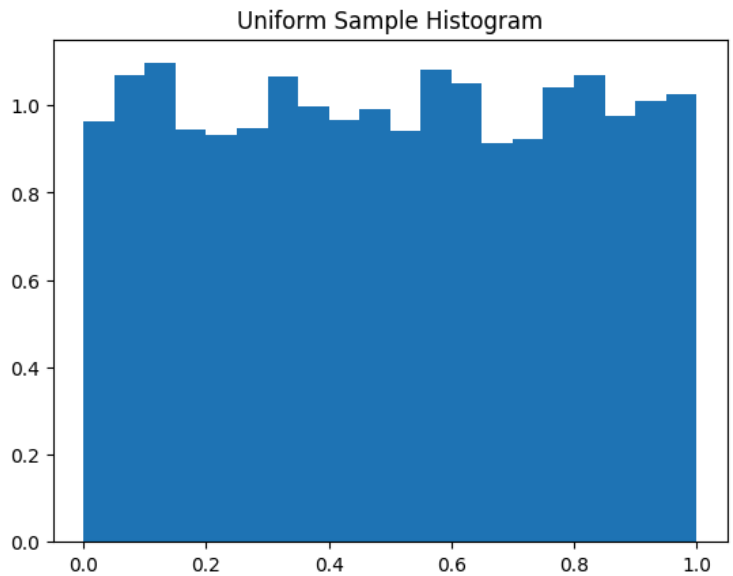
Topic models LDA, Sampling

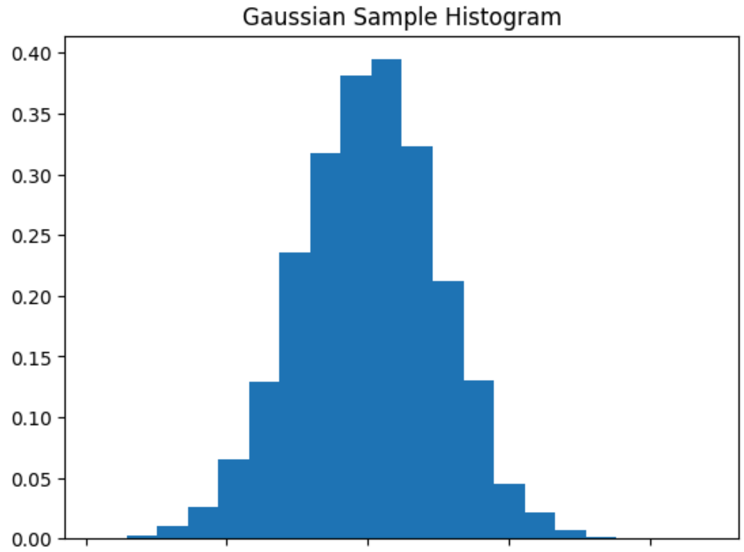
Yusuf Abdi

08/07/2023

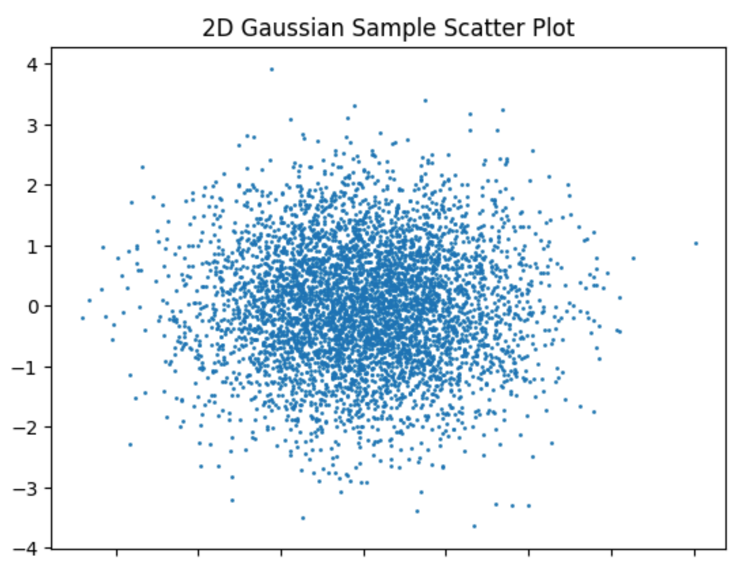
```
import random
import math
import matplotlib.pyplot as plt
from collections import defaultdict
import urllib.request
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer, ENGLISH_STOP_WORDS


def uniform_sample(minimum, maximum, sample_size):
    return [minimum + (maximum-minimum)*random.random() for i in range(sample_size)]


# uniform samples
uniform_samples = uniform_sample(0, 1, 10000)

# plot histogram
plt.hist(uniform_samples, bins=20, density=True)
plt.title("Uniform Sample Histogram")
plt.show()
```




```
"""
 Box-Muller transform, used for generating Gaussian-distributed random numbers, is essentially an application of inverse trans
 mapping from the uniform distribution to the normal distribution.
"""
def gaussian_sample(mu, sigma, sample_size):
    samples = []
    for s in range(sample_size):
        # generate random values
        u1 = random.random()
        u2 = random.random()
        # normally distributed numbers generated by the Box-Muller transform.
        z0 = math.sqrt(-2.0 * math.log(u1)) * math.cos(2 * math.pi * u2)
        sample = mu + z0 * sigma
        samples.append(sample)
    return samples


# Generate gaussian samples
gaussian_samples = gaussian_sample(0, 1, 10000)

plt.hist(gaussian_samples, bins=20, density=True)
plt.title("Gaussian Sample Histogram")
plt.show()
```




```
# for 2d sample we again use Box-Muller transfomration
def gaussian_2d_sample(mu, sigma, sample_size):
    samples = []
    for i in range(sample_size):
        # generate to random numbers
        u1 = random.random()
        u2 = random.random()
        # convert to normally distributed numbers
        z0 = math.sqrt(-2.0 * math.log(u1)) * math.cos(2 * math.pi * u2)
        z1 = math.sqrt(-2.0 * math.log(u1)) * math.sin(2 * math.pi * u2)
        sample_x = mu[0] + z0 * sigma[0]
        sample_y = mu[1] + z1 * sigma[1]
        samples.append([sample_x, sample_y])
    return samples


# generate 2D Gaussian samples
gaussian_2d_samples = gaussian_2d_sample([0, 0], [1, 1], 5000)

# split samples into X and Y coordinates
x_samples = [sample[0] for sample in gaussian_2d_samples]
y_samples = [sample[1] for sample in gaussian_2d_samples]

# plot the samples
plt.scatter(x_samples, y_samples, s=1)
plt.title("2D Gaussian Sample Scatter Plot")
plt.show()
```



```python
# group elements by their size
def group_by_size(pop):
    size_elements = {}
    for element, size in pop:
        if size not in size_elements:
            size_elements[size] = []
        size_elements[size].append(element)
    return size_elements

# sample without replacement
def sample_method(pop, sample_size):
    size_elements = group_by_size(pop)

    sizes = list(size_elements.keys())
    total_size = sum(sizes)
    probabilities = [size / total_size for size in sizes]

    samples = []
    while len(samples) < sample_size:
        index = random.choices(range(len(sizes)), probabilities)[0]
        size_group = size_elements[sizes[index]]

        if size_group:
            sample = random.choice(size_group)
            samples.append(sample)
            size_group.remove(sample)

        # If all elements in a size group are chosen, remove the group
        if not size_group:
            total_size -= sizes[index]
            del sizes[index]
            del probabilities[index]
            if sizes:
                probabilities = [size / total_size for size in sizes]
            else:
                break
    return samples


population = [(i, random.randint(1, 50)) for i in range(300)]
sample_size = 20
samples = sample_method(population, sample_size)
print(samples)
```

```
[148, 9, 38, 278, 44, 84, 202, 191, 138, 130, 66, 171, 166, 132, 100, 220, 29, 206, 293, 221]
```

```python
def gibbs_sample(mu_x, mu_y, sigma_x, sigma_y, rho, num_samples):
    samples = [(mu_x, mu_y)]

    for i in range(num_samples):
        current_x, current_y = samples[-1]

        # sample x given y
        mu_x_given_y = mu_x + rho * sigma_x/sigma_y * (current_y - mu_y)
        sigma_x_given_y = sigma_x * math.sqrt(1 - rho**2)
        x = random.gauss(mu_x_given_y, sigma_x_given_y)

        # sample y given x
        mu_y_given_x = mu_y + rho * sigma_y/sigma_x * (x - mu_x)
        sigma_y_given_x = sigma_y * math.sqrt(1 - rho**2)
        y = random.gauss(mu_y_given_x, sigma_y_given_x)

        samples.append((x, y))

    # first sample is just the initial value
    return samples[1:]



mu_x = 0
mu_y = 0
sigma_x = 1
sigma_y = 1
rho = 0.5
```

```python
num_samples = 1000
samples = gibbs_sample(mu_x, mu_y, sigma_x, sigma_y, rho, num_samples)

for sample in samples[:5]:
    print(sample)
```

```
(0.09528801656713631, -0.11767162939918066)
(0.11259989350485396, -0.13153808439436307)
(-0.18500328564128524, -0.840755116520412)
(1.3862330887995016, 0.11272593263615893)
(-0.3474779772483641, 0.10718054394342993)
```

```python
def sample_topic(topic_distribution):
    return random.choices(range(len(topic_distribution)), weights=topic_distribution)[0]



# LDA class for Latent Dirichlet Allocation
class LDA:
    def __init__(self, num_topics, alpha=0.1, beta=0.1):
        # initialize number of topics, dirichlet priors alpha and beta, and topic assignment list
        self.num_topics = num_topics
        self.alpha = alpha
        self.beta = beta
        self.topic_assignment = []

    def fit(self, docs, max_iter=100):
        # initialize counts and assignments
        self._initialize(docs)

        # iterate over all documents for a number of max iterations
        for k in range(max_iter):
            for doc_id in range(len(docs)):
                # for each word and its corresponding topic
                for i, (word, topic) in enumerate(zip(docs[doc_id], self.topic_assignment[doc_id])):
                    # decrease counts for this word-topic and doc-topic pair
                    self.counts_word_topic[word][topic] -= 1
                    self.counts_doc_topic[doc_id][topic] -= 1
                    self.counts_topic[topic] -= 1

                    # compute the conditional distribution of the current word
                    topic_distribution = self._conditional_distribution(doc_id, word)
                    # Sample a new topic for current word
                    topic = sample_topic(topic_distribution)

                    # increase counts
                    self.topic_assignment[doc_id][i] = topic
                    self.counts_word_topic[word][topic] += 1
                    self.counts_doc_topic[doc_id][topic] += 1
                    self.counts_topic[topic] += 1

    def _initialize(self, docs):
        # create a vocabulary and count the total number of unique words
        self.vocab = set(word for doc in docs for word in doc)
        self.vocab_size = len(self.vocab)

        # initialize count holders for word-topic, doc-topic, and topic
        self.counts_word_topic = defaultdict(lambda: defaultdict(int))
        self.counts_doc_topic = defaultdict(lambda: defaultdict(int))
        self.counts_topic = defaultdict(int)

        # for each document
        for doc_id in range(len(docs)):
            topics = []
            # for each word in the document
            for word in docs[doc_id]:
                # assign a random initial topic
                topic = random.randint(0, self.num_topics - 1)
                # list of topics for this document
                topics.append(topic)

                self.counts_word_topic[word][topic] += 1
                self.counts_doc_topic[doc_id][topic] += 1
                self.counts_topic[topic] += 1
            # add this document's topics to the list
```

```
            self.topic_assignment.append(topics)

    def _conditional_distribution(self, doc_id, word):
        # compute the conditional distribution of the current word
        topic_distribution = []
        # for each topic
        for topic in range(self.num_topics):
            # calculate probability of this topic given this doc and this word given this topic
            p_topic_given_doc = (self.counts_doc_topic[doc_id][topic] + self.alpha) / \
                                (sum(self.counts_doc_topic[doc_id].values()) + self.num_topics * self.alpha)
            p_word_given_topic = (self.counts_word_topic[word][topic] + self.beta) / \
                                 (self.counts_topic[topic] + self.vocab_size * self.beta)

            # multiply these probabilities and add to the distribution
            topic_distribution.append(p_topic_given_doc * p_word_given_topic)

        # normalize distribution so it sums to 1
        return [p / sum(topic_distribution) for p in topic_distribution]

    def print_topics(self, num_words=10):
        # print the top num_words per topic
        for topic in range(self.num_topics):
            print(f"Topic #{topic + 1}:")
            # get the words and their counts for this topic
            word_counts = [(word, counts.get(topic, 0)) for word, counts in self.counts_word_topic.items()]
            # sort by count
            word_counts.sort(key=lambda wc: wc[1], reverse=True)
            # print the top num_words words
            for word, count in word_counts[:num_words]:
                print(f"    {word} (count: {count})")



# Fetch sonnets dataset
url = "https://www.ccs.neu.edu/home/vip/teach/DMcourse/data/sonnetsPreprocessed.txt"
response = urllib.request.urlopen(url)
long_txt = response.read().decode()
lines = long_txt.split('\n')

# Preprocess data
docs = [line.split(' ') for line in lines]



# Run LDA
lda = LDA(num_topics=10)
lda.fit(docs, max_iter=50)
lda.print_topics(num_words=10)
```

```
    [illegible] (count: 11)
```

```
        nor (count: 11)
        live (count: 10)
        hours (count: 10)
        shalt (count: 9)
        death (count: 9)
        earth (count: 9)
    Topic #9:
        nor (count: 9)
        times (count: 8)
        none (count: 7)
        well (count: 7)
        way (count: 6)
        joy (count: 6)
        thought (count: 6)
        whilst (count: 6)
        reason (count: 6)
        desire (count: 5)
    Topic #10:
        love (count: 37)
        true (count: 20)
        why (count: 14)
        o (count: 14)
        might (count: 13)
        old (count: 13)
        though (count: 12)
        others (count: 12)
        say (count: 11)
        nothing (count: 11)
```

```
# fetch 20 newsgroups dataset
newsgroups = fetch_20newsgroups(subset='train', remove=('headers', 'footers', 'quotes'))

# preprocess data
vectorizer = CountVectorizer(lowercase=True, stop_words=list(ENGLISH_STOP_WORDS),
                             token_pattern = r'\b[a-zA-Z]{3,}\b') # words with >= 3 alpha chars
counts = vectorizer.fit_transform(newsgroups.data)
vocabulary = vectorizer.get_feature_names_out()

# convert matrix to list of documents
docs = []
for doc_id in range(counts.shape[0]):
    doc = [vocabulary[word_id] for word_id, count in zip(counts.indices[counts.indptr[doc_id]:counts.indptr[doc_id + 1]],
                                                         counts.data[counts.indptr[doc_id]:counts.indptr[doc_id + 1]])
           for _ in range(count)]
    docs.append(doc)
```

```
lda = LDA(num_topics=10)
lda.fit(docs, max_iter=50)
lda.print_topics(num_words=10)
```

```
    Topic #1:
        god (count: 1083)
        does (count: 561)
        believe (count: 425)
        church (count: 393)
        christian (count: 368)
        bible (count: 352)
        religion (count: 345)
        evidence (count: 337)
        say (count: 325)
        true (count: 321)
    Topic #2:
        chz (count: 246)
        dod (count: 173)
        air (count: 161)
        water (count: 136)
        new (count: 123)
        cover (count: 118)
        bike (count: 107)
        condition (count: 88)
        sale (count: 78)
        appears (count: 75)
    Topic #3:
        max (count: 4585)
        file (count: 1269)
        program (count: 724)
        window (count: 711)
        use (count: 693)
        windows (count: 685)
```

```
        files (count: 619)
        version (count: 463)
        server (count: 455)
        image (count: 442)
    Topic #4:
        armenian (count: 612)
        armenians (count: 511)
        turkish (count: 483)
        people (count: 438)
        said (count: 415)
        turkey (count: 271)
        went (count: 264)
        armenia (count: 214)
        came (count: 211)
        turks (count: 206)
    Topic #5:
        use (count: 1264)
        key (count: 1121)
        drive (count: 1010)
        like (count: 990)
        know (count: 879)
        does (count: 839)
        bit (count: 789)
        just (count: 773)
        card (count: 773)
        used (count: 754)
    Topic #6:
        people (count: 1887)
        don (count: 1866)
```

Start coding or generate with AI.