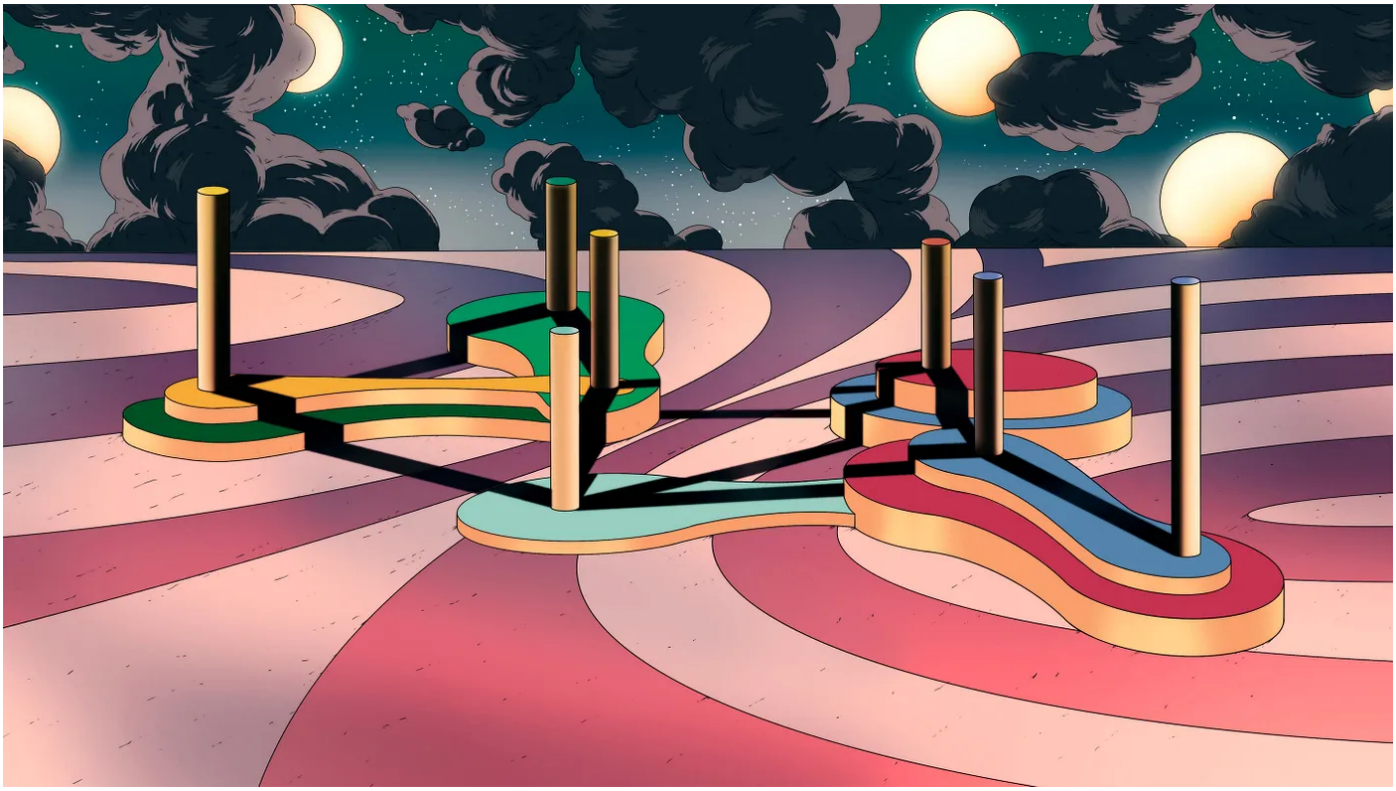
Machine Learning Coding Problems

Abhijit Mondal · Follow

10 min read · Dec 25, 2021



source: quanta magazine

In many ML interview rounds, candidates are asked to demonstrate their coding skills w.r.t. some of the common and basic ML problems. These kind of rounds helps to identify both coding skills as well as ML skills required to be a top notch ML engineer in some of the top companies.

Let's look at some of the common ML coding problems asked in such interviews:

Problem 1:

Given the **API** `rand7()` that generates a uniform random integer in the range `[1, 7]`, write a function `rand10()` that generates a uniform random integer in the range `[1, 10]`. You can only call the API `rand7()`, and you shouldn't call any other API. Please **do not** use a language's built-in random API.

Solution:

rand7()-1 generates random integer between 0 and 6.

> *To generate random integers between [1, 10], we need to generate 10 numbers which are equally probable or a range which is equally divisible into 10 parts.*

Let a=rand7()-1 and b= rand7(), then 7a+b will generate random integers between [1, 49] each with equal probability of 1/49.

If the value of 7a+b is in the range [1,40], then we can use these range to equally divide into 10 parts each of size 4 as below:

[1,4] [5,8] [9,12] [13,16] [17,20] [21,24] [25,28] [29,32] [33,36] [37,40]

> *Thus if the value of 7a+b lies between [1,4], we return 1, if the value lies between [5,8] return 2 and so on.*
>
> *Thus return int((7a+b-1)/4)+1 if 1 ≤ 7a+b ≤ 40.*

```python
class Solution(object):
    def rand10(self):
        while True:
            a = (rand7()-1)
            b = rand7()

            c = 7*a+b

            if c <= 40:
                return int((c-1)/4)+1
```

rand10.py hosted with ❤️ by GitHub view raw

Probability that the while loop returns on each call is 40/49 because out of 49 possibilities for 7a+b, only 40 are valid. **Thus the expected number of times the while loop executes for each call is 49/40.**

For an arbitrary range [a, b] using rand7(), **first normalize the range to [1, b-a+1], then find the smallest P such that b-a+1 ≤ 7^P. Then find the largest K such that K*(b-a+1) ≤ 7^P.**

> *Let Q = 7^(P-1)*(rand7()-1) + 7^(P-2)*(rand7()-1) + … + 7*(rand7()-1) + rand7()*

Q generates integers in the range [1, 7^P].

Consider only the range [1, K*(b-a+1)] from the values of Q. Among this range, if we divide it into b-a+1 equal parts each of size K, then for any integer in the 1st part i.e. [1, K] we return 1, for 2nd part [K+1, 2K] return 2 and so on.

Thus int((Q-1)/K)+1 will return random integer in the range [1, b-a+1]. To return in the range [a, b], add a-1 to the output i.e. **a+int((Q-1)/K).**

```python
def rand7():
    return random.randint(1, 7)

def generate_random(a, b):
    while True:
        p = int(math.log(b-a+1)/math.log(7))+1
        q = 0

        for r in range(p-1, 0, -1):
            x = rand7()-1
            q += x*(7**r)

        q += rand7()
        k = int((7**p)/(b-a+1))

        if q <= k*(b-a+1):
            return a+int((q-1)/k)
```

generate_random.py hosted with ❤️ by GitHub view raw

Expected number of times while loop executes is 7^P/K*(b-a+1).

Problem 2:

You are given a **0-indexed** array of positive integers `w` where `w[i]` describes the **weight** of the `ith` index.

You need to implement the function `pickIndex()`, which **randomly** picks an index in the range `[0, w.length - 1]` (**inclusive**) and returns it. The **probability** of picking an index `i` is `w[i] / sum(w)`.

Solution:

> *Store the cumulative sums of weights till index i for each index i in the array. Generate a random integer M between [1, sum(w)].*
>
> *Find the smallest index j, such that cum_sum[j] ≥ M. Return j.*

Since the cumulative sums array is a sorted array, we can do **binary search to find the index j s.t. M ≤ cum_sum[j].**

```python
class Solution(object):

    def __init__(self, w):
        self.cum_sum = [0]*len(w)
        s = 0
        for i in range(len(w)):
            s += w[i]
            self.cum_sum[i] = s


    def pickIndex(self):
        u = random.randint(1, self.cum_sum[-1])

        left, right = 0, len(self.cum_sum)-1
        p = -1

        while left <= right:
            m = int((left+right)/2)

            if self.cum_sum[m] >= u:
                p = m
                right = m-1
            else:
                left = m+1

        return p
```

random_pick.py hosted with ❤️ by GitHub view raw

Time complexity is O(logN) for each call to pickIndex().

This problem have very wide applications such as:

- Randomly selecting a server by load balancer based on the negative of current load as weights.
- Randomly sampling data from a discrete probability distribution.
- Generating artificial data for simulating traffic.

Problem 3:

You are given an integer `n` and an array of **unique** integers `blacklist`. Design an algorithm to pick a random integer in the range `[0, n - 1]` that is **not** in `blacklist`. Any integer that is in the mentioned range and not in `blacklist` should be **equally likely** to be returned.

Optimize your algorithm such that it minimizes the number of calls to the **built-in** random function of your language.

Solution:

Sort the integers in the blacklist.

Then generate intervals of the form [blacklist[i-1]+1, blacklist[i]-1], i.e. **divide the entire range of integers such that it excludes the blacklisted integers**

For example, if the range of integers is [1, 100] and sorted blacklisted integers are [10, 25, 40, 90], then the generated ranges are [1,9] [11,24] [26,39], [41,89] [91,100].

> *Then based on the size of each range, pick a range with probability = size of range/sum of sizes of all ranges.*
>
> *This can be done using the algorithm provided in problem 2.*
>
> *After picking a range, randomly pick an integer within that range.*

Probability of picking a number = Probability of picking a range * Probability of picking a number in that range.

If size of range is S, then probability = S/sum(size of ranges)*1/S = 1/sum(size of ranges)

Since sum(size of ranges) = N-len(blacklisted), thus each number not in blacklist is picked uniformly.

```python
class Solution(object):

    def __init__(self, n, blacklist):
        self.arr = []
        self.cum_sum = []

        if len(blacklist) == 0:
            self.arr = [[0, n-1]]
            self.cum_sum = [n]

        else:
            blacklist = sorted(blacklist)

            for i in range(len(blacklist)):
                start = blacklist[i-1]+1 if i > 0 else 0
                end = blacklist[i]-1

                if end >= start:
                    self.arr.append([start, end])

                    if len(self.cum_sum) > 0:
                        self.cum_sum.append(self.cum_sum[-1]+end-start+1)
                    else:
                        self.cum_sum.append(end-start+1)

            start = blacklist[-1]+1
            end = n-1

            if end >= start:
                self.arr.append([start, end])

                if len(self.cum_sum) > 0:
                    self.cum_sum.append(self.cum_sum[-1]+end-start+1)
                else:
                    self.cum_sum.append(end-start+1)

    def pick(self):
        k = random.randint(1, self.cum_sum[-1])

        left, right = 0, len(self.cum_sum)-1
        p = -1
        while left <= right:
            m = int((left+right)/2)

            if self.cum_sum[m] >= k:
```

```
                if self.cum_sum[m] >= k:
                    p = m
                    right = m-1
                else:
                    left = m+1

            return random.randint(self.arr[p][0], self.arr[p][1])
```

random_pick_blacklisted.py hosted with ❤ by GitHub view raw

Problem 4:

Given a method biased_toss() that generates 0 or 1. It generates 1 with some probability p (not necessarily 0.5). Write a method that generates 0 or 1 with probability 0.5.

Solution:

If we call biased_toss() method twice, then the possibilities are 00, 01, 10, 11. The probabilities of 00 is $(1-p)^2$, 01 is (1-p)*p, 10 is p*(1-p) and 11 is p^2.

> *Since the probabilities of 01 and 10 are equal i.e. p*(1-p), we can return 0 if we obtain 01 and 1 if we obtain 10.*

```
def get_unbiased_toss():
    while True:
        a = biased_toss()
        b = biased_toss()

        if a^b == 1:
            return a

```

biased_toss.py hosted with ❤ by GitHub view raw

Expected number of times while loop executes is 1/p*(1-p).

Problem 5:

Given an infinite stream of numbers. Write a method that would return K random samples from the stream without replacement.

Solution:

This can be solved using reservoir sampling.

For each number in the stream, we track its index.

> *If the size of the sample array is less than K, then append the number at the end of the array.*
>
> *If the size of the sample array is more than equal to K, then generate a random integer between 0 and current index of the number in the stream. If the random integer is between 0 and K-1, then replace the index in the array corresponding to the random integer with the current number.*

```python
class ReservoirSampling:
    def __init__(self, k):
        self.out = []
        self.max_length = k
        self.index = 0

    def add(self, num):
        if self.index < self.max_length:
            self.out.append(num)
        else:
            r = random.randint(0, self.index)
            if r < self.max_length:
                self.out[r] = num

        self.index += 1

    def get_samples(self):
        return self.out
```

reservoir_sampling.py hosted with ❤️ by GitHub view raw

We have to prove that the probability of each number in the stream, to be in the sample is equal to K/N where K is the length of samples array and N is the length of stream.

Assuming the length of stream is N.

For numbers with index < K, it is part of the sample if it is not replaced by any number with index ≥ K.

> *Probability that number with index J < K is part of sample = Probability that random integer generated at index K is not equal to J and Probability that random integer generated at index K+1 is not equal to J and … Probability that random integer generated at index N-1 is not equal to J.*

Probability that random integer generated at index K is not equal to J = K/K+1

Probability that random integer generated at index K+1 is not equal to J = K+1/K+2

Probability that random integer generated at index N-1 is not equal to J = N-1/N

Thus

> *Probability that number with index J < K is part of sample = K/(K+1)*(K+1)/(K+2)*…*(N-1)/N=K/N.*

For numbers with index J ≥ K, it is part of the sample if the random integer X generated between 0 and J is < K and the random integers generated at index > J is not equal to X.

> *Probability that number with index J ≥ K is part of sample = Probability that random integer X generated between 0 and J is < K and Probability that random integer generated at index J+1 is not equal to X and Probability that random integer generated at index J+2 is not equal to X and … Probability that random integer generated at index N-1 is not equal to X.*

Probability that random integer X generated between 0 and J is < K = K/(J+1)

Probability that random integer generated at index J+1 is not equal to X = J+1/(J+2)

Probability that random integer generated at index J+2 is not equal to X = J+2/(J+3)

Probability that random integer generated at index N-1 is not equal to X = N-1/N

Thus

> *Probability that number with index J ≥ K is part of sample = K/(J+1)*(J+1)/(J+2)* (J+2)/(J+3)*…*(N-1)/N=K/N.*

Problem 6:

Randomly shuffle an array.

Solution:

Randomly sample an index j from i to Len(arr)-1. Then swap arr[j] with arr[i] and increment i by 1.

```python
def shuffle(arr):
    for i in range(len(arr)):
        j = random.randint(i, len(arr)-1)
        temp = arr[j]
        arr[j] = arr[i]
        arr[i] = temp

```

random_shuffle.py hosted with ❤️ by GitHub view raw

Problem 7:

Flip a fair coin repeatedly until you get two heads in a row (HH). What is the probability of getting HH in at-most N (N > 1) tosses ? Can you write a recursive function and implementation to calculate the probability ?

Solution:

> *Let F(n, H) be the number of sequences of length n ending with HH and beginning with H and F(n, T) be the number of sequences of length n ending with HH and beginning with T.*

Thus F(n+1, T) = F(n, H) + F(n, T) because **if the sequence begins with T then the remaining sequence can begin with either H or T.**

F(n+1, H) = F(n, T) because **if the sequence begins with H then the remaining sequence can begin only with T (HH is forbidden).**

Let G(n+1) = F(n+1, H) + F(n+1, T) = F(n, T) + F(n, H) + F(n, T) = F(n-1, H) + F(n-1, T) + F(n, H) + F(n, T)

> *G(n+1) = G(n) + G(n-1)*

G(1) = 0, G(2) = 1

> *Probability P(n) = (1/2²)*G(2) + (1/2³)*G(3) + (1/2⁴)*G(4) + …. + (1/2^n)*G(n)*

```python
def probability(n):
    dp = [0]*n
    dp[1], dp[2] = 0, 1

    for i in range(3, n+1):
        dp[i] = dp[i-1] + dp[i-2]

    p = 0
    power = 1.0/2.0**2

    for i in range(2, n+1):
        p += power*dp[i]
        power /= 2.0

    return p
```

hh.py hosted with ❤️ by GitHub view raw

Problem 8:

Compute nCr where n ≥ r and 1 ≤ n ≤ 1000, 1 ≤ r ≤ n.

Solution:

nCr = n-1Cr + n-1Cr-1

Thus we will use dynamic programming to compute the values:

```python
def ncr(n, r):
    dp = [[0]*(r+1) for i in range(n+1)]

    for i in range(1, n+1):
        for j in range(1, min(r+1, i+1)):
            if i == j:
                dp[i][j] = 1
            elif j == 1:
                dp[i][j] = i
            else:
                dp[i][j] = dp[i-1][j] + dp[i-1][j-1]

    return dp[n][r]
```

ncr.py hosted with ❤️ by GitHub view raw

Time and space complexity is $O(N^2)$.

Problem 9:

Implement Stratified sampling.

Given training data, class labels and the fraction of validation data per class ‘test_size’. Write a method to sample train data, test data, train labels and test labels such that each class has ‘test_size’ fraction of data/labels for validation and 1-test_size fraction of data/labels for training.

Solution:

> *One of the solution is the first group by based on class labels, then for each group call the sampling method to get the validation samples and then use the remaining as training samples.*

Since the **size of the data is fixed** here, instead of using reservoir sampling as above (for streaming data), we will use another more cleaner method:

> *Generate a uniform random number between 0 and 1. If it is ≤ test_size, then add the current data and label in validation set else add it in the training set.*

```python
def stratified_sampling(data, labels, test_size=0.2):
    indices_per_label = {}

    for i in range(len(labels)):
        if labels[i] not in indices_per_label:
            indices_per_label[labels[i]] = []
        indices_per_label[labels[i]].append(i)

    test_data, train_data, test_labels, train_labels = [], [], [], []

    for label, indices in indices_per_label.items():
        for i in indices:
            k = random.uniform(0, 1)

            if k <= test_size:
                test_data += [data[i]]
                test_labels += [labels[i]]
            else:
                train_data += [data[i]]
                train_labels += [labels[i]]

    return train_data, test_data, train_labels, test_labels
```

stratified1.py hosted with ❤️ by GitHub view raw

But do we really need to first group by the class labels ?

If we iterate over all data, then what is the probability of me selecting a point for adding in the validation set s.t. the probability per class is test_size.

Let P(y=1) be the probability that a data /label is selected and P(Ci) be the probability of class Ci or fraction of class Ci in the data.

> *Then P(y=1) = P(y=1|C1)*P(C1) + P(y=1|C2)*P(C2) + … + P(y=1|Cm)*P(Cm) where there are 'm' classes.*

We know that P(y=1|C1) = P(y=1|C2) = …. = test_size i.e. the probability of selecting a data/label for validation given the class label is test_size.

> *Thus P(y=1) = test_size*(P(C1) + P(C2) + … + P(Cm)) = test_size, since P(C1) + P(C2) + … + P(Cm) = 1*

Thus instead of grouping by class label, we can iterate over all data and select a data/label if uniform(0,1) ≤ test_size.

```
def stratified_sampling(data, labels, test_size=0.2):
    test_data, train_data, test_labels, train_labels = [], [], [], []

    for i in range(len(data)):
        k = random.uniform(0, 1)

        if k <= test_size:
            test_data += [data[i]]
            test_labels += [labels[i]]
        else:
            train_data += [data[i]]
            train_labels += [labels[i]]

    return train_data, test_data, train_labels, test_labels
```

startified2.py hosted with ❤️ by GitHub view raw

It may not be intuitive at first since it might appear that we are sampling test_size fraction over all data instead of per class, but observe that the distribution of the classes is already handled when I am looping over all data. Thus a class with more samples will have more chance.

Problem 10:

You are given the true labels and the predicted probabilities from logistic regression model for N test examples. Approximately compute the AUC scores for ROC and PR curves.

Solution:

For ROC we need the TPR and FPR values.

TPR = TP/(Number of all data with true label=1)

FPR = FP/(Number of all data with true label=0)

For each probability threshold, find the TPR and FPR values.

Given a threshold T, TP = number of data with probability ≥ T having true label = 1 and FP = number of data with probability ≥ T having true label = 0. Also when T decreases both TPR and FPR increases and vice versa.

We can solve this efficiently by sorting the data on probability.

After sorting on probability, starting from the end, if at index i, true label = 1, then we increment TP by 1 else we increment FP by 1, because we are assuming the threshold to be probability[i].

To compute the AUC, we make use of the trapezoid formula: 0.5*width* (height1+height2) where width is the difference is FPR between 2 consecutive thresholds and height1 is the TPR at threshold T(i) and height2 is TPR at threshold T(i+1).

```python
def auc_roc(true_labels, predictions):
    data = zip(predictions, true_labels)
    data = sorted(data, key=lambda k:k[0])

    positives = sum([1 for x, y in data if y == 1])
    negatives = sum([1 for x, y in data if y == 0])

    tp, fp = 0, 0

    prev_tpr, prev_fpr = 0, 0
    area = 0

    for i in range(len(data)-1, -1, -1):
        if data[i][1] == 1:
            tp += 1
        if data[i][1] == 0:
            fp += 1

        tpr = tp/positives if positives > 0 else 0
        fpr = fp/negatives if negatives > 0 else 0

        if prev_fpr is not None and prev_tpr is not None:
            area += 0.5*(fpr-prev_fpr)*(prev_tpr+tpr)

        prev_fpr = fpr
        prev_tpr = tpr

    return area
```

roc_auc.py hosted with ❤️ by GitHub view raw

Similarly we can compute the AUC for PR curve.

Precision = TP/(Number of data predicted to be positive i.e. count of points with probability ≥ threshold).

Recall = TP/(Number of all data with true label=1).

```python
def auc_pr(true_labels, predictions):
    data = zip(predictions, true_labels)
    data = sorted(data, key=lambda k:k[0])

    positives = sum([1 for x, y in data if y == 1])

    tp = 0

    prev_p, prev_r = 0, 0
    area = 0

    for i in range(len(data)-1, -1, -1):
        if data[i][1] == 1:
            tp += 1

        p = tp/(len(data)-i) if len(data)-i > 0 else 0
        r = tp/positives if positives > 0 else 0

        if prev_p is not None and prev_r is not None:
            area += 0.5*(r-prev_r)*(prev_p+p)

        prev_r = r
        prev_p = p

    return area
```

auc_pr.py hosted with ❤️ by GitHub view raw

Problem 11:

Implement a basic binary logistic regression classifier. It should perform the following operations:

- Train model
- Predict on testing data

Assume the training and testing data is in the form of 2D matrix and the class labels is a 1-D array.

Solution:

Important methods to consider:

- Computing sigmoid scores
- Computing the logistic loss
- Gradient descent (batch or stochastic)

```python
def sigmoid_probs(data, weights, bias):
    probabilities = [0]*len(data)

    for i in range(len(data)):
        h = bias
        for j in range(len(data[i])):
            h += weights[i]*data[i][j]

        probabilities[i] = 1.0/(1.0+math.exp(-h))

    return probabilities

def loss(data, labels, weights, bias, reg_lambda):
    probs = sigmoid_probs(data, weights, bias)
    l = 0
    for i in range(len(data)):
        l += -labels[i]*math.log(probs[i])-(1-labels[i])*math.log(1-probs[i])

    l += reg_lambda*(sum([w*w for w in weights]) + bias*bias)

    return l

def gradient_descent(data, labels, weights, bias, num_epochs, learning_rate, reg_lambda):
    for epoch in epochs:
        probs = sigmoid_probs(data, weights, bias)

        for j in range(len(data[0])):
            s = 0
            for i in range(len(data)):
                s += data[i][j]*(probs[i]-labels[i])

            s += 2*reg_lambda*weights[j]
            weights[j] -= learning_rate*s

        s = 0
        for i in range(len(data)):
            s += probs[i]-labels[i]

        s += 2*reg_lambda*bias

        bias -= learning_rate*s

        curr_loss = loss(data, labels, weights, bias, reg_lambda)
        print(epoch, loss)

```

```
    return [weights, bias]

def init_weights(n):
    return [random.random()]*n

class LogisticRegression:
    def __init__(self, reg_lambda, epochs, batch_size, learning_rate):
        self.weights = []
        self.bias = 0
        self.reg_lambda = reg_lambda
        self.epochs = epochs
        self.batch_size = batch_size
        self.learning_rate = learning_rate

    def train(self, data, labels):
        n = len(labels)
        #feature transformation and selection

        self.weights = init_weights(len(data))
        self.weights, self.bias = gradient_descent(data, labels,
                                                   self.weights, self.bias,
                                                   self.epochs,
                                                   self.learning_rate, self.reg_lambda)

        #hyperparameter optimization with CV

    def predict_proba(self, data):
        output = [0]*len(data)

        for i in range(len(data)):
            h = self.bias
            for j in range(len(data[i])):
                h += self.weights[j]*data[i][j]

            output[i] = 1.0/(1.0+math.exp(-h))

        return output
```

logistic_regression.py hosted with ❤️ by GitHub view raw

Problem 12:

Implement a simple decision tree classifier. It should perform the following operations:

- Train model
- Predict on testing data

Assume the training and testing data is in the form of 2D matrix and the class labels is a 1-D array.

Solution:

Important methods to consider:

- Computing information gain
- Splitting a node on a feature and feature value.

```python
class Node:
    def __init__(self):
        self.data = []
        self.labels = []
        self.feature = None
        self.feature_value = None

def get_entropy(indices, labels):
    class_map = {}

    for index in indices:
        if labels[index] not in class_map:
            class_map[labels[index]] = 0
        class_map[labels[index]] += 1

    ent = 0

    for label, cnt in class_map.items():
        p = float(cnt)/len(indices)
        ent += -p*math.log(p)

    return ent

def get_information_gain(left_indices, right_indices, labels):
    left_ent = get_entropy(left_indices, labels)
    right_ent = get_entropy(right_indices, labels)

    total_entropy = get_entropy(left_indices+right_indices, labels)

    return total_entropy-(left_ent+right_ent)

def construct_tree(data, labels, max_samples_leaf):
    if len(data) == 0:
        return None

    elif len(data) <= max_samples_leaf or len(set(labels)) == 1:
        node = Node()
        node.data = data
        node.labels = labels
        return node

    max_gain = -float()
    best_feature = None
    best_feature_value = None

```

```
    for j in range(len(data[0])):
        temp = []
        for i in range(len(data)):
            temp.append((data[i][j], i))

        temp = sorted(temp, key=lambda k:k[0])

        for k in range(len(temp)):
            x, index = temp[k]

            # unique values of features
            if k == 0 or temp[k][0] != temp[k-1][0]:
                left_indices = [temp[h][1] for h in range(k+1)]
                right_indices = [temp[h][1] for h in range(k+1, len(temp), 1)]

                ig = get_information_gain(left_indices, right_indices, labels)

                if ig > max_gain:
                    max_gain = ig
                    best_feature = index
                    best_feature_value = x

    if best_feature is not None:
        node = Node()
        node.feature = best_feature
        node.feature_value = best_feature_value

        left_data, right_data = [], []
        left_labels, right_labels = [], []

        for i in range(len(data)):
            if data[i][best_feature] <= best_feature_value:
                left_data.append(data[i])
                left_labels.append(labels[i])
            else:
                right_data.append(data[i])
                right_labels.append(labels[i])

        node.left = construct_tree(left_data, left_labels, max_samples_leaf)
        node.right = construct_tree(right_data, right_labels, max_samples_leaf)

        return node

    return None
```

```
90
91  def predictions(root, data):
92      if root.data and len(root.data) > 0:
93          return max_class(root.labels)
94
95      f = root.feature
96      v = root.feature_value
97
98      if data[f] <= v:
99          return predictions(root.left, data)
100
101     return predictions(root.right, data)
102
103 class DecisionTree:
104     def __init__(self, max_samples_leaf=10):
105         self.max_samples_leaf = max_samples_leaf
106         self.root = None
107
108     def train(self, data, labels):
109         self.root = construct_tree(data, labels, self.max_samples_leaf)
110
111     def predict(self, data):
112         out = []
113         for d in data:
114             out.append(predictions(self.root, d))
115
116         return out
```

decision_tree.py hosted with ❤️ by GitHub view raw

Date Coding Leetcode Probability LinkedIn Machine Learning

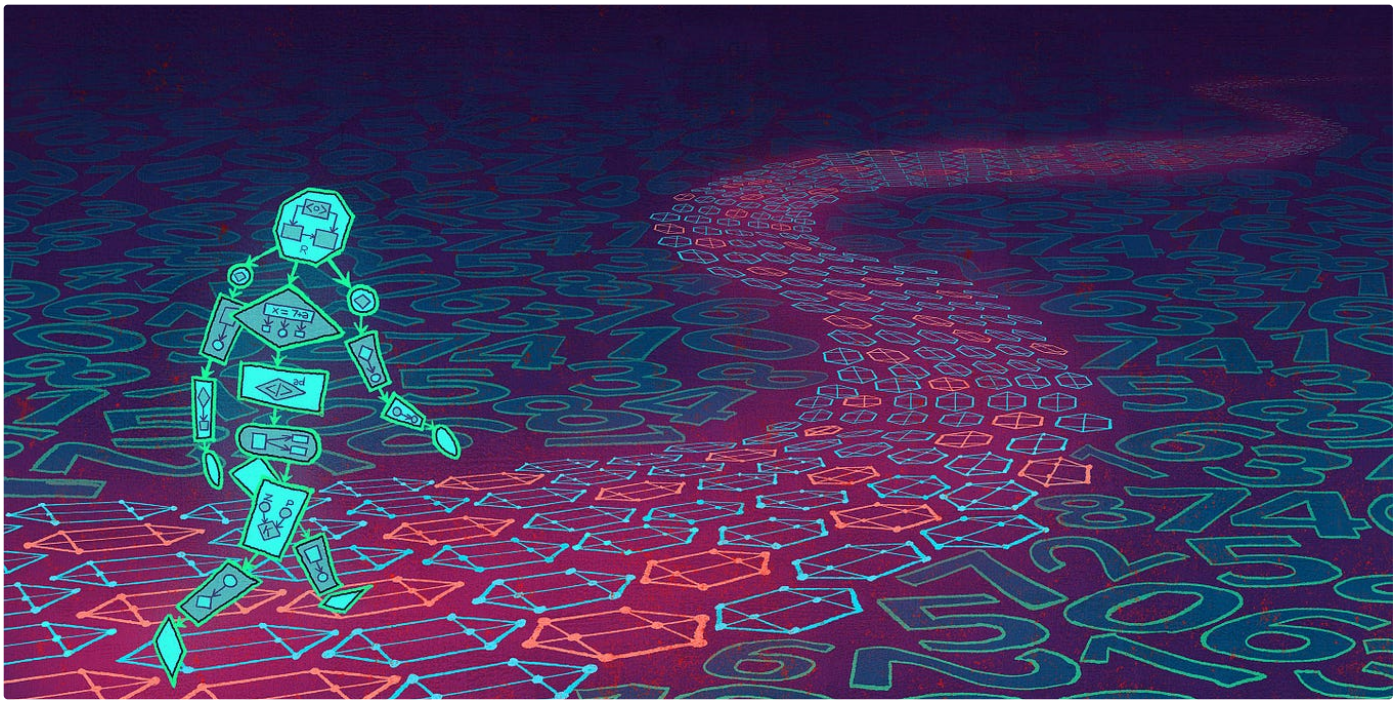
Follow

Written by Abhijit Mondal

1.6K Followers

Engineer

More from Abhijit Mondal



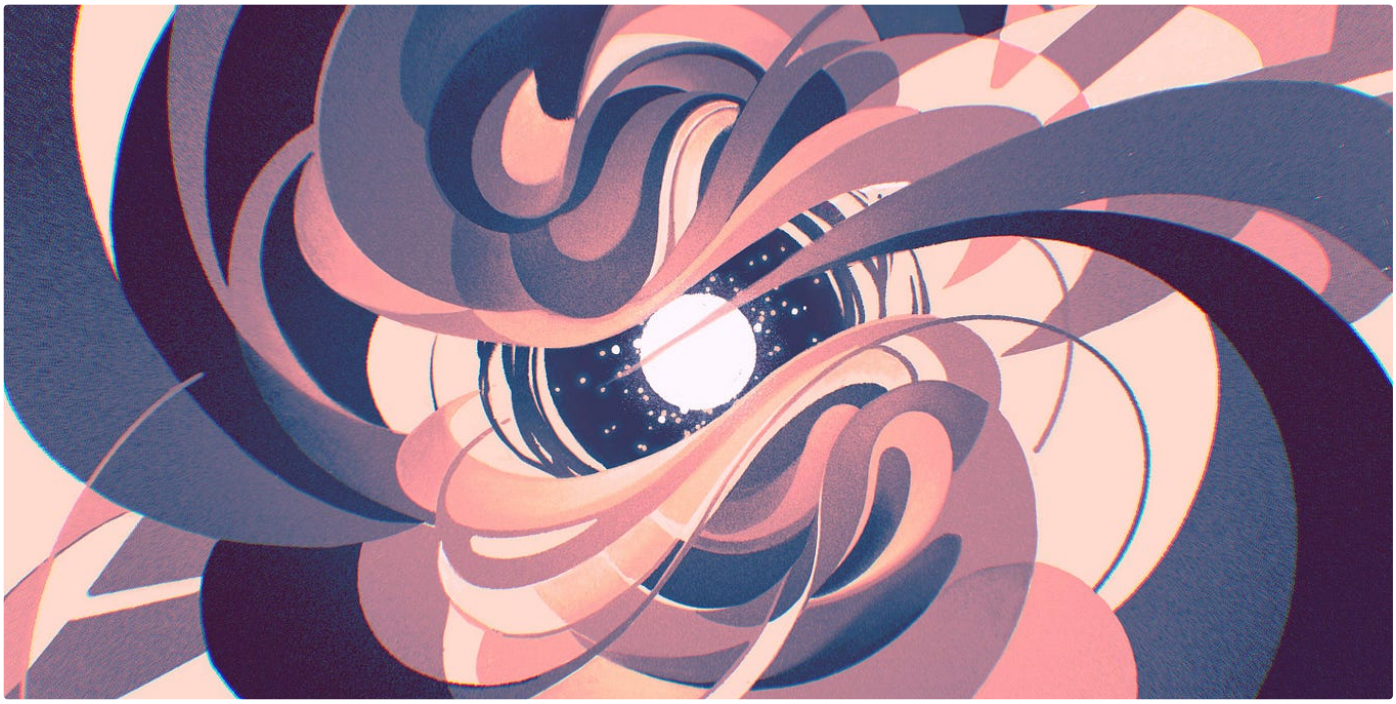
Abhijit Mondal

Think twice before using asyncio in Python

I have been using multithreading and multiprocessing in Python for quite some time now. Very recently I came across this library called...

11 min read · Jan 19

270 7



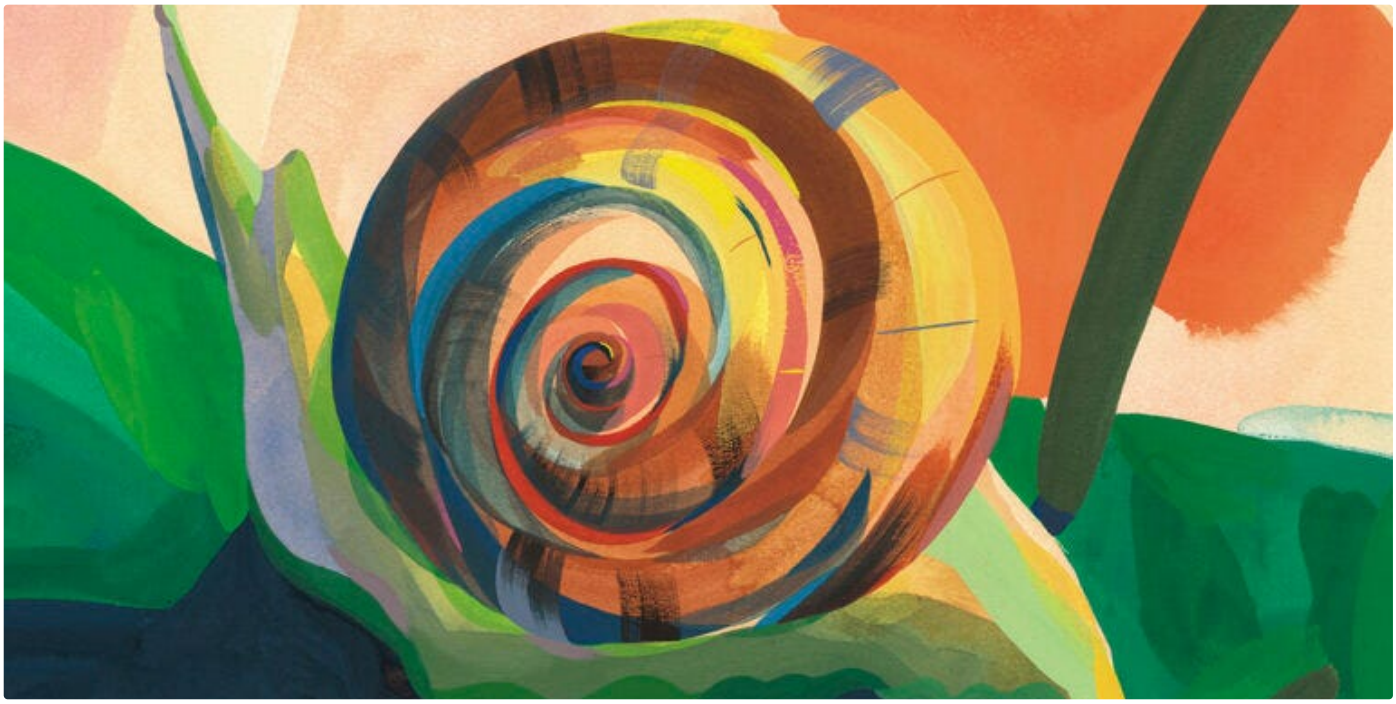
Abhijit Mondal

Algo Trading using Machine Learning on Zerodha Kite Connect

In this post I am going to explain how I (un)successfully tried my hands on Algo Trading on the Zerodha Kite Connect platform using Python...

12 min read · Sep 20, 2021

44 2



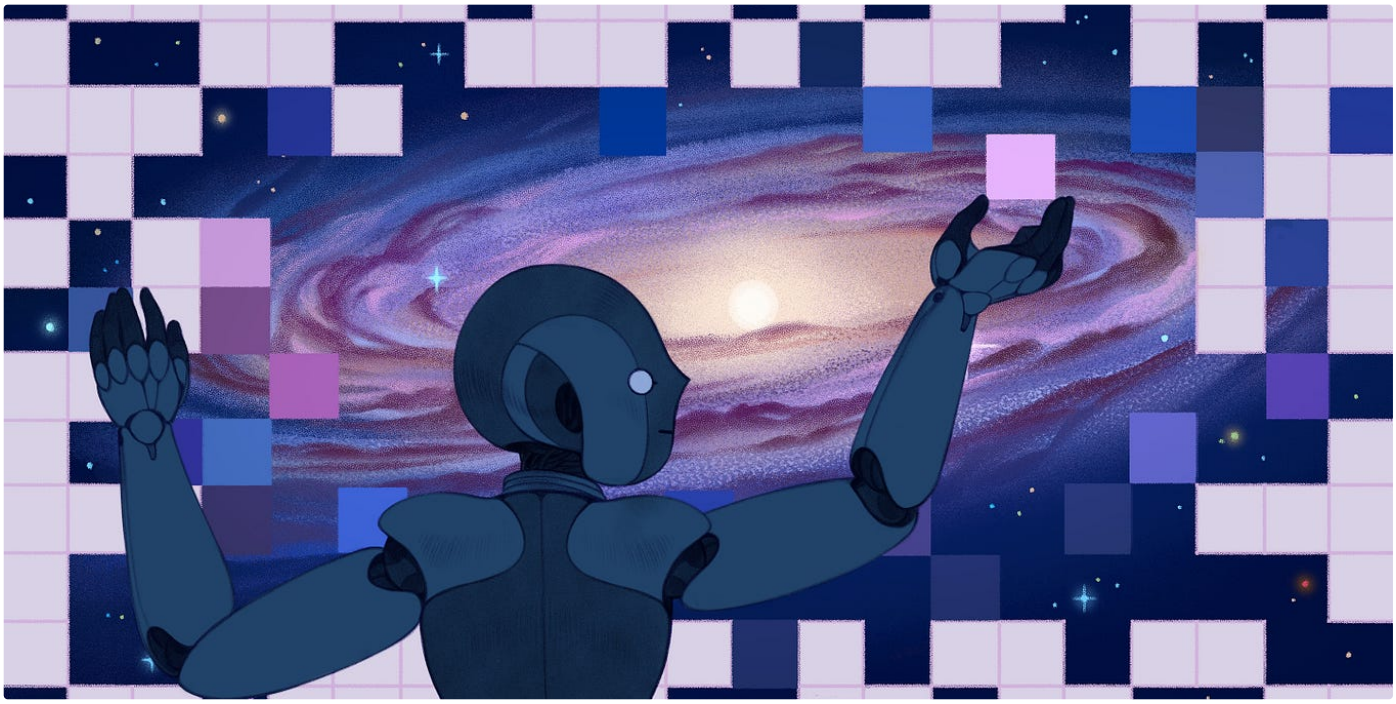
Abhijit Mondal

Deep dive into Chord for Distributed Hash Table

In a distributed hash table, a hash table with N keys is partitioned into M partitions or nodes. This is done in order to handle large...

15 min read · May 17

30



Abhijit Mondal

System Design — Top K Trending Hashtags

Design a system to compute the top K trending hashtags in the last 24 hours for Twitter/Instagram.

9 min read · Jan 29, 2022

Himanshu Joshi in Towards AI

How I cleared AWS Machine Learning Specialty With Three Weeks of Preparation (I will burst some...

How I prepared for the test, my emotional journey during preparation, and my actual exam experience

9 min read · Jan 18

130 2

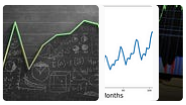
Arslan Ahmad in Level Up Coding

Don’t Just LeetCode; Follow the Coding Patterns Instead

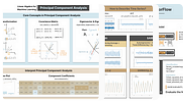
What if you don’t like to practice 100s of coding questions before the interview?

· 5 min read · Sep 15, 2022

Lists

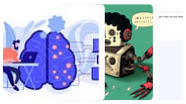


Predictive Modeling w/ Python
18 stories · 80 saves



Practical Guides to Machine Learning
10 stories · 91 saves

Natural Language Processing
377 stories · 31 saves



The New Chatbots: ChatGPT, Bard, and Beyond
13 stories · 31 saves

Mehul Gupta in Data Science in your pocket

Contextual Bandits in reinforcement learning explained with example and codes

starting deep reinforcement learning using tensorflow

5 min read · Feb 26

121

Satnam Singh in Python in Plain English

Advanced Python Programming: Writing Efficient Code with Cython and Numba

Python is a popular programming language for its simplicity, ease of use, and flexibility. However, when it comes to performance, Python...

6 min read · Feb 28

46

Youssef Hosni in Geek Culture

Crack the Machine Learning Coding Questions

Review these Algorithms and Their Code Before Your Next Machine Learning Interview

· 17 min read · Jan 31


Srushti Sonavane

Barclays Interview Experience

In the second week of October 2022, Barclays came to my college for the position of Software Intern (Summer Internship). The recruitment...

4 min read · Jun 7


41

See more recommendations