SVM From Scratch — Python

Important Concepts Summarized

Qandeel Abbassi Feb 8, 2020 · 13 min read ★

Cheers if you get it 😉

What this blog will cover:

Before diving right into the code or technical details, I would like to mention that while there are many libraries/frameworks available to implement SVM (Support Vector Machine) algorithm without writing a bunch of code, I decided to write the code with as few high-level libraries as possible so that you and I can get a good grasp of important components involved in training an SVM model (with 99% accuracy, 0.98 recall, and precision). If you are looking for a quick implementation of SVM, then you are better off using packages like scikit-learn, cvxopt, etc. Otherwise, let's get started!

SVM Introduction

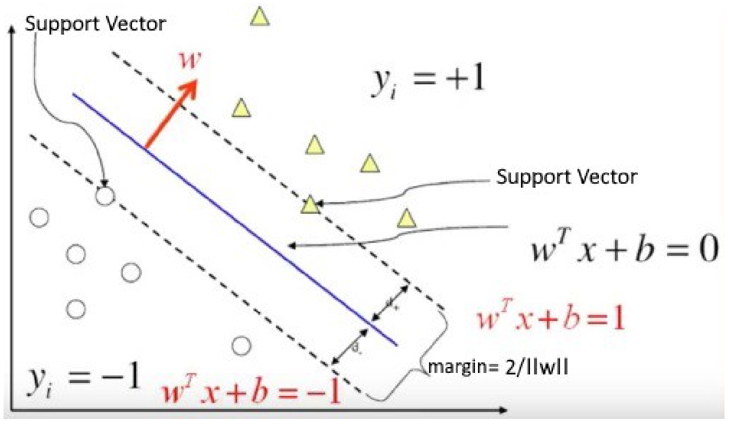


Figure 1: SVM summarized in a graph — Ireneli.eu

The **SVM** (Support Vector Machine) is a **supervised machine learning** algorithm typically used for **binary classification problems**. It's trained by feeding a *dataset* with *labeled examples* (x_i, y_i). For instance, if your examples are email messages and your problem is spam detection, then:

- An example email message $\boldsymbol{x_i}$ is defined as an *n-dimensional feature vector* that can be plotted on n-dimensional space.
- The feature vector, as the name explains, contains features (eg. word count, link count, etc.) of your email message in numerical form
- Each feature vector is labeled with a class $\boldsymbol{y_i}$
- The class $\boldsymbol{y_i}$ can either be a +ve or -ve (eg. spam=1, not-spam=-1)

Using this *dataset* the algorithm *finds a hyperplane* (or decision boundary) which should ideally have the following properties:

- It creates separation between examples of two classes with a maximum margin
- Its equation $(w.x + b = 0)$ yields a value ≥ 1 for examples from+ve class and ≤-1 for examples from -ve class

How does it find this hyperplane? By finding the optimal values *w* (weights/normal)* and *b* (intercept)* which define this hyperplane. The optimal values are found by *minimizing a cost function*. Once the algorithm identifies these optimal values, the *SVM model f(x)* is then defined as shown below:

$$f(x) = sign\left(\mathbf{w}^* \cdot x + \mathbf{b}^*\right)$$

Before we move any further let's import the required packages for this tutorial and create a skeleton of our program *svm.py*:

```
# svm.py
import numpy as np  # for handling multi-dimensional array operation
import pandas as pd  # for reading data from csv
import statsmodels.api as sm  # for finding the p-value
from sklearn.preprocessing import MinMaxScaler  # for normalization
from sklearn.model_selection import train_test_split as tts
from sklearn.metrics import accuracy_score
from sklearn.utils import shuffle


# >> FEATURE SELECTION << #
def remove_correlated_features(X):
def remove_less_significant_features(X, Y):


# >> MODEL TRAINING << #
def compute_cost(W, X, Y):
def calculate_cost_gradient(W, X_batch, Y_batch):
def sgd(features, outputs):


def init():
```

Reading the Dataset

We'll be working with a breast cancer dataset available on Kaggle. The features in the dataset are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe the characteristics of the cell nuclei present in the image. Based on these features we will train our SVM model to detect if the mass is benign **B** (generally harmless) or malignant **M** (cancerous).

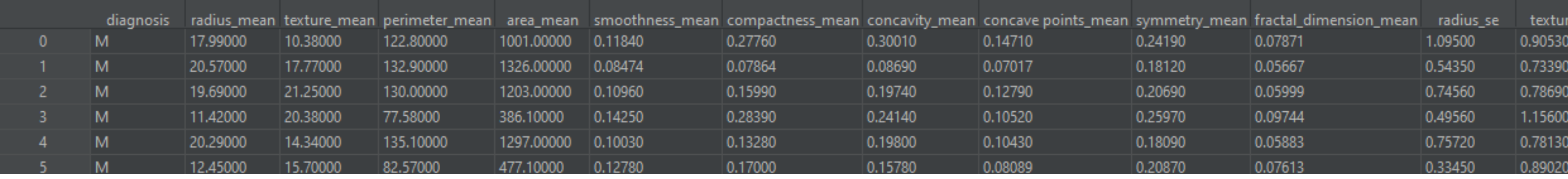
Download the dataset and place the *data.csv* file in the same folder as *svm.py*. Then add this code inside *init()* function:

```
def init():
    data = pd.read_csv('./data.csv')


    # SVM only accepts numerical values.
    # Therefore, we will transform the categories M and B into
    # values 1 and -1 (or -1 and 1), respectively.
    diagnosis_map = {'M':1, 'B':-1}
    data['diagnosis'] = data['diagnosis'].map(diagnosis_map)


    # drop last column (extra column added by pd)
    # and unnecessary first column (id)
    data.drop(data.columns[[-1, 0]], axis=1, inplace=True)
```

read_csv() function of the Pandas package reads data from the .csv file and stores it in a *DataFrame***.** Think of DataFrame as an implementation of a data structure that looks like a table with labeled columns and rows. Here's how the data read from *data.csv* looks like inside DataFrame:



	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	symmetry_mean	fractal_dimension_mean	radius_se	textu
0	M	17.99000	10.38000	122.80000	1001.00000	0.11840	0.27760	0.30010	0.14710	0.24190	0.07871	1.09500	0.9053
1	M	20.57000	17.77000	132.90000	1326.00000	0.08474	0.07864	0.08690	0.07017	0.18120	0.05667	0.54350	0.7339
2	M	19.69000	21.25000	130.00000	1203.00000	0.10960	0.15990	0.19740	0.12790	0.20690	0.05999	0.74560	0.7869
3	M	11.42000	20.38000	77.58000	386.10000	0.14250	0.28390	0.24140	0.10520	0.25970	0.09744	0.49560	1.1560
4	M	20.29000	14.34000	135.10000	1297.00000	0.10030	0.13280	0.19800	0.10430	0.18090	0.05883	0.75720	0.7813
5	M	12.45000	15.70000	82.57000	477.10000	0.12780	0.17000	0.15780	0.08089	0.20870	0.07613	0.33450	0.8902

Figure 2: A portion of breast cancer data set in a DataFrame as viewed in PyCharm IDE

As you can see the labels and structure of our dataset are preserved and that's what makes DataFrame intuitive.

Feature Engineering

Machine learning algorithms operate on a dataset that is a collection of labeled examples which consist of features and a label i.e. in our case *diagnosis* is a label, [*radius_mean*, *structure_mean*, *texture_mean*…] features, and each row is an example.

In most of the cases, the data you collect at first might be raw; its either incompatible with your model or hinders its performance. That's when **feature engineering** comes to rescue. It encompasses preprocessing techniques to compile a dataset by **extracting features from raw data.** These techniques have two characteristics in common:

- Preparing the data which is compatible with the model
- Improving the performance of the machine learning algorithm

Normalization is one of the many feature engineering techniques that we are going to use. Normalization is the process of converting a range of values, into a standard range of values, typically in the interval $[-1, 1]$ or $[0, 1]$. It's not a strict requirement but it improves the speed of learning (e.g. faster convergence in gradient descent) and prevents numerical overflow. Add following code in *init()* functions to normalize all of your features:

```
# inside init()
# put features & outputs in different DataFrames for convenience
Y = data.loc[:, 'diagnosis']  # all rows of 'diagnosis'
X = data.iloc[:, 1:]  # all rows of column 1 and ahead (features)


# normalize the features using MinMaxScalar from
# sklearn.preprocessing
X_normalized = MinMaxScaler().fit_transform(X.values)
X = pd.DataFrame(X_normalized)
```

Splitting the Dataset

We'll split the dataset into train and test set using the *train_test_split()* function from *sklearn.model_selection*. We need a separate dataset for testing because we need to see how our model will perform on unseen observations. Add this code in *init()*:

```
# inside init()

# first insert 1 in every row for intercept b
X.insert(loc=len(X.columns), column='intercept', value=1)

# test_size is the portion of data that will go into test set
# random_state is the seed used by the random number generator
print("splitting dataset into train and test sets...")
X_train, X_test, y_train, y_test = tts(X, Y, test_size=0.2,
random_state=42)
```

If you are confused about why we added 1 in every row then don't worry. You will get the answer in the next section.

Cost Function

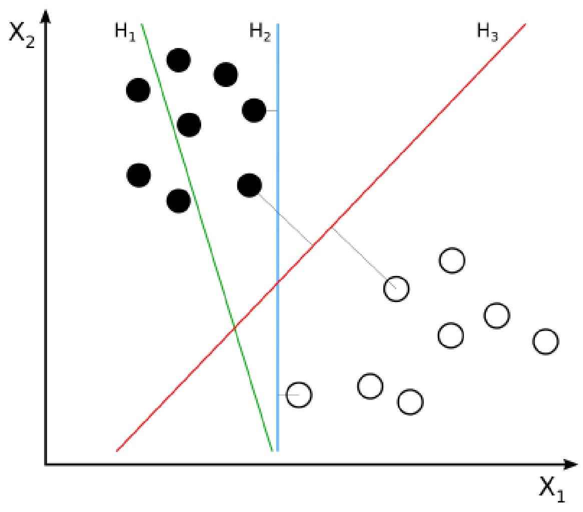


Figure 3: H1 does not separate the classes. H2 does, but only with a small margin. H3 separates them with the maximal margin — Wikipedia

Also known as the Objective Function. One of the building blocks of every machine learning algorithm, it's the function we try to *minimize or maximize* to achieve our objective.

What's our objective in SVM? Our objective is to find a hyperplane that separates +ve and -ve examples with the largest margin while keeping the misclassification as low as possible (see Figure 3).

How do we achieve this objective? We will minimize the cost/objective function shown below:

$$J(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|^2 + C\left[\frac{1}{N}\sum_{i}^{n} \max\left(0,\ 1 - y_i * (\mathbf{w} \cdot x_i + b)\right)\right] \qquad (1)$$

In the training phase, Larger C results in the narrow margin (for infinitely large C the SVM becomes hard margin) and smaller C results in the wider margin.

You might have seen another version of a cost function that looks like this:

$$J(\mathbf{w}) = \frac{\lambda}{2}\|\mathbf{w}\|^2 + \frac{1}{N}\sum_i \max\left(0,\ 1 - y_i * (\mathbf{w}\cdot x_i + b)\right) \tag{2}$$

Larger λ gives a wider margin and smaller λ results in the narrow margin (for infinitely small λ the SVM becomes hard margin).

In this cost function, λ is essentially equal to *1/C* and has the opposite effect i.e larger λ gives a wider margin and vice versa. We can use any of the above cost functions keeping in mind what each regularization parameter (C and λ) does and then tuning them accordingly. Let's see how can we calculate the total cost as given in (1) and then we will move on to its gradient which will be used in the training phase to minimize it:

```
def compute_cost(W, X, Y):
    # calculate hinge loss
    N = X.shape[0]
    distances = 1 - Y * (np.dot(X, W))
    distances[distances < 0] = 0  # equivalent to max(0, distance)
    hinge_loss = reg_strength * (np.sum(distances) / N)

    # calculate cost
    cost = 1 / 2 * np.dot(W, W) + hinge_loss
    return cost
```

As you might have noticed that the intercept term *b* is missing. That's because we have pushed it into the weight vector like this:

$$\begin{aligned} f(x_i) &= \widetilde{w}\cdot\widetilde{x_i} + w_0 = \mathbf{w}\cdot x_i \\ \textit{where}\ \mathbf{w} &= \left(\widetilde{w},\ w_0\right),\ x_i = \left(\widetilde{x_i},\ 1\right) \end{aligned} \tag{3}$$

Pushing intercept term inside weight vector.

That's why we added an extra column with all 1s before splitting our dataset. Keep this in mind for the rest of the tutorial.

The Gradient of the Cost Function

$$J(\mathbf{w}) = \frac{1}{N}\sum_i^n \left[\frac{1}{2}\|\mathbf{w}\|^2 + C\max\left(0,\ 1 - y_i * (\mathbf{w}\cdot x_i)\right)\right] \tag{4}$$

$$\nabla_w J(\mathbf{w}) = \frac{1}{N}\sum_i^n \begin{cases} \mathbf{w} & \text{if } \max\left(0,\ 1 - y_i * (\mathbf{w}\cdot x_i)\right) = 0 \\ \mathbf{w} - C y_i\, x_i & \text{otherwise} \end{cases} \tag{5}$$

The gradient of the cost function

As you would have noticed there are some changes in cost function in equation (4). Don't worry, if you solve it analytically it's the same. Now let's implement the *calculate_cost_gradient()* function using equation (5):

```
# I haven't tested it but this same function should work for
# vanilla and mini-batch gradient descent as well
def calculate_cost_gradient(W, X_batch, Y_batch):
    # if only one example is passed (eg. in case of SGD)
    if type(Y_batch) == np.float64:
        Y_batch = np.array([Y_batch])
        X_batch = np.array([X_batch])

    distance = 1 - (Y_batch * np.dot(X_batch, W))
    dw = np.zeros(len(W))

    for ind, d in enumerate(distance):
        if max(0, d) == 0:
            di = W
        else:
            di = W - (reg_strength * Y_batch[ind] * X_batch[ind])
        dw += di

    dw = dw/len(Y_batch)  # average
    return dw
```

Train Model Using SGD

Remember I said above, "To achieve our objective we try to *minimize or maximize the cost function*". In the SVM algorithm, we minimize the cost function.

Why do we minimize the cost function? Because the cost function is essentially a measure of how bad our model is doing at achieving the objective. If you look closely at J(w), to find it's minimum, we have to:

1. *Minimize* $||w||^2$ which *maximizes margin* $(2/||w||)$
2. *Minimize the sum of hinge loss* which *minimizes misclassifications*.

$$\max\left(0,\ 1 - y_i * (w \cdot x_i + b)\right) \tag{7}$$

Hinge loss function

Because both of our SVM objectives are achieved by minimizing the cost function, that's why we minimize it.

How do we minimize it? Well, there are multiple ways but the one we will use is called Stochastic Gradient Descent or SGD. Before diving into SGD, I will briefly explain how Gradient Descent works in the first place.

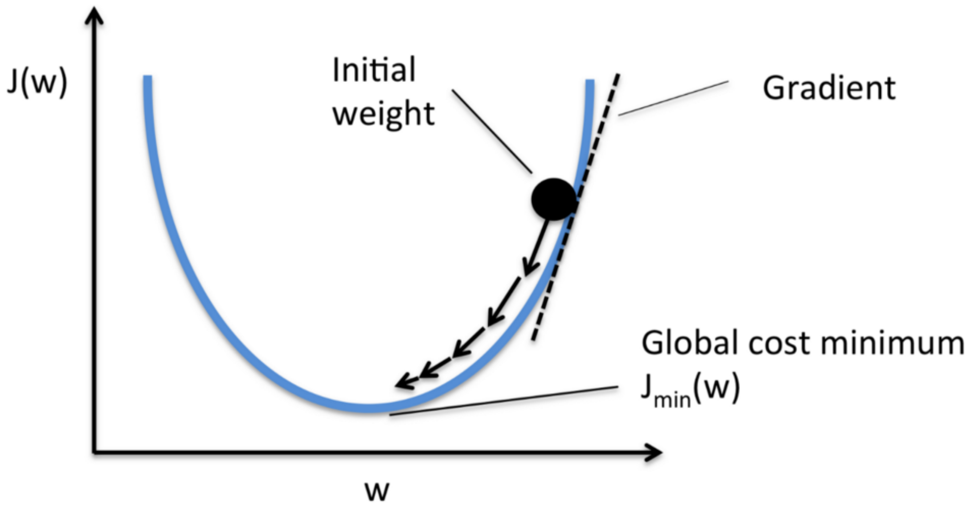


Figure 4: Gradient Descent

Gradient descent algorithm works as follows:

1. Find the gradient of cost function i.e. ∇J(w')
2. Move opposite to the gradient by a certain rate i.e. w' = w' — ∝(∇J(w'))
3. Repeat step 1–3 until convergence i.e we found w' where J(w) is smallest

Why does it move opposite to the direction of the gradient? Because gradient is the direction of the fastest increase of the function. We need to move opposite to that direction to minimize our function J(w). Hence, the word "descent" in Gradient Descent is used.

In typical gradient descent (a.k.a vanilla gradient descent) the step 1 above is calculated using all the examples (1…N). In SGD, however, only 1 example is used at a time. I won't discuss the benefits of SGD here but you can find some useful links at the end of this blog. Here's how to implement the SGD in code:

```
def sgd(features, outputs):
    max_epochs = 5000
    weights = np.zeros(features.shape[1])
    # stochastic gradient descent
    for epoch in range(1, max_epochs):
        # shuffle to prevent repeating update cycles
        X, Y = shuffle(features, outputs)
        for ind, x in enumerate(X):
            ascent = calculate_cost_gradient(weights, x, Y[ind])
            weights = weights - (learning_rate * ascent)

    return weights
```

Let's call it inside *init()* function by adding this code:

```
# inside init()

# train the model
print("training started...")
W = sgd(X_train.to_numpy(), y_train.to_numpy())
print("training finished.")
print("weights are: {}".format(W))
```

Stoppage Criterion for SGD

In the above implementation of *sgd()* we are running the loop 5000 times (could have been any number). Each iteration costs us time and extra computations. We don't need to complete all the iterations. We can terminate the loop when our stoppage criterion is met.

What should be the stoppage criterion? There are multiple options, but we will use the simplest one. We will stop the training when the current cost hasn't decreased much as compared to the previous cost. Here is how we will define *sgd()* with stoppage criterion:

```
def sgd(features, outputs):
    max_epochs = 5000
    weights = np.zeros(features.shape[1])
    nth = 0
    prev_cost = float("inf")
    cost_threshold = 0.01  # in percent
    # stochastic gradient descent
    for epoch in range(1, max_epochs):
        # shuffle to prevent repeating update cycles
        X, Y = shuffle(features, outputs)
        for ind, x in enumerate(X):
            ascent = calculate_cost_gradient(weights, x, Y[ind])
            weights = weights - (learning_rate * ascent)

        # convergence check on 2^nth epoch
        if epoch == 2 ** nth or epoch == max_epochs - 1:
            cost = compute_cost(weights, features, outputs)
            print("Epoch is:{} and Cost is: {}".format(epoch, cost))
            # stoppage criterion
            if abs(prev_cost - cost) < cost_threshold * prev_cost:
                return weights
            prev_cost = cost
            nth += 1
    return weights
```

Testing the Model

After training the model using SGD we finally got the optimal weights **w*** which defines the best possible hyperplane separating two classes. Let's test our model using this hyperplane. Add this code in *init()* function:

```
# inside init()

# testing the model on test set
y_test_predicted = np.array([])
for i in range(X_test.shape[0]):
    yp = np.sign(np.dot(W, X_test.to_numpy()[i])) #model
    y_test_predicted = np.append(y_test_predicted, yp)

print("accuracy on test dataset:
{}".format(accuracy_score(y_test.to_numpy(), y_test_predicted)))
print("recall on test dataset:
{}".format(recall_score(y_test.to_numpy(), y_test_predicted)))
print("precision on test dataset:
{}".format(recall_score(y_test.to_numpy(), y_test_predicted)))
```

Now let's call *init()* function:

```
# set hyper-parameters and call init
# hyper-parameters are normally tuned using cross-validation
# but following work good enough
reg_strength = 10000 # regularization strength
learning_rate = 0.000001
init()
```

Here’s the output:

```
# OUTPUT
reading dataset...
applying feature engineering...
splitting dataset into train and test sets...
training started...
Epoch is: 1 and Cost is: 5333.266133501857
Epoch is: 2 and Cost is: 3421.9128432834573
Epoch is: 4 and Cost is: 2437.2790231100216
Epoch is: 8 and Cost is: 1880.2998267933792
Epoch is: 16 and Cost is: 1519.5578612139725
Epoch is: 32 and Cost is: 1234.642324549297
Epoch is: 64 and Cost is: 977.3285621274708
Epoch is: 128 and Cost is: 804.8893546235923
Epoch is: 256 and Cost is: 703.407799431284
Epoch is: 512 and Cost is: 645.8275191300031
Epoch is: 1024 and Cost is: 631.6024252740094
Epoch is: 2048 and Cost is: 615.8378582171482
Epoch is: 4096 and Cost is: 605.0990964730645
Epoch is: 4999 and Cost is: 606.8186618758745
training finished.
weights are: [ 1.32516553  0.83500639  1.12489803  2.16072054
-1.24845441 -3.24246498
  3.27876342  6.83028706 -0.46562238  0.10063844  5.68881254
-1.93421932
  3.27394523  3.77331751  1.67333278 -2.43170464 -1.7683188
0.84065607
 -1.9612766  -1.84996828  2.69876618  5.32541102  1.0380137
3.0787769
  2.2140083  -0.61998182  2.66514199  0.02348447  4.64797917
2.17249278
 -9.27401088]
testing the model...
accuracy on test dataset: 0.9736842105263158
recall on test dataset: 0.9534883720930233
precision on test dataset: 0.9534883720930233
```

Feature Selection With Correlation & P-values

Feature selection encompasses statistical techniques that help in filtering irrelevant or redundant features. Correlation & p-values are among these statistical techniques. Using them we will select a subset of relevant and important features from our original set of features

What is correlation? Correlation is a degree of linear dependence (or linear relationship) between two variables. Two features are said to be correlated if the values of one feature can be explained by some linear relationship of the second feature. The degree of this relationship is given by the **correlation coefficient** (or “r”). It ranges from -1.0 to +1.0. The closer r is to +1 or -1, the more closely the two variables are related.

Why do we remove one of the correlated features? There are multiple reasons but the simplest of them is that correlated features almost have the same effect on the dependent variable. Moreover, correlated features won’t improve our model and would most probably worsen it, therefore we are better off using only one of them. After all, fewer features result in improved learning speed and a simpler model (model with fewer features).

What are p-values? It's too broad to cover in this blog. But, in the context of feature selection, p-values help us *find the features which are most significant in explaining variation in the dependent variable (y)*.

A feature with a low p-value has more significance and a feature with a high p-value has less significance in explaining the variation. Usually, we set a **Significance Level SL** (threshold) and if a feature has a p-value above this level it is discarded. I will leave some links at the end of this blog for an in-depth study on p-values.

Why do we remove features with high p-values? Because they don't tell much about the behavior of the dependent variable. So, why keep them and unnecessarily increase the complexity of our model when they are not helping us in predicting the result.

We have two functions named *remove_correlated_features()* *remove_less_significant_features()* for removing highly correlated features and less significant features (using p-values and backward elimination) respectively:

```
# >> FEATURE SELECTION << #
def remove_correlated_features(X):
    corr_threshold = 0.9
    corr = X.corr()
    drop_columns = np.full(corr.shape[0], False, dtype=bool)
    for i in range(corr.shape[0]):
        for j in range(i + 1, corr.shape[0]):
            if corr.iloc[i, j] >= corr_threshold:
                drop_columns[j] = True
    columns_dropped = X.columns[drop_columns]
    X.drop(columns_dropped, axis=1, inplace=True)
    return columns_dropped


def remove_less_significant_features(X, Y):
    sl = 0.05
    regression_ols = None
    columns_dropped = np.array([])
    for itr in range(0, len(X.columns)):
        regression_ols = sm.OLS(Y, X).fit()
        max_col = regression_ols.pvalues.idxmax()
        max_val = regression_ols.pvalues.max()
        if max_val > sl:
            X.drop(max_col, axis='columns', inplace=True)
            columns_dropped = np.append(columns_dropped, [max_col])
        else:
            break
    regression_ols.summary()
    return columns_dropped
```

Let's call these functions inside *init()* before we apply *normalization*:

```
# inside init()

# filter features
remove_correlated_features(X)
remove_less_significant_features(X, Y)
```

Rerun the code and check the output:

```
# OUTPUT WITH FEATURE SELECTION
reading dataset...
applying feature engineering...
splitting dataset into train and test sets...
training started...
Epoch is: 1 and Cost is: 7198.889722245353
Epoch is: 2 and Cost is: 6546.424590270085
Epoch is: 4 and Cost is: 5448.724593530262
Epoch is: 8 and Cost is: 3839.8660601754004
Epoch is: 16 and Cost is: 2643.2493061396613
Epoch is: 32 and Cost is: 2003.9830891013514
Epoch is: 64 and Cost is: 1595.2499320295813
Epoch is: 128 and Cost is: 1325.7502330505054
Epoch is: 256 and Cost is: 1159.7928936478063
Epoch is: 512 and Cost is: 1077.5846940303365
Epoch is: 1024 and Cost is: 1047.208390340501
Epoch is: 2048 and Cost is: 1040.2241600540974
training finished.
weights are: [ 3.53520254 11.03169318 -2.31444264 -7.89186867
10.14516174 -1.28905488
 -6.4397589   2.26113987 -3.87318997  3.23075732  4.94607957
4.81819288
 -4.72111236]
testing the model...
accuracy on test dataset: 0.9912280701754386
recall on test dataset: 0.9767441860465116
precision on test dataset: 0.9767441860465116
```

As you can see the *accuracy (99%)*, *precision (0.98)*, and *recall (0.98)* scores have improved. Moreover, SGD has converged faster; The training ended within 2048 epochs which is way less as compared to the previous one (5000 epochs)

Complete Code

You can get the complete code in this Github repository:

qandeelabbassi/python-svm-sgd

Python implementation of stochastic sub-gradient descent algorithm for SVM from scratch - qandeelabbassi/python-svm-sgd

github.com

· · ·

Useful Link

- Why SGD works?
- Benefits of SGD
- P-values explained by a data scientist
- Linear regression and p-values
- Backward elimination