CS-634 Term Project Report
(Option-1: Supervised data mining)

First Name: Gautam
Middle Name: Varma
Last Name: Datla

NJIT UCID: gvd6

Email: gvd6@njit.edu

Department of Computer Science

New Jersey Institute of Technology (NJIT)

Newark, New Jersey - 07102

Abstract

Most of the data we deal with in practice comes in "labeled" form which could have been “labeled” when the data was initially being procured or could have been labeled synthetically using generative networks. In such scenarios, our dataset has a set of attributes in which each sample is associated with a target variable (could be a continuous or a discrete variable).

Now since most of the data comes in “labeled” form it is common to find the rules associating the features to the target variable using supervised data mining techniques. In this report, we classify image data using supervised classification algorithms like support vector machines and random forests, emphasizing scenarios where our target is a discrete variable.

Introduction

1.1 Problem Statement

In this project, the MNIST image dataset was used for supervised learning (to train SVM and random forest classifiers). Though there are a plethora of algorithms to perform supervised classification on datasets, choosing the right algorithm with optimal hyperparameters is indeed a tedious task. Moreover, the machine learning model that we build needs to generalize well when fed with unseen data.

This project mainly aims at using GridSearchCV to find the optimal hyperparameters of the classifier and K-Fold cross-validation to evaluate the test accuracy of the classifier so that our model generalizes well to new data at a high classification accuracy.

1.2 Project summary and methodology

• Platforms, data, and algorithms

Programming Language: Python (Jupyter Notebook)
Operating System: Windows 11
Hardware: Dell G7 17 Gaming Laptop (9th Gen Intel Core i7-9750H, NVIDIA GTX 1660 Ti 6G, 1024HDD , 512GB SSD, 16 GB RAM)

Dataset: MNIST dataset *(Modified National Institute of Standards and Technology database),* which is a collection of 60,000 handwritten data images.

Algorithm Categories :

i) Category1 - Support Vector Machine (radial basis function kernel)
ii) Category2 - Random Forests

• Project Methodology

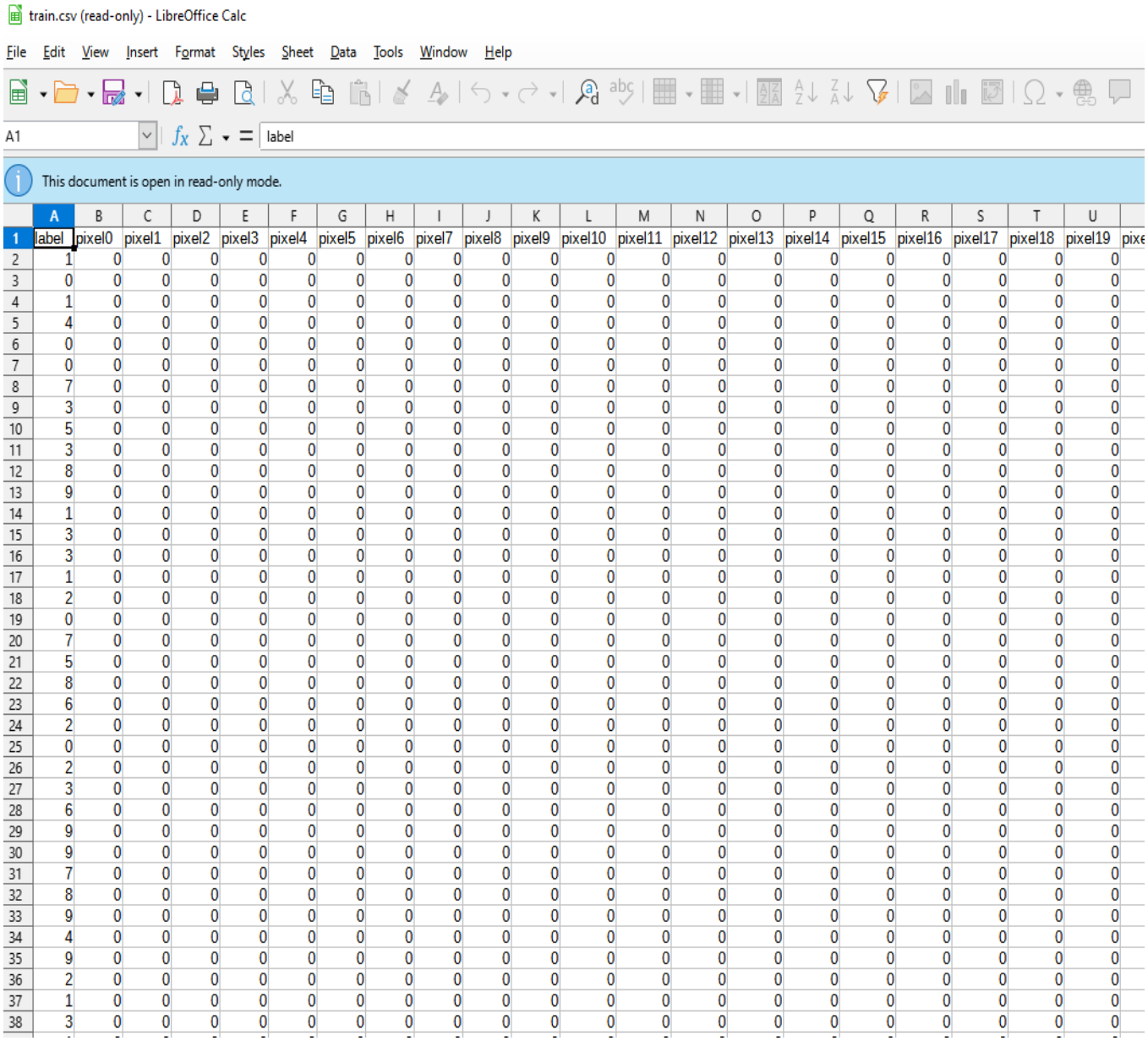
This project uses the MNIST dataset for classification. Once the dataset is procured and loaded the first step is to preprocess the data before feeding it to the classification algorithms. Data preprocessing includes steps such as checking for null values, imputing or removing missing/null values, checking if attributes are of the right data types, and feature scaling. After preprocessing the data we split the data into test and train data.

The next step involves using the GridSearchCV module to find the optimal hyperparameters and then using these to train the network. Once the machine learning model was trained to classify the images using optimal hyperparameters, I used K-fold cross-validation for evaluating the test accuracy.

Experiments and methodology

2.1 Description of data

MNIST images are 28x28 pixel images consisting of handwritten digits from 0-9. The dataset that we deal with has each image as a sample and all the 784 pixels of the corresponding image are flattened such that each attribute holds a pixel. So in total, we have a dataset with 784 attributes of pixels and one attribute to hold the class of each image.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	pixel10	pixel11	pixel12	pixel13	pixel14	pixel15	pixel16	pixel17	pixel18	pixel19
2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
22	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
23	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
29	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
30	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
32	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
33	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
34	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
35	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
36	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
37	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
38	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. Snippet of dataset

Dataset link: *https://paperswithcode.com/dataset/mnist*

Now the first step is to load our data,

```
In [3]: import numpy as np
        import pandas as pd

        %matplotlib inline
        import matplotlib
        import matplotlib.pyplot as plt
        import seaborn as sns
        import os
```

```
In [4]: os.getcwd()
        train_data = pd.read_csv("train.csv")
        test_data = pd.read_csv("test.csv")
```

```
In [5]: print(train_data.shape)
        print(test_data.shape)

        (42000, 785)
        (28000, 784)
```

```
In [4]: train_data.head()
```

Out[4]:

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel781	pi
0	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
2	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
3	4	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	

5 rows × 785 columns

Fig2. Loading the train and test datasets

Now if we have a look at the train data we find that the attribute “label” holds the class of our image and all the other attributes have values between 0-255 indicating the pixel intensity.

```
train_data["label"].unique()
```

```
array([1, 0, 4, 7, 3, 5, 8, 9, 2, 6], dtype=int64)
```

Fig3. Checking the number of unique classes in the dataset

```
train_data.describe()
```

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel775	pixel776	pixel77
count	42000.000000	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	...	42000.000000	42000.000000	42000.000000	42000.0000
mean	4.456643	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.219286	0.117095	0.059024	0.0201
std	2.887730	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	6.312890	4.633819	3.274488	1.7598
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	0.0000
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	0.0000
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	0.0000
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	0.0000
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	253.0000

8 rows × 785 columns

Fig4. Statistics of the data frame

Source code so far :

```
import numpy as np
import pandas as pd

%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
import os
os.getcwd()

train_data = pd.read_csv("train.csv")
test_data = pd.read_csv("test.csv")

print(train_data.shape)
print(test_data.shape)

train_data["label"].unique()

train_data.describe()
```

The next step is to check if the attributes hold the right data types. As we can see in the output below all the attributes are of data type int64 so there is no need to change the data types of any attribute.

```
train_data.info()
train_data.dtypes.unique()
```

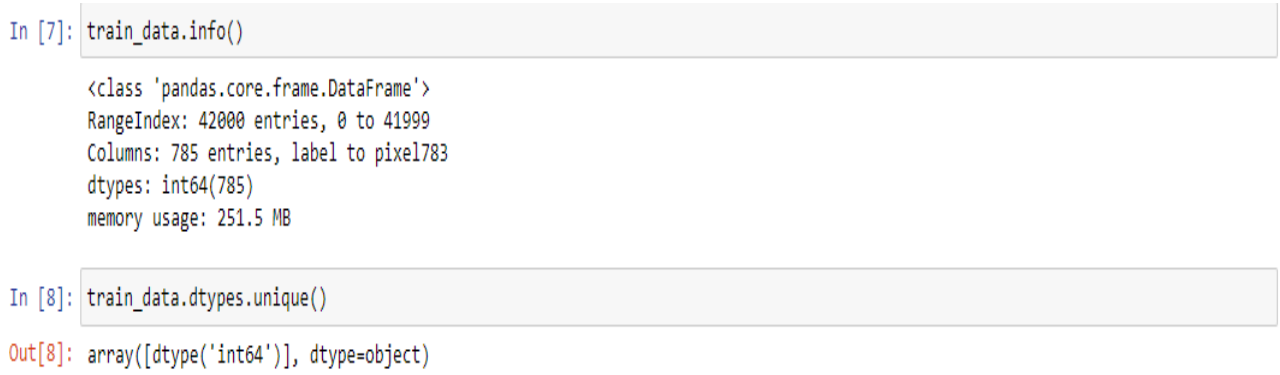


Fig 5. Checking attribute data types

Once we are done checking the data types we next check if there are any null / missing values in our data frame.

```
print("\nAre there any missing values in dataset? : {}
".format(test_data.isnull().any().any()))

print(test_data.isnull().any())
```

```
In [10]: print("\nAre there any missing values in dataset? : {} ".format(test_data.isnull().any().any()))

         print(test_data.isnull().any())
```

```
Are there any missing values in dataset? : False
pixel0      False
pixel1      False
pixel2      False
pixel3      False
pixel4      False
            ...
pixel779    False
pixel780    False
pixel781    False
pixel782    False
pixel783    False
Length: 784, dtype: bool
```

Fig6. Null/Missing values in the dataset

Now we can finally check the distribution of class labels to ensure there is no class imbalance. As we can see in Fig7. We almost have identical occurrences of all the classes, thus there is no need to oversample the dataset to tackle class imbalances.

```
sns.countplot(train_y)
```

```
In [45]: sns.countplot(train_y)
```

```
C:\Users\DELL\anaconda3\lib\site-packages\seaborn\_decorators.py:36: FutureWarning: Pass the following variable as a keyword ar
g: x. From version 0.12, the only valid positional argument will be `data`, and passing other arguments without an explicit key
word will result in an error or misinterpretation.
  warnings.warn(
```

```
Out[45]: <AxesSubplot:xlabel='label', ylabel='count'>
```

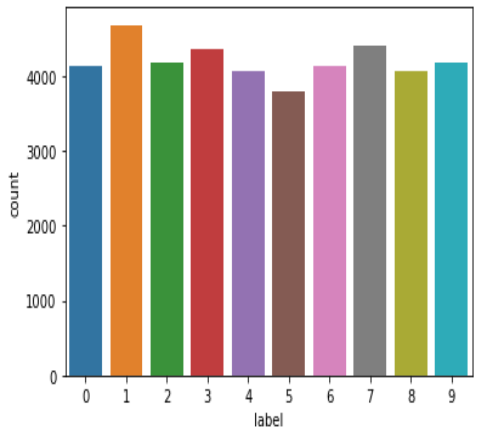


Fig7. Target distribution

Next we can define a function which takes in input a sample (image) in our dataset and it plots it for us .

```
def show_digit(row, data):
    sample_data = data.iloc[row,:].values
    sample_data =
sample_data.reshape(28,28).astype('uint8')
    plt.imshow(sample_data)
show_digit(5, train_x)
```

```
In [68]: def show_digit(row, data):
             sample_data = data.iloc[row,:].values
             sample_data = sample_data.reshape(28,28).astype('uint8')
             plt.imshow(sample_data)
         show_digit(5, train_x)
```

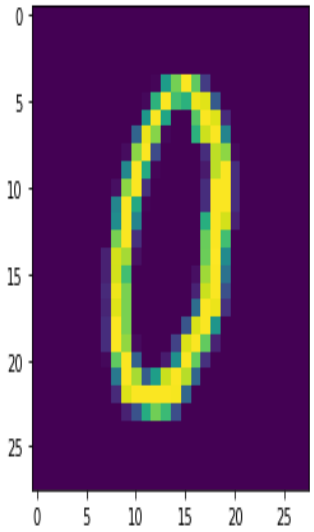


Fig.8 Plot of the 5th sample in the MNIST dataset

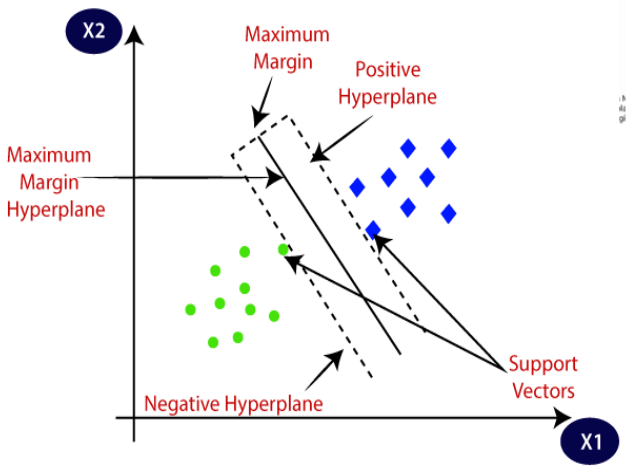
The final step before passing on our data to classification algorithms is to split the data into test data and train data. In this project, I used a train-test split ratio of 9:1..e 90% of our data is train data and 10% of our data is test data.

```
train_x = train_data.drop("label" , axis=1)
```

```
train_y = train_data["label"]

length = train_data.shape[0]
train_size = 0.9*length

x_train = train_x.loc[:train_size]
x_test = train_x.loc[train_size+1:]


y_train = train_y.loc[:train_size]
y_test = train_y.loc[train_size+1:]
```

Classification category 1: Support Vector Machines

The final step would be to classify the image data using supervised classification algorithms. So, we first implement the SVM classification algorithm using a radial basis function.



SVMs have supervised learning methods used for Classification and Regression. Support vector machines work by finding the hyperplane of the given dataset which is a plane that tries to maximize the minimum distance to each class. For example, In the figure shown below the line wx-b=0 is the hyperplane.

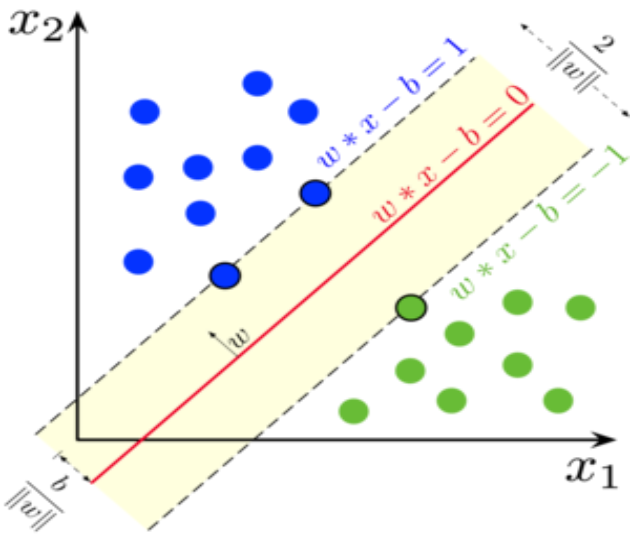


Fig9. Naive intuition of SVM

Support vector machines use kernels to tackle the non-linearities present in data . This project uses the **RBF kernel** also known as the radial basis function kernel. Due to their resemblance to the Gaussian distribution, RBF kernels are among the most generally utilized types of kernelization. RBF kernels apply linear operations to map points to higher-dimensional spaces that are simpler to separate.

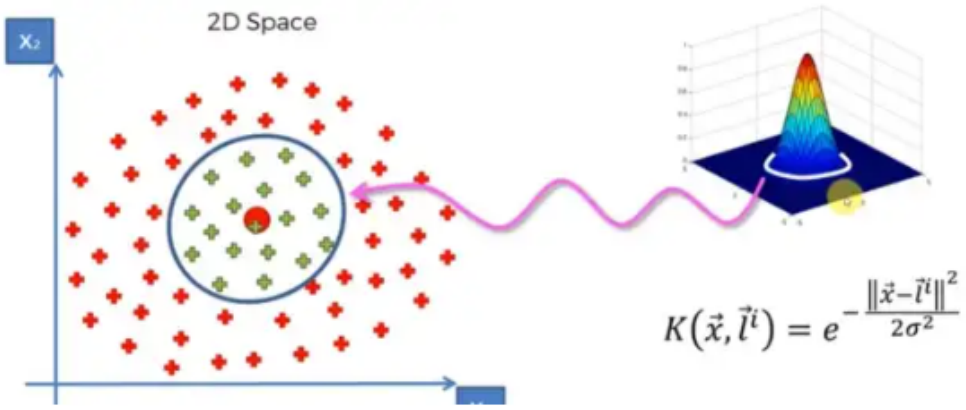


Fig10. RBF kernel

Another important aspect to consider while training on SVM is the choice of regularization parameter “C”. Regularization term “C” adds a penalty term to each sample point that was misclassified.

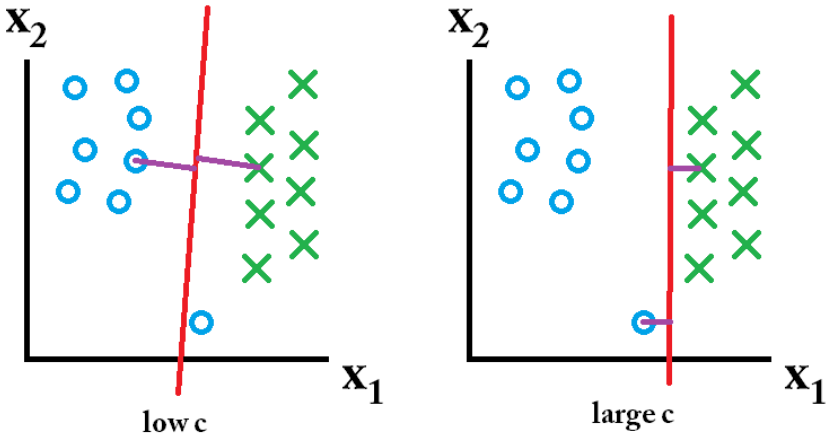


Fig11. The choice of C changes the hyperplane

Thus it is really important to choose optimal "C" values to train our network. We use gridsearchCV to find the optimal C value giving the highest accuracy. And for the cross-validation iterator in a grid search, we use 5-fold cross-validation with C values of 0.1,1,10.

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score,
KFold

parameter_grid = {"C": [0.1,1,10]}
classification_model_1 = SVC()

grid_crossvalidation = KFold(n_splits=5, shuffle=True,
random_state=0)
grid_search =

GridSearchCV(estimator=classification_model_1 ,
param_grid=parameter_grid, cv=grid_crossvalidation ,
n_jobs=2 , verbose=1)
grid_search.fit(train_x, train_y)
```

In [7]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score, KFold
```

In [8]:
```
parameter_grid = {"C": [0.1,1,10]}
classification_model_1 = SVC()
grid_crossvalidation = KFold(n_splits=5, shuffle=True, random_state=0)
grid_search = GridSearchCV(estimator=classification_model_1 , param_grid=parameter_grid, cv=grid_crossvalidation , n_jobs=2 , ver
grid_search.fit(train_x, train_y)
```

```
Fitting 5 folds for each of 3 candidates, totalling 15 fits
```

Out[8]:
```
GridSearchCV(cv=KFold(n_splits=5, random_state=0, shuffle=True),
             estimator=SVC(), n_jobs=2, param_grid={'C': [0.1, 1, 10]},
             verbose=1)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

Fig12. GridsearchCV to find the optimal C value

```
print(grid_search.best_score_)
print(grid_search.best_params_)
print(grid_search.best_estimator_)

cv_results = pd.DataFrame(grid_search.cv_results_)
Cv_results
```

```
In [9]: print(grid_search.best_score_)
        print(grid_search.best_params_)
        print(grid_search.best_estimator_)

        0.9803571428571429
        {'C': 10}
        SVC(C=10)
```

```
In [10]: cv_results = pd.DataFrame(grid_search.cv_results_)
         cv_results
```

Out[10]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	params	split0_test_score	split1_test_score	split2_test_score	split3_test_score	spli
0	193.605221	7.396947	105.337260	33.645841	0.1	{'C': 0.1}	0.952976	0.949048	0.949405	0.955357	
1	87.704245	5.740257	46.837979	0.854017	1	{'C': 1}	0.974762	0.975000	0.975119	0.976310	
2	82.723649	0.595053	44.806102	0.664158	10	{'C': 10}	0.980119	0.980476	0.981310	0.980833	

Fig13. Result of hyperparameter tuning using gridsearchCV

As we can see once the model was trained we find that the optimal value for **c is 10**, and the corresponding model gives us an accuracy of **98.035%.** At this point, using this score calls for great caution. The mistaken interpretation would be that since this mean score was calculated using cross-validation sets, we might use it to assess how well the model trained with the optimal hyper-parameters performs when it comes to generalization.

But we shouldn't forget that we choose the best model using this score. It indicates that we choose the hyper-parameter for the model itself using information from the test sets (i.e., test results).

This mean score does not accurately reflect our testing inaccuracy, as a result. It can be overly optimistic, especially when performing a parameter search on a

large grid with numerous hyper-parameters and a wide range of possible values for each hyper-parameter. Utilizing "nested" cross-validation is one approach to avoid this trap. Here the grid search that we implemented acts as the inner cross-validation. Now we can define outer cross-validation on a test set to evaluate the test accuracy. So, I used 10-fold cross-validation to find test accuracy.

```
final_model_1 = SVC(C=10)
final_model_1.fit(x_train,y_train)

cross_validation_iterator = KFold(n_splits=10 ,
shuffle=True, random_state=0)

test_score = cross_val_score(final_model_1, x_test,
y_test , cv=cross_validation_iterator, n_jobs=2)


print("\n\033[1;3m Test accuracy for each split
\033[0m\n")

for i , z in enumerate(test_score):
    print("{} -> {:.2f}%".format(i,z*100))
```

As we can see in Fig 14. We get test accuracies varying from 93.1% to 97.14% for each of the 10 splits.

```
In [18]: final_model_1 = SVC(C=10)
         final_model_1.fit(x_train,y_train)

Out[18]: SVC(C=10)
         In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
         On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [25]: cross_validation_iterator = KFold(n_splits=10 , shuffle=True, random_state=0)
         test_score = cross_val_score(final_model_1, x_test, y_test , cv=cross_validation_iterator, n_jobs=2)

In [44]: print("\n\033[1;3m Test accuracy for each split \033[0m\n")
         for i , z in enumerate(test_score):
             print("{} -> {:.2f}%".format(i,z*100))

          Test accuracy for each split

         0 -> 93.10%
         1 -> 96.43%
         2 -> 95.71%
         3 -> 95.24%
         4 -> 95.71%
         5 -> 94.76%
         6 -> 94.29%
         7 -> 96.90%
         8 -> 97.14%
         9 -> 96.18%
```

Fig16. Test accuracy for each split

```
print(" \033[1;3m Using k-fold cross validation we get the average test accuracy as {:.2f} % \033[0m".format(test_score.mean()*100))
```

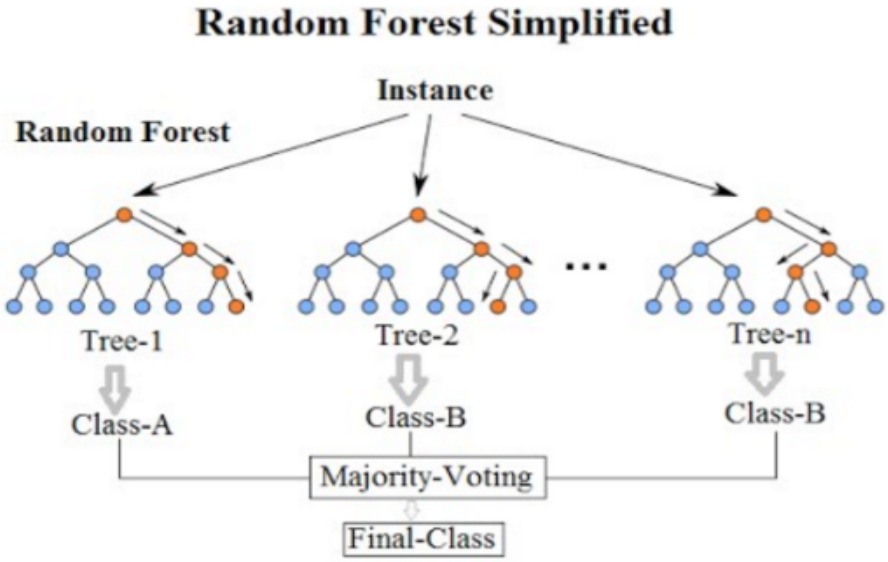
```
: print(" \033[1;3m Using k-fold cross validation we get the average test accuracy as {:.2f} % \033[0m".format(test_score.mean()*10

  Using k-fold cross validation we get the average test accuracy as 95.55 %
```

Fig15. Average Test accuracy using 10-fold cross-validation

Classification category 2: Random Forests

Random forests are a variant of the bagging method with the basic difference that the basic classifier or regressor in random forests is always a decision tree.

Another property of random forests is that when training a tree, the search for the optimum split is limited to a subset of the original features chosen at random. Each split node has a different set of random subsets. The idea is to add more randomness to the learning mechanism in order to try to decorrelate the prediction errors of the individual trees.



Random forests when used for classification have a lot of hyperparameters to be dealt with such as,

i) Number of estimators used in random forest i.e how many decision trees do we want to have in our random forest?

ii) Criterion used to measure the quality of a split

iii) Maximum depth of decision tree

For hyperparameter tuning using gridsearchCV, we set the parameter grid to have,

i) 100,200,500 estimators

ii) maximum depth of estimators as 4,5,6,7,8

iii) splitting criterion as Gini index/entropy

```
from sklearn.ensemble import RandomForestClassifier
classification_model_2=RandomForestClassifier(rando
m_state=42)

param_grid = { 'n_estimators': [100,200,500],
'max_depth' : [4,5,6,7,8],'criterion' :['gini',
'entropy'] }
grid_search_2 =
GridSearchCV(estimator=classification_model_2,
param_grid=param_grid, cv= 5)
grid_search_2.fit(x_train, y_train)
```

```
In [53]: from sklearn.ensemble import RandomForestClassifier
         classification_model_2=RandomForestClassifier(random_state=42)

In [55]: param_grid = { 'n_estimators': [100,200,500], 'max_depth' : [4,5,6,7,8],'criterion' :['gini', 'entropy'] }
         grid_search_2 = GridSearchCV(estimator=classification_model_2, param_grid=param_grid, cv= 5)
         grid_search_2.fit(x_train, y_train)

Out[55]: GridSearchCV(cv=5, estimator=RandomForestClassifier(random_state=42),
                      param_grid={'criterion': ['gini', 'entropy'],
                                  'max_depth': [4, 5, 6, 7, 8],
                                  'n_estimators': [100, 200, 500]})
         In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
         On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.
```

Fig16. Hyperparameter tuning using random forests

On checking for the optimal hyperparameters in the parameter grid we get the following results for random forests,

i) Best splitting criterion: *Entropy*
ii) Maximum depth of estimator: *8*
iii) Number of estimators to be used: *500*

```
In [57]: grid_search_2.best_params_

Out[57]: {'criterion': 'entropy', 'max_depth': 8, 'n_estimators': 500}

In [58]: model_2 = RandomForestClassifier(random_state=42,  n_estimators= 500, max_depth=8, criterion='entropy')

In [59]: model_2.fit(x_train,y_train)

Out[59]: RandomForestClassifier(criterion='entropy', max_depth=8, n_estimators=500,
                                random_state=42)
         In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
         On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.
```

Fig 17. Choosing the best hyperparameters for training random forest classifier

Next, similar to what we did with SVM, we define an outer 10-fold cross-validation for the evaluation of test accuracy.

```
cross_validation_iterator_2 = KFold(n_splits=10 ,
shuffle=True, random_state=0)

test_score_2 = cross_val_score(model_2, x_test,
y_test , cv=cross_validation_iterator_2, n_jobs=2)
print("\n\033[1;3m Test accuracy for each split
using random forest classifier \033[0m\n")
for i, z in enumerate(test_score_2):
    print("{} -> {:.2f}%".format(i,z*100))
```

```
In [60]: cross_validation_iterator_2 = KFold(n_splits=10 , shuffle=True, random_state=0)
         test_score_2 = cross_val_score(model_2, x_test, y_test , cv=cross_validation_iterator_2, n_jobs=2)

In [62]: print("\n\033[1;3m Test accuracy for each split using random forest classifier \033[0m\n")
         for i , z in enumerate(test_score_2):
             print("{} -> {:.2f}%".format(i,z*100))

 Test accuracy for each split using random forest classifier

0 -> 90.71%
1 -> 91.90%
2 -> 91.43%
3 -> 92.38%
4 -> 92.14%
5 -> 90.48%
6 -> 92.14%
7 -> 93.33%
8 -> 93.10%
9 -> 93.32%
```

Fig 18. 10-Fold cross-validation for evaluating random forest classifier's test accuracy

```
print(" \033[1;3m Using k-fold cross-validation for the given parameter search space we get the average test accuracy as {:.2f} %
\033[0m".format(test_score_2.mean()*100))
```

```
In [65]: print(" \033[1;3m Using k-fold cross validation for the given parameter search space we get the average test accuracy as {:.2f} %

 Using k-fold cross validation for the given parameter search space we get the average test accuracy as 92.09 %
```

Fig 19. Average Test accuracy for random forests using 10-fold cross-validation

Results and Conclusion

For classifying the MNIST image dataset we used supervised learning techniques such as SVM and Random Forests. Both algorithms were trained over nested-cross validation.

The inner cross-validation was used to find out the optimal hyperparameters of the corresponding machine-learning model while the outer cross-validation was used to evaluate test accuracy, thus ensuring that our model offers the best possible generalization to unseen data.

Gridsearchcv was used to find the best hyperparameters and we the best hyperparameters as the following,

i) Support vector classifier

c=10

ii) Random Forest classifier

Best splitting criterion = entropy
No of decision trees = 500
Maximum depth = 8

For evaluating the test accuracy, 10-fold cross-validation was used in both models. ***For the chosen hyperparameter space we get an average test accuracy of 95.55% for the Support vector classifier and 92.09% for the random forest classifier.***