**El objeto estimador de Scikit-Learn**

Scikit-learn es la libreria más usada de Machine Learning tradicional. La librería incluye funcionalidades de:

* Preprocesamiento de datos en **sklearn.preprocessing**
* Algoritmos de Machine Learning en **sklearn.linear\_model, sklearn.svm, sklearn.ensamble**, y muchos mas.
* Evaluación de modelos en **sklearn.model\_selectio**n y **sklearn.metrics**

Una estructura de datos esencial en Scikit-learn es el **Estimator**

Imagen que contiene texto, mapa

Descripción generada automáticamente

**Elegir el estimador correcto**

A menudo, la parte más difícil de resolver un problema de aprendizaje automático puede ser encontrar el estimador adecuado para el trabajo.

Los diferentes estimadores son más adecuados para diferentes tipos de datos y diferentes problemas.

El siguiente diagrama de flujo está diseñado para brindar a los usuarios una guía aproximada sobre cómo abordar los problemas con respecto a qué estimadores probar sus datos.

Haga clic en cualquier estimador en el cuadro a continuación para ver su documentación.

Imagen que contiene mapa

Descripción generada automáticamente

**Regresion Lasso**

Total, de los datos 35472

60 % de los datos es 21283

40 % de los datos es 14189

Captura de pantalla de un celular

Descripción generada automáticamente

Debemos evaluar de forma más fina el comportamiento de nuestro modelo.

**Evaluamos de forma más fina el comportamiento de nuestro modelo**

Los estimadores y las funciones de **sklearn** vienen con el máximo de argumentos con valores por defecto que suelen ser las mejores opciones, si no tenemos algún conocimiento particular el problema. En este caso particular la función **Estimator.score** ya viene con una de las métricas de **sklearn.metrics**, que es la métrica **sklearn.metric.r2\_score**

El score r2 entre mas cercano al 1 es mejor, podemos entender su poder predictivo

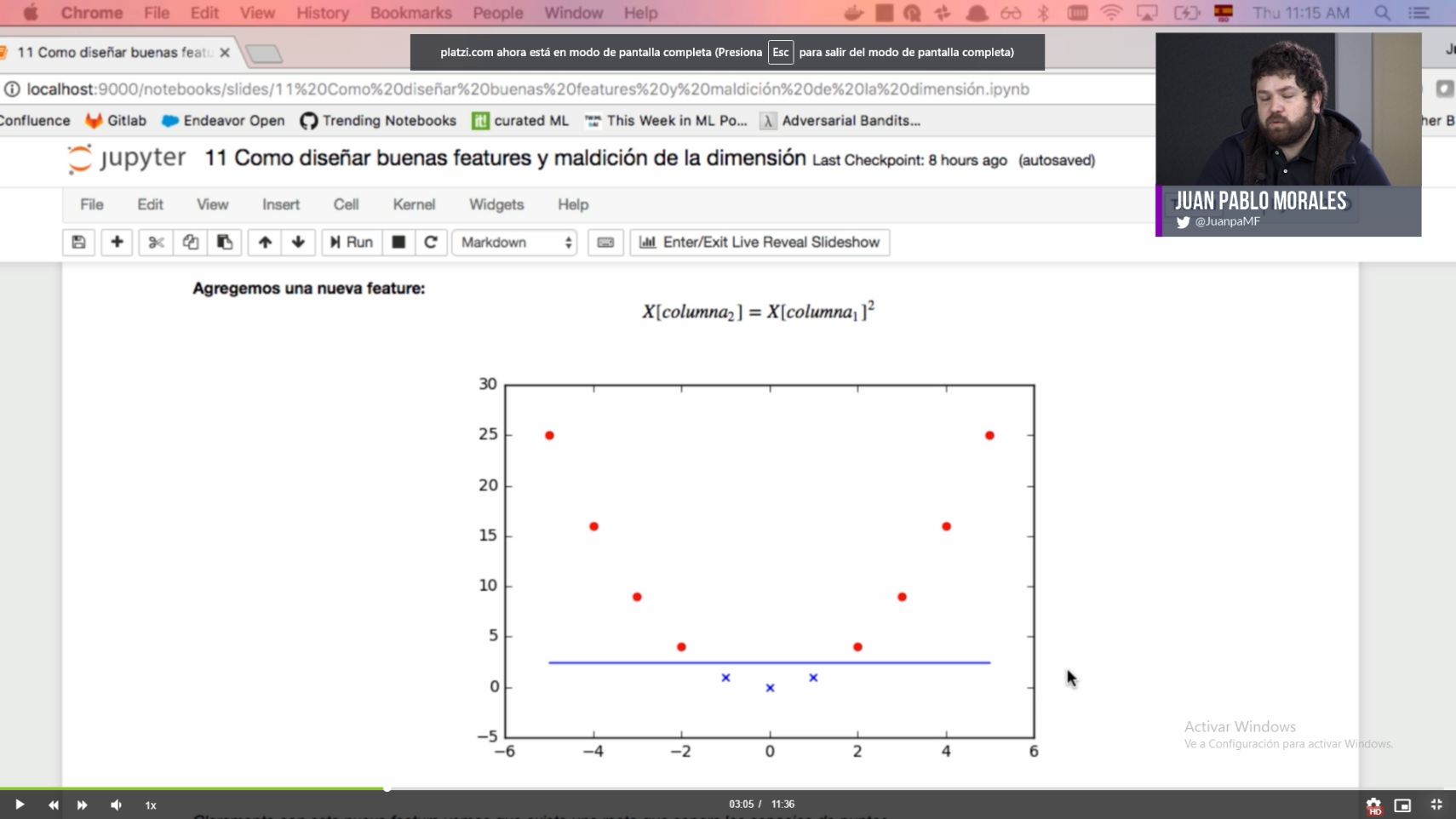
-0.00020262342253540844

Muy malo el Score para un primer modelo, ahora lo que hacemos es tratar de mejorarlo

**Agregamos una nueva feature**

Nuestros datos son de una dimensión por lo cual lo vamos a modificar agregándole una segunda columna con la siguiente formula.

X[columna2] = X[columna1]^2



Podemos observar que tenemos una linea recta, la cual se para nuestros datos ya que aumentamos la dimensión de los datos, ahora el modelo puede hacer mas cosas, todo esto se hace para tener un **performance** que se acerque al 100%

Es una buena intuición agregarle una segunda columna nuestro resultado puede mejorar bastante,

Nuestra segunda columna será X^2

**Principios de diseño de Features**

Diseñar nuestras features es un arte más que una ciencia

* **Features informativas:** Tus features son más útiles mientras más correlación tengan la variable objetivo
* **Features independientes:** Para no tener redundancia las features deberían ser lo más independientes posible entre ellas.
* **Cantidad de features controlada:** Nuestra intuición nos falla en dimensiones superiores a 3 (maldición de la dimensionalidad). En la mayoría de los casos aumentar la cantidad de features afecta negativamente la performance si no contamos con una gran cantidad de datos. Por último, pocas features aseguran una mejor interpretabilidad de los modelos.

**Visualizar interdependencia entre variables**

Matriz de correlación

Imagen que contiene dibujo

Descripción generada automáticamente

En general observamos que nuestras feaures no tienen correlación, es decir que son independientes, justo a nuestro objetivo.

Si tienes mas features tu modelo sera mas complejo. Para entrenarlo necesitaras mas datos.

**Feedback de nuestro modelo**

Observamos el feedback de nuestras features, ciertos modelos como la regresión o los árboles se dicen “**interpretables**”. Esto quiere decir que de los resultados de los modelos podemos sacar conclusiones o “**insights**”

**La regresión Lasso** nos habla de que tan importantes fueron nuestras feature, en la predicción que hizo la regresión Lasso.

Imagen que contiene ave, pájaro

Descripción generada automáticamente

Haciendo una segunda interacción en la regresión Lasso nos detalla que las mejores variables son Objectid y radicado, pero estas variables no están con respecto a nuestro objetivo.

**Creación y transformación de features**

**Escalamiento de datos**

Diversos algoritmos son sensibles a la escala en la que viene cada feature. **Re-escalarlos** puede tener significativas mejoras de rendimiento.

Existen distintas estrategias de escalamiento de las features, pero **la más común es la estandarización** donde convertimos la variable para que la distribución de esta siga una distribución que es Gaussiana de media 0 y de desviación estándar 1.

Modelo sin escalar = -0.00017992462089266859

Modelo escalado = -4.751553575021994e-05

**Crear nuevas features de forma automática**

**Generar características polinómicas y de interacción.**

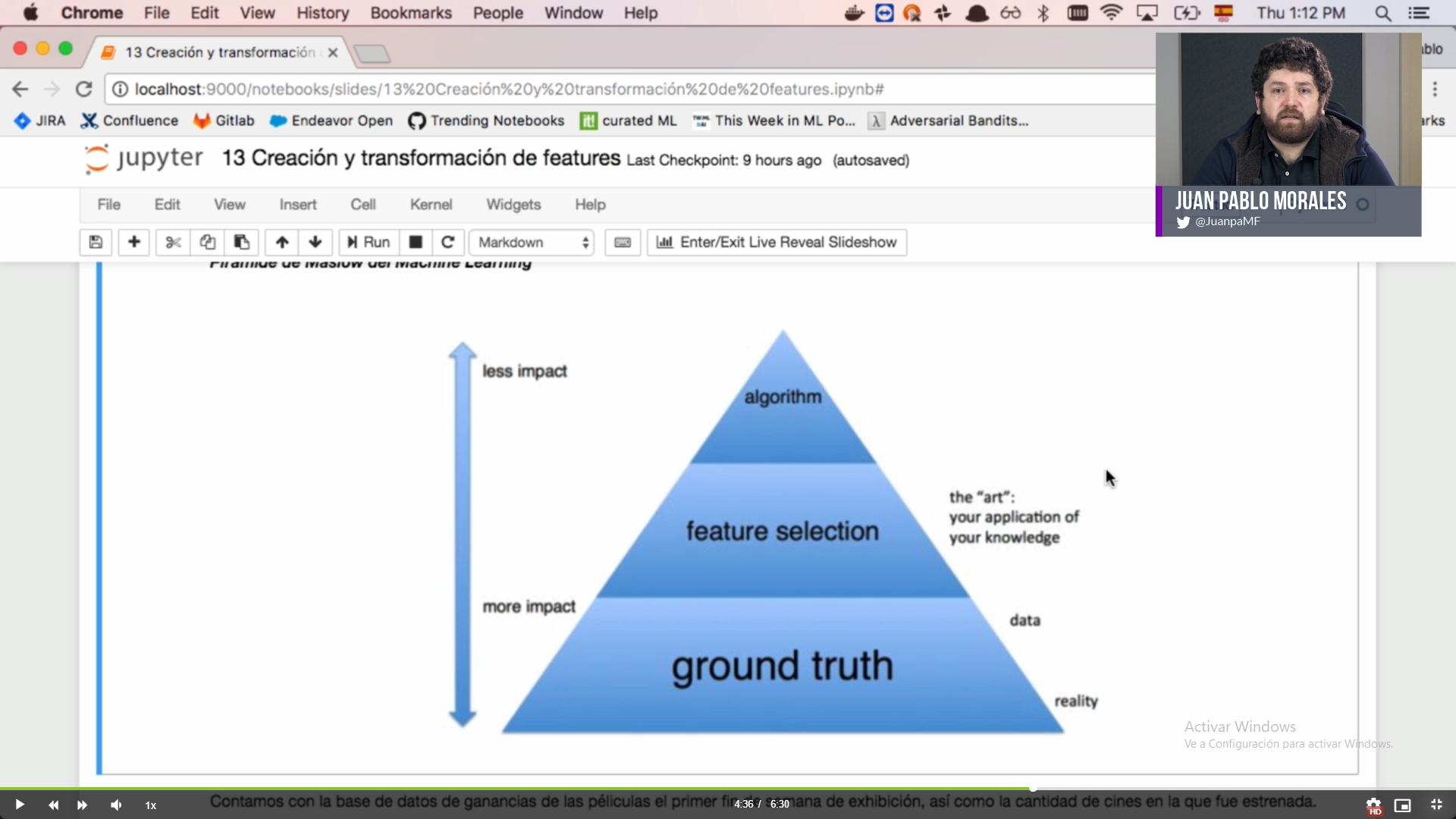
**Genere una nueva matriz de entidades que consista en todas las combinaciones polinómicas de las entidades con un grado menor o igual al grado especificado. Por ejemplo, si una muestra de entrada es bidimensional y tiene la forma [a, b], las características polinómicas de grado 2 son [1, a, b, a ^ 2, ab, b ^ 2].**

Mejoramos nuestro Score : 0.2710706417653941

**Mas datos de calidad**

**Piramide de Maslow del Machine Learning**

Entre más abajo más importantes son y entre más arriba más cómodas



Métodos de evaluación

Por ahora podemos concluir lo siguiente:

* Se necesita **separa de forma aleatoria** en datos de entrenamiento y testeo para pode evaluar performance del algoritmo
* Existen diversas **métricas para evaluar rendimiento**, y elegimos la nuestra según las características de nuestro problema.
* Es útil **apoyar la evaluación con visualización de errores**, como por ejemplo scatterplots de residuales.

Sin embargo, nuestro método hasta ahora tiene una falla. Este depende de la forma en que fueron elegidos nuestros datos de forma aleatoria:

* Podemos tener suerte y caer en un train set y test set que sea ideal para nuestro modelo.
* Podemos tener pésima performance con esa separación de datos, pero no en otros.

**Controlar la aleatoriedad en train\_test\_split**

**Train\_test\_split** separa cada vez que lo llamamos los datos de forma diferente. Para poder compara modelos, hacer un codigo mas limpio y compacto y para poder hacer nuestros experimentos reproducibles utilizaremos el parámetro **random\_state**.

**Cross Validation**



Separamos en n partes nuestros datos, y se hace n evaluaciones, primero tomamos la primera separación de los datos luego tomamos todas las separaciones y entrenamos en el espacio restante evaluando en la que se escogió, en una segunda iteración evaluamos en ella y entrenamos los restantes, tendríamos **5 scores distintos**, al tener varios score y promediar podemos llegar a un mejor score.

Score : -0.0003714434028563307

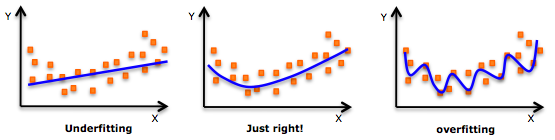
**Selección de modelos**

**Ajustando Modelos de Machine Learning, Underfitting y Overfitting**

**Underfitting** : Modelo usado muy simple, no logra capturar el patrón que siguen los datos. Se equivoca con los datos de entrenamiento y con los datos de test.

**Overfitting :** El error de entrenamiento va a ser bajo, pero el error de test es muy alto. Modelo muy ajustado a los datos de entrenamiento. Modelo muy complejo.

**Lo que buscamos:** Queremos un error de entrenamiento bajo, un error de test bajo y la diferencia entre los dos errores también debería ser bajo. Se busca un polinomio no tan basico ni tan complejo.



Para saber si estamos en overfiting y underfitting necesitamos los scores de entrenamiento y test,

Otros sinónimos son los sesgos y la varianza.

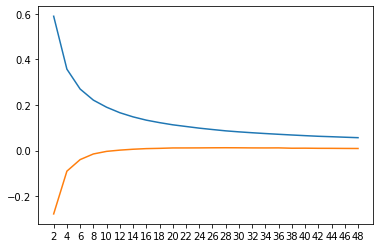
* Un sesgo alto esta ligador al underfitting
* Varianza alta está ligada al overfiting

Varianza = -0.2980530722823407

K-Nearest Neighbors

Con los algoritmos de vecino mas cercano tratamos de buscar algoritmos más complejos

**Curva de validación:**



Elegimos el mejor parámetro del score de test que siria 8 vecinos, como su valor máximo, estamos en una buena situación logrando mejorar nuestro algoritmo, el score ha mejorado y la varianza también.

Score: 0.14051686203221128

**Tratamos de mejorar nuestro modelo mediante una curva de aprendizaje**

**Curva de aprendizaje:**

El algoritmo se entrenó con estos valores

array ([ 2837, 9222, 15607, 21992, 28377])

Lo que hace la curva es evaluar con mínimos datos y con máximos datos que tenemos, si el score mejora esto nos dice que el algoritmo realmente está aprendiendo.

Imagen que contiene mapa

Descripción generada automáticamente

Imagen que contiene texto, mapa

Descripción generada automáticamente

El modelo Neighbors está aprendiendo va creciendo, pero a un no ha terminado de aprender

para que termine de aprender debemos agregarle más datos.

**Aprender:** Cuando mejora el score en función de la cantidad de datos

**Como solucionar el overfiting y el underfitting**

Varianza Alta:

* Conseguir más ejemplos
* Reducir cantidad de features
* Aumentar coeficiente de regularización

Bias Alto:

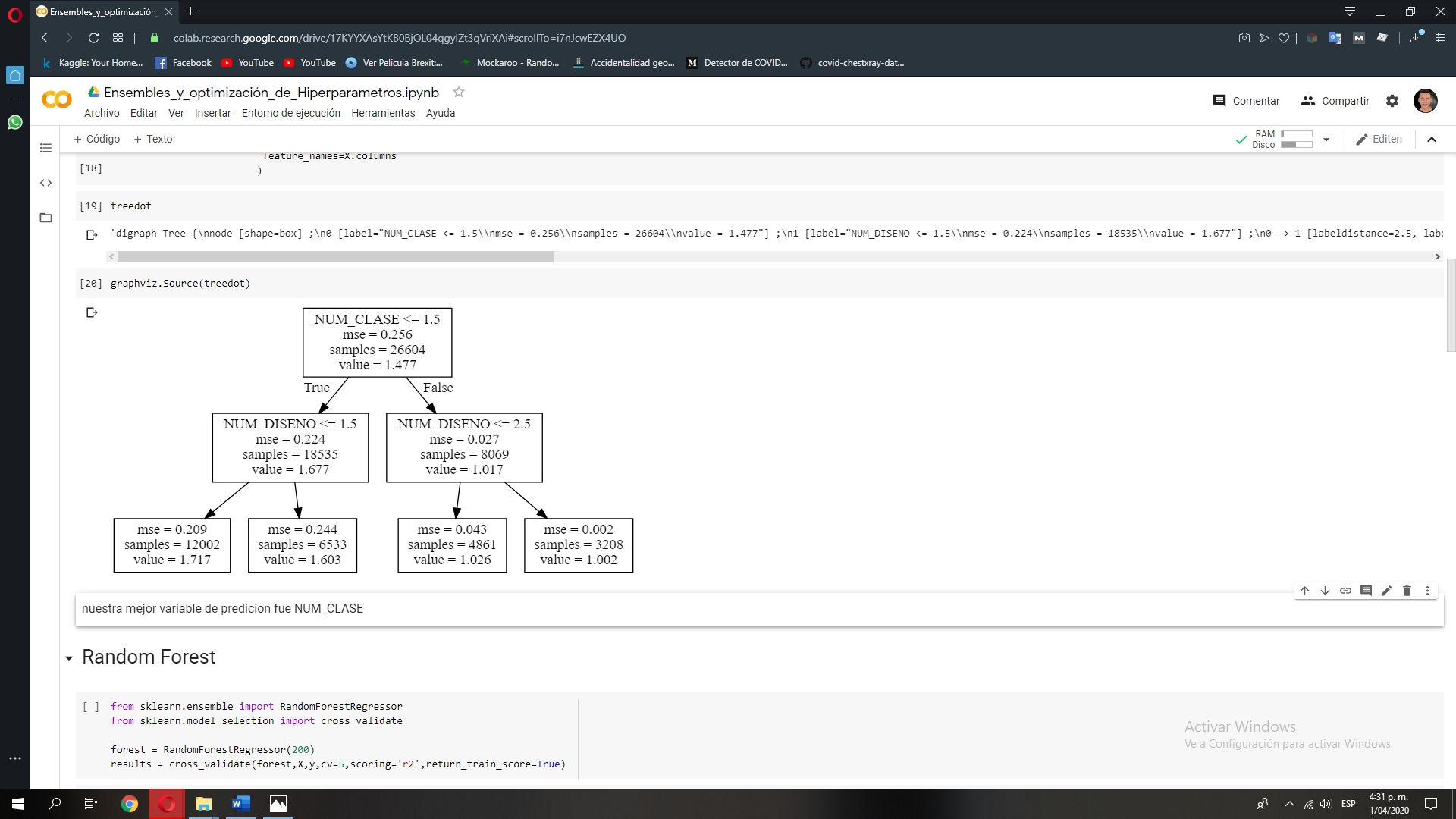
* Más features
* Modelo más complejo

Mal resultado general:

* Probar otro algoritmo/familia de modelos, quizás las hipótesis del modelo no son cumplidad por tu dataset.

**Modelo Decision Trees**

**Max\_depth = 2**



nuestra mejor variable de predicción fue **NUM\_CLASE**, separando dentro de esta feature encontramos a **NUM\_DISEÑO** como feature de buen resultado.

Virtudes de los árboles de decisión:

* Método poderoso y probado
* Interpretable
* No necesita escalar los datos (clasificación), y menos preprocesamiento de variables

Sin embargo, en la práctica existen modelo que obtienen mejor rendimiento. ¿cómo mejorar el modelo de árboles de decisión?

**Ensembles**

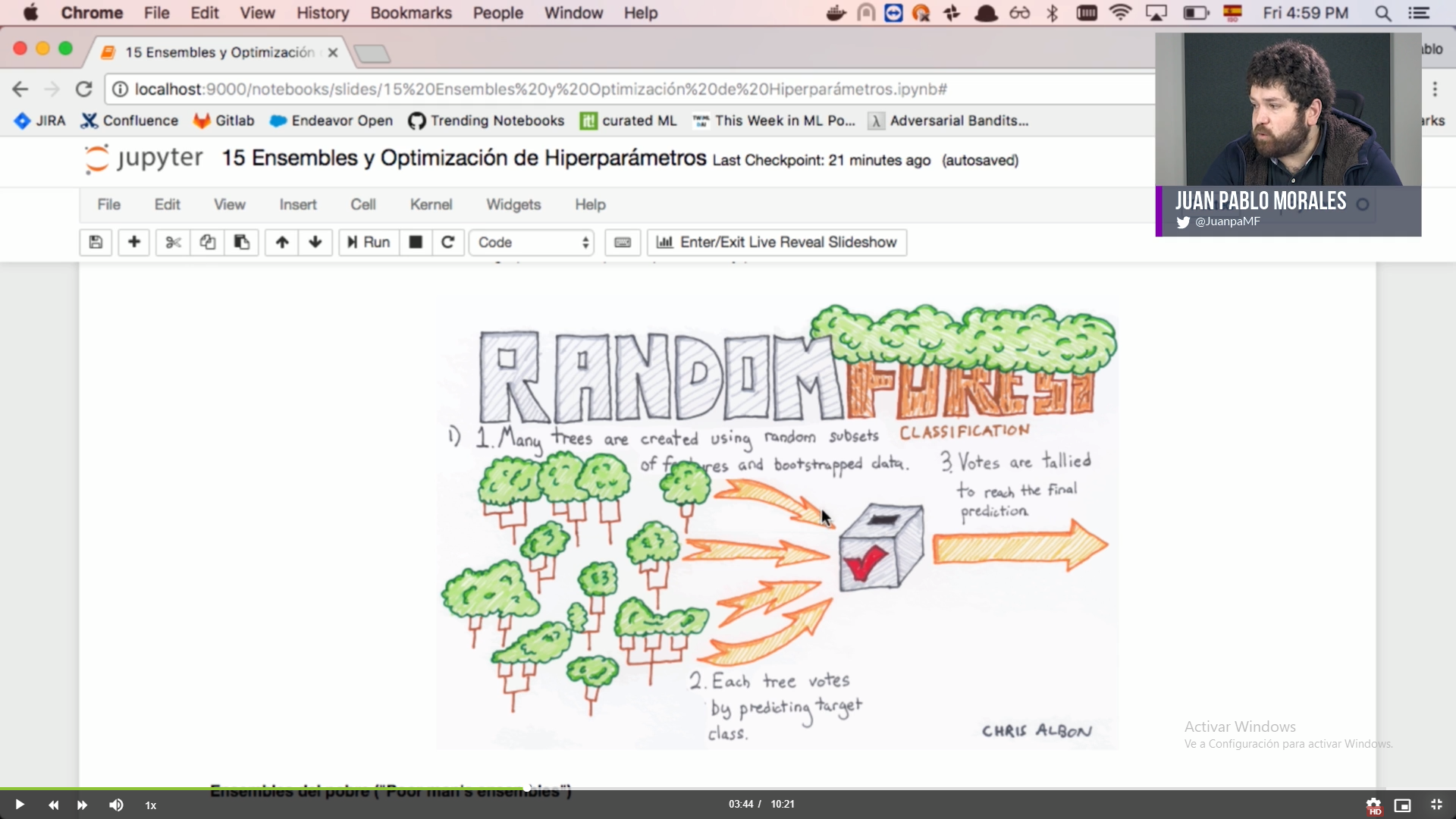
Concepto General

**Random Forest y Gradient Boosted Trees**, forma parte de una familia de algoritmos que se denominan ensembles

Ensemble = Submodelos -> Entrenamiento -> Predicciones -> Voto -> Predicción final

**Como funciona el algoritmo Random Forest**

Vamos a generar cientos de modelos de árboles de decisión que serán entrenados sobre conjuntos de datos bootstrapeados del conjunto de datos original y donde para cada etapa de separación el conjunto de features elegible será un subconjunto aleatorio original de features.



Score train: 0.9125251022459688

Score test: 0.36476038959389323

¡Mejor resultado que Lasso! Y ano tenemos Bias y tenemos un mejor score r2. Sin embargo, tenemos una diferencia importante entre score de entrenamiento y test(overfit)

Una de las consecuencias es que no lo estamos regularizando bien, no estamos controlando bien sus parámetros. Estamos en Overfiting

**Gradient Boosting Trees**

Score train: 0.9126762301257099

Score test: 0.36461608920634064

**Optimización de hiperparámetros**

Este método de Scikit learn hace lo siguiente:

* Fijar unlearning rate alto
* Fijar parámetros de los arboles
* Fijados estos parámetros, elegir el mejor numero de estimadores que conforman el ensemble
* (Tarea) Con el learning rate dado y el numero de estimadores óptimo, optimizar los parámetros de los arboles

**Grid Search**

Por ahora dijimos que:

* **Train\_test\_split** servía para evaluaciones rápidas, testeos y prototipaje
* **Cros\_validation** es un método mas robusto para poder estimar el rendimiento de nuestro algoritmo.

Sin embargo, una vez hemos finalizado nuestra etapa de prototipaje y ya queremos establecer un modelo definitivo deberíamos seguir el siguiente flujo.

Tratamos de optimizar los parámetros y como resultado tenemos el siguiente score:

Imagen que contiene mapa

Descripción generada automáticamente

Buenas practicas del Gradient Boosted Trees, siguiendo el modelo

Score: 0.396563358567635

Hemos mejorado nuestro score con respecto a todo lo que se ha desarrollado, controlando

El overfiting

Estimador Final: 0.4008068257681944

en Score r2 gracias al haber trabajado sobre las features y tomar varios modelos y finalmente con gsearch1 al haber optimizado los parametros de nuestro modelo