Facultad de Ciencias

SISTEMA DE CALIBRACIÓN AUTOMÁTICA DE PREDICCIONES METEOROLÓGICAS MEDIANTE FILTROS DE KALMAN

(Automatic calibrating system for weather forecast through Kalman filters)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMATICA

Autor: Antonio Minondo Tshuma

Director: Andrés Iglesias Prieto

Co-Director: Daniel San Martín Segura

Junio - 2016

Agradecimientos

Lo primero de todo, he de agradecer a mis padres por haber puesto mi educación ante todo, aunque no siempre haya estado de acuerdo con ellos no hay duda de que gracias a su supervisión he llegado mas lejos de lo que hubiera conseguido jamás por mi cuenta.

Quiero agradecer a la empresa Predictia por la oportunidad otorgada para realizar mi proyecto de fin de grado con ellos y formarme como profesional del sector de la informática y en especial a Daniel San Martín por sus sugerencias y comentarios, que han contribuido, con toda seguridad, a mejorar el resultado final de este proyecto.

Asimismo, me gustaría agradecerles a él y a Andrés Iglesias la labor de revisión que han realizado sobre esta memoria.

Resumen

El objetivo de este Trabajo de Fin de Grado es desarrollar un sistema que permita calibrar automáticamente predicciones meteorológicas a corto plazo. Para ello, se hará uso de observaciones meteorológicas de diferentes fuentes (como la Agencia Estatal de Meteorología) y predicciones de modelos numéricos de predicción meteorológica (como el Global Forecast System).

Se han implementado procesos de carga para extraer datos de ambas fuentes de forma periódica, a los cuales se aplicará una metodología de calibración basada en filtros de Kalman. El resultado de esta calibración será una predicción corregida en base al error cometido en predicciones anteriores.

Finalmente, se ha desarrollado una aplicación móvil híbrida para visualizar gráficamente las predicciones capturadas.

Palabras clave: filtro de Kalman, predicción meteorológica, calibración automática de predicciones, aplicación móvil híbrida, proceso ETL

Summary

The aim of this Degree's Final Project is to develop a system that will be able to automatically calibrate short-term weather predictions. To do this, we will make use of meteorological observations from diferent sources (like from AEMET) and predictions from numerical weather prediction models (such as the Global Forecast System).

We have implemented ways of extracting the data from both sources on a regular basis, to which we apply a calibration methodology based on the Kalman filter. The result of this gives us a corrected forecast based on the errors made in previous predictions.

Finally, we have developed a hybrid mobile application which we employ as an interface to graphically view the captured weather forecast.

Keywords: Kalman filter, weather forecast, automatic calibration of predictions, hybrid mobile application, ETL process

Índice

Índice de figuras

1. Introducción

La meteorología es la ciencia encargada del estudio de la atmósfera, de sus propiedades y de los fenómenos que en ella tienen lugar. Se basa en el conocimiento de una serie de magnitudes, o variables meteorológicas, como la temperatura, la presión atmosférica o la humedad en un determinado momento y lugar [1].

Es conocido que el clima ha sido un condicionante determinante en la historia de la humanidad. Aún hoy, la meteorología afecta a una gran parte de nuestras actividades diarias. Es un factor crítico a la hora de planear nuestras actividades de ocio y también económicas. Por tanto, la predicción meteorológica es y ha sido objeto de un gran interés al que se han dedicado importantes esfuerzos técnicos y científicos.

1.1. Contexto

Como ya hemos mencionado, la predicción meteorológica es una cuestión de mucha relevancia en nuestros días, ya que además de regir nuestras decisiones mas cotidianas, se emplea a la hora de planificar las labores de prevención por parte de los servicios de protección civil o al organizar eventos a gran escala en el exterior.

La predicción meteorológica consiste en la determinación anticipada de los valores de las variables meteorológicas como la temperatura, la presión, la humedad, la nubosidad, la precipitación, etc., que afectarán a una determinada región. Puede realizarse mediante técnicas estadísticas, pero la forma más común, y la que habitualmente ofrece mejores resultados, está basada en la resolución de las ecuaciones matemáticas correspondientes a las leyes físicas que describen el comportamiento de la atmósfera llamadas modelos numéricos de predicción meteorológica. Son sistemas que usan datos meteorológicos actuales para pasarlos por complejos modelos físico-matemáticos de la atmósfera que predicen la evolución meteorológica. Aunque los primeros esfuerzos por realizar predicciones utilizando este tipo de metodologías se remontan a la década de 1920, no ha sido hasta la llegada de la computación y de la simulación por ordenador cuando se ha podido implementar modelos que realicen cálculos en tiempo real [2]. Hoy día, la meteorología es una ciencia tremendamente avanzada y debido al aumento de la capacidad de cálculo mediante potentes supercomputadores se han conseguido mejorar y refinar los modelos numéricos de predicción del tiempo permitiendo así que en los últimos tiempos la meteorología experimentase una auténtica revolución. Todo esto acompañado del desarrollo de los satélites y radares meteorológicos junto con otros sistemas de teledetección ha mejorado sensiblemente la precisión y la capacidad de observación del tiempo.

Una vez resueltas estas ecuaciones, partiendo de las observaciones iniciales del estado de la atmósfera, se obtiene una descripción del futuro estado de la atmósfera. De este modo puede elaborarse una predicción meteorológica con la que conocer el tiempo de las próximas horas o días.

Ahora bien, estas ecuaciones son verdaderamente complicadas de resolver, puesto que se trata de ecuaciones para las que no siempre existe una solución exacta o son muy costosas de resolver en términos computacionales. A pesar de lo mucho que las técnicas de predicción meteorológica han avanzado, las predicciones siguen teniendo limitaciones tanto por errores en los modelos de predicción como por el carácter no lineal del sistema. Por tanto, es habitual que los modelos numéricos numéricos presenten ciertos errores sistemáticos en ciertos parámetros meteorológicos.

1.2. Objetivo del proyecto

Estas limitaciones hacen necesario, para ciertas aplicaciones, el uso de técnicas de calibración para reducir el error de las predicciones. Existen numerosas técnicas estadísticas para calibrar predicciones meteorológicas. Una de las más populares se base en el uso de filtros de Kalman [3].

El objetivo de este proyecto es desarrollar un sistema configurable y extendible que calibre predicciones meteorológicas en base a datos observados. Para ello, se desarrollarán procesos de captura de información y calibración con la flexibilidad necesaria como para poder hacer uso de datos meteorológicos de diversas fuentes con el mínimo esfuerzo posible.

2. Conceptos

2.1. NetCDF

NetCDF (Network Common Data Form) es un formato de archivo de intercambio de datos científicos, emplea una librería de funciones de acceso a datos para almacenar y recuperar datos en forma de matrices multidimensionales.

El convenio de Clima y Predicciones (Climate and Forecast Metadata, CF) son los convenios de metadatos para los datos de las ciencias de la tierra. Tienen el objetivo de promover el procesamiento y la distribución de los archivos creados con el NetCDF Application Programmer Interface(API). Los convenios definen los metadatos que se incluyen en el mismo fichero que los datos (haciendo que el archivo netCDF sea "auto-descriptivo"), metadatos que proporcionan una descripción definitiva de lo que representan los datos de cada variable, y de las propiedades espaciales y temporales de los datos. Esto permite a los usuarios de datos de diferentes fuentes decidir qué datos son comparables, y permite la creación de aplicaciones grandes capacidades de extracción, remallado, y de visualización [4].

2.2. NOAA Operational Model Archive and Distribution System

Para hacer frente a una necesidad creciente de acceder remotamente a un volumen alto de predicciones meteorológicas numéricas y a los datos de modelos climáticos globales, el Centro Nacional de Datos Climáticos (National Climatic Data Center, NCDC), junto con los Centros Nacionales de Predicción Ambiental (National Centers for Environmental Prediction, NCEP) y el Laboratorio de Dinámica de Fluidos Geofísicos (Geophysical Fluid Dynamics Laboratory, GFDL), inició el proyecto NOAA Operational Model Archive and Distribution System (NOMADS). NOMADS aborda las necesidades de acceso de datos del modelo como se indica en el Programa de Investigación Meteorológica de los EE.UU. (U.S. Weather Research Program , USWRP) para el Plan de Implementación para la Investigación en el Pronóstico de Precipitación Cuantitativa y Asimilación de Datos para "redimir el valor práctico de los resultados de investigación y facilitar su transferencia a las operaciones" [5].

NOMADS es una red de servidores de datos que utilizan tecnologías establecidas y emergentes y el protocolo de transporte de datos llamado OPeNDAP 2.5 para acceder e integrar diversos datos almacenados en repositorios distribuidos geográficamente en formatos heterogéneos. Permite el intercambio y la intercomparación de los resultados del modelo y es un importante esfuerzo de colaboración, que abarca varios organismos gubernamentales e instituciones académicas.

2.3. Global Forecast System

El Sistema Global de Predicción (más conocido por GFS, Global Forecast System) es un modelo numérico de predicción meteorológica creado y utilizado por la Administración Nacional Oceánica y Atmosférica (National Oceanic and Atmospheric Administration, NOAA). El modelo GFS tiene mas de una modelización y el modelo matemático que nosotros empleamos se actualiza cuatro veces al día, las predicciones van hasta los 10 días y tiene una resolución espacial de aproximadamente 0.25 grados y temporal de 3 horas. Eso quiere decir que la superficie terrestre está dividida horizontalmente en una malla que es de unos 28 kilómetros y que produce predicciones de tres en tres horas con un alcance de 10 días.

El GFS es un modelo con cobertura global cuya producción de salidas de predicción están disponible gratuitamente y bajo dominio público a través de Internet y es uno de los cuatro modelos más utilizados para la predicción meteorológica a medio plazo.

2.4. Filtro de Kalman

El Filtro de Kalman es una potente herramienta desarrollada por Rudolf E. Kalman en 1960 [6] empleada para controlar los sistemas ruidosos específicamente cuando están sometidos a ruido blanco aditivo. Según Andrews (2001) [7], el estimador resultante es estadísticamente óptimo con respecto a cualquier función cuadrática de estimación del error. La ventaja de este algoritmo respecto a otros, es que el filtro de Kalman es capaz de escoger la forma óptima cuando se conocen las varianzas de los ruidos que afectan al sistema.

Cuando el sistema dinámico no es lineal (como en la mayoría de los casos), se aplica una extensión del filtro con un proceso de linealización, y se conoce como Filtro de Kalman Extendido (FKE) (Haykin, 1999) [8]. Con el desarrollo de este filtro se busca linealizar el modelo de estado no lineal en cada instante de tiempo alrededor del último estado conocido, una vez hallado el modelo lineal se procede a aplicar el Filtro de Kalman [9].

2.5. OPeNDAP

OPeNDAP, un acrónimo de "Open-source Project for a Network Data Access Protocol", es una arquitectura de transporte de datos y un protocolo ampliamente utilizado en las ciencias de la tierra. El protocolo está basado en HTTP y la especificación actual es OPeNDAP 2.0. OPeNDAP incluye normas para encapsular los datos estructurados, dotando los datos con atributos y añadiendo semánticas que describan los datos. El protocolo es mantenido por OPeNDAP.org, una organización sin ánimo de lucro financiada con fondos públicos que también proporciona implementaciones de referencia de servidores y clientes OPeNDAP [10].

Un cliente de OPeNDAP podría ser un navegador web, aunque tendría una funcionalidad limitada. Por lo general, un cliente OPeNDAP es un programa de gráficos (GrADS, Ferret o ncBrowse) o una aplicación web (como DChart) vinculado a una biblioteca OPeNDAP.

Un cliente de OPeNDAP envía peticiones a un servidor OPeNDAP, y recibe varios tipos de documentos o datos binarios como una respuesta. Uno de esos documentos se llama un DDS (recibido cuando una solicitud DDS es enviada), que describe la estructura de un conjunto de datos (*dataset*). Un conjunto de datos, visto desde el lado del servidor, puede ser un archivo, una colección de archivos o una base de datos. Otro tipo de documento que pueda ser recibida es un DAS, que da valores a los campos descritos en el DDS. Los datos binarios se reciben cuando el cliente envía una solicitud DODS.

Un servidor OPeNDAP puede servir a una colección de datos de cualquier tamaño. Los datos en el servidor están a menudo en formato HDF o NetCDF 2.1, pero puede estar en cualquier formato incluyendo formatos definidos por el usuario. En comparación con los protocolos de transferencia de archivos ordinarios (por ejemplo, FTP), una importante ventaja usando OPeNDAP es la capacidad de recuperar subconjuntos de archivos, y también la capacidad de agregar datos de varios archivos en una operación de transferencia.

OPeNDAP es ampliamente utilizado por agencias gubernamentales como la NASA y la NOAA para proveer datos de satélites, meteorológicos y otros datos observados por las ciencias de la tierra.

2.6. PyPI

El Python Package Index o PyPI es el repositorio de software oficial para aplicaciones de terceros en el lenguaje de programación Python. Los desarrolladores de Python pretenden

que sea un catálogo exhaustivo de todos los paquetes de Python escritos en código abierto [11].

2.7. Proceso ETL

Un proceso ETL (Extract, Transform and Load) es el encargado de reunir información de diferentes sistemas de origen, que tendrán dicha información organizada de manera muy distinta y con diferentes formatos, formatearla para satisfacer los requisitos de un formato o estructura de datos propio eligiendo las partes útiles y cargarla en dicho formato deseado dentro del sistema de destino.

3. Herramientas empleadas

3.1. Airflow

Es un framework creado para ayudar agilizar procesos ETL 2.7 en Python. Proporciona maneras con la que interaccionar con los sistemas de gestión de bases de datos utilizadas con mayor frecuencia entre los que se encuentra MySQL. En esta sección describiremos los pasos necesarios para poder usarlo.

3.1.1. Instalación

- Como íbamos a usar el framework Airflow usé la versión 2.7 de Python para todo el proyecto, que es la mas recomendada al ser la que los desarrolladores de la herramienta usaron para probarla.
- Para instalarlo seguimos los pasos descritos en la página de documentación de Airflow [12].
  - Instalar desde PyPI 2.6 usando pip ⇒ pip install airflow
  - Inicializar la base de datos ⇒ airflow initdb
- Para iniciar el servidor web, desde una terminal nos posicionamos en el directorio en el que se instaló Airflow y ejecutamos el comando "airflow webserver". Para poder verlo accedimos en un navegador web a la dirección localhost:8080, siendo 8080 el puerto por defecto al que se conecta Airflow.

Como se anuncia, el proceso de instalación es muy simple y con eso ya pudimos empezar a configurar algunas tareas para ejecutar y a hacer pruebas con ellas para irnos familiarizando con el entorno. Pero para terminar de configurar Airflow, tuvimos que establecer una conexión hacia la base de datos que usaríamos.

Establecer BBDD

El paquete básico PyPI de Airflow solo instala lo necesario para ponerse en marcha. Como queríamos poder emplear Airflow a fondo, tuvimos que establecer una base de datos real y cambiar el "ejecutor" de "SequentialExecutor" a "LocalExecutor" en el archivo de configuración de Airflow. Nosotros

Ya que Airflow fué construido para interactuar con sus metadatos usando la librería de SqlAlchemy, es posible usar cualquier base de datos compatible con SqlAlchemy entre las que se encuentra MySQL, que es por la que nos decantamos por ser la que mejor conocíamos.

Una vez configurada la base de datos para alojar a Airlfow, tuvimos que alterar la entrada sobre la conexion de SqlAlchemy del archivo de configuración. A continuación, cambiamos la configuración del "ejecutor" a utilizar a "LocalExecutor", un ejecutor que puede paralelizar las instancias de *tasks* a nivel local, y volvimos a inicializar la base de datos desde la terminal → airflow initdb.

3.2. Python

Python es uno de los lenguajes de programación más utilizados en la ciencia de datos, en gran medida por tener una sintaxis de uso general y con una curva de aprendizaje muy corta. Eso ha hecho que se haya desarrollado una gran cantidad de librerías para extender

su funcionalidad, incluyendo librerías científicas en las que nos hemos apoyado que permiten realizar numerosas tareas de tratamiento de datos, visualización, cálculo numérico y simbólico y otras aplicaciones específicas.

3.3. MySQL

Para almacenar todos los datos con los que ibamos a operar, necesitaríamos hacer uso de una base de datos relacional. Decidimos usar MySQL como nuestro gestor de base de datos por estar ya familiarizados con ello.

3.3.1. SQuirrel SQL Client

Es un programa gráfico en Java que permite observar la estructura de una base de datos compatible con JDBC, examinar los datos en tablas, ejecutar comandos SQL, etc [13]. Nosotros usamos esta herramienta para visualizar y acceder a los datos de una manera más cómoda que a través de la línea de comandos, ya que además nos permitía guardar en sesiones los comandos que íbamos usando.

3.4. xarray

Es un proyecto de código abierto que tiene como objetivo proporcionar una herramienta para trabajar con matrices multidimensionales indexadas en python.

3.4.1. Dataset

Es una representación en memoria de un archivo netCDF, un archivo auto-descriptivo de datos científicos de uso generalizado en las ciencias de la Tierra.

3.5. web2py

A la hora de hacer la aplicación móvil, íbamos a necesitar alguna forma de proporcionarle los datos. Con esto en mente web2py se seleccionó para ser usado a modo de servidor web.

Web2py es un framework de desarrollo web de código abierto escrito y programado en Python. Permite desarrollar páginas web dinámicas empleando Python y su objetivo principal es dar soporte al desarrollo ágil de aplicaciones web escalables, seguras y portables enfocadas en el uso de bases de datos.

Web2py fue originalmente diseñado como una herramienta de enseñanza con énfasis en la facilidad de uso y despliegue, así que no tiene ningún archivo de configuración a nivel de proyecto [14] y se hace cargo de los problemas principales de seguridad, haciendo que hayan menos posibilidades de introducir ningún tipo de vulnerabilidad.

3.6. Ionic

Ionic es un potente SDK de código abierto construido sobre AngularJS y Apache Cordova que ofrece herramientas y servicios para el desarrollo de aplicaciones móviles híbridas utilizando tecnologías web como CSS, HTML5, y Javascript. Las aplicaciones pueden ser construidas con estas tecnologías web y luego ser distribuidas a través de las tiendas de aplicaciones nativas para ser instaladas en los dispositivos móviles [15].

Tomamos la decisión de usar Ionic porque se centra principalmente en la apariencia, y la funcionalidad de la interfaz de usuario de la aplicación, facilitando y simplificando toda la parte del *front-end* y permitiéndonos centrarnos en otros aspectos.

3.6.1. AngularJS

AngularJS es un framework de JavaScript que implementa el patrón MVC (Modelo, Vista, Controlador) para el desarrollo web *front-end* permitiendo crear aplicaciones SPA (Single-Page Applications), lo que ayuda a darles la sensación de ser aplicaciones nativas.

AngularJS permite extender el vocabulario HTML con directivas y atributos, manteniendo la semántica y sin necesidad de emplear librerías externas como jQuery o Underscore.js para que funcione. Sigue el patrón MVC de ingeniería de software en un esfuerzo para hacer que el desarrollo y las pruebas sean más fáciles [16].

Con el uso de la inyección de dependencias, Angular consigue llevar servicios tradicionales del lado del servidor, tales como controladores dependientes de la vista, a las aplicaciones web del lado del cliente. En consecuencia, gran parte de la carga en el *back-end* se reduce, lo que conlleva a aplicaciones web mucho más ligeras [17].

3.6.2. Apache Cordova

Es un popular framework de desarrollo de aplicaciones móviles que permite construir aplicaciones para dispositivos móviles utilizando CSS3, HTML5 y JavaScript en lugar de depender de las APIs específicas de la plataforma como los de Android, iOS o Windows Phone [18].

4. Arquitectura y Metodología

Para la implementación de este proyecto adoptamos un modelo de desarrollo iterativo e incremental. El desarrollo incremental es una estrategia programada y en etapas, que empleamos para desarrollar las diferentes partes del sistema en diferentes momentos o a diferentes velocidades. Luego las fuimos integrando a medida que las fuimos completando en un modo similiar a como se ve en la figura 1.

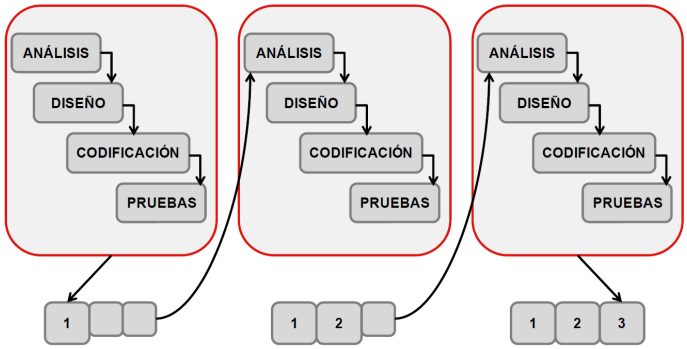


Figura 1: Modelo de ciclo de vida incremental [19]

4.1. Diagrama de despliegue

Hemos añadido esta sección para facilitar la visión general del sistema y los diferentes nodos que lo componen.

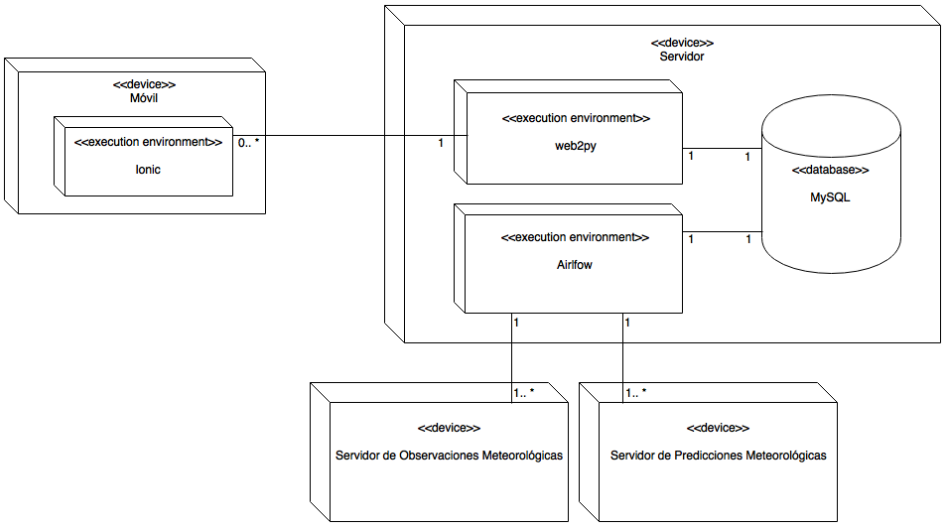


Figura 2: Diagrama de despliegue de la implementación

4.2. Esquema de la Base de Datos

En este apartado vamos a describir la función de cada tabla y daremos como ejemplo alguno de los valores que usamos para nuestro caso de uso.

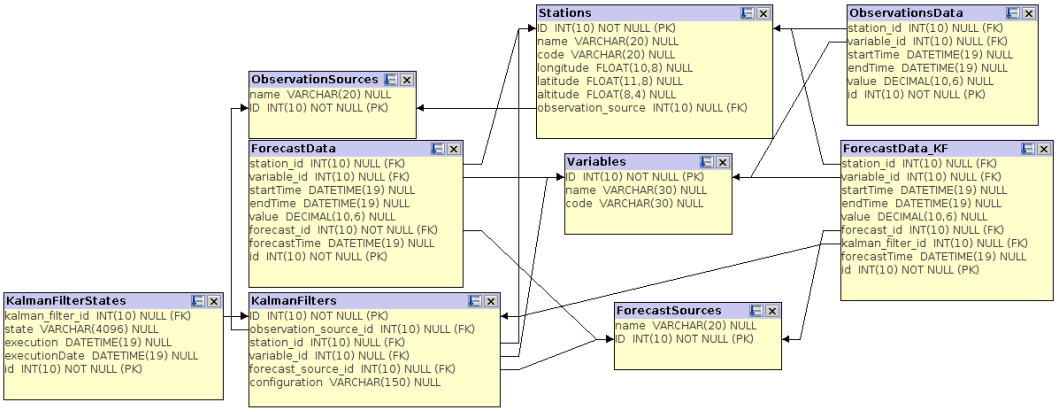


Figura 3: Diagrama de bases de datos UML

Todas las tablas tienen como clave primaria un tipo de dato entero llamado “ID” que se incrementa de forma automática al añadir nuevas entradas.

La tabla **ObservationSources** tiene un campo “name”, la usamos para indicar el nombre del origen de las observaciones, por ejemplo puede tener como valor “AEMET” o “DGT” para indicar que se van a sacar o de la Agencia Estatal de Meteorología o de las estaciones meteorológicas de la Dirección General de Tráfico.

La tabla **Stations** expone los datos de las diferentes estaciones meteorológicas de las que se hace uso. “Name” correspondería al nombre de la zona donde se encuentra preferiblemente con su dirección, “code” es el código con el que cada fuente de datos identifica cada estación, por ejemplo con el código “1111X” accedemos a los datos de la estación que tiene AEMET en Santander, “longitude” y “latitude” son sus coordenadas, “altitude” define a que altura se encuentra respecto al nivel del mar, pero no siempre es aplicable ya que no todas las fuentes de datos proporcionan esta información y “observation_source” es una clave foránea que hace referencia a la “ID” de la tabla ObservationSources y tiene el fin de indicar a que agencia le pertenece cada estación.

La tabla **Variables** define las diferentes magnitudes atmosféricas que vamos a ir guardando. En “name” se meten nombres idénticos para todas la variables del mismo tipo, sea la temperatura, el viento, etc., y “code” es el código que cada fuente de datos usa para identificar a dicha variable, por ejemplo dependiende de la fuente para sacar la temperatura puede que tengamos que usar “temperature” para una y “Temperatura °C” para otro.

La tabla **ObservationsData** es donde almacenamos los distintos valores de las observaciones de todas las variables. El campo “station_id” indica de qué estación es el dato en concreto, es una clave foránea que apunta al “ID” de la tabla Stations. El campo “variable_id” es una clave foránea que apunta a la “ID” de la tabla Variables para informar de que variable se trata. Los campos “startTime” y “endTime” definen el periodo temporal en el que se ha tomado el dato, por ejemplo para la temperatura tomada de una fuente que proporciona datos cada hora, si la temperatura se toma una vez a la hora ambos marcarán la misma fecha, mientras que si el dato se calcula como la media de la temperatura a lo largo de toda la hora, la fecha de inicio “startTime” será una hora menor que la de fin “endTime”. El valor de la variable se declara en “value”. Para evitar repeticiones añadimos una restricción de clave única sobre los campos “station_id”, “variable_id”, “startTime” y “endTime”.

La tabla **ForecastSources** establece cuales son las fuentes de las predicciones que tenemos almacenadas. El campo "name" es el nombre de la fuente, sea un modelo numérico de predicción meteorológica o la página web que proporcione los datos, por ejemplo "GFS", "WRF" o "OpenWeatherMap".

La tabla **ForecastData** es donde guardamos los valores de las predicciones obtenidas. Es bastante parecida a la tabla ObservationData; "station_id", "variable_id" y "forecast_id" son claves foráneas dirigidas a la "ID" de la tabla Stations, la de la tabla Variables y a la de ForecastSources respectivamente, e indican cual es la localización para la que extraímos los datos, que variable que fue recogida y de que fuente extrajímos los datos. En "startTime" y "endTime" se guardan las fechas entre las que se realizó el pronóstico y en "value" está el valor de la predicción de la variable indicada. Además se define un "forecastTime" que señala la hora a la que se realizó la predicción. También le pusimos una restricción de unicidad a las columnas "station_id", "variable_id", "startTime", "endTime", "forecast_id" y "forecastTime".

En la tabla **KalmanFilters** se encuentran las diferentes configuraciones de los filtros de Kalman y también se indica con que variable se corresponde cada una, con esto queremos decir que se define una cofiguración por variable. Explicamos qué es la configuración y para qué sirve con mas detalle en la sección del proceso de implementación del filtro 6.0.6. Las claves foráneas son el "observation_source_id", que indica el origen de las observaciones empleadas para una determinada ejecución del filtro de Kalman al hacer referencia al "ID" de la tabla de ObservationSources, el "station_id" que nos dice de donde vamos a sacar los datos al referenciar al "ID" de la tabla Stations, el "variable_id" que indica para que variable vamos a ejecutar el filtro al referenciar al "ID" de la tabla Variables y el "forecast_source_id" que apunta al "ID" de la tabla ForecastSources y nos dice de donde vamos a sacar las predicciones. En "configuration" guardamos cada una de las diferentes configuraciones que hemos definido para cada variable en concreto. Lo que se almacena es una cadena de caracteres que mantiene un formato JSON que luego somos capaces de leer desde Python para deserializarlo y después pasárselo al filtro de Kalman. También le añadimos una restricción de clave única en los campos "observation_source_id", "station_id", "variable_id", "forecast_source_id" y "configuration".

En la tabla **KalmanFilterStates** es donde guardamos el estado del filtro de Kalman después de cada ejecución. El campo "kalman_filter_id" es una clave foránea que hace referencia a la "ID" de KalmanFilters, así indicamos para que configuración ha salido dicho estado. En "state" guardamos los estados de las ejecuciones, es una cadena de caracteres en formato JSON. En "execution" indicamos a través de una fecha el horizonte de predicción para el cual se ejecutó el filtro cuando salió como resultado este estado y con "executionDate" indicamos de qué fecha son las predicciones que estamos pasando por el filtro de Kalman. Además le añadimos una restricción de unicidad en "kalman_filter_id", "execution" y "executionDate".

La tabla **ForecastData_KF** es donde guardamos todos las predicciones corregidas del filtro de Kalman. Las claves foráneas que la componen son el "station_id", donde esta la referencia al "ID" de la tabla Stations, que nos dice de donde se sacaron las observaciones de los días anteriores con los que se corrigió el valor de la predicción actual, el "variable_id" que hace referencia al "ID" de la tabla Variables y nos indica para que variable es el valor corregido obtenido del filtro, el "forecast_id" que hace referencia al "ID" de la tabla ForecastSources que nos indica de donde se han sacado las predicciones que se han usado para realizar la corrección y el "kalman_filter_id" que hace referencia al "ID" de la tabla KalmanFilters e indica que configuración del filtro se ha usado para obtener el valor. En "startTime" y "endTime", como en las ocasiones anteriores en las que han aparecido, se

define el periodo temporal para el que se ha realizado la predicción, “value” es el valor de la predicción corregida y “forecastTime” indica la fecha en la que se realizó la predicción. También le pusimos una restricción de clave única en los campos “station_id”, “variable_id”, “startTime”, “endTime”, “forecast_id”, “kalman_filter_id” y “forecastTime”.

5. Requisitos

Realizamos el proyecto en coordinación con personal de la empresa Predictia, por lo que fueron ellos los que nos fijaron por etapas los requisitos que deberíamos satisfacer.

- El sistema tendrá que ser compuesto de manera modular para que séa escalable y que sea faćil hacerle alguna modificación.
- El sistema deberá tener implementado un sistema de logs.
- El sistema deberá emplear una de bases de datos donde almacenar todos los datos necesarios.
- Deberá de ser capaz de recoger observaciones meteorológicas de cualquier fuente.
- Deberá de ser capaz de recoger predicciones meteorológicas de cualquier fuente.
- Todos los datos guardados deberán tener sus fechas en formato UTC.
- No pueden haber datos duplicados en el sistema.
- Tendrá que ser capaz de alimentar los datos al filtro de Kalman en cuanto estos estén disponibles.
- Para la aplicación móvil:
  - Se deberán poder ver las predicciones de ciudad dada de manera sencilla.
  - Deberá haber un buscador que te permita elegir de que ciudad se quieren ver las predicciones.
  - Se deberá asegurar que el valor introducido en la búsqueda es uno válido.
  - Se deberá poder tener una lista de las ciudades consultadas.
  - Se deberá poder borrar entradas de dicha lista.
  - Deberá ser intuitiva y fácil de manejar.
  - Se deberá poder configurar que magnitudes meteorológicas mostrar en todo momento.
  - Se deberá tener un sistema de caches para acelerar la carga de datos ciudad ya consultada.
  - Se deberá alertar al usuario cuando haya habido algún problema que no permita mostrar los datos.

6. Proceso de desarrollo

El objetivo del proyecto consiste en desarrollar un sistema que extraiga observaciones y predicciones meteorológicas de diversas fuentes, aplique una metodología estadística de calibración y almacene los resultados. Seguirá, por tanto, un proceso conocido como ETL (extract-transform-load) 2.7. Los pasos a seguir para su desarrollo fueron definidos por el cliente que en este caso es la empresa Predictia.

El desarrollo de sistemas ETL suelen presentar requisitos comunes como la notificación de errores, la secuenciación de tareas, el poder realizar reintentos de las tareas fallidas, la planificación, etc. Por ello, existen varios *frameworks* que facilitan enormemente el desarrollo de este tipo de sistemas. Uno de estos *frameworks* es Airflow, desarrollado por la empresa Airbnb y publicado bajo licencia Apache. A continuación describiremos las características de este *framework* y su configuración para nuestro caso de uso, para ver como se instaló diríjanse a la sección 3.1.

Como caso de uso y a modo ilustrativo, mostraremos cómo se ha configurado el sistema para obtener y calibrar predicciones con el modelo Global Forecast System (GFS) en España. No obstante, se podría aplicar la misma metodología en cualquier otra parte del mundo siempre que se disponga de las observaciones y predicciones adecuadas.

Inicialmente, se requirió desarrollar un proceso de carga de observaciones meteorológicas en una base de datos, seguido por las predicciones. Mas adelante, deberíamos re-implementar una implementación ya existente del filtro de Kalman en otro lenguaje de programación con fines de investigación. Finalmente, conseguir que todo se ejecutase de forma autónoma y almacenase los resultados en una base de datos.

6.0.1. Airflow

Airflow es una plataforma que facilita el desarrollo de procesos ETL y nos permite mediante la programación, planificar y controlar los flujos de trabajo. Podríamos haber usado un servicio *cron* para realizar esta labor sin embargo este tipo de herramientas más avanzadas nos dotan de una gran cantidad de funcionalidades muy útiles para este tipo de desarrollos. Por ejemplo, Airflow proporciona una interfaz web con la que explorar el conjunto de tareas programadas con la que hacer un seguimiento de las que se han ejecutado con éxito y las fallidas así como una interfaz de línea de comandos que nos permite, entre otras cosas, probar, ejecutar, describir y aclarar partes de estos *DAGs*.

En Airflow, un *DAG* (Directed Acyclic Graph o un grafo dirigido acíclico) es la colección de todas las tareas que se desee ejecutar, organizadas de una manera que refleje las relaciones y dependencias que hay entre ellas, lo que facilita dictar el orden en que se desea que se ejecuten. El encargado de hacerlo es el planificador que ejecuta las *tasks* en una matriz de trabajadores (*workers*) mientras sigue las dependencias especificadas.

Mientras que los *DAG* describen cómo ejecutar un flujo de trabajo, los *operators* son los que determinan lo que realmente se hace.

Un *operator* describe una única *task* en un flujo de trabajo. Los *operators* son por lo general (pero no siempre) atómicos, lo que significa que no necesitan compartir recursos con otros *operators*. El *DAG* se asegura de que los *operators* se ejecuten en el orden correcto; aparte de esas dependencias (definidas por el usuario) los *operators* generalmente se ejecutan de forma independiente.

Airflow proporciona *operators* para muchas tareas comunes, pero el único que usamos en nuestro caso es el PythonOperator, que se encarga de llamar a funciones en Python [20].

Al ser instanciado, un *operator* pasa a ser referido como un *task*. La instanciación define valores específicos al llamar al *operator* abstracto, y la *task* parametrizada se convierte en un nodo de un *DAG* [21].

La secuencia en la que se ejecutan las *tasks* se configura por código 1, usando el lenguaje de programación Python, haciendo que sea más fácil de mantener, de hacer un control de versiones y de comprobar.

Listing 1: Ejemplo de un DAG

```
from airflow import DAG
from airflow.operators import BashOperator
from airflow.operators import PythonOperator
from datetime import datetime, timedelta
from un_modulo import hacer_algo

startDate=datetime.strptime("2016-07-20 10:00:00","%Y-%m-%d %H:%M:%S")
default_args = {
    "owner": "airflow",
    "depends_on_past": False,
    "start_date": startDate,
    "email": ["airflow@airflow.com"],
    "email_on_failure": False,
    "email_on_retry": False,
    "retries": 1,
    "retry_delay": timedelta(minutes=5)
}

dag = DAG("ejemplo", default_args=default_args,
     schedule_interval=timedelta(days=1)) #que se ejecute cada 24 horas

# A, B y C son ejemplos de tasks creados al instanciar unos operators
A = BashOperator(
    task_id="print_date",
    bash_command="date",
    dag=dag)

B = BashOperator(
    task_id="sleep",
    bash_command="sleep 5",
    retries=3,
    dag=dag)

args_adicionales=["Los argumentos que", "se le pasan a la funcion"]
C = PythonOperator(
    task_id="python_function",
    python_callable=hacer_algo,
    op_args=args_adicionales,
    dag=dag)

#esto indica que B tiene que esperar a que termine de ejecutarse A, y C tiene que
     esperar a B
B.set_upstream(A)
```

Por ejemplo, un *DAG* simple podría consistir en tres *tasks*: A, B, y C. En este caso, A se tiene que ejecutar con éxito antes de que B pueda ejecutarse, pero que C pueda hacerlo en cualquier momento. Además, la *task* A tiene 5 minutos para ejecutarse, y B que puede reiniciarse hasta 3 veces en el caso de que falle. Finalmente, el flujo de trabajo se ejecutará cada noche a las 10 pm, pero no debe comenzar hasta la fecha determinada en "start_date".

De esta manera, un *DAG* describe cómo desea llevar a cabo su flujo de trabajo pero no define nada acerca de lo que realmente queremos hacer (A, B, y C pueden ser cualquier cosa), de eso se encargan los *operators*.

Lo importante es que el *DAG* no se preocupa por lo que sus *tasks* constituyentes hacen, su trabajo es asegurarse de que independientemente de lo que hagan, que suceda en el momento adecuado, o en el orden correcto, o que maneje sucesos y problemas inesperados de manera adecuada.

Los *DAG* se definen en ficheros estándar de Python que se colocan en el DAG_FOLDER de Airflow (Airflow define una carpeta donde guardar los *DAGs*). Airflow ejecutará el código de cada archivo para generar dinámicamente los objetos *DAG*. Se pueden tener tantos *DAGs* como se desee, cada uno con un número arbitrario de *tasks*. En general, cada uno corresponderá a un único flujo de trabajo [22].

Además proporciona maneras con la que interaccionar con los sistemas de gestión de bases de datos utilizadas con mayor frecuencia. Una de esas maneras es mediante *hooks*, que son interfaces hacia plataformas externas y bases de datos entre las que se encuentra MySQL.

6.0.2. Conectando Airflow con nuestro entorno

Airflow proporciona además algunas utilidades como definir variables de entorno y gestionar conexiones a bases de datos.

Para poder establecer una conexión a una base de datos, Airflow necesita saber cómo conectarse a la misma. La información tal como nombre del host, puerto, nombre de usuario y contraseñas se maneja en la sección de conexión, Admin → Connection, de la interfaz de usuario como se muestra en la figura 4. En nuestro código se hace referencia a la 'id_con' de los objetos de conexión, como se muestra en este fragmento de código 2.

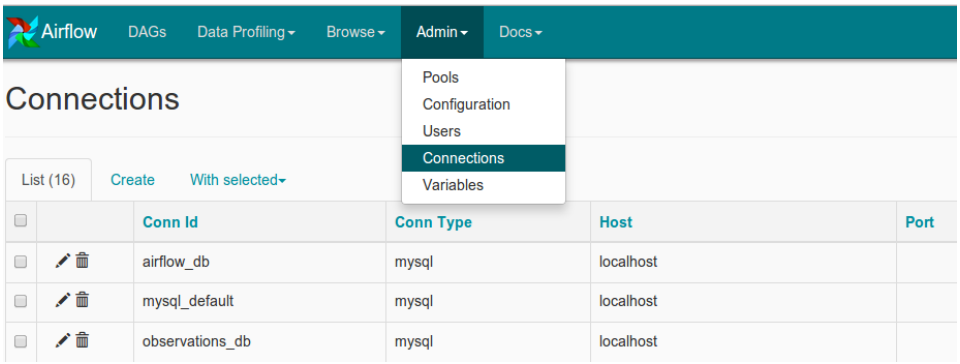


Figura 4: Interfaz web de Airflow

Listing 2: sharedMethods.py

```
from airflow.hooks import BaseHook

def connectDb():
   return BaseHook.get_connection("observations_db").get_hook().get_conn()
```

Ciertas funciones comunes como el acceso a la base de datos, la configuración del registrador de logs, etc., fueron implementadas en un módulo de utilidades al que podían acceder las diferentes *tasks* desarrolladas.

Con el fin de hacer mas sencilla la depuración añadimos un logger 3 para cada uno de los módulos y creamos una variable de entorno en Airflow de donde sacar los parámetros de configuración.

Listing 3: sharedMethods.py

```
import logging
import configparser
from airflow.models import Variable

def logSetup():
    #sacar datos del archivo de configuracion de airflow
    config = configparser.RawConfigParser()
    config.read(Variable.get('bd'))
    logFile=config.get('core','info_log')

    #preparar el archivo de log
    logger = logging.getLogger("")
    logger.setLevel(logging.DEBUG)
    handler = logging.FileHandler(logFile,mode='a')
    formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s -
        %(message)s")
    handler.setFormatter(formatter)
    logger.addHandler(handler)
    return logger
```

6.0.3. Carga de observaciones

El primer paso fue definir un esquema de datos en los que almacenar las observaciones, predicciones, predicciones corregidas y metadatos de configuración. Nos decidimos por el esquema descrito anteriormente en la sección 4.2. Pero no lo concebimos así desde el principio. Para ir comprobando que funcionaría empezamos con un esquema mas reducido, el indicado por la figura 5, y lo fuimos ampliando a medida que avanzamos con el proyecto.

Posteriormente, decidimos implementar algunos procesos de carga de datos de observaciones. Identificamos la página de AEMET donde hay disponibles datos de observaciones a escala horaria de una gran variedad de localizaciones en España. En nuestro caso, AEMET proporciona los datos en formato CSV (comma-separated-value), que es un tipo de documento de texto para representar datos en forma de tabla, donde las columnas se separan por comas y las filas por saltos de línea [23].

En los ficheros CSV proporcionados aparecen las observaciones de las últimas 24 horas de la ciudad que hayas indicado y para una gran variedad de variables meteorológicas. En

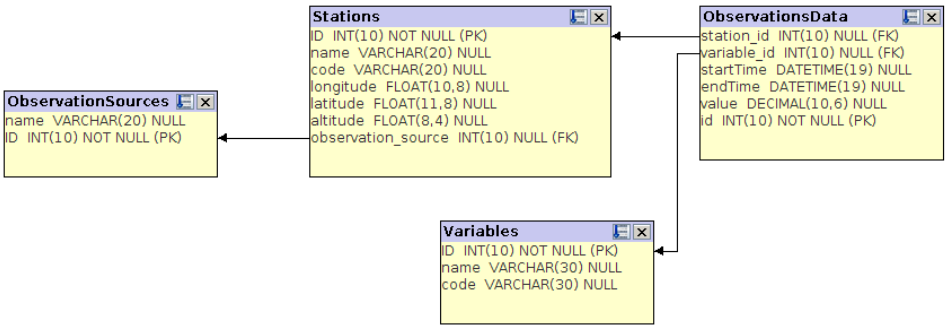


Figura 5: Diagrama de bases de datos UML

nuestro caso, filtramos la información recogiendo sólo los datos de la variable que haya sido indicada.

El objetivo es *parsear* los datos contenidos en el archivo CSV y almacenarlos en la base de datos. Esta operación la abordamos de forma específica dependiendo de la fuente de información escogida.

- Primero establecemos la conexión a la base de datos haciendo uso de los *hooks* que ofrece Airflow 4.

Listing 4: obsAemetLoader.py

```
#connect to db
db = connectDb()

#setup cursor object
cursor = db.cursor() #It will allow me to execute queries
```

- A la función de python le pasamos como argumentos el código de la ciudad de la que se desean extraer datos y la variable atmosférica de interés (temperatura, presión, humedad, etc.).
- Con esta información hemos de averiguar los metadatos necesarios de la base de datos para saber dónde almacenar los datos meteorológicos que se sacarán del archivo CSV 5.

Listing 5: obsAemetLoader.py

```
cursor.execute("SELECT ID FROM ObservationSources WHERE name='AEMET'")
obsSource=cursor.fetchone()[0]

if cursor.execute("SELECT ID FROM Stations WHERE code='%s' AND
    observation_source='%s'" % (ciudad,obsSource)):
   stationsID=cursor.fetchone()
else:
   closeConections(cursor, db)
   raise ValueError('No existe ninguna estacion con ese codigo en la base de datos')
if cursor.execute("SELECT ID FROM Variables WHERE code='%s'" % medida):
   variablesID=cursor.fetchone()
else:
```

```
    closeConections(cursor, db)
    raise ValueError('El codigo {} no existe en la tabla Variables de la base de
        datos'.format(medida))
```

- A continuación, hay que encontrar los valores de la variable definida y añadirlo a la base de datos.
  - Hay que tener en cuenta que como queríamos que todos los datos de la bases de datos fueran uniformes independientemente de donde se sacase la información, decidimos que todas las horas se guardarían en horario UTC (tiempo universal coordinado), y en caso de la temperatura, que se guardaría en grados Celsius.
  - En este caso, por como estaban repartidos los datos, tuvimos que ir fila por fila para poder acceder al dato deseado.
  - En nuestro caso de uso, solo contemplamos la opción de usar el filtro de Kalman con la temperatura, pero como se puede observar en el fragmento de código siguiente 6, es sencillo adaptar el código para contemplar la opción de hacerlo con otras variables.

Listing 6: obsAemetLoader.py

```
#para imprimir toda la columna
for reg in entrada:
    #ejemplo 06/03/2016 20:00
    st=time.strptime(reg['Fecha y hora oficial']+':00',"%d/%m/%Y %H:%M:%S")
    utcTimeBdEnd=time.strftime("%Y-%m-%d %H:%M:%S",time.gmtime(time.mktime(st)))

    if medida=='temperatura':
        utcTimeBdStart=utcTimeBdEnd
    elif medida=='precipitacion':
        #la precipitacion mostrada es el dato recogido desde la hora anterior, por lo
            que utcTimeBdStart debe ser una hora menos que utcTimeBdEnd
        st_writable=list(st) #lo convierto en una lista ya que el struct es readonly
        substract=st_writable[3]-1 #aemet recoge los datos cada hora
        st_writable[3]=substract%24
        if substract < 0:
            st_writable[2]-=1 #restamos uno al dia porque empezo el dia anterior (sino
                sale ValueError: hour out of range mas adelante)

        st=time.struct_time(tuple(st_writable)) #lo vuelvo a convertir en un struct
        utcTimeBdStart=time.strftime("%Y-%m-%d
            %H:%M:%S",time.gmtime(time.mktime(st))) #hago lo mismo que antes para
            pasarlo al formato utc

    valorStr=reg[codMedida(medida)[0]]
    logger.info('=======archivoCsvMysql=========\nFecha {} {}
        {}'.format(utcTimeBdStart, medida, valorStr))
    if type(valorStr)==str:
        valorStr=valorStr.strip()
    valor=tipo(valorStr)

    #para este execute necesito el tiempo en formato 2016-03-07 09:00:00
    addObsData(cursor,staID,varID,utcTimeBdStart,utcTimeBdEnd,valor)
```

Listing 7: sharedMethods.py

```
def addObsData(cursor,staID,varID,utcTimeStart,utcTimeEnd,valor):
   cursor.execute('REPLACE into ObservationsData
       (station_id,variable_id,startTime,endTime,value) VALUES
       ({0},{1},{2},{3},{4})'.format(staID,varID,"'"+utcTimeStart+"'",
       "'"+utcTimeEnd+"'",valor))
```

Listing 8: sharedMethods.py

```
def codMedida(x):
   return {
      'temperatura': ('Temperatura (\xbaC)',float),
      'velocidad del viento': ('Velocidad del viento (km/h)',int),
      'direccion del viento': ('Direcci\xf3n del viento',str),
      'racha': ('Racha (km/h)',int),
      'direccion de racha': ('Direcci\xf3n de racha',str),
      'precipitacion': ('Precipitaci\xf3n (mm)',float),
      'presion': ('Presi\xf3n (hPa)',float),
      'tendencia': ('Tendencia (hPa)',float),
      'humedad': ('Humedad (%)',int),
   }[x]
```

- En último lugar, no debemos olvidarnos de cerrar todas las conexiones abiertas y dejar así recursos libres para futuras ejecuciones 9.

Listing 9: obsAemetLoader.py

```
csvArchivo.close()
db.commit() # hacer un commit uso las operaciones INSERT, UPDATE o DELETE
closeConections(cursor, db)
```

Listing 10: sharedMethods.py

```
from airflow.hooks import BaseHook

def closeConections(cursor,db):
    cursor.close()
    db.close()
```

Al comenzar, nos descargamos los datos en una carpeta local e hicimos uso de ese fichero CSV para llevar a cabo el desarrollo de la función. Cuando vimos que las pruebas con datos estáticos funcionaban correctamente pasamos a modificar el código para que fueran descargados directamente de su URL original. Para ello hicimos uso de librerías HTTP (urllib2 en el caso de AEMET) para hacer una petición a la página de AEMET, descargar los datos y poder leerlos 11. No hizo falta hacer ninguna otra modificación ya que el formato del fichero descargado iba a ser idéntico al del fichero local con el que hicimos las pruebas.

Listing 11: obsAemetLoader.py

```
def loadData(ciudad, medida):
   csvFileUrl='http://www.aemet.es/es/eltiempo/observacion/
       ultimosdatos_%s_datos-horarios.csv?l=%s&datos=det&w=0&x=h24'%(ciudad,ciudad)
   csvArchivo=urllib2.urlopen(csvFileUrl)
```

Para demostrar la facilidad con la que puede añadir una fuente de observaciones y a modo de prueba creamos otro módulo 12 con el que hicimos la carga de observaciones de las estaciones meteorológicas que tiene la DGT instaladas en las carreteras.

Listing 12: obsDgtLoader.py

```
     # -*- coding: utf-8 -*-
import requests, time
from sharedMethods import logSetup,connectDb,closeConections,addObsData

def loadData(sensor, medida):
    jsonFileUrl='http://infocar.dgt.es/etraffic/BuscarElementos?accion=getDetalles&codEle={}&tipo=SensorMe
    #voy a hacerlo usando requests
    r =requests.get(jsonFileUrl)
    resp=r.json() #r.json() retorna un diccionario con todos los datos

    #preparar el archivo de log
    logger=logSetup()

    if 'noDatos' not in resp.keys(): #cuando no hay datos actualizados devuelve un
        diccionario con un key 'noDatos'
        #sacamos los datos de conexion a la base de datos del archivo de
            configuracion de airflow
        db = connectDb()
        #setup cursor
        cursor = db.cursor()

        #el resto es practicamente identico a lo hecho con los datos de AEMET
```

Una vez que ya estaba todo listo y nos habíamos asegurado que funcionaba sin problemas, procedimos a hacer lo mismo con las predicciones.

6.0.4. Carga de predicciones

El siguiente paso en el desarrollo consistió en la carga de predicciones de modelos numéricos disponibles en diferentes orígenes de datos, por lo que fuimos ampliando la base de datos para acoger los nuevos datos que se le irían añadiendo como se muestra en la figura 6.

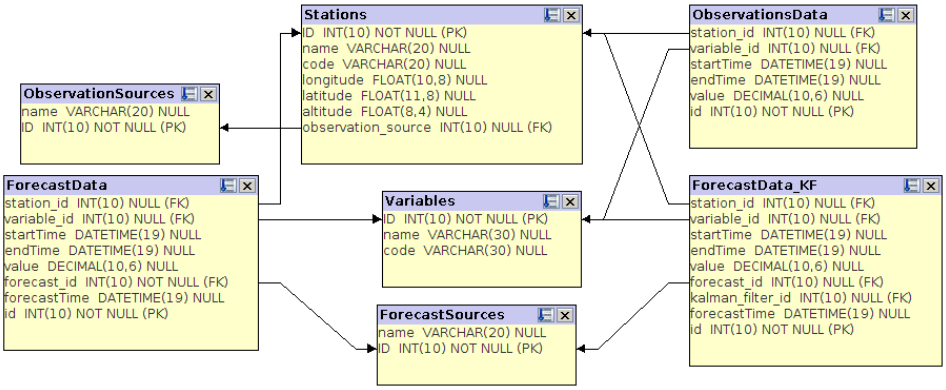


Figura 6: Diagrama de bases de datos UML

Implementamos un proceso de carga para obtener las predicciones del modelo GFS de el NOAA Operational Model Archive and Distribution System (NOMADS) 2.2, con cobertura global y ofrecido de manera gratuita.

En el caso de las observaciones, bastaba con pasar como argumentos el código de la ciudad y una variable ya que desde AEMET solo están accesibles los datos de las últimas 24 horas en un único fichero, que renuevan cada hora, haciendo que sea imposible recoger los datos de, por ejemplo, hace tres días. En cambio, desde la página de NOMADS, mantienen los datos de las últimas dos semanas indexados por la fecha en la que se hicieron las predicciones y cargan datos nuevos de su modelo climático cada 6 horas. Por lo tanto, para saber cuáles son los datos que precisamos, es necesario indicar la fecha de cuándo se realizaron las predicciones. Además, es necesario indicar el código de la ciudad de la que queremos saber la predicción meteorológica y la variable que nos interesa. A continuación explicaremos por qué es necesaria toda esta información.

- Hay que indicar la fuente de donde sacaremos las predicciones porque en la base de datos podemos guardar predicciones de más de una fuente.
- Con la fecha, ya es suficiente para acceder a los datos del GFS (siempre que existan datos para la fecha definida). Como NOMADS emplea el protocolo OPeNDAP2.5 que facilita los datos en formato netCDF 2.1 el acceder a los datos se logra cómodamente con las funcionalidades aportadas por la librería xarray 3.4.
- El GFS es un modelo con cobertura global. Como divide la superficie terrestre en una malla, es imprescindible saber las coordenadas de los puntos de la malla para acceder a la predicción de un determinado lugar. Xarray nos permite definir una longitud y latitud cualquiera y se encarga de encontrar las coordenadas mas cercanas en la malla. Por esto es indispensable tener el código de la fuente de donde sacamos las observaciones ya que nos permite averiguar las coordenadas de la estación meteorológica de la que se sacaron las observaciones para la ciudad en la que estamos interesados.
- Indicamos la ciudad en la que estamos interesados al pasar como argumento su código.

Lo siguiente es recorrer el *dataset* 3.4.1 en el que han sido guardados los datos y seleccionar los deseados, en nuestro caso las temperaturas, para cada una de las horas que vienen dadas por el GFS. Estos datos son entonces guardados en la tabla ForecastData de la base de datos, indicando de qué ciudad procede el dato, qué variable es, la fecha cuando se recogió el dato, el valor del dato, el código de la fuente de donde proviene la predicción y la fecha en la que se hizo la predicción. En este caso no es necesario transformar las fechas ya que ya vienen dadas en UTC.

Para terminar, como con las observaciones, cerramos las conexión a la base de datos que se abrió al principio de la función.

6.0.5. Carga de predicciones desde el servidor de Predictia

Para ilustrar la flexibilidad del sistema, se desarrolló otro proceso de captura de datos de predicciones. En este caso, las generadas por la empresa Predictia con el modelo WRF (Weather Research and Forecasting) [24]. Estas predicciones estaban también almacenadas en ficheros NetCDF alojados en un servidor accesible desde la red interna de la empresa.

Los pasos generales a seguir eran los mismos que con las predicciones del GFS así que también empleamos la librería xarray 3.4.

Aunque el fichero sigue el mismo formato, una de las dificultades a las que nos enfrentamos es que los datos no están indexados de la misma manera. En este caso, la malla en la que están codificados los datos usaba la proyección Lambert Conformal 7, a diferencia de la malla regular del GFS.

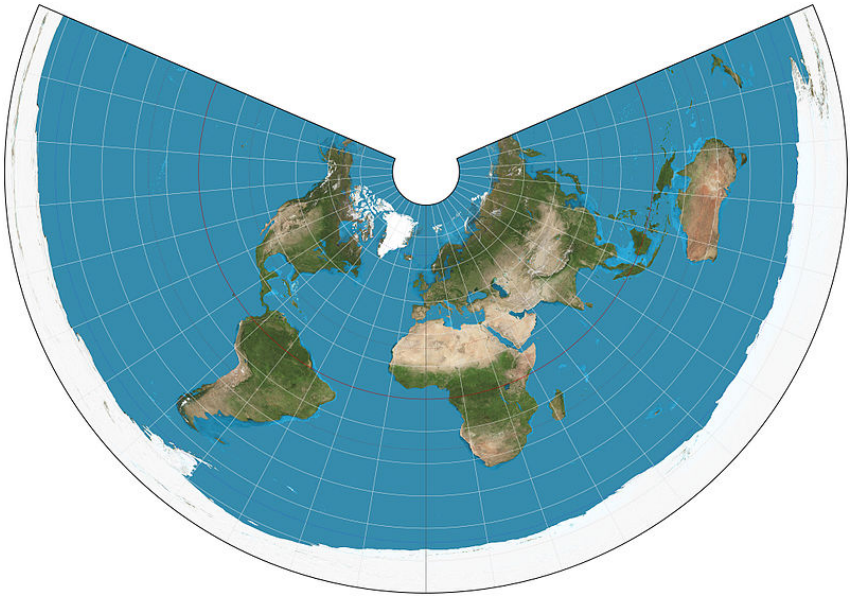


Figura 7: Proyección conforme cónica de Lambert, sacado de [25]

En este caso, al no ser una malla regular, el procedimiento para obtener los índices de los vectores de longitud y latitud desde unas coordenadas concretas es diferente, siendo necesario recorrer la lista de puntos para encontrar el más cercano a la localización buscada.

6.0.6. Implementación del Filtro de Kalman

El tercer paso era implementar una variante del filtro de Kalman 2.4 para la corrección de predicciones meteorológicas. Para ello, la empresa Predictia me proporcionó una implementación propia que llevaron a cabo en otro lenguaje de programación con fines de investigación. Esta implementación, validada con series de datos históricas, está basada en lo descrito por el artículo [3].

Para almacenar los resultados realizamos una última ampliación a la base de datos que quedó como se muestra en la figura 8. Las tablas que añadimos fueron fueron KalmanFilters, para guardar las configuraciones de los distintos filtros que podríamos querer definir y ejecutar, KalmanFilterStates, donde guardamos los estados necesarios para hacer los cálculos de filtros siguientes (para calcular la predicción corregida de un día a cierta hora es necesario tener el estado de la ejecución del día anterior a la misma hora como se muestra en la figura 9), y ForecastData_KF, que es donde almacenamos las predicciones corregidas.

La aplicación descrita en [3] propone definir diferentes filtros de Kalman, uno para cada horizonte de predicción. Es decir, si consideramos la predicción tres horaria del modelo GFS con un alcance de 72 horas, se definirán un total de 24 filtros de Kalman independientes. Para ello, se asume que el error cometido por el modelo en una predicción con un determinado alcance (por ejemplo, H+6) está relacionado los errores que cometió ese modelo en predicciones de días anteriores para ese mismo alcance.

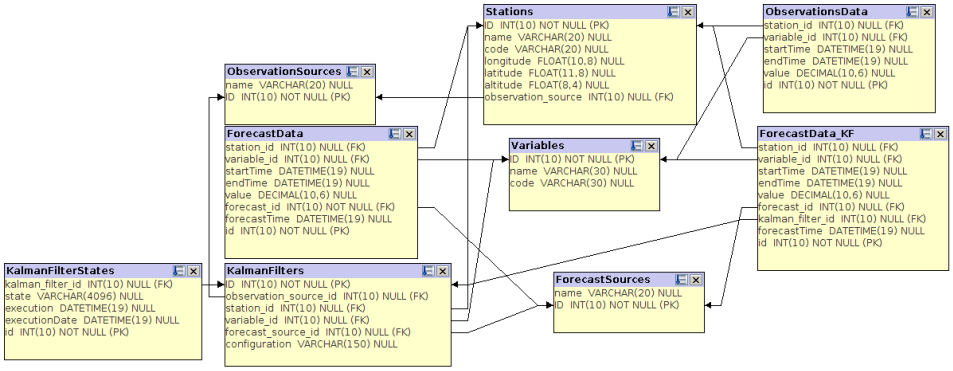


Figura 8: Diagrama de bases de datos UML

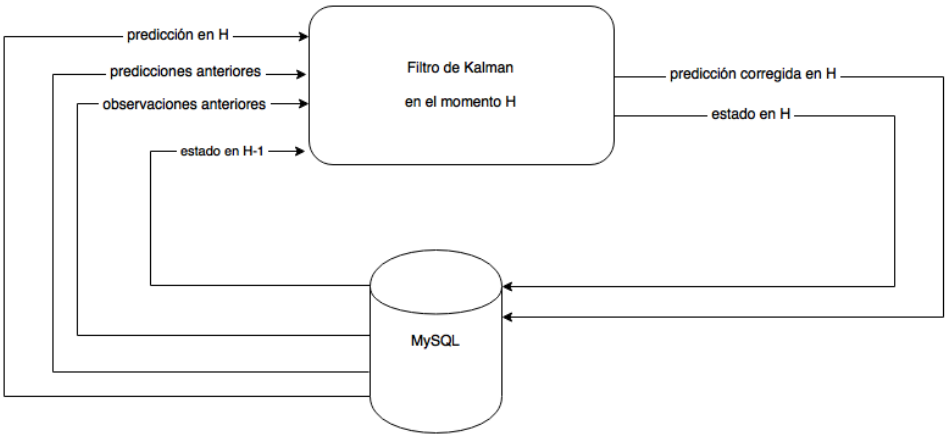


Figura 9: Diagrama mostrando las entradas y salidas del filtro de Kalman

- Primero nos proporcionaron una serie de observaciones y predicciones tomadas en el transcurso del año 2011 que usamos a modo prueba para comprobar si los resultados obtenidos por nuestra implementación del filtro coincidían con los que la empresa ya tenía.

- Además, era necesario implementar una manera de almacenar y consultar los estado del filtro en la base de datos. Por otra parte, era también necesario leer la configuración de los distintos filtros definidos (dada por una serie de parámetros). Todo esto se implementó en kalmanConfig.py.

- El filtro de Kalman suele ejecutarse a la resolución temporal del modelo numérico que se quiere corregir (en el caso de GFS tres horaria). Sin embargo, las observaciones pueden estar almacenadas en resoluciones temporales muy diferentes (horarias o diez minutales por ejemplo). Por tanto, era necesario desarrollar funciones de extracción de datos (observaciones y predicciones) que, además, permitiesen, mediante funciones de agregación (media para la temperatura y suma para la precipitación), homogeneizar las escalas temporales de los mismos. Estas funciones están implementadas en el módulo getKFData.py.

- Antes de continuar, para poder ejecutar el filtro hay que configurar los parámetros con los que lo ejecutaremos. Con esto especificamos la agregación temporal de los datos, el horizonte de predicción máximo (para cuántos días vamos a generar una predicción),

cuántos días anteriores se considerarán para la estimación del error y, por último, el grado del filtro (lineal o no lineal si el grado es mayor de uno). Puesto que ya existía una implementación del filtro que habían probado con series históricas, se disponía de una configuración adecuada de estos parámetros.

- Teniendo toda la información de entrada sólo es necesario aplicar propiamente el filtro de Kalman. Los módulos que se ocupan de ello son kalmanTest.py (que se encarga a a coger los datos que necesitará el filtro de Kalman, enviárselos y a guardar los resultados que este le devuelve) y kalmanFilter.py (que realiza los cálculos y devuelve la predicción corregida y el estado del filtro actualizado).

- Finalmente, una vez que verificamos nuestros resultados con los que tenía la empresa, pasamos a usar los datos de AEMET y del GFS en un esquema de ejecución operativa.

A continuación explicaremos un poco más a fondo cómo funciona esta implementación del filtro de Kalman.

- Para una determinada fecha en la que estemos ejecutando el filtro, necesitamos primero configurarlo y después ejecutarlo. Los datos que el filtro necesita saber son:

  - Las observaciones y predicciones para el mismo horizonte de predicción de los días anteriores, de tantos días atrás como se haya definido en su configuración (un dato por día).
  - La predicción del horizonte de predicción para cuando estemos ejecutando el filtro, sacada de entre las predicciones obtenidas a la hora de la ejecución del filtro.
  - El estado del filtro del día anterior para ese mismo horizonte de predicción (o unos valores por defecto en el caso de que no haya un estado anterior)
  - La configuración del filtro.

- Dependiendo de la fuente de las predicciones, puede que no proporcionen los datos de forma horaria. En nuestro caso por ejemplo, el GFS ofrece los datos con una resolución de 3 horas, y habrá otros lo que ofrecerán con una de 6 horas, etc. Esto lo tuvimos que tener en cuenta a la hora de configurar el filtro, ya que la resolución indicada no puede ser menor que la resolución del modelo. Esto también quiere decir que tuvimos que suministrarle las observaciones con una agregación de 3 horas. Pero como este sistema ha de funcionar con cualquier tipo de agregación, creamos el módulo *getKFData.py* para que se ocupara de conectarse a la base de datos, cogiese las observaciones y las predicciones y agrupase los datos en la resolución que le indicásemos y que lo hiciese teniendo en cuenta el tipo de variable (para promediarlos, sumarlos o coger el mínimo o el máximo, en el fragmento de código 13 hay un pequeño trozo del código mostrando como formulamos las consultas).

  - Con esto tendríamos todos los datos para la franja temporal que hubiésemos definido agrupados en función de una resolución dada, pero mas adelante de ahí sacamos los datos que correspondan con el horizonte de predicción para el que se esté ejecutando el filtro.

Listing 13: getKFData.py

```
'''
Pequenio ejemplo del codigo

hacemos una query a la base de datos indicando que la funcion de agrupacion
es haciendo la media de los datos
```

```
'''
elif aggFunction == 'MEAN':
        if aggWindow[1]=='H':
            query='select DATE_FORMAT(startTime,"{}'.format(form0)+'"),
                DATE_FORMAT(DATE_ADD(startTime,INTERVAL {}'.format(aggWindow[0])+'
                HOUR),"{}'.format(form0)+'"), AVG(value) from '+table+' where
                station_id={}'.format(staID)+' and
                variable_id={}'.format(variableID)+' and startTime >=
                "{}'.format(startDate)+'" and startTime < "{}'.format(endDate)+'"
                group by FLOOR(TIMESTAMPDIFF(HOUR,"{}'.format(startDate)+'",
                startTime)/{})'.format(aggWindow[0])
```

- Para sacar la predicción generada a la hora de la ejecución del filtro de Kalman también usamos este método teniendo en cuenta la misma resolución temporal.
- Cada vez que se ejecuta el filtro de Kalman, el resultado generado es una predicción corregida de un horizonte de predicción y el estado actualizado del filtro, y ambos se guardan en la base de datos en sus respectivas tablas. Este estado es el que se utiliza en las siguientes ejecuciones como otra de las entradas del filtro de Kalman. En cambio, si se está ejecutando por primera vez y no existe ningún estado anterior, se crea uno siguiendo unas reglas establecidas.

Para verificar de un modo mas visual que la predicción corregida se acercaba mas a las observaciones que la predicción original, implementamos un pequeño programa que dibuja una gráfica mostrando todos los datos 10 (por si no se aprecia la leyenda del gráfico, azul es para las predicciones corregidas, amarillo para las predicciones viejas o sin corregir y rojo para las observaciones).

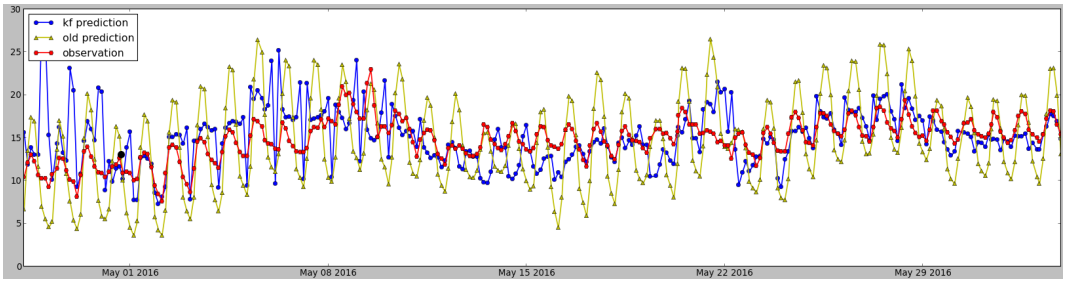


Figura 10: Temperaturas

6.0.7. Creación de los DAGs

Ahora que teníamos todos los ingredientes, lo único que quedaba por hacer era unirlo todo. Esto consistía en definir el orden en el que cada módulo sería ejecutado mediante *tasks*. Creamos cinco módulos, uno para las observaciones y cuatro para las predicciones (una para cada momento del día en el que se subían nuevos datos del GFS).

Lo más importante era que averiguásemos a que horas actualizan los datos cada una de nuestras fuentes. Para las observaciones, AEMET indica en su página web que se realiza cada hora, haciendo la tarea bastante sencilla. En cambio, para las predicciones, teníamos que tener en cuenta el cambio horario ya que los datos del modelo GFS se suben a una determinada hora en horario UTC. Los datos los suben cuatro veces al día (para a medianoche, a las 6, a las 12 y a las 18), pero aparecen alrededor de 5 horas mas tarde. Teniendo

en cuenta que en España nos regimos por el horario europeo central (UTC+1 y en verano UTC+2) teníamos que hacer que el *DAG* se ejecutase cada 4 horas, pero 7 horas mas tarde. Definimos 4 módulos, uno para cada hora, y hacemos que cada uno se ejecute una vez al día.

Como se puede ver en este fragmento de código 14, para las observaciones simplemente indicamos una fecha de inicio, cuantas veces queremos que reintente la descarga de datos en el caso de que algo falle, cuanto tiempo dejar entre reintentos, cada cuanto tiempo se ejecuta el *DAG*, los argumentos que se le van a pasar al *task* y la definición del *operator*.

Listing 14: jobObs.py

```
# -*- coding: utf-8 -*-
from airflow import DAG
from airflow.operators import PythonOperator
from datetime import datetime, timedelta
from obsAemetLoader import *

args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2016, 5, 30),
    'retries': 4,
    'retry_delay': timedelta(minutes=10)
}

dag = DAG('loadObsData', default_args=args, schedule_interval=timedelta(hours=1))
    #que se ejecute con un intervalo de 1 hora

print 'La hora de la ejecucion del DAG es:
    '+datetime.strftime(datetime.now(),'%Y-%m-%d %H:%M:%S')
argsData=[1111, 'temperatura']
t1 = PythonOperator(
    task_id='obs',
    python_callable=loadData,
    op_args=argsData,
    dag=dag)
```

Para las predicciones es parecido 15, pero con el añadido que ordenamos que se ejecute un *task* que haga pasar las predicciones por el filtro de Kalman después de haber cargado las predicciones en la base de datos.

Listing 15: jobPred00.py

```
# -*- coding: utf-8 -*-
from airflow import DAG
from airflow.operators import PythonOperator
from datetime import datetime, timedelta
from frcGfsLoader import *
from kalmanTest import *
from dateutil import tz

startDate=datetime.strptime("2016-05-30 07:00:00",'%Y-%m-%d %H:%M:%S') # durante
    todo el anio los datos de las 00 ya estaran disponibles a las 7
args = {
    'owner': 'airflow',
```

```
    'depends_on_past': False,
    'start_date': startDate,
    'retries': 4,
    'retry_delay': timedelta(minutes=15)
}

dag = DAG('pred_00', default_args=args, schedule_interval=timedelta(days=1)) #que
    se ejecute con un intervalo de 12 horas

#argsData=['2016','04','19','00',1111,'temperatura','GFS','AEMET']
print 'El tiempo a la hora de la ejecucion del DAG es:
    '+datetime.strftime(datetime.now(),'%Y-%m-%d %H:%M:%S')

year=datetime.now().year
mes=datetime.now().month
dia=datetime.now().day
hora=0


#se descarga los datos de 10 dias con una resolucion de 3 horas
argsData=[year, mes, dia, hora, 1111, 'temperatura', 'GFS', 'AEMET']
t1 = PythonOperator(
    task_id='pred00',
    python_callable=loadData,
    op_args=argsData,
    dag=dag)

argsKalman=[7,datetime.strftime(datetime(year, mes, dia, hora),'%Y-%m-%d
    %H:00:00')]
t2 = PythonOperator(
    task_id='kalman00',
    python_callable=kalmanTest,
    op_args=argsKalman,
    dag=dag)

t2.set_upstream(t1)
```

6.0.8. Creación de la aplicación híbrida

Llegados a este punto, decidimos crear la aplicación móvil para Android con la que mostrar los datos. Por el momento la estamos usando para mostrar los datos generadas por la empresa Predictia con el modelo WRF ya que estos muestran los datos de una variedad de magnitudes atmosféricas, mientras que el filtro de Kalman no es capaz de corregir con exactitud todas las variables meteorológicas (sí vale para temperatura o viento pero no para precipitación). Además uno de los objetivos de la aplicación era proporcionar predicciones en cualquier punto de España. Por tanto, para mostrar las predicciones corregidas tendríamos que definir un filtro de Kalman (con la captura de observaciones, etc.) en cada localidad con una estación meteorológica lo cual supondría un coste computacional y muy importante. Aun así la aplicación la hemos construido de manera que sea lo suficiente versátil como para poder mostrar los datos originarios de cualquier fuente.

Para desarrollar la aplicación decidimos usar Ionic 3.6, un framework que permite que creemos una aplicación en el lado del cliente con el uso de AngularJS y que diseñemos la aplicación como si se tratase de una aplicación web, con el uso de HTML, CSS y Javascript, al mismo tiempo que ofrece la funcionalidad de una aplicación nativa como el acceso al

giroscopio, al GPS, etc. En el lado del servidor íbamos a necesitar un servicio que estuviese a la espera de recibir peticiones de la aplicación para pasarle los datos que pidiese. Elegimos abordar esta tarea con la ayuda de web2py 3.5. Nos decantamos por usar web2py por ser un framework escrito y programable en Python y muy fácil de aprender a usar, funcionó desde el primer momento, no hizo falta ni instalar ni configurarlo, ya que funciona en cualquier arquitectura en la que se pueda ejecutar Python. Además tiene una interfaz web desde la que se puede realizar las fases de desarrollo, despliegue y mantenimiento de las aplicaciones. También proporciona un debugger lo que nos fue de bastante utilidad a la hora de detectar e identificar los errores que fuimos cometiendo.

La interfaz de aplicación se divide en tres partes. Por un lado tenemos la página inicial que muestra los datos de la pronóstico del tiempo, para los próximos 3 días en nuestro caso, como se puede ver en la figura 11.

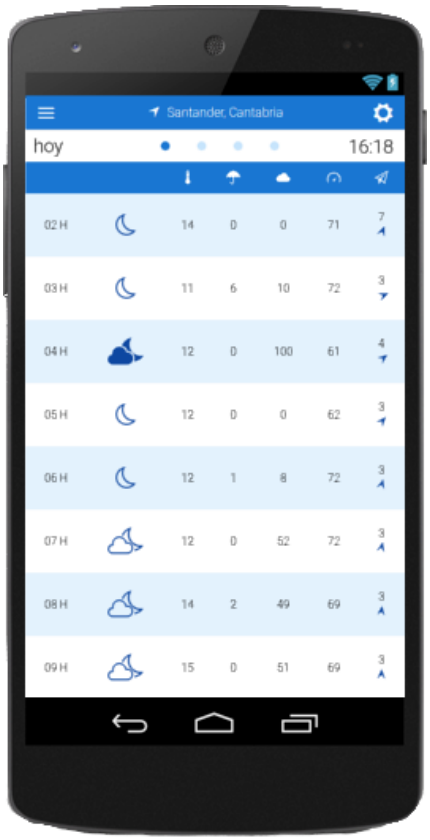


Figura 11: Página principal

Al iniciar la aplicación, un controlador se encarga de llamar a una función 16 que mediante la geolocalización implementada con la API de Google Maps devuelve las coordenadas en las que te encuentras. Después se las pasa mediante una petición HTTP al servidor web2py para que este, con una implementación del módulo de carga de datos del servidor de Predictia 6.0.5, serialice los datos sacados a un formato JSON para que la aplicación la lea y sea capaz de interpretarlos. Dependiendo de la hora que sea y del estado del tiempo atmosférico, se mostrarán distintos iconos para describir el tiempo. En el código del controlador 17 se vé como se saca información adicional para mostrar y organizar los datos en la interfaz de la aplicación con la ayuda de AngularJS.

Listing 16: controller.js

```
$scope.refreshCurrent = function() {
   GeoService.getLocation().then(function(position) {
      var lat = position.coords.latitude;
      var lng = position.coords.longitude;

      _this.predictions(lat,lng);

      GeoService.reverseGeocode(lat, lng).then(function(locString) {
         $scope.currentLocationString = locString;
      });
    });
};
```

Listing 17: controller.js

```
this.predictions = function(lat,lng) {
   ForecastService.getForecasts(lat,lng).then(
     function(data){
       $scope.forecasts = data.temperaturas;

       var dictionary= {};

      function insertIntoDic(key, value) {
         if (!dictionary[key] || !(dictionary[key] instanceof Array)) {
            dictionary[key] = [];
         }
         dictionary[key] =
             dictionary[key].concat(Array.prototype.slice.call(arguments, 1));
      }
       for (pred in $scope.forecasts){
         var now = new Date($scope.forecasts[pred].time);
         var day = now.getDate(); //returns from 1 to 31
         var month = now.getMonth()+1; //returns from 0 to 11
         var year = now.getFullYear();
         //var date = String(day+"/"+month+"/"+year);
         var date = String(year+"/"+month+"/"+day);
         insertIntoDic(date,$scope.forecasts[pred]);
      }
      $scope.dictDays=dictionary;
        hideLoading();
      },
      function(mensajeError){
      hideLoading();
      $scope.msj=mensajeError;
      var alertPopup = $ionicPopup.show({
        title: '<h3>Error</h3>',
        subTitle: '<h5>Error a la hora de obtener la prediccion:</h5>',
          templateUrl: 'templates/alert.html',
          scope: $scope,
          buttons: [
          { text: '<b>Aceptar</b>',
            type: 'button-positive'
          }
          ]
        });
      }
```

```
    );
  };
```

Listing 18: services.js

```
angular.module('ionic.weather.services', ['ngResource'])

.factory('GeoService', function($q) {
  return {
    reverseGeocode: function(lat, lng) {
      var q = $q.defer();
      var geocoder = new google.maps.Geocoder();
      geocoder.geocode({
        'latLng': new google.maps.LatLng(lat, lng)
      }, function(results, status) {
        if (status == google.maps.GeocoderStatus.OK) {
          if(results.length > 1) {
            var r = results[1];
            var a, types;
            var parts = [];
            var foundLocality = false;
            var foundState = false;
            for(var i = 0; i < r.address_components.length; i++) {
              a = r.address_components[i];
              types = a.types;
              for(var j = 0; j < types.length; j++) {
                if(!foundLocality && types[j] == 'locality') {
                  foundLocality = true;
                  parts.push(a.long_name);
                } else if(!foundState && types[j] == 'administrative_area_level_1') {
                  foundState = true;
                  parts.push(a.short_name);
                }
              }
            }
            q.resolve(parts.join(', '));
          }
        } else {
          q.reject(results);
        }
      });
      return q.promise;
    },
    getLocation: function() {
      var q = $q.defer();
      navigator.geolocation.getCurrentPosition(function(position) {
        q.resolve(position);
      }, function(error) {
        q.reject(error);
      });
      return q.promise;
    }
  };
})

.factory('ForecastService', ['$http', '$q', function($http, $q){
   this.getForecasts = function(latitude,longitude){
```

```
      var deferred = $q.defer();
      $http.get('http://127.0.0.1:8000/Prueba/callWrf/
            index.json?latitude='+latitude+'&longitude='
            +longitude).success(function(data){
         deferred.resolve({'temperaturas': data.temperaturas});
      })
      .error(function(msg, code) {
         deferred.reject(msg);
      });
      return deferred.promise;
   };
   return this;
}])
```

Por la parte del servidor web2py, los únicos puntos de los que tuvimos que estar pendientes fueron que teníamos que configurar el sistema de la cache y asegurarnos de definir las cabeceras de respuesta para poder aceptar peticiones de cualquier origen como se puede ver en el siguiente fragmento de código 19.

Listing 19: callGfs.js

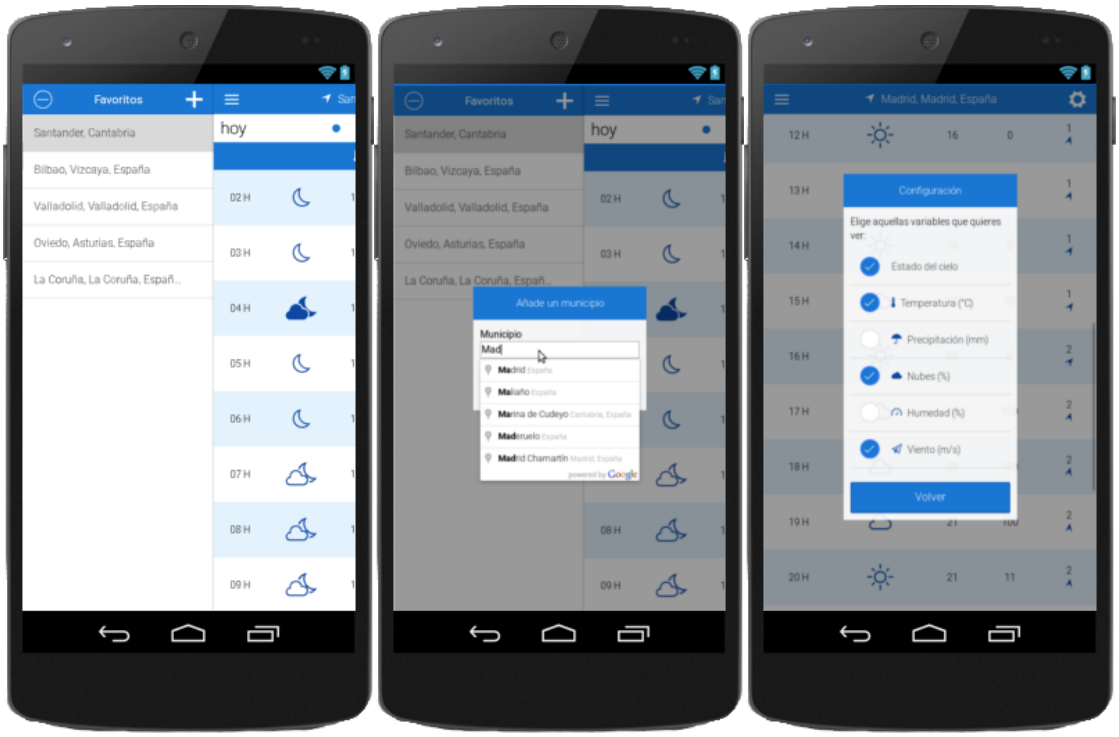
```
cache.disk.folder = os.path.join('/home/minondoa/',
    'Appz/web2py/applications/My_Aplication/')
@cache.action(time_expire=21600, cache_model=cache.disk)
def index():
    response.headers['Access-Control-Allow-Origin'] = request.env.http_origin
    response.headers['Access-Control-Allow-Origin'] = "*"
    response["Access-Control-Allow-Methods"] = "POST, GET, OPTIONS"
```

Otra parte de la aplicación es el un menú de favoritos que se despliega como un menú lateral. En él podemos buscar y añadir ciudades, seleccionar una entre las que tienes definidas y borrarlas como se puede ver en la figura 12a.

El menú se puede desplegar o deslizando el dedo de izquierda a derecha o dándole al icono de menú. En este panel podemos pulsar el botón de añadir ciudad, representado por el icono de la suma, que hará que una ventana emergente aparezca donde tendremos que escribir la ciudad de la que deseamos obtener las predicciones. El campo de texto usa un paquete de auto-completado que emplea la API de Google Places para ir sugiriendo direcciones dentro de España como se puede apreciar en la figura 12b.

Al añadir la ciudad a la lista de favoritos se reciben las coordenadas de dicha ciudad y se vuelve a pasar por el mismo proceso que al iniciar la aplicación solo que con las nuevas coordenadas, permitiéndonos ver su pronóstico. Una vez añadido a la lista, se mantiene hasta que decidamos borrarlo.

Es posible seleccionar entre las ciudades guardadas, lo que causará que se muestren las predicciones de la ciudad señalada, con la diferencia de que esta vez no hará falta buscar las coordenadas, ya estarán guardadas, y si los datos de las predicciones no han sido actualizadas se mostrarán los datos de manera bastante mas rápida ya que tenemos la cache que configuramos en web2py para los ciudades visitadas con anterioridad.



(a) Menú de favoritos (b) Añadiendo una ciudad nueva (c) Menú emergente

Figura 12: Múltiples imágenes de la aplicación

La tercera parte consiste en un menú emergente que permite que elijamos las variables meteorológicas que queramos ver, como muestra la figura 12c.

7. Conclusiones y Trabajos Futuros

Para finalizar la memoria, expondremos cuales han sido las conclusiones a las que hemos llegado una vez que acabamos con el proyecto.

Por un lado, logramos satisfacer todos los requisitos que se establecieron y las herramientas empleadas fueron de gran ayuda para resolver los problemas a los que nos enfrentamos. La interfaz proporciona por Airflow y web2py fueron muy útiles a la hora de identificar los fallos cometidos durante el desarrollo

Una vez que conseguimos extraer, procesar y guardar los datos provenientes de una fuente de datos, fuimos capaces de hacer lo mismo desde otras fuente con mayor facilidad y reduciendo de manera considerable el tiempo de desarrollo y aunque todavía podríamos incluir algunas optimizaciones todo funciona correctamente.

El filtro de Kalman se ha implementado en otros ámbitos que han conseguido mejorar gracias a ello, y lo mismo se puede apreciar con las predicciones y testimonio de ello es que hemos podido comprobar como las predicciones se iban acercando al valor de las observaciones con el paso del tiempo. Aunque no sea capaz de corregir todas las variables meteorológicas sigue siendo una aportación que merece la pena. Sobre la implementación del filtro de Kalman solo nos quedaría extender el sistema para que considerase el resto de las variables meteorológicas válidas.

En un futuro podríamos ampliar la aplicación móvil para que mostrase los datos corregidos obtenidos del filtro de Kalman. Podríamos añadir nuevas funcionalidades como mostrar gráficos que describan el cambio del tiempo, mostrar el tiempo actual además de las predicciones e incluir una apartado en la que se comparen predicciones anteriores con los valores obtenidos de las observaciones para ver cuánto nos acercamos, añadir la opción de buscar localidades a través de un mapa además de poder hacerlo con un buscador, etc. También nos queda subir la aplicación a la plataforma de distribución digital de aplicaciones móviles *Google Play Store*.

Las aplicaciones que tiene el sistema son extensas, ya que se puede adaptar para funcionar en cualquier parte del mundo en la que hayan mediciones del tiempo disponibles, y sobre la aplicación móvil, todo depende de lo que se quiera expandir.

Referencias

[1] Rosa María Rodríguez Jiménez, Meteorología y Climatología, Edita: FECYT (Fundación Española para la Ciencia y la Tecnología), https://cab.inta-csic.es/uploads/culturacientifica/adjuntos/20130121115236.pdf

[2] Modelo numérico de predicción meteorológica, Wikipedia, La enciclopedia libre, https://es.wikipedia.org/wiki/Modelo_numérico_de_predicción_meteorológica

[3] G. Galanis, "Applications of Kalman filters based on non-linear functions to numerical weather predictions", 24 August 2006

[4] Climate and Forecast Metadata Conventions, Wikipedia, La enciclopedia libre, https://en.wikipedia.org/wiki/Climate_and_Forecast_Metadata_Conventions

[5] NOAA Operational Model Archive and Distribution System NOMADS, http://nomads.ncep.noaa.gov/txt_descriptions/NOMADS_doc.shtml

[6] Rudolf E. Kalman, "A New Approach to Linear Filtering and Prediction Problems", The Seminal Kalman Filter Paper (1960)

[7] Andrews, Grewal. Kalman Filtering Theory And Practice Using Matlab. New York: Wiley, 2001

[8] Haykin, Simon. Neural Networks. A comprehensive Foundation. Pearson, India, 1999

[9] Dorado, J; Ruíz, J. F., Implementación de Filtros de Kalman como Método de Ajuste a los Modelos de Pronóstico (GFS) de Temperaturas Máximas y Mínima para algunas Ciudades de Colombia, Grupo de Modelamiento de Tiempo y Clima

[10] OPeNDAP, Wikipedia, La enciclopedia libre, https://en.wikipedia.org/wiki/OPeNDAP

[11] Python Package Index, Wikipedia, La enciclopedia libre, https://es.wikipedia.org/wiki/Python_Package_Index

[12] Airflow, http://pythonhosted.org/airflow/

[13] SQuirreL SQL Client, http://squirrel-sql.sourceforge.net

[14] Web2py, Wikipedia, La enciclopedia libre, https://es.wikipedia.org/wiki/Web2py

[15] Ionic (mobile app framework), Wikipedia, La enciclopedia libre, https://en.wikipedia.org/wiki/Ionic_(mobile_app_framework)

[16] Carlos Azaustre, ¿QUÉ ES ANGULARJS?, https://carlosazaustre.es/blog/empezando-con-angular-js

[17] AngularJS, Wikipedia, La enciclopedia libre, https://es.wikipedia.org/wiki/AngularJS

[18] Apache Cordova, Wikipedia, La enciclopedia libre, https://en.wikipedia.org/wiki/Apache_Cordova

[19] Metodologías y Ciclos de Vida, Laboratorio Nacional de Calidad del Softwaree de INTECO

[20] Airflow Concepts, https://pythonhosted.org/airflow/concepts.html#operators

[21] Airflow Concepts, https://pythonhosted.org/airflow/concepts.html#tasks

[22] Página: Airflow Concepts, https://pythonhosted.org/airflow/concepts.html#dags

[23] CSV, Wikipedia, La enciclopedia libre, https://es.wikipedia.org/wiki/CSV

[24] The Weather Research & Forecasting Model, http://www.wrf-model.org/index.php

[25] https://prontoprofvi.wordpress.com/2015/09/25/vulcani-e-planisferi/