ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

GRADO EN INGENIERÍA TELEMÁTICA

PLATAFORMA EN TIEMPO REAL BIG DATA DE PROCESAMIENTO DE SENSORES IOT PARA LA PREDICCIÓN DE DEMANDA EN UNA FRANQUICIA DE RESTAURACIÓN

Autor: José María Rodríguez Cano de Santayana

Director: Dr. David Contreras Bárcena

Madrid

Junio 2018

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Plataforma en Tiempo Real Big Data de procesamiento de sensores IoT para la predicción de la demanda de una franquicia de restauración

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2017/18 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido

tomada de otros documentos está debidamente referenciada.

Fdo.: José María Rodríguez Cano de Santayana Fecha: ……/ ……/ ……

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Dr. David Contreras Bárcena Fecha: ……/ ……/ ……

AUTORIZACIÓN PARA LA DIGITALIZACIÓN, DEPÓSITO Y DIVULGACIÓN EN RED DE PROYECTOS FIN DE GRADO, FIN DE MÁSTER, TESINAS O MEMORIAS DE BACHILLERATO

1º. Declaración de la autoría y acreditación de la misma.

El autor D. José María Rodríguez Cano de Santayana

DECLARA ser el titular de los derechos de propiedad intelectual de la obra: Plataforma en Tiempo Real Big Data de procesamiento de sensores IoT para la predicción de la demanda de una franquicia de restauración, que ésta es una obra original, y que ostenta la condición de autor en el sentido que otorga la Ley de Propiedad Intelectual.

2º. Objeto y fines de la cesión.

Con el fin de dar la máxima difusión a la obra citada a través del Repositorio institucional de la Universidad, el autor **CEDE** a la Universidad Pontificia Comillas, de forma gratuita y no exclusiva, por el máximo plazo legal y con ámbito universal, los derechos de digitalización, de archivo, de reproducción, de distribución y de comunicación pública, incluido el derecho de puesta a disposición electrónica, tal y como se describen en la Ley de Propiedad Intelectual. El derecho de transformación se cede a los únicos efectos de lo dispuesto en la letra a) del apartado siguiente.

3º. Condiciones de la cesión y acceso

Sin perjuicio de la titularidad de la obra, que sigue correspondiendo a su autor, la cesión de derechos contemplada en esta licencia habilita para:

a) Transformarla con el fin de adaptarla a cualquier tecnología que permita incorporarla a internet y hacerla accesible; incorporar metadatos para realizar el registro de la obra e incorporar "marcas de agua" o cualquier otro sistema de seguridad o de protección.
b) Reproducirla en un soporte digital para su incorporación a una base de datos electrónica, incluyendo el derecho de reproducir y almacenar la obra en servidores, a los efectos de garantizar su seguridad, conservación y preservar el formato.
c) Comunicarla, por defecto, a través de un archivo institucional abierto, accesible de modo libre y gratuito a través de internet.
d) Cualquier otra forma de acceso (restringido, embargado, cerrado) deberá solicitarse expresamente y obedecer a causas justificadas.
e) Asignar por defecto a estos trabajos una licencia Creative Commons.
f) Asignar por defecto a estos trabajos un HANDLE (URL *persistente).*

4º. Derechos del autor.

El autor, en tanto que titular de una obra tiene derecho a:

a) Que la Universidad identifique claramente su nombre como autor de la misma
b) Comunicar y dar publicidad a la obra en la versión que ceda y en otras posteriores a través de cualquier medio.
c) Solicitar la retirada de la obra del repositorio por causa justificada.
d) Recibir notificación fehaciente de cualquier reclamación que puedan formular terceras personas en relación con la obra y, en particular, de reclamaciones relativas a los derechos de propiedad intelectual sobre ella.

5º. Deberes del autor.

El autor se compromete a:

a) Garantizar que el compromiso que adquiere mediante el presente escrito no infringe ningún derecho de terceros, ya sean de propiedad industrial, intelectual o cualquier otro.
b) Garantizar que el contenido de las obras no atenta contra los derechos al honor, a la intimidad y a la imagen de terceros.
c) Asumir toda reclamación o responsabilidad, incluyendo las indemnizaciones por daños, que pudieran ejercitarse contra la Universidad por terceros que vieran infringidos sus derechos e intereses a causa de la cesión.

d) Asumir la responsabilidad en el caso de que las instituciones fueran condenadas por infracción de derechos derivada de las obras objeto de la cesión.

6º. Fines y funcionamiento del Repositorio Institucional.

La obra se pondrá a disposición de los usuarios para que hagan de ella un uso justo y respetuoso con los derechos del autor, según lo permitido por la legislación aplicable, y con fines de estudio, investigación, o cualquier otro fin lícito. Con dicha finalidad, la Universidad asume los siguientes deberes y se reserva las siguientes facultades:

- La Universidad informará a los usuarios del archivo sobre los usos permitidos, y no garantiza ni asume responsabilidad alguna por otras formas en que los usuarios hagan un uso posterior de las obras no conforme con la legislación vigente. El uso posterior, más allá de la copia privada, requerirá que se cite la fuente y se reconozca la autoría, que no se obtenga beneficio comercial, y que no se realicen obras derivadas.
- La Universidad no revisará el contenido de las obras, que en todo caso permanecerá bajo la responsabilidad exclusive del autor y no estará obligada a ejercitar acciones legales en nombre del autor en el supuesto de infracciones a derechos de propiedad intelectual derivados del depósito y archivo de las obras. El autor renuncia a cualquier reclamación frente a la Universidad por las formas no ajustadas a la legislación vigente en que los usuarios hagan uso de las obras.
- La Universidad adoptará las medidas necesarias para la preservación de la obra en un futuro.
- La Universidad se reserva la facultad de retirar la obra, previa notificación al autor, en supuestos suficientemente justificados, o en caso de reclamaciones de terceros.

Madrid, a 7 de Julio de 2018

ACEPTA

Fdo……

Motivos para solicitar el acceso restringido, cerrado o embargado del trabajo en el Repositorio Institucional:

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

GRADO EN INGENIERÍA TELEMÁTICA

PLATAFORMA EN TIEMPO REAL BIG DATA DE PROCESAMIENTO DE SENSORES IOT PARA LA PREDICCIÓN DE DEMANDA EN UNA FRANQUICIA DE RESTAURACIÓN

Autor: José María Rodríguez Cano de Santayana

Director: Dr. David Contreras Bárcena

Madrid

Junio 2018

Agradecimientos

Agradezco a la Universidad Pontificia de Comillas, especialmente al departamento de Ingeniería Telemática, el haber facilitado los recursos materiales para el desarrollo de este proyecto. Agradezco especialmente a mi director Dr. David Contreras por la orientación y formación obtenida durante este proyecto.

PLATAFORMA EN TIEMPO REAL BIG DATA DE PROCESAMIENTO DE SENSORES IOT PARA LA PREDICCIÓN DE LA DEMANDA DE UNA FRANQUICIA DE RESTAURACIÓN.

Autor: Rodríguez Cano de Santayana, José María.
Director: Dr. Contreras Bárcena, David.
Entidad Colaboradora: ICAI – Universidad Pontificia Comillas.

RESUMEN DEL PROYECTO

En este proyecto se ha desarrollado e implementado en un *cluster Big Data* una plataforma capaz de recoger y procesar datos IoT en tiempo real en terrazas de franquicias de restauración para realizar predicciones de demanda. Esta predicción de demanda será mostrada a las empresas a través de un *dashboard web* que permite visualizar el estado de la demanda y las condiciones atmosféricas del establecimiento en tiempo real.

Palabras clave: IoT, *Big Data*, tiempo real, *Machine Learning*, predicción de demanda.

1. Introducción

Las tecnologías *Big Data* permiten a las empresas conocer y predecir la demanda, pudiendo reaccionar a estos cambios de forma casi instantánea. Sin embargo, este tipo de soluciones no son fáciles de implementar en sectores tradicionales como el *retail* o la restauración. Esto es debido a la dificultad que presenta el estudio del comportamiento de los clientes en tiempo real en entornos no informáticos. Este proyecto pretende desarrollar una plataforma IoT *Big Data* que permita a empresas de restauración conocer el estado de la demanda en tiempo real para poder reaccionar de forma inmediata.

2. Definición del proyecto

A continuación se presentan los objetivos establecidos para este proyecto.

- Conectar la plataforma IoT existente al *cluster Big Data* Cloudera. Dicha conexión se establecerá mediante el protocolo MQTT (*Message Queue Telemetry Transport*) [1], especialmente diseñado para aplicaciones con sensores IoT.

- Desarrollar e implementar en el *cluster* un módulo en Scala que vuelque la información recibida por MQTT a una cola Kafka [2] en el *cluster Big Data*, facilitando el posterior procesamiento en tiempo real.

- Desarrollar e implementar en el *cluster* una red neuronal tipo *feed forward* [3] de tres capas para la predicción de la demanda en base a los parámetros atmosféricos. Al no disponerse de datos reales de facturación se entrenará el algoritmo con un modelo sintético. Los parámetros se consumirán de la cola Kafka mencionada anteriormente. Tras realizarse la predicción; ésta, junto con los parámetros originales, se volcará en otra cola Kafka.

- Desarrollar e implementar una aplicación *web* tipo *dashboard* en la que se pueda visualizar en tiempo real la predicción de demanda junto con la demanda observada y los parámetros atmosféricos.

3. Descripción del sistema

Como se puede observar en la Figura 1, el sistema está formado por tres grandes bloques o entornos. El primero es la plataforma IoT, encargada de medir diferentes parámetros atmosféricos y enviarlos mediante MQTT al *cluster Big Data*. En segundo lugar, se encuentra el *cluster Big Data*, en este se procesarán los datos atmosféricos para realizar la predicción de demanda y, posteriormente, se enviarán mediante *WebSocket* [4] para ser visualizados por los usuarios. Por último, desde un explorador web en un establecimiento u oficina, se podrán visualizar los datos atmosféricos y de demanda, que estarán siendo enviados en tiempo real desde un módulo que extrae estos datos de Kafka.

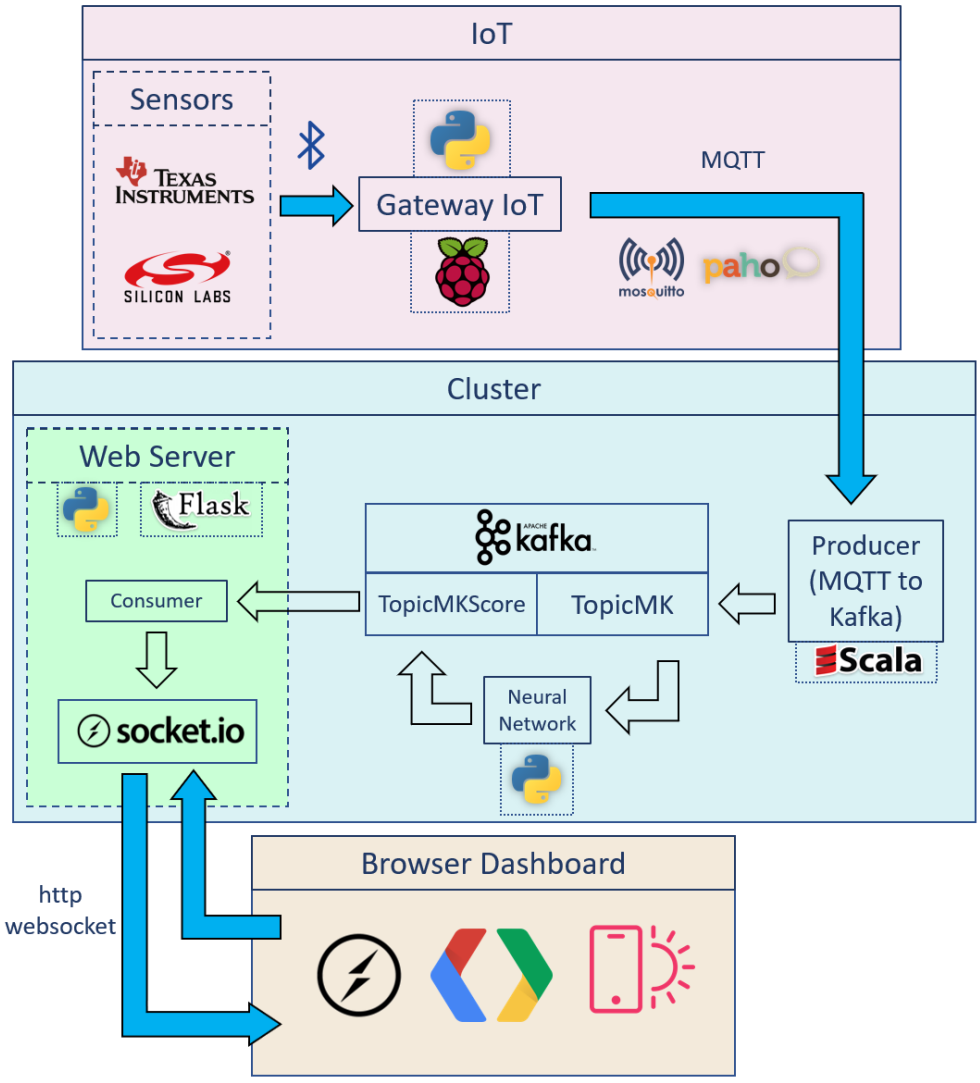

Figura 1 Diagrama de Arquitectura del sistema

4. Resultados

Como resultado final se dispone de una plataforma capaz de recolectar datos IoT, procesarlos para realizar una predicción de demanda y finalmente mostrarlos a través de una aplicación *web* de visualización en tiempo real. Se ha implementado con éxito la plataforma en el *cluster Big Data* de la Universidad Pontificia Comillas.

En la Figura 2 se muestra la herramienta de visualización final. En ella se muestran los parámetros atmosféricos, un indicador con la predicción de demanda y una gráfica que muestra cronológicamente el comportamiento de la demanda real y estimada.

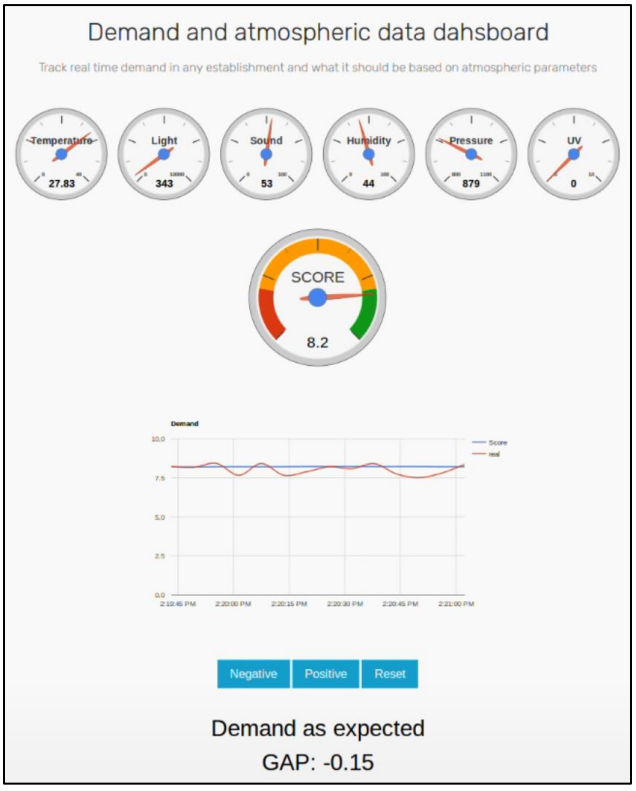

Figura 2 Dashboard de visualización

5. Conclusiones

La plataforma diseñada resulta una herramienta viable para el procesado de datos IoT en tiempo real. En primer lugar, se ha conectado con éxito la plataforma IoT existente con el *cluster Big Data*. Del mismo modo, se ha implementado con éxito un sistema de *streaming* con Kafka para la gestión de datos en tiempo real en la plataforma. Por otro lado, se ha diseñado e implementado con éxito un algoritmo de *Machine Learning* capaz de procesar datos IoT en tiempo real para la predicción de demanda. Por último, se ha desarrollado una herramienta de visualización capaz de mostrar los datos IoT y la predicción de demanda en tiempo real.

6. Referencias

[1] «mqtt.org,» [En línea]. Available: http://mqtt.org/.

[2] «kafka.apache.org,» [En línea]. Available: https://kafka.apache.org/intro.

[3] T. Gupta, «towardsdatascience.com,» [En línea]. Available: https://towardsdatascience.com/deep-learning-feedforward-neural-network-26a6705dbdc7.

[4] «developer.mozilla.org,» [En línea]. Available: https://developer.mozilla.org/es/docs/WebSockets-840092-dup.

BIG DATA REAL TIME PLATFORM FOR IOT SENSOR DATA PROCESSING FOR DEMAND PREDICTION OF A BEERHOUSE FRANCHISE.

Author: Rodríguez Cano de Santayana, José María.
Director: Dr. Contreras Bárcena, David.
Colaborating Entity: ICAI – Universidad Pontificia Comillas.

ABSTRACT

In this project, a platform capable of collecting and processing IoT data in real time in terraces of restoration franchises has been developed and implemented in a Big Data cluster to make demand predictions. This demand prediction will be shown to the companies through a web dashboard that allows them to visualize the state of the demand and the atmospheric conditions of the establishment in real time.

Keywords: IoT, Big Data, real time, Machine Learning, demand prediction. .

1. Introduction

Big Data technologies allow companies to study and predict demand, being able to react to these changes almost instantaneously. However, these sorts of solutions can't be easily implemented in traditional sectors, such as retail or restaurants. This is due to the difficulty that presents studying the behavior of customers in real time in non-computing environments. This project aims to develop an IoT Big Data platform that allows restaurants and beerhouse franchise to know the status of demand in real time in order to react immediately.

2. Objectives

The objectives established for this project are presented below.

- Connect the existing IoT platform to the Big Data Cloudera cluster. This connection will be established through the MQTT (Message Queue Telemetry Transport) [1] protocol, specially designed for applications with IoT sensors.
- Develop and implement in the cluster a Scala module that dumps the information received by MQTT to a Kafka [2] queue in the Big Data Cluster, facilitating further real time processing.
- Develop and implement a three-layer feed forward neural network [3] in the cluster for the prediction of demand based on atmospheric parameters. As there is no real billing data, the algorithm will be trained with a synthetic model. The parameters will be consumed from the Kafka queue mentioned above. After the prediction is made, this, and the original parameters will be dumped in to another Kafka queue.
- Develop and implement a dashboard type web application in which the prediction of demand can be visualized in real time together with the observed demand and atmospheric parameters.

3. System Description

As you can see in the Figure 1, the system consists of three large blocks or environments. The first one is the IoT platform, which is responsible for measuring different atmospheric parameters and sending them by MQTT to the Big Data cluster. The second one is the Big Data cluster, in which the atmospheric data will be processed to make the demand prediction. This will be sent through WebSocket [4] to be visualized by the users. Finally, from a web browser in an establishment or office, atmospheric and demand data can be displayed, sent in real time from a module that extracts this data from Kafka.

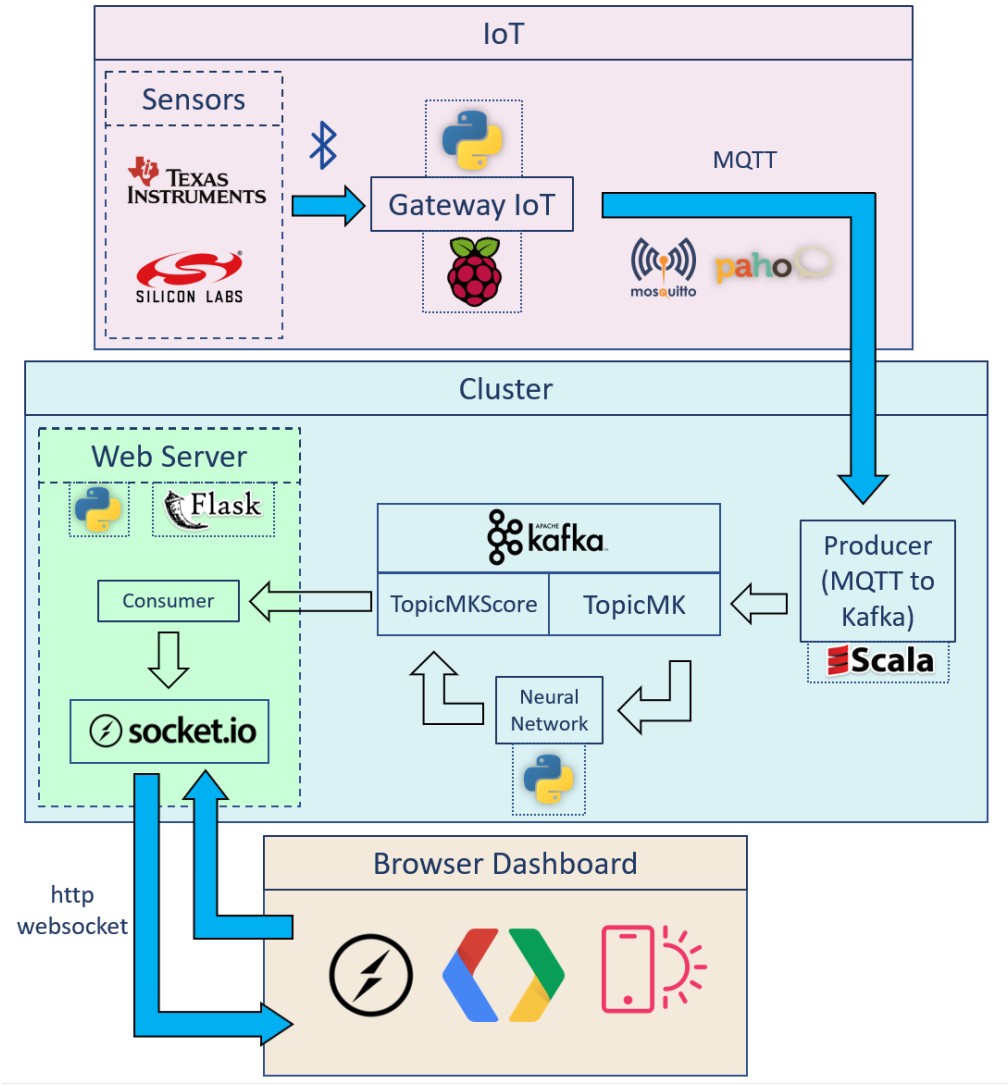

Figure 1 Architecture diagram

4. Results

The result is a platform capable of collecting IoT data, processing it to make a demand prediction and, finally, showing it through a real-time visualization web application. The platform has been successfully implemented in the Big Data cluster of the Universidad Pontificia Comillas.

In Figure 2, the final visualization tool is shown. It shows the atmospheric parameters, an indicator with the demand prediction and a graph that chronologically shows the behavior of the real and estimated demand.

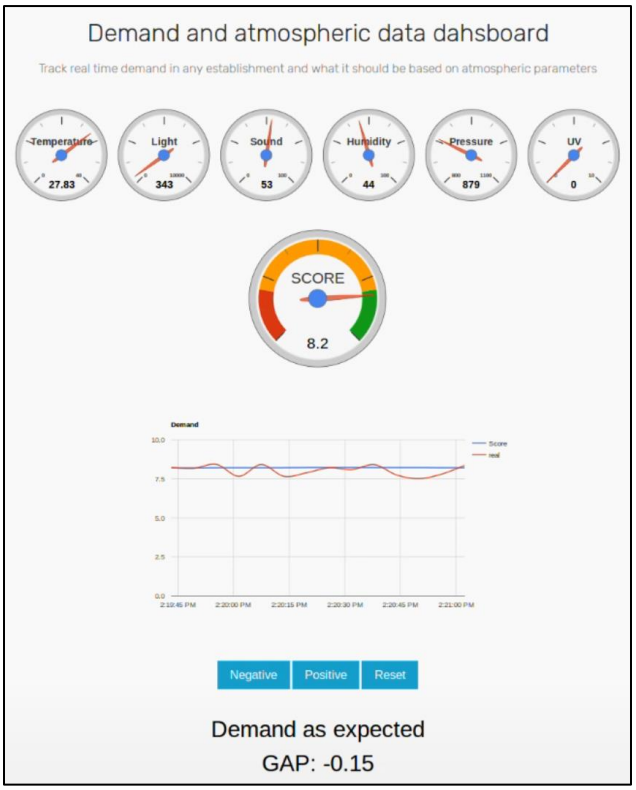

Figure 2 Visualization dashboard

5. Conclusions

The designed platform is a viable tool for the processing of IoT data in real time. First, the existing IoT platform has been successfully connected to the Big Data Cluster. In the same way, a streaming system with Kafka has been successfully implemented for real-time data management on the platform. On the other hand, a Machine Learning algorithm capable of processing IoT data in real time for demand prediction has been designed and implemented successfully. Finally, a visualization tool has been developed capable of displaying the IoT data and the prediction of demand in real time.

6. References

[1] «mqtt.org,» [En línea]. Available: http://mqtt.org/.

[2] «kafka.apache.org,» [En línea]. Available: https://kafka.apache.org/intro.

[3] T. Gupta, «towardsdatascience.com,» [En línea]. Available: https://towardsdatascience.com/deep-learning-feedforward-neural-network-26a6705dbdc7.

[4] «developer.mozilla.org,» [En línea]. Available: https://developer.mozilla.org/es/docs/WebSockets-840092-dup.

Índice de la memoria

Índice de figuras

Índice de tablas

Capítulo 1. INTRODUCCIÓN

Las tecnologías *Big Data* ofrecen a las empresas la capacidad de conocer a sus clientes, pudiendo predecir el comportamiento de la demanda. Conocer el estado de la demanda en tiempo real permite reaccionar de forma inmediata empleando diversas técnicas de márquetin, y, de este modo, aumentando notablemente los beneficios [5]. Estas estrategias se emplean principalmente en sectores muy tecnológicos como el *e-comerce* [6]. Por ejemplo, si se conoce que el estado de la demanda es muy alto se pueden emplear técnicas como el *item push*, que consiste en promocionar los productos que dejan más margen de beneficio a la empresa. También se pueden ofrecer ofertas exprés si se conoce que la demanda es muy baja y se desea que suba rápidamente.

Sin embargo, en sectores tradicionalmente menos tecnológicos (como el *retail* o la restauración) no se suele emplear este tipo de estrategias. Principalmente porque resulta mucho más difícil estudiar el comportamiento de los clientes al tratarse de un entorno no informático. Con el desarrollo de las tecnologías *Internet of Things* [IoT] empieza a ser posible este estudio del cliente para sectores tradicionales.

Este proyecto pretende diseñar una plataforma *Big Data* IoT para dar solución a este problema. Se centrará particularmente en la predicción y estudio de la demanda en franquicias de cervecerías que centran gran parte de su negocio en el servicio de terraza. Mediante sensores IoT capaces de medir las condiciones atmosféricas en la terraza y *Machine Learning* los establecimientos podrían conocer en tiempo real la demanda esperada. De este modo podrían reaccionar de forma instantánea a las condiciones de demanda, empleando las técnicas de márquetin que consideren oportunas.

Capítulo 2. DESCRIPCIÓN DE LAS TECNOLOGÍAS

En este capítulo se procede a describir y explicar las principales tecnologías empleadas en el proyecto. Para ello se divide en dos secciones correspondientes a los entornos tecnológicos más relevantes del sistema: la plataforma IoT y el cluster Big Data.

2.1 PLATAFORMA IOT

2.1.1 MQTT

MQTT (*Message Queue Telemetry Transport)* es un protocolo de red especialmente diseñado para las comunicaciones M2M [*Machine to Machine*] en plataformas IoT [1]. Está orientado a facilitar la comunicación entre sensores, aportando varias ventajas en estos entornos [7]:

- Consumo muy bajo de recursos (CPU, RAM, etc). Los sensores y demás dispositivos IoT aspiran a ser lo más compactos y baratos posible, lo cual limita su potencia computacional.

- Consumo muy bajo de ancho de banda. Las aplicaciones IoT suelen incorporar un gran número de sensores y dispositivos. Normalmente, al hacer uso de redes no específicamente diseñadas para IoT, es crítico que se consuma poco ancho de banda para no saturar la red.

- Los clientes MQTT son sencillos de implementar. MQTT es un protocolo de código abierto y estandarizado. Esto permite que sea fácil de implementar en cualquier tipo de dispositivo IoT.

La topología habitual en redes MQTT es la topología en estrella. Existe un nodo central (o *broker*) que recibe información del resto de dispositivos de la red. Esta topología se describe en el siguiente diagrama, Figura 1.

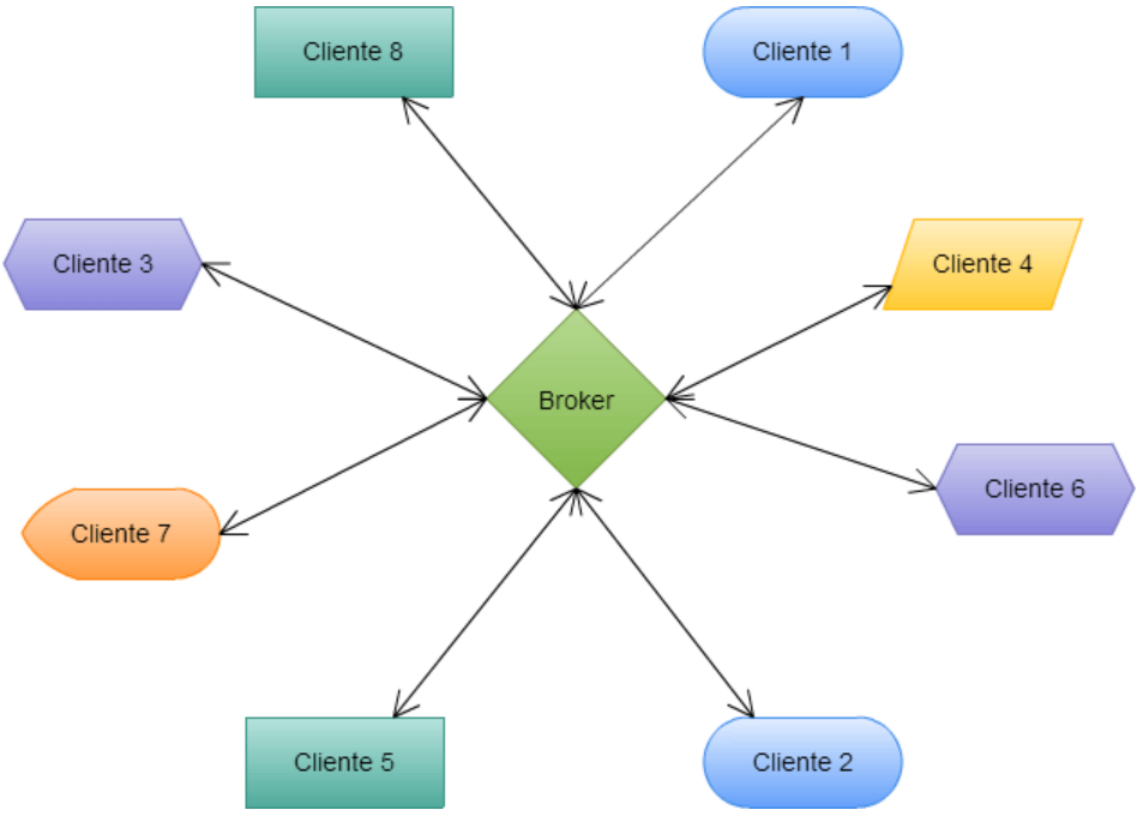

Figura 1 Arquitectura MQTT [8]

La comunicación entre dispositivos se basa en *topics* (o colas), gestionados por el *broker*. Los diferentes dispositivos pueden enviar sus datos a uno o varios *topics*. Del mismo modo, otros dispositivos pueden suscribirse a uno o varios *topics*, consumiendo los datos almacenados estos.

2.1.2 SENSORES ATMOSFÉRICOS

El sistema permite el uso de dos modelos diferentes de kits de sensores, Thunderboard™ Sense de Silicon Labs [9] y SimpleLink™ SensorTag de Texas Instruments [10]. El kit Thunderboard aporta más sensores, sin embargo, el SensorTag ofrece mejor resistencia a los elementos ya que incorpora dos capas de protección: una carcasa interior de plástico y una funda exterior de silicona. Los sensores que incorpora cada kit son los siguientes:

- Thunderboard™ de Silicon Labs: Figura 2.
  - Humedad relativa.
  - Temperatura
  - Índice ultravioleta
  - Luz Ambiente
  - Presión atmosférica
  - Nivel de sonido
  - CO2 en el aire
  - Partículas volátiles en el aire [VOC]

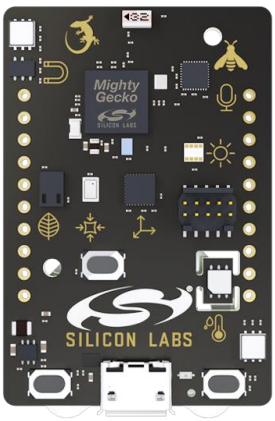

Figura 2 Thunderboard Sense [9]

- SimpleLink™ SensorTag de Texas Instruments: Figura 3.
  - Humedad relativa.
  - Temperatura
  - Luz Ambiente
  - Presión atmosférica
  - Nivel de sonido

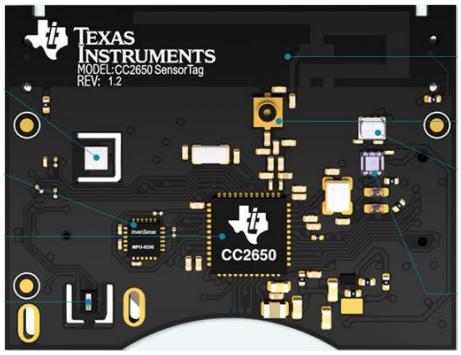

Figura 3 SimpleLink SensorTag [10]

Durante el desarrollo del proyecto se ha utilizado principalmente en Thunderboard, por aportar más parámetros atmosféricos.

Estos kits de sensores envían cada 3 segundos las medidas al gateway IoT a través de BLE (*Bluetooth Low Energy*) [11].

2.1.3 GATEWAY IOT

El *gateway* IoT (desarrollado previamente en otro Trabajo Fin de Grado) está implementado en una Raspberry Pi 3 Model B [12], con sistema operativo Linux Raspbian. Este emite una red WLAN con ssid "IoTGW". En esta red el *gateway* sirve una aplicación *web* (Figura 4) en la dirección "192.168.42.1:5050/scripts" desde la cual se activa el *gateway* ejecutando el script 3.

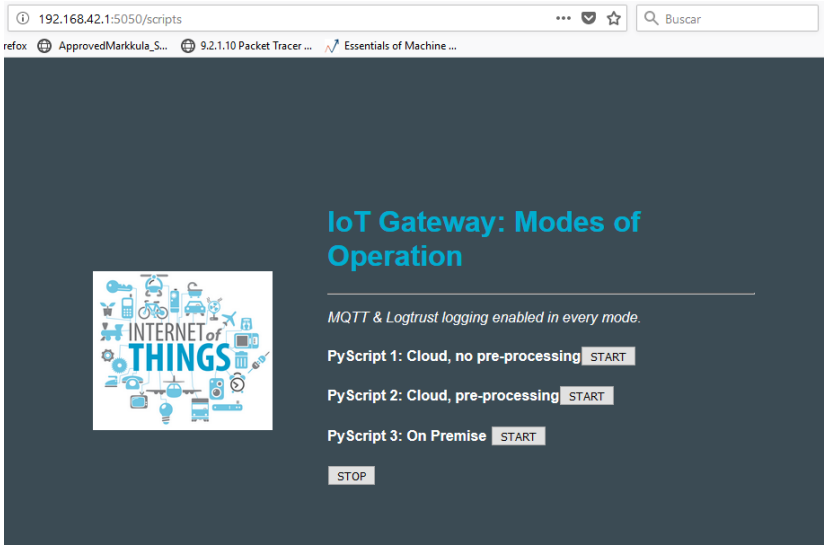

Figura 4 Aplicación web del gateway IoT

Una vez activado, el *gateway* comienza a recoger los datos atmosféricos de los sensores y a enviarlos mediante MQTT a una dirección IP establecida en el archivo "ips.json".

2.2 CLUSTER BIG DATA

2.2.1 APACHE KAFKA

Apache Kafka es una plataforma de *streaming* distribuido [2]. Es ampliamente utilizado en aplicaciones en tiempo real ya que permite crear un *pipeline* que comunica eficientemente sistemas o aplicaciones de tiempo real. Tiene tres funcionalidades principales [2]:

- Publicar y subscribirse a *topics* o *streams* de datos a modo de cola de mensajería. Cada topic tiene un nombre, con el cual se puede referenciar a la hora de consumir o producir datos.
- Almacenar la información de forma durable robusta y escalable.
- Procesar los *streams* de datos en tiempo real.

Existen cuatro tipos principales de interacción con Kafka [2]:

- **Producer**: produce (introduce) datos en uno o varios *topics* Kafka.
- **Consumer**: consume (extrae) datos de uno o varios *topics* Kafka. Los consumidores se subscriben a un *topic* Kafka de manera que reciben los datos nuevos cada vez que aparezcan en el *topic*.
- **Conector**: comunican uno o varios *topics* Kafka con otro sistema de almacenamiento.
- **Stream processor**: consume y produce datos simultáneamente. Extrae datos de un *topic*, los procesa y los devuelve al mismo o bien a otro *topic*.

Cada uno de estos elementos se pueden identificar gráficamente en el diagrama: Figura 5.

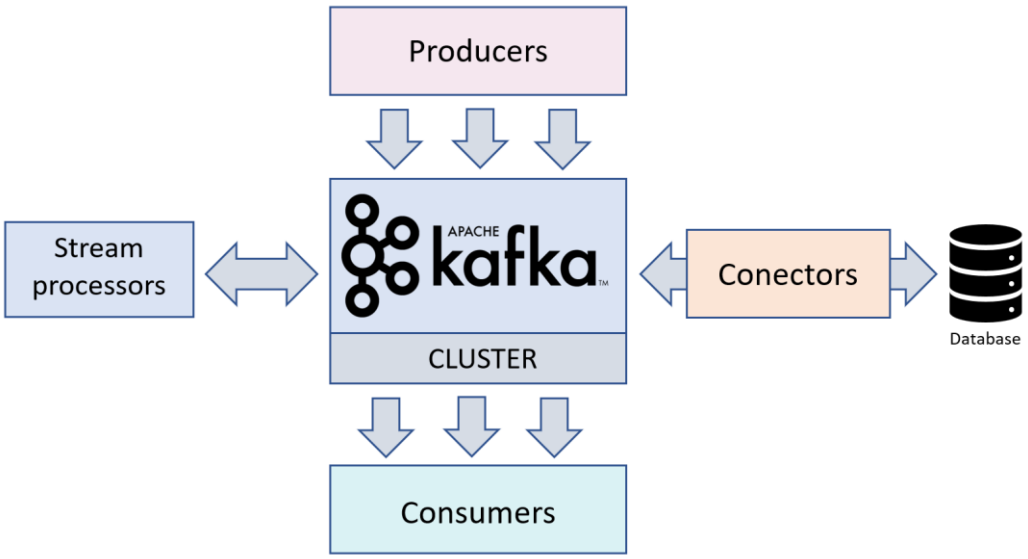

Figura 5 Elementos que interaccionan con Kafka

Capítulo 3. DEFINICIÓN DEL TRABAJO

3.1 JUSTIFICACIÓN

Ya que no existen en el mercado soluciones *Big Data* basadas en el consumo de datos IoT para la predicción de la demanda en terrazas de franquicias de restauración, en este proyecto se va a desarrollar una plataforma que permita a estos negocios ser capaces de predecir la demanda en tiempo real.

3.2 OBJETIVOS

3.2.1 CONECTAR LA PLATAFORMA IOT EXISTENTE AL CLUSTER BIG DATA

En otro trabajo fin de grado, realizado por otro alumno en la Universidad Pontificia Comillas, se desarrolló una plataforma IoT formada por sensores Bluetooth y un *gateway* IoT que recoge los datos de los sensores. Estos datos posteriormente son enviados mediante el protocolo MQTT a un servicio de almacenamiento *cloud*.

En este proyecto, esta plataforma será modificada para enviar los datos recogidos al *cluster Big Data* de la Universidad Pontificia Comillas.

3.2.2 DESARROLLAR UNA HERRAMIENTA QUE VUELQUE LOS DATOS RECIBIDOS POR MQTT EN UNA COLA KAKFA

Para el procesamiento y la gestión de los datos IoT en tiempo real es necesario almacenarlos en una plataforma de *streaming* apropiada. La plataforma de *streaming* elegida en este proyecto es Apache Kafka.

Se desarrollará un módulo Scala [13] que vuelque automáticamente cada dato recibido desde el *gateway* en un *topic* Kafka del *cluster*.

3.2.3 DESARROLLAR UN ALGORITMO DE MACHINE LEARNING PARA LA PREDICCIÓN DE DEMANDA

Se desarrollará una red neuronal tipo *feed forward* [3] para realizar la predicción de demanda a partir de los datos atmosféricos recibidos. Se desarrollará en Python [14] mediante el uso de la librería de cómputo científico Numpy [15]. La red neuronal constará de tres capas: *input*, capa oculta y *output*. La capa de *input* constará de tantas neuronas como parámetros atmosféricos se midan más una neurona de *bias* [16]. Para asegurar una predicción suficientemente precisa, la capa oculta constará de suficientes neuronas para ponderar las posibles relaciones entre datos de entrada. La capa *output* constará de una única neurona que representará la predicción.

Dado que no se dispone de datos reales de facturación en una terraza para el entrenamiento de la red neuronal, se diseñará un modelo de entrenamiento sintético. Este modelo no pretende representar el comportamiento real de la demanda, solo verificar que la red neuronal podría adaptarse a comportamientos relativamente complejos. La demanda se valorará con un valor de 0 a 1, siendo 0 condiciones extremadamente negativas (lluvia, frío, etc.) y 1 condiciones extremadamente favorables (temperatura agradable, soleado, etc.).

3.2.4 DESARROLLO DE UN DASHBOARD WEB PARA LA VISUALIZACIÓN DE LOS DATOS

Una vez estimado un valor de demanda en base a los parámetros atmosféricos es necesario que los usuarios o clientes puedan visualizar el comportamiento de la demanda. Para esto se desarrollará una aplicación *web* basada en Flask [17] que muestre los datos atmosféricos en tiempo real, así como el valor de demanda calculado por la red neuronal junto con el valor que simula la facturación real (calculado por el modelo de entrenamiento).

Adicionalmente, si la demanda se estuviera comportando peor de lo esperado es necesario notificarlo para que el establecimiento tome las medidas que considere necesarias. Si la demanda real observada resulta mucho menor que la demanda esperada (estimada por la red neuronal) se activará un indicador a modo de alarma para advertir al establecimiento.

Esta aplicación *web* recibirá los datos mediante *WebSocket* [4], de manera que se actualicen en tiempo real sin necesidad de refrescar la página. Los datos enviados por *WebSocket* desde el *cluster* serán consumidos directamente de una cola Kafka.

3.3 METODOLOGÍA

Se ha seguido una metodología de desarrollo incremental basada en *sprints* similar a Scrum [18]. Se realizarán reuniones cada pocas semanas entre el director y el autor del proyecto para revisar los requisitos del sistema y adaptarlos si fuera necesario, así como comprobar que el producto desarrollado cumple los mismos. Del mismo modo cuando sea necesario se ejecutarán pruebas en las que se comprueba que las herramientas desarrolladas se ejecutan correctamente en el *cluster Big Data*.

Dado que el proyecto consta de tres grandes bloques o capas (IoT, *cluster Big Data* y visualización en *web*) se seguirá un orden incremental. Primero se adaptará la plataforma IoT existente al sistema a desarrollar. Seguidamente se desarrollarán las herramientas

necesarias para la gestión de la información en el *cluster* y el algoritmo de predicción. Por último, se desarrollará la herramienta de visualización *web*. Si fuera necesario se podrá retroceder para realizar cambios en una capa anterior.

Todo el código se almacenará en un repositorio Git privado de la plataforma Bitbucket [19].

3.4 PLANIFICACIÓN Y ESTIMACIÓN ECONÓMICA

3.4.1 PLANIFICACIÓN DEL PROYECTO

Como se menciona en el apartado de 3.3, en este proyecto se ha empleado una metodología de desarrollo ágil. Por esto, el proceso de desarrollo se ha dividido en *sprints* de pocas semanas de duración.

En el siguiente diagrama (Tabla 1) se muestra el proceso de desarrollo del proyecto.

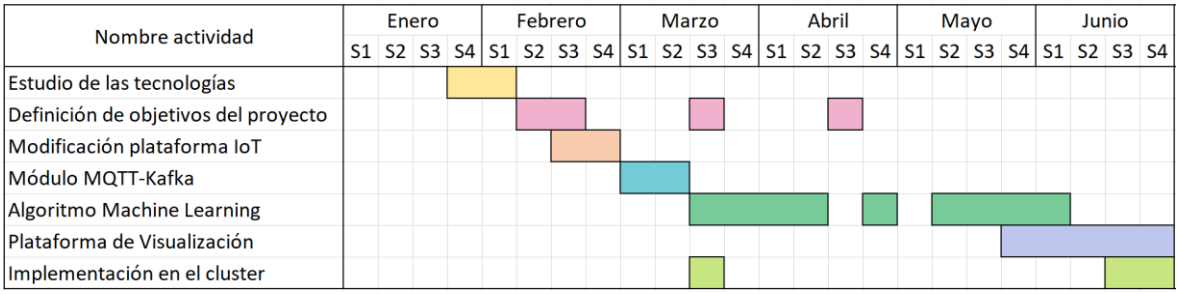

Tabla 1 Diagrama de planificación del proyecto

Como se puede observar, durante las primeras semanas se realizó un estudio de las tecnologías que se deseaba emplear y se definieron los objetivos iniciales del proyecto. Posteriormente se llevó a cabo la modificación de la plataforma IoT existente, así como el módulo que vuelca los datos MQTT en una cola Kafka. Llegados a este punto se implementaron las herramientas en el *cluster Big Data* para comprobar su correcto funcionamiento.

Como se muestra en el diagrama se realizó una redefinición de objetivos, añadiendo la idea de la aplicación comercial de la plataforma. Tras esta redefinición se empezó a desarrollar un algoritmo *Random Forest* [20] para la predicción de demanda. Unas semanas después se llegó a la conclusión de que era conveniente implementar otro tipo de algoritmo y se realizó otra redefinición de objetivos. Se desarrolló posteriormente la red neuronal a lo largo de las siguientes semanas, con una interrupción debida al periodo de exámenes. Finalmente se desarrolló la herramienta de visualización y se implementó el sistema completo en el *cluster Big Data*.

3.4.2 ESTIMACIÓN ECONÓMICA

En este apartado se procede a estimar un presupuesto para el proyecto, considerando los recursos materiales empleados y los recursos humanos que se emplearían si se emprendiese el proyecto. Esto se puede ver en la Tabla 2.

RECURSOS	CANTIDAD	PRECIO/UD	TOTAL
Recursos Humanos			
Horas programador	400	40 €	16.000 €
Horas jefe proyecto	40	100 €	4.000 €
Recursos materiales			
Sensores IoT	2	20 €	40 €
Gateway IoT	1	35 €	35 €
Cluster Big Data	1	60.000 €	60.000 €
TOTAL			80.075 €

Tabla 2 Recursos económicos

Capítulo 4. PLATAFORMA DESARROLLADA

4.1 ARQUITECTURA DE LA PLATAFORMA

Como ya se ha dicho en capítulos anteriores, la plataforma consta de tres capas, entorno IoT, *cluster Big Data* y entorno *web* para la visualización. Esto se puede ver en la Figura 6.

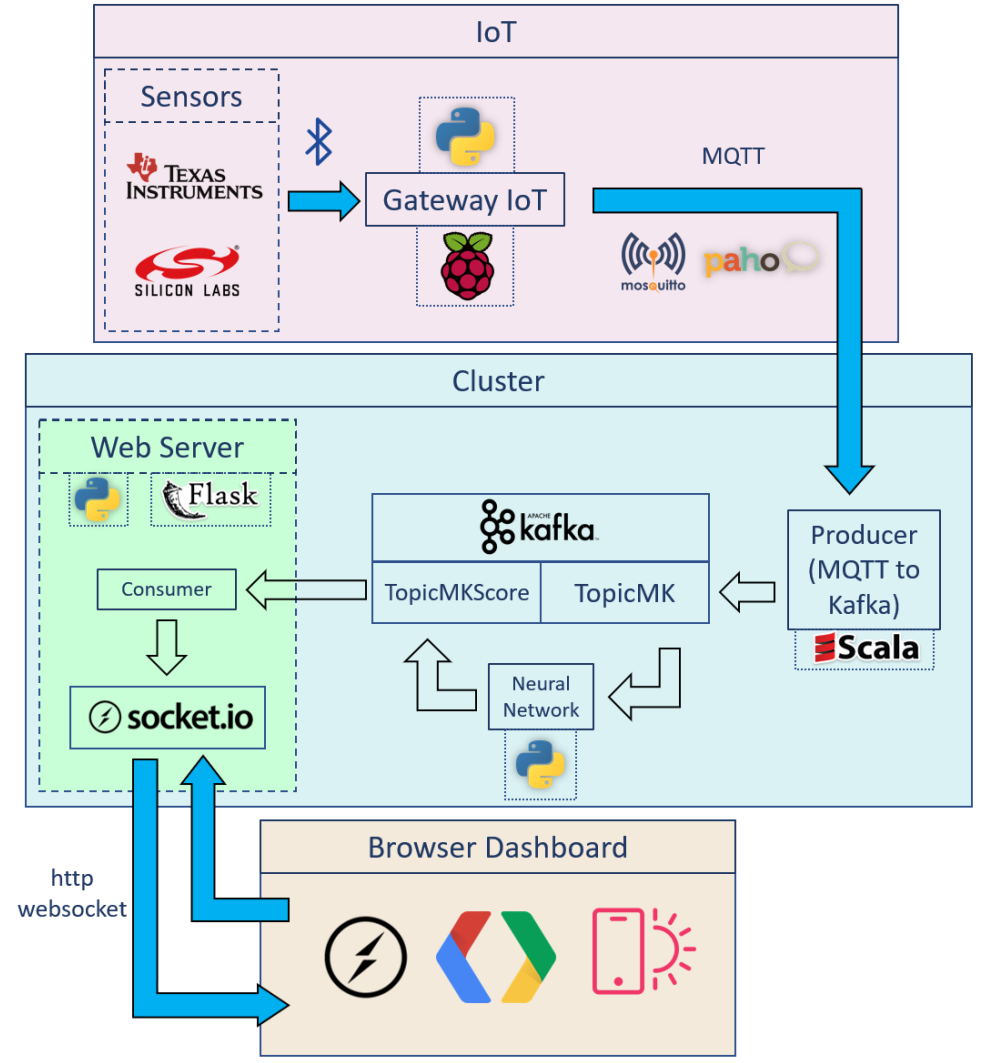

Figura 6 Diagrama de la arquitectura del sistema

4.1.1 PLATAFORMA IOT

El entorno IoT estaría situado en el propio establecimiento, donde los sensores recogerían los parámetros atmosféricos. Mediante BLE los sensores enviarán la información a un *gateway* IoT, también situado en el establecimiento. El uso de BLE es crítico, ya que, debido al reducido tamaño de los sensores, no pueden almacenar grandes cantidades de energía. De este modo ampliamos notablemente su autonomía.

El *gateway* IoT, al recibir los datos, realizará una función de pre-filtrado de la información, descartando valores imposibles (temperaturas demasiado altas o bajas, humedad superior al 100%, etc), de esta manera se evita que se desestabilice el sistema haciéndose predicciones sin sentido.

Una vez filtrada la información se enviarán los datos al *cluster Big Data* utilizando el protocolo MQTT. Debido a que en este proyecto sólo se dispone de un *gateway* IoT solamente se definirá un *topic* MQTT, llamado "mqtt". Si se implementasen más *gateways* en el sistema sería conveniente crear un *topic* para cada uno. El cliente MQTT utilizado en el *gateway* IoT está implementado con Eclipse Paho [21].

4.1.2 CLUSTER BIG DATA

Una vez en el *cluster Big Data*, los datos atmosféricos serán recibidos por el *broker* MQTT Eclipse Mosquitto en el *topic* MQTT "mqtt". Para el posterior procesado de la información es necesario trasladar esta información a un sistema especializado en *streaming* distribuido. Por esto, cuando sea recibido un mensaje, un módulo Scala lo volcará inmediatamente en un *topic* Kafka "TopicMK".

Una vez en Kafka, se procede al procesado de estos datos para realizar las predicciones de demanda. Para ello, una red neuronal *feed forward* procesará los datos atmosféricos y calculará un valor de demanda esperada. Al no disponer de datos reales de facturación en

una franquicia de restauración este algoritmo se entrenará con un modelo de demanda sintético.

Según lleguen al *topic* Kafka "TopicMK", los datos serán consumidos por el módulo encargado de realizar las predicciones. Una vez procesados estos datos atmosféricos, estos, junto con los valores de demanda calculados (el calculado por la red neuronal y el calculado por el modelo de entrenamiento), serán volcados en otro *topic* Kafka "TopicMKScore".

Una vez realizadas las predicciones es necesario que los clientes o usuarios puedan visualizar los datos y conocer el estado de la demanda en sus establecimientos. Para ello, podrán acceder a un *dashboard web* en el que se muestran los parámetros atmosféricos y la estimación de demanda.

Un servidor *web* Flask, implementado en el propio *cluster* consumirá del *topic* "TopicMKScore" y, mediante *WebSocket*, enviará los datos en tiempo real a la página *dashboard* en el explorador del cliente o usuario.

En la Figura 7 se pueden ver los diferentes elementos del sistema que interaccionan con Kafka.

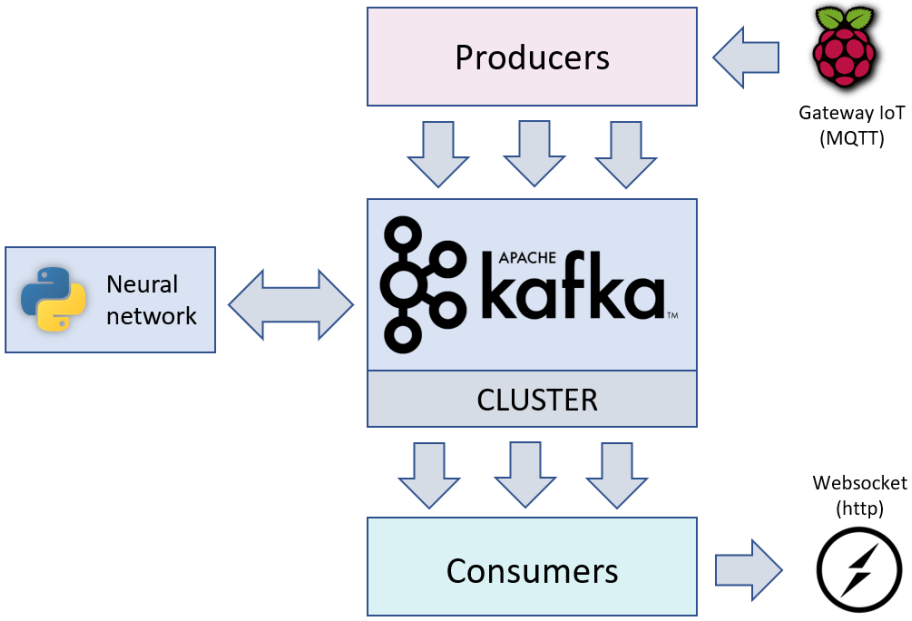

Figura 7 Arquitectura del entorno Kafka

4.1.3 DASHBOARD WEB

Desde cada establecimiento o desde una oficina se podrá acceder a una aplicación *web* que muestre los datos de un establecimiento en el que se haya implantado el sistema. La aplicación *web* que se ha desarrollado en este proyecto consta de dos páginas: una página de bienvenida en la que se muestra información acerca del funcionamiento del sistema y una página *dashboard* en la que se visualizan los datos. En esta última, se puede monitorizar en tiempo real a través de gráficos e indicadores tanto los parámetros atmosféricos en el establecimiento como la predicción de demanda y la demanda real.

Desde la página de bienvenida se puede acceder a la de *dashboard* a través de un botón. Una vez en la página de *dashboard*, el explorador web del usuario o cliente se comunicará mediante *WebSocket* con el servidor web situado en el *cluster Big Data*. El *cluster*, cada vez que se introduzca un dato nuevo en el *topic* "topicMKScore" enviará esta información al explorador *web* para que sea mostrada en las diferentes gráficas e indicadores.

El sistema de comunicaciones por *WebSocket* se ha basado en un proyecto existente [22] [23].

Cuando se ejecuta la aplicación web se lanza un hilo que consume datos del *topic* "topicMKScore" y cada vez que un dato nuevo aparezca se envía al explorador para que lo muestre. La arquitectura de la aplicación web de visualización se muestra en la Figura 8.

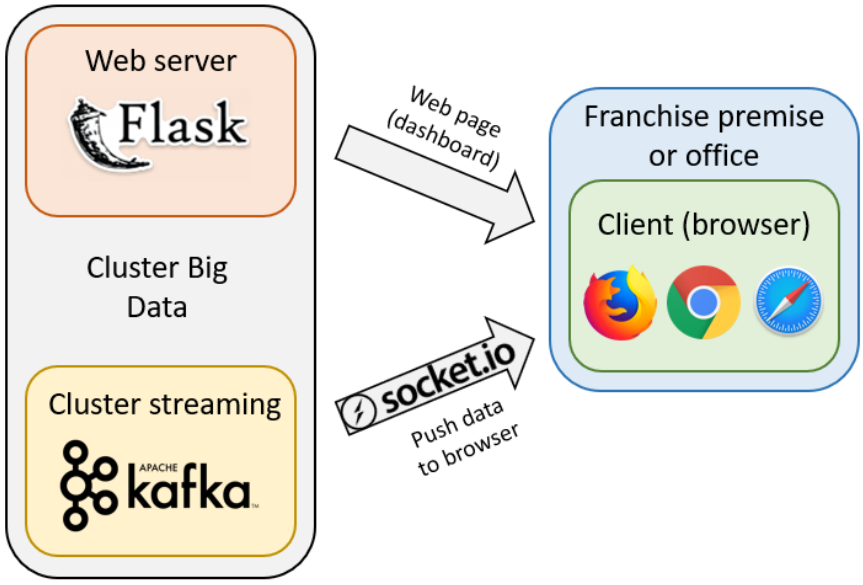

Figura 8 Diagrama de la arquitectura de la herramienta de visualización

4.2 DESARROLLO DE LOS ENTORNOS

4.2.1 ENTORNO IOT

En este proyecto se ha modificado el envío de los datos recogidos por el *gateway* IoT. Por un lado, se han serializado los datos a enviar en una sola cadena de caracteres. Se separa cada parámetro con el carácter "%". Cada parámetro contiene el nombre del parámetro y el valor correspondiente separado por ":". Se ha elegido este método de serialización ya que facilita la lectura de los datos, necesitándose únicamente funciones *split*. Los parámetros enviados y sus respectivos títulos son los siguientes:

- **name**: identificador del dispositivo (sensor) que ha recogido los datos. Por ejemplo "Thunder Sense #30175"
- **temp**: temperatura en grados centígrados
- **sound**: sonido en decibelios.
- **co2**: nivel de co2 en el aire.
- **voc**: partículas volátiles en el aire (volatile organic compounds)
- **humi**: humedad relativa.
- **light**: nivel de iluminación en lúmenes.
- **pres**: presión atmosférica en pascales.
- **uvin**: índice ultravioleta.

Un ejemplo de paquete serializado sería:

```
name:Sensor Tag #30175%sound:55%co2:0%temp:23.54%voc:0%humi:26%light:19%pres:886%
uvin:0
```

Del mismo modo, se ha añadido una función que ejecuta el envío mediante MQTT al *cluster Big Data*, cuya dirección se configura en un archivo .json. La función de envío es la siguiente:

```
def sendMqtt(topic, packet):
      with open('ips.json') as json_data:
            d = json.load(json_data)
            dest_ip = d['dest_ip']
      publish.single('mqtt', packet, hostname=dest_ip)
      return
```

4.2.2 ENTORNO CLUSTER BIG DATA

En cuanto al sistema implantado en el cluster se pueden distinguir tres grandes bloques: *streaming* de datos IoT con Kafka, predicción de demanda con algoritmo *Machine Learning* y servidor *web* para la visualización.

4.2.2.1 Kafka streaming

Antes de desarrollar ninguna herramienta ha sido necesario crear los dos *topics* Kafka que utiliza el sistema. Para ello hay que situarse en el directorio donde este instalado Kafka:

```
$ cd /home/spark/apps/kafka_2.11-0.10.2.0
```

Posteriormente se introducen los comandos de creación de *topics*:

```
$ kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --
partition 1 --topic topicMK
$ kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --
partition 1 --topic topicMKScore
```

Una vez creados los *topics* necesarios se comienza con el desarrollo de las herramientas. En primer lugar, se creará un módulo en Scala que consuma los datos recibidos por MQTT y los vuelque en el *topic* “topicMKScore”.

Para realizar estas transferencias sin tener que comprobar periódicamente los mensajes MQTT se sobrescriben varias funciones del cliente MQTT Eclipse Paho:

- La función **messageArrived**, que se ejecuta cuando se recibe un mensaje MQTT.

- La función **connectionLost**, que se ejecuta cuando se pierde alguna conexión.
- La función **deliveryComplete**, que se ejecuta cuando se realiza un envío satisfactoriamente.

La redefinición de dichas funciones se puede ver a continuación:

```
val callback = new MqttCallback{

  override def messageArrived(topic: String, message: MqttMessage): Unit = {

    println("Receiving data from mqtt topic '" + topic + "', Message: " +
            message.toString.slice(0,4))
    val data = new ProducerRecord[String,String](kafkaTopic,
                                                 "values",
                                                 message.toString)
    //each time a mqqt msg arives it's sent to the kafka desired topic
    producer.send(data)
    println("Sent to kafka topic '" + kafkaTopic + "'")
  }
  override def connectionLost(cause: Throwable): Unit = {
    println(cause)
  }
  override def deliveryComplete(token: IMqttDeliveryToken):Unit = {}
}
```

Por otro lado, se ha desarrollado un módulo en Python capaz de consumir y producir datos en diferentes *topics* Kafka. A este módulo se le añade posteriormente el código necesario para calcular la predicción de demanda.

En primer lugar, se crean los objetos consumidor y productor. Se especifica en ambos casos el nodo al que se deben conectar y el puerto de Kafka. En el caso del consumidor debemos establecer por adelantado de que *topic* queremos que consuma datos:

```
consumer = KafkaConsumer('topicMK', bootstrap_servers='master01:9092')
producer = KafkaProducer(bootstrap_servers='master01:9092')
```

Posteriormente se crea un bucle que itere al consumidor, es decir, que se ejecute para cada nuevo mensaje recibido en el *topic* establecido al crear el objeto. Una vez en ese bucle, se ejecutan las funciones del algoritmo de *Machine Learning* necesarias para calcular el valor

de demanda esperado ("score") y real ("real"). En cada iteración del bucle, tras realizar los cálculos, se publica en el *topic* "topicMKScore" el mensaje original con los resultados obtenidos añadidos, así como la marca de tiempo del momento en que se realiza la predicción.

```
for msg in consumer:
	'''
	Some code executing the demand prediction and updating the variable msg
	with the results.
	'''
	producer.send('topicMKScore', bytes(msg))
```

La misma estructura de consumidor se utiliza más adelante en el desarrollo de la aplicación *web*. En esta se consumirán datos del *topic* "topicMKScore" para enviarlos mediante *WebSocket* al *dashboard* web del cliente o usuario. El código completo de este módulo se puede ver en el anexo A ().

4.2.2.2 Algoritmo de Machine Learning

Inicialmente en este proyecto se implementó un algoritmo *Random Forest* para la predicción de demanda. *Random Forest* es un algoritmo diseñado para resolver problemas de clasificación, pero también se puede aplicar a problemas de regresión (como el que se plantea en este proyecto) [20].

Random Forest funciona evaluando los datos a través de varios árboles de decisión. Tras obtener un resultado de cada árbol de decisión se pondera cual es el más frecuente y este se presenta como resultado final. Frecuentemente se implementa especializando diferentes árboles de decisión en diferentes partes de los datos de entrada. De este modo se pueden analizar datos con comportamientos complejos. En la Figura 9 se muestra la arquitectura básica de un *Random Forest*:

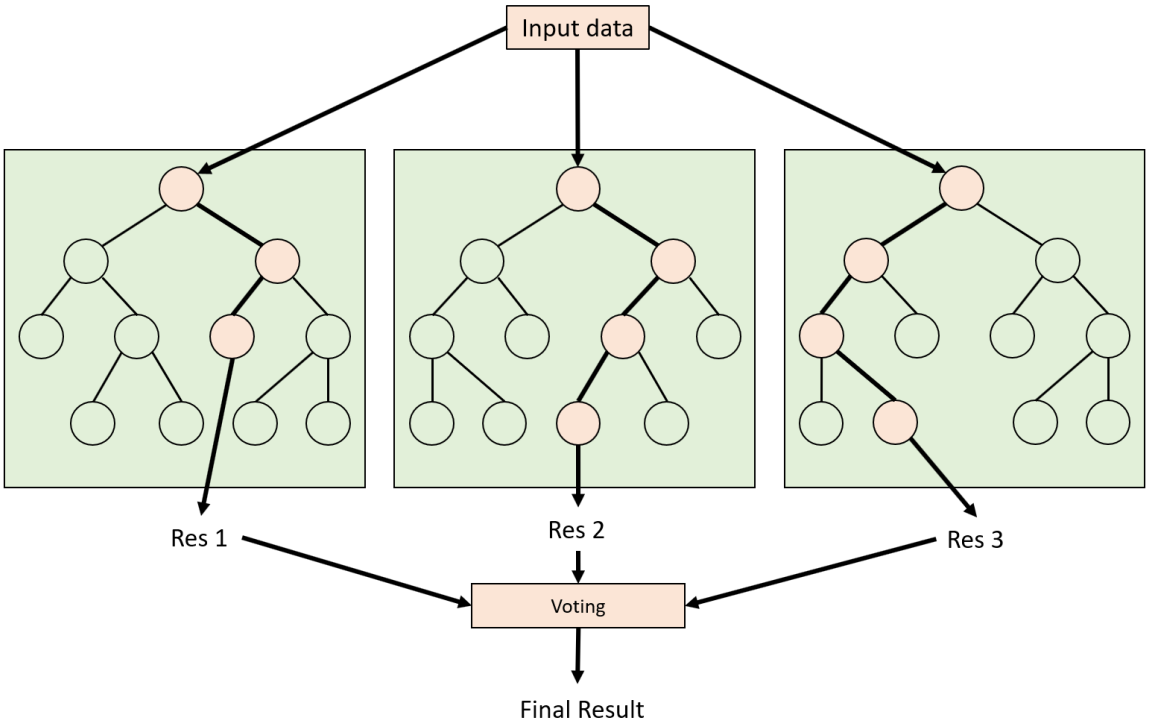

Figura 9 Funcionamiento del algoritmo Random Forest

Para ello se desarrolló un módulo Pyspark [24] (Python especializado para entornos de procesamiento distribuido). Se hizo uso de la implementación de *Random Forest* que aporta la librería de *Machine Learning* de Pyspark: mllib [25].

Para entrenar el algoritmo y conseguir que realice la predicción de demanda deseada es necesario aportar un *dataset* con datos de ejemplo. Estos datos se almacenan en un archivo de texto plano, en el que cada fila es un set de datos atmosféricos y valor de demanda. El primer elemento de la fila será el valor de demanda (en €/m2) y los siguientes serán los parámetros atmosféricos (siempre en el mismo orden). Cada elemento de una fila estará separado por un espacio. Un ejemplo de archivo *dataset* sería:

```
2334.30761905 1:54 2:14.02 3:907 4:18 5:887
```

```
2583.6152381 1:56 2:16.07 3:1114 4:17 5:883
2018.92285714 1:57 2:11.19 3:621 4:22 5:891
2197.66666667 1:54 2:12.81 3:800 4:19 5:889
1706.73047619 1:62 2:9.39 3:403 4:34 5:893
2223.76857143 1:54 2:12.92 3:813 4:18 5:888
2072.07619048 1:56 2:12.03 3:670 4:23 5:890
2328.75142857 1:60 2:13.4 3:851 4:19 5:888
```

Por supuesto, para realizar un entrenamiento correcto es necesario aportar gran cantidad de datos. Suficientes para contemplar todas las situaciones que se quiere ser capaz de predecir. Para ello se generan los datos en un archivo Excel, tratando de contemplar diferentes situaciones meteorológicas.

Una vez creado el archivo Excel se convierte a un archivo .CSV (*Comma Separated Values*). Este archivo será leído por un módulo, que aplicando un modelo sintético calculará el valor de demanda con el que se entrenará el algoritmo *Random Forest*. Como ya se ha mencionado, al no disponer de datos de facturación reales los datos de entrenamiento se generan en base a un modelo sintético. Este modelo sintético no pretende reproducir el comportamiento de la demanda, solo asegurar que el algoritmo se puede entrenar correctamente y predecir comportamientos medianamente complejos.

Por último, dos datos junto con el valor de demanda calculado por el modelo sintético se escribirán en un archivo de *dataset* (.dts) para entrenar posteriormente el algoritmo *Random Forest*.

En cuanto al desarrollo y entrenamiento del *Random Forest*, para utilizar las capacidades de computación en paralelo del *cluster Big Data*, es necesario utilizar un contexto Spark [26].

```
sc = SparkContext(appName="PythonRandomForestRegressionExample")
sc.setLogLevel("WARN")
```

Del mismo modo es necesario cargar los datos y separarlos en *set* de entrenamiento y de prueba:

```
DATASET_PATH = 'Files/dataset.dts'
```

```
data = MLUtils.loadLibSVMFile(sc, DATASET_PATH)
(trainingData, testData) = data.randomSplit([0.7, 0.3])
```

Una vez cargados el contexto Spark y los datos de entrenamiento y test se puede proceder al entrenamiento del algoritmo. En este paso se deben configurar los diferentes parámetros del algoritmo. Los parámetros más importantes son:

- **numTrees**: número de árboles de decisión de los que constará el *Random Forest*.
- **maxBins**: número que establece la discretización de los resultados.
- **maxDepth**: número de árboles de decisión a los que se puede someter un dato para alcanzar un resultado.

El código para crear y entrenar el modelo se muestra a continuación:

```
model = RandomForest.trainRegressor(trainingData,
                                    categoricalFeaturesInfo={},
                                    numTrees=110,
                                    featureSubsetStrategy="auto",
                                    impurity='variance',
                                    maxDepth=20,
                                    maxBins=32)
```

Una vez creado y entrenado el modelo se procede a evaluar los datos de *test* y calcular el error cometido. De esta manera sabremos si el algoritmo se ha entrenado correctamente y se comporta como se desea.

```
predictions = model.predict(testData.map(lambda x: x.features))
labelsAndPredictions = testData.map(lambda lp: lp.label).zip(predictions)
testMSE = labelsAndPredictions.map(lambda (v, p): (v - p) * (v - p)).sum() /
float(testData.count())
```

Estos resultados, junto con una descripción textual del algoritmo se guardan en un archivo de texto “model_debug_string.txt”. A continuación, se muestra un fragmento de dicho archivo (el archivo completo consta de casi 35000 líneas):

```
Test Mean Squared Error :387.864215141
Test Mean Error :19.6942685861
```

```
TreeEnsembleModel regressor with 110 trees

  Tree 0:
    If (feature 2 <= 753.0)
     If (feature 1 <= 10.9)
      If (feature 2 <= 493.0)
       If (feature 4 <= 892.0)
        If (feature 0 <= 55.0)
         If (feature 2 <= 470.0)
          Predict: 1759.41809524
         Else (feature 2 > 470.0)
          Predict: 1758.623809526667
        Else (feature 0 > 55.0)
         If (feature 2 <= 470.0)
          If (feature 0 <= 56.0)
           Predict: 1785.54666667
       [...]
```

Una vez desarrollado el algoritmo *Random Forest* se decidió cambiar de algoritmo (a una red neuronal *feed fordward*) por una serie de motivos:

- *Random Forest* se puede utilizar para afrontar problemas de regresión, pero los resuelve como una clasificación. Esto quiere decir que los resultados que se obtengan siempre estarán dentro de un conjunto cerrado de números. Para el desarrollo de esta plataforma resulta más apropiado un sistema de regresión que no esté limitado a un número concreto de resultados.

- Es necesario crear un archivo de datos. Dado que el entrenamiento del algoritmo se hace en base a un modelo sintético no es eficiente ni escalable tener que crear un *dataset* que represente el modelo. Resultará una mejor solución entrenar el algoritmo directamente comparando con el modelo que se quiere reproducir.

- Al tener que utilizarse librerías específicas de *Machine Learning* para implementarlo resulta menos escalable al no poder modificarse el funcionamiento y los parámetros internos del algoritmo.

Como ya se ha mencionado anteriormente como algoritmo de *Machine Learning* definitivo se ha elegido una red neuronal de tipo *feed forward*. Se ha elegido este tipo de algoritmo por tres motivos principalmente:

- Al no disponer de datos reales es estrictamente necesario usar un algoritmo de *Machine Learning* supervisado. No necesita un *dataset* para el entrenamiento ya que se puede entrenar directamente comparando los resultados con los obtenidos por el modelo de entrenamiento.

- Puede utilizarse para resolver problemas muy variados, tanto de regresión como clasificación. Las redes neuronales son consideradas uno de los algoritmos más potentes y versátiles. A diferencia del *Random Forest* no está limitado a un conjunto finito de resultados, se realizan operaciones que pueden resultar en cualquier número real dentro de un rango determinado.

- Al ser un algoritmo muy versátil resulta muy adecuado ya que no se dispone de datos reales y por tanto no se conoce que comportamiento sigue la demanda. Si en el futuro se obtuviesen datos de facturación reales es muy probable que pudieran modelarse con una red neuronal.

- Se puede implementar sin el uso de librerías específicas de *Machine Learning*, de este modo se puede modificar fácilmente cualquier característica de la red, aportando gran escalabilidad y control sobre la herramienta.

Se ha desarrollado en Python, utilizando la librería Numpy para la realización de los cómputos. Se basado el código en una red neuronal para resolver problemas de clasificación [27]. Se ha elegido Python ya que actualmente es el lenguaje más utilizado y demandado por las empresas para proyectos de *Machine Learning* y *data science,* como se puede observar en la Figura 10. Otro factor para la elección de Python es que la curva de aprendizaje del leguaje es muy rápida en comparación con otros lenguajes utilizados en *Machine Learning* como R. o Scala. También permite realizar las mismas operaciones en menor número de

líneas de código, lo que facilita la legibilidad del código y el poder modificarlo en el futuro, haciendo la herramienta más escalable.

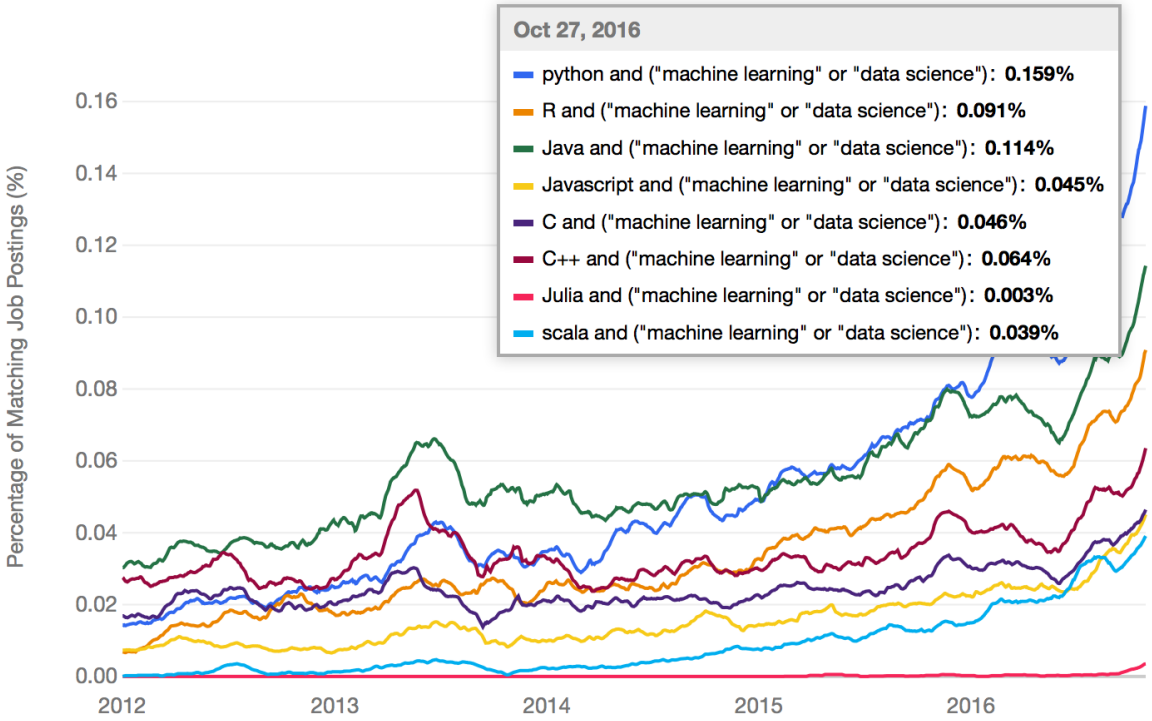

Figura 10 Comparativa de demanda de lenguajes de programación [28]

En primer lugar, se ha desarrollado la red neuronal y se ha entrenado el modelo, posteriormente se implementó en el sistema de *streaming* Kafka.

Las redes neuronales *feed forward* están com puestas de capas de neuronas. Las neuronas de cada capa se conectan con las neuronas de la capa siguiente a través de sinapsis o pesos. Estos pesos tendrán que ser ajustados en un proceso de entrenamiento para obtener el comportamiento deseado. A continuación, se muestra un ejemplo de red neuronal *feed forward* sencilla (Figura 11):

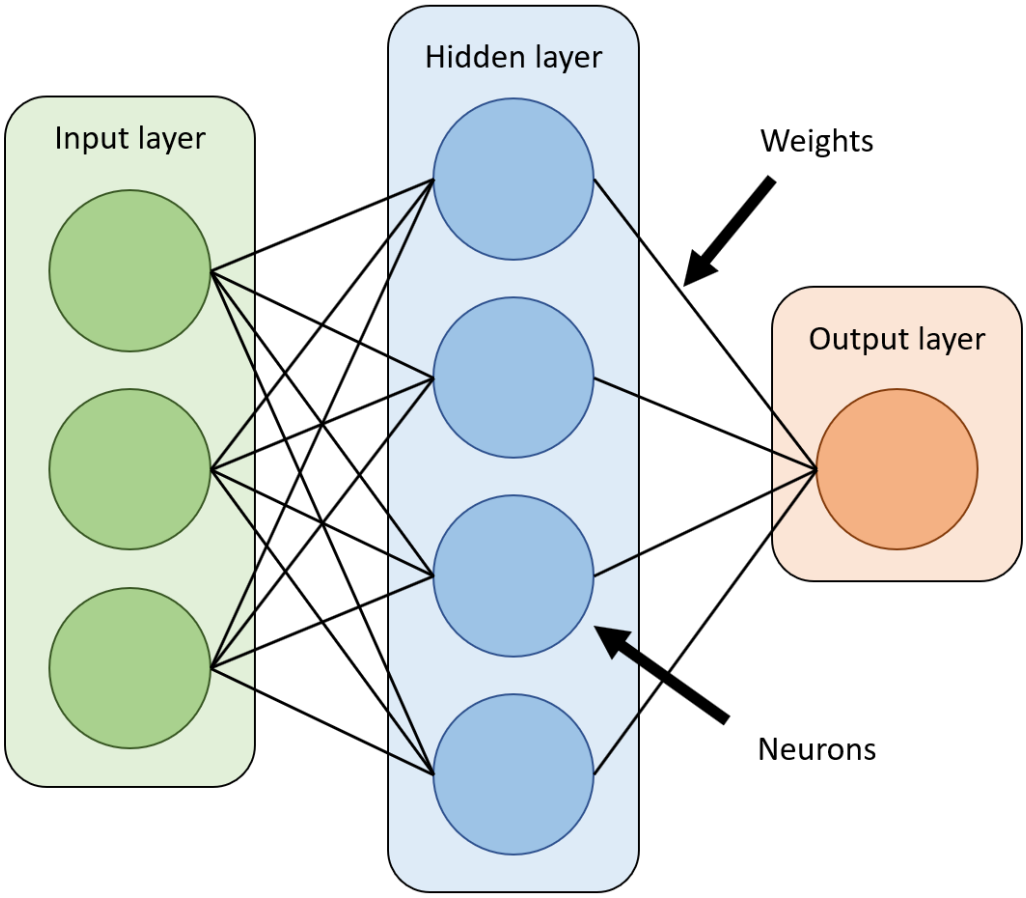

Figura 11 Arquitectura de una red neuronal feed forward

La red neuronal que se ha desarrollado en este proyecto está compuesta de tres capas:

- **Capa de entrada** o *input* (X) en la que se introducen los valores atmosféricos.
- **Capa oculta** o *hidden layer* (z) que permite valorar las relaciones entre parámetros de entrada.
- **Capa de salida** u *output* (o) que representa la estimación de demanda.

A la capa de entrada se le ha añadido una neurona adicional que actúa de *bias* (siempre con valor 1.0). El *bias* es una práctica muy común y recomendable en *Machine Learning* ya que

te permite obtener un resultado con una componente constante, que no depende únicamente de los datos de entrada.

Adicionalmente, es frecuente implementar una función de activación en cada neurona. En este caso se ha elegido la función sigmoide (Figura 12), que permite mantener el valor de una neurona entre 0 y 1 sea cual sea la entrada. Por esto, los valores de entrada se normalizan entre 0.0 y 1.0, estableciéndose un límite inferior y superior para cada parámetro. Por ejemplo, la temperatura se divide entre 40, de manera que 40ºC se conviertan en 1.0, 20ºC en 0.5, etc.

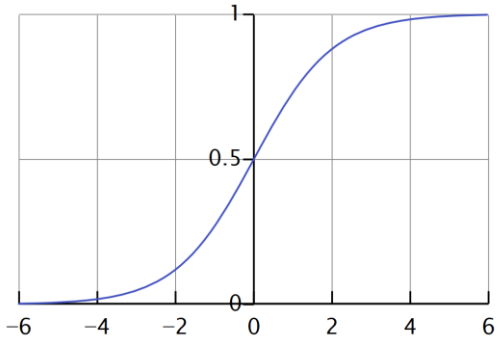

Figura 12 Función sigmoide [29]

Del mismo modo el código desarrollado para aplicar la función sigmoide o su derivada es el siguiente:

```
import numpy as np
def sigmoid(self, s, deriv=False):
       if not deriv:
            return 1/(1+np.exp(-s))
        else:
            return s * (1 - s)
```

Cada capa es por tanto el resultado del producto vectorial entre la capa anterior y una serie de pesos o sinapsis (W). Adicionalmente se aplica la función de activación en cada capa para garantizar que los valores de las neuronas están entre 0.0 y 1.0. Este proceso se ilustra en la Figura 13.

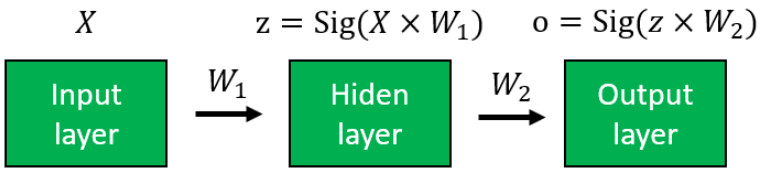

Figura 13 Proceso de propagación en la red neuronal

En cuanto al desarrollo de la red neuronal se ha diseñado un objeto "Network" que consta de los siguientes atributos:

- **inputSize**: tamaño en número de neuronas de la capa de entrada.
- **hiddenSize**: tamaño en número de neuronas de la capa oculta.
- **outputSize**: tamaño en número de neuronas de la capa de salida.
- **W1**: pesos o sinapsis que conectan la capa de entrada con la capa oculta. Será una matriz de dimensiones inputSize x hiddenSize. Se inicializan con valores aleatorios en el rango [-1,1]
- **W2**: pesos o sinapsis que conectan la capa oculta con la capa de salida. Será una matriz de dimensiones hiddenSize x outputSize. Se inicializan con valores aleatorios en el rango [-1,1]

Y las siguientes funciones:

- **forward(self, X):** devuelve el valor de la capa de salida tras introducir X en la capa de entrada y realizar los cálculos de propagación hacia delante.
- **backward(self, X, y, o):** ajusta los atributos W1 y W2 en base a unos valores de entrada (X), un valor de salida calculado mediante la función forward(o) y un valor esperado (y) calculado por el modelo de entrenamiento.
- **sigmoid(self, s, deriv=False):** como se indicado anteriormente devuelve los valores de entrada (s) tras aplicarles la función sigmoide o su derivada. Si no se especifica no utilizará la derivada.

- **train(self, X, y):** realiza una propagación hacia delante con la función *forward* y una hacia atrás con la función *backward*.

Como no se dispone de datos reales de facturación, se entrenará el algoritmo con un modelo predefinido que simule la realidad. Esta simulación no pretende ser precisa, solo comprobar que la red neuronal puede entrenarse correctamente y ajustarse lo más posible al modelo de entrenamiento. A los datos de este modelo de entrenamiento se les puede añadir un ruido aleatorio para reproducir en cierta medida condiciones reales de consumo. Un ejemplo de modelo de entrenamiento se puede ver en el siguiente código:

```
def reality_model(data, noise = False):
	if type(data) is dict:
		t = float(data['temp'])/40.
		h = float(data['humi'])/100.
		l = float(data['light'])/10000.
		p = float(data['pres'])/1200.
		s = float(data['sound'])/100.
		c = float(data['co2'])/10.
		u = float(data['uvin'])/10.
	else:
		t = float(data[0][0])
		h = float(data[0][1])
		l = float(data[0][2])
		p = float(data[0][3])
		s = float(data[0][4])
		c = float(data[0][5])
		u = float(data[0][6])

	res = ((1 - h)+t)*0.4 + l*0.2 + (p+h)*0.2 + s*0.1 + (c + u)/2*0.1

	if noise:
		res += (np.random.random()*2-1)*0.05
	if res > 1:
		res = 1
	elif res < 0:
		res = 0

	return res
```

Para entrenar la red neuronal, es decir, ajustar los pesos iterativamente hasta que se adapte al modelo deseado, se utiliza el proceso de *backpropagation* [30], que consta de varios pasos. Inicialmente, se realiza la propagación hacia delante, es decir se multiplica cada capa por los

correspondientes pesos y se aplica la función de activación hasta que se obtiene un resultado (o).

$$z = Sig(X \times W_1)$$

$$o = Sig(z \times W_2)$$

Esto expresado en código Python es:

```
def forward(self, X):
    self.z = np.dot(X, self.W1)
    self.z2 = self.sigmoid(self.z)
    self.z3 = np.dot(self.z2, self.W2)
    o = self.sigmoid(self.z3)
    return o
```

Posteriormente se calcula el error cometido como la diferencia entre el resultado (o) y el valor esperado (y), calculado con el modelo de entrenamiento.

$$o_{error} = y - o$$

A continuación, se realizan los siguientes cálculos para ajustar los pesos (W1 y W2):

$$\delta_o = o_{error} \cdot Sig'(o)$$

$$z_{error} = \delta_o \times W_2^T$$

$$\delta_z = z_{error} \cdot Sig'(z)$$

$$W_1 = W_1 + X^T \times \delta_z$$

$$W_2 = W_2 + z^T \times \delta_o$$

Este proceso de *backpropagation* expresado en código Python es:

```
def backward(self, X, y, o):
      self.o_error = y - o
      self.o_delta = self.o_error*self.sigmoid(o, deriv = True)
```

```
        self.z2_error = self.o_delta.dot(self.W2.T)
        self.z2_delta = self.z2_error*self.sigmoid(self.z2, deriv = True)

        self.W1 += (X.T.dot(self.z2_delta))
        self.W2 += (self.z2.T.dot(self.o_delta
```

De este modo, el proceso de entrenamiento serían iteraciones de propagaciones hacia delante y hacia atrás, progresivamente ajustando los pesos. Para asegurar el comportamiento correcto de la red neuronal en cualquier condición atmosférica los datos de entrenamiento se generan sintéticamente. Si se utilizasen datos atmosféricos reales medidos por los sensores sería muy costoso obtener muestras lo suficientemente variadas: diferentes lugares, ángulos respecto al sol, diferentes momentos del día, del año, etc. Por esto se generan combinaciones de datos atmosféricos aleatorios en cada iteración del entrenamiento. De esta manera, se genera un *set* de datos sintéticos, son evaluados por el modelo de entrenamiento y por la red neuronal, se comparan los resultados y por último se realiza el ajuste en base al error. La función para ejecutar una iteración de entrenamiento se puede ver en el siguiente código:

```
def train (self, X, y):
        o = self.forward(X)
        self.backward(X, y, o)
```

Para guardar el estado de la red una vez entrenada se salvará el objeto “Network” en un archivo Python *pickle* [31], que permitirá cargarlo cuando se vaya a utilizar la red. De este modo el proceso completo de entrenamiento con N iteraciones sería:

```
from neural import *
net = Network()

N = 20000

for _ in range(N):
        x = get_random_data()
        X = dic_to_array(x)
        y = reality_model(X)
        r = net.forward(X)
        net.train(X,y)

save_as_pickle(net, 'net.plk')
```

Para ilustrar el proceso de entrenamiento se ha desarrollado una herramienta que genera imágenes representativas de la red neuronal. Esta herramienta se ha desarrollado en Python mediante la librería OpenCV [32]. Esta herramienta muestra gráficamente el estado de la red neuronal en cualquier iteración del proceso de entrenamiento. Cada neurona se muestra como un círculo gris, el tono de gris varía según el valor de esa neurona, siendo 0.0 negro y 1.0 blanco. Con propósito ilustrativo las neuronas de la capa *input* tienen un aro verde alrededor, las de la capa oculta azul y la neurona de *output* rojo.

En esta representación también se muestras las sinapsis (pesos o conexiones entre neuronas) como líneas que unen distintas neuronas. Del mismo modo que con las neuronas, los colores de las sinapsis también muestran orientativamente el valor de esta. El color verde representa valores positivos, el rojo valores negativos y el negro valores cercanos a 0. La herramienta permite representar cualquier combinación de tamaños en las capas. En la Figura 14 se muestran cuatro estados de la red a lo largo del proceso de entrenamiento. En este caso se muestran las generaciones (o iteraciones) 0, 7000, 12000 y 20000, con una configuración de 8 neuronas de entrada (7 parámetros atmosféricos más el *bias*), 9 neuronas ocultas y una neurona de *output*.

Adicionalmente en cada representación se muestra el valor “real” (calculado por el modelo de entrenamiento en base a los parámetros atmosféricos de entrada), el valor “guess” (calculado por la red neuronal) y el error como porcentaje del rango posible de salidas [0.0, 1.0].

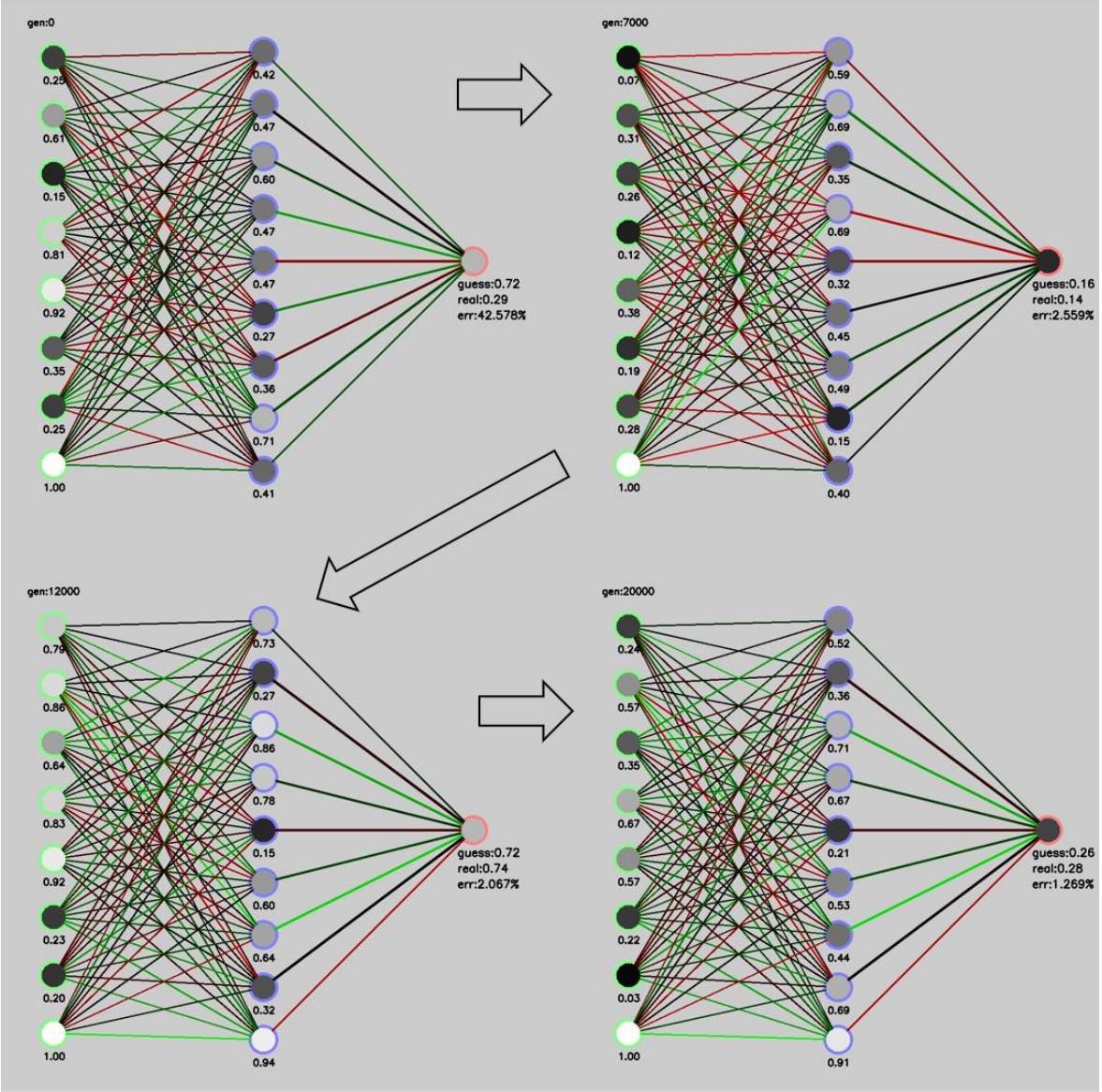

Figura 14 Evolución de la red tras 20000 iteraciones de entrenamiento

Una vez finalizado el proceso de entrenamiento el modelo de la red neuronal se guarda en un archivo *pickle* "net.plk". Para realizar una predicción en el sistema de *streaming* bastará con cargar el modelo desde el archivo y ejecutar la función *forward* con los datos que se quieran evaluar:

```
net = load_from_pickle('net.plk')
score = net.forward(dic_to_array(data ))[0][0]
```

Finalmente, ya que el modelo de entrenamiento no es demasiado complejo se ha decidido reducir el número de neuronas de la capa oculta a 5, haciendo el proceso de entrenamiento más rápido. A continuación (Figura 15), se puede observar el estado de la red con la nueva configuración tras 12000 iteraciones de entrenamiento.

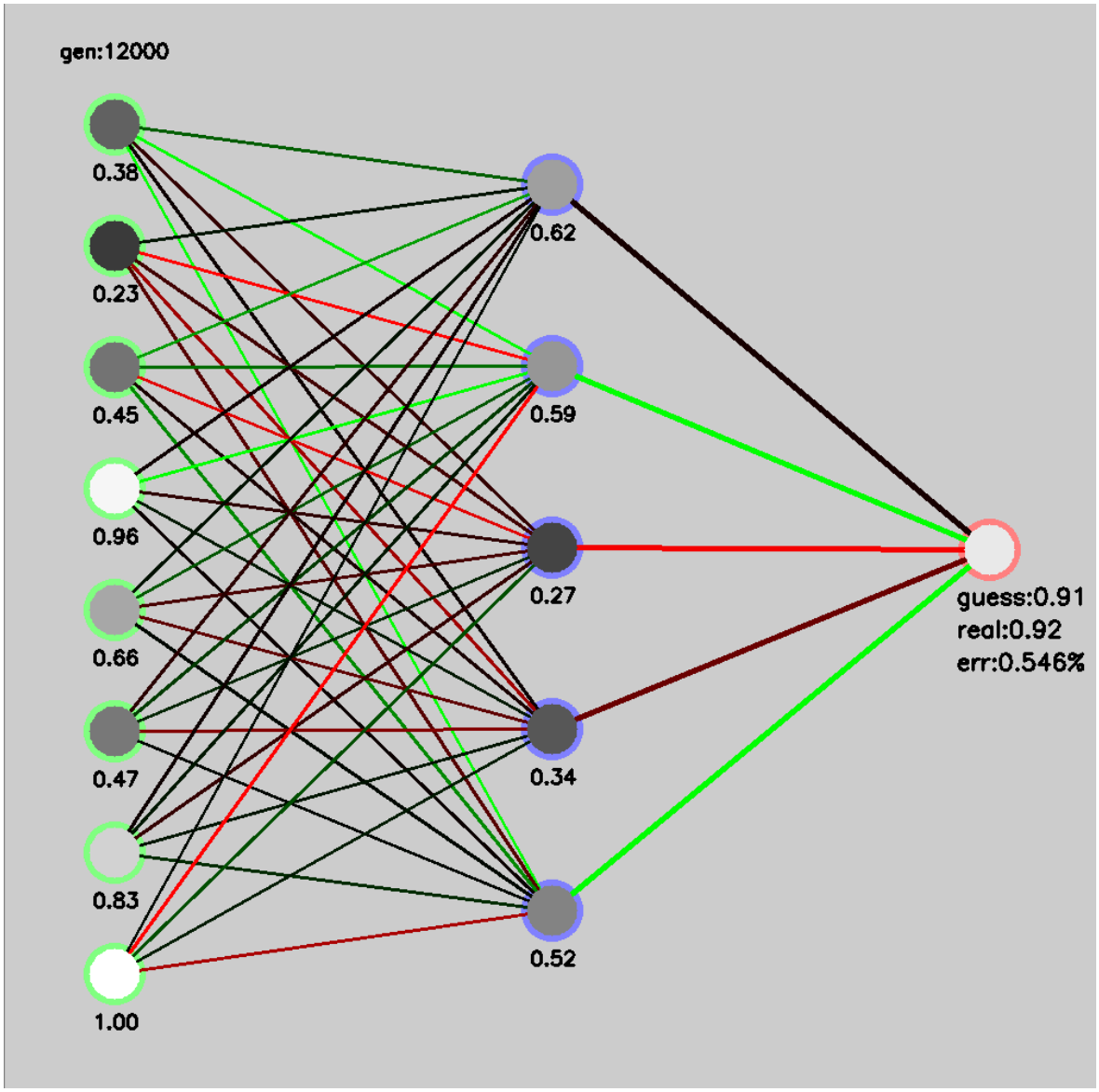

Figura 15 Configuración de la red con 5 neuronas ocultas

Analizando el error obtenido durante el proceso de entrenamiento, como se puede observar en la figura, se llega a la conclusión de que el algoritmo mejora muy rápidamente. Aproximadamente tras 2500 iteraciones de entrenamiento el error habitual permanece muy bajo, pero no sigue reduciéndose. Esto se debe a que la red neuronal se está entrenando con un modelo sintético y no con datos reales. En este proceso de entrenamiento en concreto el error medio en la puntuación, superada la cota de las 2500 iteraciones es de 0.017 (ó 1.7%). En la Figura 16 se puede observar el progreso del error al lo largo del proceso de entrenamiento.

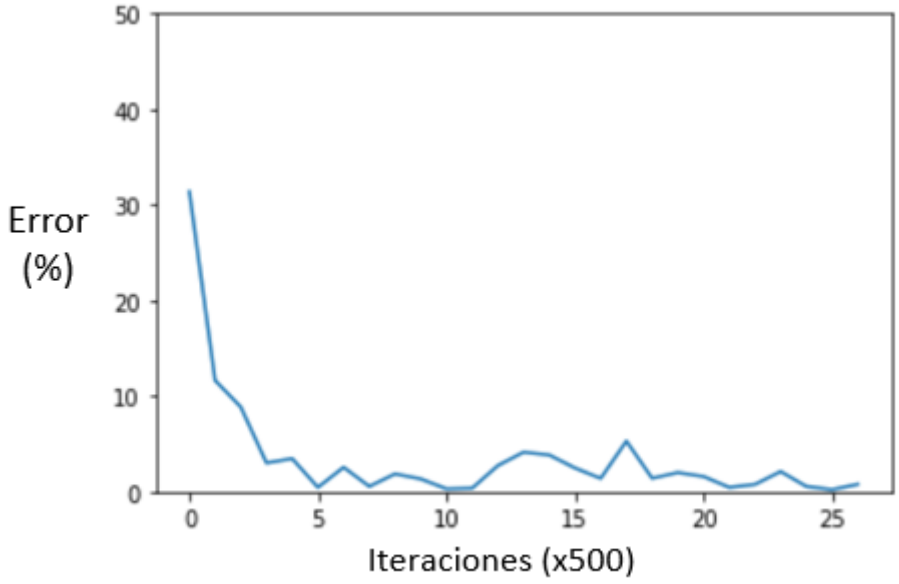

Figura 16 Valor del error a lo largo del entrenamiento

4.2.3 ENTORNO WEB

El entorno web proporciona la herramienta de visualización para que los usuarios o clientes puedan monitorizar las condiciones de demanda en un establecimiento, así como los parámetros atmosféricos. La aplicación web que se ha desarrollado puede dividirse en dos bloques. Un servidor web normal que muestra las páginas a los clientes y una herramienta que envía datos mediante *WebSocket* al explorador del cliente. Para esta conexión *WebSocket* se utilizará la librería de Python socket.io [33].

El servidor web y el sistema de conexión por *WebSocket* que se ha desarrollado está basado en un proyecto existente [23].

Podemos distinguir dos bloques principales en el desarrollo del entorno web: el *backend* o parte del servidor y el *frontend* o parte del cliente.

4.2.3.1 Backend

Para activar la aplicación web desarrollada basta con ejecutar un archivo "main.py". Al ejecutarse, se levanta el servidor *web* Flask y se inicia un hilo "listener" que se encarga del envío de información por *WebSocket*. Es necesario definir en este hilo la dirección y puerto de Kafka, el *topic* del que se desea consumir:

```
KAFKA_TOPIC = 'topicMKScore'
KAFKA_CONNECTION_STRING = 'master01:9092'
```

El código del *thread* encargado de la conexión *websocket* es el siguiente:

```
from threading import Thread
from kafka import KafkaConsumer

def listener(topic, callback, connection):

    print '---- listener thread started ----'
    consumer = KafkaConsumer(topic, bootstrap_servers=[connection])
    print 'now listening to kafka on ', connection, ' against topic: ', topic
    for msg in consumer:
        print '-----new message received from kafka-----'
        message = msg
        callback(message[6])

def launch(args, target):
    thread = Thread(target=target, args=args)
    thread.daemon = True
    thread.start()
```

De este modo, en el archivo "main.py" se lanza este *thread* y se levanta el servidor *web* como se puede ver a continuación:

```
import eventlet
```

```
eventlet.monkey_patch()
import socketio
from flask import Flask, current_app
import threads


KAFKA_TOPIC = 'topicMKScore'
KAFKA_CONNECTION_STRING = 'master01:9092'

sio = socketio.Server()
app = Flask(__name__)

@app.route('/<path:path>')
def index(path):
    print 'request on server: ', path
    return current_app.send_static_file(path)


@sio.on('connect')
def connect(sid, environment):
    print 'connected: ', sid

@sio.on('disconnect')
def disconnect(sid):
    print 'client disconnected: ', sid


def average_callback(average):
    sio.emit('event', average)


if __name__ == '__main__':
    threads.launch(
        (
            KAFKA_TOPIC,
            average_callback,
            KAFKA_CONNECTION_STRING
        ),
        threads.listener
    )

    app = socketio.Middleware(sio, app)
    print 'server started on port 8088'
    eventlet.wsgi.server(eventlet.listen(('', 8088)), app)
```

4.2.3.2 Frontend

En cuanto al diseño estético del *frontend* se ha desarrollado con la herramienta Mobirise [34] que permite incorporar estructuras dinámicas a la página rápidamente. Como ya se ha

mencionado anteriormente la aplicación *web* consta de dos páginas: una página de bienvenida y el *dashboard web* (al que se accede desde la página de bienvenida). Son necesarios dos archivos javascript para la comunicación mediante *WebSocket* con el *cluster Big Data*. Por un lado, el archivo "socket.io.js" que no se ha modificado en este proyecto. En este archivo se establecen las funciones necesarias para la comunicación *WebSocket*. Por otro lado, se ha desarrollado otro archivo javascript que se encarga de recibir los datos del cluster, dibujar los gráficos en la página y actualizar los datos. Para la creación de estos gráficos se ha utilizado la librería de gráficos Google Charts [35].

En la página de *dashboard* se pueden distinguir 5 funciones principales:

- Mostrar en tiempo real los parámetros atmosféricos en diferentes indicadores: (Figura 17).

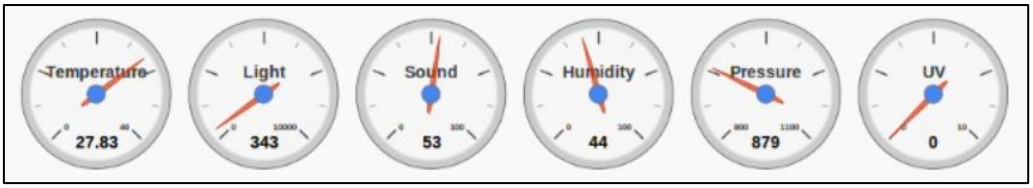

Figura 17 Indicadores atmosféricos

- Mostrar en un indicador la valoración de la demanda hecha por la red neuronal ("score"): (Figura 18).

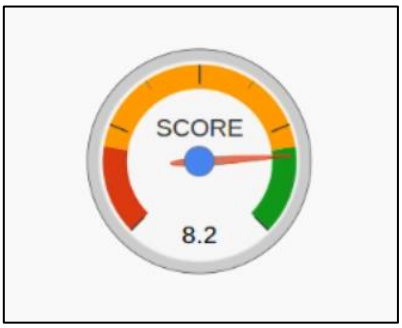

Figura 18 Indicador de la predicción de demanda

- Mostrar una gráfica histórica que compare el valor de demanda calculado por la red neuronal ("Score") y el valor calculado por el algoritmo de entrenamientos ("real"): (Figura 19)

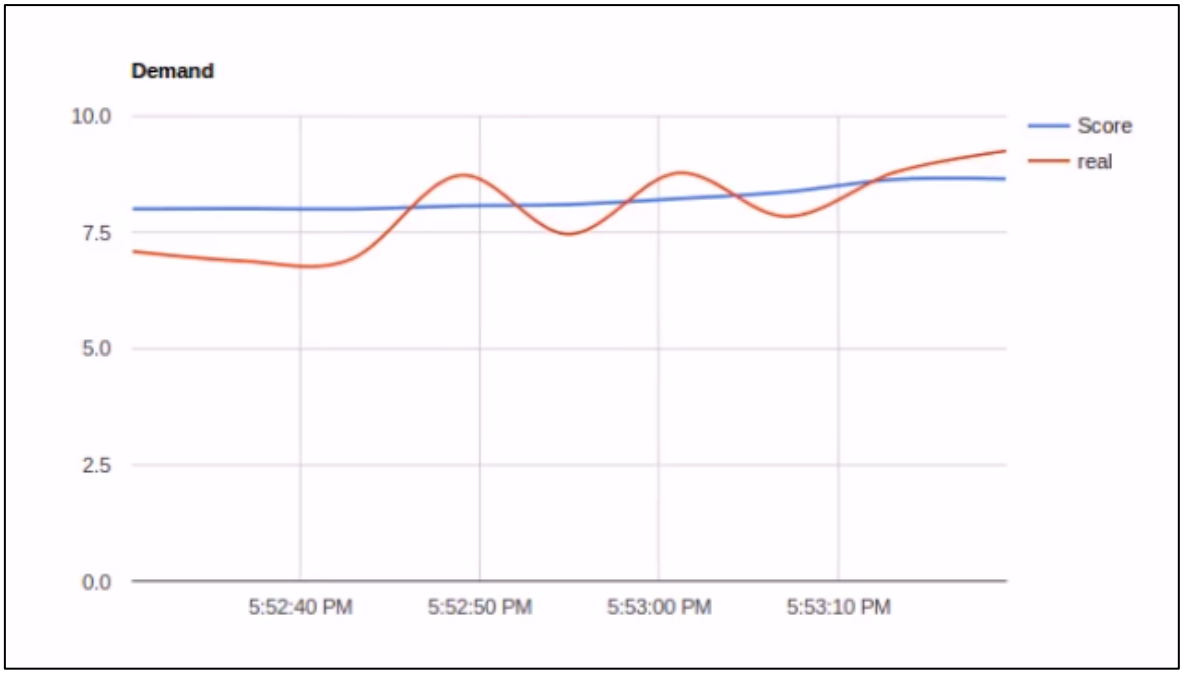

Figura 19 Comportamiento de la demanda real y estimada

- Indicar de forma textual la diferencia entre los valores "Score" y "real" (Figura 20). Si "real" supera a "Score" en más de 3 puntos se mostrará el texto del color verde para indicar que la demanda se está comportando mejor de lo esperado. Por el contrario, si el valor "real" es inferior a "Score" el texto se mostrará en rojo, indicando que la demanda se está comportando peor de lo esperado. Si no se cumplen las condiciones se considera que la demanda se está comportando de acuerdo con lo esperado por lo que se muestra el texto de color negro.

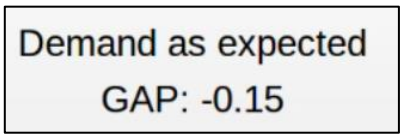

Figura 20 Indicador textual del comportamiento de la demanda

- Por último, se añaden 3 botones para modificar artificialmente el valor de la demanda real (Figura 21). Esto tiene un propósito puramente ilustrativo, ya que como la red neuronal se ha entrenado con un modelo sintético siempre predecirá correctamente su comportamiento. Modificando de forma artificial el parámetro "real" (añadiendo un *offset* positivo o negativo) podemos simular como serían condiciones de menor o mayor demanda de la esperada. Con un botón se resta un punto al valor real, con otro se añade un punto y con el último se resetea este *offset*.

Figura 21 Botones para el control de la demanda real

Capítulo 5. ANÁLISIS DE RESULTADOS

En primer lugar, se analiza el correcto comportamiento de cada elemento del sistema. Se seguirá el orden seguido en desarrollo de estos.

5.1 GATEWAY IOT

Tras arrancar el *gateway* IoT comprobamos que funciona correctamente. Podemos acceder a la página de control desde la cual se activa la recolección de datos (Figura 22):

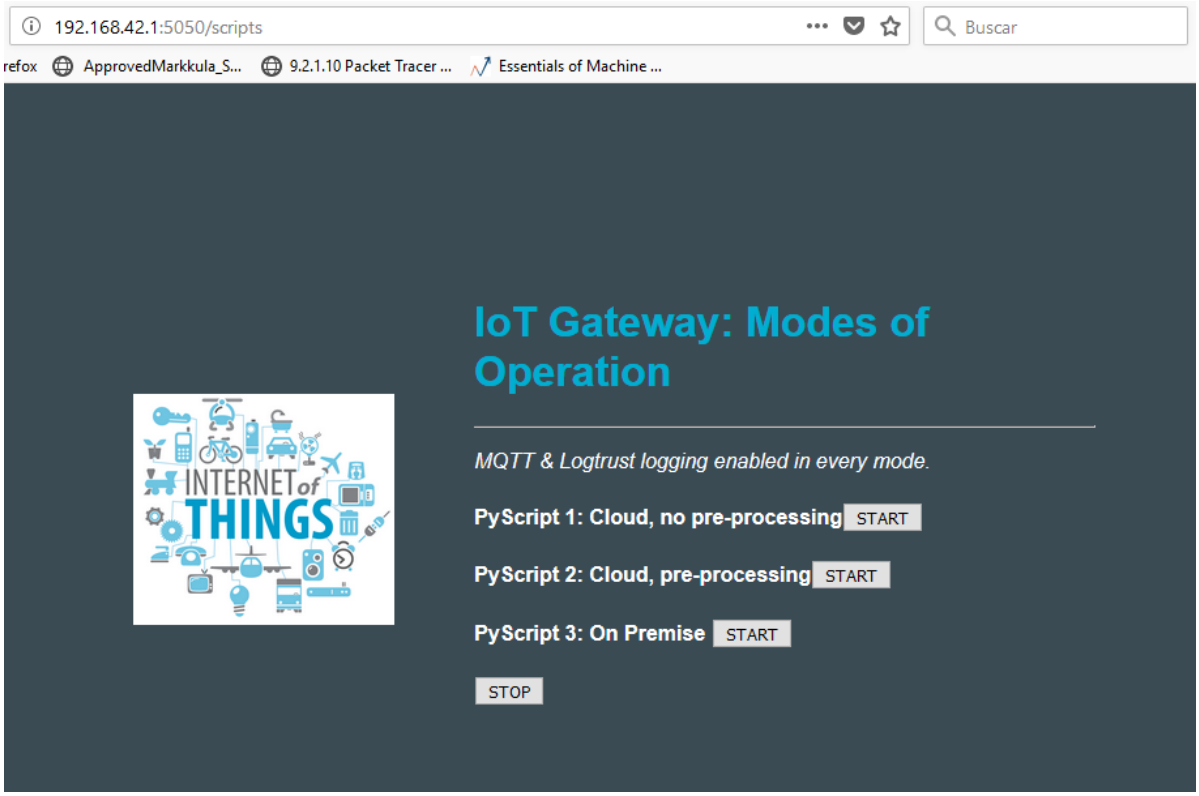

Figura 22 Panel de control del gateway IoT

Por otro lado, observando el registro de este servidor *web* observamos que se inicia un *thread*, correspondiente con el sensor que se está utilizando para la prueba (Figura 23).

```
(flask) root@raspberrypi:/usr/local/lib/python2.7/dist-packages/fase2tfm/control
panel# python -m flask run --host 192.168.42.1 --port 5050
 * Serving Flask app "welcome"
 * Forcing debug mode on
 * Running on http://192.168.42.1:5050/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 288-513-626
192.168.42.17 - - [03/Jul/2018 12:58:03] "GET / HTTP/1.1" 200 -
192.168.42.17 - - [03/Jul/2018 12:58:03] "GET /static/stylesheets/style.css HTTP
/1.1" 200 -
192.168.42.17 - - [03/Jul/2018 12:58:03] "GET /static/images/iot.png HTTP/1.1" 2
00 -
192.168.42.17 - - [03/Jul/2018 12:58:03] "GET /favicon.ico HTTP/1.1" 404 -
192.168.42.17 - - [03/Jul/2018 12:58:04] "GET /scripts HTTP/1.1" 200 -
192.168.42.17 - - [03/Jul/2018 12:58:10] "GET /startscp3 HTTP/1.1" 200 -
Starting thread <Thread(Thread-1, initial)> for 30096:00:0b:57:36:75:90
```

Figura 23 Terminal al activar el gateway IoT

5.2 PRODUCER KAFKA

Tras ejecutar el módulo Scala se puede observar en el terminal que se están recibiendo correctamente datos MQTT del *gateway* y se están enviado al *topic* Kafka “topicMK”: (Figura 24).

```
spark@hadoopSlave01:~/sources/sparkCode/DemostrableIoT/demostrable/mkconnector$
./run.sh
log4j:WARN No appenders could be found for logger (org.apache.kafka.clients.prod
ucer.ProducerConfig).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more in
fo.
Receiving data from mqtt topic 'mqtt', Message: name
Sent to kafka topic 'topicMK'
Receiving data from mqtt topic 'mqtt', Message: name
Sent to kafka topic 'topicMK'
Receiving data from mqtt topic 'mqtt', Message: name
Sent to kafka topic 'topicMK'
Receiving data from mqtt topic 'mqtt', Message: name
Sent to kafka topic 'topicMK'
Receiving data from mqtt topic 'mqtt', Message: name
Sent to kafka topic 'topicMK'
```

Figura 24 Terminal al activar el conector MQTT-Kafka

5.3 SISTEMA DE PREDICCIÓN DE DEMANDA

Al ejecutar el módulo de predicción de demanda podemos observar en el terminal que los datos se están consumiendo correctamente del *topic* “topicMK” (Figura 25). Se puede ver también el mensaje recibido con los parámetros atmosféricos. Tras realizar la predicción se imprime el valor “real” y el valor “score”. Por último, se indica que los datos han sido enviados correctamente al *topic* “topicMKScore”.

```
spark@hadoopSlave01:~/sources/sparkCode/DemostrableIoT/demostrable/Neural$ pytho
n test_consumer.py
----parameters received from topic: topicMK----
name:Thunder Sense #30096%sound:50%co2:0%temp:25.16%voc:0%humi:30%light:372%pres
:883%uvin:0
        real value: 0.782467301064
        score value: 0.821976669056
----message sent to topic: topicMKScore----
----parameters received from topic: topicMK----
name:Thunder Sense #30096%sound:50%co2:0%temp:25.16%voc:0%humi:30%light:373%pres
:883%uvin:0
        real value: 0.804192997388
        score value: 0.82199775816
----message sent to topic: topicMKScore----
----parameters received from topic: topicMK----
name:Thunder Sense #30096%sound:50%co2:0%temp:25.16%voc:0%humi:31%light:372%pres
:883%uvin:0
        real value: 0.774266832564
        score value: 0.819912744505
----message sent to topic: topicMKScore----
```

Figura 25 Terminal al activar la red neuronal

5.4 SISTEMA DE VISUALIZACIÓN WEB

En primer lugar, al activar la aplicación web podemos ver en el registro de terminal que se ejecuta el *thread* "*listener*" correctamente (Figura 26). Posteriormente se indica que se levanta el servidor, especificando la dirección y el puerto. Finalmente podemos observar en el terminal que están recibiendo los mensajes Kafka del *topic* "topicMKScore".

```
spark@hadoopSlave01:~/sources/sparkCode/DemostrableIoT/web/webapp$ python main.p
y
---- listener thread started ----
server started on port 8001
(5032) wsgi starting up on http://0.0.0.0:8001
now listening to kafka on  192.168.42.21:9092  against topic:  topicMKScore
-----new message received from kafka-----
-----new message received from kafka-----
-----new message received from kafka-----
-----new message received from kafka-----
-----new message received from kafka-----
(5032) accepted ('127.0.0.1', 43470)
request on server:  index.html
127.0.0.1 - - [03/Jul/2018 14:59:42] "GET /index.html HTTP/1.1" 200 14969 0.0031
81
request on server:  assets/web/assets/mobirise-icons/mobirise-icons.css
127.0.0.1 - - [03/Jul/2018 14:59:42] "GET /assets/web/assets/mobirise-icons/mobi
rise-icons.css HTTP/1.1" 200 7916 0.001986
(5032) accepted ('127.0.0.1', 43472)
(5032) accepted ('127.0.0.1', 43474)
(5032) accepted ('127.0.0.1', 43476)
(5032) accepted ('127.0.0.1', 43478)
(5032) accepted ('127.0.0.1', 43480)
```

Figura 26 Terminal al activar la herramienta de visualización

Una vez en el explorador *web* se accede directamente a la página de bienvenida (Figura 27). En esta se muestra cierta información acerca del sistema y se ofrece un botón (“GO!”) para acceder al *dashboard web*.

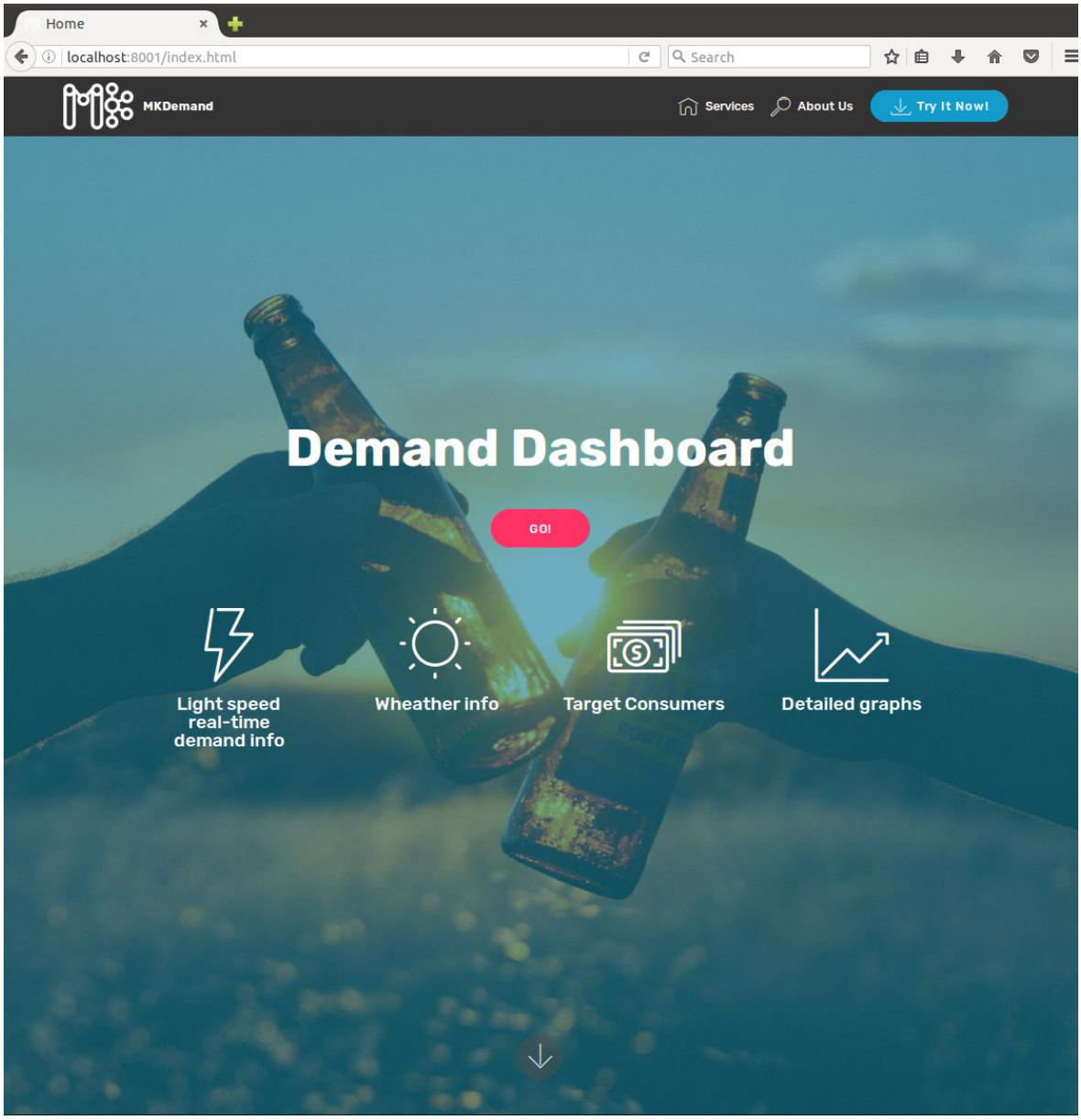

Figura 27 Página de bienvenida de la herramienta de visualización

Si se hace *click* en dicho botón se accede finalmente al *dashboard web*. En él podemos encontrar diferentes indicadores y funciones. En primer lugar, podemos ver que los indicadores superiores muestran correctamente los parámetros atmosféricos medidos por los sensores IoT. Debajo de los indicadores atmosféricos se muestra otro indicador, esta vez

muestra el valor de demanda calculado por la red neuronal (multiplicado por 10 con propósitos ilustrativos). Seguidamente se encuentra una gráfica cronológica que muestra tanto el valor score (calculado por la red neuronal) en azul como el valor real (calculado por el modelo de entrenamiento) en rojo. Esta gráfica incorpora tres botones que permiten modificar artificialmente el comportamiento del valor real añadiendo un *offset* positivo o negativo. Esto es necesario ya que al haber entrenado la red neuronal con un modelo sintético esta siempre realizará predicciones correctas. Se añaden estos controles para ilustrar el funcionamiento del sistema cuando la demanda no se comporta de acuerdo con lo estimado. Por último, se muestra un mensaje que indica al usuario o cliente si la demanda se está comportando como se espera o no. Si la demanda real (calculada por el modelo de entrenamiento) está tres puntos por debajo de la demanda esperada (calculada por la red neuronal) el mensaje se mostrará en rojo, si la demanda real supera la esperada se mostrará en verde, y si son parecidas el mensaje se mostrará en negro. La página de *dashboard* completa se muestra a continuación: (Figura 28).

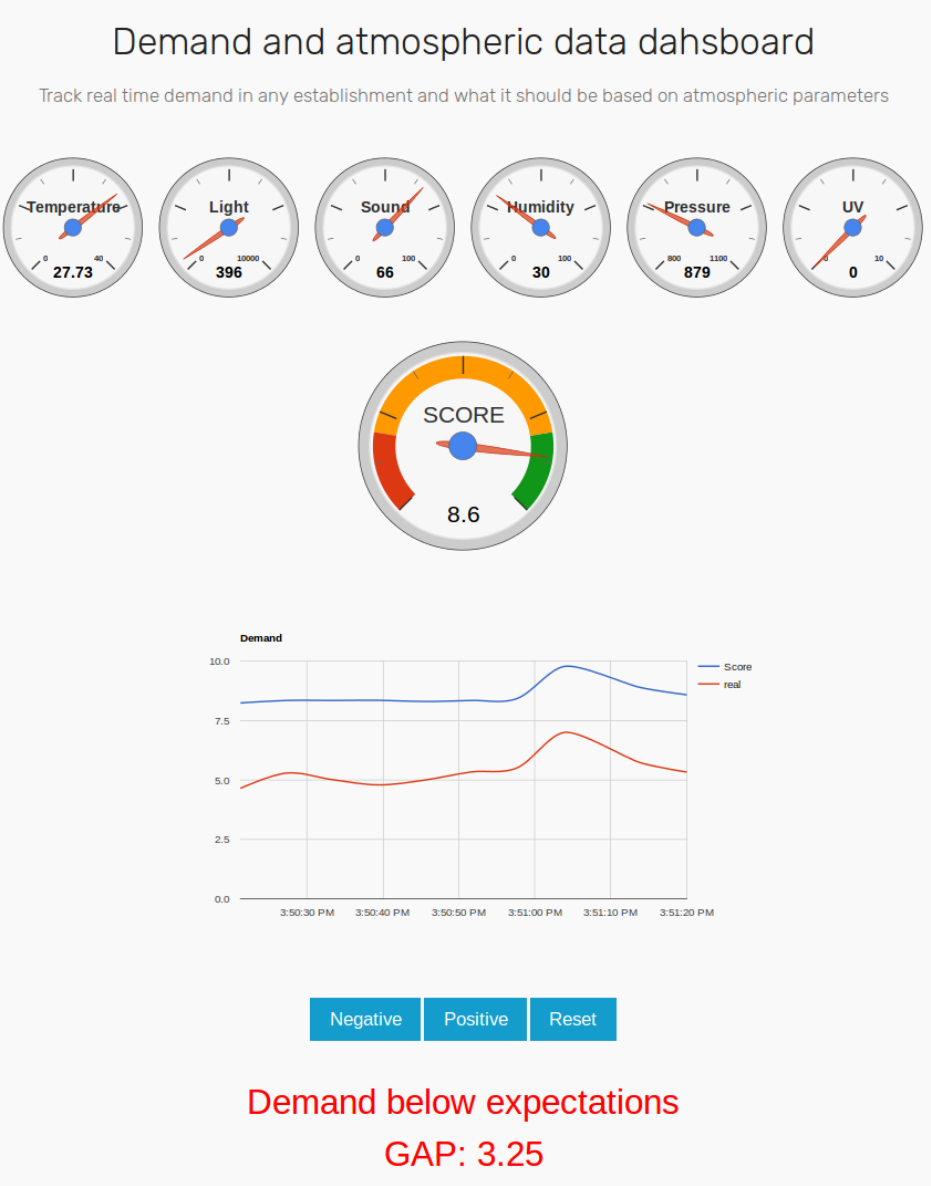

Figura 28 Herramienta de visualización mostrando demanda por debajo de las expectativas

En la siguiente figura se muestra el comportamiento de la gráfica si se aplica un *offset* negativo al comienzo de la visualización y se va eliminando progresivamente (Figura 29).

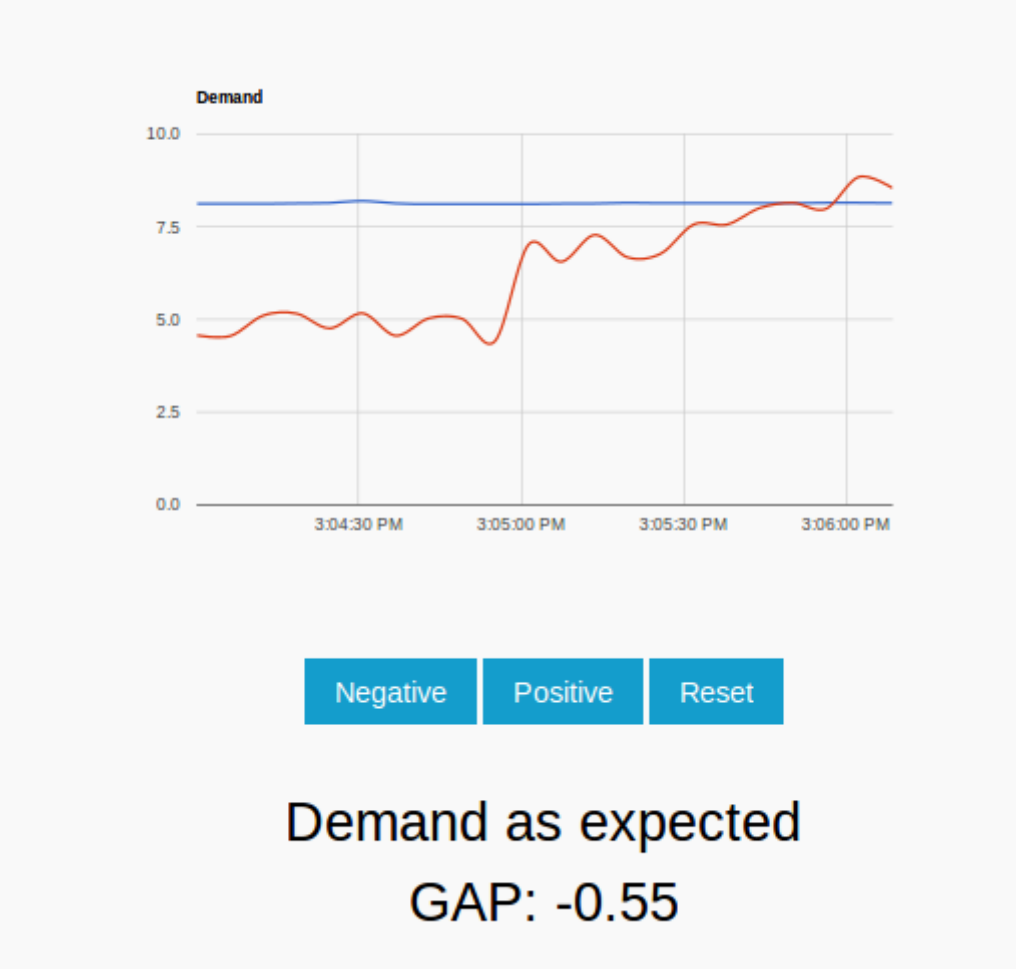

Figura 29 Herramienta de visualización tras aplicar cambios a la demanda real a través de los botones inferiores

Capítulo 6. CONCLUSIONES Y TRABAJOS FUTUROS

6.1 CONCLUSIONES

Se ha desarrollado e implantado una plataforma viable para el procesado de datos IoT y visualización de una estimación de demanda en tiempo real.

En primer lugar, se ha conectado con éxito la plataforma IoT existente al *cluster Big Data* de la Universidad Pontificia Comillas.

Se ha desarrollado con éxito una herramienta capaz de volcar los datos MQTT recibidos en una cola Kafka. Esto permite la gestión y procesado de los datos IoT en tiempo real, así como el consumo de estos por la herramienta de visualización.

Se ha desarrollado con éxito una herramienta *Machine Learning* capaz de consumir datos en *streaming*, procesarlos y publicarlos, de manera que puedan ser visualizados por los clientes o usuarios en tiempo real.

Por último, se ha diseñado y desarrollado una aplicación web capaz de mostrar en tiempo real una serie de indicadores y gráficos que ilustran el comportamiento de la demanda en un establecimiento. Mediante este *dashboard web* se pueden monitorizar en tiempo real múltiples parámetros atmosféricos en un establecimiento. Del mismo modo, se muestra una gráfica comparativa que permite al usuario o cliente ver las condiciones de demanda real y esperada en el establecimiento. Adicionalmente, si la demanda real está muy por debajo de la esperada se muestra un mensaje a modo de alerta para que se tomen las medidas que se consideren oportunas.

En resumen, la plataforma desarrollada e implementada puede suponer una herramienta de gran valor. Con ella, se puede reaccionar de forma casi instantánea a los cambios en el

comportamiento de la demanda mediante diferentes estrategias. Se podría, por ejemplo, ofrecer "ofertas exprés" en momentos en los que se espere que decaiga la demanda o promocionar los productos con más margen de beneficio para el establecimiento cuando se sepa que la demanda va a subir.

6.2 TRABAJOS FUTUROS

Aunque en este proyecto se han cumplido todos los objetivos establecidos existen diversos aspectos en los que la plataforma puede ser expandida y mejorada.

En primer lugar, se podría ampliar el sistema de *streaming*, creando más *topics* Kafka para soportar los datos de diferentes establecimientos. De este modo mediante una estructura jerárquica de *topics* se podría distinguir entre diferentes franquicias y dentro de cada franquicia los diferentes establecimientos. Esto es muy necesario si se decidiera instalar el sistema en un gran número de establecimientos, permitiendo sacar el máximo provecho a las capacidades de procesado en paralelo del *cluster Big Data*.

Una gran limitación en este proyecto ha sido el no disponer de datos reales de facturación para estudiar la demanda. Si se pudiesen integrar sensores en un establecimiento y se tuviera acceso a los datos de facturación se podría adaptar y reentrenar el algoritmo de *Machine Learning* para realizar predicciones de demanda basadas en datos reales.

Por otro lado, en cuanto al procesado de los datos atmosféricos, hay varios aspectos que se podrían mejorar y ampliar pero que han quedado fuera del alcance de este proyecto. Si se integran en el sistema un gran número de establecimientos sería necesario adaptar el algoritmo de *Machine Learning* para poder procesar los datos en paralelo mediante Pyspark. Por otro lado, también sería conveniente ampliar el algoritmo para ponderar más aspectos que puedan afectar a la demanda: tamaño del establecimiento, localización, fecha, acontecimientos deportivos, etc.

Por último, si se implementase el sistema en varias franquicias sería necesario adaptar la herramienta de visualización. En primer lugar, se debería implementar un sistema de seguridad que requiera de cierta acreditación para acceder a un determinado *dashboard*. Por otro lado, se debería ampliar la aplicación *web* para dar la posibilidad de comparar el rendimiento de diferentes establecimientos de una misma franquicia.

Capítulo 7. BIBLIOGRAFÍA

[1] «mqtt.org,» [En línea]. Available: http://mqtt.org/.

[2] «kafka.apache.org,» [En línea]. Available: https://kafka.apache.org/intro.

[3] T. Gupta, «towardsdatascience.com,» [En línea]. Available: https://towardsdatascience.com/deep-learning-feedforward-neural-network-26a6705dbdc7.

[4] «developer.mozilla.org,» [En línea]. Available: https://developer.mozilla.org/es/docs/WebSockets-840092-dup.

[5] B. Smith, «businessoptics.biz,» [En línea]. Available: https://www.businessoptics.biz/blog/demand-forecasting-adding-your-bottom-line-using-big-data-prescriptive-analytics/.

[6] N. Joshi, «allerin.com,» [En línea]. Available: https://www.allerin.com/blog/4-ways-big-data-is-impacting-the-world-of-e-commerce.

[7] R. Gupta, «ibm.com,» [En línea]. Available: https://www.ibm.com/developerworks/community/blogs/5things/entry/5_things_to_know_about_mqtt_the_protocol_for_internet_of_things?lang=en.

[8] J. A. Y. Gálvez, «geekytheory.com,» [En línea]. Available: https://geekytheory.com/que-es-mqtt.

[9] «silabs.com,» [En línea]. Available: https://www.silabs.com/products/development-tools/thunderboard/thunderboard-sense-kit.

[10] «ti.com,» [En línea]. Available: http://www.ti.com/ww/en/wireless_connectivity/sensortag/tearDown.html.

[11] «web.archive.org,» [En línea]. Available: https://web.archive.org/web/20140214120404/http://www.bluetooth.com/Pages/low-energy-tech-info.aspx.

[12] «raspberrypi.org,» [En línea]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[13] «docs.scala-lang.org,» [En línea]. Available: https://docs.scala-lang.org/?_ga=2.180069485.832139265.1531053739-1945807691.1531053739.

[14] «python.org,» [En línea]. Available: https://www.python.org/about/.

[15] «numpy.org,» [En línea]. Available: http://www.numpy.org/.

[16] J. Collins, «medium.com,» [En línea]. Available: https://medium.com/deeper-learning/glossary-of-deep-learning-bias-cf49d9c895e2.

[17] «flask.pocoo.org,» [En línea]. Available: http://flask.pocoo.org/.

[18] «proyectosagiles.org,» [En línea]. Available: https://proyectosagiles.org/que-es-scrum/.

[19] «bitbucket.org,» [En línea]. Available: https://DavidContrerasICAI@bitbucket.org/DavidContrerasICAI/iotbigdata.git.

[20] N. Donges, «towardsdatascience.com,» [En línea]. Available: https://towardsdatascience.com/the-random-forest-algorithm-d457d499ffcd.

[21] «eclipse.org,» [En línea]. Available: https://www.eclipse.org/paho/.

[22] Abhinav, «cloudxlab.com,» [En línea]. Available: https://cloudxlab.com/blog/real-time-analytics-dashboard-with-apache-spark-kafka/.

[23] «github.com,» [En línea]. Available: https://github.com/emumba-com/live_twitter_sentiment_analysis/tree/master/webapp.

[24] «spark.apache.org,» [En línea]. Available: https://spark.apache.org/docs/0.9.0/python-programming-guide.html.

[25] «spark.apache.org,» [En línea]. Available: http://spark.apache.org/docs/2.0.0/api/python/pyspark.mllib.html.

[26] «jaceklaskowski.gitbooks.io,» [En línea]. Available: https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-SparkContext.html.

[27] «iamtrask.github.io,» [En línea]. Available: https://iamtrask.github.io/2015/07/12/basic-python-network/.

[28] J. F. Puget, «ibm.com,» [En línea]. Available: https://www.ibm.com/developerworks/community/blogs/jfp/entry/What_Language_Is_Best_For_Machine_Learning_And_Data_Science?lang=en.

[29] «en.wikipedia.org,» [En línea]. Available: https://en.wikipedia.org/wiki/Sigmoid_function#/media/File:Logistic-curve.svg.

[30] «neuralnetworksanddeeplearning.com,» [En línea]. Available: http://neuralnetworksanddeeplearning.com/chap2.html.

[31] «docs.python.org,» [En línea]. Available: https://docs.python.org/3/library/pickle.html.

[32] «opencv.org,» [En línea]. Available: https://opencv.org/.

[33] «socket.io,» [En línea]. Available: https://socket.io/docs/#.

[34] «mobirise.com,» [En línea]. Available: https://mobirise.com/es/.

[35] «developers.google.com,» [En línea]. Available: https://developers.google.com/chart/.