ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
INGENIERÍA DEL SOFTWARE

Análisis de información predictivo en el sector turístico

Predictive analysis of information in the tourism sector

Realizado por
Francisco Mesa Pérez
Tutorizado por
Francisco López Valverde
Departamento
Lenguajes y ciencias de la computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, Julio 2017

Fecha defensa:
El Secretario del Tribunal

Resumen: En este trabajo de fin de grado se ha presentado un estudio sobre los modelos predictivos ARIMA, MLP y LSTM, donde se ha realizado una comparativa entre todos estos con la finalidad de tomar el más adecuado como modelo de predicción. Además, se ha realizado el tratamiento de las series temporales para conseguir resultados óptimos para nuestro problema, la predicción del número de pasajeros que vendrán a Málaga en un futuro para la toma de decisiones de los diversos negocios, en nuestro caso se tomaron como referencia la base de datos de los vuelos "Amadeus" y es sobre esta base de datos sobre la cual se ha realizado todos los estudios. Sin embargo, los modelos son construidos de tal manera que pueden ser probados con cualquier otra base de datos, siempre y cuando estemos hablando de series temporales con estacionalidad de 12 meses, ya que todos los tratamientos realizados en los modelos giran en torno a esto.

Palabras claves: predicción, ARIMA, LSTM, MLP, series temporales, redes neuronales, entrenamiento.

Abstract: In this end-of-grade paper a study has been presented on the ARIMA, MLP and LSTM predictive models, where a comparison has been made between all these three models with the aim of taking the most appropriate one as our final prediction model. In addition, our time series have been treated in order to achieve optimum results for our problem, the prediction of the number of flights that will come to Malaga in the future for the decision making of the various businesses, in our case we are taken as reference the database of flights " Amadeus "and it is on this database on which all the studies have been carried out. However, the models have been constructed in such a way that they can be tested with any other database, as long as they can be treated as time series with 12 months seasonality because all the treatments made in the models revolve around this.

Keywords: prediction, ARIMA, LSTM, MLP, time series, neuronal networks, training.

ÍNDICE

1. Introducción

1.1 Motivación: ¿Por qué es necesaria la predicción de datos?

Vivimos en una época donde la información es una de las piezas clave para perdurar en un mercado tan competitivo como el que hay hoy en día, por lo tanto, la información es una de las armas más poderosas que pueden tener las empresas. Si bien es bueno tener en cuenta cómo está funcionando el mercado a tiempo real, hay que estar preparado para el futuro, es decir, hay que saber leer tanto como están cambiando las preferencias y los gustos de los usuarios para poder satisfacer sus necesidades en un futuro y focalizar los esfuerzos y recursos de la empresa en empezar a desarrollar un producto que satisfaga a ese tipo de usuarios en lugar de desarrollar otro producto que no será demandado. Por ejemplo, tenemos dos empresas A y B, las dos se dedicas a la fabricación de aviones comerciales. La empresa A como siempre ha tenido buena clientela nunca se preocupó de cómo estaba evolucionando el mercado y focalizo todos sus esfuerzos en mejorar su único producto, los aviones comerciales. Sin embargo, la empresa B, la cual estaba bastante por detrás de la empresa A en cuanto a ventas, a partir de la información predicha por sus equipos de análisis de datos se percató de que los jets privados iban a crecer bastante en cuando a la demanda en unos pocos años, así como que la demanda de aviones comerciales estaba en caída. Al cabo de unos años cuando la empresa A quiso reaccionar y fue a quiebra siendo absorbida por la empresa B, que gracias a saber leer el mercado tuvo un gran crecimiento aumentando sus ingresos. Con este ejemplo, se queda bastante claro que la predicción de datos puede llegar a convertirse en una de las piezas fundamentales a la hora de la toma de decisiones por parte de las empresas ya que si bien no existen máquinas de predicción del futuro, es muy útil tener unas nociones aproximadas de cómo está evolucionando el mercado.

1.2 Objetivos

-Búsqueda del modelo más óptimo para la predicción de datos en series temporales

+Comprobar su mejora con respecto a los métodos tradicionales

-Testear TensorFlow y comprobar que tanto nos ofrece en el campo de análisis de datos

2. Tecnologías

2.1 TensorFlow

“TensorFlow es una biblioteca de código abierto para aprendizaje automático a través de un rango de tareas, y desarrollado por Google para satisfacer sus necesidades de sistemas capaces de construir y entrenar redes neuronales para detectar y descifrar patrones y correlaciones, análogos al aprendizaje y razonamiento usados por los humanos.”

(*https://es.wikipedia.org/wiki/TensorFlow)*

2.2 Python

“Python es un lenguaje de programación multiparadigma cuya filosofía se basa en tener una sintaxis que favorezca un código legible, este lenguaje soporta tanto orientación a objetos, programación imperativa como programación funcional.”

(https://es.wikipedia.org/wiki/Python)

2.3 SQL

“Lenguaje de programación diseñado para almacenar, manipular y recuperar datos almacenados en bases de datos relacionales.”

(http://www.1keydata.com/es/sql/)

2.4 Keras

Keras es una biblioteca de redes neuronales de código abierto escrita en Python. Es capaz de funcionar sobre Deeplearning4j, Tensorflow, CNTK o Theano. Keras esta diseñado para permitir la experimentación rápida con redes neurales profundas, se centra en ser mínimo, modular y extensible.

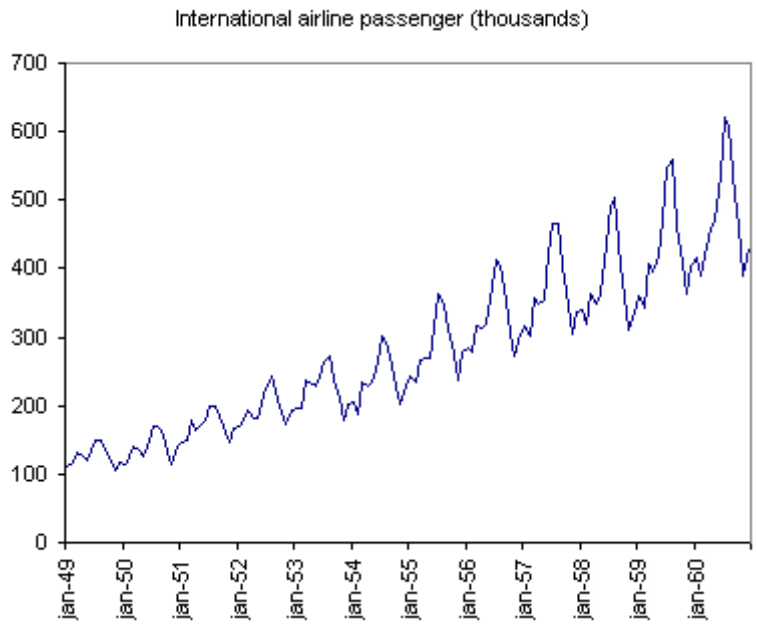
(https://en.wikipedia.org/wiki/Keras)

3. Proyecto

Para comenzar el proyecto vamos a ir presentando el cómo se ha sido el desarrollo del proyecto desde sus etapas iniciales hasta el final donde se van a ir viendo todas las conclusiones y datos relevantes obtenidos de esta investigación.

Para comenzar vamos a trabajar con series temporales, pero, ¿Qué es una serie temporal?

Bueno, una serie temporal no es más que un conjunto de valores tomados en determinados momentos.



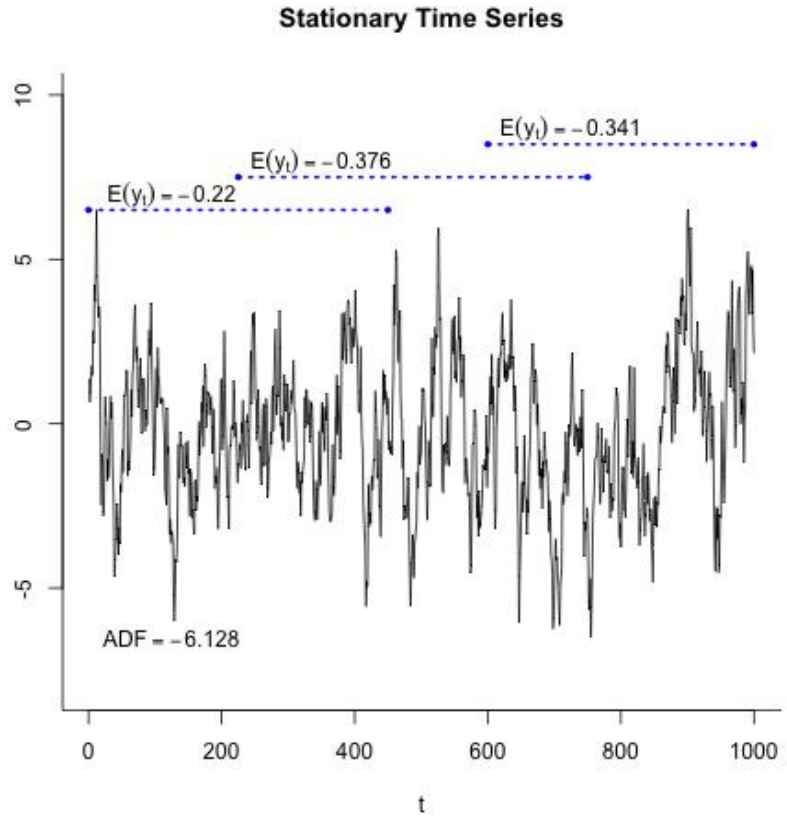
http://static.xlstat.com/uploads/assets/Tutorials/Time/Differencing/hw1.png

En un primer lugar vamos a hablar un poco de uno de los métodos más antiguos de predicción de datos en series temporales ARIMA.

3.1 ARIMA

3.1.1 ¿Qué es?

ARIMA (autoregressive integrated moving average) es un modelo estadístico que se basa en el uso de series temporales para la predicción de tendencias futuras. Este modelo no es más que una generalización del modelo ARMA (autoregressive moving average), que hace posible la construcción de modelos de predicción de datos en series temporales las cuales pueden ser transformadas en series temporales estacionarias, es decir, series temporales carentes de tendencia y con periodicidad en las muestras a lo largo del tiempo.



https://upload.wikimedia.org/wikipedia/commons/thumb/e/e1/Stationarycomparison.png/390px-Stationarycomparison.png

Este modelo se basa en:

-**AR (Autoregression):** Modelo que se basa en la predicción de la variable de interés usando una combinación de los valores pasados de la variable. El término de autorregresión, ya deja claro que es una regresión de la variable sobre ella misma.

De esta manera, un modelo autorregresivo *AR(p)* de orden p puede escribirse como:

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \cdots + \phi_p y_{t-p} + e_t$$

Dónde:

- c: es una constante.
- e_t: white noise.
- $\phi_1, \dots, \phi_q$: son parámetros del modelo.

-**MA (Moving average):** El modelo de media móvil se basa en el uso de la dependencia entre una observación y un error residual de la media móvil aplicada a observaciones anteriores.

De esta manera, un modelo autorregresivo *MA(q)* de orden q puede escribirse como:

$$y_t = c + e_t + \theta_1 e_{t-1} + \cdots + \theta_q e_{t-q}$$

Dónde:

- c: es una constante.
- e_t: white noise.
- $\theta_1, \dots, \theta_q$: son parámetros del modelo.

-**I (Integrated):** El uso de diferencia entre las observaciones para convertir series temporales en series estacionarias

3.1.2 ¿Cómo construimos ARIMA?

El modelo ARIMA como ya se explicó con anterioridad surge de la combinación de los modelos *AR* y *MA* junto con la diferenciación de la serie temporal no estacionaria a una estacionaria. El modelo completo podría ser escrito como:

$$y'_t = c + \phi_1 y'_{t-1} + \cdots + \phi_p y'_{t-p} + \theta_1 e_{t-1} + \cdots + \theta_q e_{t-q} + e_t$$

Nota: y'_t es la seria con la diferenciación ya realizada (Puede haberse hecho más de una).

Una simplificación del modelo que se puede conseguir mediante el uso de la notificación de backshift sería ***ARIMA(p,d,q)***:

$$\underset{\uparrow}{(1-\phi_1 B-\cdots-\phi_p B^p)} \quad \underset{\uparrow}{(1-B)^d y_t} \quad = \quad c+\underset{\uparrow}{(1+\theta_1 B+\cdots+\theta_q B^q)} e_t$$

$$\text{AR}(p) \qquad d \text{ diferencias} \qquad \text{MA}(q)$$

***Figura 1**: Simplificación modelo ARIMA*

Dónde:

- p (Orden de retraso): Número de observaciones retrasadas incluidas en el modelo.
- d (Grado de diferenciación): El número de veces que las observaciones brutas van a ser sometidas a una diferencia
- q (Orden de la media móvil): Variable que nos permite fijar el error de nuestro modelo como una combinación lineal de los errores de los valores observados en el pasado.

Ahora bien, ¿Qué valores le corresponderán a cada una de estas variables? Esto lo veremos en el próximo punto.

3.1.3 Calculando los valores de p, d, q para el modelo ARIMA

Calculo de la variable d

Para calcular el valor de la variable ***d*** vamos a utilizar el método de *diferenciación* hasta conseguir que nuestra serie temporal sea estacionaria. Para definir cuando una serie temporal es estacionaria vamos a aplicar el test de "*Dickey-Fuller*", el cual, si bien no nos va a asegurar al 100% que nuestra serie temporal es estacionaria si nos lo puede llegar a asegurar hasta un 99%. Sin centrarnos demasiado en el funcionamiento de este test, lo único que es necesario conocer es la interpretación de los resultados que nos retorna. Tal y como se puede observar en la *figura 2*, nuestro test nos va a retornar un valor, en este caso "-2.71", ¿Cómo interpretamos este valor? Bueno, según nuestro test vemos que nuestro valor es menor que "-2.57" y por lo tanto podemos afirmar al 90% que nuestra serie temporal es estacionaria. Sin embargo, si hubiese sido menor que "-2.88", podríamos decir que es estacionaria al 95% y aplicaríamos la misma lógica para asegurar que nuestra serie es estacionaria al 99% con el valor "-3.48".

```
Results of Dickey-Fuller Test:
Test Statistic                  -2.717131
Number of Observations Used    128.000000
Critical Value (5%)             -2.884398
Critical Value (1%)             -3.482501
Critical Value (10%)            -2.578960
dtype: float64
```

***Figura 2**: Test de Dickey-Fuller*

Por tanto, en función del grado de diferenciación que se haya requerido para la conversión de una serie temporal no estacionaria en una estacionaria variara el valor de *d*. Siendo más concretos, si con una diferenciación de grado 1 obtenemos una serie temporal estacionaria bajo un porcentaje aceptable mediante el test de "*Dickey-Fuller*", entonces d=1, si se ha necesitado de una diferenciación de grado 2 entonces, d=2…

Calculo de la variable p

Con la finalidad de calcular el valor de la variable ***p*** para nuestra serie temporal se va a utilizar la ***Función de Autocorrelación (ACF)***, la cual mide la correlación entre una serie temporal con una versión retardada de sí misma. Por ejemplo, para un retardo de 5, ACF compararía series en el instante t_1,t_2… con los instantes t_{1-5},t_{2-5}…

Pasando a como se extraería el valor de esta variable, es bastante intuitivo, se basa en calcular el primer punto donde ACF corta con el intervalo de confianza y redondearlo. Tal y como se ve en la *figura 3* el primer punto donde se corta con el intervalo de confianza es aproximadamente 2, por lo tanto *p=2*.

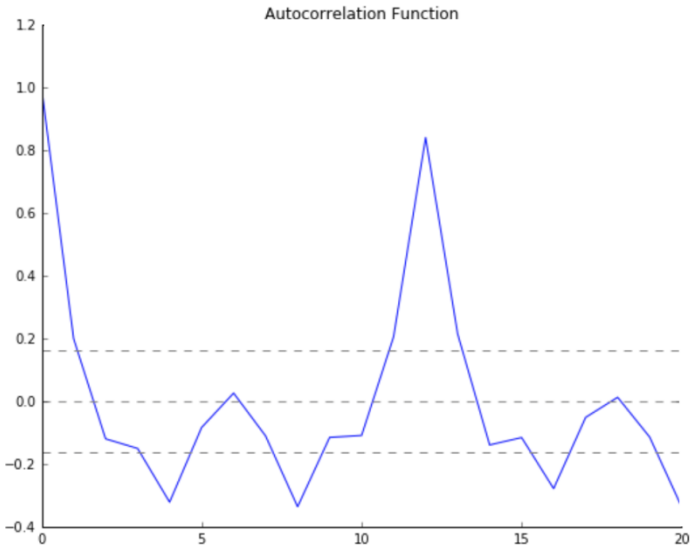


***Figura 3**: Función ACF*

Calculo de la variable q

Para el cálculo de la variable *q* se va a utilizar la ***Función de Autocorrelación Parcial (PACF)***, la cual mide la correlación entre una serie temporal con la versión retardada de sí misma pero después de eliminar las variaciones ya explicadas por las comparaciones intermedias. Un ejemplo sería, a retardo 5, comprobaría la correlación, sin embargo, eliminaría los efectos ya explicados por los retardos 1 y 4.

Ahora bien, el uso de esta función para la extracción de la variable q es equivalente al método utilizado para obtener p mediante la función ACF. La variable ***q*** es el valor redondeado del primer punto donde PACF corta con el intervalo de confianza. Por ejemplo, en la *figura 4* el primer punto donde se corta con el intervalo de confianza es aproximadamente 2, por lo tanto *q=2*.

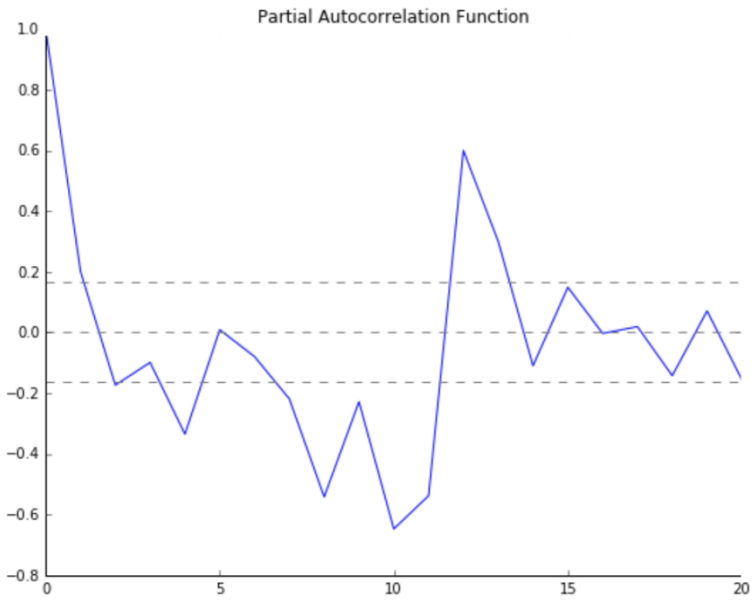


Figura 4*: Función PACF*

3.1.4 Implementación

Nuestra implementación, si bien es propia, está basada en algunas fuentes, las cuales, estarán disponibles en la sección de *Referencias* al final de la memoria. Dicho esto pasemos a explicar nuestra implementación del modelo ARIMA.

En este caso, al ser un modelo estadístico, TesorFlow carecía de sentido alguno como forma de implementar este modelo, así que se optó por el uso de Python junto con algunas librerías como ***numpy***, para la manipulación de arrays, ***pandas***, para la lectura de ficheros…

Tratamiento de datos

El tratamiento de los datos que se ha tenido que utilizar para poder realizar ha sido un poco tedioso, aun así es la única manera viable de poder realizar este tipo de operaciones en Python. El método es el siguiente:

En un primer lugar se procede a la creación de una conexión con la base de datos

```
db = MySQLdb.connect(host="localhost", # host,
                     user="root",      # nombre de usuario
                     passwd="123456",  # contraseña
                     db="dbturismo")   # nombre de la base de datos
```

Una vez se ha generado esta conexión se procede a la creación del cursor y ejecución de nuestra query, en nuestro caso vamos a obtener la información de la tabla de amadeus_vuelos:

```
# Creación del cursor
cur = db.cursor()

#Ejecución de la query
cur.execute("SELECT date, sum(travelers) FROM amadeus_vuelos GROUP BY date")
```

Ahora bien, en nuestro caso no vamos a poder trabajar con los datos de esta forma así que se va a realizar una copia de todos estos datos en un documento ***".csv"*** el cual leeremos más tarde para la recolección de toda la información obtenida por nuestra consulta.

```
with open('temp.csv','wb') as f:
    writer = csv.writer(f)
    writer.writerow(["date","travelers"])
    writer.writerows(cur.fetchall())
cur.close()
db.close()
```

Y finalmente se procede a la lectura de este fichero para almacenar toda la información en un array con el cuál se trabajará a lo largo de todo nuestro programa. Al estar trabajando con series temporales a la hora de hacer la lectura de los datos indicamos que la columna, la cuál va a actuar como índice es aquella que contiene las fechas, para ello deberemos hacer de

manera adicional un “parseo” indicándole a nuestro programa que tipo de estructura tiene nuestra fecha.

```
dateparse = lambda dates: pd.datetime.strptime(dates,'%Y-%m-%d %H:%M:%S')

data = pd.read_csv('temp.csv', parse_dates=[0],index_col='date',date_parser=dateparse)
```

Con todo esto ya podemos trabajar con los datos sobre los pasajeros:

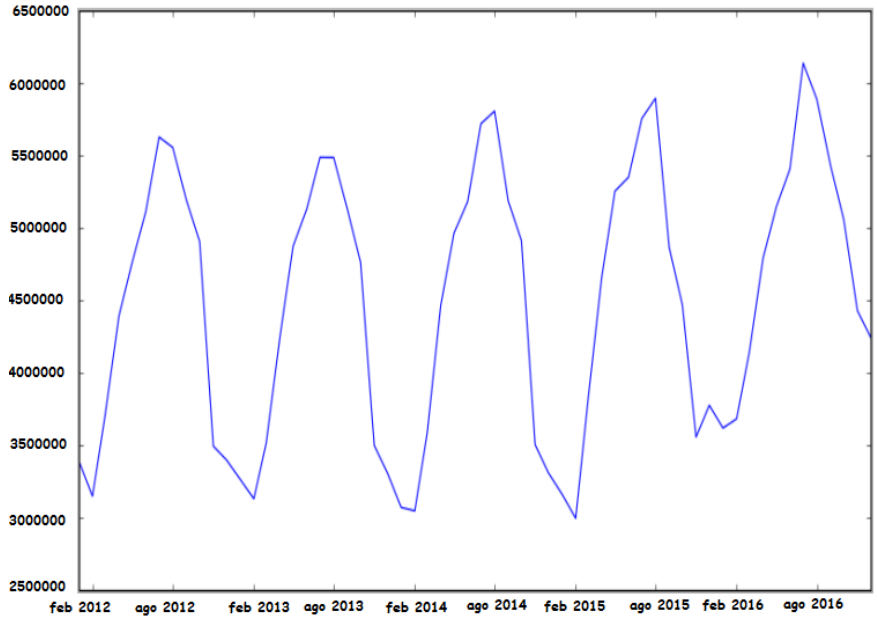


***Figura 5**: Gráfica de los pasajeros de Amadeus a Málaga (2012-2016)*

Test de “*Dickey-Fuller*”

Para realizar este test hemos hecho uso de la librería ***statsmodels.tsa.stattools***. El test ha sido construido en una función aparte donde se realizaran todos los cálculos relacionados con la estacionalidad de nuestra función. En sí los valores que nos van a interesar son:

-**Valor de p:** Este valor nos va a indicar que probabilidad hay de que nuestro modelo sea no estacionario. Por ejemplo, para un valor, ***p=0.96***, la interpretación sería tal que *“hay un 96% de probabilidad de que nuestro modelo sea no estacionario”*

-Lags usados: Indica la cantidad de términos Y_{t-i} a incluir para asegurarnos de que u_t sea "White noise".

$$Y_t = \alpha + \beta\, t + Y_{t-1} + \sum_{i=1}^{p} d_i \,\Delta\, Y_{t-i} + u_t$$

Figura 6: *Fórmula Test Dickey-Fuller*

-Número de observaciones usadas

-Valores críticos: Son valores que nos indicarán sobre qué porcentaje de seguridad nuestro modelo es no estacionario. Los más utilizados son **10%,5% y 1%**.

La parte del código la cual implementa este test es la siguiente:

```
def test_stationarity(timeseries):


    #Test Dickey-Fuller: Comprobando si el modelo es estacionario o no
    print 'Resultados del test Dickey-Fuller:'
    dftest = adfuller(timeseries,autolag='AIC')
    dfoutput = pd.Series(dftest[0:4], index=['Test','valor de p','Lags usados','Número de observaciones usadas'])
    for key,value in dftest[4].items():
        dfoutput['Critical Value (%s)'%key] = value
    print dfoutput
```

En este caso no hay mucho que detallar sobre esta parte así que pasemos al siguiente capítulo donde se hablará de las técnicas utilizadas para convertir un modelo no estacionario en un modelo estacionario y así como la asignación de las variables para nuestro modelo ***ARIMA.***

Cálculo valores para el modelo ARIMA (p,d,q)

Valor d

Tal y como se explicó con anterioridad ***d*** es el grado de diferenciación utilizado para conseguir que nuestro modelo sea estacionario. Para comprobar si nuestro modelo es o no estacionario vamos a valernos del test de *Dickey-Fuller* mencionado en el punto anterior.

En una primera instancia el test de Dickey-Fuller nos devuelve los siguientes resultados:

Resultados del test de Dickey-Fuller

Test	2.943245
Valor de p	1.000000
Lags usados	11.000000
Número de observaciones usadas	48.000000
Valor crítico (5%)	-2.923954
Valor crítico (1%)	-3.574589
Valor crítico (10%)	-2.600039

***Figura 7**: Test de Dickey-Fuller (Modelo original)*

Tal y como se puede observar en los resultados de nuestro test *(Figura 7),* nuestro modelo es 100% no estacionario si miramos el valor de p. Además, nuestro valor está muy lejos del valor de *-2.600039*, que se necesita para comenzar a afirmar que nuestro modelo es un modelo estacionario con un margen de error “pequeño”, ya que 10% sigue siendo un valor considerablemente alto. Por lo tanto, d≠0 y se hace necesaria la aplicación de la diferenciación mencionada con anterioridad.

Realicemos una diferenciación de grado 1 entonces:

Resultados del test de Dickey-Fuller	
Test	-2.491302
Valor de p	0.117605
Lags usados	11.000000
Número de observaciones usadas	47.000000
Valor crítico (5%)	-2.925338
Valor crítico (1%)	-3.577848
Valor crítico (10%)	-2.600774

***Figura 9**: Test de Dickey-Fuller (Modelo con diferenciación de grado 1)*

El código para realizar esta operación de diferenciación no es complejo, se vería tal que:

```
ts_dif_log = ts_log-ts_log.shift(1)
ts_dif_log.dropna(inplace=True)
```

Como se puede apreciar en la *figura superior* eliminamos todos los *Na* que se van a generar en nuestra diferenciación. Conclusión d≠1.

Y ya finalmente con una diferenciación de grado 2 vamos a obtener que nuestro modelo es no estacionario a una probabilidad la cual podría considerarse cero. Por lo tanto, d=2.

Resultados del test de Dickey-Fuller	
Test	-12.71093
Valor de p	1.030576×10^{-23}
Lags usados	10.000000
Número de observaciones usadas	47.000000
Valor crítico (5%)	-2.925338
Valor crítico (1%)	-3.577848
Valor crítico (10%)	-2.600774

***Figura 10**: Test de Dickey-Fuller (Modelo con diferenciación de grado 2)*

El código para realizar esta diferenciación de grado 2 es el siguiente:

```
ts_dif_log = ts_log-2*(ts_log.shift(1))+ts_log.shift(2)
ts_dif_log.dropna(inplace=True)
```

Valor p

En este caso vamos a utilizar la ***Función de Autocorrelación (ACF)***, donde todo lo referente a este ya quedo explicado con anterioridad.Por tanto, únicamente vamos a presentar el código y el proceso de captura de este valor.

El código es extremadamente simple, ya que nos vamos a valer de una librería externa, " *from statsmodels.tsa.stattools import acf* ", la cual se va a encargar de realizar todas las operaciones necesarias para conseguir el modelo.

```
lag_acf = acf(ts_dif_log,nlags=12)
```

El modelo que se crea es tal y como se muestra en la *figura 11*. Ahora, como bien se puede apreciar, el primer corte con el intervalo de confianza es en el punto x=1 y por tanto, p=1.

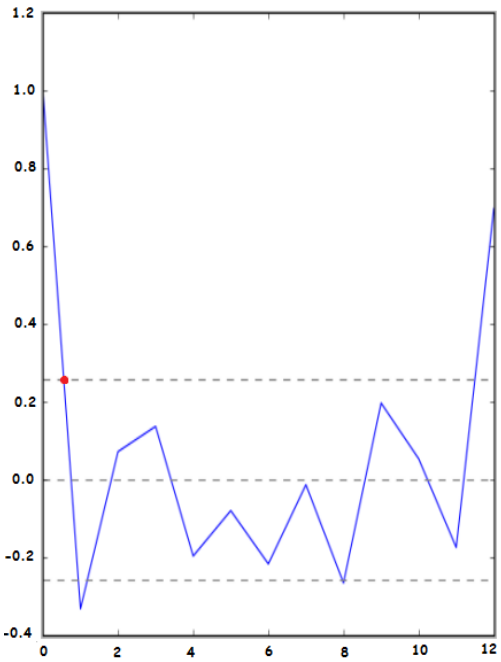


***Figura 11**: Función ACF*

Valor q

Aquí se utilizará la ***Función de Autocorrelación Parcial (PACF)***, explicada anteriormente**,** y por lo tanto vamos a pasar directamente al código. Como se puede apreciar en la *figura de abajo,* la función utilizada para la generación del modelo ***PACF*** es muy similar a la utilizada para el cálculo de **ACF**.

```
lag_pacf = pacf(ts_dif_log, nlags=12, method='ols')
```

Pasando al modelo generado en la *figura 12,* se deja ver que el primer corte con el intervalo de confianza ha sido en x=1.Y por ende, q=1.

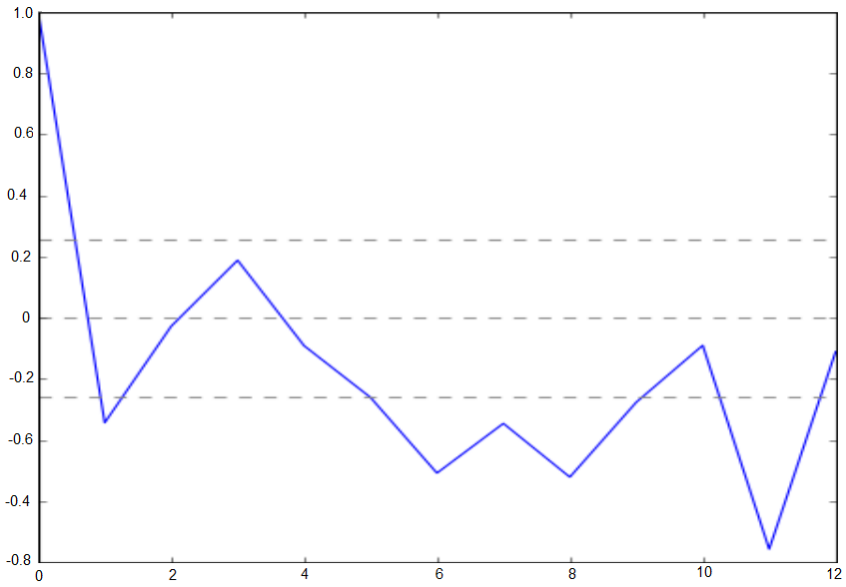


Figura 12*: Función PACF*

Con esto todos los valores del modelo ARIMA (p,d,q) quedan totalmente calculados. Ahora pasemos a la elaboración del modelo en sí.

Modelo ARIMA (p,d,q)

En este apartado vamos a desarrollar el modelado del modelo ***ARIMA*** en nuestro código entrado además, en la comparación de los datos que predijo nuestro modelo ARIMA con los reales.

El modelo ARIMA que se va a utilizar es perteneciente a la librería " *from statsmodels.tsa.arima_model import ARIMA* ". Por lo tanto el código se simplificó bastante como se ve en la *siguiente figura*.

```
#ARIMA
ts_log = ts_log.astype(np.float)
model = ARIMA(ts_log, order=(1, 2, 1))
results_ARIMA = model.fit(disp=-1)
```

Ya en la *Figura 13* vamos a ver el resultado de todo el proceso, además, vamos a calcular la ***RMSE o error cuadrático medio***, valor que nos va a indicar que tan buena es nuestra aproximación y además será un medio de comparación con otros modelos como los siguientes que se mostraran en este proyecto.

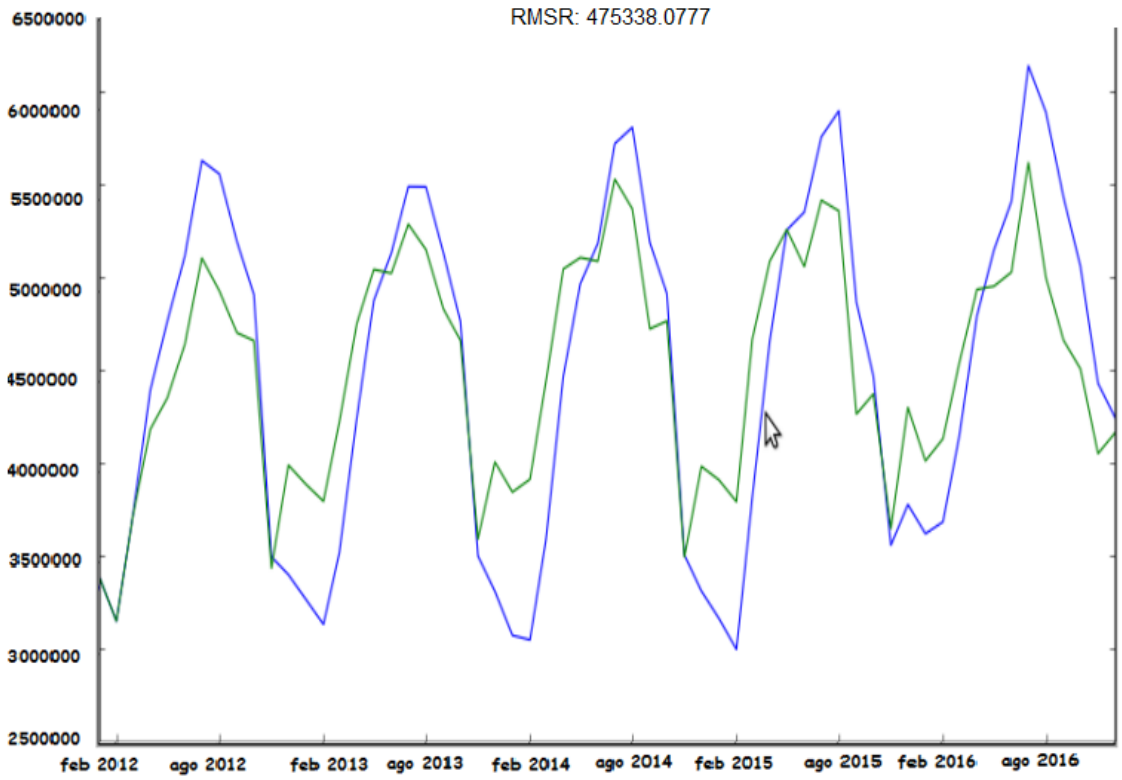


***Figura 13**: Predicción ARIMA*

3.4 Perceptrón multicapa

3.2.1 ¿Qué es perceptrón multicapa?

Una RNA es una red neuronal artificial multicapa, la cual mapea conjuntos de datos de entrada en outputs apropiados. Esta estructura le permite resolver problemas linealmente no separables. Nuestra red neuronal se va a dividir en tres capas *(Figura 14)*:

- **Capa de entrada**: Formada por aquellas neuronas que van a introducir los patrones de entrada en nuestra red. Cabe destacar que en estas neuronas no se produce procesamiento.
- **Capa oculta**: Constituida por aquellas neuronas las cuales reciben la entrada de la capa anterior *(Capa de entrada)* y cuyas salidas pasan a neuronas de capas posteriores.
- **Capa de salida**: Formada por aquellas neuronas cuyos valores de salida se corresponderán con las salidas de toda la red.

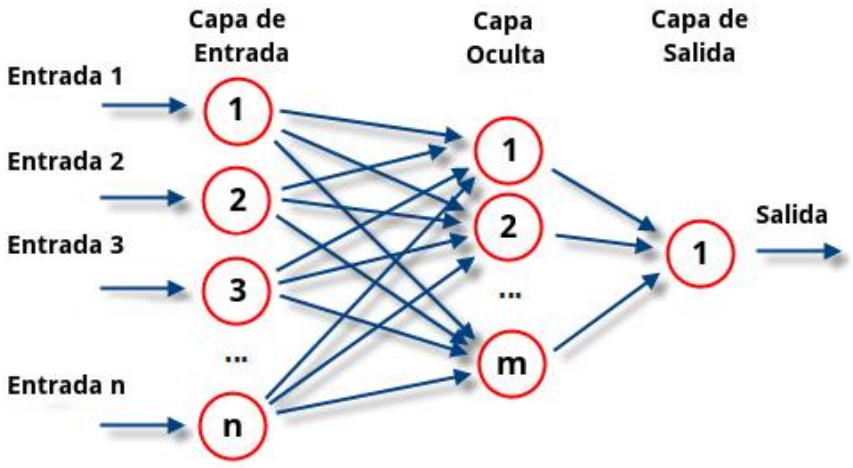


***Figura 14**: Ejemplo perceptrón multicapa*

https://es.wikipedia.org/wiki/Perceptr%C3%B3n_multicapa

3.2.2 ¿De qué está compuesto nuestro modelo?

Como ya se podía intuir, nuestro modelo está compuesto de **neuronas**, es decir, unidades de procesamiento de información, las cuales van a recibir información de las neuronas de la capa anterior, procesaran la información y emitirán un resultado a través de sus conexiones a las neuronas de las capa siguiente. Además, cada una de las conexiones tendrá un determinado peso, también denominado ***"peso sináptico"***, dato sobre el que se basara nuestra neurona para decidir la importancia de la información de cada una de las entradas que recibe.

Procesamiento de la información

Nivel de Neurona

El procesamiento de información de una neurona ***Y***, va a consistir en una función ***F***, la cual va a operar los valores obtenidos de la capa anterior, y que además tendrá en cuenta el valor sináptico de la conexión por la que se recibió la información W_i.

Un modelo sencillo de la función ***F*** sería:

$$F = X_1W_1 + X_2W_2 + \cdots + X_iW_i$$

Cabe destacar que en este tipo de modelos las funciones de activación pueden ser no lineales. Continuando con la explicación, si el resultado de esta función es mayor que un umbral **U**, la neurona se activa y emite una señal hacia las neuronas de la capa siguiente. Por el contrario, si el resultado es menor que el umbral, la neurona permanecerá inactiva y no enviará ninguna señal.

Neurona inactiva: $F = X_1W_1 + X_2W_2 + \cdots + X_iW_i < U$

Neurona activa: $F = X_1W_1 + X_2W_2 + \cdots + X_iW_i > U$

Nivel de Red

Definido un conjunto inicial de pesos para las conexiones, al presentarse un “estímulo” en la capa de entrada, las neuronas de cada capa realizarán su propio proceso de procesamiento de la información entrante a través de la función ***F***. En función de si cada una de estas neuronas superó o no el valor umbral se activarán o no, de tal manera que al final del proceso, en la capa de salida se genera un resultado.

3.2.3 Entrenamiento

En el entrenamiento de este modelo va a provocar el cambio de los valores umbral ***U*** de cada neurona, así como los pesos de las conexiones W_i con la finalidad de conseguir que los resultados generados por la red coincidan con los resultados esperados.

3.2.4 Implementación

Nuestra implementación, si bien es propia, está basada en algunas fuentes, las cuales, estarán disponibles en la sección de *Referencias* al final de la memoria. Dicho esto pasemos a explicar nuestra implementación del modelo Perceptrón multicapa.

En este caso, al contrario de lo que sucedió con el modelo ARIMA, la utilización de *TensorFlow* se encuentra totalmente justificada, ya que nos encontramos ante un modelo que requiere de entrenamiento, y es en este tipo de tareas en las que *TensorFlow* despunta como herramienta. Ahora bien, la utilización de *TensorFlow* sin ningún tipo de framework complica innecesariamente el código y dado que tiene una comunidad muy activa, posee una gran cantidad de frameworks con los que se puede trabajar, en nuestro caso, vamos a utilizar uno de los más famosos llamado, ***Keras***¸ el cual nos va a permitir utilizar toda la potencia de TensorFlow de una manera bastante simplificada, ya que contiene la mayoría de los modelos de predicción, lo que convierte varias páginas de código con sintaxis de TensorFlow a un conjunto reducido de funciones fáciles de tratar.

Tratamiento de datos

En este caso el tratamiento de datos es prácticamente equivalente al apartado de tratamiento de datos del *modelo ARIMA*, así que vamos a prescindir de entrar en mucho detalle. Solo vamos a puntualizar que en este caso por problemas relacionados con las funciones de Keras, no se pudo utilizar fechas como índices, sin embargo, el resultado es totalmente equivalente y seguimos recibiendo la misma información del modelo. El código sería:

```
dataframe = pandas.read_csv('temp.csv', usecols=[1],engine='python')
dataset = dataframe.values
dataset = dataset.astype('float32')
```

Y la gráfica resultante sería:

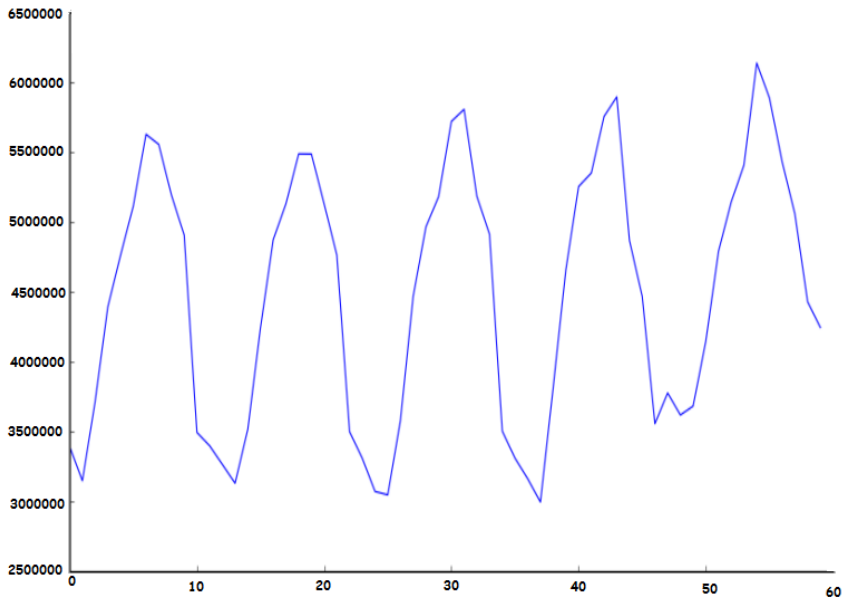


***Figura 15**: Gráfica de los pasajeros de Amadeus a Málaga (2012-2016)*

Configuración del entrenamiento y test

En este caso vamos a separar nuestro conjunto de datos en 2. Por una parte vamos a tener el conjunto de datos correspondientes al entrenamiento (67% de nuestras observaciones) y por otro lado vamos a tener el conjunto de datos correspondiente a la parte de pruebas (33% de las observaciones) para comprobar cómo actúa nuestro modelo al recibir datos con los cuales no había entrenado previamente.

```
train_size = int(len(dataset)*0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
```

A continuación, a través de la utilización de la función ***create_dataset*** vamos a generar los conjuntos para poder entrenar y testear nuestro modelo. Esta función toma dos argumentos, por una parte tenemos el conjunto de los datos que en nuestro caso, bien podría ser *"train"* o *"test"*. Por otra parte toma como argumento un valor denominado ***look_back***, este valor se corresponde con el número de puntos anteriores utilizados para la predicción del siguiente dato, en este caso hemos utilizado 1.

```
def create_dataset(dataset,look_back=1):
    dataX, dataY = [],[]
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back),0]
        dataX.append(a)
        dataY.append(dataset[i + look_back,0])
    return np.array(dataX), np.array(dataY)
```

Tal y como se puede observar en la *figura superior,* lo que nuestra función devuelve son 2 arrays, unos contendrá la información del estado en el instante ***t*** y en el otro array se almacenará la información para el instante ***t+1***.

El uso de esta función en nuestro se ve reflejada en la *figura inferior,* donde generamos los conjuntos de entrenamiento y pruebas de los que se hablaron al inicio de este apartado.

```
trainX, trainY = create_dataset(train,look_back)
testX, testY = create_dataset(test,look_back)
```

Creación del modelo: Perceptrón multicapa

Para la generación de nuestro ***perceptrón multicapa*** vamos a utilizar una librería llamada *Keras*, librería la cual ya tiene implementada las distintas herramientas que vamos a necesitar para la creación de modelos basados en neuronas. En este caso vamos a generar una red bastante sencilla con una entrada, como capa oculta tendremos un conjunto de 8 neuronas y una capa de salida.

```
model = Sequential()
model.add(Dense(8, input_dim=look_back,activation='relu'))
model.add(Dense(1))
model.compile(loss='mean_squared_error',optimizer='adam')
model.fit(trainX,trainY,epochs=200,batch_size=2,verbose=2)
```

Para compilar nuestro modelo vamos a utilizar el *Error cuadrático medio*, el cual va a medir el promedio de los errores al cuadrado entre el estimador y lo que se estima.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$$

***Figura 16**:* Fórmula para el cálculo de la MSE
https://en.wikipedia.org/wiki/Mean_squared_error

En el caso del optimizador utilizado hemos decidido utilizar ***Adam (Adaptative Moment Estimation)*** por los resultados que ofrecía. No vamos a entrar mucho en detalle en como este optimizador funciona, ya que, el uso de un optimizador u otro suele basarse en el método de prueba y error. Para más información revísese el enlace adjuntado en el apartado de *Referencias*.

Hablando del entrenamiento de nuestro modelo vamos a hablar de 2 variables que van a ser relevantes:

- **Epochs:** Es un ciclo de entrenamiento completo sobre el conjunto de datos,
  es decir, una pasada por el conjunto de datos tanto en sentido ascendente (t,t+1,t+2...) como uno descendente (t+k,t+(k-1)…)

- **Batch_size:** Número de datos que vamos a utilizar en un entrenamiento *"epoch"*

En nuestro modelo, el número de epochs será 200 y nuestro batch_size será igual a 2.

Estudio de los resultados

En este apartado vamos a comprobar que tan buena es nuestra aproximación con este modelo. Puntualizar que en la figura 17 cada color representa algo diferente:

- Verde: Predicciones sobre el conjunto de datos de entrenamiento
- Rojo: Predicciones sobre el conjunto de datos de testeo
- Azul: Datos originales

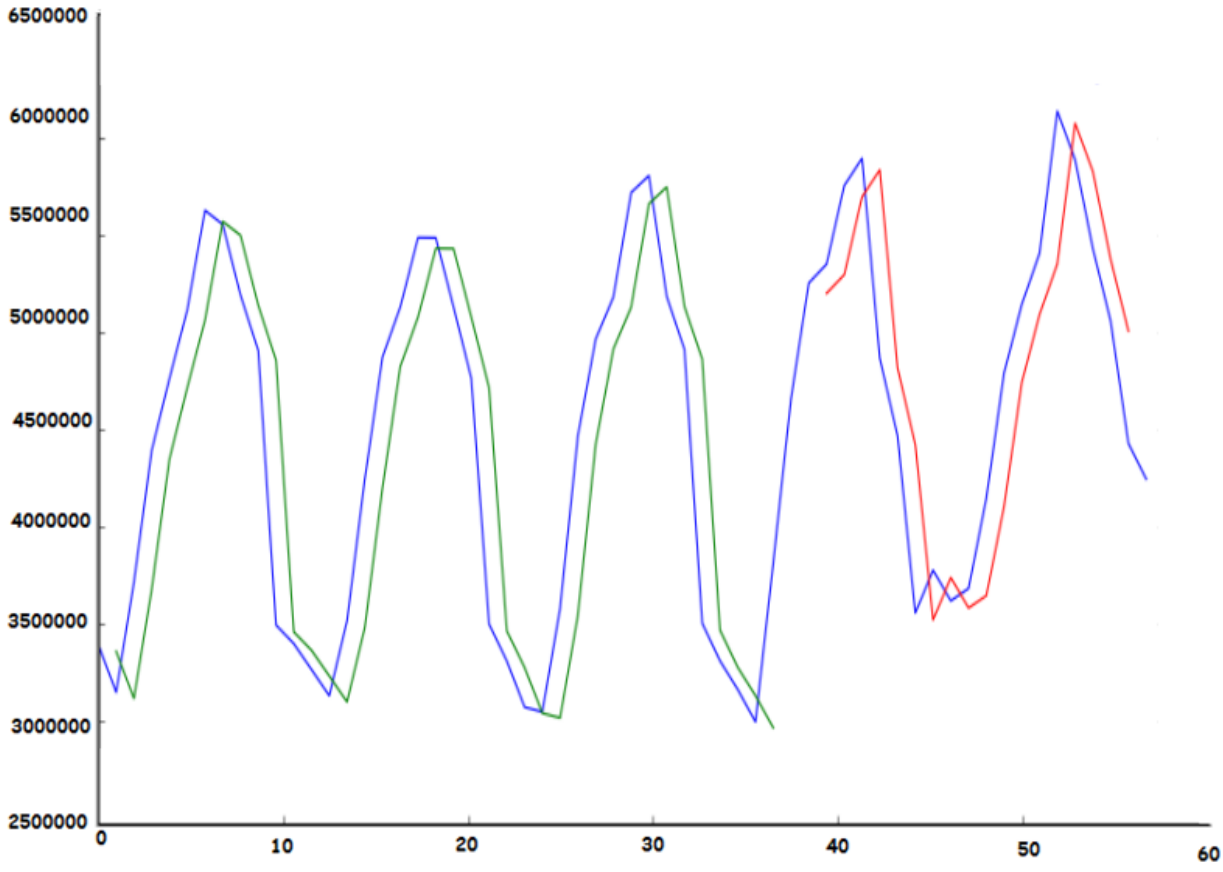


***Figura 17**: Resultados predicción modelo MLP*

Además vamos a tener los valores de RSME tanto para el apartado del entrenamiento como para el de testeo de tal manera que podremos comparar los resultados obtenidos con los demás modelos de predicción que estamos implementando en este proyecto y así poder cotejar cuál de estos es la mejor opción para nuestro caso.

Resultados RSME

Puntuación entrenamiento:	554855.38 RMSE
Puntuación testeo:	494596.54 RMSE

***Figura 18**: Resultados RSME para MLP*

3.3 Red LSTM

3.3.1 ¿Qué es una red LSTM?

Del inglés LSTM significa ***Long Short-Term Memory***, LSTM es una red neuronal recurrente, así que comencemos explicando que es una red neuronal recurrente (RNN)

3.3.2 RNN

3.3.2.1 ¿Qué es una RNN?

A diferencia de las redes neuronales en las cuales asumimos que todas las entradas y salidas son independientes las unas de las otras, en las redes neuronales recurrentes (***RNN***) las salidas dependen de las computaciones previas. Otra forma de entender esto es que las *redes neuronales recurrentes* poseen una “memoria” que captura de lo que ha sido calculado hasta el momento, sin embargo, típicamente no va a poder capturas muchos instantes pasados.

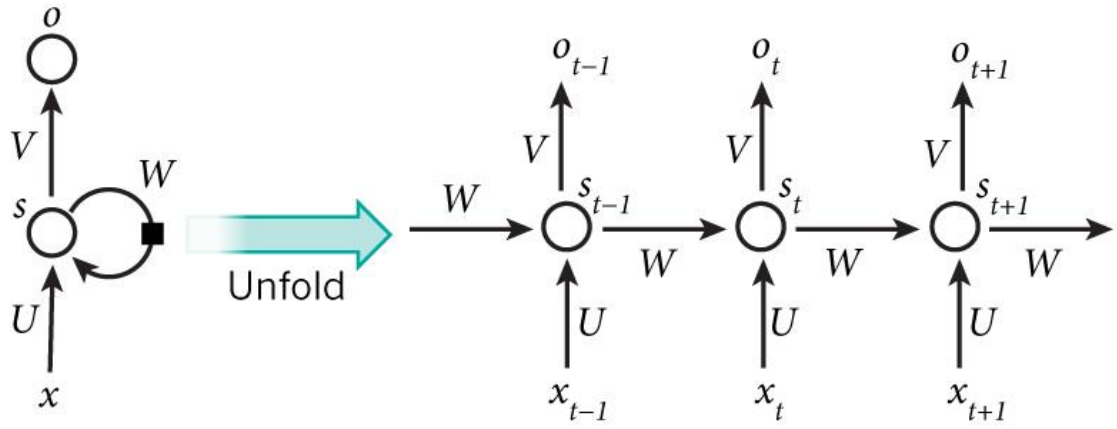


Figura 19*: Despliegue red RNN*

http://d3kbpzbmcynnmx.cloudfront.net/wp-content/uploads/2015/09/rnn.jpg

En el diagrama de la *Figura 19* se puede apreciar el despliegue de una *red neuronal recurrente* en una red completa.

Explicando las variables que aparecen en la figura:

- x_t: Esta variable se corresponde con el valor de entrada en el instante *t*.
- s_t: Corresponde al estado oculto en el instante t. Esta variable podría corresponderse con la *"memoria"* de la red ya que captura lo que ocurrió en todos los instantes anteriores. s_t es calculado en base a s_{t-1} y a la entrada del instante actual, es decir, $s_t = f(Ux_t + Ws_{t-1})$.
- o_t: Es la salida en el instante *t*. Esta salida está calculada únicamente en base a la *"memoria"* de nuestra red en el instante t.

3.3.2.2 ¿Qué problemas tienen las RNN?

El motivo por el que hemos ignorado las RNN en su forma básica para nuestras predicciones es debido al ***gradiente de desaparición***, sin entrar en detalle en las fórmulas matemáticas que esto lleva asociado, únicamente nos interesa conocer es el hecho de que debido a este problema el *gradiente de contribución* de los datos de instantes "alejados" pasan a ser cero y los estados de esos instantes dejan de contribuir al aprendizaje. Sin embargo, las LSTM solucionan este problema.

3.3.3 LSTM

Las LSTMs son *redes neuronales recurrentes* diseñadas para combatir el ***gradiente de desaparición*** a través de un mecanismo de puertas. Con la finalidad de entender esto, veamos como una LSTM calcula el estado oculto s_t:

$$i = \sigma(x_t U^i + s_{t-1} W^i)$$

$$f = \sigma(x_t U^f + s_{t-1} W^f)$$

$$o = \sigma(x_t U^o + s_{t-1} W^o)$$

$$g = \tanh(x_i U^g + s_{t-1} W^g)$$

$$c_t = c_{t-1} \circ f + g \circ i$$

$$s_t = \tanh(c_t) \circ o$$

Nota: El operador ∘ significa *multiplicación por elementos*.

Todas estas operaciones pueden parecer complicadas, sin embargo, esta es solo otra forma de computar el *estado oculto*. En las RNNs, el estado oculto lo computábamos como $s_t = f(Ux_t + Ws_{t-1})$ como ya se explicó anteriormente, ahora una *LSTM* va a realizar exactamente lo mismo pero de una manera diferente. Si tratásemos a la las unidades de la *LSTM* como una caja negra tendríamos algo como:

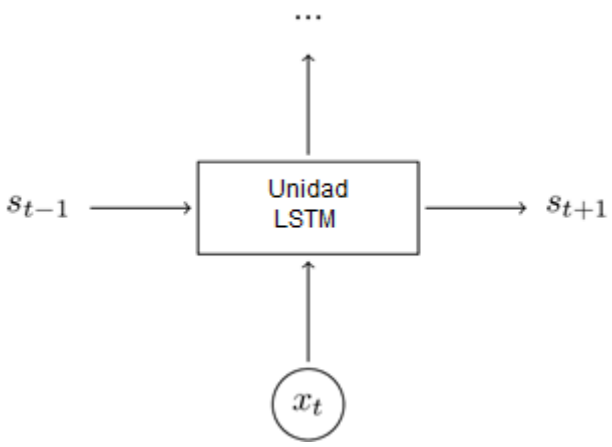


***Figura 20**: Modelo LSTM simplificado*

Donde dado una entrada y el estado oculto anterior se computa el siguiente estado oculto.

-$\boldsymbol{i, f, o}$: Estas variables representan las puertas de entrada, olvido y salida, respectivamente. Tal y como se puede apreciar en las ecuaciones ofrecidas con anterioridad estas son exactamente iguales solo que operan con diferentes matrices. El motivo por el que reciben el nombre de puertas es debido a que la función *sigmoid* reemplaza los valores de los vectores que están entre 0 y 1 multiplicándolos con otro vector el cual definamos de tal manera que se defina cuanta información del vector deseamos "dejar pasar".

-$\boldsymbol{g}$: Es el candidato para ser el estado oculto calculado en base a la entrada actual y el estado oculto anterior. La ecuación es exactamente la utilizada en ***vanilla RNN***, sin embargo, en lugar de tomar a *g* como nuestro nuevo estado oculto, usaremos la puerta de entrada para tomar "algo de él".
-$\boldsymbol{c_t}$: Es la memoria interna de nuestra unidad. Una combinación de la memoria previa c_{t-1} multiplicada por la puerta del olvido y el valor del estado oculto recién calculado multiplicado por el valor de la puerta de entrada.

Ya una vez dada la memoria c_t, finalmente computamos el estado oculto de salido s_t, multiplicando la memoria con la puerta de salida.

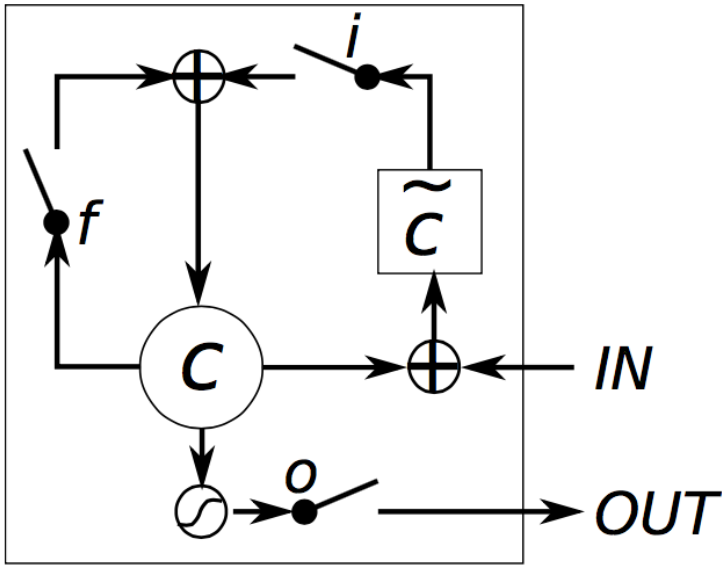


***Figura 21**: Funcionamiento puertas LSTM*
http://d3kbpzbmcynnmx.cloudfront.net/wp-content/uploads/2015/10/Screen-Shot-2015-10-23-at-10.00.55-AM.png

3.3.4 Implementación

La implementación que vamos a mostrar a continuación está inspirada en una implementación la cual se encuentra disponible en el apartado de *Referencias*. Una vez comentado esto pasemos a hablar sobre la implementación y el cómo se ha llevado está a cabo. En este caso, y al igual que hicimos con la implementación del perceptrón multicapa, vamos a utilizar ***TensorFlow*** como tecnología para la implementación del modelo predictivo y para ello utilizaremos de nuevo el framework ***Keras***, el cuál simplificara en gran medida la complejidad del código.

Tratamiento de datos

El tratamiento de datos es equivalente al utilizado en *ARIMA*, sin embargo, tal y como ocurrió con el perceptrón multicapa los índices temporales no son viables con la lógica de *Tensorflow*, así que hubo que eliminarlos.

La gráfica resultante por lo tanto es totalmente equivalente a la presentada en el modelo ***MLP***

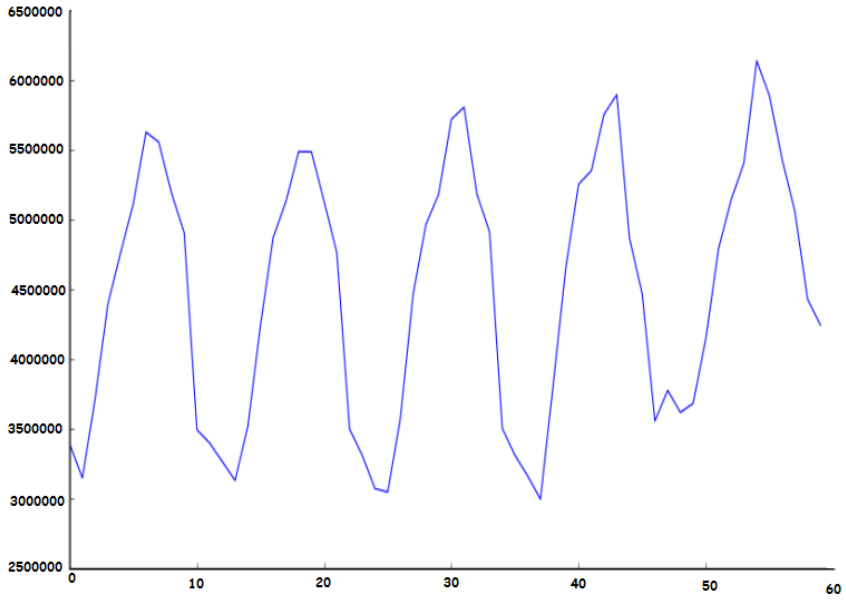


***Figura 22**: Gráfica de los pasajeros de Amadeus a Málaga (2012-2016)*

Normalización de los datos

Debido a que las funciones LSTM son sensibles a la escala de los datos de entrada, específicamente cuando las funciones de activación sigmoid o tanh son usadas. Es buena práctica el reescalar los datos a un rango comprendido entre 0 y 1 (***normalización**).* En nuestro caso esta normalización se va a realizar mediante el uso de la librería *"sklearn.preprocessing import MinMaxScaler"*

```
normalizador = MinMaxScaler(feature_range=(0, 1))
dataset = normalizador.fit_transform(dataset)
```

Configuración del entrenamiento y test

En este apartado vamos a separar nuestro conjunto de datos en 2. Por una parte vamos a tener el conjunto de datos correspondientes al entrenamiento (67% de nuestras observaciones) y por otro lado vamos a tener el conjunto de datos correspondiente a la parte de pruebas (33% de las observaciones) para comprobar cómo actúa nuestro modelo al recibir datos con los cuales no había entrenado previamente.

```
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size], dataset[train_size:len(dataset)]
```

A continuación, a través de la utilización de la función ***create_dataset*** vamos a generar los conjuntos para poder entrenar y testear nuestro modelo. Esta función toma dos argumentos, por una parte tenemos el conjunto de los datos que en nuestro caso, bien podría ser *"train"* o *"test"*. Por otra parte toma como argumento un valor denominado ***look_back***, este valor se corresponde con el número de puntos anteriores utilizados para la predicción del siguiente dato, en este caso hemos utilizado 1.

```
def create_dataset(dataset,look_back=1):
    dataX, dataY = [],[]
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back),0]
        dataX.append(a)
        dataY.append(dataset[i + look_back,0])
    return np.array(dataX), np.array(dataY)
```

Tal y como se puede observar en la *figura superior,* lo que nuestra función devuelve son 2 arrays, unos contendrá la información del estado en el instante ***t*** y en el otro array se almacenará la información para el instante ***t+1***.

El uso de esta función en nuestro se ve reflejada en la *figura inferior,* donde generamos los conjuntos de entrenamiento y pruebas de los que se hablaron al inicio de este apartado.

```
trainX, trainY = create_dataset(train,look_back)
testX, testY = create_dataset(test,look_back)
```

En estos momentos nuestros datos tienen la siguiente forma ***[datos,características]***, sin embargo, nuestro modelo LSTM nos va a demandar una estructura como la siguiente **[datos, número de instantes, características]**. Así que de esta manera, vamos a reajustar la forma de nuestros datos tal y como se muestra en la *figura inferior*.

```
trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
```

Construcción del modelo LSTM

Tal y como se comentó, para la creación del modelo LSTM vamos a utilizar ***Keras***. En nuestro caso vamos a generar una red con una capad de entrada, una capa oculta con 4 bloques LSTM o neuronas, y una capa de salida que devuelve un único valor en la predicción. La función de activación usada es la predeterminada, *"sigmoid"*. Además la red será entrenada por 100 epochs tal y como se hizo en la red *MLP*.

```
model = Sequential()
model.add(LSTM(4, input_shape=(1, look_back)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=100, batch size=1, verbose=2)
```

Estudio de los resultados

En este apartado vamos a comprobar que tan buena es nuestra aproximación con el modelo LSTM. Puntualizar que en la *figura 38* cada color representa algo diferente:

- Verde: Predicciones sobre el conjunto de datos de entrenamiento
- Rojo: Predicciones sobre el conjunto de datos de testeo
- Azul: Datos originales

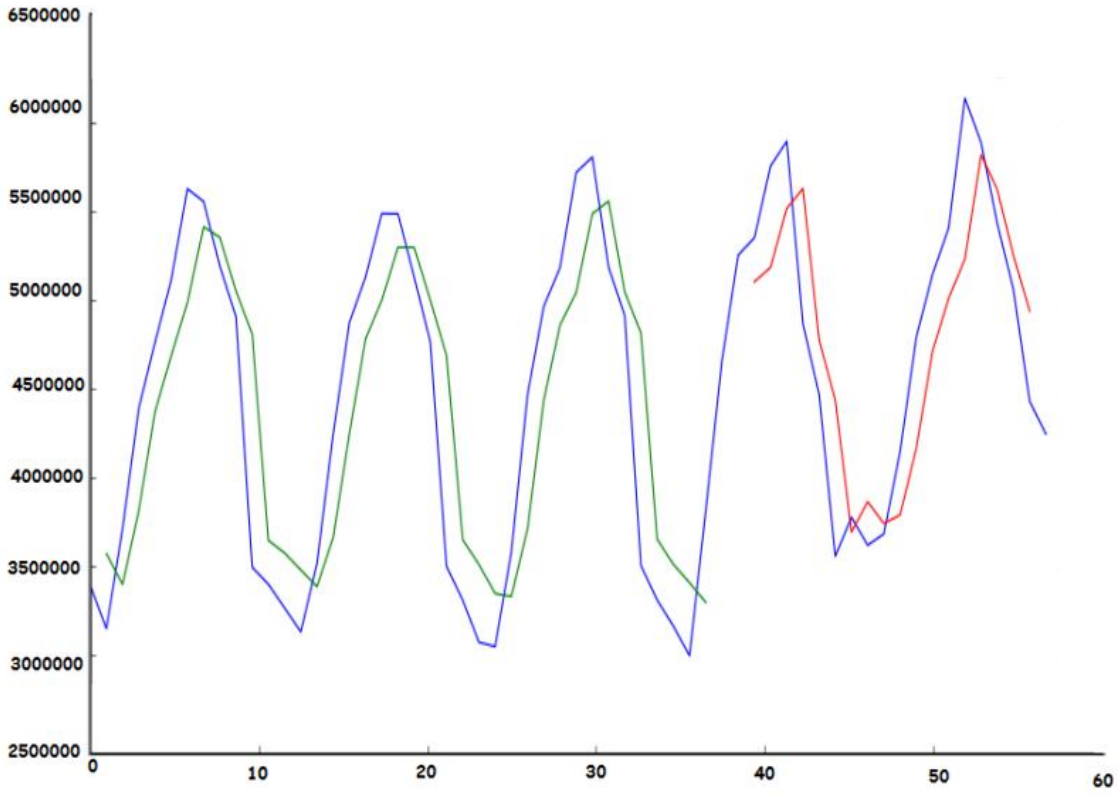


***Figura 22**: predicción datos del modelo LSTM*

Además vamos a tener los valores de RSME tanto para el apartado del entrenamiento como para el de testeo de tal manera que podremos comparar los resultados obtenidos con los demás modelos de predicción que estamos implementando en este proyecto y así poder cotejar cuál de estos es la mejor opción para nuestro caso.

Resultados RSME

Puntuación entrenamiento:	528315.98 RMSE
Puntuación testeo:	475759.49 RSME

***Figura 23**: Resultados RSME para las predicciones del modelo LSTM*

3.3.5 LSTM con mayor feed

En el caso anterior las predicciones estaban realizadas en base a la información ocurrida en el instante anterior $t-1$, sin embargo, esto no parece óptimo ya que estaríamos desaprovechando la potencia de las LSTM, la cual reside principalmente en su capacidad de poder tener en cuenta información pasada consiguiendo que esta tenga siempre una relevancia en nuestras predicciones. En este caso hemos decidido tomar un $lookback = 3$, el porqué de esto es para prevenir las zonas de "ruptura estacionaria", y darle un feed no deseado a nuestra red neuronal, es decir, en aquellas zonas de cambio de tendencia encontramos datos ascendentes a la izquierda y descendentes a la derecha, por lo tanto vamos a buscar minimizar este impacto lo mayormente posible. Lo comentado se puede observar gráficamente en la *Figura 24:*

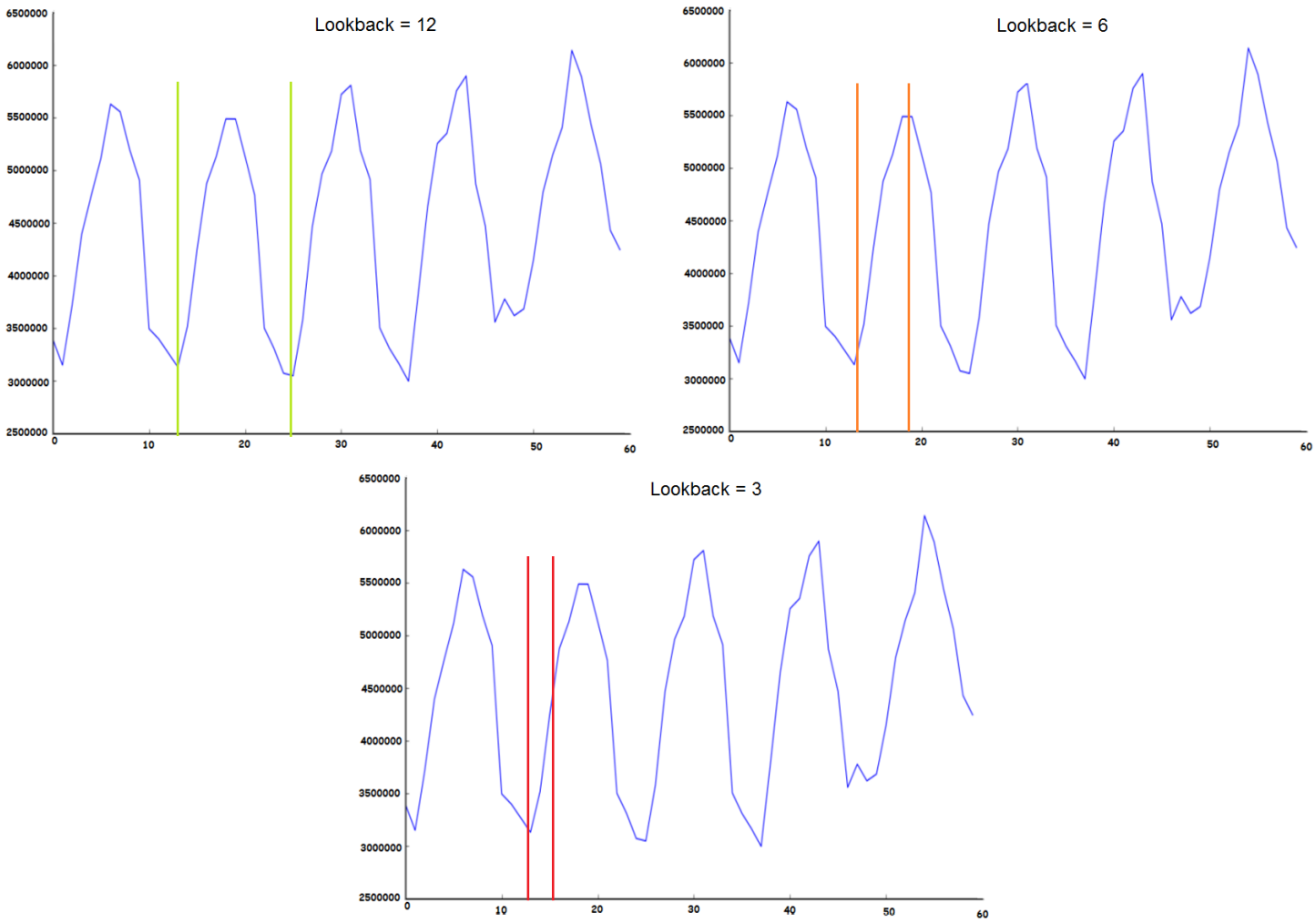


***Figura 24**: Diferentes lookbacks para la gráfica de los pasajeros de Amadeus a Málaga (2012-2016)*

Como se puede intuir, la única modificación que fue necesaria para conseguir esto fue reajustar el valor de lookback en nuestro programa seteando $lookback = 3$. Veamos ahora los resultados obtenidos con esta mejora.

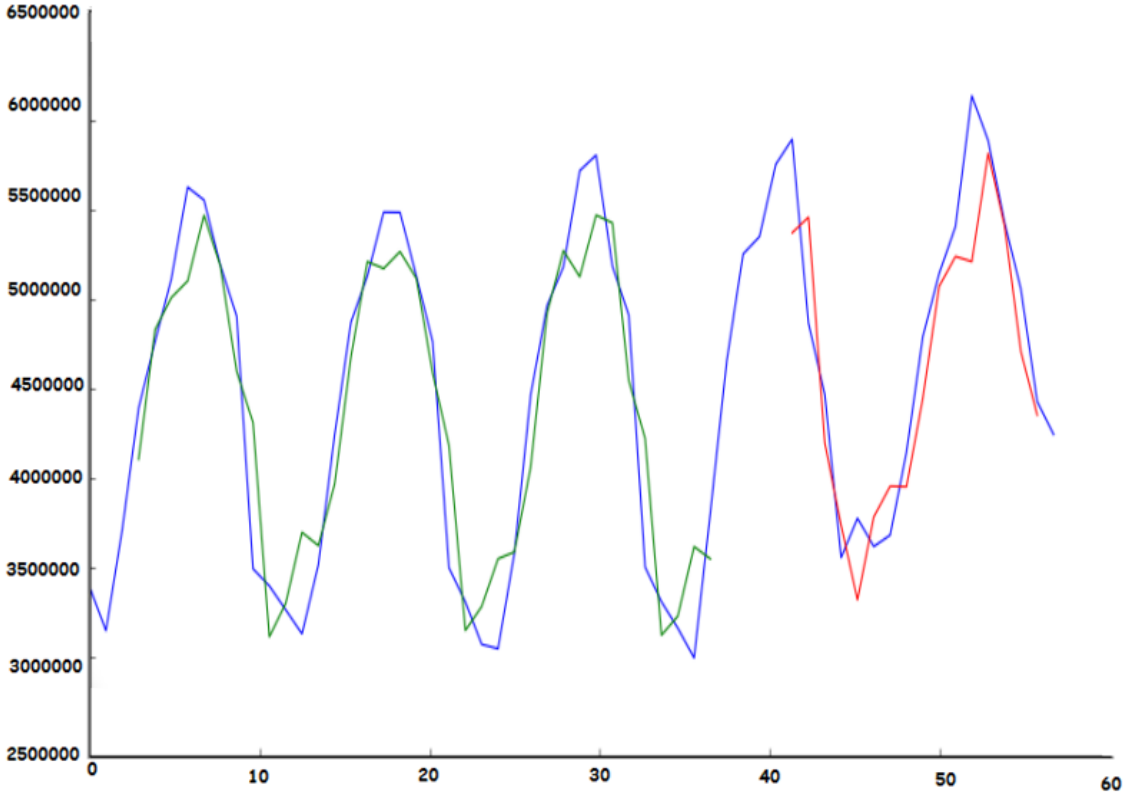


***Figura 25**: Predicción LSTM con mejora en feed*

A simple vista las mejoras son visibles, sin embargo, comprobemos que estamos en lo cierto mediante el cálculo de los RMSE:

Resultados RSME

Puntuación entrenamiento:	353334.66 RMSE
Puntuación testeo:	372737.62 RMSE

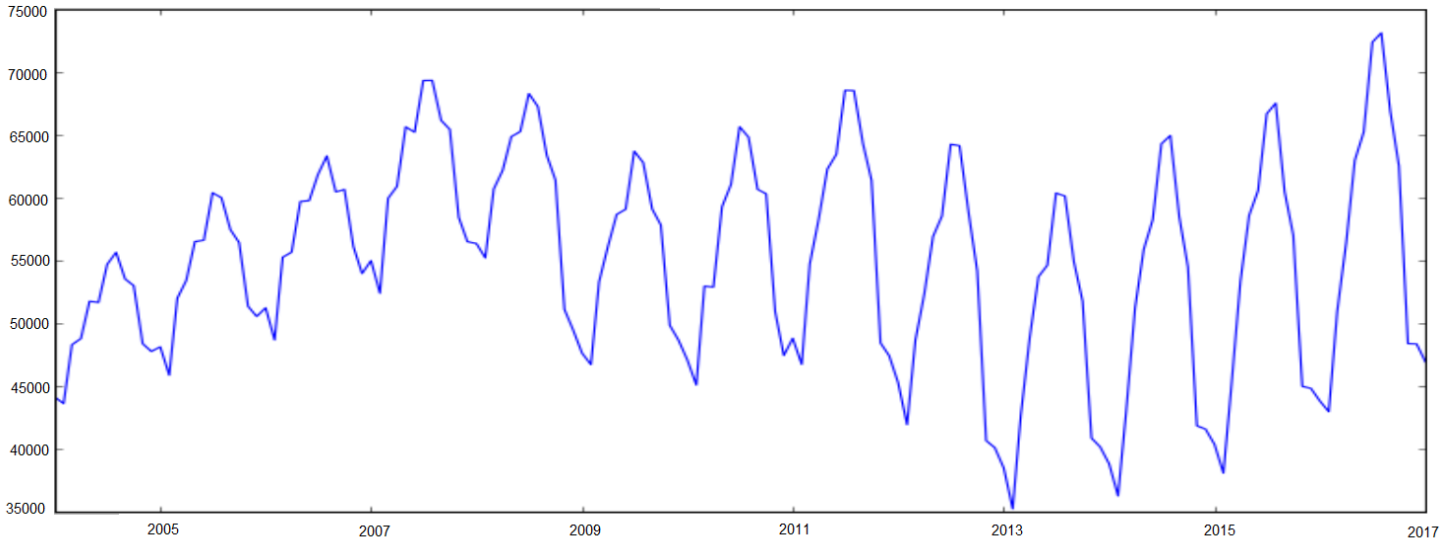
***Figura 26**: Resultados RSME para la predicción de LSTM con la mejora en el feed*

4. Otros datos

Además de la base de datos de *Amadeus*, se decidió cotejar lo dicho anteriormente con otro tipo de datos tales como ***los vuelos de aena*** y ***los pasajeros de Aena***.

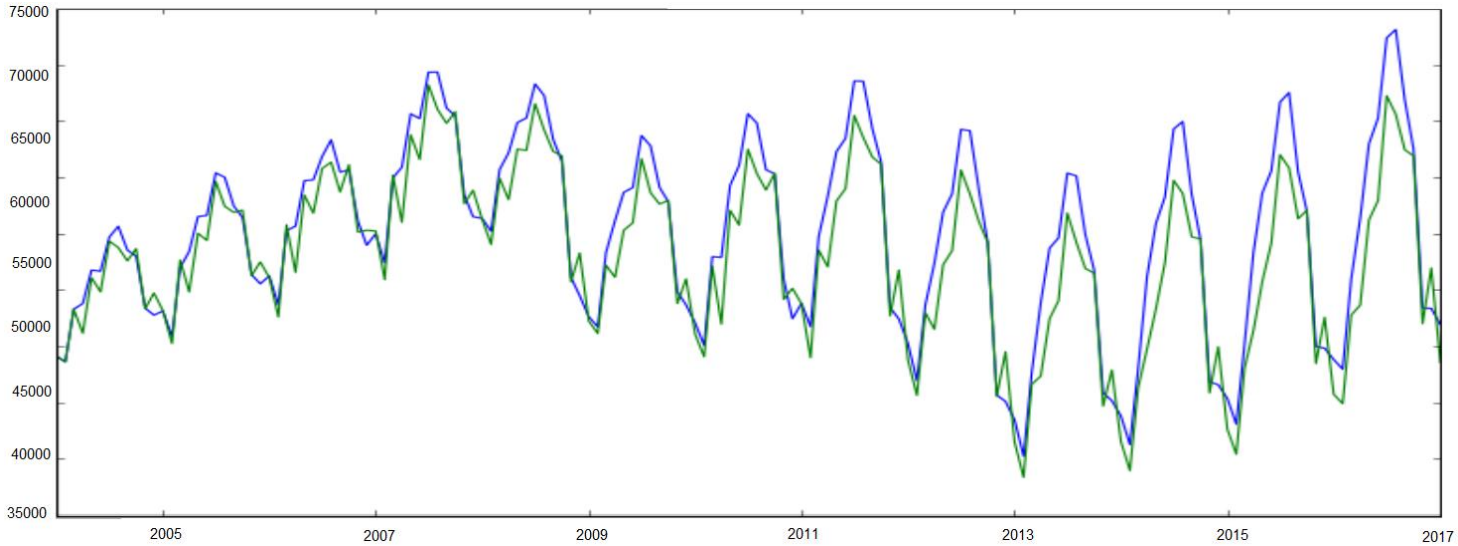
4.1 Vuelos de Aena

Datos originales:

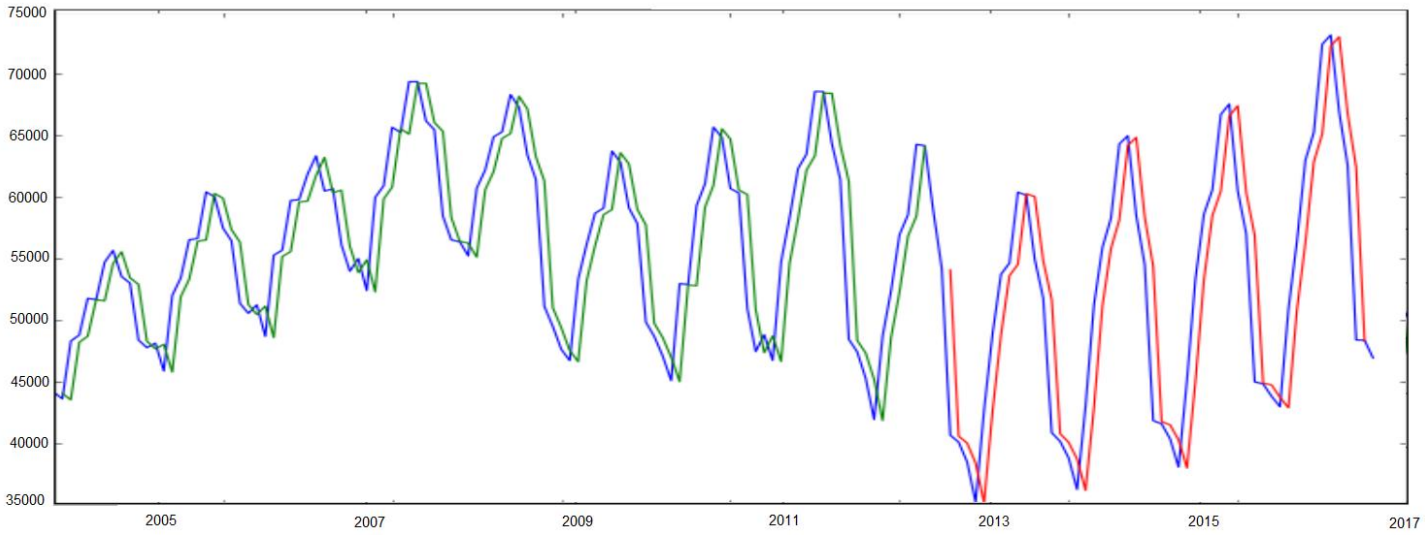


Predicciones

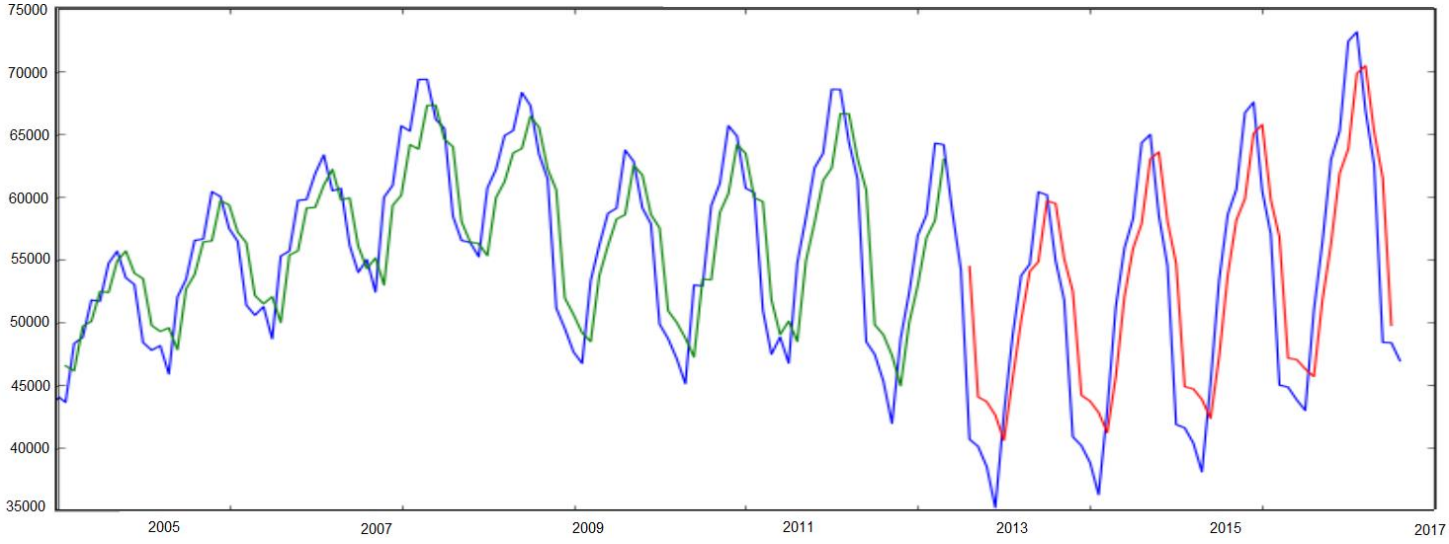
ARIMA



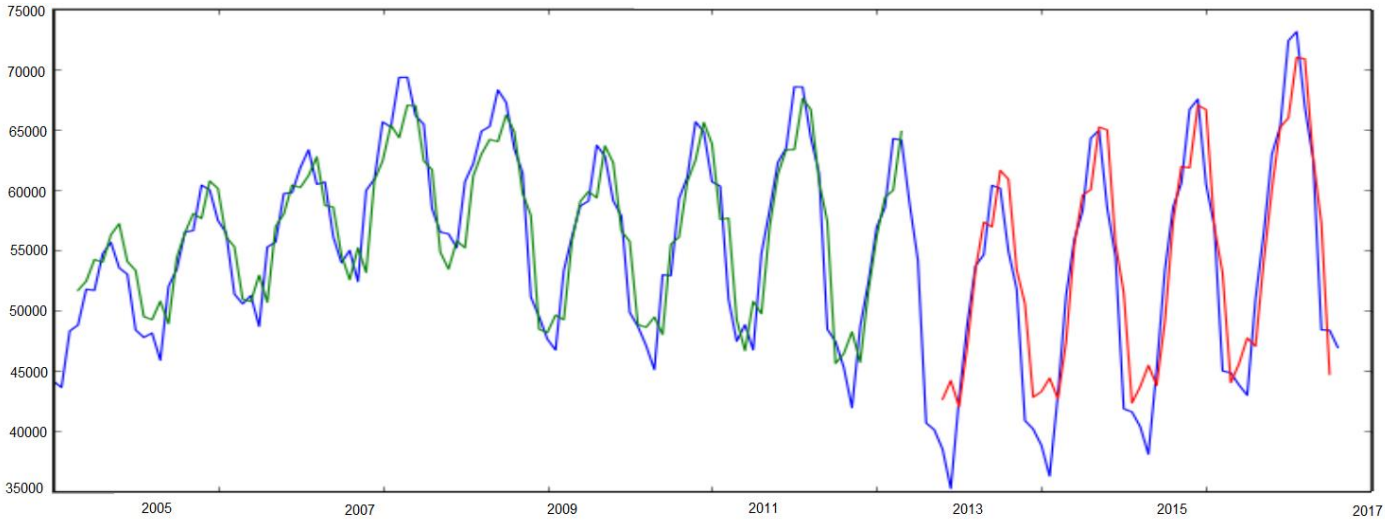
MLP


75000
70000
65000
60000
55000
50000
45000
40000
35000
2005
2007
2009
2011
2013
2015
2017

LSTM


75000
70000
65000
60000
55000
50000
45000
40000
35000
2005
2007
2009
2011
2013
2015
2017

LSTM mejora

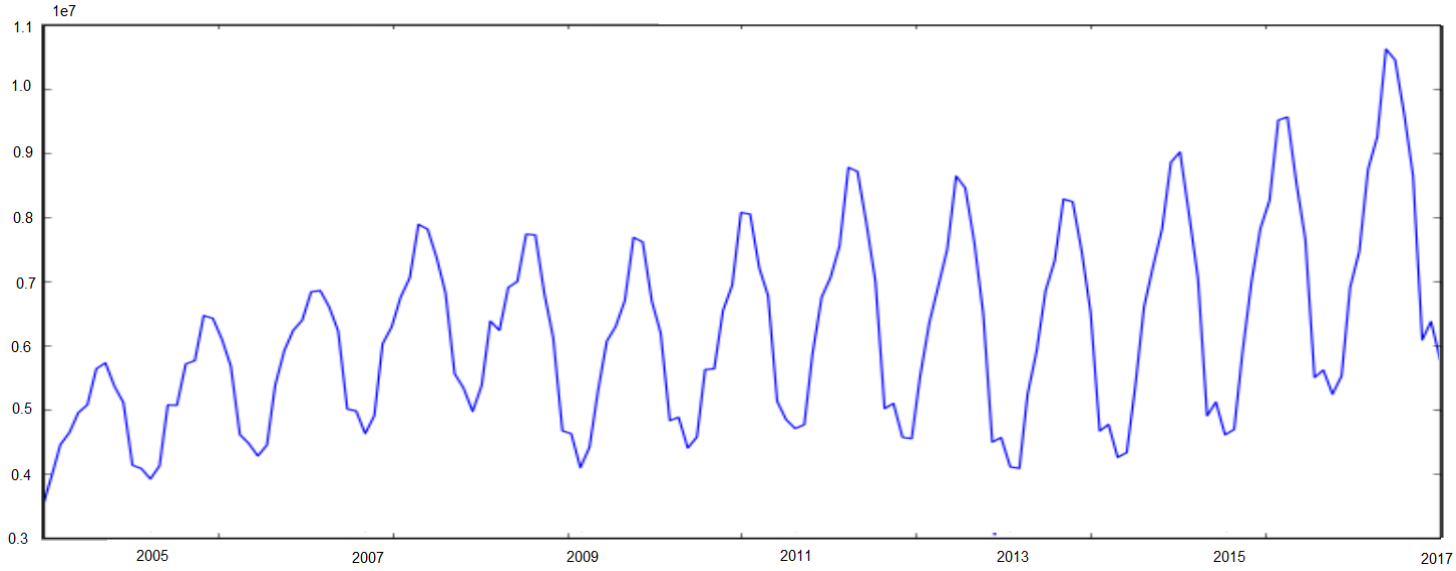


Los resultados de la RSME son los siguientes:

Modelo	*RSME*
ARIMA	*3309.52*
MLP	*5927.73*
LSMT	*5787.50*
LSMT (mejora en el feed)	*4439.83*

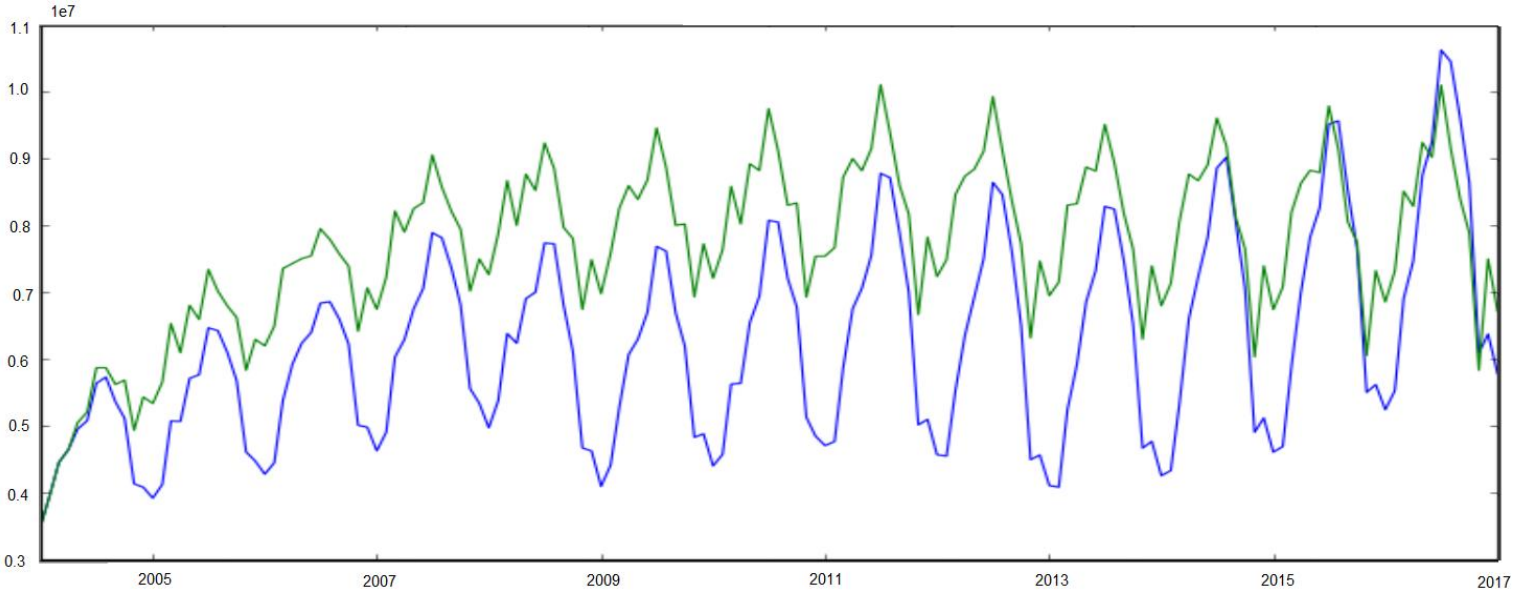
4.2. Pasajeros de Aena

Datos originales:

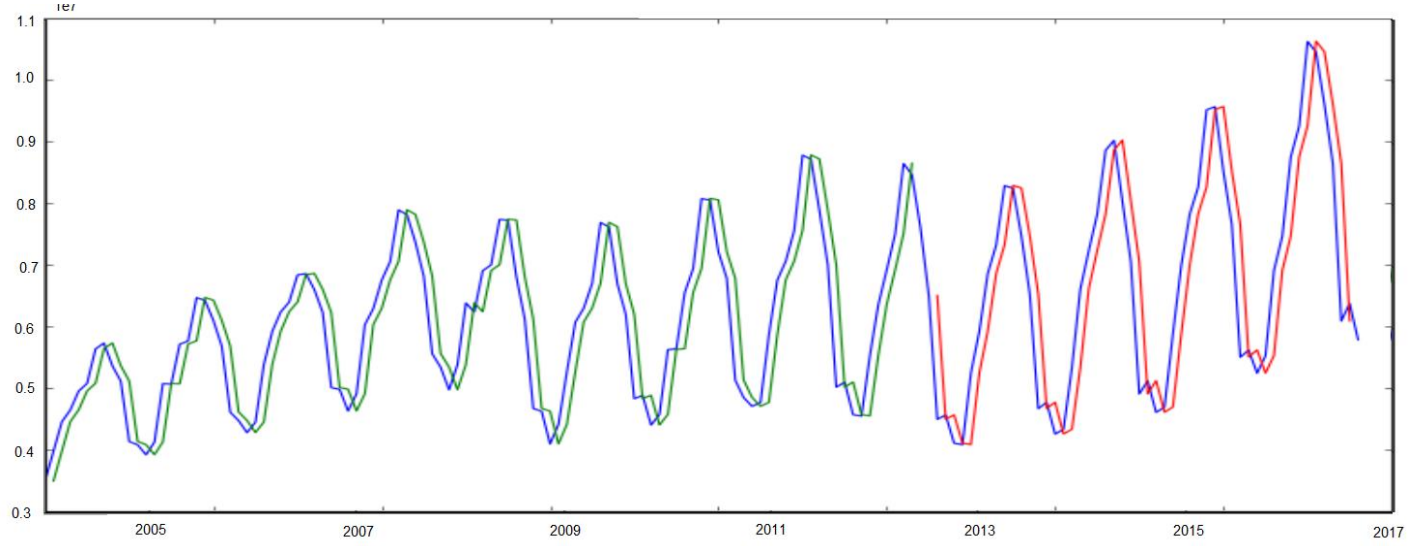


Predicciones:

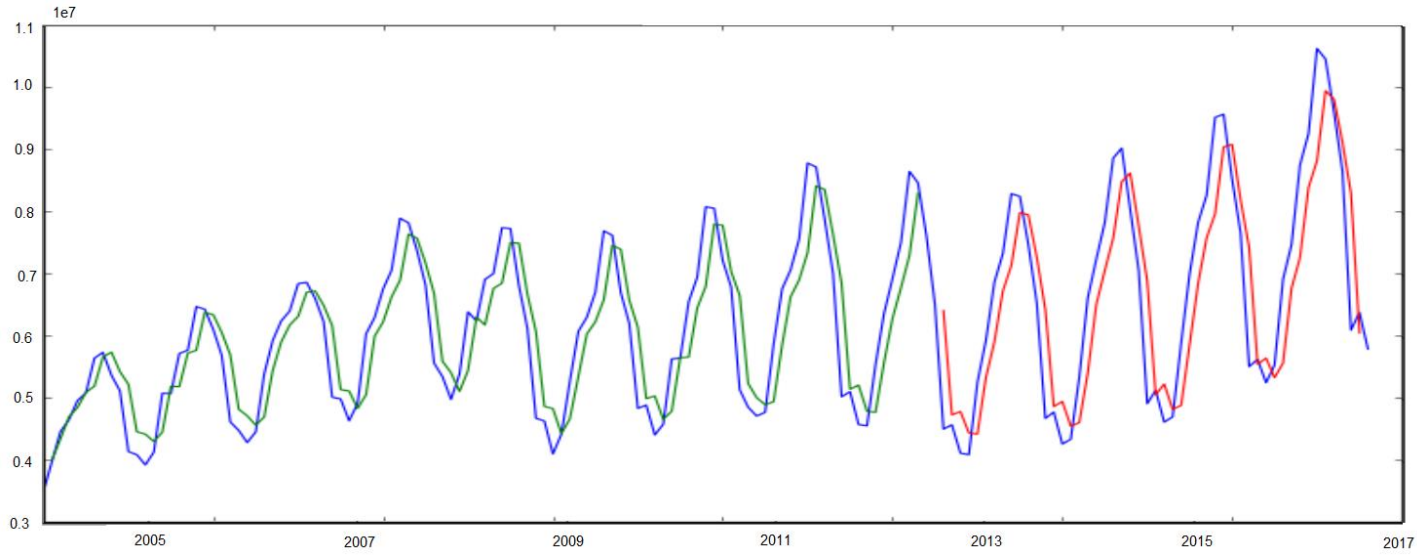
ARIMA



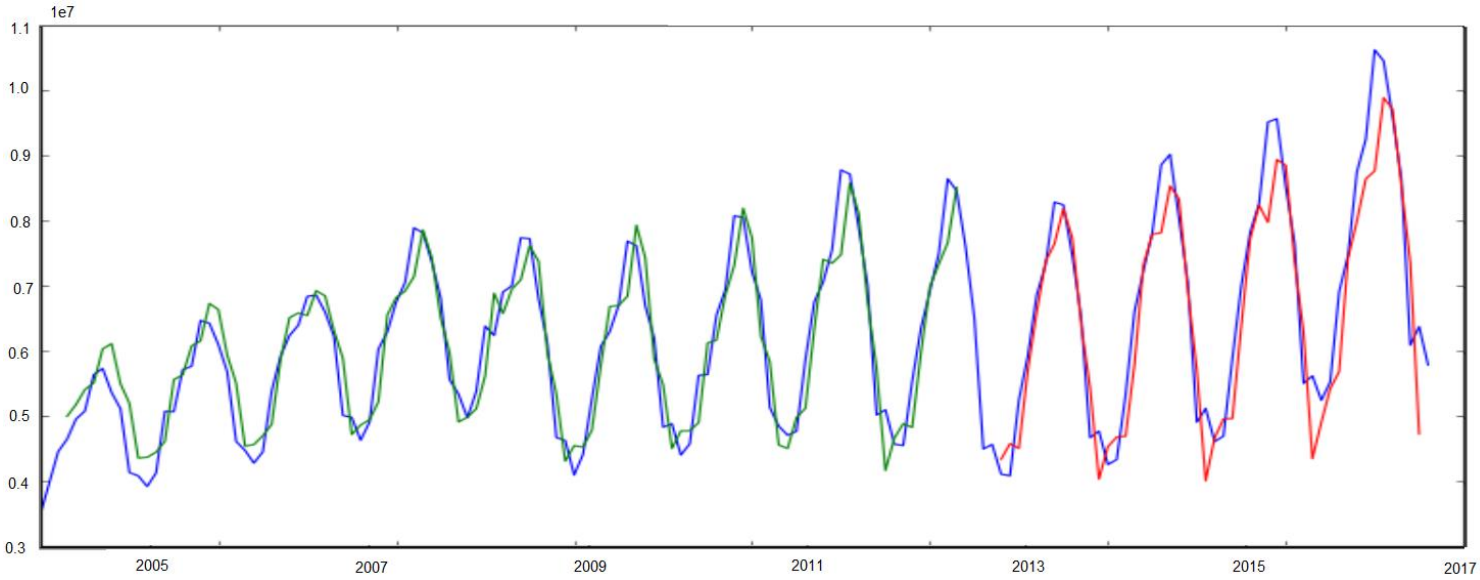
MLP


1e7
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
2005
2007
2009
2011
2013
2015
2017

LSTM


1e7
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
2005
2007
2009
2011
2013
2015
2017

LSTM mejora



Los resultados del RSME son los siguientes:

Modelo	*RSME*
ARIMA	*174434.85*
MLP	*1007230.63*
LSMT	*975317.48*
LSMT (mejora en el feed)	*712989.32*

5. Conclusiones

Una vez llevadas a cabo todas estas pruebas con los diferentes modelos hemos podido obtener los siguientes datos.

Tal y como se puede observar *ARIMA,* a pesar de no tener ningún aprendizaje asociado suele obtener unos mejores resultados que MLP. Esto se debe mayormente a que MLP no tiene memoria y por lo tanto olvida en cada iteración las entradas anteriores. Sin embargo, en el caso de LSMT, no siempre es ARIMA quien se lleva la victoria además, debido a que LSMT presenta varias variantes donde se puede mejorar su rendimiento, LSTM ha conseguido ser nuestro modelo ganador en 3 de las 4 pruebas.

Por lo tanto, si bien es necesario el estudio puntual de cada caso para determinar qué modelo de predicción se va a utilizar, LSTM siempre va a ser uno de los candidatos más fiables en la mayoría de los casos.

6. Referencias

- http://www.seanabu.com/2016/03/22/time-series-seasonal-ARIMA-model-in-python/
- https://medium.com/@erikhallstrm/hello-world-rnn-83cd7105b767
- https://arxiv.org/pdf/1506.00019.pdf
- http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/
- https://help.xlstat.com/customer/es/portal/articles/2062303
- https://people.duke.edu/~rnau/411arim.htm
- https://www.r-bloggers.com/stationarity/
- https://www.otexts.org/fpp/8/3
- https://www.otexts.org/fpp/8/5
- http://machinelearningmastery.com/arima-for-time-series-forecasting-with-python/
- https://www.analyticsvidhya.com/blog/2016/02/time-series-forecasting-codes-python/
- https://es.wikipedia.org/wiki/Prueba_de_Dickey-Fuller
- http://mourafiq.com/2016/05/15/predicting-sequences-using-rnn-in-tensorflow.html
- https://medium.com/@alexrachnog/neural-networks-for-algorithmic-trading-part-one-simple-time-series-forecasting-f992daa1045a
- https://stackoverflow.com/questions/40646783/predicting-time-series-values-with-mlp-and-tensorflow
- https://nicholastsmith.wordpress.com/2016/04/20/stock-market-prediction-using-multi-layer-perceptrons-with-tensorflow/
- http://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/
- https://stats.stackexchange.com/questions/242787/how-to-interpret-root-mean-squared-error-rmse-vs-standard-deviation
- https://www.quora.com/How-do-I-interpret-the-results-in-an-augmented-Dickey-Fuller-test
- https://es.wikipedia.org/wiki/Perceptr%C3%B3n_multicapa
- http://perso.wanadoo.es/alimanya/funcion.htm
- http://machinelearningmastery.com/time-series-prediction-with-deep-learning-in-python-with-keras/
- https://es.wikipedia.org/wiki/Error_cuadr%C3%A1tico_medio
- http://sebastianruder.com/optimizing-gradient-descent/index.html#adam
- https://en.wikipedia.org/wiki/Long_short-term_memory
- http://www.wildml.com/2015/10/recurrent-neural-network-tutorial-part-4-implementing-a-grulstm-rnn-with-python-and-theano/
- http://colah.github.io/posts/2015-08-Understanding-LSTMs/

- https://stackoverflow.com/questions/11788900/importerror-no-module-named-statsmodels
- http://relopezbriega.github.io/blog/2016/09/26/series-de-tiempo-con-python/
- http://danielhnyk.cz/predicting-sequences-vectors-keras-using-rnn-lstm/