Support Vector Machines

Implementation

The implementation includes 3 files:

- svm.py: the main file that runs the models based on given arguments.
- models.py: the file that includes the primal SVM and dual formulation implementations.
- extract_features.py: the file that contains the sift feature extractor. It can also be run separately to save the features.

To run the svm:

- **python svm.py [options]**

All options can be seen by executing,

- **python svm.py -h**

Options include the following,

1. **data_location:** if the data is going to be used locally, the path to the data must be given using this option. The directory must include the files with names x_train.npy, y_train.npy, x_test.npy and y_test.npy.
2. **use_tf:** if given, the code uses tensorflow to download the data
3. **use_sift:** if given sift features will be extracted from the data and they will be used as input to the models
4. **num_dim:** if sift features are going to be used, this option can be used to specify the number of dimensions the features will have.
5. **C:** used to set the regularization parameter.
6. **gamma:** used to set the gamma of sklearn's SVM with rdf kernel.
7. **use_sklearn:** if given, scikit-learn models will be used
8. **nonlinear:** if given, nonlinear models will be used otherwise linear
9. **grid_search:** if given, a grid search will be performed to find the best hyperparameters
10. **plot:** if given, the support vectors of the dual formulation will be plotted

Please keep in mind that not all options work with each other for example use_tf overrides data_location and grid_search can not be used without use_sklearn.

The feature extractor can be run separately using the following,

- **python extract_features.py [options]**

Similarly, -h can be used to get all options.

They include the following,

1. **data_location:** if the data is going to be used locally, the path to the data must be given using this option. The directory must include the files with names x_train.npy, y_train.npy, x_test.npy and y_test.npy.
2. **use_tf:** if given, the code uses tensorflow to download the data
3. **extract_location:** specifies the path the features will be extracted to

4. **num_dim:** this option can be used to specify the number of dimensions the features will have.

1) Flattening the Images and Using Them Directly

a. Primal 4-class linear SVM with QP

I used the Hard margin primal formulation given in the slides and I did not use slack variable. Thus, there was no hyperparameter to tune.

The parameters of CVXOPT's QP solver are the following,

$$\begin{array}{ll} \text{minimize} & (1/2)x^T Px + q^T x \\ \text{subject to} & Gx \preceq h \\ & Ax = b \end{array}$$

And the primal formulation in the slides is the following,

$$\begin{aligned} \min_{\mathbf{u}\in\mathbb{R}^q} \quad & \frac{1}{2}\mathbf{u}^T Q\mathbf{u} + \mathbf{p}^T\mathbf{u} \\ \text{subject to:} \quad & A\mathbf{u} \geq \mathbf{c} \end{aligned}$$

Since the primal formulation does not include the last equality in the solver's equations, I did not use A and b.

The parameters given to the QP solver are the following,

- **P**: Q matrix in the slides,

$$\begin{bmatrix} 0 & \mathbf{0}_d^T \\ \mathbf{0}_d & I_d \end{bmatrix}$$

- **q**: p in the slides, 0 vector with length (number_of_features + 1)

$$\mathbf{p} = \mathbf{0}_{d+1}$$

- **G**: I multiplied the inequality the system is subject to with -1 to make the inequality the same with the solver. So G correspond to -A in the primal formulation in the slides.

$$-\begin{bmatrix} y_1 & y_1\mathbf{x}_1^T \\ \vdots & \vdots \\ y_N & y_N\mathbf{x}_N^T \end{bmatrix}$$

- **h**: Similar to G, h corresponds to -c in the primal formulation. Vector of -1s with length equal to sample size.

After training it with the whole training data, I was able to achieve a test accuracy of **0.9083**.

b. Scikit-learn 4-class linear SVM

In this part, I used Scikit-learn package's SVC class with a Linear Kernel. There is another class called LinearSVC and it works faster, but according to the online sources I checked (a.k.a stackoverflow) the solver it uses is not an SVM.

I run trials with C values of 0.1, 1, 10 and 100. The training and test accuracies are,

- Accuracy for C=0.1: Train = 0.9691, Test = 0.9600
- Accuracy for C=1: Train = 0.9750, Test = 0.9573
- Accuracy for C=10: Train = 0.9797, Test = 0.9493
- Accuracy for C=100: Train = 0.9841, Test = 0.9434

Best test accuracy: 0.9600 achieved with C=0.1 which is higher than the accuracy my implementation could achieve. I believe this is because I used hard margin while the scikit-learn's solver uses a soft margin approach.

c. 4-class Non-Linear SVM with Dual Formulation

In this part, I used the dual formulation in the slides.

The parameters of CVXOPT's QP solver are the following,

$$\begin{array}{ll} \text{minimize} & (1/2)x^T P x + q^T x \\ \text{subject to} & Gx \preceq h \\ & Ax = b \end{array}$$

For this section, all three components of the system are required. Following the formulation in the slides,

$$\max_{\alpha \in \mathbb{R}^N} \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{m=1}^{N} \sum_{n=1}^{N} \alpha_n \alpha_m y_n y_m \Phi(\mathbf{x}_n)^T \Phi(\mathbf{x}_m)$$

$$\text{subject to } \sum_{n=1}^{N} y_n \alpha_n = 0, \alpha_n \geq 0, \forall n$$

I transformed the maximization of the formulation to minimization by multiplying it with -1 to make it similar to QP solver.

$$\max_{\alpha \in \mathbb{R}^N} \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{m=1}^{N} \sum_{n=1}^{N} \alpha_n \alpha_m y_n y_m \Phi(\mathbf{x}_n)^T \Phi(\mathbf{x}_m)$$

To transform the above formulation,

- **q:** a vector of -1s with size equal to sample size.
- **P:** for each element of this matrix $P[i,j] = y_i y_j \phi(x_i)^T \phi(x_j)$ where ϕ represents the gaussian kernel.

$$\sum_{n=1}^{N} y_n \alpha_n = 0,$$

To convert the equality above,

- **A:** $A = y^T$
- **b:** is just 0.

$$\alpha_n \geq 0, \forall n$$

Lastly, to convert the inequality above,

- **G:** $G = -I$ where I represents the identity matrix
- **h:** a vector of 0s with size equal to sample size

The solver returns the Lagrange multipliers. Then I combed out the non-positive ones and used the following formula to calculate the bias terms

$$b^* = y_s - \sum_{n=1}^{N} y_n \alpha_n^* \mathbf{x}_n^T \mathbf{x}_s$$

Finally, the predictions are made using the following,

$$\text{sign}\left(\sum_{\alpha_n^* > 0} y_n \alpha_n^* \mathbf{x}_n^T \mathbf{x} + b^* \right)$$

Using the code I was able to achieve a test accuracy of **0.9915.**

d. Scikit-learn 4-class Non-Linear SVM

In this part, I again used the SVC function, but this time with RBF kernel. In the implementation, I used a exhaustive grid search with a cross validation of 5. I included C and gamma as hyperparameters (for C: 0.1, 1, 10, 100 and for gamma: 0.001, 0.01, 0.1, 1). The grid search took too long and after it I couldn't get the results for each parameter combination, but here is the best results:

- Best hyperparameters: C=10, gamma=0.01
- Train accuracy: 0.9999 Test accuracy: 0.9898

2) Using Feature Extraction

I used SIFT to extract features and formed a Bag of Words to transform the data. I first extracted the features using openCV's extractor, then used scikit-learn's k-means function to find the feature centers, then formed normalized histograms for each data sample. I experimented with 16, 32 and 64 centers and compared their results.

a. Primal 4-class linear SVM with QP

The parameters I gave to the solver are similar to section 1. The only difference is instead of flattened images, the histograms are used as input to the system. Here are the results for different number of feature centers,

- 16 centers : Accuracy for : Train = 0.1853, Test = 0.1801
- 32 centers : Accuracy for : Train = 0.5432, Test = 0.5401
- 64 centers : Accuracy for : Train = 0.6575, Test = 0.6621

As one can see, as the number of centers increases so does the accuracy of the model.

b. Scikit-learn 4-class linear SVM

The procedure is the same as in section 1. Here are the results,

- 16 centers
  - Accuracy for C=0.1: Train = 0.5975, Test = 0.589
  - Accuracy for C=1: Train = 0.6004, Test = 0.5923
  - Accuracy for C=10: Train = 0.6007, Test = 0.5935
  - Accuracy for C=100: Train = 0.6005, Test = 0.5938
  - Best test accuracy: 0.5938 achieved with C=100
- 32 centers
  - Accuracy for C=0.1: Train = 0.7607, Test = 0.7670
  - Accuracy for C=1: Train = 0.7696, Test = 0.7764
  - Accuracy for C=10: Train = 0.7715, Test = 0.7769
  - Accuracy for C=100: Train = 0.7717, Test = 0.7764
  - Best test accuracy: 0.7769 achieved with C=10
- 64 centers
  - Accuracy for C=0.1: Train = 0.8357, Test = 0.8489
  - Accuracy for C=1: Train = 0.8491, Test = 0.8527
  - Accuracy for C=10: Train = 0.8522, Test = 0.8554

- Accuracy for C=100: Train = 0.8526, Test = 0.8537
- Best test accuracy: 0.8554 achieved with C=10

The results show that as the number of centers increases the accuracy of the system also increases.

c. 4-class Non-Linear SVM with Dual Formulation

I used the same model in the first section. Here are the results,

- 16 centers: Accuracy for : Train = 0.8470, Test = 0.4750
- 32 centers: Accuracy for : Train = 0.9930, Test = 0.6991
- 64 centers: Accuracy for : Train = 0.9880, Test = 0.8092

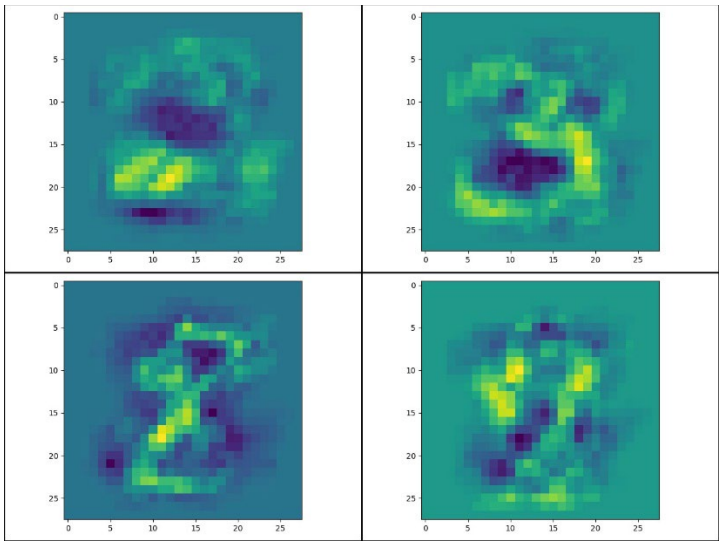
d. Scikit-learn 4-class Non-Linear SVM

The procedure and the limitations are the same as in section 1. Here are the results,

- 16 centers
  - Train accuracy: 0.6803 16 Test accuracy: 0.6504
  - Best hyperparameters: C=100, gamma=1
- 32 centers
  - Train accuracy: 0.8185 32 Test accuracy: 0.8099
  - Best hyperparameters: C=10, gamma=1
- 64 centers
  - Train accuracy: 0.8917 64 Test accuracy: 0.8713
  - Best hyperparameters: C=10, gamma=1

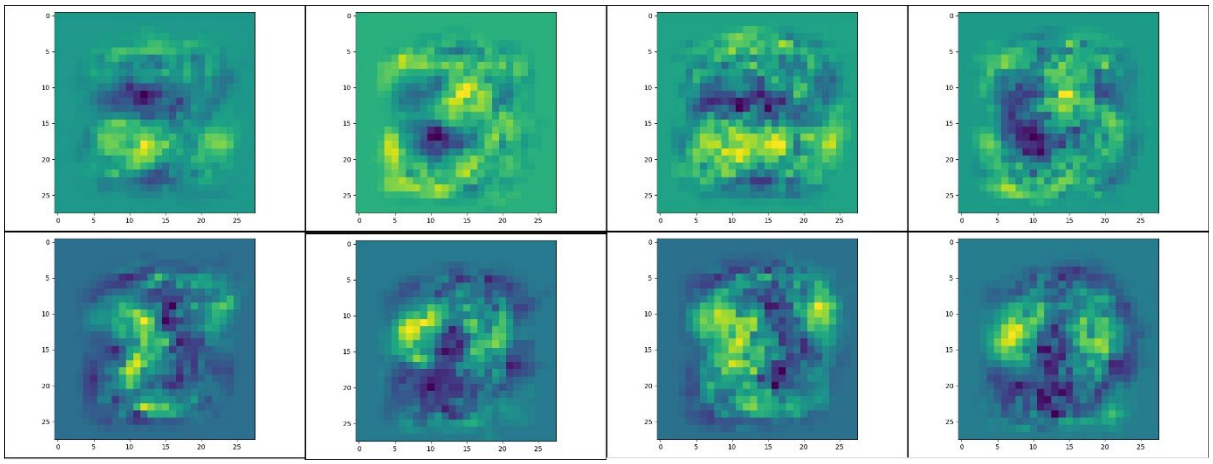
3) Support Vectors of Dual Formulation

In this part, I plotted the support vectors of the models trained by different sized training datasets.



The support vectors of a model trained using a sample size of 100 can be seen above. 100 may not sound like much, but the system still can achieve around %85 accuracy with the full test set of sample size around 4000. The top left figure is the support vector that separates 2 from the rest, the top right figure is the support vector that separates 3s from the rest, the bottom right

figure is the support vector that separates 9s from the rest and the bottom left figure is the support vector that separates 8s from the rest. As can be seen in the figures, they look a lot like the elements they separate.



You can see the support vectors when the model is trained with training datasets of size 1000 and 5000. As seen from the figures, as the training data increases, the figures become to look a lot less like the characters they separate. I believe this stems from the noise that exists in the data. Each character in the same class is slightly different from each other and as the data increases, the system also learns these intraclass differences in addition to the interclass differences.