

8E and 8F: Finding the Probability $P(Y=1|X)$

8E: Implementing Decision Function of SVM RBF Kernel

After we train a kernel SVM model, we will be getting support vectors and their corresponding coefficients α_i

Check the documentation for better understanding of these attributes:

<https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html>

Attributes:	support_ : array-like, shape = [n_SV] Indices of support vectors.
	support_vectors_ : array-like, shape = [n_SV, n_features] Support vectors.
	n_support_ : array-like, dtype=int32, shape = [n_class] Number of support vectors for each class.
	dual_coef_ : array, shape = [n_class-1, n_SV] Coefficients of the support vector in the decision function. For multiclass, coefficient for all 1-vs-1 classifiers. The layout of the coefficients in the multiclass case is somewhat non-trivial. See the section about multi-class classification in the SVM section of the User Guide for details.
	coef_ : array, shape = [n_class * (n_class-1) / 2, n_features] Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel.
	coef_ is a readonly property derived from <code>dual_coef_</code> and <code>support_vectors_</code> .
	intercept_ : array, shape = [n_class * (n_class-1) / 2] Constants in decision function.
	fit_status_ : int 0 if correctly fitted, 1 otherwise (will raise warning)
	probA_ : array, shape = [n_class * (n_class-1) / 2] If probability=True, the parameters learned in Platt scaling to produce probability estimates from decision values. If probability=False, an empty array. Platt scaling uses the logistic function $1 / (1 + \exp(\text{decision_value} * \text{probA_} + \text{probB_}))$ where <code>probA_</code> and <code>probB_</code> are learned from the dataset [R20c70293ef72-2]. For more information on the multiclass case and training procedure see section 8 of [R20c70293ef72-1].
	probB_ : array, shape = [n_class * (n_class-1) / 2] If probability=True, the parameters learned in Platt scaling to produce probability estimates from decision values. If probability=False, an empty array. Platt scaling uses the logistic function $1 / (1 + \exp(\text{decision_value} * \text{probA_} + \text{probB_}))$ where <code>probA_</code> and <code>probB_</code> are learned from the dataset [R20c70293ef72-2]. For more information on the multiclass case and training procedure see section 8 of [R20c70293ef72-1].

As a part of this assignment you will be implementing the `decision_function()` of kernel SVM, here `decision_function()` means based on the value return by `decision_function()` model will classify the data point either as positive or negative

Ex 1: In logistic regression After training the models with the optimal weights w we get, we will find the value $\frac{1}{1+\exp(-(wx+b))}$, if this value comes out to be < 0.5 we will mark it as negative class, else its positive class

Ex 2: In Linear SVM After training the models with the optimal weights w we get, we will find the value of $\text{sign}(wx + b)$, if this value comes out to be -ve we will mark it as negative class, else its positive class.

Similarly in Kernel SVM After training the models with the coefficients α_i we get, we will find the value of $\text{sign}(\sum_{i=1}^n (y_i \alpha_i K(x_i, x_q)) + \text{intercept})$, here $K(x_i, x_q)$ is the RBF

kernel. If this value comes out to be -ve we will mark x_q as negative class, else its positive class.

RBF kernel is defined as: $K(x_i, x_q) = \exp(-\gamma \|x_i - x_q\|^2)$

For better understanding check this link: <https://scikit-learn.org/stable/modules/svm.html#svm-mathematical-formulation>

▼ Task E

1. Split the data into $X_{train}(60)$, $X_{cv}(20)$, $X_{test}(20)$
2. Train $SVC(\gamma = 0.001, C = 100.)$ on the (X_{train}, y_{train})
3. Get the decision boundry values f_{cv} on the X_{cv} data i.e. $f_{cv} = \text{decision_function}(X_{cv})$ **you need to implement this decision_function()**

```
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
import numpy as np
from sklearn.svm import SVC
```

```
X, y = make_classification(n_samples=5000, n_features=5, n_redundant=0, n_classes=2, weigh
```

```
X
array([[ -0.31030127, -1.47300088,  0.59124743,  0.8705559 , -0.42524742],
       [ -0.17942473, -1.13412607, -0.14583869, -1.58198284, -1.32817149],
       [  0.14981056,  0.77537917, -0.75092228,  0.31810543,  0.19646665],
       ...,
       [-1.09494183,  1.57206292,  1.09204273, -0.20535612,  0.00210579],
       [  0.58144391, -0.16742201,  1.55598071,  0.82693028, -0.63111542],
       [-0.25435274,  0.10570486, -1.1573751 , -0.1002696 , -0.49148178]])
```

```
y
array([0, 0, 0, ..., 0, 0, 0])
```

▼ Pseudo code

```
clf = SVC(gamma=0.001, C=100.)
clf.fit(Xtrain, ytrain)
```

```
def decision_function(Xcv, ...): #use appropriate parameters
```

```
    for a data point  $x_q$  in Xcv:
```

```
        #write code to implement  $(\sum_{i=1}^{\text{all the support vectors}} (y_i \alpha_i K(x_i, x_q)) + \text{intercept})$ , here
```

the values y_i , α_i , and *intercept* can be obtained from the trained model

return # the decision_function output for all the data points in the Xcv

fcv = decision_function(Xcv, ...) # based on your requirement you can pass any other parameters

Note: Make sure the values you get as fcv, should be equal to outputs of
clf.decision_function(Xcv)

you can write your code here:

```
from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)
```

```
X_train, X_cv, y_train, y_cv = train_test_split(X_train, y_train, test_size=0.25, stratify=y_train)
```

```
gamma = 0.001
```

```
C = 100
```

```
clf = SVC(gamma=0.001, C=100)
```

```
clf.fit(X_train, y_train)
```

```
def rbf(sv, xq, gamma):
```

```
    norm = np.linalg.norm(sv-xq) #Ref: https://numpy.org/doc/stable/reference/routines.linalg
```

```
    return np.exp(-gamma * (norm**2))
```

```
def decision_function(X_cv, gamma, support_vectors, dual_coef, intercept):
```

```
    decision_vector = []
```

```
    for xq in X_cv :
```

```
        dec_func = 0
```

```
        for sv, dc in zip(support_vectors, dual_coef[0]):
```

```
            dec_func += (dc * rbf(sv, xq, gamma))
```

```
        dec_func += intercept[0]
```

```
        decision_vector.append(dec_func)
```

```
    return decision_vector
```

#Ref: <https://stackoverflow.com/questions/28503932/calculating-decision-function-of-svm-model>

```
support_vectors = clf.support_vectors_
```

```
dual_coef = clf.dual_coef_
```

```
intercept = clf.intercept_
```

```
f_cv = decision_function(X_cv, gamma, support_vectors, dual_coef, intercept)
```

```
f_cv
```

```
-3.1748997412459854,
-2.366717641975759,
-2.901590701699055,
-2.3192528399363876,
-2.373278255987037,
-3.5735612349099366,
2.230949074865487,
-1.3626440834731905,
-3.751153721776168,
-4.249949703533437,
-3.1268468440302595,
-1.8384547649728855,
```

```

-1.7931390598356582,
-2.3536582065896337,
0.6596433638622532,
-2.4275465811304766,
-3.0741882553244775,
-2.014893883063001,
-2.206103266686304,
-3.967644211831063,
-2.4788490996646058,
-0.24307561356567042,
-0.8340415545414341,
-3.381495197484062,
1.466648216275809,
-3.5288946145997837,
1.6122858794697752,
-2.242126068824751,
-2.130527864246451,
-4.55915081926023,
-3.3055768413916464,
-4.538021731808419,
-0.5624791865838485,
-4.384676214485301,
1.3113715568940965,
-0.5951872715785864,
-3.252127982368682,
-3.2003656921917294,
0.6801193892455988,
-0.24478785152286342,
-4.250642322973738,
-1.5406460638303425,
-1.8918388089433247,
1.0109727887849917,
-1.1925276002116245,
0.655739860794947,
-0.6934681349253164,
1.637995589935196,
-2.1731271075475913,
-0.8404261997400564,
-2.8858385669219526,
0.1977137289995543,
-2.5610688247787206,
-3.0265637971080164,
-1.7269803168228943,

-4.091764811005637,
1.218336823884039,
-2.9703137478718613,
0.6783630045649094,
E 411EE0022706011

```

```

# The decision_function output for all the data points in the Xcv:
clf.decision_function(X_cv)

```

```

-1.72933334, -3.77037575, -3.01170517, 1.0099410 , -1.2942700 ,
-2.68852276, -1.3925468 , -3.23558252, -2.3200091 , -2.97706875,
-2.31286139, 0.22882942, -2.27795692, -2.36421912, -1.90817487,
-2.72642553, -3.02881163, 0.40798879, 0.4395717 , -1.23873678,
-0.21057563, -3.53557214, 1.11880212, 1.63538729, -3.67599531,
-3.51539854, -2.45992361, -2.16139896, -3.52305814, -1.36736211,
-3.12621073, -1.83833515, 1.89675418, 2.06933816, -1.76660455,
1.03229334, -3.00619463, -3.19375237, 1.38519937, -1.88456953,
-2.79600489, -2.86321146, -0.97257066, -3.84177916, 1.905331 ,
-2.09548223, -3.10524789, -0.42642993, 0.23485501, -3.24949081,

```

```

-1.46056528, -3.58565278, -0.66161349, 1.63352162, -3.02119787,
-0.83988327, -2.58605312, -2.65402395, -2.79869721, -3.09330434,
1.6729751 , 1.09062559, -5.1634246 , -2.51656483, -0.05687025,
-3.32941649, -1.6903695 , -1.620158 , -4.20885442, -2.37526288,
1.50425079, -2.68460966, -1.62326581, -2.92920055, -3.78685586,
1.77078717, -3.33319929, 0.4979334 , 1.33764543, -1.20099865,
1.45208765, -3.06360694, -3.46351682, -2.90619908, -0.91782615,
-1.75822344, -4.2374109 , -1.86406829, -3.64509418, -2.8222831 ,
-3.32655432, -2.87528889, -2.36162303, 2.13779414, -2.76507641,
-2.60536498, -2.75974166, -2.85542468, 0.83706501, 2.44983352,
-0.02473069, -3.46600621, 2.19262518, -5.12008012, -2.46893698,
-3.2326429 , -2.99677102, -2.07125582, 3.47727913, -3.61559047,
1.91880542, 0.67657938, -3.49360467, -3.04056889, 1.69923583,
-0.44479607, -2.10531892, -4.09388469, -0.20387945, -1.57513472,
-1.07807313, -4.34771262, -3.06045908, -3.11146032, -3.57687815,
0.67917467, -2.80618142, -0.60553749, 1.36955234, -0.75435127,
-2.02488966, -2.41127424, -3.88531351, -3.0133622 , 0.97189442,
1.172032 , -3.48189226, -3.77407935, 2.28718274, 1.70030878,
-3.07360795, -2.540047 , -1.49707139, -1.59582857, -2.39360785,
-3.92524522, 2.04713801, 1.03753176, -2.85507535, -3.09802005,
-3.09099971, -2.90973483, 2.30831486, 1.82377642, -4.77074081,
-1.77118232, -0.69140543, -3.3901122 , -3.92010083, -1.08668557,
2.00397567, 1.55469013, -2.78804528, -3.08971049, -3.88374231,
-3.31931005, -3.15500523, -3.68031916, -2.6553185 , 1.46030682,
0.28160632, 1.31413107, 1.71576595, -3.22089574, 1.68365276,
-3.11392537, -4.10493374, -0.67828322, -3.32591427, -4.00975732,
-1.64462349, -2.35139509, 0.49085591, 1.8198019 , -2.48400644,
-0.04273748, -1.12371372, 1.73767945, -1.67470768, 2.25514328,
2.13046134, 0.95606246, -2.51449864, 0.54182106, -3.2245202 ,
1.3261451 , 3.92505931, -2.62961149, -0.25417885, -3.7417615 ,
1.62386292, -2.17839623, -2.38059804, -4.15080552, 0.9673356 ,
-2.12019889, 2.33173966, -3.48685402, 1.77225149, 2.60361242,
2.05396132, 1.92023785, -2.37430995, -3.29616114, -1.18051616,
-1.76495936, -2.25729339, -5.07551389, 2.31161035, 1.49596473,
-4.76661302, -0.36216339, -3.26427218, -3.16155285, 1.680044 ,
1.72005311, -0.20172613, 1.91757899, -3.78592742, -2.45786093,
-3.44665774, 0.40743096, -2.57651175, -2.18236254, 1.51120899,
3.41273663, -3.31343402, -3.17489974, -2.36671764, -2.9015907 ,
-2.31925284, -2.37327826, -3.57356123, 2.23094907, -1.36264408,
-3.75115372, -4.2499497 , -3.12684684, -1.83845476, -1.79313906,
-2.35365821, 0.65964336, -2.42754658, -3.07418826, -2.01489388,
-2.20610327, -3.96764421, -2.4788491 , -0.24307561, -0.83404155,
-3.3814952 , 1.46664822, -3.52889461, 1.61228588, -2.24212607,

-2.13052786, -4.55915082, -3.30557684, -4.53802173, -0.56247919,
-4.38467621, 1.31137156, -0.59518727, -3.25212798, -3.20036569,
0.68011939, -0.24478785, -4.25064232, -1.54064606, -1.89183881,
1.01097279, -1.1925276 , 0.65573986, -0.69346813, 1.63799559,
-2.17312711, -0.8404262 , -2.88583857, 0.19771373, -2.56106882,
-3.0265638 , -1.72698032, -4.09176481, 1.21833682, -2.97031375,
0.678363 , -5.41155093, -1.51324634, 1.43029028, -1.80320408,

```

8F: Implementing Platt Scaling to find $P(Y=1|X)$

- Let the output of a learning method be $f(x)$. To get calibrated probabilities, pass the output through a sigmoid:

$$P(y = 1|f) = \frac{1}{1 + \exp(Af + B)} \quad (1)$$

where the parameters A and B are fitted using maximum likelihood estimation from a fitting training set (f_i, y_i) . Gradient descent is used to find A and B such that they are the solution to:

$$\underset{A, B}{\operatorname{argmin}} \left\{ - \sum_i y_i \log(p_i) + (1 - y_i) \log(1 - p_i) \right\}, \quad (2)$$

where

$$p_i = \frac{1}{1 + \exp(Af_i + B)} \quad (3)$$

Two questions arise: where does the sigmoid train set come from? and how to avoid overfitting to this training set?

If we use the same data set that was used to train the model we want to calibrate, we introduce unwanted bias. For example, if the model learns to discriminate the train set perfectly and orders all the negative examples before the positive examples, then the sigmoid transformation will output just a 0,1 function. So we need to use an independent calibration set in order to get good posterior probabilities. This, however, is not a draw back, since the same set can be used for model and parameter selection.

To avoid overfitting to the sigmoid train set, an out-of-sample model is used. If there are N_+ positive examples and N_- negative examples in the train set, for each training example Platt Calibration uses target values y_+ and y_- (instead of 1 and 0, respectively), where

$$y_+ = \frac{N_+ + 1}{N_+ + 2}; \quad y_- = \frac{1}{N_- + 2} \quad (4)$$

For a more detailed treatment, and a justification of these particular target values see (Platt, 1999).

Check this [PDE](#)

▼ TASK F

4. Apply SGD algorithm with (f_{cv}, y_{cv}) and find the weight W intercept b Note: here our data is of one dimensional so we will have a one dimensional weight vector i.e $W.shape (1,)$

Note1: Don't forget to change the values of y_{cv} as mentioned in the above image. you will calculate y_+ , y_- based on data points in train data

Note2: the Sklearn's SGD algorithm doesn't support the real valued outputs, you need to use the code that was done in the 'Logistic Regression with SGD and L2' Assignment after modifying loss function, and use same parameters that used in that assignment.

```
def log_loss(w, b, X, Y):
    N = len(X)
    sum_log = 0
    for i in range(N):
        sum_log += Y[i]*np.log10(sig(w, X[i], b)) + (1-Y[i])*np.log10(1-sig(w, X[i], b))
    return -1*sum_log/N
```

if $Y[i]$ is 1, it will be replaced with y_+ value else it will be replaced with y_- value

5. For a given data point from X_{test} , $P(Y = 1|X) = \frac{1}{1+\exp(-(W*f_{test}+b))}$
 where $f_{test} = \text{decision_function}(X_{test})$, W and b will be learned as

Note: in the above algorithm, the steps 2, 4 might need hyper parameter tuning, To reduce the complexity of the assignment we are excluding the hyperparameter tuning part, but interested students can try that

If any one wants to try other calibration algorithm isotonic regression also please check these tutorials

1. <http://fa.bianp.net/blog/tag/scikit-learn.html#fn:1>
2. https://drive.google.com/open?id=1MzmA7QaP58RDzocB0RBmRiWf7Co_VJ7
3. https://drive.google.com/open?id=133odBinMOIVb_rh_GQxxsyMRyW-Zts7a
4. https://stat.fandom.com/wiki/Isotonic_regression#Pool_Adjacent_Violators_Algorithm

```
from sklearn.linear_model import SGDClassifier
```

```
N_positive = []
N_negative = []
for i in y_train:
    if i == 1:
        N_positive.append(i)
    else:
        N_negative.append(i)
```

```
N_positive = len(N_positive)
N_negative = len(N_negative)
N_positive, N_negative
```

```
(904, 2096)
```

```
N_positive = (N_positive + 1) / (N_positive + 2)
```

y_positive

```
y_negative = 1 / (N_negative + 2)
y_negative
```

```
new_y_cv = []
```

[illegible]

0 00047664442326024784


```

0.00047664442326024784,
0.00047664442326024784,
0.9988962472406181,
0.00047664442326024784,
0.9988962472406181,
0.00047664442326024784,
0.00047664442326024784,
0.00047664442326024784,
0.00047664442326024784,
0.00047664442326024784,
0.9988962472406181,
0.9988962472406181,
0.9988962472406181,
0.00047664442326024784,
0.00047664442326024784,
0.9988962472406181,
0.00047664442326024784,
0.00047664442326024784,
0.00047664442326024784,

```

```
%%time
```

```
from tqdm import tqdm
```

```
import math
```

```
from math import log
```

```
def initialize_weights(dim):
```

```
    dim = X_train[0]
```

```
    w = np.zeros((1,))
```

```
    b = 0
```

```
    return w,b
```

```
dim = X_train[0]
```

```
w,b = initialize_weights(dim)
```

```
#1)sigmoid:
```

```
def sigmoid(z):
```

```
    return 1. / (1 + np. exp(-z))
```

```
#2)logloss:
```

```
def logloss(y_true,y_pred):
```

```
    loss = -sum(map(lambda y_true, y_pred: y_true*np.log10(y_pred) + (1-y_true)*np.log10(1-y
```

```
    return loss
```

```
#3)gradient_dw
```

```
def gradient_dw(x,y,w,b,alpha,N):
```

```
    dw = x * (y - sigmoid(np.dot(w, x) + b)) - (alpha / N) * w
```

```
    return dw
```

```
#4)gradient_db
```

```
def gradient_db(x,y,w,b):
```

```
    db = y - sigmoid(np.dot(w, x) + b)
```

```
    return db
```

```
#log loss:
```

```
log_loss=[]
```

```
epoch_list = []
```



```

-3.59523922e+00, -4.52080117e+00, -3.77689992e+00, -3.41166659e+00,
-3.18618742e+00, 1.99021661e+00, -2.52270231e+00, -3.46606799e+00,
6.68654114e-01, -5.37027658e+00, -1.14530760e+00, -2.45496128e+00,
-3.68907769e+00, -2.91417291e+00, 4.90915297e-01, -3.21503398e+00,
-3.01566992e+00, -2.00825284e+00, -2.69119621e+00, 2.22608429e+00,
1.74155203e+00, -2.65047122e+00, 1.65188133e+00, -3.75059683e+00,
2.51137876e+00, -2.36923150e+00, -3.03025083e+00, -3.24664901e+00,
-2.30646013e+00, -1.22281224e+00, -1.44444127e+00, -3.65098977e+00,
-2.73011063e+00, -2.05955674e+00, -4.87353721e+00, -1.59053692e+00,
1.66046654e+00, -3.55099212e+00, -2.45352280e+00, -2.65886597e+00,
5.86697997e-01, -3.09574362e+00, -3.12306804e+00, -1.17176338e+00,
-3.74892300e-01, -1.68697968e+00, -2.73834563e+00, 2.22813673e+00,
-2.53027927e+00, -2.42061192e+00, -3.82462824e+00, -3.34845630e+00,
-3.75944857e+00, 1.91770377e-01, 1.57347978e+00, -2.10038279e+00,

-2.61491132e+00, -1.67874856e+00, -2.75247867e+00, -4.23111263e+00,
-3.33772117e+00, -3.43778021e+00, -4.48576207e+00, -1.67445934e+00,
-2.18377704e+00, -1.99360145e+00, -2.34089854e-01, -2.61515230e+00,
-2.51657064e+00, -1.79671624e+00, 9.20627983e-01, -3.45605261e+00,
-7.70787257e-01, -1.64953105e+00, -3.98472395e-01, -3.01114945e+00,
-4.58166451e+00, -2.70021543e+00, 2.70649920e+00, 1.94458888e+00,
-3.16566177e+00, -3.66567609e+00, -2.47042520e-01, -3.69887228e+00,
-2.00546396e+00, -1.45785428e+00, -5.02026335e+00, -2.68965520e+00,
-2.45829100e+00, 1.14086854e+00, -2.49590628e+00, 1.07287426e+00,
-4.23847054e+00, 1.15870206e+00, 2.06851052e+00, -3.13894169e+00,
-2.87222352e+00, -3.71626775e+00, 2.13425735e+00, -1.97077126e+00,
1.46486033e+00, -9.52910164e-01, -1.62037327e+00, -2.88138121e+00,
2.09140642e+00, -2.77043150e+00, 1.01308891e+00, -3.08549254e+00,
-2.48995231e+00, 1.50559703e+00, -2.38621217e+00, -2.34260343e+00,
-2.62590859e+00, -3.22025762e+00, -3.27893480e+00, -2.26079375e+00,
-3.61855525e+00, 1.34740079e+00, -3.37299648e+00, -1.85521809e+00,
-2.10185975e+00, -2.67758972e+00, 2.11931852e+00, -2.71422556e+00,
-2.81069693e+00, -2.85557060e+00, -2.98131487e+00, -2.46965954e+00,
1.02681755e-01, 2.08996262e-01, 1.94186842e+00, -3.08416664e+00,
-1.60606304e+00, 1.94687752e+00, -6.95347024e-01, -3.66891558e+00,
1.14434808e+00, -5.06359006e+00, 2.43894871e+00, -2.05024078e+00,
-2.06784438e+00, -2.52516966e+00, -7.30076228e-01, -2.22061748e+00,
1.29788380e+00, -4.45178750e+00, 1.16867640e+00, -1.36205553e+00,
1.59695258e+00, -3.23130852e+00, 1.23174095e+00, -3.01110772e+00,
8.24880155e-02, -3.10850968e+00, 1.43829253e+00, -2.01077658e+00,
-8.80885932e-01, -1.63019777e+00, -3.41880838e+00, -2.43434892e+00,
3.69958499e-01, 2.08012418e+00, -2.48556329e+00, -2.40031503e+00,
1.96588613e+00, -6.95972181e-01, 1.53325535e+00, -1.68198576e+00,
-2.42146730e+00, -2.83981982e+00, -3.44607266e+00, 1.51830537e+00,
8.37071343e-01, -2.20860436e+00, -1.01348273e+00, -1.85339103e+00,
-1.69429741e+00, -3.26373164e+00, 1.77303571e+00, -3.68328708e+00,
-2.10447427e+00, 1.99736719e+00, 2.52180539e+00, -3.51027996e+00,
-3.61140790e+00, -1.48676548e+00, -2.11817479e+00, -3.20642657e+00,
1.94208136e+00, 1.83280491e+00, -3.97629657e+00, 2.00567497e+00,
-3.35274854e+00, 1.18641243e+00, -2.83866860e+00, 3.23038762e+00,
5.07539953e-01, -3.65167202e+00, 1.70988485e+00, -3.17271333e+00.

```

```

p = 1 / 1 + np.exp(-((w*f_test)+b))
p

```

```

array([5.68013225e+00, 1.21520137e+00, 2.28100774e+01, 7.10257907e+00,
1.76823449e+01, 9.29237190e+01, 4.45242147e+01, 5.35348742e+00,
3.55283825e+01, 3.07035937e+01, 6.29381700e+01, 1.22639657e+00,
3.14370270e+00, 1.75961812e+00, 3.85539263e+01, 7.40486716e+01,
1.65420132e+00, 1.12566407e+00, 5.98681477e+01, 7.28745114e+01,
2.42806023e+00, 1.69557126e+01, 1.14738668e+00, 8.49188615e+00,

```

```

4.96403197e+01, 6.60032102e+01, 1.53300093e+01, 7.50474655e+01,
1.38618392e+00, 1.02684308e+00, 1.06097682e+00, 1.14697853e+00,
1.04557881e+01, 9.10536779e+00, 1.96524867e+00, 3.11652488e+01,
8.39519217e+01, 1.14684946e+00, 2.46644511e+00, 2.05839174e+01,
3.50569088e+01, 2.66775510e+01, 3.56734385e+01, 1.28024194e+00,
2.15920044e+00, 6.36923187e+00, 1.16039128e+02, 5.82434552e+01,
1.49616539e+01, 7.99974427e+01, 3.71636355e+01, 1.69986244e+02,
1.12962290e+00, 3.99525216e+00, 4.71932437e+00, 2.05939748e+01,
2.01806381e+01, 2.18234108e+01, 1.13913329e+00, 1.22670083e+01,
1.71409132e+02, 5.87457777e+00, 4.24764929e+01, 1.64706802e+01,
4.99947531e+00, 1.11242617e+01, 1.23210156e+00, 1.26896730e+02,
1.86090104e+01, 1.18039781e+01, 6.28564959e+01, 2.52553378e+01,
2.91260672e+01, 6.81936980e+00, 1.05592394e+00, 7.00671867e+01,
2.34096923e+01, 2.48783626e+00, 1.12828391e+00, 1.15384966e+00,
1.09823253e+00, 8.38405013e+00, 2.08277304e+01, 1.05939172e+01,
4.70543183e+01, 1.84240626e+00, 5.20620549e+01, 1.28459471e+01,
1.35825472e+02, 2.59510758e+01, 8.35729489e+00, 2.03782086e+01,
3.17897415e+01, 2.94338093e+01, 1.32114273e+01, 8.69707956e+01,
1.12160017e+01, 1.14930818e+01, 1.90737889e+01, 2.21305612e+02,
1.09900183e+00, 2.90273551e+00, 6.54205948e+00, 3.96604918e+00,
3.23918692e+00, 2.54445202e+01, 1.10656499e+00, 1.19282206e+00,
1.30692151e+00, 1.40364765e+00, 1.11946338e+01, 1.22372899e+00,
1.34294114e+01, 1.15676464e+00, 1.09785744e+00, 1.25482745e+02,
4.15338424e+01, 7.05634510e+01, 1.76093404e+01, 1.03291936e+02,
6.55452059e+00, 9.18370590e+01, 1.12553269e+01, 2.86572026e+01,
2.95774776e+00, 1.17311986e+02, 6.40929271e+01, 1.08596259e+02,
8.82590937e+01, 1.09239253e+02, 1.11944996e+00, 1.15253127e+00,
3.11878985e+01, 3.21400947e+01, 1.88477936e+02, 2.61174291e+01,
1.59139930e+00, 1.12332097e+00, 2.51027235e+00, 1.16732628e+00,
1.06225393e+02, 2.72521052e+01, 7.99867344e+00, 3.99563323e+01,
2.26927569e+02, 1.21261518e+00, 2.98785164e+01, 1.77445311e+01,
1.18224144e+00, 4.25202231e+01, 4.70380199e+01, 1.00509845e+00,
5.66043973e+01, 2.79333089e+00, 1.13124867e+00, 1.10400015e+00,
5.94801666e+01, 4.77298977e+01, 3.07362705e+01, 1.02294361e+01,
3.02130700e+00, 3.25873541e+01, 3.96788828e+01, 1.39836679e+01,
7.98790854e+00, 1.65797651e+00, 1.06041213e+00, 1.53676439e+00,
2.87615931e+00, 2.93887954e+01, 1.99210008e+00, 8.42762395e+01,
1.11559063e+00, 2.57945204e+00, 1.03249130e+00, 8.34198577e+01,
1.18186482e+02, 4.62469540e+00, 1.09616643e+00, 5.11993363e+01,
2.93135054e+01, 1.11416992e+00, 4.89177238e+01, 3.49669890e+00,
1.79813826e+00, 2.92240633e+00, 2.31486913e+01, 1.06001148e+00,
2.03081795e+01, 4.92913287e+01, 1.77061572e+00, 1.34907374e+01,
1.17241303e+00, 1.66137942e+01, 1.43824937e+00, 4.39769921e+01,
3.40260366e+00, 1.44016047e+01, 1.15529799e+01, 2.46464415e+01,
1.37663680e+00, 3.75394870e+01, 1.40544336e+00, 1.95855620e+00,
2.81664209e+01, 8.75094156e+01, 1.27347541e+00, 1.35564903e+00,
2.84845924e+01, 6.47080581e+00, 1.49896284e+02, 7.28677930e+00,
1.06433566e+00, 4.42117731e+01, 7.74922064e+01, 1.16807028e+00,
1.08489510e+00, 2.42696465e+00, 4.17332906e+01, 3.07456140e+00,
5.66459954e+01, 1.84964733e+01, 1.25667575e+00, 6.48357778e+01,
1.35040966e+02, 1.18342324e+01, 2.20257333e+00, 1.05982573e+00,
1.31953383e+02, 3.36890408e+01, 1.14181050e+00, 4.21550263e+00,
1.56270317e+00, 1.01424505e+02, 1.04210716e+00, 1.20120008e+01

```

Thank you!

✓ 0s completed at 5:13 PM

● ×