Introduction

Machine Learning (ML) is a powerful tool that allows humans to make the computers analyze the features of the collected data and make use of it for further predictions. Different algorithms utilize different approaches for learning and decision procedures.

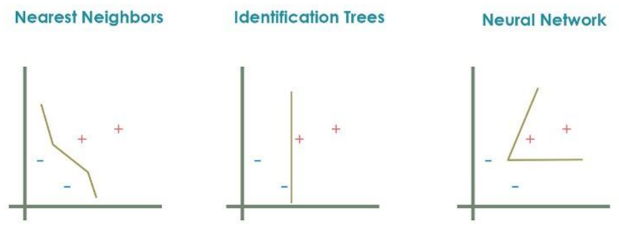


Figure 1

In Figure 1 various linear decision strategies of 3 well-known ML algorithms are shown. As can be seen from the graphs, although they use different rules to classify the samples, the fundamental principle of the decision-making process is same. All the described algorithms draw line(s) (also called decision boundary) to separate negative and positive samples.

Support Vector Machines

Support Vector Machines (SVM) is one of the sophisticated supervised ML algorithms that can be applied for both classification and regression problems. The idea was first introduced by Vladimir Naumovich Vapnik during the early ’90s. The main question that V. Vapnik asked during the development process of the algorithm was:

How to draw a straight line that best separates 2 classes?

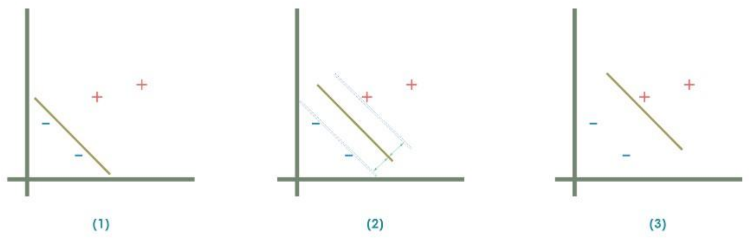


Figure 2

In Figure 2 lines with different positions were drawn to separate negative and positive samples. In the graph (1) and (3) the drawn lines are too close to the negative or positive samples respectively which decrease/increase the allotted gap (margin) for one of the classes. However, to sustain the objectivity of the decision process, the line must be drawn such that the allowed margin for both classes is equally wide.

Mathematical Definition of Decision Boundary

In the N-Dimensional (ND) feature space, the dimension of the decision boundary corresponds to N-1. For example:

— For 2D feature space, the decision boundary will be a line.
— For 3D feature space, the decision boundary will be a 2D plane.

Thus, to generalize the representation of the plane of the decision boundary the vector that is normal (perpendicular) to the medium of the plane can be utilized. That is how any plane (line) can be described using a single equation.

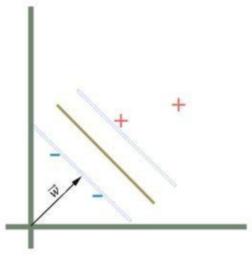


Figure 3

To represent the decision boundary, a vector (w) starting from the origin and normal the plane is drawn. However, there’s no piece of information about the parameters as well as the length of the vector w.

Decision Rule

Now let’s imagine we include an unknown sample (u) into the feature space. It is worth noting that any point in the cartesian coordinate system can be represented using a vector starting from the origin.

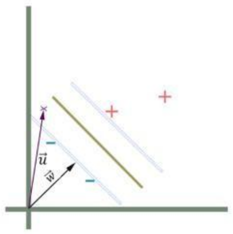


Figure 4

To determine whether an unknown sample is on the right or left side of the decision boundary the vector u can be projected onto the vector w by finding their dot product. If the value of projection of u onto w is greater than an unknown c, then a sample u is on the right side of the boundary or vice-versa.

$$\vec{u} \cdot \vec{w} \geq c \tag{1}$$

Without loss of generality the equation 1 can be modified by assuming c = -b.

$$\vec{u} \cdot \vec{w} + b \geq 0 \tag{2}$$

Above-written equation 2 is a decision rule of the SVM containing two unknown variables — w and b which are obtained during the training process of the SVM model.

Parametrizing Margin of SVM model

To be able to define the distance from each support-vector to the decision boundary, the decision rule for both classes should be considered separately. Due to mathematical convenience, the margin distance of 1 will be used after which the derived equation can be multiplied by any constant to increase/decrease margin proportionally.

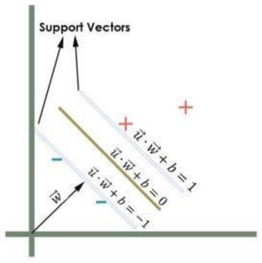

Support Vectors
+
+
$\vec{u} \cdot \vec{w} + b = 1$
$\vec{u} \cdot \vec{w} + b = 0$
$\vec{u} \cdot \vec{w} + b = -1$
−
−
$\vec{w}$

Figure 5

Thus, equation 2 for the positive (x+) and negative (x-) samples can be modified as it is described in equation 3.1 and 3.2.

$$\overrightarrow{x_+} \cdot \vec{w} + b \geq 1 \tag{3.1}$$
$$\overrightarrow{x_-} \cdot \vec{w} + b \leq -1 \tag{3.2}$$

Basically, by doing so we force the decision boundary to have a separation of distance from -1 to +1 for all the training samples.

Although there are 2 separate equations for different classes, by multiplying equation 3.1 with +1 and 3.2 with -1 the equations can be merged into one (equation 5) which will be more mathematically convenient. For this, a new variable y_i (equation 4) will be included in the equation.

$$y_i = \begin{cases} -1, & \vec{x} \in \vec{x}_+ \\ +1, & \vec{x} \in \vec{x}_- \end{cases} \tag{4}$$

$$y_i(\vec{x} \cdot \vec{w} + b) \geq 1 \tag{5}$$

By considering equation 5 the result of the decision rule for the samples on the support vectors can be derived which is also described in Figure 6.

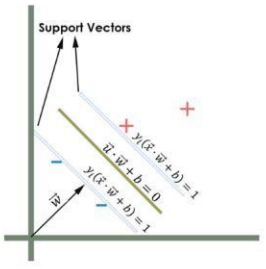

Support Vectors
+
+
−
−
$\vec{w}$
$y_i(\vec{x} \cdot \vec{w} + b) = 1$
$\vec{u} \cdot \vec{w} + b = 0$
$y_i(\vec{x} \cdot \vec{w} + b) = 1$

Figure 6

$$y_i(\vec{x} \cdot \vec{w} + b) = 1$$

Optimization Problem

In the previous sections, the constraints for determining the decision boundary of SVM were discussed. However, the main objective of SVM is to find a decision boundary with the widest distance of margin.

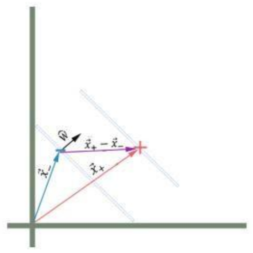


Figure 7

To find the distance between the support vectors, any of the positive (+) and negative (-) samples that are located on the support vectors can be used. The distance will be equal to the projection of the difference of the vectors onto the unit vector in the direction of normal vector w. The unit vector can be calculated by dividing the components of the vector w by its length.

$$width = (\vec{x}_+ - \vec{x}_-) \cdot \hat{w} = (\vec{x}_+ - \vec{x}_-) \cdot \frac{\vec{w}}{|\vec{w}|} \tag{6}$$

Equation 6 can be further simplified by using equation 3.1 and 3.2

$$\begin{cases} \vec{x}_+ \cdot \vec{w} = 1 - b \\ \vec{x}_- \cdot \vec{w} = -1 - b \end{cases} \tag{7}$$

$$width = \frac{(\vec{x}_+ \cdot \vec{w} - \vec{x}_- \cdot \vec{w})}{|\vec{w}|} = \frac{1 - b + 1 + b}{|\vec{w}|} = \frac{2}{|\vec{w}|} \tag{8}$$

Last, a simple equation (8) was derived that describes the distance between the support vectors. The objective is to maximize it as well as considering the constraints discussed above.

$$\max(width) \xrightarrow{yields} \max\left(\frac{2}{|\vec{w}|}\right) \xrightarrow{yields} \min(|\vec{w}|) \xrightarrow{yields} \min\left(\frac{|\vec{w}|^2}{2}\right) \quad (9)$$

Equation 9 describes the conversion steps of optimization problem into quadratic optimization which is more mathematically convenient during the obtaining process of gradients with respect to w.

Constrained Quadratic Optimization with Lagrange Multipliers

The function that has to be minimized in equation 9 is constrained by additional functions given in Figure 6. For the problem of constrained optimization Lagrange multipliers can be utilized, simply, to convert a constrained optimization problem into unconstrained one which is more computationally convenient. What theory of Lagrange multipliers claims is that:

For the minimum/maximum solution of the function f(x) constrained by g(x), the level curve of the f(x) is tangent to the level curve of the g(x). Thus, if the level curves are tangent, their gradients must be parallel.

$$\begin{aligned} \nabla f(x) &= \alpha \nabla g(x) \qquad (10) \\ \nabla f(x) - \alpha \nabla g(x) &= 0 \\ L(x,\alpha) &= f(x) - \alpha\, g(x) \end{aligned}$$

By using equation 10 the constrained optimization problem of SVM is converted to the unconstrained one. A new equation will be the objective function of SVM with the summation over all constraints.

$$f(w,b) = \frac{1}{2}|w|^2 \quad (11)$$

$$g(w,b) = y_i(\vec{x} \cdot \vec{w} + b) - 1 = 0$$

$$L_{\min(w,b)}(w,b) = \frac{1}{2}|w|^2 - \sum_i \alpha_i [y_i(\vec{x_i} \cdot \vec{w} + b) - 1]$$

In equation 11 the Lagrange multiplier was not included as an argument to the objective function L(w,b). As the value of the multiplier α will be set conditionally during the numerical optimization process such that:

- If the sample is on the support vector: α = 1
- else: α = 0

To minimize the objective function the approach of Gradient Descent will be implemented. Here computation of the gradient of L(w,b) with respect to w and b is required which are described in the equations below.

$$By\ considering: \frac{|w|^2}{\partial w} = \frac{\vec{w} \cdot \vec{w}}{\partial w} \quad (12.1)$$

$$\frac{L(w,b)}{\partial w} = \vec{w} - \sum_i \alpha_i y_i \vec{x} \quad (12.2)$$

$$\frac{L(w,b)}{\partial b} = - \sum_i \alpha_i y_i \quad (12.3)$$

As the mathematical foundation of the SVM was completely discussed, it is time for the implementation in Python by utilizing only ‘NumPy’ module.

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).

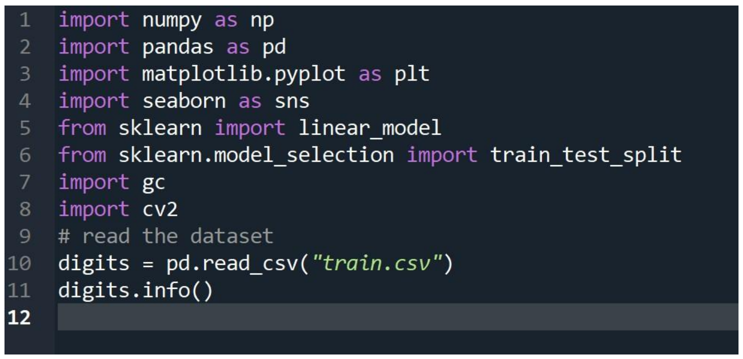
Before the popularity of neural networks, though, models such as SVMs and boosted trees were the state-of-the-art in such problems.

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.

Data understanding and cleaning



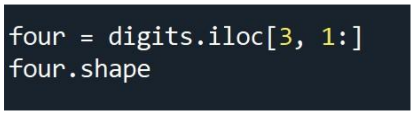
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import linear_model
from sklearn.model_selection import train_test_split
import gc
import cv2
# read the dataset
digits = pd.read_csv("train.csv")
digits.info()

```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 42000 entries, 0 to 41999
Columns: 785 entries, label to pixel783 dtypes:
int64(785)
memory usage: 251.5 MB
```

The out put is :

	lable	Pixel0	Pixel1	Pixel2	Pixel3	……	Pixel781	Pixel782	Pixel783
0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0
3	4	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0

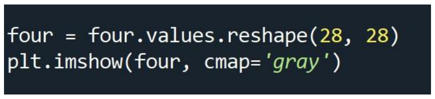
5 rows × 785 columns



```python
four = digits.iloc[3, 1:]
four.shape
```

The out put:

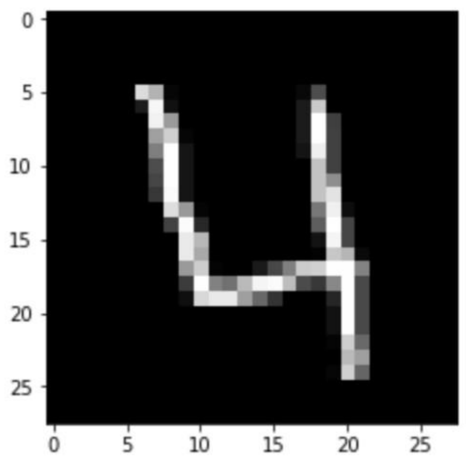
```
(784,)
```



```
four = four.values.reshape(28, 28)
plt.imshow(four, cmap='gray')
```

Out put :

<matplotlib.image.AxesImage at 0x1a26b33208>



Side note: Indexing Recall list
= [0, 4, 2, 10, 22, 101, 10]
indices = [0, 1, 2, 3, ...,]
reverse = [-n -3 -2 -1]

```
# visualise the array
print(four[5:-5, 5:-5])
```

```
[[  0 220 179   6   0   0   0   0   0   0   0   0   9  77   0   0   0   0]
 [  0  28 247  17   0   0   0   0   0   0   0   0  27 202   0   0   0   0]
 [  0   0 242 155   0   0   0   0   0   0   0   0  27 254  63   0   0   0]
 [  0   0 160 207   6   0   0   0   0   0   0   0  27 254  65   0   0   0]
 [  0   0 127 254  21   0   0   0   0   0   0   0  20 239  65   0   0   0]
 [  0   0  77 254  21   0   0   0   0   0   0   0   0 195  65   0   0   0]
 [  0   0  70 254  21   0   0   0   0   0   0   0   0 195 142   0   0   0]
 [  0   0  56 251  21   0   0   0   0   0   0   0   0 195 227   0   0   0]
 [  0   0   0 222 153   5   0   0   0   0   0   0   0 120 240  13   0   0]
 [  0   0   0  67 251  40   0   0   0   0   0   0   0  94 255  69   0   0]
 [  0   0   0   0 234 184   0   0   0   0   0   0   0  19 245  69   0   0]
 [  0   0   0   0 234 169   0   0   0   0   0   0   0   3 199 182  10   0]
 [  0   0   0   0 154 205   4   0   0  26  72 128 203 208 254 254 131   0]  [  0   0   0   0  61
254 129 113 186 245 251 189  75  56 136 254  73   0]
 [  0   0   0   0  15 216 233 233 159 104  52   0   0   0  38 254  73   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0  18 254  73   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0  18 254  73   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   5 206 106   0]]
```

```
# Summarise the counts of 'label' to see how many labels of each digit are present
digits.label.value_counts()
```

```
1    4684
7    4401
3    4351
9    4188
2    4177
6    4137
0    4132
4    4072
8    4063
5    3795
Name: label, dtype: int64
```

```
# Summarise count in terms of percentage
100*(round(digits.label.astype('category').value_counts()/len(digits.index), 4))
```

```
1    11.15
7    10.48
3    10.36
9     9.97
2     9.95
6     9.85
0     9.84
4     9.70
8     9.67
5     9.04
Name: label, dtype: float64
```

Thus, each digit/label has an approximately 9%-11% fraction in the dataset and the **dataset is balanced**. This is an important factor in considering the choices of models to be used, especially SVM, since **SVMs rarely perform well on imbalanced data** (think about why that might be the case).

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.iloc[:, 1:]
Y = digits.iloc[:, 0]

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

# train test split with train_size=10% and test size=90%
x_train, x_test, y_train, y_test = train_test_split(X, Y, train_size=0.10,
random_state=101) print(x_train.shape) print(x_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
/anaconda3/lib/python3.6/site-packages/sklearn/model_selection/_split.py:2026: Future
Warning: From version 0.21, test_size will always complement train_size unless both ar
e specified.
  FutureWarning)
```

```
(4200, 784)
(37800, 784)
(4200,)
(37800,)
```

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

In :
```
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)
```
Out:

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
decision_function_shape='ovr', degree=3, gamma='auto', kernel='linear',
max_iter=-1, probability=False, random_state=None, shrinking=True,   tol=0.001,
verbose=False)
In :
# predict
predictions = svm_linear.predict(x_test)
predictions[:10] Out: array([1, 3, 0, 0, 1,
9, 1, 5, 0, 6]) In :
# 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_true = y_test, y_pred = predictions)
confusion Out:
array([[3615,    0,   12,    8,    8,   28,   28,    5,    9,    2],
[   0, 4089,   16,   23,    9,    3,    3,   13,   25,    4],
      [  54,   48, 3363,   64,   74,   13,   53,   52,   59,   10],
[  20,   28,  121, 3387,    8,  175,    5,   54,   58,   44],
      [  12,   12,   26,    2, 3399,    7,   41,   41,    4,  158],
      [  49,   42,   32,  177,   41, 2899,   54,   14,   82,   28],
      [  36,   16,   55,    5,   34,   37, 3486,    3,   21,    0],
      [   9,   27,   37,   22,   70,   10,    4, 3619,   14,  142],
      [  26,   86,   71,  137,   24,  137,   29,   26, 3096,   33],
[  38,   11,   39,   26,  182,   19,    1,  207,   27, 3228]]) In :
# measure accuracy metrics.accuracy_score(y_true=y_test,
y_pred=predictions) Out:
0.9042592592592592
In :
# 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      4185
2      0.89      0.89      0.89      3790
3      0.88      0.87      0.87      3900
4      0.88      0.92      0.90      3702
5      0.87      0.85      0.86      3418
6      0.94      0.94      0.94      3693
7      0.90      0.92      0.91      3954
8      0.91      0.84      0.88      3665
9      0.88      0.85      0.87      3778

avg / total       0.90      0.90      0.90     37800
```

In:

```
# 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() Out:
87
```

RBF kernel:

RBF short for **Radial Basis Function Kernel** is a very powerful kernel used in SVM. Unlike linear or polynomial kernels, RBF is more complex and efficient at the same time that it can combine multiple polynomial kernels multiple times of different degrees to project the non-linearly separable data into higher dimensional space so that it can be separable using a hyperplane.

The RBF kernel works by mapping the data into a high-dimensional space by finding the dot products and squares of all the features in the dataset and then performing the classification using the basic idea of Linear SVM. For projecting the data into a higher dimensional space, the RBF kernel uses the so-called radial basis function which can be written as:

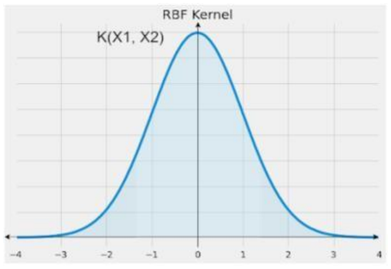
$$K(X_1, X_2) = \exp(-\frac{\|X_1 - X_2\|^2}{2\sigma^2})$$

Here **||X1 - X2||^2** is known as the **Squared Euclidean Distance** and **σ** is a free parameter that can be used to tune the equation.

When introducing a new parameter γ **= 1 / 2σ^2,** the equation will be

$$K(X_1, X_2) = \exp(-\gamma \|X_1 - X_2\|^2)$$

The equation is really simple here, the **Squared Euclidean Distance** is multiplied by the **gamma** parameter and then finding the **exponent** of the whole. This equation can find the **transformed inner products** for mapping the data into higher dimensions directly without actually transforming the entire dataset which leads to inefficiency. And this is why it is known as the RBF kernel **function.** The distribution graph of RBF Kernel will look like this:



As you can see that the Distribution graph of the RBF kernel actually looks like the **Gaussian Distribution curve** which is known as **a bell-shaped curve.** Thus RBF kernel is also known as **Gaussian Radial Basis Kernel.**

RBF kernel is most popularly used with **K-Nearest Neighbors** and **Support Vector Machines.**

In :

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

Out:

```
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)
```

In:

```
# predict
predictions = svm_rbf.predict(x_test)

# accuracy
print(metrics.accuracy_score(y_true=y_test, y_pred=predictions))
0.925582010582
```

The Polynomial Kernel

A polynomial kernel is a kind of SVM kernel that uses a polynomial function to map the data into a higher-dimensional space. It does this by taking the dot product of the data points in the original space and the polynomial function in the new space.

In a polynomial kernel for SVM, the data is mapped into a higher-dimensional space using a polynomial function. The dot product of the data points in the original space and the polynomial function in the new space is then taken. The polynomial kernel is often used in SVM classification problems where the data is not linearly separable. By mapping the data into a higher-dimensional space, the polynomial kernel can sometimes find a hyperplane that separates the classes.

The polynomial kernel has a number of parameters that can be tuned to improve its performance, including the degree of the polynomial and the coefficient of the polynomial.

For degree d polynomials, the polynomial kernel is defined as:

$$K(x_1, x_2) = (x_1^T x_2 + c)^d$$

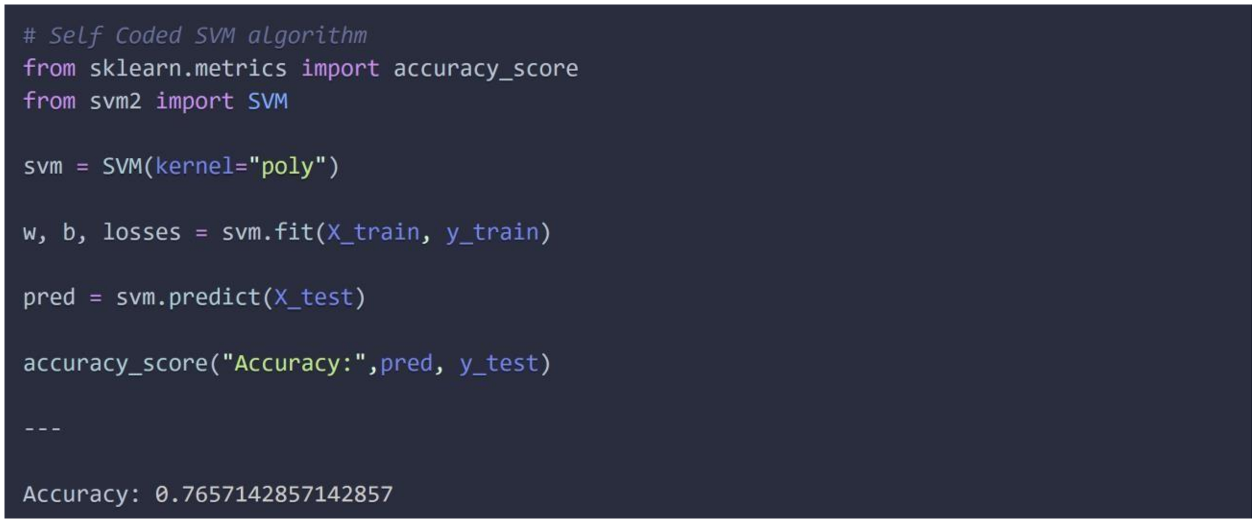
Where **c** is a constant and **x1** and **x2** are vectors in the original space.

The parameter **c** can be used to control the trade-off between the fit of the training data and the size of the margin. A large **c** value will give a low training error but may result in overfitting. A small **c** value will give a high training error but may result in underfitting. The degree **d** of the polynomial can be used to control the complexity of the model. A high degree **d** will result in a more complex model that may overfit the data, while a low degree **d** will result in a simpler model that may underfit the data.

When a dataset is given containing features x1 and x2, the equation can be transformed as:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \cdot \begin{bmatrix} x_1 & x_2 \end{bmatrix} = \begin{bmatrix} x_1^2 & x_1 x_2 \\ x_1 x_2 & x_2^2 \end{bmatrix}$$

The important terms we need to note are **x1, x2, x1^2, x2^2**, and **x1 * x2.** When finding these new terms, the non-linear dataset is converted to another dimension that has features **x1^2, x2^2**, and **x1 * x2.**



```
# Self Coded SVM algorithm
from sklearn.metrics import accuracy_score
from svm2 import SVM

svm = SVM(kernel="poly")

w, b, losses = svm.fit(X_train, y_train)

pred = svm.predict(X_test)

accuracy_score("Accuracy:",pred, y_test)

---

Accuracy: 0.7657142857142857
```

Random forest

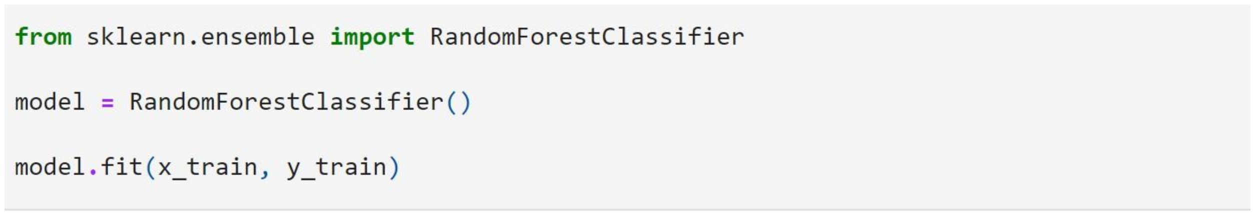
You'll use the Random Forests algorithm to build a handwritten digit classifier. As discussed before, this has some *pros* and *cons* when comparing to the neural network

The biggest *pro* is the training speed – the training process will finish in a minute or so on CPU, whereas the training process for neural networks can take anywhere from minutes (GPU) to hours (CPU) – depending on the model architecture and your hardware.

The downside of using Random Forests (or any other machine learning algorithm) is the loss of 2D information. When you flatten the image (go from 28×28 to 1×784), you're losing information on surrounding pixels. A convolution operation is a go-to approach for any more demanding image classification problem.

Still, the Random Forest classifier should suit you fine on the MNIST dataset.

The following code snippet shows you how to import the library, train the model, and print the results. The execution will take a minute or so, depending on your hardware:



```python
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier()

model.fit(x_train, y_train)
```

```
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages/sklear
n/ensemble/forest.py:245: FutureWarning: The default value of n_estimators will change
from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
```

In: # precting the model
y_pred **=** model**.**predict(x_test)

model**.**score(x_test, y_test)

out: 0.9489

Naive Bayes Classifier

In machine learning, Naive Bayes classification is a straightforward and powerful algorithm for the classification task. Naïve Bayes classification is based on applying Bayes' theorem with strong independence assumption between the features. Naïve

Bayes classification produces good results when we use it for textual data analysis such as Natural Language Processing.

Naïve Bayes models are also known as simple Bayes or independent Bayes. All these names refer to the application of Bayes' theorem in the classifier's decision rule. Naïve Bayes classifier applies the Bayes' theorem in practice. This classifier brings the power of Bayes' theorem to machine learning.

Naïve Bayes Classifier uses the Bayes' theorem to predict membership probabilities for each class such as the probability that given record or data point belongs to a particular class. The class with the highest probability is considered as the most likely class. This is also known as the **Maximum A Posteriori (MAP)**.

The **MAP for a hypothesis with 2 events A and B is MAP (A)**

= max (P (A | B))

= max (P (B | A) * P (A))/P (B)

= max (P (B | A) * P (A))

Here, P (B) is evidence probability. It is used to normalize the result. It remains the same, So, removing it would not affect the result.

Naïve Bayes Classifier assumes that all the features are unrelated to each other. Presence or absence of a feature does not influence the presence or absence of any other feature.

In real world datasets, we test a hypothesis given multiple evidence on features. So, the calculations become quite complicated. To simplify the work, the feature independence approach is used to uncouple multiple evidence and treat each as an independent one.

3. Types of Naive Bayes algorithm

There are 3 types of Naïve Bayes algorithm. The 3 types are listed below:-

1. Gaussian Naive Bayes
2. Multinomial Naive Bayes
3. Bernoulli Naive Bayes

These 3 types of algorithm are explained below.

Gaussian Naive Bayes algorithm

When we have continuous attribute values, we made an assumption that the values associated with each class are distributed according to Gaussian or Normal distribution. For example, suppose the training data contains a continuous attribute x. We first segment the data by the class, and then compute the mean and variance of x in each class. Let μi be the mean of the values and let σi be the variance of the values associated with the ith class. Suppose we have some observation value xi . Then, the probability distribution of xi given a class can be computed by the following equation –

$$p(x_i|y_j) = \frac{1}{\sqrt{2\pi\sigma_j^2}} e^{-\frac{(x_i-\mu_j)^2}{2\sigma_j^2}}$$

Multinomial Naive Bayes algorithm

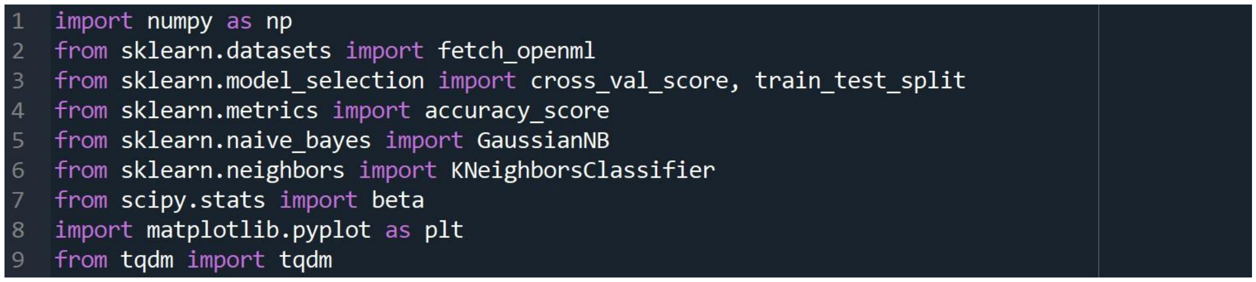
With a Multinomial Naive Bayes model, samples (feature vectors) represent the frequencies with which certain events have been generated by a multinomial (p1, . . . ,pn) where pi is the probability that event i occurs. Multinomial Naïve Bayes algorithm is preferred to use on data that is multinomially distributed. It is one of the standard algorithms which is used in text categorization classification.

Bernoulli Naive Bayes algorithm

In the multivariate Bernoulli event model, features are independent boolean variables (binary variables) describing inputs. Just like the multinomial model, this model is also popular for document classification tasks where binary term occurrence features are used rather than term frequencies.

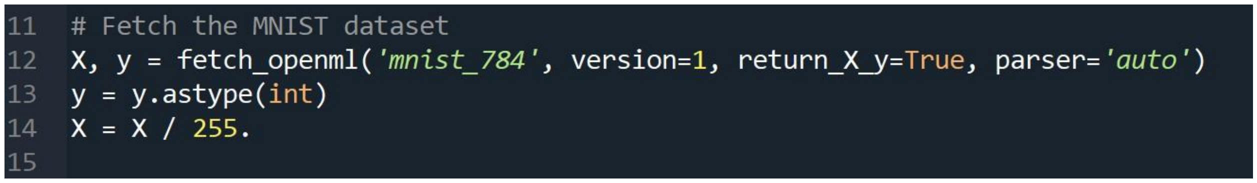
Now lets start the code :

Imports: These lines import necessary libraries and modules. Notable imports include NumPy for numerical operations, scikit-learn for machine learning tools, SciPy for statistical functions, Matplotlib for plotting, and tqdm for displaying progress bars.



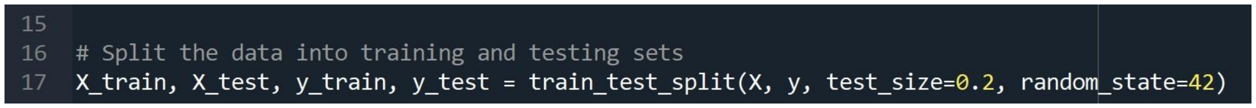
```python
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.metrics import accuracy_score
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from scipy.stats import beta
import matplotlib.pyplot as plt
from tqdm import tqdm
```

Fetching MNIST Dataset: This block of code fetches the MNIST dataset using scikitlearn's fetch_openml function. The data is loaded into X (features) and y (labels). The parser='auto' parameter is added to suppress a future warning. Labels are converted to integers, and the pixel values are normalized between 0 and 1.



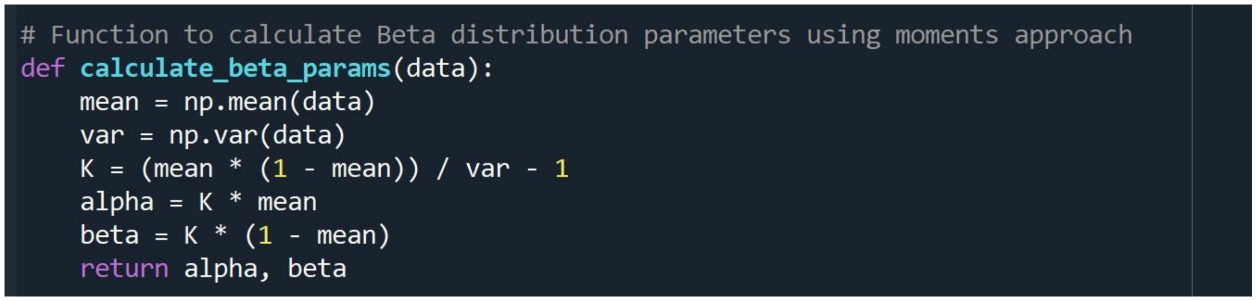
```python
# Fetch the MNIST dataset
X, y = fetch_openml('mnist_784', version=1, return_X_y=True, parser='auto')
y = y.astype(int)
X = X / 255.

```

Train-Test Split: The dataset is split into training and testing sets using the train_test_split function. 80% of the data is used for training, and 20% for testing. The random_state parameter ensures reproducibility.



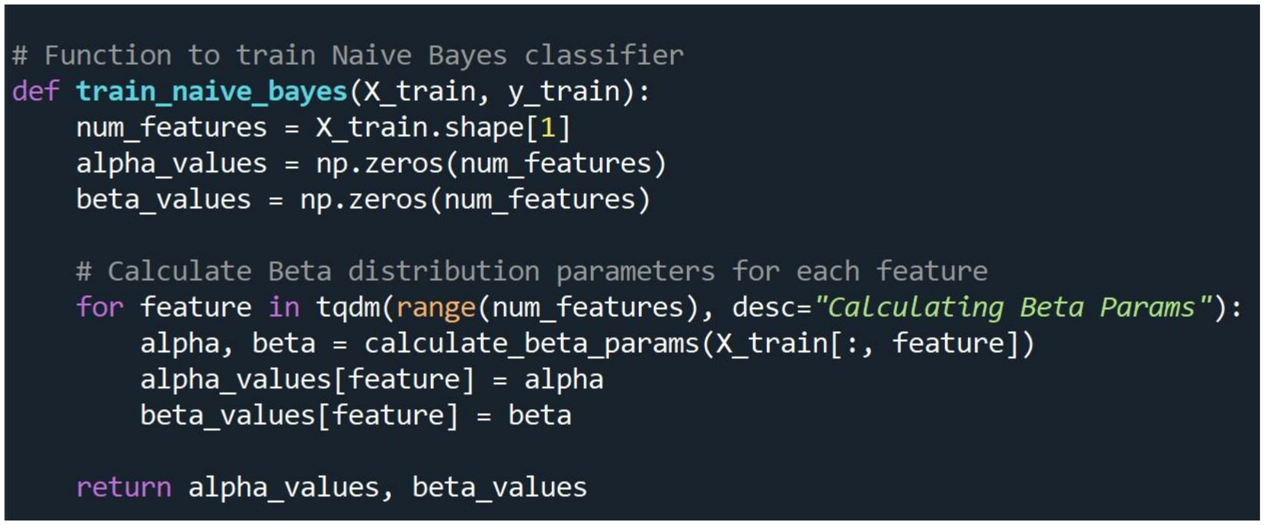
```python

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Beta Distribution Parameters Calculation Function: This function calculates the parameters (alpha and beta) for a Beta distribution using the moments approach. It takes a 1D array of data as input.



```python
# Function to calculate Beta distribution parameters using moments approach
def calculate_beta_params(data):
    mean = np.mean(data)
    var = np.var(data)
    K = (mean * (1 - mean)) / var - 1
    alpha = K * mean
    beta = K * (1 - mean)
    return alpha, beta
```

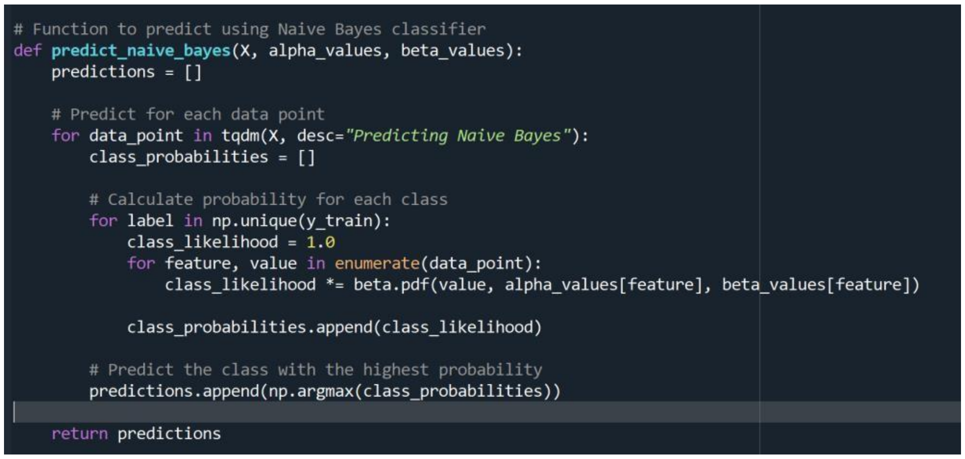
Naive Bayes Training Function: This function trains a Naive Bayes classifier. It calculates Beta distribution parameters for each feature using the calculate_beta_params function and stores them in arrays.


```python
# Function to train Naive Bayes classifier
def train_naive_bayes(X_train, y_train):
    num_features = X_train.shape[1]
    alpha_values = np.zeros(num_features)
    beta_values = np.zeros(num_features)

    # Calculate Beta distribution parameters for each feature
    for feature in tqdm(range(num_features), desc="Calculating Beta Params"):
        alpha, beta = calculate_beta_params(X_train[:, feature])
        alpha_values[feature] = alpha
        beta_values[feature] = beta

    return alpha_values, beta_values
```

Naive Bayes Prediction Function: This function predicts labels for given data points using the trained Naive Bayes classifier. It calculates the likelihood for each class and predicts the one with the highest likelihood.

The remaining code, which is not included, would involve calling these functions, evaluating the model's performance, and potentially visualizing the results.

Note: The code uses the Beta distribution to model each feature's distribution, and the Naive Bayes assumption is applied independently to each feature given the class label. This approach might not be the most suitable for image data, and other models like kNN or convolutional neural networks (CNNs) might be more effective for image classification tasks.


```python
# Function to predict using Naive Bayes classifier
def predict_naive_bayes(X, alpha_values, beta_values):
    predictions = []

    # Predict for each data point
    for data_point in tqdm(X, desc="Predicting Naive Bayes"):
        class_probabilities = []

        # Calculate probability for each class
        for label in np.unique(y_train):
            class_likelihood = 1.0
            for feature, value in enumerate(data_point):
                class_likelihood *= beta.pdf(value, alpha_values[feature], beta_values[feature])

            class_probabilities.append(class_likelihood)

        # Predict the class with the highest probability
        predictions.append(np.argmax(class_probabilities))

    return predictions
```

K-Nearest Neighbors Algorithm

The K-nearest neighbors (KNN) algorithm is a type of supervised machine learning algorithms. KNN is extremely easy to implement in its most basic form, and yet performs quite complex classification tasks. It is a lazy learning algorithm since it doesn't have a specialized training phase. Rather, it uses all of the data for training while classifying a new data point or instance. KNN is a non-parametric learning algorithm, which means that it doesn't assume anything about the underlying data.

Pros and Cons of KNN Pros:

It is extremely easy to implement

As said earlier, it is lazy learning algorithm and therefore requires no training prior to making real time predictions. This makes the KNN algorithm much faster than other algorithms that require training e.g SVM, linear regression, etc.

Since the algorithm requires no training before making predictions, new data can be added seamlessly.

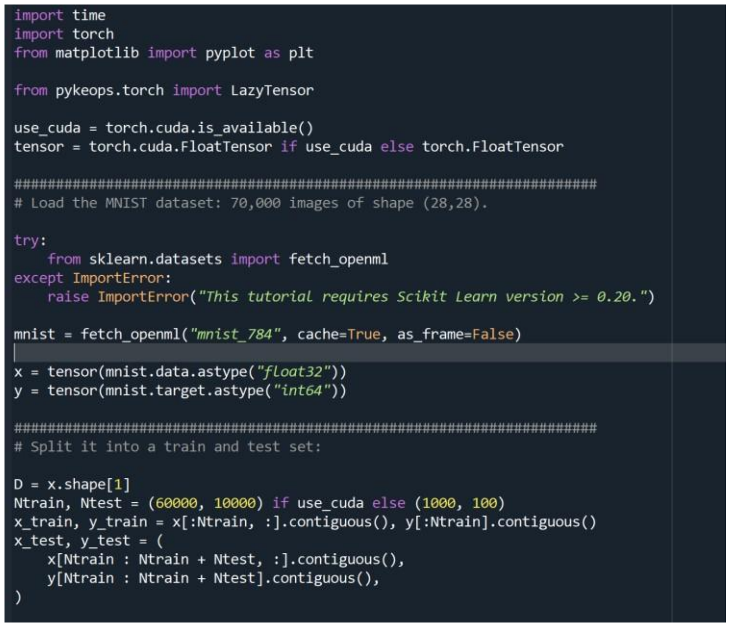
There are only two parameters required to implement KNN i.e. the value of K and the distance function (e.g. Euclidean or Manhattan etc.) **Cons:**

The KNN algorithm doesn't work well with high dimensional data because with large number of dimensions, it becomes difficult for the algorithm to calculate distance in each dimension.

The KNN algorithm has a high prediction cost for large datasets. This is because in large datasets the cost of calculating distance between new point and each existing point becomes higher.

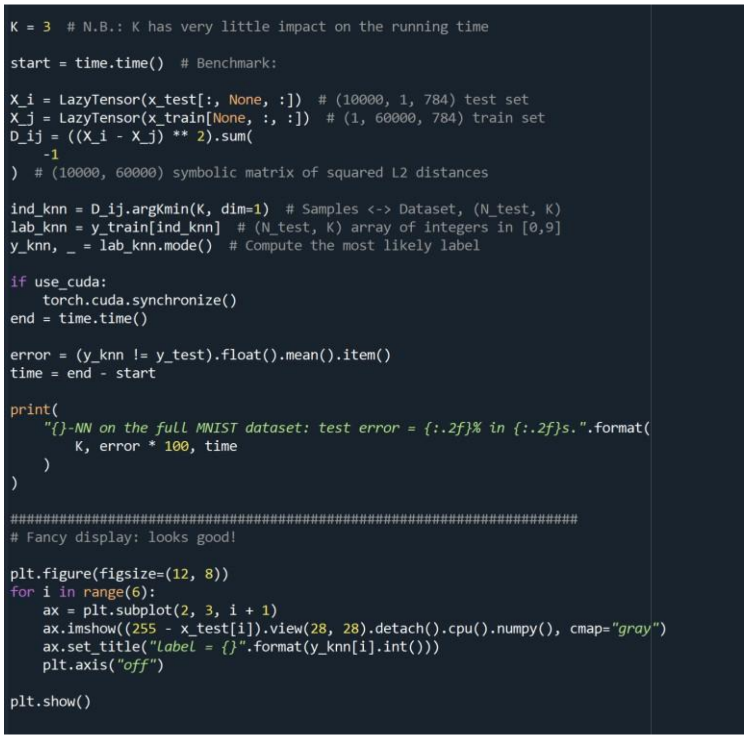
Finally, the KNN algorithm doesn't work well with categorical features since it is difficult to find the distance between dimensions with categorical features.

First like others we should import the dataset and split it :



```python
import time
import torch
from matplotlib import pyplot as plt

from pykeops.torch import LazyTensor

use_cuda = torch.cuda.is_available()
tensor = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor

#######################################################################
# Load the MNIST dataset: 70,000 images of shape (28,28).

try:
    from sklearn.datasets import fetch_openml
except ImportError:
    raise ImportError("This tutorial requires Scikit Learn version >= 0.20.")

mnist = fetch_openml("mnist_784", cache=True, as_frame=False)

x = tensor(mnist.data.astype("float32"))
y = tensor(mnist.target.astype("int64"))

#######################################################################
# Split it into a train and test set:

D = x.shape[1]
Ntrain, Ntest = (60000, 10000) if use_cuda else (1000, 100)
x_train, y_train = x[:Ntrain, :].contiguous(), y[:Ntrain].contiguous()
x_test, y_test = (
    x[Ntrain : Ntrain + Ntest, :].contiguous(),
    y[Ntrain : Ntrain + Ntest].contiguous(),
)
```

After that we have the main KNN algorithm :

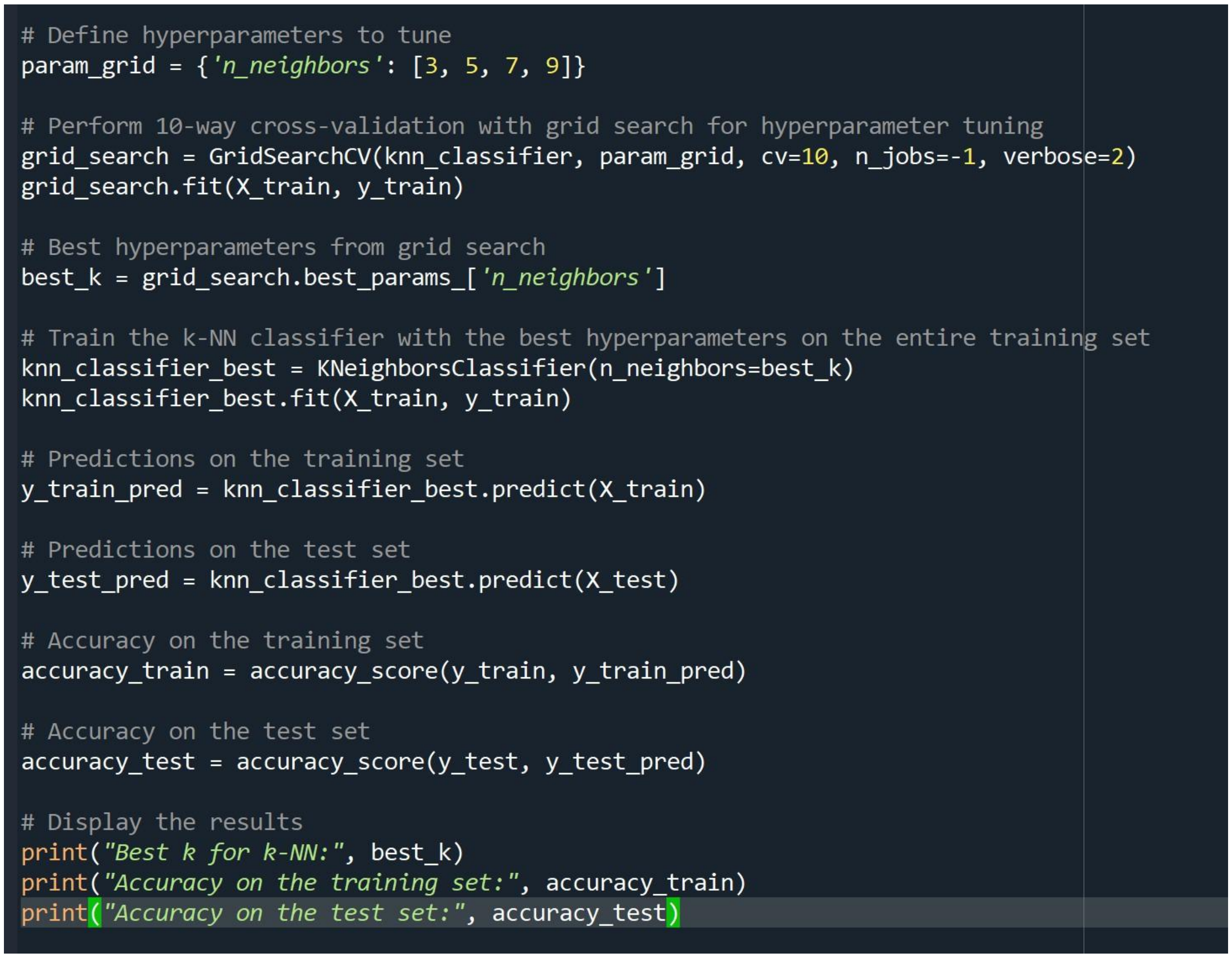


```python
K = 3  # N.B.: K has very little impact on the running time

start = time.time()  # Benchmark:

X_i = LazyTensor(x_test[:, None, :])  # (10000, 1, 784) test set
X_j = LazyTensor(x_train[None, :, :])  # (1, 60000, 784) train set
D_ij = ((X_i - X_j) ** 2).sum(
    -1
)  # (10000, 60000) symbolic matrix of squared L2 distances

ind_knn = D_ij.argKmin(K, dim=1)  # Samples <-> Dataset, (N_test, K)
lab_knn = y_train[ind_knn]  # (N_test, K) array of integers in [0,9]
y_knn, _ = lab_knn.mode()  # Compute the most likely label

if use_cuda:
    torch.cuda.synchronize()
end = time.time()

error = (y_knn != y_test).float().mean().item()
time = end - start

print(
    "{}-NN on the full MNIST dataset: test error = {:.2f}% in {:.2f}s.".format(
        K, error * 100, time
    )
)

#######################################################################
# Fancy display: looks good!

plt.figure(figsize=(12, 8))
for i in range(6):
    ax = plt.subplot(2, 3, i + 1)
    ax.imshow((255 - x_test[i]).view(28, 28).detach().cpu().numpy(), cmap="gray")
    ax.set_title("label = {}".format(y_knn[i].int()))
    plt.axis("off")

plt.show()
```

But in the assignment you wants “you must implement the Naive Bayes and k-NN classifiers yourself.

Use 10 way cross validation to optimize the parameters for each classifier.

Provide the code, the models on the training set, and the respective performances in testing and in 10 way cross validation." So for improving the code we should :



```python
# Define hyperparameters to tune
param_grid = {'n_neighbors': [3, 5, 7, 9]}

# Perform 10-way cross-validation with grid search for hyperparameter tuning
grid_search = GridSearchCV(knn_classifier, param_grid, cv=10, n_jobs=-1, verbose=2)
grid_search.fit(X_train, y_train)

# Best hyperparameters from grid search
best_k = grid_search.best_params_['n_neighbors']

# Train the k-NN classifier with the best hyperparameters on the entire training set
knn_classifier_best = KNeighborsClassifier(n_neighbors=best_k)
knn_classifier_best.fit(X_train, y_train)

# Predictions on the training set
y_train_pred = knn_classifier_best.predict(X_train)

# Predictions on the test set
y_test_pred = knn_classifier_best.predict(X_test)

# Accuracy on the training set
accuracy_train = accuracy_score(y_train, y_train_pred)

# Accuracy on the test set
accuracy_test = accuracy_score(y_test, y_test_pred)

# Display the results
print("Best k for k-NN:", best_k)
print("Accuracy on the training set:", accuracy_train)
print("Accuracy on the test set:", accuracy_test)
```

let's delve into the differences between k-NN (k-Nearest Neighbors) and Naive Bayes models in terms of classification performance and computational requirements: k-NN (k-Nearest Neighbors):

Classification Performance:

Strengths: k-NN is a non-parametric, instance-based algorithm that makes predictions based on the majority class of its k-nearest neighbors in the feature space. It can be effective in capturing complex decision boundaries and works well with locally varying class distributions.

Weaknesses: k-NN might struggle with high-dimensional data or datasets where the relevance of features varies. It is sensitive to the choice of k, and outliers can have a significant impact on predictions.

Computational Requirements:

Training: k-NN has minimal training requirements since it essentially involves storing the training data.

Prediction: Prediction in k-NN can be computationally expensive, especially with large datasets. It requires calculating distances between the query point and all training points. The time complexity is O(Nd), where N is the number of data points and d is the number of features.

Naive Bayes:

Classification Performance:

Strengths: Naive Bayes is a probabilistic, simple classifier based on Bayes' theorem. It assumes independence between features given the class, making it computationally efficient. It can perform well, especially in text classification and simple datasets, where the independence assumption is reasonable.

Weaknesses: Naive Bayes might struggle with complex relationships in the data and may not capture non-linear dependencies.

Computational Requirements:

Training: Naive Bayes has low computational requirements during training. It involves calculating probabilities for each feature, which is a linear process.

Prediction: Prediction in Naive Bayes is also efficient, involving multiplying probabilities. The time complexity is generally O(d), where d is the number of features.
In summary:

k-NN is flexible and effective in capturing complex patterns but can be computationally expensive, particularly during prediction with large datasets.

Naive Bayes is computationally efficient during both training and prediction but makes the assumption of feature independence, which may limit its performance in capturing certain types of relationships in the data.

The choice between k-NN and Naive Bayes depends on the characteristics of the dataset, the nature of the problem, and considerations related to computational resources. For large datasets, k-NN's prediction time complexity may lead to longer computation times compared to Naive Bayes. It's recommended to experiment with both models and assess their performance on the specific task at hand. Cross-validation can aid in obtaining a more robust evaluation of their performance.