Métodos de Conjuntos
(Bishop, Capítulo 14)

Miguel Palomino

Combinación de modelos

En lugar de utilizar un único modelo, a menudo se puede mejorar el rendimiento combinando múltiples modelos.

- Bagging. Se crean múltiples versiones del mismo modelo mediante muestreo con reemplazo y se promedian los resultados.
- Boosting (AdaBoost, XGBoost). Los modelos se entrenan secuencialmente, cada uno corrigiendo los errores del anterior, y sus predicciones se combinan para mejorar la precisión.
- Bosques aleatorios. Se utilizan múltiples árboles de decisión, cada uno entrenado con un subconjunto aleatorio de los datos y características, y las predicciones de todos los árboles se combinan.
- Mezclas condicionales. Se pueden interpretar como versiones probabilísticas de árboles de decisión.

Mixturas de modelos de regresión lineal

Consideramos K modelos de regresión lineal, con pesos $\mathbf{w}_k$.

$$y(\mathbf{x}, \mathbf{w}_k) = \sum_j w_{kj}\phi_k(\mathbf{x}) = \mathbf{w}_k^{\mathsf{T}}\boldsymbol{\phi}(\mathbf{x}),$$

donde $\mathbf{w}_k = (w_{k0}, \ldots, w_{kM})^{\mathsf{T}}$ y $\boldsymbol{\phi} = (\phi_0, \ldots, \phi_M)^{\mathsf{T}}$, y los transformamos en modelos probabilísticos añadiendo ruido con una precisión común β:

$$t = y(\mathbf{x}, \mathbf{w}_k) + \epsilon = \mathbf{w}_k^{\mathsf{T}}\boldsymbol{\phi}(\mathbf{x}) + \epsilon \qquad \epsilon \sim \mathcal{N}\left(\epsilon|0, \beta^{-1}\right)$$

Ahora, si denotamos por π_k los coeficientes de la mezcla, la mixtura se puede escribir como

$$p\left(t|\theta, \mathbf{x}\right) = \sum_{k=1}^{K} \pi_k \mathcal{N}\left(t\middle|\mathbf{w}_k^{\mathsf{T}}\boldsymbol{\phi}, \beta^{-1}\right),$$

donde θ representa todos los parámetros del modelo: $\mathbf{W} = \{\mathbf{w}_k\}$, $\boldsymbol{\pi} = \{\pi_k\}$ y β.

Aprendiendo los parámetros θ

Como en ocasiones anteriores, podemos recurrir al algoritmo EM. Para cada dato observado t_n introducimos variables latentes $z_{nk} \in \{0, 1\}$, de manera que todos los elementos $k = 1, \ldots, K$ son cero salvo por un único 1 que indica qué componente de la mezcla generó el dato.

Probabilidad conjunta

Para un dato observado t_1, teniendo en cuenta que todas las z_{1k} son cero salvo una, la probabilidad conjunta se puede expresar:

$$p\left(t_1, z_{11}, \ldots, z_{1k} | \theta\right) = \prod_{k=1}^{K} (\pi_k \mathcal{N}\left(t \middle| \mathbf{w}_k^{\mathsf{T}} \boldsymbol{\phi}, \beta^{-1}\right))^{z_{1k}}$$

A partir de aquí se puede obtener la log-verosimilitud de un conjunto $\{t_1, \ldots, t_n\}$ y derivar ecuaciones para los pasos E y M.

Sin embargo, en esta ocasión vamos a seguir otro camino.

Mezclas de expertos

Podemos incrementar la capacidad de los modelos anteriores permitiendo que los coeficientes de la mezcla también sean funciones de las variables de entrada, de manera que

$$p\left(t|\theta,\mathbf{x}\right)=\sum_{k=1}^{K}\pi_k(\mathbf{x})\mathcal{N}\left(t\middle|\mathbf{w}_k^{\mathsf{T}}\boldsymbol{\phi},\beta^{-1}\right),$$

Esto se conoce como un modelo de mezcla de expertos en el que los coeficientes de mezcla $\pi_k(\mathbf{x})$ se denominan funciones de control (*gating functions*).

Las funciones de control determinan qué componentes dominan en cada región y deben satisfacer $0 \leq \pi_k(\mathbf{x}) \leq 1$ y $\sum_k \pi_k(\mathbf{x}) = 1$. Cuando los expertos son lineales, el modelo se puede entrenar con el algoritmo EM.

A su vez, este modelo se puede generalizar considerando que las componentes también sean mezclas de expertos y dando lugar a las mezclas de expertos jerárquicas.

Lenguajes de programación probabilísticos

- Son un tipo de lenguaje de programación diseñado para facilitar la descripción y el razonamiento sobre modelos probabilísticos.
- Permiten a los programadores expresar la incertidumbre de manera explícita utilizando variables aleatorias y distribuciones de probabilidad.
- Proveen herramientas para realizar inferencia probabilística, es decir, calcular distribuciones posteriores de las variables en función de los datos observados. Por ejemplo, la inferencia variacional o métodos de Montecarlo basados en cadenas de Markov (MCMC).

Algunos ejemplos son Stan, PyMC3, Pyro/NumPyro, TensorFlow Probability.

NumPyro

NumPyro is a lightweight probabilistic programming library that provides a NumPy backend for Pyro. We rely on JAX for automatic differentiation and JIT compilation to GPU / CPU. NumPyro is under active development, so beware of brittleness, bugs, and changes to the API as the design evolves.

Modelos en NumPyro

Esta introducción a NumPyro será superficial: cubriremos lo necesario para ser capaces de implementar las mixturas de modelos de regresión lineal.

Modelos

Los modelos probabilísticos en NumPyro, como en Pyro, se especifican como funciones en Python `model(*args, **kwargs)` que generan datos observados a partir de variables latentes utilizando una serie de primitivas. Fundamentalmente:

- `numpyro.sample`: Para muestrear de una distribución.
- `numpyro.plate`: Constructor para secuencias de variables independientes.

Un ejemplo simple

Vamos a implementar un modelo de regresión lineal:

$$y = \theta * x + b + \epsilon, \qquad \text{donde } \epsilon \sim \mathcal{N}(\epsilon|0, 0.5)$$

Nos interesará determinar θ y b a partir de un conjunto de observaciones. (En este caso la varianza del ruido está fijada y es 0.5 pero se podría plantear un modelo más complejo con un parámetro σ^2 que también hubiera que encontrar.)

```
def linear_regression(X, y=None):
    # Parámetros para la regresión lineal.
    theta = sample('theta', dist.Normal(0, 2))
    b = sample('b', dist.Normal(0, 3))
```

Nótese que los parámetros se inicializan de forma aleatoria utilizando distribuciones que representan nuestras creencias previas (las distribuciones a priori).

Puesto que el modelo es

$$y = \theta * x + b + \epsilon, \qquad \text{donde } \epsilon \sim \mathcal{N}\left(\epsilon | 0, 0.5\right),$$

se puede expresar de manera equivalente como

$$p\left(y|x\right) = \mathcal{N}\left(y|\theta * x + b, 0.5\right)$$

De esta manera, nuestro modelo en NumPyro se completa así:

```
mean = theta * X + b
sigma = np.sqrt(0.5) # desviación típica
with numpyro.plate('data', X.shape[0]):
    y_obs = sample('y_obs', dist.Normal(mean, sigma), obs=y)

return y_obs
```

Tanto `plate` como `sample` usan nombres de los que se aprovecha NumPyro para separar las especificaciones de modelos y observaciones, y en los algoritmos de inferencia.

Un ejemplo simple: observaciones

`plate`

Conceptualmente, la instrucción `numpyro.plate` es equivalente a

```
result = np.empty(len(X))
for i in range(len(X)):
    result[i] = numpyro.sample(f"obs_{i}", dist.Normal(mean, sigma),
                               obs=y[i] if y is not None else None)
return result
```

La principal utilidad de `numpyro.plate` es como herramienta para vectorizar.

`sample`

Cuando el parámetro `obs` es `None`, simplemente muestrea de la distribución correspondiente y devuelve el valor obtenido.
Cuando `obs` toma un valor, devuelve ese valor. Sin embargo, los algoritmos de inferencia de NumPyro "ejecutan el programa hacia atrás" y se encargan de asignar valores matemáticamente consistentes a todas las instrucciones `sample` del modelo.

Inferencia

Dado un conjunto de datos $\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, podemos utilizarlos para calcular los valores de θ y m que maximizan la verosimilitud.

$$\theta, b = \underset{\theta,b}{\operatorname{argmax}}\ p\left(\mathcal{D}|\theta, b\right)$$

O podemos seguir un enfoque bayesiano y puesto que hemos asignado distribuciones previas a θ y m, calcular su distribución posterior:

$$p\left(\theta, b|\mathcal{D}\right) = \frac{p\left(\mathcal{D}\right) p\left(\mathcal{D}|\theta, b\right)}{p\left(\mathcal{D}\right)}$$

Habitualmente, el cálculo de $p\left(\mathcal{D}\right)$ exige calcular integrales que no son analíticas y que resultan computacionalmente intratables por lo que hay que recurrir a métodos de inferencia aproximados:

- Muestreo. Se trata de obtener muestras de la variable x de una distribución $p(x)$. Por ejemplo, los métodos de Montecarlo basados en cadenas de Markov (MCMC).
- Inferencia variacional. Se trata de aproximar $p(x)$ mediante otra distribución $q(x)$ con la que sea más sencillo trabajar.

Inferencia: MCMC

En la mayoría de las ocasiones la distribución posterior solo nos interesa para poder calcular esperanzas, por ejemplo para realizar predicciones. Esto es, dada una función $f(\mathbf{x})$, el problema que deseamos afrontar es el cálculo de $\langle f(\mathbf{x}) \rangle_{p(\mathbf{x})}$.

La idea es obtener un conjunto de muestras $\{\mathbf{x}_1, \ldots, \mathbf{x}_L\}$ obtenidas de manera independiente de la distribución $p(\mathbf{x})$, lo que permite aproximar la esperanza mediante

$$\hat{f} = \frac{1}{L} \sum_{l=1}^{L} f(\mathbf{x}_l).$$

Los métodos de Montecarlo nos permiten realizar ese muestreo. Sin embargo, las estrategias más sencillas tienen limitaciones importantes en espacios de alta dimensionalidad por lo que se desarrollaron los métodos de Montecarlo basados en cadenas de Markov para afrontarlas.

Inferencia en NumPyro

NumPyro, al igual que Pyro, implementa dos métodos de inferencia: métodos de Montecarlo basados en cadenas de Markov (MCMC) e inferencia variacional. A diferencia de Pyro, el método en el que se hace más hincapié es MCMC.

En este curso no estudiaremos ninguno de estos métodos y nos limitaremos a utilizar la implementación de MCMC disponible en NumPyro.

Un ejemplo simple: inferencia

Supongamos que tenemos nuestros datos observados en variables `X` e `y`. El código para realizar inferencia en NumPyro es:

```
nuts_kernel = NUTS(linear_regression)
mcmc = MCMC(nuts_kernel, num_samples=1000, num_warmup=500)
rng_key = random.PRNGKey(0)
mcmc.run(rng_key, X, y)
samples = mcmc.get_samples()
```

- NUTS es el algoritmo de MCMC más habitual en NumPyro.
- `num_warmup` indica las muestras de "calentamiento" (se rechazan) utilizadas para fijar hiperparámetros antes de empezar a muestrear.
- `get_samples` devuelve un diccionario con un array de tamaño `num_samples` para cada variable muestreada; en nuestro ejemplo, `theta` y `m`. Si quisiéramos valores concretos de los parámetros para realizar predicciones podríamos utilizar `samples["theta"].mean()` y `samples["b"].mean()`.