75.71 Seminario de Ing. en Informatica I

Amura, Federico Gavrilov, Sebastian

Índice general

Sistema de valoración de cierre de lotes de soja

Estructura

El proyecto se encuentra dividido en diferentes módulos que se debieran ejecutar secuencialmente para poder construir el modelo final. Los módulos son

- Base de datos
- Repositorio de entidades comunes
- ETL para carga inicial
- Entrenador de modelo de evaluación
- API de consulta de valores de cierre

La gestión para el desarrollo de los módulos se maneja bajo un proyecto umbrella que maneja las dependencias entre ellos, en general agregando las dependencias comunes al proyecto especifico

Ejecución

Para correr el proyecto, tenemos diferentes alternativas. La única dependencia del sistema es docker y compose. Para correr el sistema podemos usar una de las distintas opciones:

```
# correr completo en modo desarrollo
./script/upDev.sh

# correr completo en modo produccion
./script/upProd.sh

# para correr partes especificas en modo desarrollo
docker-compose -f ./docker/docker-compose.base.yml \
               -f ./docker/docker-compose.dev.yml up postgres
docker-compose -f ./docker/docker-compose.base.yml \
               -f ./docker/docker-compose.dev.yml up dbFiller
docker-compose -f ./docker/docker-compose.base.yml \
               -f ./docker/docker-compose.dev.yml up trainer
docker-compose -f ./docker/docker-compose.base.yml \
               -f ./docker/docker-compose.dev.yml up api

# para correr partes especificas en modo producción
docker-compose -f ./docker/docker-compose.base.yml \
               -f ./docker/docker-compose.dev.yml up postgres
docker-compose -f ./docker/docker-compose.base.yml \
               -f ./docker/docker-compose.dev.yml up dbFiller
docker-compose -f ./docker/docker-compose.base.yml \
               -f ./docker/docker-compose.dev.yml up trainer
docker-compose -f ./docker/docker-compose.base.yml \
               -f ./docker/docker-compose.dev.yml up api
```

Base de datos

La base de datos esta sobre el motor PostgreSQL, cuenta con una única tabla que almacena los valores de los distintos cierres de los lotes de soja. Cuenta con los siguientes campos:

Campo	Tipo
id	**integer, primary key**
fecha	text
open	double
high	double
low	double
last	double
cierre	double
ajdif	double
mon	text, default 'D'
oivol	integer
oidif	integer
volope	integer
unidad	text, default 'TONS'
dolarbn	double
dolaritau	double
difsem	double
hash	**integer, unique**

Se agrega el hash para poder, desde los datos, determinar si esa entrada ya se encuentra evaluada y persistida en la tabla.

Clases comunes

Nombre de proyecto: commons

En este subproyecto se incluyen las clases comunes que se usan a través de todo el resto de los proyectos. Incluye:

- Cierre: encapsula el valor de cierre que tuvo una valuación
- DB: conexión a la base de datos y queries que se ejecutan sobre ella
- Row: que representa una entrada en la base de datos
- SoyRequest: una consulta al sistema sobre la cual se puede generar un cierre y generar entradas en la base de datos

Carga de datos

Nombre de proyecto: dbFiller

Este proyecto toma la entrada desde un archivo de datos y los inserta en la base de datos para posteriormente entrenar el modelo de evaluación o ser dispuestos por la API.

El programa de este proyecto esta compuesto por 2 mónadas IO. La primera mónada IO, lee las lineas del archivo de entrenamiento, las mapea a la clase Row y devuelve una lista de estas. Acá se emplea la clase `Resource` de la biblioteca cats para asegurar el cierre del archivo una vez hecho el procesamiento. La segunda IO, simplemente inserta la lista de Rows en la base de datos. Para el acceso a base de datos utilizamos la biblioteca `doobie`, que da una capa funcional de acceso a la base de datos. Usando `doobie` los accesos a la base de datos se pueden representar como mónadas IO, y luego se pueden componer.

Las mónadas se componen con una for comprehension como un ETL y finalmente se lo ejecuta.

Prediccion de valores

Nombre de proyecto: trainer

Este paso levanta un cluster Spark en modo local y entrena un modelo RandomForest en su versión para regresión basado en los datos cargados en la base. Este modelo luego sirve para generar datos de cierre para nuevas valuaciones recibidas en la API.

Se compone de tres etapas:

- Obtención de los datos de la base de datos
- Entrenamiento del modelo en spark
- Serialización del modelo en un archivo en formato PMML

El primer paso es un simple acceso a la base de datos, encerrado en una mónada IO proveída por `doobie`. El segundo paso tiene varias etapas.

- Primero la creación del cluster en modo local, que está modelada como un `Resource` de `cats`, que asegura que el cluster es terminado correctamente. Luego el procesamiento de datos.
- Luego procesamiento de datos de entrada para convertirlos en un set de entrenamiento y de prueba. Para eso se utiliza la biblioteca `frameless`, que provee una capa funcional sobre los procesamientos de spark. En este paso los datos de entrada se filtran para quedar con solo los valores usados para entrenar y la etiqueta a predecir (Cierre). Para entrenar se eligió un subset pequeño de columnas representativas: DolarBN, DolarItau y DifSem. Una vez filtrados, se los separa en sets de entrenamiento y de test (80% y 20% respectivamente), y se los compone en un solo vector de features (utilizando Vector Assembler de `frameless`).
- Se realiza el entrenamiento. Elegimos el algoritmo de Random Forest, por su relativa facilidad de empleo en ese pipeline (para la otra alternativa, regresión de XGBoost, no se encontró una forma fácil de serializarla). La implementación del algoritmo que se utilizó es la que provee `frameless`, asi aprovechamos sus checkeos de tipo en tiempo de compilación.
- Finalmente se convierte el modelo en una tira de bytes en formato PMML para su futura serialización. Para esto, se juntan el *Vector Assembler* y el modelo de *Random Forest* en un *Pipeline* de spark. Como las implementaciones de `frameless` no son directamente compatibles, para generar ese pipeline se tuvo que acceder a atributos de implementación, dejandonos para después crear una abstracción adecuada. Para convertir el Pipeline en PMML se utilizó la biblioteca `jpmml-sparkml`.

El último paso escribe la tira de bytes del paso anterior en un archivo. Este paso nuevamente es modelado como un Resource y una mónada IO que realiza la escritura.

API de consultas

Nombre de proyecto: api

Dispone un servicio API para consultar los valores de cierre de ciertos lotes, en caso de ser un dato conocido, se devuelve el valor de cierre que tuvo; si no lo es, entonces utiliza el modelo para generar un nuevo dato de cierre, persistiendolo en la base de datos y devolviendo esta nueva valuación.

Basado en http4s, el programa básico con cada request, mediante una for comprehension, es - Tomar el body del request como SoyRequest - Procesarlo en nuestra aplicacion - Devolver el resultado de cierre como JSON

El procesamiento del request es implica primero conseguir el cierre del mismo a partir de sus datos, como depende del modelo, lo tenemos incluido en una mónada Try, la cual después, mediante pattern matching podemos operar en su valor o tirar la exception directamente (esto podría mejorarse). En caso de ser exitosa la evaluación, entonces se ejecuta el programa que inserta y devuelve el cierre sobre la base de datos Este ultimo programa, es una composición de uno, que inserta los datos en la base de datos, pero solo en caso que no existan aprovechando que podemos identificarlo mediante el hash, y otro, que recupera el dato de cierre de la base (que teníamos desde los datos de entrenamiento o del modelo)