Implementar E xpectation -M axi mization Algorithm (EM) em Python a partir do zero

Modelos de mistura gaussiana não supervisionados e semissupervisionados (GMM)

Quando as empresas lançam um novo produto, geralmente desejam descobrir os clientes-alvo. Se eles tiverem dados sobre o histórico de compras e preferências de compras dos clientes, eles podem utilizá-los para prever quais tipos de clientes têm mais probabilidade de comprar o novo produto. Existem muitos modelos para resolver esse problema típico de aprendizado não supervisionado e o Gaussian Mixture Model (GMM) é um deles.

GMM e EM

GMMs são modelos probabilísticos que umasuponha que todos os pontos de dados sejam gerados a partir de uma mistura de várias distribuições gaussianas com parâmetros desconhecidos. Eles diferem do agrupamento k-means porque os GMMs incorporam informações sobre o centro

(média) e a variabilidade (variância) de cada agrupamento e fornecem probabilidades posteriores.

No exemplo mencionado anteriormente, temos 2 clusters: pessoas que gostam do produto e pessoas que não gostam. Se soubermos a qual cluster cada cliente pertence (os rótulos), podemos facilmente estimar os parâmetros (média e variância) dos clusters, ou se conhecermos os parâmetros de ambos os clusters, podemos prever os rótulos. Infelizmente, não conhecemos nenhum deles. Para resolver esse problema do ovo e da galinha, o Algoritmo de Maximização de Expectativa (EM) é útil.

EM é um algoritmo iterativo para encontrar a probabilidade máxima quando há variáveis latentes. O algoritmo itera entre a realização de uma etapa de expectativa (E), que cria uma heurística da distribuição posterior e a log-verossimilhança usando a estimativa atual para os parâmetros, e uma etapa de maximização (M), que calcula os parâmetros maximizando o log- probabilidade da etapa E. As estimativas de parâmetro da etapa M são então usadas na próxima etapa E. Nas seções a seguir, vamos nos aprofundar na matemática por trás do EM e implementá-la em Python a partir do zero.

Dedução Matemática

W define as variáveis conhecidas como x, e o rótulo desconhecido como y. Fazemos duas suposições: a distribuição a priori $p(y)$ é binomial e$p(x \mid y)$ em cada cluster é gaussiana.

$$\text{observed: } x, \text{ latent: } y$$

$$\begin{aligned} \text{prior: } & p(y;\phi) = \phi^{1\{y=1\}}(1-\phi)^{1\{y=0\}} \\ \text{evidence: } & p(x|y=0;\mu_0,\Sigma_0) = N(\mu_0,\Sigma_0) \\ & p(x|y=1;\mu_1,\Sigma_1) = N(\mu_1,\Sigma_1) \end{aligned}$$

$$\theta := \phi, \mu_0, \mu_1, \Sigma_0, \Sigma_1$$

Todos os parâmetros são inicializados aleatoriamente. Para simplificar, usamos θ para representar todos os parâmetros nas seguintes equações.

$$\theta := \phi, \mu_0, \mu_1, \Sigma_0, \Sigma_1$$

Na etapa de expectativa (E), calculamos as heurísticas dos posteriores. Nós os chamamos de heurísticas porque são calculados com parâmetros adivinhados θ.

E step:

$$\begin{aligned} &\text{for each } i \in \{1,\ldots,n\}\text{: set} \\ &\qquad Q(y^i=1|x^i) := p(y^i=1|x^i;\theta) \\ &\qquad\qquad = \frac{p(x^i|y^i=1;\theta)p(y^i=1;\theta)}{\sum_{y^i\in(0,1)} p(x^i|y^i;\theta)p(y^i;\theta)} \\ &\qquad Q(y^i=0|x^i) := p(y^i=0|x^i;\theta) \\ &\qquad\qquad = \frac{p(x^i|y^i=0;\theta)p(y^i=0;\theta)}{\sum_{y^i\in(0,1)} p(x^i|y^i;\theta)p(y^i;\theta)} \end{aligned}$$

Na etapa de maximização (M), encontramos os maximizadores da log-verossimilhança e os usamos para atualizar θ. Observe que a soma dentro do logaritmo na equação (3) torna a complexidade computacional NP-difícil. Para mover a soma do logaritmo, usamos a desigualdade de Jensen para encontrar o limite inferior da evidência (ELBO), que é restrito apenas quando Q (y | x) = P (y | x). Se você estiver interessado nos detalhes matemáticos da equação (3) à equação (5), este artigo tem uma explicação decente.

M step:

$$
\begin{aligned}
\theta &= \text{argmax}\,\ell(\theta) && (1)\\
&= \text{argmax} \sum_{i=1}^{n} \log p(x^i;\theta) && (2)\\
&= \text{argmax} \sum_{i=1}^{n} \log \sum_{y^i \in (0,1)} p(x^i, y^i;\theta) && (3)\\
&\text{(Using Jensen's inequality and ELBO)} && (4)\\
&\geq \text{argmax} \sum_{i=1}^{n} \sum_{y^i \in (0,1)} Q(y^i|x^i) \log p(x^i, y^i;\theta) && (5)\\
&= \text{argmax} \sum_{i=1}^{n} \mathrm{E}_{Q(y^i|x^i)} \log p(x^i|y^i;\theta) p(y^i;\theta) && (6)
\end{aligned}
$$

Felizmente, existem soluções de formato fechado para os maximizadores no GMM.

$$\phi = \frac{\sum_{i=1}^{n} Q(y^i = 1|x^i)}{n} \tag{7}$$

$$\mu_0 = \frac{\sum_{i=1}^{n} x^i Q(y^i = 0|x^i)}{\sum_{i=1}^{n} Q(y^i = 0|x^i)} \tag{8}$$

$$\mu_1 = \frac{\sum_{i=1}^{n} x^i Q(y^i = 1|x^i)}{\sum_{i=1}^{n} Q(y^i = 1|x^i)} \tag{9}$$

$$\Sigma_0 = \frac{\sum_{i=1}^{n} Q(y^i = 0|x^i)(x^i - \mu_0)(x^i - \mu_0)^T}{\sum_{i=1}^{n} Q(y^i = 0|x^i)} \tag{10}$$

$$\Sigma_1 = \frac{\sum_{i=1}^{n} Q(y^i = 1|x^i)(x^i - \mu_1)(x^i - \mu_1)^T}{\sum_{i=1}^{n} Q(y^i = 1|x^i)} \tag{11}$$

Usaremos esses parâmetros atualizados na próxima iteração da etapa E, obteremos as novas heurísticas e executaremos a etapa M. O que o algoritmo EM faz é repetir essas duas etapas até que o log da verossimilhança médio convirja.

Antes de entrar no código, vamos comparar as soluções dos parâmetros acima do EM com as estimativas diretas dos parâmetros quando os rótulos são conhecidos. Você achou que eles são muito semelhantes? Na verdade, a única diferença é que as soluções EM usam as heurísticas de *Q* posteriores, enquanto as estimativas diretas usam os rótulos verdadeiros.

$$\phi' = \frac{\sum_{i=1}^{m} 1\{y^i = 1\}}{n} \tag{12}$$

$$\mu_0' = \frac{\sum_{i=1}^{m} x^i 1\{y^i = 0\}}{\sum_{i=1}^{m} 1\{y^i = 0\}} \tag{13}$$

$$\mu_1' = \frac{\sum_{i=1}^{m} x^i 1\{y^i = 1\}}{\sum_{i=1}^{m} 1\{y^i = 1\}} \tag{14}$$

$$\Sigma_0' = \frac{\sum_{i=1}^{m} 1\{y^i = 0\}(x^i - \mu_0')(x^i - \mu_0')^T}{\sum_{i=1}^{m} 1\{y^i = 0\}} \tag{15}$$

$$\Sigma_1' = \frac{\sum_{i=1}^{m} 1\{y^i = 1\}(x^i - \mu_1')(x^i - \mu_1')^T}{\sum_{i=1}^{m} 1\{y^i = 1\}} \tag{16}$$

Implementação Python

Existem muitos pacotes, incluindo scikit-learn, que oferecem APIs de alto nível para treinar GMMs com EM. Nesta seção, demonstrarei como implementar o algoritmo do zero para resolver problemas não supervisionados e semissupervisionados. O código completo pode ser encontrado aqui .

1. GMM não supervisionado

Vamos ficar com o novo exemplo de produto. Usando os dados pessoais conhecidos, projetamos 2 recursos *x1, x2* representados por uma matriz *x* , e nosso objetivo é prever se cada cliente gostará do produto (y = 1) ou não (y = 0).

```
1  data_unlabeled = pd.read_csv("data/unlabeled.csv")
2  x_unlabeled = data_unlabeled[["x1", "x2"]].values
```

load_unlabled_data.py hosted with ❤ by GitHub   view raw

Primeiro, inicializamos todos os parâmetros desconhecidos. `get_random_psd()` garante que a inicialização aleatória das matrizes de covariância seja semi-definida positiva.

```python
def get_random_psd(n):
    x = np.random.normal(0, 1, size=(n, n))
    return np.dot(x, x.transpose())


def initialize_random_params():
    params = {'phi': np.random.uniform(0, 1),
              'mu0': np.random.normal(0, 1, size=(2,)),
              'mu1': np.random.normal(0, 1, size=(2,)),
              'sigma0': get_random_psd(2),
              'sigma1': get_random_psd(2)}
    return params
```

initialize_random_params.py hosted with ❤ by GitHub view raw

Em seguida, passamos os parâmetros inicializados `e_step()`e calculamos as heurísticas *Q (y = 1 | x) e Q (y = 0 | x)* para cada ponto de dados, bem como as verossimilhanças logarítmicas médias que maximizaremos na etapa M.

```python
def e_step(x, params):
    np.log([stats.multivariate_normal(params["mu0"], params["sigma0"]).pdf(x),
            stats.multivariate_normal(params["mu1"], params["sigma1"]).pdf(x)])
    log_p_y_x = np.log([1-params["phi"], params["phi"]])[np.newaxis, ...] + \
                np.log([stats.multivariate_normal(params["mu0"], params["sigma0"]).pdf(x)
            stats.multivariate_normal(params["mu1"], params["sigma1"]).pdf(x)]).T
    log_p_y_x_norm = logsumexp(log_p_y_x, axis=1)
    return log_p_y_x_norm, np.exp(log_p_y_x - log_p_y_x_norm[..., np.newaxis])
```

e_step.py hosted with ❤ by GitHub view raw

Em `m_step()`, os parâmetros são atualizados usando as soluções de forma fechada na equação (7) ~ (11). Observe que, se não houvesse soluções de forma fechada, precisaríamos resolver o problema de otimização usando a subida de gradiente e encontrar as estimativas dos parâmetros.

```python
def m_step(x, params):
    total_count = x.shape[0]
    _, heuristics = e_step(x, params)
    heuristic0 = heuristics[:, 0]
    heuristic1 = heuristics[:, 1]
    sum_heuristic1 = np.sum(heuristic1)
    sum_heuristic0 = np.sum(heuristic0)
    phi = (sum_heuristic1/total_count)
    mu0 = (heuristic0[..., np.newaxis].T.dot(x)/sum_heuristic0).flatten()
    mu1 = (heuristic1[..., np.newaxis].T.dot(x)/sum_heuristic1).flatten()
    diff0 = x - mu0
    sigma0 = diff0.T.dot(diff0 * heuristic0[..., np.newaxis]) / sum_heuristic0
    diff1 = x - mu1
    sigma1 = diff1.T.dot(diff1 * heuristic1[..., np.newaxis]) / sum_heuristic1
    params = {'phi': phi, 'mu0': mu0, 'mu1': mu1, 'sigma0': sigma0, 'sigma1': sigma1}
    return params
```

m_step.py hosted with ❤ by GitHub view raw

Agora podemos repetir a execução das duas etapas até que o log da verossimilhança médio convirja. `rum_em()`retorna os rótulos previstos, os posteriores e as verossimilhanças logarítmicas médias de todas as etapas de treinamento.

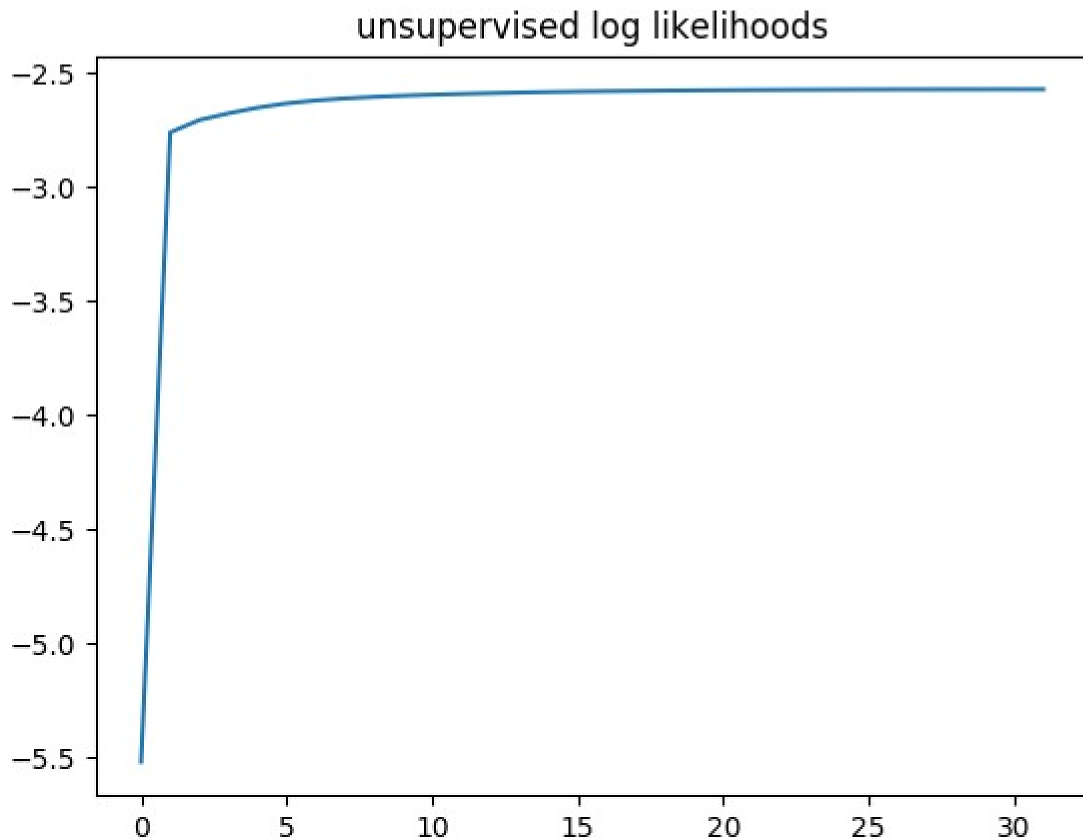
```
def get_avg_log_likelihood(x, params):
    loglikelihood, _ = e_step(x, params)
    return np.mean(loglikelihood)


def run_em(x, params):
    avg_loglikelihoods = []
    while True:
        avg_loglikelihood = get_avg_log_likelihood(x, params)
        avg_loglikelihoods.append(avg_loglikelihood)
        if len(avg_loglikelihoods) > 2 and abs(avg_loglikelihoods[-1] - avg_loglikelihoo
            break
        params = m_step(x_unlabeled, params)
    print("\tphi: %s\n\tmu_0: %s\n\tmu_1: %s\n\tsigma_0: %s\n\tsigma_1: %s"
               % (params['phi'], params['mu0'], params['mu1'], params['sigma0'], params[
    _, posterior = e_step(x_unlabeled, params)
    forecasts = np.argmax(posterior, axis=1)
    return forecasts, posterior, avg_loglikelihoods
```

run_em.py hosted with ❤ by GitHub view raw

Executando o modelo não supervisionado, vemos o log de verossimilhanças médio convergido em mais de 30 etapas.

```
random_params = initialize_random_params()
unsupervised_forecastsforecasts, unsupervised_posterior, unsupervised_loglikelihoods = ru
print("total steps: ", len(unsupervised_loglikelihoods))
plt.plot(unsupervised_loglikelihoods)
plt.title("unsupervised log likelihoods")
plt.savefig("unsupervised.png")
plt.close()
```

unsupervised.py hosted with ❤ by GitHub view raw



2. GMM semissupervisionado

Em alguns casos, temos uma pequena quantidade de dados rotulados. Por exemplo, podemos saber as preferências de alguns clientes a partir de pesquisas. Considerando que a base de clientes em potencial é enorme, a quantidade de dados rotulados que temos é insuficiente para o aprendizado supervisionado completo, mas podemos aprender os parâmetros iniciais dos dados de uma forma semissupervisionada.

Usamos os mesmos dados não rotulados de antes, mas também temos alguns dados rotulados desta vez.

```python
data_labeled = pd.read_csv("data/labeled.csv")
x_labeled = data_labeled[["x1", "x2"]].values
y_labeled = data_labeled["y"].values
```

load_labeled_data.py hosted with ❤ by GitHub view raw

Em `learn_params()`, aprendemos os parâmetros iniciais dos dados rotulados implementando a equação (12) ~ (16). Esses parâmetros aprendidos são usados na primeira etapa E.

```python
def learn_params(x_labeled, y_labeled):
    n = x_labeled.shape[0]
    phi = x_labeled[y_labeled == 1].shape[0] / n
    mu0 = np.sum(x_labeled[y_labeled == 0], axis=0) / x_labeled[y_labeled == 0].shape[0]
    mu1 = np.sum(x_labeled[y_labeled == 1], axis=0) / x_labeled[y_labeled == 1].shape[0]
    sigma0 = np.cov(x_labeled[y_labeled == 0].T, bias= True)
    sigma1 = np.cov(x_labeled[y_labeled == 1].T, bias=True)
    return {'phi': phi, 'mu0': mu0, 'mu1': mu1, 'sigma0': sigma0, 'sigma1': sigma1}
```

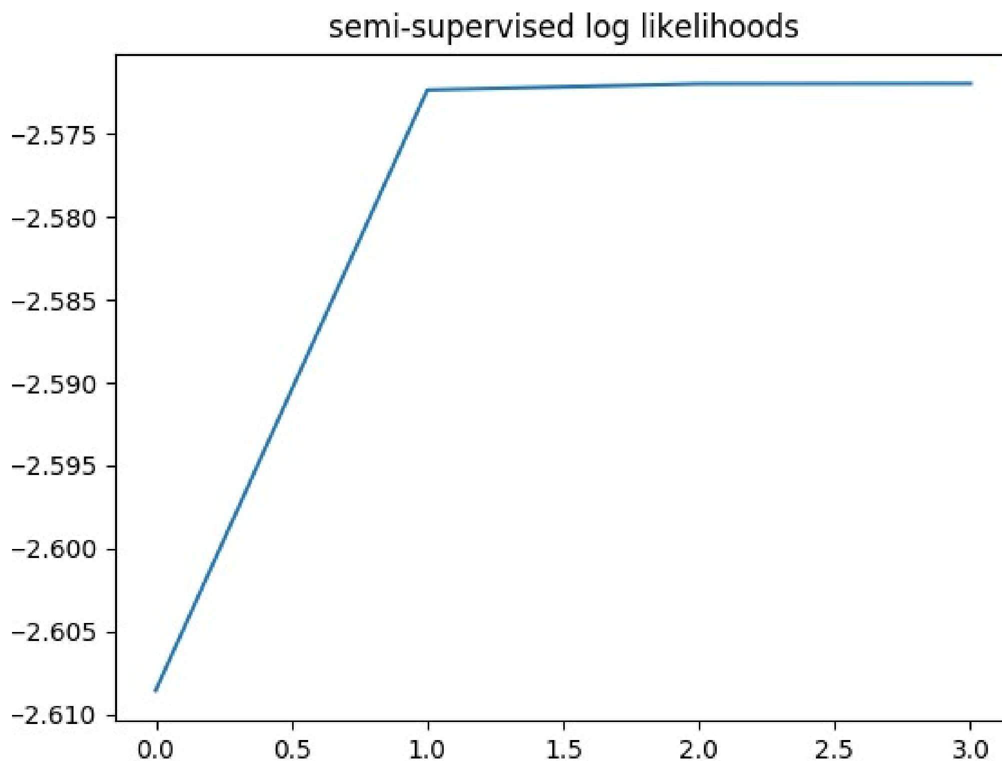
learn_params.py hosted with ❤ by GitHub view raw

Exceto os parâmetros iniciais, tudo o resto é o mesmo para que possamos reutilizar as funções definidas anteriormente. Vamos treinar o modelo e representar graficamente as verossimilhanças logarítmicas médias.

```python
learned_params = learn_params(x_labeled, y_labeled)
semisupervised_forecasts, semisupervised_posterior, semisupervised_loglikelihoods = run_e
print("total steps: ", len(semisupervised_loglikelihoods))
plt.plot(semisupervised_loglikelihoods)
plt.title("semi-supervised log likelihoods")
plt.savefig("semi-supervised.png")
```

semi-supervised.py hosted with ❤ by GitHub view raw

Desta vez, o log de verossimilhanças médio convergiu em 4 etapas, muito mais rápido do que o aprendizado não supervisionado.



Para verificar nossa implementação, comparamos nossas previsões com as previsões da API do scikit-learn. Para construir o modelo no scikit-learn, simplesmente chamamos a API GaussianMixture e ajustamos o modelo com nossos dados não rotulados. Não se esqueça de passar os parâmetros aprendidos ao modelo para que tenha a mesma inicialização que nossa implementação semissupervisionada. `GMM_sklearn()` retorna as previsões e posteriores do scikit-learn.

```python
def GMM_sklearn(x, weights=None, means=None, covariances=None):
    model = GaussianMixture(n_components=2,
                            covariance_type='full',
                            tol=0.01,
                            max_iter=1000,
                            weights_init=weights,
                            means_init=means,
                            precisions_init=covariances)
    model.fit(x)
    print("\nscikit learn:\n\tphi: %s\n\tmu_0: %s\n\tmu_1: %s\n\tsigma_0: %s\n\tsigma_1:
               % (model.weights_[1], model.means_[0, :], model.means_[1, :], model.covar
    return model.predict(x), model.predict_proba(x)[:,1]


learned_params = learn_params(x_labeled, y_labeled)
weights = [1 - learned_params["phi"], learned_params["phi"]]
means = [learned_params["mu0"], learned_params["mu1"]]
covariances = [learned_params["sigma0"], learned_params["sigma1"]]
sklearn_forecasts, posterior_sklearn = GMM_sklearn(x_unlabeled, weights, means, covarian
output_df = pd.DataFrame({'semisupervised_forecasts': semisupervised_forecasts,
                          'semisupervised_posterior': semisupervised_posterior[:, 1],
                          'sklearn_forecasts': sklearn_forecasts,
                          'posterior_sklearn': posterior_sklearn})
print("\n%s%% of forecasts matched." % (output_df[output_df["semisupervised_forecasts"]
```

compare_sklearn.py hosted with ❤ by GitHub view raw

Comparando os resultados, vemos que os parâmetros aprendidos de ambos os modelos são muito próximos e 99,4% das previsões coincidem. Caso você esteja curioso, a pequena diferença é causada principalmente pela regularização dos parâmetros e precisão numérica no cálculo da matriz.

```
semi-supervised:
    phi: 0.586349881794546
    mu_0: [-1.04546727 -1.02704636]
    mu_1: [0.98763329 0.99661118]
    sigma_0: [[0.36018609 0.30853357]
 [0.30853357 0.75384027]]
    sigma_1: [[0.7196797  0.1437903 ]
 [0.1437903  0.30853791]]
total steps:  4

scikit learn:
    phi: 0.59647894226803
    mu_0: [-1.06169376 -1.0563389 ]
    mu_1: [0.96408565 0.98206315]
    sigma_0: [[0.35027155 0.29629092]
 [0.29629092 0.73083581]]
    sigma_1: [[0.74510804 0.16156928]
 [0.16156928 0.32021029]]

99.4% of forecasts matched.
```

É isso! Acabamos de desmistificar o algoritmo EM.

Conclusões

Neste artigo, exploramos como treinar modelos de mistura gaussiana com o algoritmo de maximização da expectativa e o implementamos em Python para resolver problemas de aprendizagem não supervisionados e semissupervisionados. EM é um método muito útil para encontrar a probabilidade máxima quando o modelo depende de variáveis latentes e, portanto, é frequentemente usado no aprendizado de máquina.
Espero que você tenha se divertido lendo este artigo.

Siwei Causevic
Engenheiro de aprendizado de máquina no Google. https://www.linkedin.com/in/siwei-causevic/