OBS Business School

RED DE EDUCACIÓN SUPERIOR
Planeta Formación y Universidades

Moisés Cantón Jara

Fundamentos de Matemáticas

Modelos de ML e Introducción a la Optimización Avanzada

Modelos de ML e Introducción a la Optimización Avanzada

1. Modelos de Regresión[i]

Regresión Lineal

La regresión lineal intenta modelar la relación entre una variable dependiente y y una o más variables dependientes x mediante una función lineal. Suponiendo una sola variable dependiente, su ecuación se simplificaría tal que:

$$y = \beta_0 + \beta_1 x$$

Una vez definida la ecuación, la forma estándar de encontrar los coeficientes β es minimizando la suma de los cuadrados de las diferencias entre las predicciones y las observaciones reales:

$$J(\beta) = \sum_{i=1}^{m} \left(y_i - (\beta_0 + \beta_1 x_i)\right)^2$$

python:

```
from sklearn.linear_model import LinearRegression

# Datos de ejemplo
X = [[1], [2], [3], [4]]
y = [2, 4, 5.8, 7.9]

# Crear y entrenar el modelo
model = LinearRegression().fit(X, y)

# Predecir
predictions = model.predict(X)
print(predictions)
```

Regresión Logística

La regresión logística modela la probabilidad de que una entrada pertenezca a una clase particular, siendo ideal para problemas de clasificación binaria. La función logística (o sigmoide) para una sola variable:

$$p(x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}}$$

La salida es una probabilidad que traduce a una forma binaria en función de un umbral, comúnmente 0.5

En este caso, la función de coste utilizada en la regresión logística es la log-loss siendo:

$$J(\beta) = -\frac{1}{m}\sum_{i=1}^{m}\left(y_i \log\left(p(x_i)\right) + (1 - y_i)\log(1 - p(x_i))\right)$$

python:
```
from sklearn.linear_model import LogisticRegression

X = [[1], [2], [3], [4]]
y = [0, 0, 1, 1]
model = LogisticRegression().fit(X, y)
predictions = model.predict(X)
print(predictions)
```

Correlación (Correlation en inglés)

La correlación es una medida que se emplea para evaluar la relación lineal entre dos conjuntos de variables o datos. Es especialmente útil para comprender como dos variables están relacionadas entre sí y si existe alguna dependencia lineal entre ellas.

En Machine Learning, la correlación se calcula generalmente utilizando el coeficiente de correlación de Pearson. Este se denota como r, mide la fuerza y la dirección de la relación lineal de dos variables continuas. Su fórmula es la siguiente:

$$r = \frac{\sum_{i=1}^{n}(x_i - \hat{x})(y_i - \hat{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \hat{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \hat{y})^2}}$$

donde:

- x_i y y_i son los valores individuales de las dos variables
- $\hat{x}$ y $\hat{x}$ son las medias (promedios) de las dos variables
- el denominador es el producto de las desviaciones estándar de ambas variables

y varía entre entre -1 y 1:

- Si $r = 1$, indica una correlación positiva perfecta, lo que significa que a medida que una variable aumenta, la otra también aumenta en una relación lineal perfecta
- Si $r = -1$, indica una correlación negativa perfecta, lo que significa que a medida que una variable aumenta, la otra disminuye en una relación lineal perfecta
- Si $r = 0$, indica que no hay correlación lineal entre las dos variables

En el contexto del aprendizaje automático, la correlación se utiliza en varios escenarios:

1. **Selección de características**: Puede utilizarse para identificar características altamente correlacionadas y eliminar una de ellas para reducir la redundancia y mejorar la eficiencia computacional.
2. **Análisis exploratorio de datos**: Ayuda a comprender la relación entre diferentes variables y cómo influyen en el objetivo del problema.
3. **Detección de multicolinealidad**: Permite identificar si varias variables predictoras están altamente correlacionadas, lo que podría afectar la estabilidad de algunos algoritmos de aprendizaje automático.

Además, es importante destacar que la correlación solo mide relaciones lineales. Si la relación entre dos variables es no lineal, la correlación de Pearson puede no capturarla adecuadamente, y otras medidas de similitud o dependencia pueden ser más apropiadas.

Regularización

Una vez analizados los métodos anteriores, podemos encontrarnos casos donde el modelo se ajuste demasiado bien a los datos de entrenamiento tanto que capture el ruido o fluctuaciones aleatorias presentes en esos datos. Este sobreajuste se combate añadiendo un término de penalización a la función de coste original, lo que impone una penalización a los coeficientes forzándolos a ser pequeños, y en el caso de L1, incluso puede llegar a que alguno de los coeficientes sean 0.

A. Regularización L1 (Lasso)

Esta regularización agrega una penalización equivalente al valor absoluto del coeficiente multiplicado por el parámetro de regularización λ:

$$J_{L1}(\beta) = J(\beta) + \lambda \sum_{i=1}^{n} |\beta_i|$$

lo que puede resultar en coeficientes que son exactamente cero, lo que significa que ciertas características son completamente ignoradas por el modelo.

B. Regularización L2 (Ridge)

La regularización L2, en cambio, agrega una penalización equivalente al cuadrado del coeficiente:

$$J_{L2}(\beta) = J(\beta) + \lambda \sum_{i=1}^{n} \beta_i^2$$

A diferencia de la regularización de Lasso, esta regularización tiende a reducir todos los coeficientes por igual, pero no necesariamente hace que ninguno de ellos sea cero.

python:

```
from sklearn.linear_model import Lasso, Ridge, LogisticRegression

# Datos de ejemplo
X = [[1], [2], [3], [4]]
y = [2, 4, 5.8, 7.9]

# L1 Regularización para Regresión Lineal (Lasso)
model_lasso = Lasso(alpha=0.1).fit(X, y)

# L2 Regularización para Regresión Lineal (Ridge)
model_ridge = Ridge(alpha=0.1).fit(X, y)

# L1 Regularización para Regresión Logística
model_logistic_L1 = LogisticRegression(penalty='l1', solver='saga').fit(X, y)

# L2 Regularización para Regresión Logística
model_logistic_L2 = LogisticRegression(penalty='l2').fit(X, y)
```

Ejemplo práctico

Un ejemplo comúnmente empleado para ilustrar las capacidades de los modelos de regression, especialmente, del de regresión lineal, es la predicción de precios de viviendas, donde se toman características de una casa, como el tamaño, el número de habitaciones, la ubicación, entre otras, para predecir su precio.

Considerando un conjunto de datos ficticios donde contamos con el tamaño de la vivienda (en metros cuadrados) y el número de habitaciones como características para predecir el precio tendríamos:

python:

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# Datos de ejemplo
X = np.array([
    [70, 2],   # 70m^2, 2 habitaciones
    [80, 3],   # 80m^2, 3 habitaciones
    [120, 4],  # 120m^2, 4 habitaciones
    [150, 4]   # 150m^2, 4 habitaciones
])
y = np.array([200000, 250000, 400000, 450000])  # Precios

# Dividir el conjunto en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Crear y entrenar el modelo
model = LinearRegression().fit(X_train, y_train)

# Predecir los precios para el conjunto de prueba
predictions = model.predict(X_test)
print(predictions)
```

2. Modelos basados en Árboles[ii]

Árboles de Decisión (Decision Trees)[iii]

Los árboles de decisión son estructuras jerárquicas y recursivas que dividen el espacio de características en regiones basándose en la pureza de los datos que se suele medir a partir de las siguientes variables:

- Índice de Gini que mide la impureza de un conjunto de datos siendo 0 la pureza perfecta (todos los elementos de una única clase)

$$Gini(S) = 1 - \sum_{i=1}^{c} p_i^2$$

donde p_i es la proporción de elementos en S que pertenecen a la clase C_i.

- Entropía que mide el desorden en el conjunto de datos

- Ganancia de Información que se información se calcula como la diferencia entre la entropía del conjunto original y la suma ponderada de las entropías de los subconjuntos resultantes después de realizar una división basada en un atributo

$$IG(S, A) = H(S) - \sum_{v \in Valores(A)} \frac{|S_v|}{|S|} H(S_v)$$

donde:

- A es el atributo sobre el que se está considerando dividir.
- Valores(A) son todos los posibles valores que A puede tener.
- S_v es el subconjunto de S donde el atributo A tiene el valor v.

El atributo que proporciona la mayor ganancia de información es típicamente el elegido para realizar la división en un nodo particular del árbol de decisión.

Aunque la ganancia de información puede ser una métrica útil, tiene una inclinación hacia atributos con un gran número de valores distintos (puede favorecer indebidamente a tales atributos). Por esta razón, en la práctica, a menudo se utilizan otras métricas, como el índice Gini o la razón de ganancia, para decidir sobre las divisiones en árboles de decisión.

Además, a medida que se agregan más y más decisiones al árbol, existe un riesgo creciente de sobreajuste, por lo que una solución es podar el árbol, es decir, eliminar algunas ramas del árbol para reducir su profundidad y complejidad.

Random Forests[iv]

Random Forest es una extensión de los árboles de decisión donde se construye un "bosque" de árboles de decisión utilizando un subconjunto de los datos y características, seleccionados aleatoriamente, para cada uno de ellos.

A la hora de realizar una predicción, todos los árboles "votan" y la clase (o valor, en regresión) que obtenga la mayoría de los votos es la predicción final del bosque.

Random Forests: Bootstrap Aggregating (Bagging)

El bagging es la técnica que permite a Random Forests mejorar la precisión de los árboles de decisión individuales. Bagging es la abreviatura de "Bootstrap Aggregating". Aquí está la matemática detrás de él:

- **Bootstrap**: Supongamos que tenemos un conjunto de datos D con N puntos. El bootstrap implica tomar N muestras con reemplazo de D, creando así un nuevo conjunto de datos D′. Debido al reemplazo, D′ puede tener puntos duplicados y omitir otros puntos de D.
- **Agregación**: Una vez que hemos construido múltiples árboles usando conjuntos de datos bootstrap, tomamos la media (para regresión) o la moda (para clasificación) de las predicciones de todos los árboles para llegar a la predicción final.

python:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris

# Cargar el conjunto de datos
data = load_iris()
X = data.data
y = data.target
```

```
# Crear el modelo Random Forest
clf = RandomForestClassifier(n_estimators=100)  # 100 árboles en el bosque

# Entrenar el modelo
clf.fit(X, y)

# Hacer predicciones
predictions = clf.predict(X)
```

Ejemplo práctico: Clasificación usando Árboles de Decisión

Tal y como hemos comentado previamente, los árboles de decisión son herramientas poderosas debido a su naturaleza intuitiva y su capacidad para ser visualizados, siendo muy útiles cuando deseamos entender las decisiones que el modelo está tomando.

A continuación, se describen los pasos a tomar para resolver un problema de este tipo:

1. Elección del conjunto de datos (en nuestro caso, vamos a escoger el conjunto iris).
2. Preprocesamiento, donde se deben eliminar datos nulos, la conversión de características categóricas en numéricas y la división del conjunto de datos en conjuntos de entrenamiento y prueba.
3. Entrenamiento del modelo. Utilizamos un algoritmo de árbol de decisión para entrenar el modelo en el conjunto de entrenamiento escogiendo este último automáticamente las mejores características y puntos de división para clasificar los datos.
4. Evaluación, donde una vez entrenado el modelo, se evalúa su precisión en el conjunto de prueba.
5. Visualización. Los árboles de decisión tienen la ventaja de ser visualmente interpretables, lo que permite una fácil comprensión de las decisiones tomadas por el modelo.

python:

```
from sklearn import datasets
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# Cargar el conjunto de datos
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Dividir en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Entrenar el árbol de decisión
clf = DecisionTreeClassifier()
clf.fit(X_train, y_train)

# Evaluar precisión
accuracy = clf.score(X_test, y_test)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Visualizar el árbol
```

```
plt.figure(figsize=(20,10))
plot_tree(clf, filled=True, feature_names=iris.feature_names, class_names=iris.target_names, rounded=True)
plt.show()
```

6. Modelos de Ensamble y Máquinas de Soporte Vectorial (SVM)

Modelos de Ensamble

Modelos de Ensamble hace referencia a un paradigma donde se combinan múltiples modelos para resolver un problema en particular con el objetivo de mejorar el rendimiento, la precisión y robustez del modelo.

A continuación, se destacan algunos de los métodos de ensamble más populares:

- Bagging (Bootstrap Aggregating) del que ya hablamos en el apartado de Random Forests y donde se toman múltiples subconjuntos aleatorios del conjunto de datos y se entrena un modelo en cada subconjunto.
- Boosting, donde a través de un enfoque iterativo cada modelo trata priorizar las instancias mal clasificadas en los modelos anteriores con el objetivo de corregir los errores cometidos, y algunos de sus algoritmos más destacados son: AdaBoost, Gradient Boosting Machines (GBM), y XGBoost.
- Stacking, donde se entrenan varios modelos de diferentes tipos (por ejemplo, regresión logística, arboles de decisión, SVMs, …) para generar la entrada de un metamodelo que aprende a combinar estas predicciones para generar la predicción final.

Máquinas de Soporte Vectorial

Las Maquinas de Soporte Vectorial (SVM, por sus siglas en inglés) es un conjunto de algoritmos de aprendizaje supervisado, especialmente conocido por su capacidad en tareas de clasificación binaria, utilizados para clasificación y regresión.

Supongamos, por simplicidad, un conjunto de puntos rojos y azules representados en un plano (2 dimensiones). SVM intentara dibujar la línea que mejor separe los puntos azules de los rojos. A esta línea se la conoce en mayores dimensiones como hiperplano.

Matemáticamente, el modelo primero intenta el concepto conocido como margen que representa la distancia mínima entre esta línea y sus puntos más cercanos ya sean azules o rojos, y luego intenta encontrar aquella línea donde se cumpla que el margen sea el mayor psible.

python:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC

# Generar datos aleatorios para ilustrar el problema
np.random.seed(0)
X_azul = np.random.randn(50, 2) + [2, 2]  # Centramos puntos azules alrededor de (2,2)
X_rojo = np.random.randn(50, 2) + [-2, -2]  # Centramos puntos rojos alrededor de (-2,-2)

X = np.vstack([X_azul, X_rojo])
y = np.array([1] * 50 + [-1] * 50)  # Etiquetas: 1 para azul, -1 para rojo
```

```
# Entrenar SVM
clf = SVC(kernel='linear')  # Utilizamos un kernel lineal
clf.fit(X, y)

# Función para visualizar el resultado
def plot_svm_decision_boundary(clf, xmin, xmax):
    w = clf.coef_[0]
    b = clf.intercept_[0]
    # Línea de decisión: -w[0] * x - b = 0 => x = -w[0]/w[1] * x - b/w[1]
    x = np.linspace(xmin, xmax)
    y_decision = -w[0]/w[1] * x - b/w[1]
    plt.plot(x, y_decision, "k-", label="Línea de decisión")
    plt.legend()

# Visualizar datos y líneas de decisión
plt.scatter(X_azul[:, 0], X_azul[:, 1], color='blue', label="Azul")
plt.scatter(X_rojo[:, 0], X_rojo[:, 1], color='red', label="Rojo")
plot_svm_decision_boundary(clf, -5, 5)
plt.xlabel("X1")
plt.ylabel("X2")
plt.title("SVM para separar puntos azules y rojos")
plt.legend()
plt.show()
```

Una vez dicho esto, podríamos encontrar casos, donde nuestros puntos no se pudiesen separar con una línea recta, debido a que todos se encuentran mezclados, para lo cual las SVM, en conjunción, con el Algebra Lineal nos proporciona un truco que reside en mover estos puntos a un espacio donde si se puedan separar.

python:

```
import numpy as np
import matplotlib.pyplot as plt

# Datos originales
puntos_azules = np.array([2])
puntos_rojos = np.array([3])

# Transformación: (x, x^2)
puntos_azules_transformados = np.array([[2, 4]])
puntos_rojos_transformados = np.array([[3, 9]])

# Visualización
plt.figure(figsize=(10, 5))

# Gráfica 1: Espacio original
plt.subplot(1, 2, 1)
plt.scatter(puntos_azules, [0] * len(puntos_azules), color='blue', label="Azul")
plt.scatter(puntos_rojos, [0] * len(puntos_rojos), color='red', label="Rojo")
plt.title("Espacio Original")
plt.xlabel("X")
plt.ylim(-1, 10)
plt.grid(True)
plt.legend()
```

```
# Gráfica 2: Espacio transformado
plt.subplot(1, 2, 2)
plt.scatter(puntos_azules_transformados[:, 0], puntos_azules_transformados[:, 1], color='blue', label="Azul")
plt.scatter(puntos_rojos_transformados[:, 0], puntos_rojos_transformados[:, 1], color='red', label="Rojo")
plt.title("Espacio Transformado")
plt.xlabel("X")
plt.ylabel("Transformación (X^2)")
plt.grid(True)
plt.legend()

plt.tight_layout()
plt.show()
```

7. Optimización Avanzada[v]

Método de Newton-Raphson

A parte del método de descenso del gradiente otro método para localizar los máximos y mínimos de una función objetivo es el de Newton – Raphson que lo hace mediante la aproximación de la función por un polinomio de segundo grado y es útil cuando tratamos funciones suaves que se comportan bien, ya que puede converger mucho más rápido que el Descenso del Gradiente.

A diferencia de este último, donde se aproxima la función objetivo por una línea recta, Newton Raphson trata de aproximar la función objetivo empleado un polinomio de segundo orden (una parábola) para aproximar la función localmente alrededor del punto actual.

Al igual que hicimos para el Método de Descenso del Gradiente, si lo dividimos en etapas tendríamos:

1. **Inicialización**: Se elige un valor inicial θ_0.
2. **Cálculo de la Derivada Segunda**: Se calcula la derivada segunda de la función objetivo J''(θ). Esta derivada se conoce como la "segunda derivada" o "hessiano".
3. **Actualización de Parámetros**:

$$\theta_{n+1} = \theta_n - \frac{J'(\theta_n)}{J''(\theta_n)}$$

4. **Convergencia**: El proceso se repite hasta que se cumpla algún criterio de convergencia, como cuando el cambio en θ entre iteraciones es menor que un umbral predefinido o después de un número fijo de iteraciones.

Si ahora consideramos, por ejemplo, la función objetivo simple: $J(\theta) = \theta^2 - 4\theta + 4$, tendremos

1. **Inicialización**: Se elige el valor inicial $\theta_0 = 3$
2. **Cálculo de la Derivada Segunda**: Se calcula la derivada segunda de la función objetivo J''(θ).

La derivada primera de $J(\theta)$ es:

$$J'(\theta) = 2\theta - 4$$

La derivada segunda de J(θ) es:

$$J''(\theta) = 2$$

3. **Actualización de Parámetros**: Sustituyendo,

$$\theta_1 = \theta_0 - \frac{J'(\theta_0)}{J''(\theta_0)} = 3 - \frac{2 \cdot 3 - 4}{2} = 2$$

4. **Convergencia**: Si seguimos iterando hasta que se cumpla un criterio de convergencia, llegaremos a nuestro valor aproximado del mínimo. Sin embargo, dado que, con una función cuadrática simple, alcanzamos la solución óptima en un solo paso.

Implementando esto para la función $f(x) = x^2 - 2$ en Python nos quedaría:

python:

```
def newton_raphson(f, df, x0, tol=1e-5, max_iter=1000):
    """
    Approximate root of f using Newton-Raphson method.

    Parameters:
    - f: function for which to find the root
    - df: derivative of f
    - x0: initial guess for the root
    - tol: tolerance (stop when absolute value of f(x) < tol)
    - max_iter: maximum number of iterations

    Returns:
    - Approximate root of f
    """
    x = x0
    for i in range(max_iter):
        if abs(f(x)) < tol:
            return x
        x = x - f(x) / df(x)
    raise ValueError("Failed to converge after {} iterations".format(max_iter))


# Example usage:
if __name__ == "__main__":
    # Define a sample function and its derivative
    # f(x) = x^2 - 2, f'(x) = 2x
    f = lambda x: x**2 - 2
    df = lambda x: 2*x

    x0 = 1.0  # Initial guess
    root = newton_raphson(f, df, x0)

    print("Approximate root:", root)
    print("f(root):", f(root))
```

[i] James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). An introduction to statistical learning. New York: Springer.

[ii] Breiman, L., Friedman, J., Stone, C. J., & Olshen, R. A. (1984). Classification and regression trees. CRC press.
[iii] Quinlan, J. R. (1986). Induction of decision trees. Machine learning, 1(1), 81-106.
[iv] Breiman, L. (2001). Random forests. Machine learning, 45(1), 5-32.
[v] Nocedal, J., & Wright, S. (2006). Numerical optimization. Springer Science & Business Media.