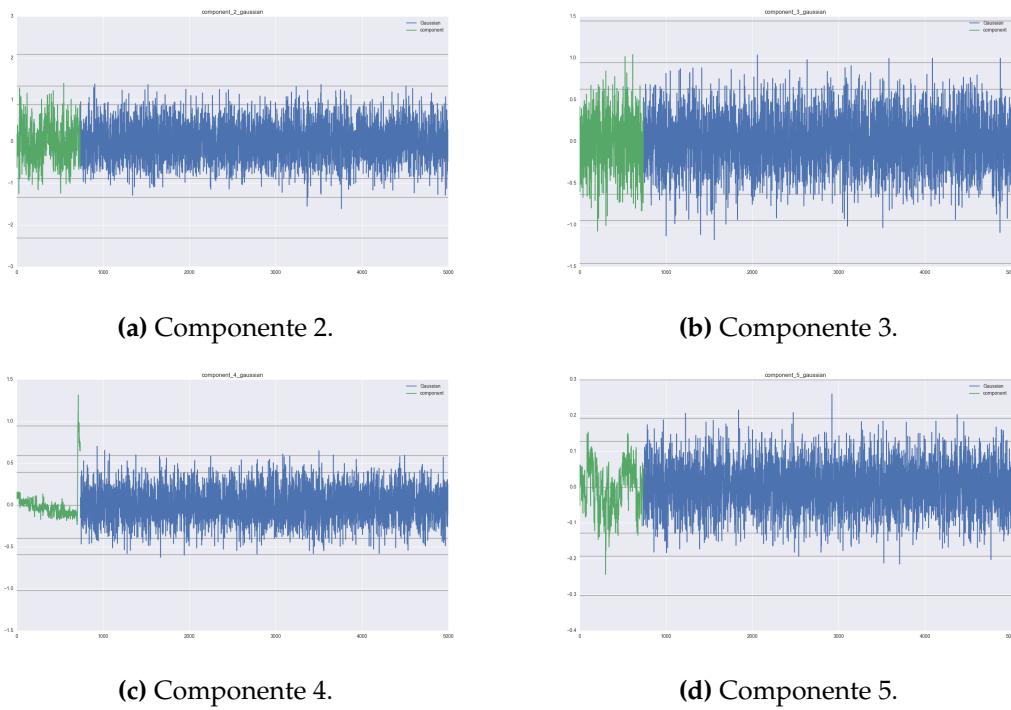


Figura 5.17: Datos originales y simulados de las componentes.

Una vez generados los nuevos datos de las componentes, se re-proyectan al espacio original y se obtiene la nueva población multivariante, simulada. La Figura 5.18 da un ejemplo de la re-proyección usando la simulación Gaussiana, para la variable *APHu*.

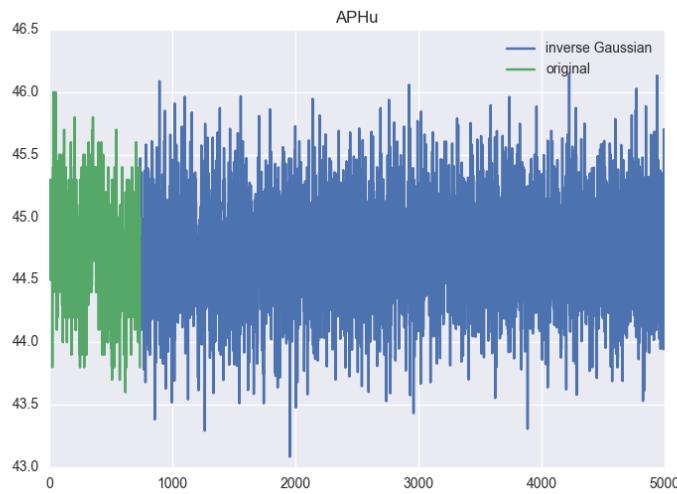


Figura 5.18: Datos originales y simulados de la variable *APHu*.

Sometiendo los nuevos datos al test T^2 de Hotelling, se verifica que es igual a la población original (código 5.8). El estadístico tiene un valor poco significativo de 0,037391 y el p-valor es 1, así que la hipótesis nula (las poblaciones son diferentes) puede ser rechazada.

```
> res <- hotelling.test(x = data, y = inverted_data_gaussian)
> print(res)
Test stat: 0.037391
Numerator df: 14
Denominator df: 100723
P-value: 1
```

Listing 5.8: Test T^2 de Hotelling para la simulación con distribución Gaussiana.

Para validar el correcto funcionamiento del test implementado en `hotelling.test`, se ha generado una serie con valores aleatorios con distribución no normal. Con esto, es de esperar que el estadístico adopte valores significativos y que el p-valor se aleje de 1. El código 5.9 corrobora esa expectativa.

```
> res <- hotelling.test(x = data, y = inverted_data_non_normal)
> print(res)
Test stat: 719.81
Numerator df: 14
Denominator df: 100723
P-value: 0
```

Listing 5.9: Test T^2 de Hotelling para la simulación con distribución no normal.

La simulación con distribución Gaussiana constituye la fundación de nuestro estudio para la simulación de datos. Asumimos, a partir de ahora, que las componentes simuladas con distribución Gaussiana, al ser re-proyectados al espacio original, producen una nueva población que es similar a la original. Esta base sustenta algunas decisiones de los próximos capítulos y, por consiguiente, los resultados finales del estudio.

5.4 ARIMA

Una vez determinado el numero de componentes a usar y realizadas las primeras simulaciones con distribución normal, el siguiente paso consiste en construir series temporales sobre cada una de las componentes para simular con cierto sentido temporal. El objetivo es que los datos generados tengan forma de serie temporal. Para poder aplicar los modelos ARIMA introducidos anteriormente hay que preparar y estudiar los datos previamente. Una vez preparada la serie, se pasa a la predicción del comportamiento futuro, la simulación propiamente dicha.

5.4.1. Preparación de la serie temporal

Las mediciones de los sensores son dadas a periodos temporales que no siempre son regulares. Los datos de los sensores tienen muchos huecos, es decir, períodos largos sin mediciones. Además la frecuencia de mediciones no es constante, a veces es cada 10 segundos y otras cada minuto. La primera consideración para el análisis de series temporales es que éstas deben ser un conjunto de datos observados a periodos fijos.

La preparación consiste en transformar la serie para que sea homogénea en los períodos de observaciones. Se verifica también que las mediciones son muy discrepantes en días diferentes, potencialmente por corresponder a diferentes pruebas en las máquinas industriales. Para simular un proceso productivo continuo, no tiene sentido juntar los varios días si hay demasiada discrepancia entre ellos; unirlos supondría hacer asociaciones potencialmente muy fuertes, que podrían deteriorar bastante los modelos de series. Así

que es conveniente elegir los datos de un día determinado y, sobre estos, aplicar agrupación y interpolación para un periodo determinado.

Nota: Esta transformación se hace antes incluso de aplicar PCA, para poder comparar las poblaciones original y simulada en la misma escala.

La librería pandas ofrece muchas herramientas para análisis de estructuras de datos en Python. Está construida sobre numpy, lo que le garantiza mucha eficiencia. En particular, nos interesan las facilidades en el tratamiento de series temporales, como la indexación por fechas, ocupación de huecos vacíos, cambio de frecuencia o *resampling* (re-muestreo).

En nuestro caso procederemos a cambiar la frecuencia de muestreo a periodos fijos e respectiva interpolación de los nuevos espacios con la media. Para el análisis, se ha elegido el día 2015-10-06 (cuya frecuencia de muestreo es de ≈ 10 segundos), al que se aplicará un periodo exacto de 10 segundos. El código 5.10 demuestra la aplicación de la transformación con estructuras de datos pandas.

```
# Specify a date to analyze the timeseries
date = "2015-10-06"
data = data[date]

# Resampling and Interpolation
data = data.resample(TS_FREQUENCY).mean().interpolate()
```

Listing 5.10: *Resampling* y interpolación con pandas.

5.4.2. Estacionaridad de la serie temporal

El análisis de estacionaridad de la serie es un ejercicio que se hace iterativamente. En cada iteración, si la serie no es estacionaria todavía, se aplican técnicas para conseguirlo y se vuelve a analizar. La medida objetiva de estacionaridad de la serie es el test de Dickey-Fuller, que se calcula con el módulo Python statsmodels. Como se va a generar un modelo ARIMA para cada componente, este análisis se hará para cada componente.

No se ha podido aplicar logaritmos para estabilizar, una vez que hay valores negativos. Una de las formas más usuales para conseguir estacionaridad de una serie es aplicando diferenciación (*differencing*). Se aplica esa técnica para las componentes que no son estacionarias, otra vez usando facilidades de pandas (ver código 5.11).

```
timeseries_sample_diff = timeseries_sample.sub(timeseries_sample.shift())
```

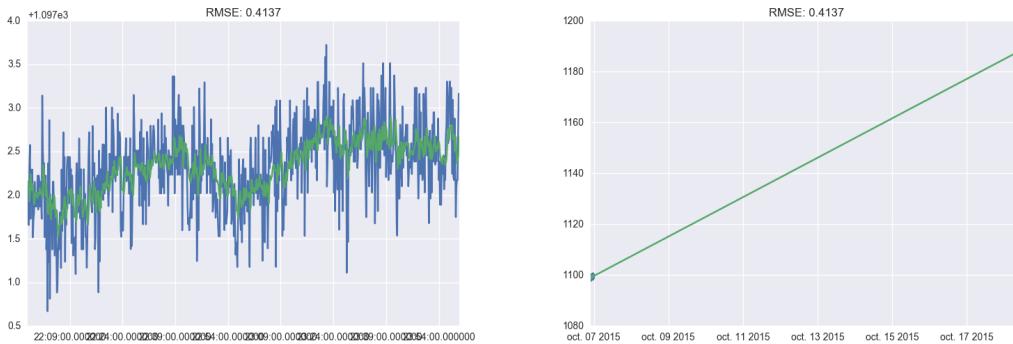
Listing 5.11: Diferenciación de la serie usando pandas.

Así se obtiene el valor de la componente d de $ARIMA(p, d, q)$. Para el cálculo de las componentes AR (p) y MA (q) de ARIMA se usa un algoritmo que calcula exhaustivamente todas las combinaciones y respectivo *root mean squared error* (RMSE). El modelo con menor RMSE es el elegido para esa componente.

5.4.3. Predicción ARIMA con PCA de todo el dataset

En una primera instancia, y en fases siguientes entendido como erróneo, se ha aplicado el cálculo de PCA a todo el *dataset* mientras que la serie temporal se construía sobre un solo día. Si la serie la vamos a aprender sobre un día es conveniente que la proyección sea también sobre ese día. Sin embargo, parece relevante enseñar las primeras predicciones.

ciones ARIMA para el día 2015-10-06. La Figura 5.19 representa los gráficos de ajuste y predicción para la componente 1.



(a) Ajuste ARIMA para la componente 1. (b) Predicción ARIMA para la componente 1.

Figura 5.19: ARIMA para la componente 1 calculada con PCA sobre todo el *dataset*.

En el gráfico de predicción *in-sample* (Figura 5.19a), podemos ver que el modelo se ajusta bastante bien a la serie de la componente. En el gráfico de predicción *out-of-sample* ((Figura 5.19b)), vemos que los valores siguen una tendencia creciente, que seguirá hasta infinito.

5.4.4. Predicción ARIMA con PCA de un día

El cálculo de PCA debe hacerse sobre los datos del día de la serie y no sobre todo el *dataset* como visto en la sección anterior. Con esta corrección, los gráficos de predicción ya no se deterioran y tienden para la media de los datos. La Figura 5.20 da cuenta de ese efecto - en azul vemos los datos de la componente y en verde la predicción *out-of-sample*.

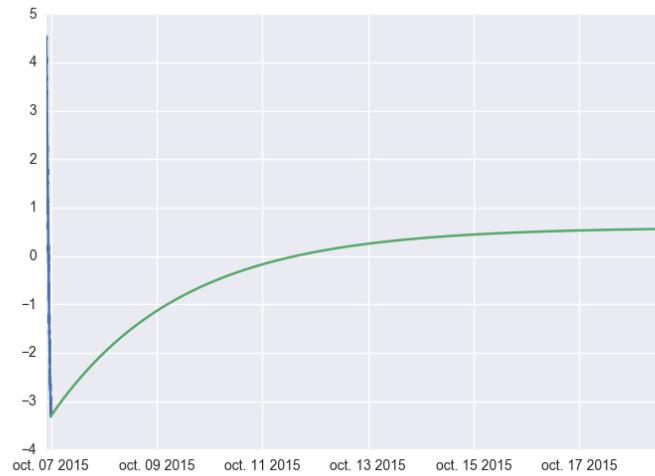


Figura 5.20: Predicción ARIMA *out-of-sample* para la componente 1 calculada con PCA de un día.

Los resultados del test T^2 de Hotelling han mejorado con la corrección, asumiendo ahora el estadístico valores más bajos que anteriormente, pero todavía no se puede aceptar la hipótesis de similitud de poblaciones. En la sección siguiente se prueba un nuevo acercamiento al problema.

5.4.5. Simulación con búsqueda de puntos cercanos de la Gaussiana

Teniendo en cuenta que la población generada de la simulación ARIMA no pasa el test T^2 de Hotelling, se propone una nueva solución, que se explica a continuación. La solución consiste en generar una simulación de millones de puntos con distribución Gaussiana, que se ha demostrado que es válida. Luego se obtiene un punto simulado con el modelo ARIMA, al cual se añade o no ruido residual, y se busca el punto más cercano a este en los datos Gaussianos. El punto más cercano se usará como representante y se usará para re-alimentar el modelo ARIMA. Luego se procede a generar un nuevo punto en el instante siguiente.

El pseudo-código siguiente explica más secuencialmente la predicción y búsqueda de puntos:

1. Predicir un valor nuevo para cada componente c con su respectivo modelo ARIMA. Con esto, para N componentes, generamos el punto N -dimensional $p = \{c_1, \dots, c_N\}$. Digamos que tenemos $N = 5$ componentes, entonces el punto sería $p = \{c_1, c_2, c_3, c_4, c_5\}$, donde c_x es el valor predicho de cada componente.
2. (Opcional) Añadir ruido.
3. Buscar el punto $p' = \{c'_1, c'_2, c'_3, c'_4, c'_5\}$ más cercano en los puntos de la simulación Gaussiana.
4. Substituir el punto predicho originalmente p por p' , en el modelo ARIMA. Con esto, cada c_x es substituído por el c'_x en los datos del modelo ARIMA de cada componente x .
5. Volver al punto 1.

En cuanto a distancias se puede usar la Euclídea o la de Mahalanobis. La distancia de Mahalanobis es más adecuada si se usan los valores naturales de la serie, pues considera la estructura de correlación existente en las variables implicadas. Como los de la simulación Gaussiana son sobre las componentes, la implementación actual usa la distancia Euclídea. Esta es, desde luego, la tarea más exigente del punto de vista computacional - además su exigencia es función del numero de puntos Gaussianos simulados.

La re-alimentación de los modelos con los puntos provenientes de la simulación Gaussiana ha constituido un desafío. La primera implementación se hizo con ARIMA del módulo `statsmodels.tsa.arima_model`, que no permitía la introducción de nuevos datos al modelo aprendido de forma sencilla. Esto obligaba a crear y entrenar un modelo nuevo por cada iteración y para cada componente, que se traducía en un coste de tiempo de ejecución. Para dar un ejemplo, en un ordenador con CPU i5 de 4 núcleos y 4GB de memoria RAM, cada iteración tardaba 0,9 segundos - y esto sin la búsqueda en los puntos Gaussianos todavía.

Siguiendo la recomendación de los desarrolladores de `statsmodels`, se ha adoptado SARIMAX, del módulo `statsmodels.tsa.statespace.sarimax`. Este modelo está basado en ARIMA pero es más completo y mejor mantenido y además permite modelar la parte estacional de la serie e introducir variables exógenas. Con SARIMAX evitamos el paso de volver a entrenar el modelo completo, que mejora bastante el rendimiento de cada iteración.

Antes de pasar a la implementación con búsqueda en los puntos Gaussianos, se presentan las pruebas de predicción con la estrategia de re-alimentación del modelo SARIMAX. La predicción sin ruido tiende rápidamente para valores constantes como se puede verificar en la Figura 5.21a. El gráfico de la predicción con ruido (5.21b) muestra una tendencia

positiva para la primera componente, al contrario de las tendencias que se han visto hasta ahora para esa misma componente. No hay nada de errado, simplemente son pruebas distintas donde el cálculo de PCA tiene valores inversos, que no afecta los resultados. Se verifica que, con ruido, la simulación es más realista y parece que tiene una estabilidad creciente con una ligera curvatura. Lo que no parece correcto son los picos y cambios de tendencia.

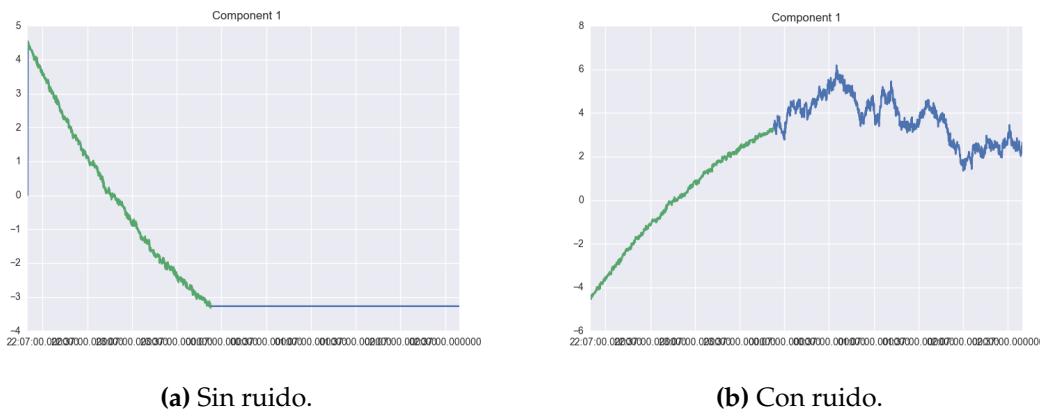


Figura 5.21: Simulación para la componente 1, usando SARIMAX con re-alimentación.

Pasemos a los resultados de las pruebas de simulación con búsqueda en los datos Gaussianos. Las pruebas que se describen a continuación tienen las siguientes consideraciones:

- Se han generado 1.000.000 puntos con distribución normal.
- Se han predicho 1000 puntos con ARIMA, uno a uno para cada componente: $p = \{c_1, c_2, c_3, c_4, c_5\}$, donde c_x es la predicción de la componente x . Para cada punto p , se busca el más cercano en los puntos Gaussianos. Se re-alimenta el modelo con este nuevo punto, para la siguiente iteración de predicción.
- Se han verificado los puntos simulados transformados y todos pertenecen todos al conjunto de puntos Gaussianos. Así que el resultado de Hotelling T2 debería ser bueno.
- Se está introduciendo ruido (valor aleatorio con desviación del error del modelo y media 0), que da un toque realista a la simulación. Si no se añade, la predicción tiende para valores constantes - eso hace que el punto más cercano sea siempre el mismo, como sucede en la Figura 5.21b.

Los gráficos de los resultados de simulación para la primera componente se presentan en la Figura 5.22 y para las demás componentes en la Figura 5.23. La leyenda es:

- verde - datos reales de la componente.
- azul - datos generados con SARIMAX y búsqueda de puntos en los datos Gaussianos.
- puntos gris - datos Gaussianos aleatorios generados.

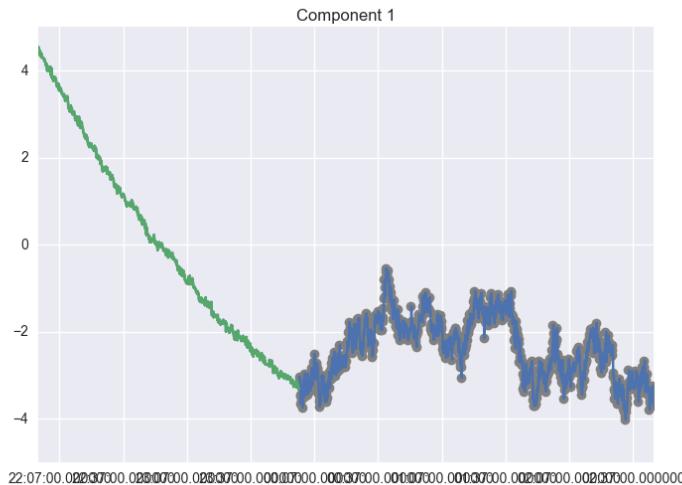


Figura 5.22: Simulación con búsqueda en los datos Gaussianos para la componente 1.

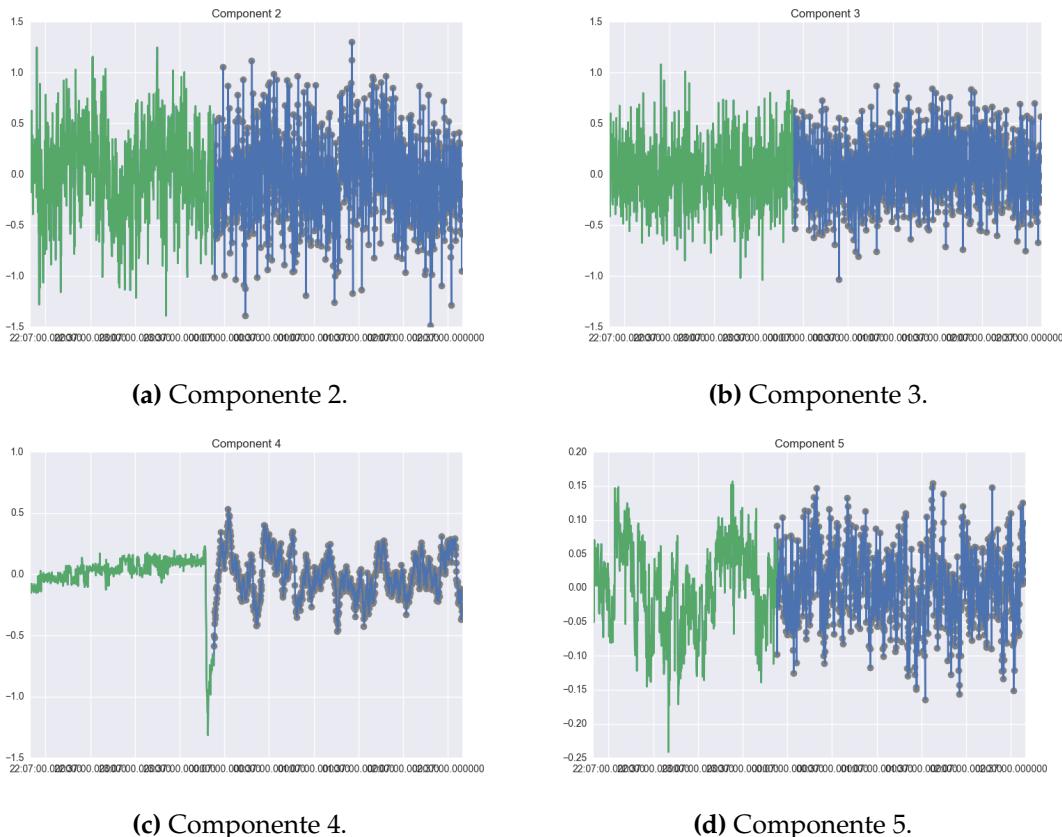


Figura 5.23: Simulación con búsqueda en los datos Gaussianos.

Las componentes 2 y 3 parecen que han captado bien la serie. La componente 1, que es la más relevante de todas, parece que tiene algún problema. También las componentes 4 y 5 presentan algunos problemas, aunque la última en menor medida. Probablemente no se ha conseguido completa estacionaridad del proceso.

Respecto a la componente 1, puede tener sentido la simulación. Lo que se pretendía era evitar que la serie bajara o subiera hasta el infinito y que se mantuviera dentro de los límites de los datos Gaussianos, que se ha mostrado que eran válidos. Eventualmente

se podría mejorar generando muchos más puntos Gaussianos, pues parece que al seleccionar los puntos más cercanos, estos están muy distantes. Esto se debe quizás a que nuestro espacio es N-dimensional y cuando el número de dimensiones crece es necesario una mayor cantidad (no lineal) de puntos para llenar ese espacio.

Con esta solución, la hipótesis de similitud de las poblaciones sigue siendo inválida, aunque el valor del estadístico vaya bajando. Es posible que cuando introduzcamos la serie el valor Hotelling T^2 ya no nos indique que es significativo porque seguramente estará simulando únicamente el último de los estados con lo que la población generada ya no será la inicial. Lo importante es que entre los dos procesos la simulación tenga forma de serie pero que no decrezca o crezca hasta el infinito y que su comportamiento temporal sea coherente con el proceso. El uso del test Hotelling T^2 , que se hizo al principio, era para constatar que efectivamente el método propuesto de las proyecciones funcionaba cuando simulaba valores aleatorios multidimensionales.

5.5 Big Data

La introducción de Big Data en esta implementación surge como una necesidad de escalado asociada a la generación de millones de puntos con distribución Gaussiana. Hasta la propuesta de esa solución, la exigencia computacional del algoritmo no era demasiado alta. Una maquina convencional, con un procesador i5 de 4 núcleos y 4 GB de memoria RAM, era suficiente para ejecutar todo el proceso en poco tiempo. Sobre todo porque el *dataset* es pequeño y además le estamos reduciendo la ventana de tiempo a un solo día.

Las estructuras `numpy` son eficientes, poderosas y están muy optimizadas. Pero, incluso optimizadas, pueden tener problemas de cálculo si se presenta un volumen muy grande de datos. El problema está sobre todo relacionado con la cantidad de memoria que la maquina es capaz de asignar para guardar los datos y poder trabajar sobre ellos. Por ejemplo, para el cálculo de una matriz `numpy` de 5 dimensiones con distribución normal, el consumo de memoria sería el indicado en la Tabla 5.2:

Tamaño de matriz numpy	Memoria
1.000.000	40 MB
10.000.000	400 MB
100.000.000	4000 MB = 4 GB

Tabla 5.2: Consumo de memoria de una matriz `numpy` según la cantidad de datos.

Si se generan 100.000.000 puntos multidimensionales, la memoria de la maquina referida anteriormente no es suficiente para cargar toda la estructura `numpy`. Así que el algoritmo `numpy` tendría que hacer *swapping* de datos entre la memoria y disco, volviendo la ejecución ineficiente y lenta.

Cuando el volumen de datos es exageradamente grande, se necesitan herramientas adecuadas para procesarlos. En este caso de estudio, se necesita guardar una gran cantidad de datos de la simulación Gaussiana en un repositorio, con buenas prestaciones de escritura y de lectura. Y, por otro lado, se necesita gran capacidad de procesamiento para calcular la distancia Euclídea a todos los puntos de esa simulación, de forma a encontrar el más cercano. Hay que dar énfasis al hecho de que cuanto más puntos de la simulación Gaussiana tengamos, mejor será para la simulación final, pues así se conseguirán puntos simulados más precisos. La cantidad de puntos de simulación Gaussiana deberá ser de millones o miles de millones. Las herramientas Big Data dan respuesta a estos requerimientos y, por lo tanto, viabilizan la solución propuesta en este estudio. Para resolver

los problemas de almacenamiento y de procesamiento se han elegido las herramientas *MongoDB* y *Apache Spark*, respectivamente.

5.5.1. MongoDB

MongoDB es una base de datos NoSQL. El concepto de NoSQL está relacionado con la abdicación de las reglas básicas de las bases de datos relacionales. Por ejemplo, en NoSQL, se suprime el uso de *joins* y *foreign keys*, en virtud de la rapidez de escritura y acceso a los datos. Justamente los *joins* y *foreign keys* son dos de los pilares de las bases de datos relacionales, que sustentan la integridad relacional.

Como referido en su documentación, *MongoDB* es una base de datos *open-source* que ofrece alto rendimiento, alta disponibilidad y escalado automático. En nuestro caso, la más importante característica es el alto rendimiento de persistencia y lectura para grandes cantidades de datos.

Los registros en *MongoDB* se llaman documentos y tienen formato JSON, con inmediata correspondencia con los diccionarios de Python. Así que el uso de Python ha facilitado bastante la introducción de *MongoDB* en el proyecto. Cada documento en *MongoDB* debe contener todos los datos necesarios por la aplicación, aunque para eso se cree información redundante. Con esto se obtienen lecturas muy rápidas, porque no hay necesidad de cruzar información. La verbosidad de los documentos debe ser bien pensada antes de crearlos - aunque la estructura de los datos es flexible en NoSQL. La aplicación se ha de encargar de la integridad de los datos, quitando también esa carga al motor de base de datos.

Los documentos se organizan en un nivel superior, las colecciones. En esta implementación se usan las colecciones:

- `data` - datos originales
- `component` - datos de componentes PCA
- `gaussian` - simulación Gaussiana
- `generated` - simulación con ARIMA

Todas las colecciones (excepto `data`) almacenan datos de la proyección al espacio reducido y la respectiva inversión al espacio original. La distinción se hace a través del campo `type`, que puede tener valores `component` o `inverse`. Con esto hacemos uso de las facilidades de *MongoDB* en lo que respecta a esquema de datos flexible y verbosidad de los documentos con redundancia.

En los códigos 5.12 y 5.13 podemos ver ejemplos de documentos de tipo `component` y de tipo `inverse` de la colección `generated`:

```
{
    "_id" : ObjectId("58a0442befb6f73ad7c8efb2"),
    "c2" : -0.03986047103177716,
    "type" : "component",
    "c4" : 0.059014880723704685,
    "c1" : -0.18597384414099433,
    "c0" : 1.6352153422327225,
    "c3" : 0.15650074726631932
}
```

Listing 5.12: Documento de tipo `component`.

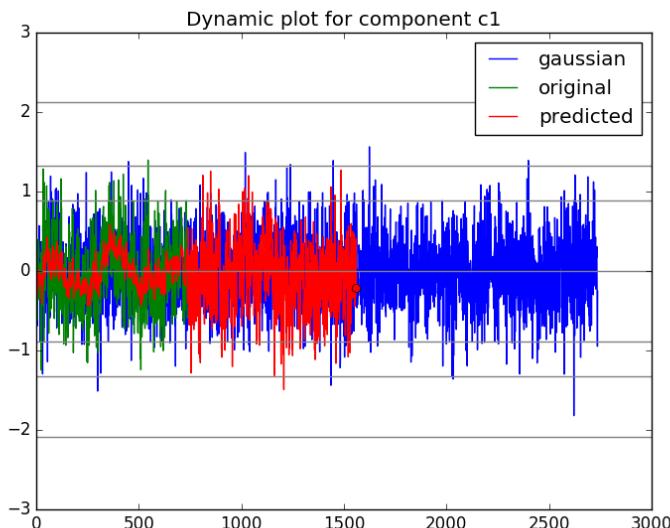
```
{
    "_id" : ObjectId("586fd37cefb6f73b6808b23d"),
    "H5x" : 37.97450920399996,
    "H2x" : 31.76233049441768,
    "ACPv" : 3.8062855363245203,
    "APHu" : 44.52178748294144,
    "APVs" : 68.84145878414336,
    "ACPx" : 3.3028060969846424,
    "H6x" : 39.41787367081148,
    "Svo" : 36.00581790038601,
    "type" : "inverse",
    "H3x" : 35.41299167251737,
    "H1x" : 44.85859802098669,
    "H7x" : 3276.7000000000003,
    "H4x" : 36.5817454605649,
    "ZUs" : 8.82171955708649,
    "ZSx" : 0.5999999999999999
}
```

Listing 5.13: Documento de tipo `inverse`.

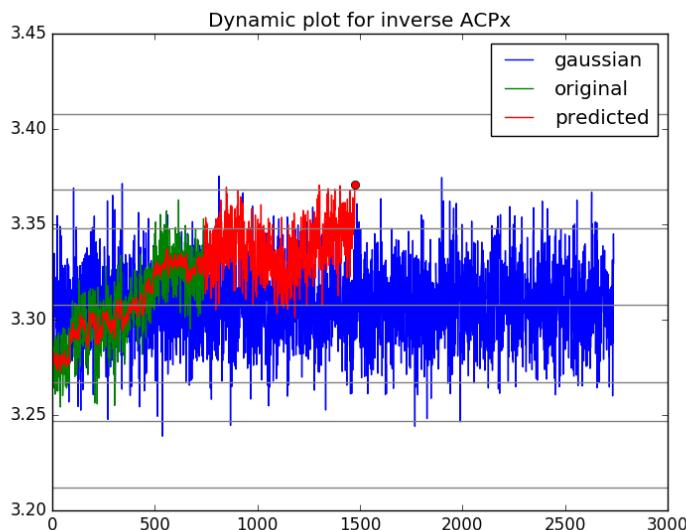
En la misma colección conviven dos estructuras de datos muy diferentes. Al leer estos datos, en cada documento, tenemos acceso a toda la información necesaria por la aplicación. Para acelerar el acceso a los documentos de un determinado tipo, se ha creado un índice en el campo `type`.

A parte de cumplir con la función de repositorio de datos de todas las partes del proceso, *MongoDB* se está usando como *buffer* para la generación de gráficos en tiempo real. Los nuevos puntos simulados (con ARIMA y posterior búsqueda en los datos Gaussianos) se van guardando en *MongoDB* progresivamente. De esta forma, se consigue una arquitectura *decoupled*, que permite que otro programa vaya leyendo esos datos y representando en gráficos con *matplotlib*.

En la Figura 5.24 se pueden ejemplos de la herramienta creada para generación de gráficos en tiempo real, para la segunda componente (gráfico 5.24a) y para los datos invertidos de *ACPx* (gráfico 5.24b).



(a) Datos de la componente 2.



(b) Datos invertidos de ACPx.

Figura 5.24: Gráficos de simulación en tiempo real.

Con esta herramienta podemos ver el progreso del algoritmo de simulación representado gráficamente en tiempo real. Se incluye solamente una parte de los datos Gaussianos (azul), una vez que la representación de todos los millones de puntos tendría una escala exagerada y no permitiría ver la evolución de la señal simulada con ARIMA. A los datos Gaussianos se superponen los datos reales (verde) y simulados con ARIMA y búsqueda en los puntos Gaussianos (rojo). El punto rojo indica el último punto simulado. Las líneas gris de los gráficos representan algunas estadísticas de la totalidad de la distribución Gaussiana: media, mínimo, máximo y intervalos de confianza de 95 % ($\mu \pm 2\sigma$) y 99 % ($\mu \pm 3\sigma$).

La representación gráfica permite observar determinados comportamientos de la simulación. Por ejemplo, la señal de ACP_x tiene una tendencia creciente. Se verifica que la simulación mantiene esa tendencia, pero se contiene por los límites de los datos Gaussianos, como se pretende.

5.5.2. Spark

La búsqueda del punto más cercano en los millones de datos Gaussianos es un claro ejemplo para la aplicación del paradigma de *MapReduce*. En el modelo de programación *MapReduce* los datos son divididos en partes más pequeñas y procesados en paralelo, en un algoritmo distribuido en un *cluster*. La fase *Map* se encarga de transformar los datos y emitir resultados, que serán agrupados en la fase de síntesis *Reduce*. En esta aplicación, la fase *Map* calcula la distancia de un punto simulado por ARIMA a cada punto de la simulación Gaussiana y emite cada resultado. Luego los *Reducers* se encargarán de ir comparando esos resultados, quedándose con un único punto (el de menor distancia) de cada comparación.

En el caso de que las instancias del *cluster* no sean suficientes, este paradigma permite escalar la capacidad de cómputo añadiendo más maquinas, de forma transparente para la aplicación. La programación del paradigma *MapReduce*, llamada programación funcional, es diferente de la programación imperativa convencional. Aquí hay que tener en cuenta que cada bloque de código tiene que ser independiente, para poder ser ejecutado en paralelo, lo que se convierte en una abstracción más difícil.

Una de las implementaciones más populares del paradigma *MapReduce* fue desarrollada por Google y se llama *Apache Hadoop*. Este framework es *open-source* y usa el almacenamiento HDFS (*Hadoop Distributed File System*) como núcleo de la implementación. La siguiente generación de herramientas *MapReduce* está encabezada por otro framework también *open-source* llamado *Apache Spark*. Aunque mantiene las fundaciones de Hadoop, *Spark* usa la memoria como principal recurso de trabajo, al contrario de Hadoop, que usa el disco (HDFS). Por ese hecho, las prestaciones de *Spark* son, evidentemente, mejores, llegando a valores de 100 veces más rápido que *Hadoop*.

Una vez más, Python ha contribuido para la interacción con herramientas Big Data, en este caso *Spark*. La programación del proyecto se hizo en Python desde el primer momento, imaginando la necesidad de posterior migración a paradigmas Big Data. Además la síntesis de escrita en Python es a la vez intuitiva y poderosa, como podemos ver en las fases de *Map* (código 5.14) y de *Reduce* (código 5.15).

```
# calculate Euclidean distances
distances = _points.map(lambda x: (x, np.sqrt((x - _point) ** 2).sum()))
```

Listing 5.14: Map en Python.

```
# take the minimum distance point
min_distance = distances.takeOrdered(1, key=lambda x: x[1])
```

Listing 5.15: Reduce en Python.

Spark tiene varias librerías construidas sobre el motor principal. Un ejemplo es *MLlib*, una librería para *Machine Learning*, que se ha usado para la generación de los puntos de simulación Gaussiana, a través de RandomRDDs del módulo `pyspark.mllib.random`.

Otra librería es *Spark SQL*, que ha sido de utilidad para crear *DataFrames* y así interactuar fácilmente con *MongoDB*. En este caso, a través de un conector, se habilita la combinación de las dos herramientas para que la persistencia pueda ser hecha de forma distribuida.

CAPÍTULO 6

Conclusiones

TODO

CAPÍTULO 7

Trabajos futuros

ARIMA: añadirle el ruido residual a la predicción y realizar la predicción del siguiente valor con el valor de la predicción actual que incluye el ruido.

Bibliografía

- [1] Jon Shlens. *A TUTORIAL ON PRINCIPAL COMPONENT ANALYSIS. Derivation, Discussion and Singular Value Decomposition.* Version 1, 25 March, 2003.
- [2] Unknown Authors. *The Truth about Principal Components and Factor Analysis.* 28 September, 2009.
- [3] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.
- [4] Montgomery, Douglas C. Jennings, Cheryl L. Kulahci, Murat *Introduction to Time Series Analysis and Forecasting.* Springer, 2006.
- [5] *Principal component analysis (PCA).* Consultar <http://scikit-learn.org/stable/modules/decomposition.html#principal-component-analysis-pca>.

