

MAJOR PROJECT

Name : Dharshanala Naveen

Batch : July ML 1

Mentor : Liqzan Manna

Digit-recognition-using-SVM

Objective:

We will develop a model using Support Vector Machine which should correctly classify the handwritten digits from 0-9 based on the pixel values given as features. Thus, this is a 10-class classification problem.

Data Description

For this problem, we use the MNIST data which is a large database of handwritten digits. The 'pixel values' of each digit (image) comprise the features, and the actual number between 0-9 is the label.

Since each image is of 28 x 28 pixels, and each pixel forms a feature, there are 784 features. MNIST digit recognition is a well-studied problem in the ML community, and people have trained numerous models (Neural Networks, SVMs, boosted trees etc.) achieving error rates as low as 0.23% (i.e. accuracy = 99.77%, with a convolutional neural network).

Before the popularity of neural networks, though, models such as SVMs and boosted trees were the state-of-the-art in such problems. We'll first explore the dataset a bit, prepare it (scale etc.) and then experiment with linear and non-linear SVMs with various hyperparameters.

We'll divide the analysis into the following parts:

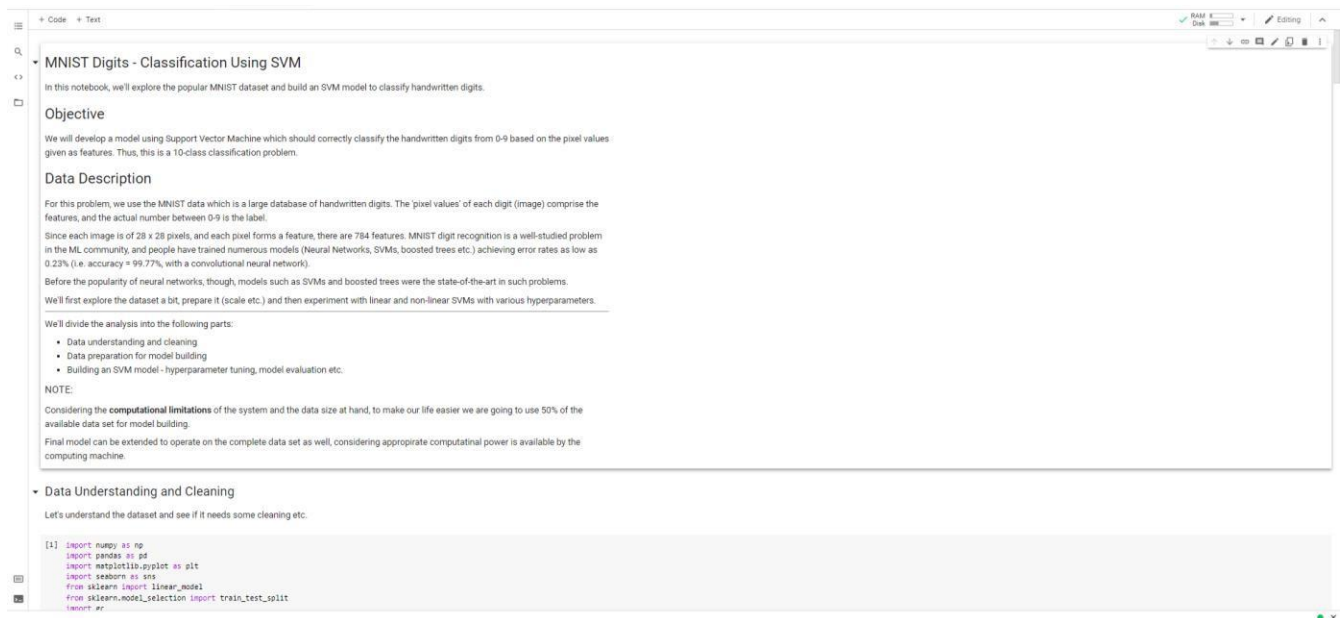
- Data understanding and cleaning
- Data preparation for model building
- Building an SVM model - hyperparameter tuning, model evaluation etc.

NOTE:

Considering the computational limitations of the system and the data size at hand, to make our life easier we are going to use 50% of the available data set for model building.

Final model can be extended to operate on the complete data set as well, considering appropriate computational power is available by the computing machine.

CODE:



The screenshot shows a Jupyter Notebook interface with a title bar containing '+ Code' and '+ Text' tabs. The notebook title is 'MNIST Digits - Classification Using SVM'. The content includes an introduction, an objective, a data description, a note about computational limitations, and a section for data understanding and cleaning with associated Python code.

MNIST Digits - Classification Using SVM

In this notebook, we'll explore the popular MNIST dataset and build an SVM model to classify handwritten digits.

Objective

We will develop a model using Support Vector Machine which should correctly classify the handwritten digits from 0-9 based on the pixel values given as features. Thus, this is a 10-class classification problem.

Data Description

For this problem, we use the MNIST data which is a large database of handwritten digits. The 'pixel values' of each digit (image) comprise the features, and the actual number between 0-9 is the label.

Since each image is of 28 x 28 pixels, and each pixel forms a feature, there are 784 features. MNIST digit recognition is a well-studied problem in the ML community, and people have trained numerous models (Neural Networks, SVMs, boosted trees etc.) achieving error rates as low as 0.23% (i.e. accuracy = 99.77%, with a convolutional neural network).

Before the popularity of neural networks, though, models such as SVMs and boosted trees were the state-of-the-art in such problems.

We'll first explore the dataset a bit, prepare it (scale etc.) and then experiment with linear and non-linear SVMs with various hyperparameters.

We'll divide the analysis into the following parts:

- Data understanding and cleaning
- Data preparation for model building
- Building an SVM model - hyperparameter tuning, model evaluation etc.

NOTE:

Considering the **computational limitations** of the system and the data size at hand, to make our life easier we are going to use 50% of the available data set for model building.

Final model can be extended to operate on the complete data set as well, considering appropriate computational power is available by the computing machine.

Data Understanding and Cleaning

Let's understand the dataset and see if it needs some cleaning etc.

```
[1]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import linear_model
from sklearn.model_selection import train_test_split
import re
```



```
+ Code + Text
Also, let's look at the average values of each column, since we'll need to do some rescaling in case the ranges vary too much.

[ ] # average values/distributions of features
description = digits.describe()
description

      label  pixel0  pixel1  pixel2  pixel3  pixel4  pixel5  pixel6  pixel7  pixel8  ...  pixel174  pixel175  pixel176  pixel177  pixel178  pixel179  pixel178  pixel178  pixel178  pixel178  pixel178
count  42000 42000.0 42000.0 42000.0 42000.0 42000.0 42000.0 42000.0 42000.0 42000.0 ... 42000.000000 42000.000000 42000.000000 42000.000000 42000.000000 42000.000000 42000.000000 42000.000000 42000.000000 42000.000000
mean    4.45643  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 ... 0.219286  0.117995  0.090024  0.02019  0.017238  0.002857  0.0  0.0  0.0  0.0
std     2.057730  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 ... 0.312890  0.433819  0.374488  0.17907  0.184495  0.414264  0.0  0.0  0.0  0.0
min     0.000000  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 ... 0.000000  0.000000  0.000000  0.00000  0.000000  0.000000  0.0  0.0  0.0  0.0
25%     2.000000  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 ... 0.000000  0.000000  0.000000  0.00000  0.000000  0.000000  0.0  0.0  0.0  0.0
50%     4.000000  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 ... 0.000000  0.000000  0.000000  0.00000  0.000000  0.000000  0.0  0.0  0.0  0.0
75%     7.000000  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 ... 0.000000  0.000000  0.000000  0.00000  0.000000  0.000000  0.0  0.0  0.0  0.0
max     9.000000  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 ... 254.000000  254.000000  253.000000  253.00000  254.000000  62.000000  0.0  0.0  0.0  0.0
8 rows x 785 columns

You can see that the max value of the mean and maximum values of some features (pixels) is 139, 255 etc., whereas most features lie in much lower ranges (look at description of pixel 0, pixel 1 etc. above).
Thus, it seems like a good idea to rescale the features.

• Data Preparation for Model Building

Let's now prepare the dataset for building the model. We'll only use a fraction of the data else training will take a long time.

[ ] # creating training and test sets
# splitting the data into train and test
X = digits.data[:, 1:]
y = digits.target[:, 1:]

# rescaling the features
from sklearn.preprocessing import scale
X = scale(X)

# train test split with train_size=0.8 and test_size=0.2
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, random_state=0)
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)

~/anaconda/lib/python3.6/site-packages/sklearn/model_selection/_split.py:2826: FutureWarning: From version 0.21, test_size will always complement train_size unless both are specified.
FutureWarning:
...
X
```

```
+ Code + Text
# delete test set from memory, to avoid a memory error
# we'll anyway use CV to evaluate the model, and can use the separate test.csv file as well
# to evaluate the model finally

# del X_test
# del y_test

• Model Building

Let's now build the model and tune the hyperparameters. Let's start with a linear model first.

Linear SVM

Let's first try building a linear SVM model (i.e. a linear kernel).

[ ] from sklearn import svm
from sklearn import metrics

# an initial svm model with linear kernel
svm_linear = svm.SVC(kernel='linear')

# fit
svm_linear.fit(X_train, y_train)

SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ov', degree=3, gamma='auto', kernel='linear',
    max_iter=1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=0)

[ ] # predict
predictions = svm_linear.predict(X_test)
predictions[0:5]

array([1, 3, 0, 1, 0, 1, 5, 0, 6])

[ ] # evaluation: accuracy
# C(i, j) represents the number of points known to be in class i
# but predicted to be in class j
confusion = metrics.confusion_matrix(y_test, y_pred = predictions)
confusion

array([[1615,  0, 12,  5,  0, 25, 20,  5,  0, 21],
       [ 0, 4889, 36, 23,  7,  7, 33, 25,  4],
       [ 0, 40, 1863, 60, 74, 13, 52, 50, 107],
       [20, 20, 121, 3387,  0, 179,  7, 94, 58, 44],
       [12, 10, 20,  0, 1897,  7, 41, 40,  4, 110],
       [40, 40, 32, 177, 41, 2899, 54, 14, 22, 18],
       [20, 10, 55,  7, 24, 37, 3480, 10, 21, 47],
       [ 0, 27, 37, 22, 70, 10,  4, 3639, 14, 142],
       [20, 10, 75, 137, 24, 137, 29, 20, 3086, 30],
       [30, 15, 39, 20, 122, 19,  1, 207, 27, 3220]])

[ ] # measure accuracy
...
X
```

```
+ Code + Test
[ ] * measure accuracy
metrics.accuracy_score(y_true=y_test, y_pred=predictions)

0.9429292929292929

[ ] * class-wise accuracy
class_wise = metrics.classification_report(y_true=y_test, y_pred=predictions)
print(class_wise)

precision    recall  f1-score   support

0   0.94   0.97   0.95   3715
1   0.94   0.98   0.96   4385
2   0.89   0.89   0.89   3788
3   0.88   0.87   0.87   3888
4   0.88   0.92   0.90   3782
5   0.87   0.85   0.86   3610
6   0.94   0.94   0.94   3693
7   0.90   0.92   0.91   3854
8   0.91   0.84   0.88   3665
9   0.88   0.85   0.87   3770

avg / total   0.90   0.90   0.90   37888

[ ] * run gc.collect() (garbage collect) to free up memory
# else, since the dataset is large and svm is computationally heavy,
# it'll throw a memory error while training
gc.collect()

87

Non-Linear SVM
Let's now try a non-linear model with the RBF kernel.

[ ] * rbf kernel with other hyperparameters kept to default
svm_rbf = svm.SVC(kernel='rbf')
svm_rbf.fit(x_train, y_train)

SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
    max_iter=1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)

[ ] * predict
predictions = svm_rbf.predict(x_test)

# accuracy
print(metrics.accuracy_score(y_true=y_test, y_pred=predictions))

0.925826816562
```

```
+ Code + Test
Non-Linear SVM
Let's now try a non-linear model with the RBF kernel.

[ ] * rbf kernel with other hyperparameters kept to default
svm_rbf = svm.SVC(kernel='rbf')
svm_rbf.fit(x_train, y_train)

SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
    max_iter=1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)

[ ] * predict
predictions = svm_rbf.predict(x_test)

# accuracy
print(metrics.accuracy_score(y_true=y_test, y_pred=predictions))

0.925826816562

The accuracy achieved with a non-linear kernel is slightly higher than a linear one. Let's now do a grid search CV to tune the hyperparameters C and gamma.

Grid Search Cross-Validation

[ ] * conduct (grid search) cross-validation to find the optimal values
# of cost C and the choice of kernel

from sklearn.model_selection import GridSearchCV

parameters = {'C': [1, 10, 100],
              'gamma': [1e-3, 1e-2, 1e-1]}

# instantiate a model
svm_grid_search = svm.SVC(kernel='rbf')

# create a classifier to perform grid search
clf = GridSearchCV(svm_grid_search, param_grid=parameters, scoring='accuracy')

# fit
clf.fit(x_train, y_train)

GridSearchCV(cv=None, error_score='raise',
             estimator=SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
                           decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
                           max_iter=1, probability=False, random_state=None, shrinking=True,
                           tol=0.001, verbose=False),
             fit_params=None, iid=True, n_jobs=1,
             param_grid={'C': [1, 10, 100], 'gamma': [0.01, 0.001, 0.0001]},
             pre_dispatch='2m_jobs', refit=True, return_train_score='warn',
             scoring='accuracy', verbose=0)
```

```

+ Code + Test
[7] # results
cv_results = pd.DataFrame(cv_results_)
cv_results

# converting C to numeric type for plotting on x-axis
cv_results['param_c'] = cv_results['param_c'].astype('int')

# plotting
plt.figure(figsize=(14,6))

# subplot 1/3
plt.subplot(131)
gamma_01 = cv_results[cv_results['param_gamma']==0.01]

plt.plot(gamma_01['param_c'], gamma_01['mean_test_score'])
plt.plot(gamma_01['param_c'], gamma_01['mean_train_score'])
plt.xlabel('C')
plt.ylabel('accuracy')
plt.title('Gamma=0.01')
plt.legend(['test accuracy', 'train accuracy'], loc='lower right')
plt.ylim([0.65, 1])
plt.xscale('log')

# subplot 2/3
plt.subplot(132)
gamma_001 = cv_results[cv_results['param_gamma']==0.001]

plt.plot(gamma_001['param_c'], gamma_001['mean_test_score'])
plt.plot(gamma_001['param_c'], gamma_001['mean_train_score'])
plt.xlabel('C')
plt.ylabel('accuracy')
plt.title('Gamma=0.001')
plt.legend(['test accuracy', 'train accuracy'], loc='lower right')
plt.ylim([0.65, 1])
plt.xscale('log')

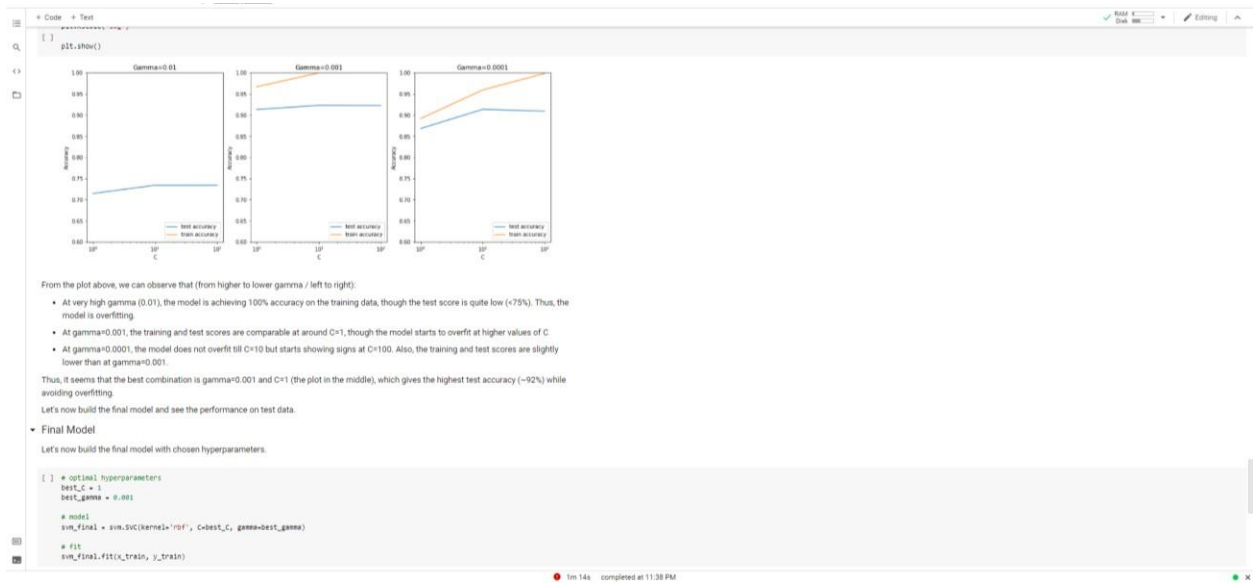
# subplot 3/3
plt.subplot(133)
gamma_0001 = cv_results[cv_results['param_gamma']==0.0001]

plt.plot(gamma_0001['param_c'], gamma_0001['mean_test_score'])
plt.plot(gamma_0001['param_c'], gamma_0001['mean_train_score'])
plt.xlabel('C')
plt.ylabel('accuracy')
plt.title('Gamma=0.0001')
plt.legend(['test accuracy', 'train accuracy'], loc='lower right')
plt.ylim([0.65, 1])
plt.xscale('log')

plt.show()

```

Gamma=0.01 Gamma=0.001 Gamma=0.0001 1m 14s completed at 11:38 PM




```
+ Code + Text
Final Model
Let's now build the final model with chosen hyperparameters.

[ ] # optimal hyperparameters
best_C = 1
best_gamma = 0.001

# model
svm_final = svm.SVC(kernel='rbf', C=best_C, gamma=best_gamma)

# fit
svm_final.fit(x_train, y_train)

SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=0.001, kernel='rbf',
    max_iter=1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)

[ ] # predict
predictions = svm_final.predict(x_test)

[ ] # evaluation: OR
confusion = metrics.confusion_matrix(y_true = y_test, y_pred = predictions)

# measure accuracy
test_accuracy = metrics.accuracy_score(y_true=y_test, y_pred=predictions)
print(test_accuracy, "\n")
print(confusion)

# 0.92079944074

[[[0 0 10 10 5 15 50 12 25 1]
 [0 4380 14 10 3 3 0 10 10 0]
 [24 23 3007 45 44 0 24 123 94 9]
 [4 21 86 1882 5 89 11 77 76 33]
 [0 11 26 7 340 13 23 43 5 110]
 [20 29 14 114 10 3628 79 93 24 95]
 [31 12 13 1 14 94 7911 44 25 0]
 [4 24 27 0 36 7 1 9735 7 97]
 [14 19 12 69 22 97 25 44 1251 41]
 [22 11 13 10 90 7 0 176 10 1379]]]

Conclusion
The final accuracy on test data is approx. 92%. Note that this can be significantly increased by using the entire training data of 42,000 images (we have used just 10% of that!).

1m 14s completed at 11:38 PM
```

CONCLUSION:

The final accuracy on test data is approx. 92%. Note that this can be significantly increased by using the entire training data of 42,000 images (we have used just 10% of that!).

....Thank You....