Model: Support Vector Machines

Support Vector Machine (SVM) is a supervised machine learning algorithm that can be used to perform classification. The main objective of SVM is to find the best splitting boundary between data of different classes. It achieves this by finding support vectors for each class and uses these vectors to find the separating boundary. In two-dimensional space, this can be thought as the best fit line that divides a given dataset. Since SVM primarily deals with vectors, the separating vector is a hyperplane. The best separating hyperplane is defined as the hyperplane that contains the "widest" margin between support vectors. The hyperplane may also be referred to as the "decision boundary". The primary reason behind my choosing this model would be its complexity, and boy was I right! The SKlearn model will be referred to as the "Prebuilt Model" while the custom SVM model that was built ground up, will be referred to as the "Custom Model" [1]

Data preparation:

This is a common step that is performed for both, the custom and the prebuilt models. Firstly, the tab delimited data withing the "wilfires.txt" files is loaded as a pandas dataframes. This data frame is then passed onto the **prepare_data()** function which preprocesses the data with respect to its input arguments. These arguments promote for flexibility and so they can be tweaked be as per the user's requirements.

Preprocessing Arguments:

- df - The input dataframe that requires preprocessing
- label_column - The name of the label column, passed in as a string
- test_sample_ratio - The train-test split ratio. This argument specifies the test sample size
- columns_to_encode - Columns that are required one-hot encoding, passed in as list of strings
- columns_to_scale - Columns that are required to be min-max scaled, passed in as list of strings
- columns_to_znorm - Columns that are required to be z-normalized, passed in as list of strings
- columns_to_drop - Columns that are required to be dropped, passed in as list of strings

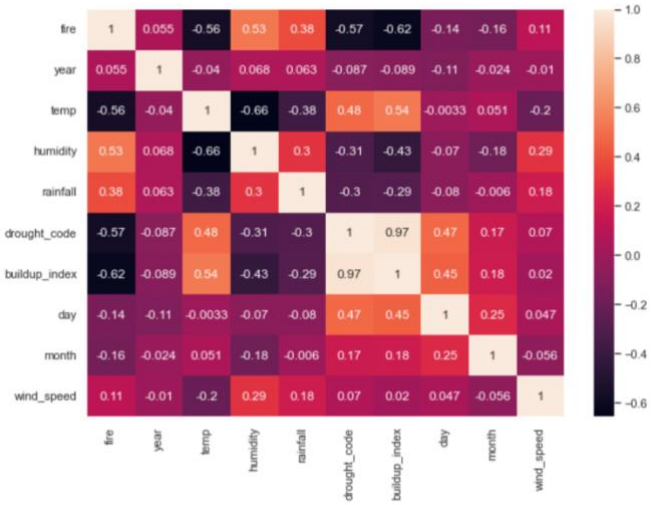


Figure 1. Heatmap of the dataset

Upon running a heatmap over the correlation of the df, we can see that drought code, temp and buildup index has no linear correlation with fire while humidity and rainfall has the most correlation. This gives us a base idea as to how to feature engineer these attributes and to extract as much information as possible.

1. All attributes were assigned their respective names. The entries in the fire column (label) were stripped of white spaces and converted to a categorical attribute. The "No" values which denoted no fire was replaced with -1 and likewise the "Yes" value which denoted fire was replaced with 1. This was done as that column had some noise in the data, and DNN models tends to understand better the numerical relationships between the various classes of an attribute.

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

$$Z = \frac{x - \mu}{\sigma}$$

Z = standard score

x = observed value

μ = mean of the sample

σ = standard deviation of the sample

2.Usually, values must be normalized or standardized before being fed into a model. This is done so that a model will give equal weightage to all features as their value range will be a lot more similar to one another. Min-Max scaler scales the values of an attribute to a range of 0 to 1. It takes the minimum and maximum value of the attribute and uses that to scale every other value in that attribute. The result will always be a value between the range of 0 to 1. This technique holds well for most use cases, but unfortunately certain attributes (like rainfall and drought_code) tend to have many outliers that cannot be dropped or replaced. Outliers tend to plague the minmax scaler, causing biased results. Enter Z normalization. This technique linearly transforms the data in such a way, that the mean value of the transformed data equals zero while their

standard deviation equals one. The transformed values themselves do not lie in a particular interval like min max scaler. [2]

3. One-hot encoding is another feature engineering technique that can allow for leveraging information from certain categorical columns. In this case the Month and Year attributes can be one-hot encoded. Again, this was done as many machine learning algorithms cannot perfectly interpret categorical data and extract underlying relationships. For instance, month attribute has values ranging from 6 to 9, although month 9 by no means is greater than month 6, they are just 2 separate unrelated values. Hence month 6 to 9 values were converted to attribute jun, Jul, aug and sep with values 0 and 1. The same procedure was repeated for year and days, even though they have -0.16 correlation with fire. Although the year and day attributes were dropped as they seemed to not have much information to offer and tends to increase processing times significantly. [3]

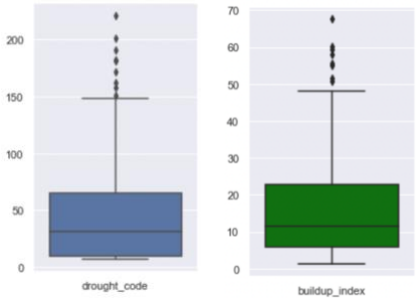


Figure 2 Boxplot of attributes with outliers

	yes	temp	humidity	rainfall	drought_code	buildup_index	wind_speed	jun	jul	aug	...	2008	2009	2010	2011	2012	2013	2014	2015	2016
189	1	0.571429	0.098592	-0.388832	0.226371	0.429634	-1.112329	0	0	0	...	0	1	0	0	0	0	0	0	0
47	1	0.666667	0.464789	-0.388832	0.394281	0.414602	-0.789548	0	0	1	...	0	0	0	1	0	0	0	0	0
165	1	0.714286	0.323944	-0.388832	1.834443	2.338074	-0.143986	0	0	1	...	0	0	0	0	1	0	0	0	0
183	-1	0.285714	0.732394	0.130536	-0.818133	-0.779796	-0.143986	0	0	0	...	0	0	0	0	0	0	0	0	0
26	1	0.523810	0.464789	-0.388832	0.659478	0.647605	0.501577	1	0	0	...	0	0	0	0	0	1	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
121	-1	0.523810	0.507042	-0.011109	-0.814470	-0.647237	0.501577	1	0	0	...	0	0	0	0	0	0	0	0	0
188	1	0.476190	0.436620	-0.388832	0.044824	0.111903	-0.466767	0	0	0	...	0	0	0	0	0	0	1	0	0
126	1	0.666667	0.197183	-0.388832	0.086140	0.749416	-0.789548	1	0	0	...	0	0	0	0	0	1	0	0	0
178	-1	0.333333	0.535211	1.546995	-0.811417	-0.828309	-1.112329	0	0	0	...	0	0	0	0	0	0	0	0	0
171	1	0.571429	0.436620	-0.152755	2.048961	0.991302	0.501577	0	0	1	...	0	0	0	0	0	0	1	0	0

204 rows × 22 columns

Figure 3 data processed using the pre-process function

The above image shows a processed dataframe that uses all possible features of the preprocessing function. The temperature and humidity attributes have been min max scaled. The rainfall, drought_code, buildup_index and wind_speed attributes have been z normalized. Year and Month attributes have been one-hot encoded. The Day

```
cols_to_scale  = ['temp', 'humidity', "rainfall", "drought_code", "buildup_index", "wind_speed"]
cols_to_znorm  = ["rainfall", "drought_code", "buildup_index", "wind_speed"]
cols_to_encode = {"month" : {6:'jun', 7:'jul', 8:'aug', 9:'sep'},
                  "year"  : [2007,2008,2009,2010,2011,2012,2013,2014,2015,2016,2017]}

dataset = pd.read_csv("wildfires.txt", sep="\t")

train_set, test_set, df = prepare_data(dataset, label_column="yes",  test_sample_ratio=0.33,
                                       cols_to_encode, cols_to_scale, cols_to_znorm,
                                       columns_to_drop = ["day", "year"])

```

attributes have been dropped. The **prepare_data()** function is primarily designed to promote reusability and ease of use. The above mentioned processed dataframe can be easily achieved with only a few lines of code.

However, for development and testing purposes, I have preprocessed my data as follows:

- Encoded my output attribute to 1s and 0s
- Min max scaled all primary attributes
- One hot encoded the month attribute
- Dropped year and day attributes
- Shuffled and split the data in a 0.6 : 0.3 ratio

	yes	temp	humidity	rainfall	drought_code	buildup_index	wind_speed	jun	jul	aug	sep
157	1	0.571429	0.338028	0.017857	0.219965	0.271098	0.333333	0	0	1	0
54	1	0.666667	0.436620	0.000000	0.438810	0.330396	0.583333	0	0	1	0
185	1	0.428571	0.690141	0.000000	0.086567	0.123376	0.333333	0	0	0	1
115	-1	0.380952	0.704225	0.279762	0.004482	0.039134	0.458333	1	0	0	0
7	1	0.333333	0.718310	0.000000	0.146986	0.194025	0.458333	1	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...
168	1	0.523810	0.239437	0.000000	0.750619	0.938163	0.500000	0	0	1	0
3	-1	0.142857	0.929577	0.148810	0.000000	0.016430	0.375000	1	0	0	0
195	1	0.571429	0.619718	0.000000	0.462997	0.464675	0.375000	0	0	0	1
131	1	0.619048	0.492958	0.005952	0.094037	0.136669	0.458333	0	1	0	0
162	1	0.619048	0.830986	0.000000	0.466732	0.551307	0.416667	0	0	1	0

204 rows × 11 columns

Figure 4 Processed data which was used to train the models

Model Description:

The idea behind SVM is to take known data and to find the "best separating" line for the data, known as the decision boundary.

For the sake of simplicity, I have devised a simple example is in 2D space (i.e. I am only considering two features). The separating hyperplane (decision boundary) is a simple straight line int the center. This decision boundary is separating the blue diamond group from the green circle group. To predict the class of a unknown value, all we'll have to do is to plot its features on the graph, we'd just do a simple check to see which side of the separating hyperplane it was on, and just like that we have our answer.

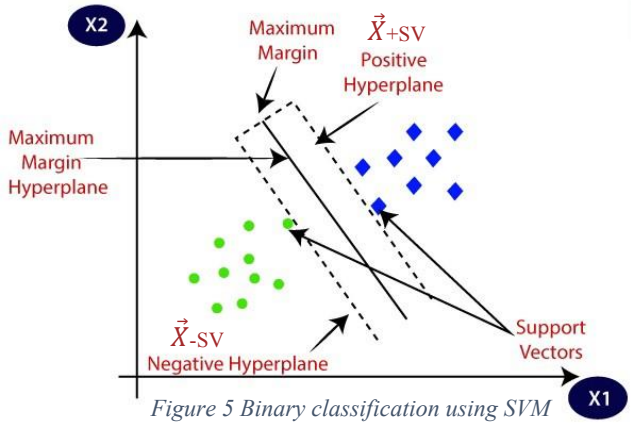
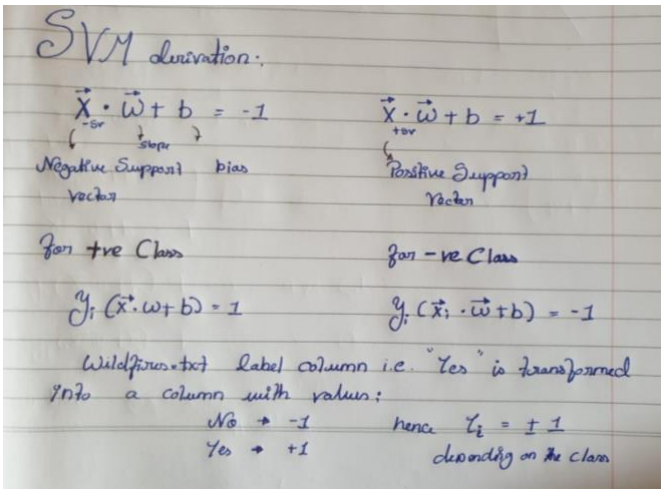
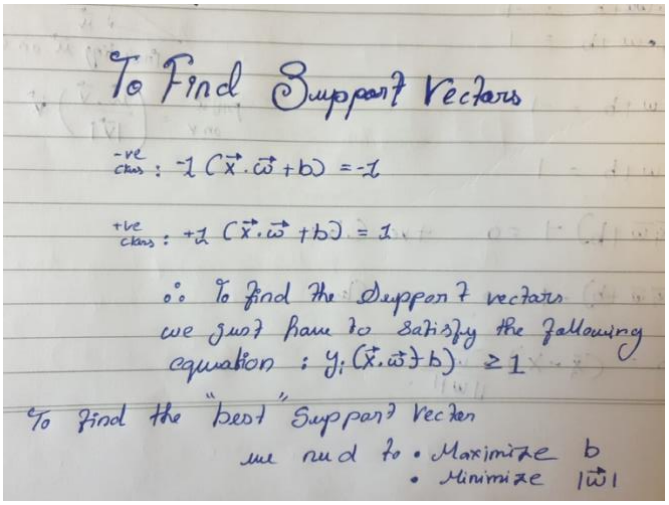


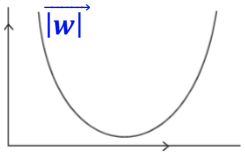
Figure 5 Binary classification using SVM

In our dataset (wildfires.txt), each feature is considered to be a dimension. Thus, a simple decision boundary line in this case becomes a complex n-dimensional hyperplane, in this case, to predict the class of an unknown sample, we take datapoints as a vector $\overrightarrow{u}$, and then we project that onto the vector pointing perpendicular to the hyperplane $\overrightarrow{w}$. Depending on scalar value we obtain from the projection, we can identify the class of the given unknown sample.



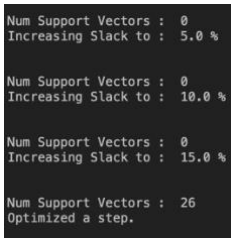


As we can see from the derivation, to find support vectors, we would have to find values for $\overrightarrow{w}$ and b such that satisfies all samples in the input dataset. And to find the best possible support vectors on each side, we will have to **maximize b** while **minimizing** $|\overrightarrow{w}|$ which makes this an **optimization problem.**



The SVM's optimization problem is a convex problem, where the convex shape is the magnitude of vector w. Our objective here is to find the global minimum (which thankfully in this case is the local minimum). One commonly used iterative optimization algorithm would be Gradient descent. But in this case, I have chosen to take a more workaday approach. The custom model will try to optimize for this by "stepping down" approach. The model will start by taking large step, quickly climbing down. Once it finds the bottom and crosses it, the model will stop and go back the other way, only this time it reduces the size of the steps. It keeps repeating this until it finds the absolute bottom. This is like dropping a ball down the inside edge of the bowl. Though, this is a convex problem, the plan is to give us a vector, and slowly step down the magnitude of the vector For each vector, example [10,10], we can transform that vector by the following transformations (1, 1), (-1, 1), (-1, -1), and (1, -1). This will give us all the variances of that vector that we need to check for, despite them all having the same magnitude. Until this point, everything seems to be straight forward. But would this model scale for a real-world dataset like wildfires? Unfortunately, no! A model build on the above constraints alone will be able to classify perfectly separated data. Should there be any outliers, the model would instantly fail to find support vectors as shown [4].
Enter, Soft Margin SVM**.**

Figure 6 Hard margin SVM



Soft Margin SVM:

This idea behind Soft Margin SVM is simple: It allow SVM to make a certain number of mistakes and tries to keep the margin as wide as possible so that other points can still be classified correctly. The custom model achieves this using something called the "slack". Slack specifies the percentage of the dataset that can be misclassified. The Slack variable

coded into the model starts of at 0, that is it does not allow for any misclassification. On not finding sufficient support vectors, that is more than 5 in this case, the model increases the slack percentage in steps. Slack_step is a variable that allows the user to define the number of steps in which the slack can be increased when the model fails to find any support vectors. [4]

Implementation:

- The fit method trains the model with the given samples and tries to find the best possible w and b values (slope and bias of the hyperplane)
- The search for w starts from: maximum attribute value across the entire dataset * 10
- latest_optimum houses the magnitude of w at each step
- The size of the step to be taken each iteration for w is stored as a list of float values.
- The slack value always starts of at 0. This means that the model will start of as a Hard Margin SVM and will try to compute a boundary margin that perfectly separates the dataset. If it fails to find any support vectors, it will immediately increase the slack by a user defined margin and will try again. By default, the model will try to find at least 5 support vectors. This is done for the custom cost function (discussed below).
- The model begins optimization by taking huge steps for w. w is nothing but the latest_optimum in the form of a (1 x number of features) matrix. This is done for the sake of dot producing w with each sample. Similarly, the bias value is also taken in slopes
- The local_optimum in w is transformed in accordance with the transformation matrix
- Each sample provided within the train dataset is fitted with these values of w and b and checked if the make up a suitable support vector
- Depending on its classification performance and the slack value at that point, the support vector is either saved or discarded
- This process is repeated until the latest_optimum value reaches or crosses 0
- If the model does not find enough support vectors, it increases the slack by a certain percentage and retries the entire process, resetting latest_optimum to max_attr_value * 10
- Should the model find enough support vectors, the model sends these into the custom cost function I have devised.
- The cost function finds the best possible support vector among them.
- the latest_optimum is updated to: latest_optimum used to find the best possible support vector at this point + a step size times 2. What this does is to change the latest_optimum a value that is few steps behind the best possible support vector, so that the model can start traversing in smaller steps.
- Upon optimizing for all steps and finding all possible support vectors the best fit support vector is found by the cost function

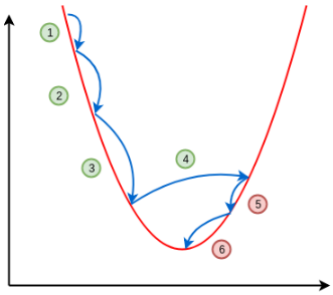


Figure 7 Convex Optimization

Cost function:

Typically, a SVM model would try and compute the **hinge loss** for each of the selected support vectors. Based on this loss value and $\overrightarrow{|w|}$, Soft margin SVM would choose the best support vector among them. Hinge loss punishes the model based on the magnitude of misclassification, there by producing a superior model. But in this case, I have used a simple and modest approach:

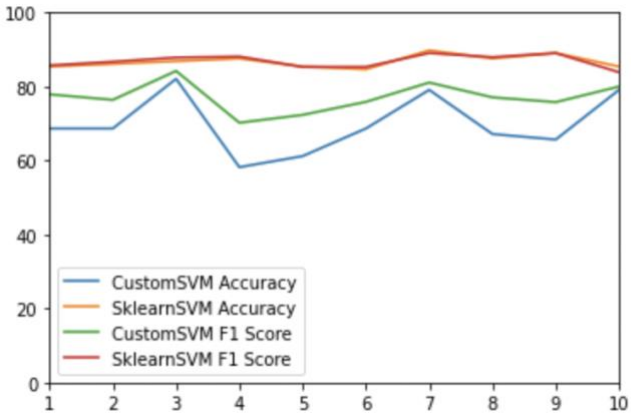
- The custom model would find all possible support vectors at each step
- With these support vector, it would try to classify all the training samples and would count the misclassification of each support vector.
- This loss value and the magnitude value of each vector is normalized for all selected support vectors
- These values are then added with each other as shown
- The α parameter lets the user adjust the accuracy / margin width tradeoff for the model
- A high α parameter would allow the model to pick a support vector high margin distance whereas a low α parameter would result in the model picking a support vector primarily focuses on the accuracy of the support vector rather than its inter marginal distance.

$$\frac{loss_i - \min(loss)}{\max(loss) - \min(loss)} + \alpha \frac{\overrightarrow{|w|}_i - \min(\overrightarrow{|w|})}{\max(\overrightarrow{|w|}) - \min(\overrightarrow{|w|})}$$

loss = loss calculate per support vector
α = (loss / margin) width trade of
w = margin width

Performance:

To compare the performance of the models, 2 evaluation metrics are used. We can see that the accuracy is fluctuating between 60% and 89% for the CustomSVM model whereas the Prebuilt Model has a stable 83% to 91%. As a matter of fact, the f1 scores for the CustomSVM model seem to be better in comparison with the Accuracy metric. The CustomSVM model seems to be holding its ground to a certain extent when it comes to evaluation metrics. But when it comes to the time taken to training the model



	CustomSVM Accuracy	SklearnSVM Accuracy	CustomSVM F1 Score	SklearnSVM F1 Score
1	68.657	85.401460	77.895	85.714286
2	68.657	86.131387	76.404	86.713287
3	82.090	86.861314	84.211	87.837838
4	58.209	87.591241	70.213	88.111888
5	61.194	85.401460	72.340	85.294118
6	68.657	84.671533	75.862	85.314685
7	79.104	89.781022	81.081	89.062500
8	67.164	87.591241	77.083	87.943262
9	65.672	89.051095	75.789	89.051095
10	79.104	85.401460	80.000	83.870968
11	58.209	89.051095	66.667	90.445860

```
f1 Score      : 66.667
Precision     : 50.0
Recall        : 100.0
```

Conclusion:

Is the Custom Model better than the prebuilt model? **Obviously NOT!** The Custom model is slower and less accurate in comparison to its counterpart. The primary purpose behind building this model was to show and understand the underlying workings of SVM and its ability to finding the best separating hyperplane to classify the dataset. SVM seems to have been a good choice for this particular dataset. The accuracy seems to have fluctuated between lower 60% to upper the upper 80% for the custom model and has been a consistent 85% to 90% for the predefined model. A better cost function should be able to boost the accuracy of the custom model, but what really plagues this model is its optimization function and the time it takes to find support vectors. Lastly, the preprocessing of the data worked wonders in achieving higher accuracy. Removing unnecessary attributes and using two types of normalization techniques, boosted the accuracy by at least 20% on both models.

Improvements:

The following are some ideas that I am currently working or planning to add to this model

- Better optimization techniques like Gradient descent, options to choose from different optimization techniques. This would surely help with cutting down on training times
- Multithreading the Optimization step
- Implementation of different Kernels and the ability to switch between them

Bibliography:

[1] C. P. Diehl and G. Cauwenberghs, "SVM incremental learning, adaptation and optimization," *Proceedings of the International Joint Conference on Neural Networks, 2003.*, 2003, pp. 2685-2690 vol.4, doi: 10.1109/IJCNN.2003.1223991

[2] Minmax Scaler: https://developers.google.com/machine-learning/data-prep/transform/normalization

[3] One hot encoding: https://www.researchgate.net/publication/320465713_A_Comparative_Study_of_Categorical_Variable_Encoding_Techniques_for_Neural_Network_Classifiers

[4] Soft margin: https://towardsdatascience.com/support-vector-machines-soft-margin-formulation-and-kernel-trick-4c9729dc8efe

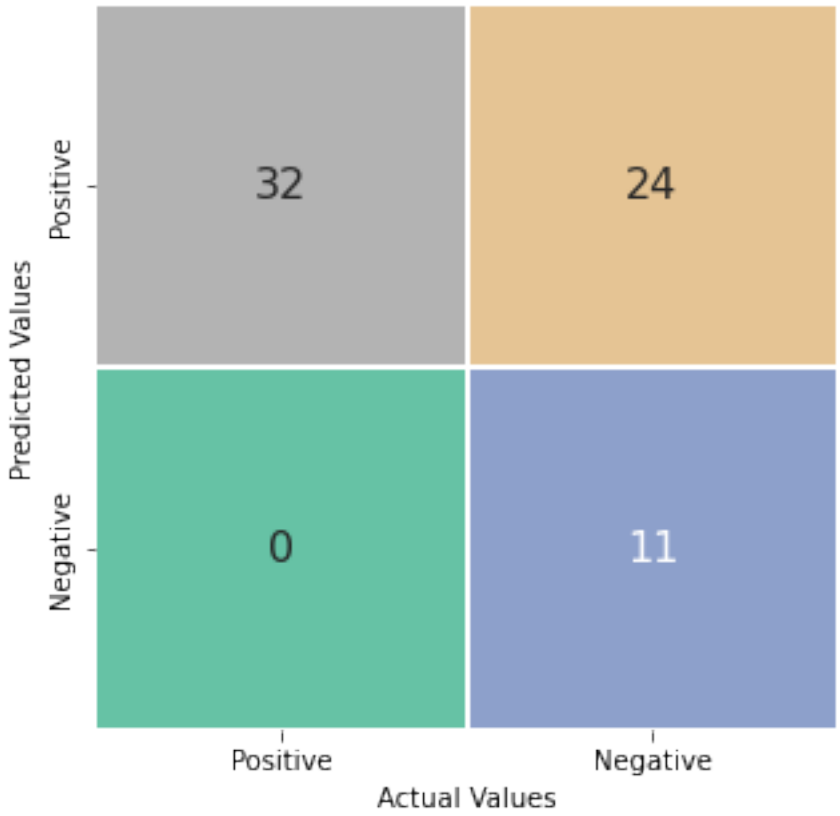
[5] https://www.mltut.com/svm-implementation-in-python-from-scratch/

```
    clf = svm.SVC(kernel='linear')
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print("Accuracy SKlearn:", metrics.accuracy_score(y_test, y_pred) * 100)

    accuracy_dict.update({itr+1 : (customSvm.accuracy , metrics.
↪accuracy_score(y_test, y_pred) * 100,
                                   customSvm.f1_score , metrics.
↪f1_score(y_test, y_pred) * 100)})
```

```
Training time : 13.02
Acurracy      : 68.657
f1 Score      : 77.419
Precision     : 63.158
Recall        : 100.0
Model saved!

Accuracy SKlearn: 89.78102189781022


Training time : 13.06
Acurracy      : 56.716
f1 Score      : 67.416
Precision     : 50.847
Recall        : 100.0
Model saved!

Accuracy SKlearn: 85.40145985401459


Training time : 55.63
Acurracy      : 82.09
f1 Score      : 86.364
Precision     : 79.167
Recall        : 95.0
Model saved!

Accuracy SKlearn: 86.86131386861314


Training time : 13.03
Acurracy      : 64.179
f1 Score      : 73.333
```

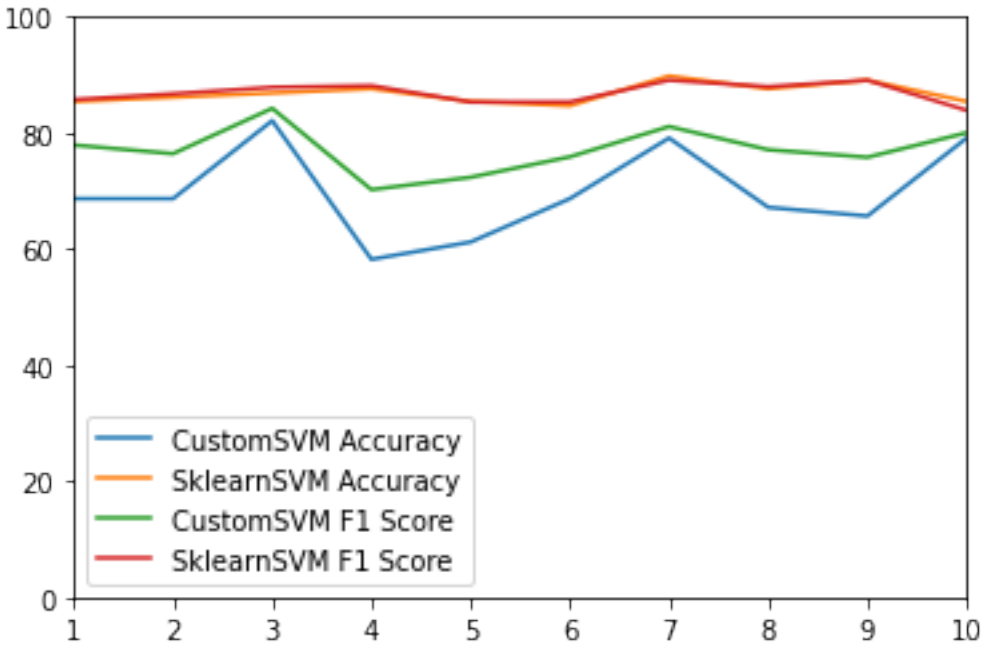
```
Precision     : 57.895
Recall        : 100.0
Model saved!

Accuracy SKlearn: 87.59124087591242


Training time : 14.21
Acurracy      : 59.701
f1 Score      : 69.663
Precision     : 53.448
Recall        : 100.0
Model saved!

Accuracy SKlearn: 91.97080291970804


Training time : 11.99
Acurracy      : 65.672
f1 Score      : 75.789
Precision     : 61.017
Recall        : 100.0
Model saved!

Accuracy SKlearn: 88.32116788321169


Training time : 60.82
Acurracy      : 83.582
f1 Score      : 86.42
Precision     : 87.5
Recall        : 85.366
Model saved!

Accuracy SKlearn: 89.05109489051095


Training time : 12.6
Acurracy      : 64.179
f1 Score      : 73.913
Precision     : 58.621
Recall        : 100.0
Model saved!

Accuracy SKlearn: 85.40145985401459


Training time : 13.36
```

```
Acurracy      : 64.179
f1 Score      : 71.429
Precision     : 55.556
Recall        : 100.0
Model saved!

Accuracy SKlearn: 92.7007299270073


Training time : 13.46
Acurracy      : 62.687
f1 Score      : 72.527
Precision     : 56.897
Recall        : 100.0
Model saved!

Accuracy SKlearn: 87.59124087591242


Training time : 13.42
Acurracy      : 64.179
f1 Score      : 72.727
Precision     : 57.143
Recall        : 100.0
Model saved!

Accuracy SKlearn: 88.32116788321169
```



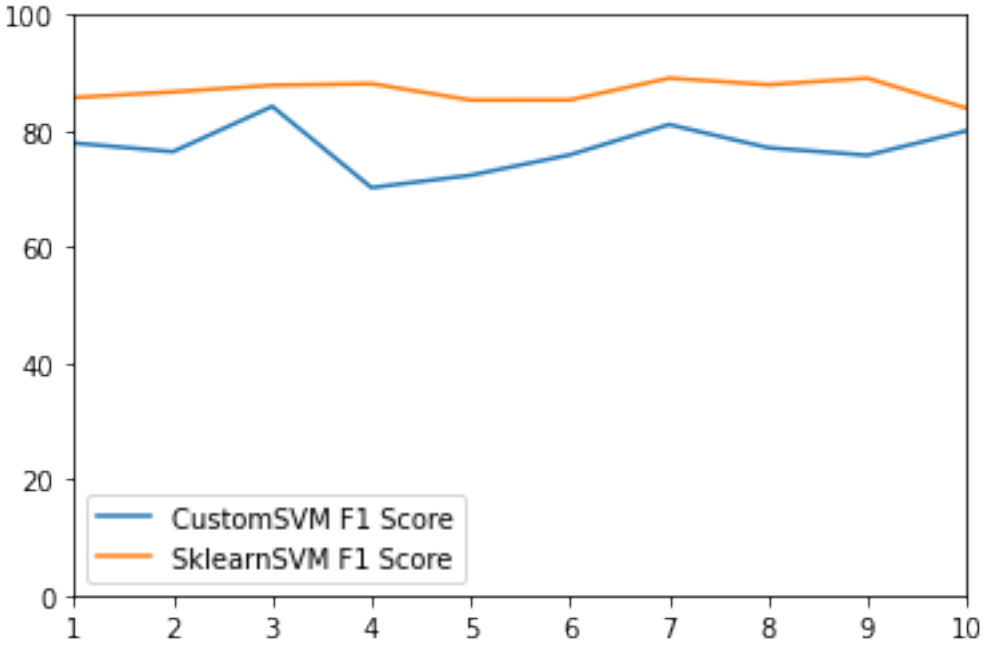
[11]:
```
acc_df  = pd.DataFrame.from_dict(accuracy_dict).transpose()
acc_df.columns = ['CustomSVM Accuracy', 'SklearnSVM Accuracy', 'CustomSVM F1␣
 ↪Score', 'SklearnSVM F1 Score']
acc_df.plot(ylim=(0,100), xlim=(1, 10))
```

[11]: <AxesSubplot:>



[12]:
```
acc_df[['CustomSVM F1 Score', 'SklearnSVM F1 Score']].plot(ylim=(0,100),␣
↪xlim=(1, 10))
```

[12]: <AxesSubplot:>



[13]:
```
acc_df
```

[13]:

	CustomSVM Accuracy	SklearnSVM Accuracy	CustomSVM F1 Score	SklearnSVM F1 Score
1	68.657	85.401460	77.895	85.714286
2	68.657	86.131387	76.404	86.713287
3	82.090	86.861314	84.211	87.837838
4	58.209	87.591241	70.213	88.111888
5	61.194	85.401460	72.340	85.294118
6	68.657	84.671533	75.862	85.314685
7	79.104	89.781022	81.081	89.062500
8	67.164	87.591241	77.083	87.943262
9	65.672	89.051095	75.789	89.051095
10	79.104	85.401460	80.000	83.870968
11	58.209	89.051095	66.667	90.445860