

Sistema de Pronóstico de Demanda con Arquitectura POO

Implementación de Modelos de Ensamble (Prophet, SARIMA, Random Forest)

Presentado por: Roberto Alejandro Chagra Martinez



El Problema y la Solución

Problema: Predecir la demanda futura de productos basándose en históricos de ventas, un desafío crítico para la optimización de inventarios y recursos.

Solución: Desarrollar un sistema de software robusto que combine múltiples algoritmos de Inteligencia Artificial para generar pronósticos precisos.

Enfoque: Una arquitectura Orientada a Objetos (POO) para garantizar la modularidad, escalabilidad y facilidad de mantenimiento del sistema.

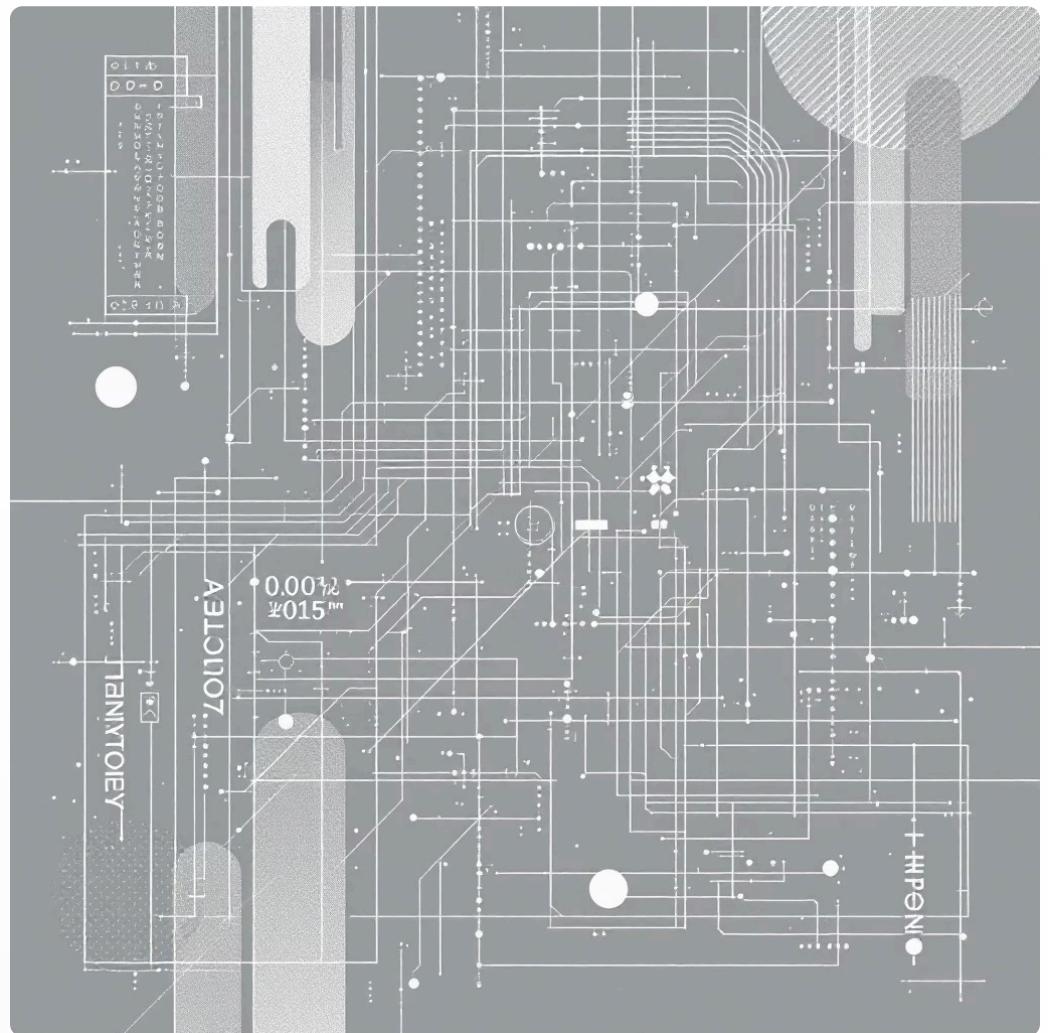


Abstracción: La Superclase

La abstracción es fundamental en POO, permitiendo definir una plantilla estricta. En nuestro sistema, la clase `ModeloPronostico` actúa como una interfaz, asegurando que todos los modelos de pronóstico que desarrollemos cumplan con los métodos esenciales: `entrenar` y `predecir`. Esto fomenta un diseño coherente y robusto.

Código (Líneas 118-132):

```
class ModeloPronostico(ABC):
    @abstractmethod
    def entrenar(self, df_train):
        pass
    @abstractmethod
    def predecir(self, fechas_futuras):
        pass
```





Herencia: Reutilización y Especialización

Concepto

La herencia permite a las clases heredar propiedades y comportamientos de una clase base. Esto promueve la reutilización de código y la creación de clases más específicas sin duplicar funcionalidades.

En nuestro proyecto, las subclases `ModeloProphet`, `ModeloSARIMA` y `ModeloRandomForest` heredan de `ModeloPronostico`.

Código (Líneas 134, 153, 172)

```
class ModeloProphet(ModeloPronostico): ...
class ModeloSARIMA(ModeloPronostico): ...
class ModeloRandomForest(ModeloPronostico): ...
```

Como pueden ver en los constructores `__init__`, utilizamos correctamente `super().__init__()` para inicializar los atributos de la clase padre antes de configurar los parámetros específicos de cada algoritmo.

Polimorfismo: El Núcleo de la Flexibilidad

El polimorfismo nos permite tratar objetos de diferentes clases de manera uniforme si comparten una interfaz común (la superclase). Con él, podemos llamar al mismo método y obtener un comportamiento específico para cada tipo de objeto.

Aquí está la parte más importante del proyecto: En la clase gestora, iteramos sobre una lista de objetos mixtos. Gracias al polimorfismo, llamamos al método `.entrenar()` sin usar ningún `if` o `else` para verificar qué modelo es. Python sabe dinámicamente si debe ejecutar el entrenamiento de redes neuronales de Prophet o la regresión de Random Forest. Esto hace que el código sea limpio y fácil de extender, permitiendo añadir nuevos modelos sin modificar la lógica principal.

Código (Línea 248):

```
for modelo in self.modelos:  
    modelo.entrenar(df_serie)  
    predicciones = modelo.predecir(fechas_futuras)
```



Clase Gestora: Encapsulamiento y Composición

La MotorPronostico es la clase orquestadora de nuestro sistema. Encapsula toda la lógica de negocio, gestionando el flujo de datos, el entrenamiento de modelos y la generación de resultados.

Cumple con el requisito de tener una lista de objetos como atributo (self.modelos). Además, utiliza el principio de Composición al crear instancias de GestorDatos y AnalizadorClusters dentro de su constructor, delegando tareas específicas a cada componente y manteniendo la cohesión.

Código a mostrar (Línea 210):

```
class MotorPronostico:  
    def __init__(self):  
        self.gestor = GestorDatos()  
        self.clusterer = AnalizadorClusters()  
        self.resultados = []  
        self.modelos = [ModeloProphet(), ModeloSARIMA(),  
...]
```



Resultados y Demostración Visual

El resultado final no es solo código; es una herramienta funcional que impacta directamente en la toma de decisiones. El sistema procesa los datos históricos, entrena los tres modelos de forma independiente, y luego promedia sus resultados mediante una estrategia de Ensamble. Esto genera pronósticos más estables y precisos que los de un modelo individual.

Finalmente, el sistema genera visualizaciones claras y concisas, como la que se muestra, permitiendo a los usuarios empresariales tomar decisiones informadas basadas en datos, optimizando así inventarios, logística y estrategias de venta.





Conclusión



Código Modular (POO)

He diseñado un sistema con componentes bien definidos y desacoplados, facilitando su comprensión y desarrollo futuro.



Escalable

La arquitectura permite agregar nuevos modelos predictivos con mínimo esfuerzo, adaptándose a las necesidades cambiantes del negocio.



Funcional

El sistema resuelve un problema real y crítico, proporcionando pronósticos de demanda precisos para la toma de decisiones estratégicas.

En conclusión, he logrado unir la ciencia de datos con la ingeniería de software. No solo tenemos modelos predictivos precisos, sino una arquitectura de software profesional que cumple con todos los estándares de la Programación Orientada a Objetos.

Consideraciones Adicionales

1 ¿Por qué usamos ABC?

Abstracción: Usamos la clase ABC (Abstract Base Class) para asegurar que cualquier subclase de `ModeloPronostico` implemente necesariamente los métodos `entrenar` y `predecir`. Esto evita la creación de modelos incompletos que romperían el sistema en tiempo de ejecución, actuando como una medida de seguridad en el diseño y garantizando la coherencia.

2 ¿Qué es self?

Referencia de Instancia: `self` es una convención en Python que se refiere a la instancia actual del objeto. Permite a los métodos acceder a los atributos y otros métodos específicos de esa instancia. Es cómo el objeto "se conoce a sí mismo" y maneja sus propios datos.

3 ¿Dónde se encuentra el Encapsulamiento?

Clase GestorDatos: El encapsulamiento se evidencia en la clase `GestorDatos`. Aquí, ocultamos la complejidad interna de tareas como la limpieza de archivos CSV y la transformación de datos (hacer el '`melt`'). La clase `MotorPronostico` solo necesita pedirle a `GestorDatos` "cargar datos" sin preocuparse por los detalles de cómo se realiza esa operación internamente, promoviendo la separación de preocupaciones y la simplicidad de la interfaz.