Assignment 2: Discriminative and Generative Classifiers

881177 Grosu Victoria

Summary

INTRODUCTION

In this paper we will analyze four machine learning techniques to predict images come from a dataset composed by images 70000 28x28 pixel gray-scale images.

As it seen we have images representing numbers from 0 to 9. Our goal is to create four models that predict the number represented in these images.

However for our purpose the dataset will be converted in a 784-dimensional feature vectors (28*28) of values between 0 (white) and 1 (black).

The technique used in this paper to analyze the dataset and make predictions is called **supervised learning**. Supervised learning consists in learning the link between two datasets: the observed data X and an external variable y that we are trying to predict, usually called "target" or "labels". All supervised estimators implement a fit(X, y) method to fit the model and a predict(X) method that, given unlabeled observations X, returns the predicted labels y.

We will train this following classifiers on the dataset:

1. **Support-Vector Machine** (SVM) using linear, polynomial of degree 2, and RBF kernels – (sklearn)
2. **Random forests** – (sklearn)
3. **Naive Bayes classifier** where each pixel is distributed according to a Beta distribution of parameters α and β – (my code)
4. **K-nearest neighbors** (k-NN) – (my code)

Note: due to poor performance of my device for cross validation I will use X_train and y_train of size 10000. Instead to analyze the models I will take an X_train, y_train of size 60000 and X_test, y_test of size 10000.

To see the differences between the various models we would see the following analyses:

1- **K-Fold Cross Validation**, with k = 10, to tune the parameters. In K Fold cross validation, the data is divided into k subsets. Each time, one of the k subsets is used as the test set/ validation set and the other k-1 subsets are put together to form a training set. The error estimation is averaged over all k trials to get total effectiveness of our model.
2- **Fit** on training set and **predict** on testing set and **accuracy**, that tells the percentage of correctly predicted values out of all the data points.
3- A **confusion matrix** is a tool for summarizing the performance of a classification algorithm. A confusion matrix will give us a clear picture of classification model performance and the types of errors produced by the model. It gives us a summary of correct and incorrect predictions broken down by each category. The summary is represented in a tabular form. Given below is the description of each cell:

  - *TP (True Positives):* Actual positives in the data, which have been correctly predicted as positive by our model.
  - *TN (True Negatives):* Actual Negatives in the data, which have been correctly predicted as negative by our model.
  - *FP (False Positives):* Actual Negatives in data, but our model has predicted them as Positive.
  - *FN (False Negatives):* Actual Positives in data, but our model has predicted them as Negative.

Discriminative and Generative models.

Machine learning models can be classified into two types of models – **Discriminative** and **Generative** models. In simple words, a discriminative model makes predictions on the unseen data based on conditional probability and can be used either for classification or regression problem statements. On the contrary, a generative model focuses on the distribution of a dataset to return a probability for a given example.

- **Discriminative** Models:

The discriminative model refers to a class of models used in Statistical Classification, mainly used for supervised machine learning. These types of models are also known as conditional models since they learn the *boundaries* between classes or labels in a dataset. Discriminative models (just as in the literal meaning) separate classes instead of modeling the conditional probability and *don't make any assumptions* about the data points. But these models are not capable of generating new data points. Therefore, the ultimate objective of discriminative models is to separate one class from another.

If we have some outliers present in the dataset, then discriminative models work better compared to generative models i.e., discriminative models are more robust to outliers. However, there is one major drawback of these models is the misclassification problem, i.e., wrongly classifying a data point.

Some Examples of Discriminative Models:

- Scalar Vector Machine (SVMs)
- Nearest neighbor
- Decision Trees and Random Forest

- **Generative** Models:

Generative models are considered as a class of statistical models that can generate new data instances. These models are used in unsupervised machine learning as a means to perform tasks such as:

- Probability and Likelihood estimation,
- Modeling data points,
- To describe the phenomenon in data,
- To distinguish between classes based on these probabilities.

Since these types of models often rely on the *Bayes theorem* to find the joint probability, so generative models can tackle a more complex task than analogous discriminative models.

So, Generative models focus on the distribution of individual classes in a dataset and the learning algorithms tend to model the underlying patterns or distribution of the data points. These models use the concept of joint probability and create the instances where a given feature (x) or input and the desired output or label (y) exist at the same time.

These models use probability estimates and *likelihood* to model data points and differentiate between different class labels present in a dataset. Unlike discriminative models, these models are also capable of generating new data points. However, they also have a major drawback – If there is a presence of outliers in the dataset, then it affects these types of models to a significant extent.

Some Examples of Generative Models: Naïve Bayes Classifier

- Difference between discriminative and generative:

Discriminative models draw boundaries in the data space, while **generative** models try to model how data is placed throughout the space. A generative model focuses on explaining how the data was generated, while a discriminative model focuses on predicting the labels of the data.

In mathematical terms, a **discriminative** machine learning trains a model which is done by learning parameters that maximize the *conditional probability* P(Y|X), while on the other hand, a **generative** model learns parameters by maximizing the *joint probability* of P(X, Y).

Discriminative models recognize existing data i.e., discriminative modeling identifies tags and sorts data and can be used to classify data while **Generative** modeling produces something. Since these models use different approaches to machine learning, so both are suited for specific tasks i.e., Generative models are useful for unsupervised learning tasks while discriminative models are useful for supervised learning tasks.

Generative models have more impact on outliers than **discriminative** models.

Discriminative models are computationally cheap as compared to **generative** models.

SUPPORT VECTOR MACHINES (SVMs)

SVM is a supervised machine learning algorithm that aims to find an optimal boundary between the possible outputs. Simply put, SVM does complex data transformations depending on the selected kernel function and based on that transformations, it tries to maximize the separation boundaries between your data points depending on the labels or classes you've defined. The data points with the minimum distance to the boundary (closest points) are called Support Vectors.

The computations of data points separation depend on a kernel function. There are different kernel functions in this paper we will see which is the best between Linear, Polynomial and Radial Basis Function (RBF). Simply put, these functions determine the smoothness and efficiency of class separation, and playing around with their hyperparameters may lead to overfitting or underfitting.

Linear Kernel is used when the data is Linearly separable, that is, it can be separated using a single Line. It is one of the most common kernels to be used. It is mostly used when there are a Large number of Features in a particular Data Set.

RBF kernels are the most generalized form of kernelization and is one of the most widely used kernels due to its similarity to the Gaussian distribution. The RBF kernel function for two points X_1 and X_2 computes the similarity or how close they are to each other. The maximum value that the RBF kernel can be is 1 and occurs when d_{12} is 0 which is when the points are the same, i.e. $X_1 = X_2$.

- When the points are the same, there is no distance between them and therefore they are extremely similar
- When the points are separated by a large distance, then the kernel value is less than 1 and close to 0 which would mean that the points are dissimilar

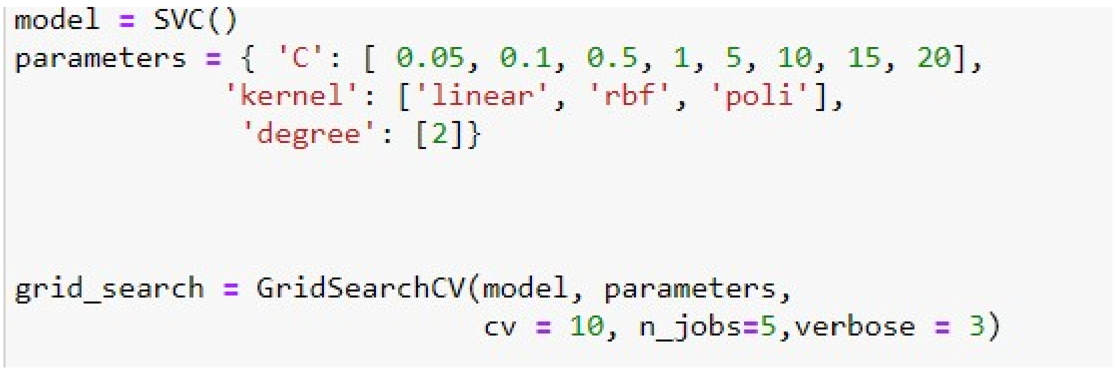
In a **polynomial kernel** for SVM, the data is mapped into a higher-dimensional space using a polynomial function. The dot product of the data points in the original space and the polynomial function in the new space is then taken. The polynomial kernel is often used in SVM classification problems where 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.

Cross validation:

Taking all the values of C and checking out the accuracy score with kernel as *linear*, *RBF* and *poli* with degree equal to 2.

C = the **Regularization Parameter** tells the SVM optimization how much you want to avoid miss classifying each training example.

- If the C is higher, the optimization will choose smaller margin hyperplane, so training data miss classification rate will be lower.
- On the other hand, if the C is low, then the margin will be big, even if there will be miss classified training data examples.



```
model = SVC()
parameters = { 'C': [ 0.05, 0.1, 0.5, 1, 5, 10, 15, 20],
              'kernel': ['linear', 'rbf', 'poli'],
               'degree': [2]}



grid_search = GridSearchCV(model, parameters,
                           cv = 10, n_jobs=5,verbose = 3)
```

Result:

- Time: 26,34 minutes
- Best Score: 96,7% of accuracy
- Best Parameters: C = 15, kernel = RBF

After the analysis of the cross validation it can be seen that the best kernel to build the model that will study our data set is RBF, with regularization parameter equal to 15.

So, the rule of thumb is: use linear SVMs for linear problems, and nonlinear kernels such as the Radial Basis Function (RBF) kernel for non-linear problems.

The RBF kernel SVM decision region is actually also a linear decision region. What RBF kernel SVM actually does is to create non-linear combinations of your features to uplift your samples onto a higher-dimensional feature space where you can use a linear decision boundary to separate your classes. This means that our dataset is not linearly separable in the current dimension, but thanks to the RBF kernel it is possible to find a higher dimension where to linearly separate it.

Fit-Predict:

Step 1: FIT

```
svc = SVC(C = 15, kernel = 'rbf')
svc.fit( X_train, y_train)
```

Result:

- Time: 3,24 minutes

Step 2: PREDICT on Train

```
y_pred = svc.predict( X_train )
```

Result:

- Time: 8,32 minutes
- Accuracy: 100%

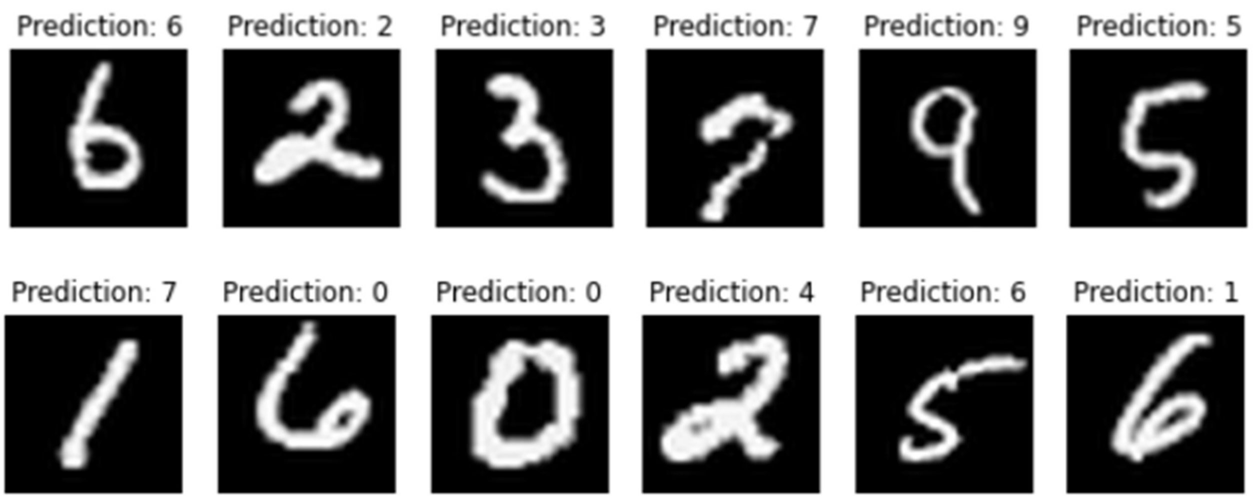
Step 3: PREDICT on Test

```
y_pred = svc.predict( X_test )
```

Result:

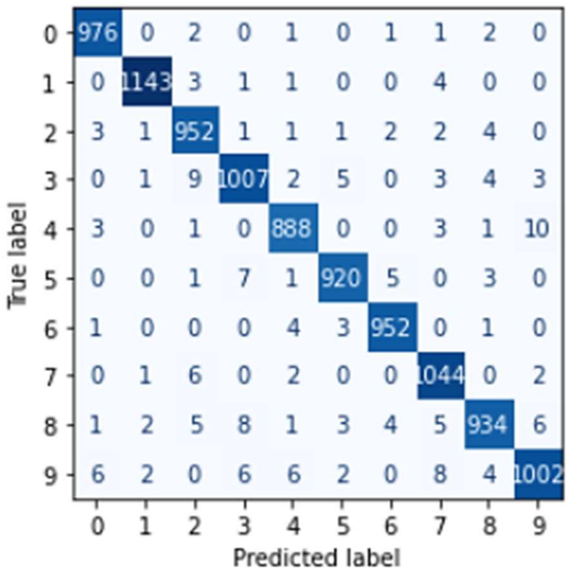
- Time: 1,30 minutes
- Accuracy: 98,2 %

Some prediction examples:



As we can see some numbers are misclassified. The misclassifications are only for numbers that may resemble other numbers. For example the 1 that got misclassified as 7, the line is positioned as 7 without the horizontal line. Or the number 6 that got misclassified as 0, it only need a line to close the circle and be a 0.

Confusion Matrix


True label
Predicted label

	0	1	2	3	4	5	6	7	8	9
0	976	0	2	0	1	0	1	1	2	0
1	0	1143	3	1	1	0	0	4	0	0
2	3	1	952	1	1	1	2	2	4	0
3	0	1	9	1007	2	5	0	3	4	3
4	3	0	1	0	888	0	0	3	1	10
5	0	0	1	7	1	920	5	0	3	0
6	1	0	0	0	4	3	952	0	1	0
7	0	1	6	0	2	0	0	1044	0	2
8	1	2	5	8	1	3	4	5	934	6
9	6	2	0	6	6	2	0	8	4	1002

The image shows the matrix confusion of the SVM model with RBF kernel applied to our dataset.

The confusion matrix shows us excellent results, the values predicted as one class instead of another are very few. The highest value can be seen in the 10 images predicted as 9 but in reality they were 4 and in the 9 images predicted as 2 but in reality they were 3. These two were the peaks, even if not too high.

Some cells are also equal to 0, this means that there is no 'confusion' between the predictions of the classes with others in these images.

The numbers he predicted more correctly than the others were 1 , 7 and 9.

Considerations:

SVM works relatively well when there is a clear margin of separation between classes and is more effective in high dimensional spaces, this is probably why the cross validation returned “RBF” as the kernel.

However SVM algorithm is not suitable for large data sets, the computation time is very high, both for cross validation and for train and predict. In some cases the images are similar to each other and this causes the SVM does not perform very well and lead to a wrong prediction.

RANDOM FOREST

A random forest algorithm consists of many decision trees. The ‘forest’ generated by the random forest algorithm is trained through bagging or bootstrap aggregating. Bagging is an ensemble meta-algorithm that improves the accuracy of machine learning algorithms.

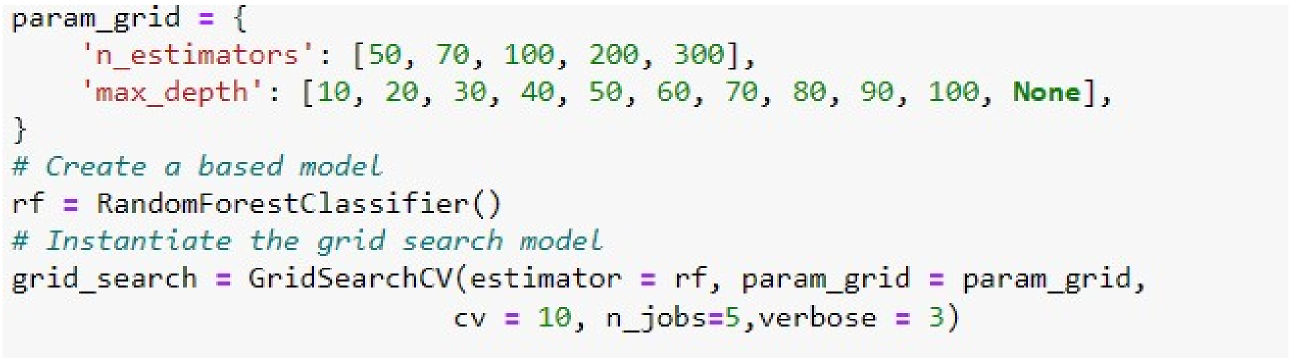
Ensemble methods are techniques that create multiple models and then combine them to produce improved results. Ensemble methods usually produces more accurate solutions than a single model would. The models used to create such ensemble models are called ‘base models’.

The (random forest) algorithm establishes the outcome based on the predictions of the decision trees. It predicts by taking the average or mean of the output from various trees. Increasing the number of trees increases the precision of the outcome. A random forest eradicates the limitations of a decision tree algorithm. It reduces the overfitting of datasets and increases precision.

Cross validation:

n_estimators = This is the number of trees you want to build before taking the maximum voting or averages of predictions. Higher number of trees give you better performance but makes your code slower. You should choose as high value as your processor can handle because this makes your predictions stronger and more stable.

max_depth = This parameter restricts the number of levels of each tree. Creating more levels increases the possibility of considering more features in each tree. A deep tree would create an overfit model, but in Random forest this would be overcome as we would ensemble at the end.



```
param_grid = {
    'n_estimators': [50, 70, 100, 200, 300],
    'max_depth': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, None],
}
# Create a based model
rf = RandomForestClassifier()
# Instantiate the grid search model
grid_search = GridSearchCV(estimator = rf, param_grid = param_grid,
                          cv = 10, n_jobs=5,verbose = 3)
```

Result:

- Time: 19,72 minutes
- Best Score: 95,2% of accuracy
- Best Parameters: max_depth = 100, n_estimators = 300

Fit-Predict:

Step 1: FIT

```
rf = RandomForestClassifier(n_estimators = 300, max_depth = 100)
rf.fit( X_train, y_train)
```

Result:

- Time: 3,71 minutes

Step 2: PREDICT on train

```
y_pred = rf.predict( X_train )
```

Result:

- Time: 11,38 seconds
- Accuracy: 100%

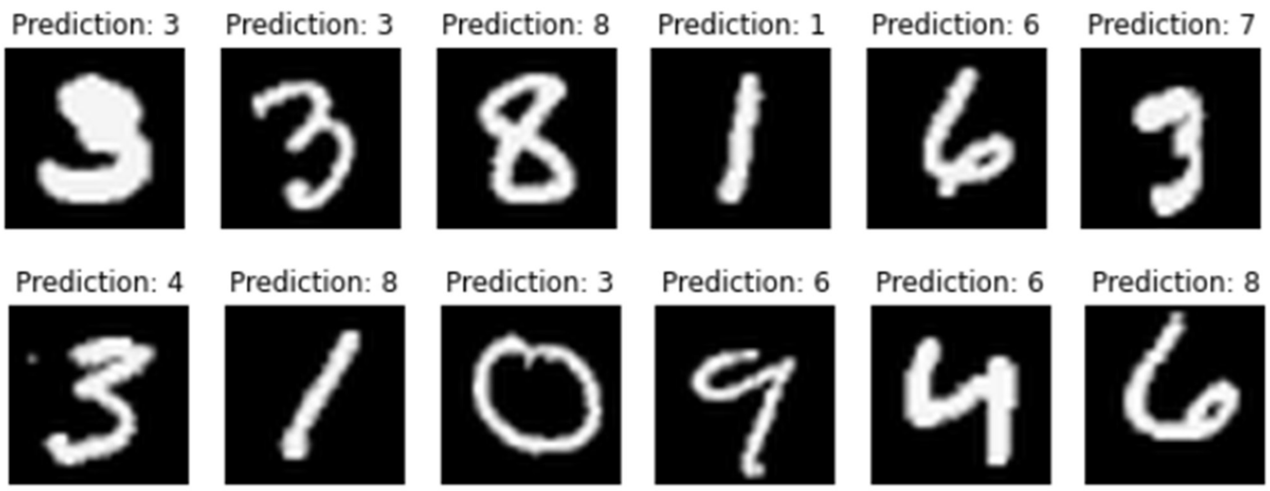
Step 3: PREDICT on test

```
y_pred = rf.predict( X_test )
```

Result:

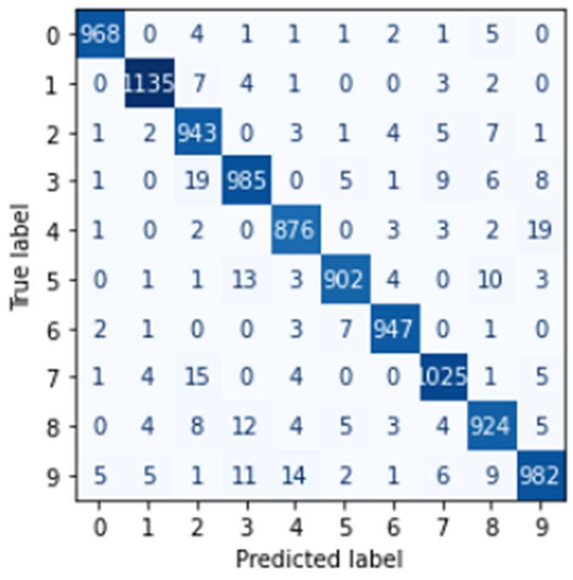
- Time: 1,29 seconds
- Accuracy: 96,8%

Some prediction examples:



As we can see from above, some images have not been classified correctly. However it is not possible to understand the causes because the creation of the random forest is like a "black box", we cannot really understand how the 300 trees were built to create the forest.

Confusion Matrix



True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	968	0	4	1	1	1	2	1	5	0
1	0	1135	7	4	1	0	0	3	2	0
2	1	2	943	0	3	1	4	5	7	1
3	1	0	19	985	0	5	1	9	6	8
4	1	0	2	0	876	0	3	3	2	19
5	0	1	1	13	3	902	4	0	10	3
6	2	1	0	0	3	7	947	0	1	0
7	1	4	15	0	4	0	0	1025	1	5
8	0	4	8	12	4	5	3	4	924	5
9	5	5	1	11	14	2	1	6	9	982

The image shows the matrix confusion of the random forest model applied to our dataset. as we can see it is much worse than the confusion matrix of the SVM model, the false predictions are much more numerous in terms of number and classes predicted as other classes. In this case we note that 19 times an image representing the number 4 was predicted as 9. The same number of times an image representing the number 3 was predicted as 2. 15 times an image representing a 7 was predicted as a 2. 14 times an image with a 9 was predicted as a 4.

The numbers he predicted more correctly than the others were 1 and 7.

Considerations:

Random Forest is based on the bagging algorithm and uses Ensemble Learning technique. It creates as many trees on the subset of the data and combines the output of all the trees. In this way it reduces overfitting problem in decision trees and also reduces the variance and therefore improves the accuracy. Random Forest is usually robust to outliers and can handle them automatically. Random Forest algorithm is very stable. Even if a new data point is introduced in the dataset, the overall algorithm is not affected much since the new data may impact one tree, but it is very hard for it to impact all the trees.

However we have a problem with computation time, Random Forest creates a lot of trees (unlike only one tree in case of decision tree) and combines their outputs. To do so, this algorithm requires much more computational power and resources. On the other hand decision tree is simple and does not require so much computational resources.

Another problem is that Random forest is like a black box algorithm, you have very little control over what the model does.

K-NEAREST NEIGHBORS

The k-nearest neighbors algorithm, also known as KNN, is a supervised learning classifier, which uses proximity to make classifications or predictions about the grouping of an individual data point. While it can be used for either regression or classification problems, it is typically used as a classification algorithm, working off the *assumption that similar points can be found near one another.*

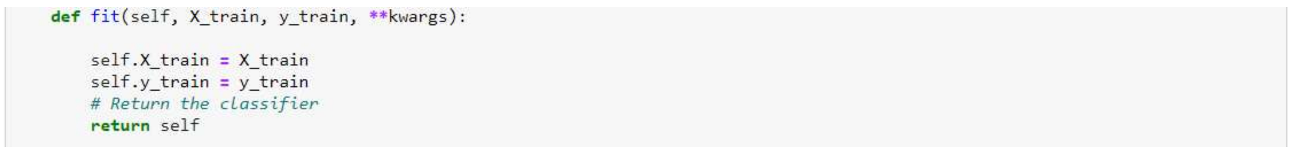
For classification problems, a class label is assigned on the basis of a majority vote—i.e. the label that is most frequently represented around a given data point is used. While this is technically considered "plurality voting", the term, "majority vote" is more commonly used in literature.

This algorithm is used in cases where the Linear and Logistic Regression cannot be implemented i.e when the data is non-linear separable . Non-linear data is a type of data in which the dependency of dependent variable (y) on the independent variable (x) is not linear . In other words , we cannot draw a best fit straight linear regression line for non-linear separable data .

Implementation

Step 1: Fit()

To do the fit we only save our training set.



```
def fit(self, X_train, y_train, **kwargs):

    self.X_train = X_train
    self.y_train = y_train
    # Return the classifier
    return self
```

Step 1: Predict()

To do the predict we do for each row in the testing set se following operations :

First calculate the Euclidian distance between the row and the *training set.* The Euclidian distance is calculated by the following function:

```python
def euclidean(point, data):
    # Euclidean distance between points a & data
    return np.sqrt(np.sum((point - data)**2, axis=1))
```

The KNN algorithm first finds out the nearest "K" number of neighbors, this means that we take the indexes of the k rows in the training set closest in distance to the row we are testing. After that, use the found indexes to get the corresponding labels. Whichever label has the highest frequency is allotted as the prediction for the new point .

Let's see an *example* how does this work:

```python
A = [[0], [1], [2], [3], [4], [5]]
a = [2]
```

A is a matrix composed by six rows and *a* is a row. We compute the distance between *a* and all the rows of the matrix *A*. The result: [2., 1., 0., 1., 2., 3.]

Next sort the neighbors in ascending order based on their distance from the new point and take the k = 3 sorted nearest neighbors, so we will take the ones with the smallest distance. The indexes of the nearest rows are: [2, 1, 3].

In the end find the label with highest frequency and give that label as output prediction for the new point .

```python
    def predict(self, X_test):
        y_pred = []
        indexes = []
        count = 0
        for index, x in X_test.iterrows():
            count += 1
            if count == 1000:
                count = 0
                print('you did 1000 iterrations')
            x = np.asarray(x)
            distances = euclidean(x, self.X_train)

            nearest_neighbor_ids = distances.argsort().to_numpy()[0:self.k]
```

Returning to our code, after calculating the Euclidean distance and finding the indexes corresponding to the k closest points, we can use them to find the labels using the training set which contains only the classes.

We may have several labels, we must take only the most frequent label. So we use "*np.unique*" to extract the labels and their frequency, and use majority vote to extract the predict of the row *x* of the testing set.

```
            nearest_neighbor_rings = self.y_train.iloc[nearest_neighbor_ids]
            #find unique values in array along with their counts
            vals, counts = np.unique(nearest_neighbor_rings, return_counts=True)

            #find mode
            mode_value = np.argwhere(counts == np.max(counts))

            #mode
            pred = vals[mode_value].flatten().tolist()[0]

            y_pred.append(pred)
            indexes.append(index)
        return pd.Series(data=y_pred,index=indexes)
```

Thus by running this code for each line of the testing set it is possible to predict its class.

Cross validation:

K = the number of neighbors to be used in the majority vote while deciding the class. As K approaches 1, your prediction becomes less stable.

- As your value of K increases, your prediction becomes more stable due to the majority of voters.
- When you start receiving an increasing number of errors, you should know you are pushing your K too far.

```
param_grid = {
    'k': [3, 5, 7, 10, 15, 20]
}
# Create a based model
kNN = my_KNN()
# Instantiate the grid search model
grid_search = GridSearchCV(estimator = kNN, param_grid = param_grid,
                           cv = 10, n_jobs=5,verbose = 3)
```

Result:

- Time: 46,80 minutes
- Best Score: 94,6% of accuracy
- Best Parameters: k = 3

Fit-Predict:

Step 1: FIT

```
knn = my_KNN(k = 3)
knn.fit(X_train, y_train)
```

Result:

- Time: 0 seconds

Step 2: PREDICT on train

```
y_pred = knn.predict( X_test )
```

Result:

- Time: more than 3 hours
- Accuracy: 100%

<u>Note</u>: on the notebook there isn't the output cause I delete it by mistake and redoing all the predict again requires too much time. However, it is obvious that 100% accuracy is achieved since the training set is present in the fit and therefore analyzes the distance with itself. This prediction is practically useless.

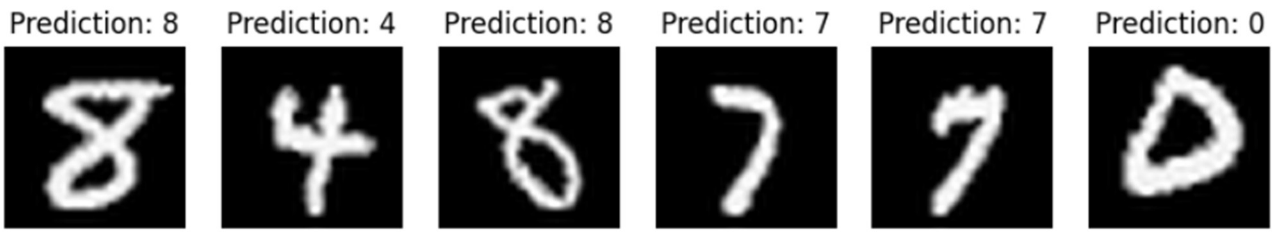
Step 3: PREDICT on test

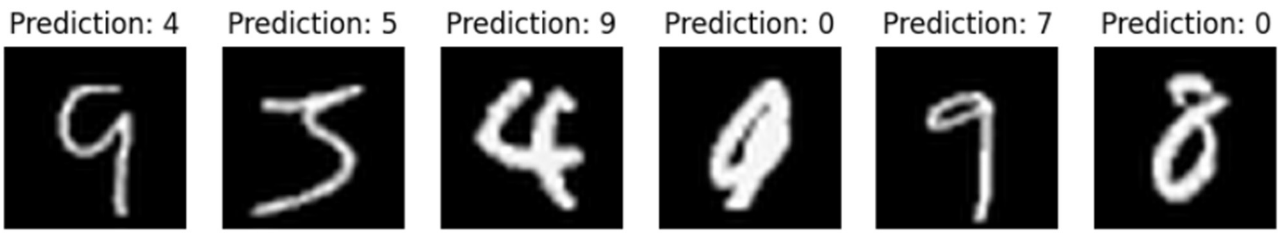
```
y_pred = knn.predict( X_test )
```

Result:

- Time: 38,44 minutes
- Accuracy: 96,8%

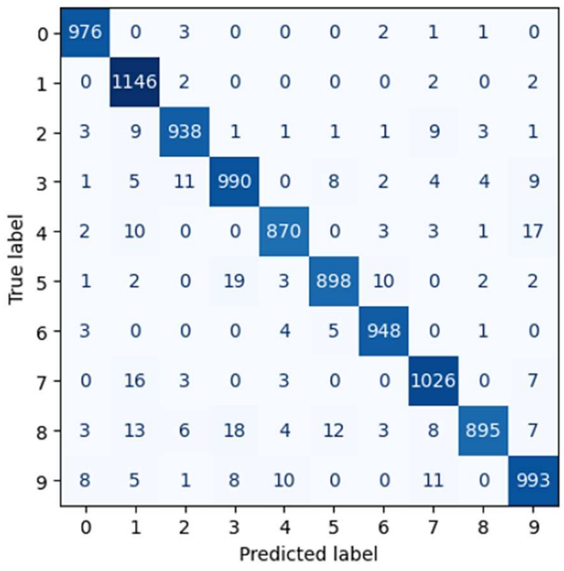
Some prediction examples:





In the case of misclassification it is similar to the SVM model. If we look at the incorrectly predicted images it can be seen that they can clearly be mistaken for other numbers. If we look at the predicted 3 as 5, even a human could be mistaken. Same goes for the number 9 misclassified as 0. The other wrong predictions are more visible to the human eye, however the similarity to other numbers is very clear.

Confusion Matrix



True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	976	0	3	0	0	0	2	1	1	0
1	0	1146	2	0	0	0	0	2	0	2
2	3	9	938	1	1	1	1	9	3	1
3	1	5	11	990	0	8	2	4	4	9
4	2	10	0	0	870	0	3	3	1	17
5	1	2	0	19	3	898	10	0	2	2
6	3	0	0	0	4	5	948	0	1	0
7	0	16	3	0	3	0	0	1026	0	7
8	3	13	6	18	4	12	3	8	895	7
9	8	5	1	8	10	0	0	11	0	993

The image shows the matrix confusion of the KNN model applied to our dataset.

In this case the confusion matrix in terms of number of classes not correctly predicted is similar to the random forest model, even if the classes predicted with wrong labels are different. The KNN model 19 times predicted an image 5 as 3. 18 times it predicted an image representing an 8 as 3. 17 times it predicted 4 as 9. and 16 times it predicted 7 as 1. These 3 were the peaks, even if not too high, but higher than the SVM model.

The numbers he predicted more correctly than the others were 1 and 7.

Considerations:

KNN modeling does not include training period as the data itself is a model which will be the reference for future prediction and because of this it is very time efficient in term of improvising for a random modeling on the available data. As there is no training period thus new data can be added at any time since it won't affect the model.

KNN is very easy to implement as the only thing to be calculated is the distance between different points on the basis of data of different features and this distance can easily be calculated using distance formula such as Euclidian or Manhattan.

However does not work well with our dataset because is very large and calculating distances between each data instance it's very costly. The execution time is one of the highest seen so far. In fact, the calculation time between an array of length 784 and a dataset of 60,000 takes a long time.

NAIVE BAYES CLASSIFIER

It is a classification technique based on Bayes' Theorem with an *assumption of independence among predictors*. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature.

Bayes theorem provides a way of calculating *posterior probability P(c/x)* from *P(c)*, *P(x)* and *P(x/c)*. Look at the equation below:

$$P(c|x) = \frac{P(x|c)P(c)}{P(x)}$$

Above,

- *P(c/x)* is the *posterior probability* of class (c, target) given predictor (x, attributes).
- *P(c)* is the *prior probability of class.*
- *P(x/c)* is the *likelihood* which is the probability of predictor given class.
- *P(x)* is the *prior probability of predictor.*

In this paper to calculate the likelihood *P(x/c)* we will use the *Beta distribution.*

The probability density function (PDF) of the Beta distribution, for $0 \le x \le 1$, and shape parameters α, β > 0, is a power function of the variable *x* and of its reflection (1 − *x*) as follows:

$$f(x,\alpha,\beta) = \frac{1}{B(\alpha,\beta)}\, x^{\alpha-1}\,(1-x)^{\beta-1}$$

The cumulative distribution function (CDF) is:

$$f(x,\alpha,\beta) = \frac{B(x,\alpha,\beta)}{B(\alpha,\beta)}$$

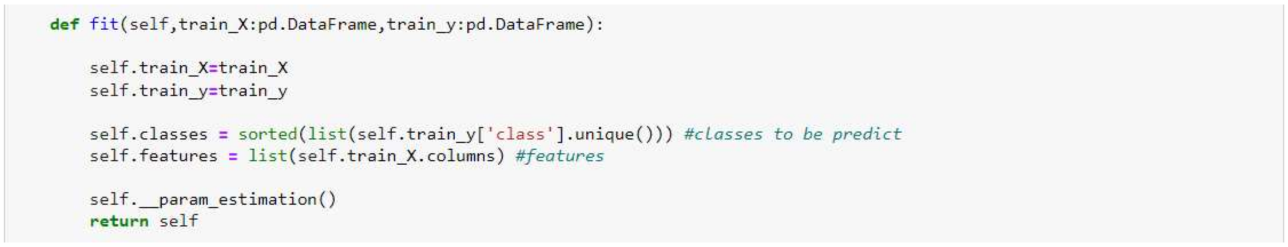
Mean: $E[X] = \frac{\alpha}{\alpha+\beta}$

Variance: $Var[X] = \frac{\alpha\beta}{(\alpha+\beta+1)(\alpha+\beta)^2}$

Implementation

Step 1: Fit()

The image below shows the function fit. Obviously we save our train, our classes to predict and features to be used when we do the predictions. Then we call the function __param_estimation().



```
def fit(self,train_X:pd.DataFrame,train_y:pd.DataFrame):

    self.train_X=train_X
    self.train_y=train_y

    self.classes = sorted(list(self.train_y['class'].unique())) #classes to be predict
    self.features = list(self.train_X.columns) #features

    self.__param_estimation()
    return self
```

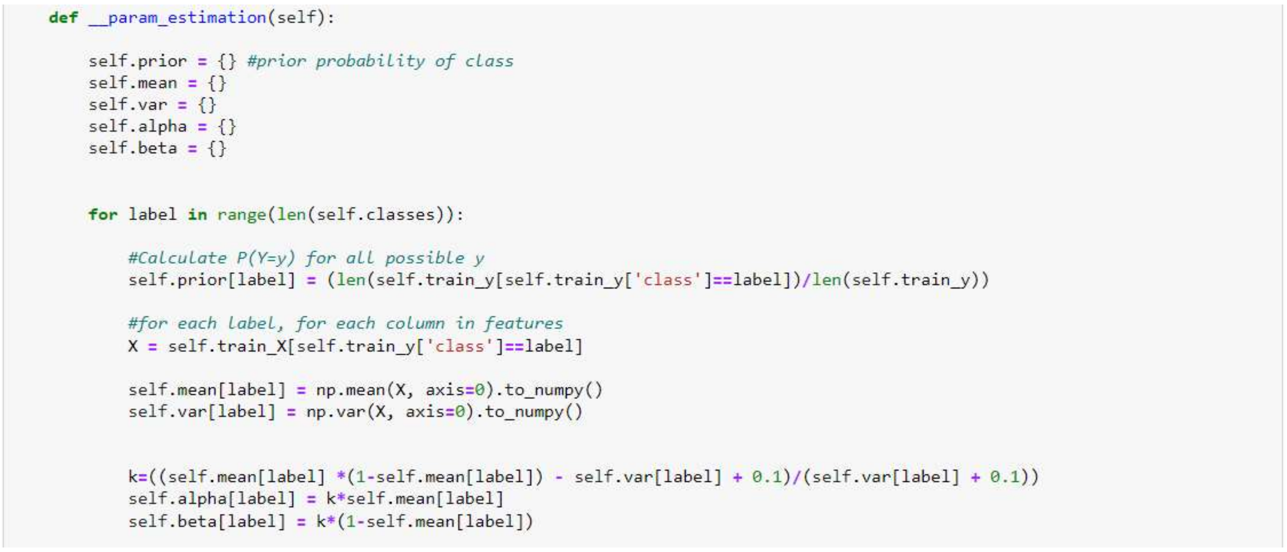
The function __param_estimation() first thing calculates *P(c)* the *prior probability* of each class and save this result in an array. Next for each subset containing the rows which have the same label calculates an array of size 784 which contains the *mean* and the *variance* of each feature of the data set. These arrays are used to compute *k* which will be used to compute alpha and beta. To calculate *k* we use the formula:

$$k = mean * (1 - mean) - var + \varepsilon/var + \varepsilon$$

This because in some cases the variance is equal to 0, to solve this problem it sum an epsilon, "*0.1*" in this case, this is called "*variance smoothing*". For each subset of labels it compute also *alpha* and *beta* using the following formulas:

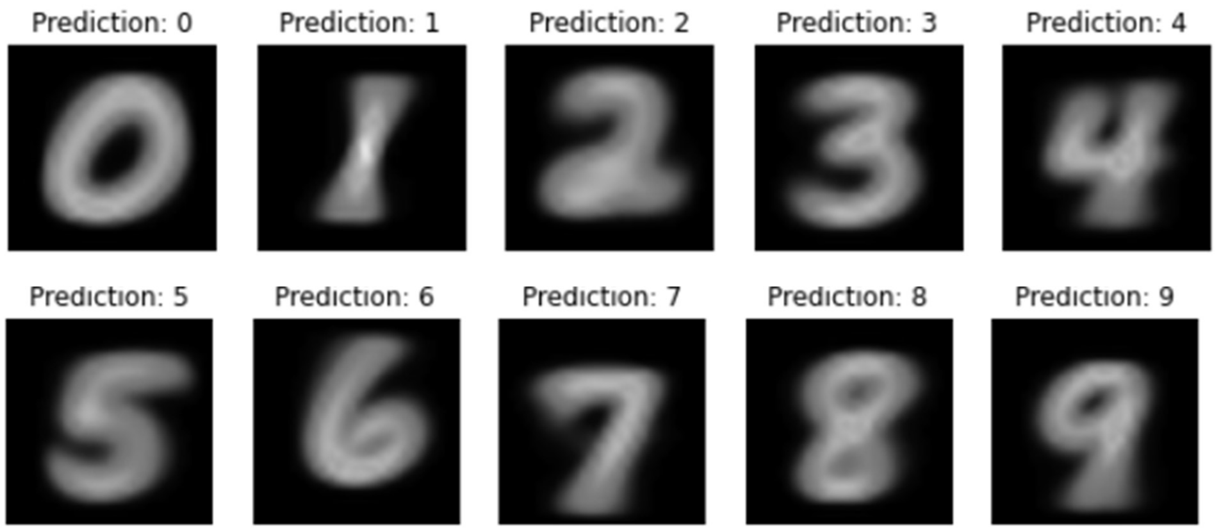
$$\alpha = k * mean \qquad \beta = k * (1 - mean)$$

As we see $\alpha > 0$ and $\beta > 0$, however cause in some cases mean is equal to 0, the result of the Beta distribution may be equal to NAN, however we will just ignore it. This is caused by our dataset. Some features are always equal to 0.



```
def __param_estimation(self):

    self.prior = {} #prior probability of class
    self.mean = {}
    self.var = {}
    self.alpha = {}
    self.beta = {}


    for label in range(len(self.classes)):

        #Calculate P(Y=y) for all possible y
        self.prior[label] = (len(self.train_y[self.train_y['class']==label])/len(self.train_y))

        #for each label, for each column in features
        X = self.train_X[self.train_y['class']==label]

        self.mean[label] = np.mean(X, axis=0).to_numpy()
        self.var[label] = np.var(X, axis=0).to_numpy()


        k=((self.mean[label] *(1-self.mean[label]) - self.var[label] + 0.1)/(self.var[label] + 0.1))
        self.alpha[label] = k*self.mean[label]
        self.beta[label] = k*(1-self.mean[label])
```

Note: α/(α+β) is the mean of the beta distribution. if you compute the mean for each of the 784 models and reshape them into 28x28 images you can have a visual indication of what the model is learning:



Step 2: Predict()

The image below shows the function predict. For each row in our test set the function calculates the likelihood. Calculating the probability or likelihood of observing a given real-value like x is difficult. In our case we assume that x values are drawn from a Beta distribution.

However for our problem we don't use the PDF cause some result in output may be -INF or + INF, so to overcome this problem we use CDF with epsilon equal to 0.1. Cumulative distribution function (CDF) of a real-valued random variable X evaluated at x, is the probability that X will take a value less than or equal to x.

As we seen in before alpha can be equal to 0 and this cause NAN as the result of the function, so to overcome this problem we will substitute all NAN values with 1, so it doesn't count in the product.

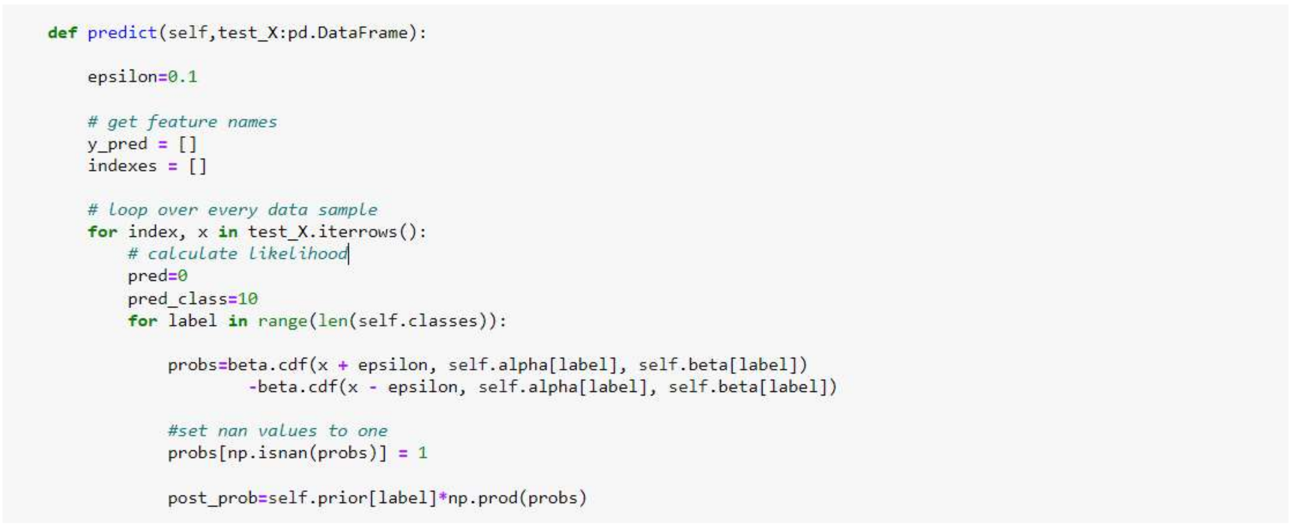
Now that we have all the pieces in place, let's see how we can calculate the probabilities we need for the Naive Bayes classifier. Now it is time to use the statistics calculated from our training data to calculate probabilities for new data.

Probabilities are calculated separately for each class. This means that we first calculate the probability that a new piece of data belongs to the first class, then calculate probabilities that it belongs to the second class, and so on for all the classes.

The probability that a piece of data belongs to a class is calculated as follows:

$$P(Class|Pixel1, Pixel2, \ldots, Pixel784) = P(Pixel1, Pixel2, \ldots, Pixel784|Class) * P(Class)$$

You may note that this is different from the Bayes Theorem described above. The division has been removed to simplify the calculation. This means that the result is no longer strictly a probability of the data belonging to a class. The value is still maximized, meaning that the calculation for the class that results in the largest value is taken as the prediction. This is a common implementation simplification as we are often more interested in the class prediction rather than the probability.



```
def predict(self,test_X:pd.DataFrame):

    epsilon=0.1

    # get feature names
    y_pred = []
    indexes = []

    # loop over every data sample
    for index, x in test_X.iterrows():
        # calculate likelihood
        pred=0
        pred_class=10
        for label in range(len(self.classes)):

            probs=beta.cdf(x + epsilon, self.alpha[label], self.beta[label])
                    -beta.cdf(x - epsilon, self.alpha[label], self.beta[label])

            #set nan values to one
            probs[np.isnan(probs)] = 1

            post_prob=self.prior[label]*np.prod(probs)
```

Now, use Naive Bayesian equation to calculate the posterior probability for each class. The class with the highest posterior probability is the outcome of prediction.

```
                if post_prob > pred:
                    pred=post_prob
                    pred_class=label


            y_pred.append(pred_class)
            indexes.append(index)

        return pd.Series(data=y_pred,index=indexes)
```

Cross validation:

We don’t do cross validation cause we don’t have parameters to tune.

Fit-Predict:

Step 1: FIT

```
betaD = BetaDistribution_NaiveBayes()
betaD.fit(X_train, y_train)
```

Result:

- Time: 0 seconds

Step 2: PREDICT on Train

```
y_train_pred = betaD.predict(X_train)
```

Result:

- Time: 8,87 minutes
- Accuracy: 83,4%

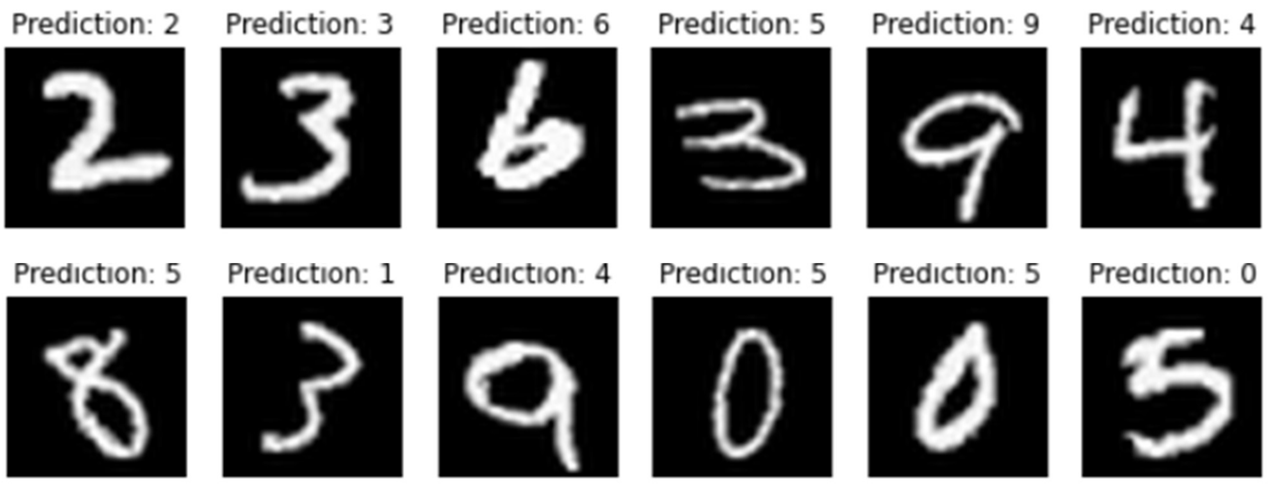
Step 3: PREDICT on Test

```
y_pred = betaD.predict(X_test)
```

Result:

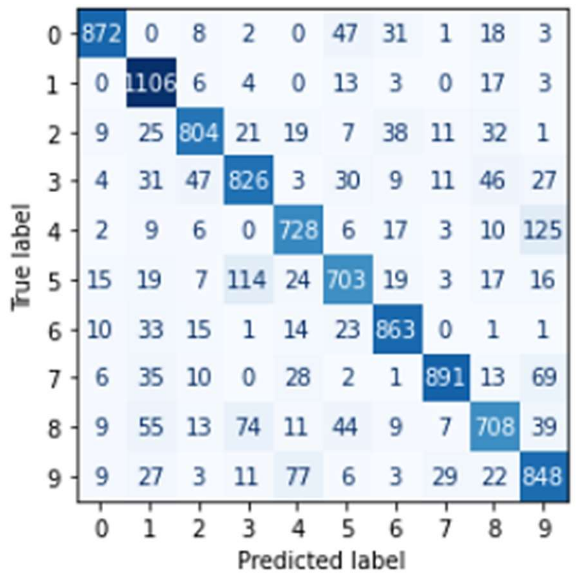
- Time: 1,60 minutes
- Accuracy: 83,5%

Some prediction examples:



If the number in the image doesn't follow the mean enforced by the training set, the Bayes classifier may misclassify because it doesn't correctly follow the beta distribution.

Confusion Matrix



True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	872	0	8	2	0	47	31	1	18	3
1	0	1106	6	4	0	13	3	0	17	3
2	9	25	804	21	19	7	38	11	32	1
3	4	31	47	826	3	30	9	11	46	27
4	2	9	6	0	728	6	17	3	10	125
5	15	19	7	114	24	703	19	3	17	16
6	10	33	15	1	14	23	863	0	1	1
7	6	35	10	0	28	2	1	891	13	69
8	9	55	13	74	11	44	9	7	708	39
9	9	27	3	11	77	6	3	29	22	848

The image shows the confusion matrix of the naïve bayes classifier model applied to our dataset. This confusion matrix is worse than all the others previously seen.

Straightaway we can see that there are some large values in off-diagonal cells. The most prominent values are 125 times an image was predicted as class 9 instead of 4 and 114 times an image was predicted as 3 instead of 5.

There are very few cells with 0 values compared to the confusion matrix of previous models and the misclassification values are very high. There is only one class that was predicted correctly more than the others, and that is label 1.

Considerations:

This algorithm works quickly and can save a lot of time and Naive Bayes is suitable for solving multi-class prediction problems. If its assumption of the independence of features holds true, it can perform better than other models and requires much less training data.

However Naive Bayes assumes that all predictors (or features) are independent, rarely happening in real life, as in this case some features may be correlated. If we use the correlation matrix to select the columns which are having absolute correlation greater than 0,95, we have the following list of features:

['pixel16', 'pixel87', 'pixel225', 'pixel281', 'pixel421', 'pixel617',
'pixel644', 'pixel646', 'pixel727']

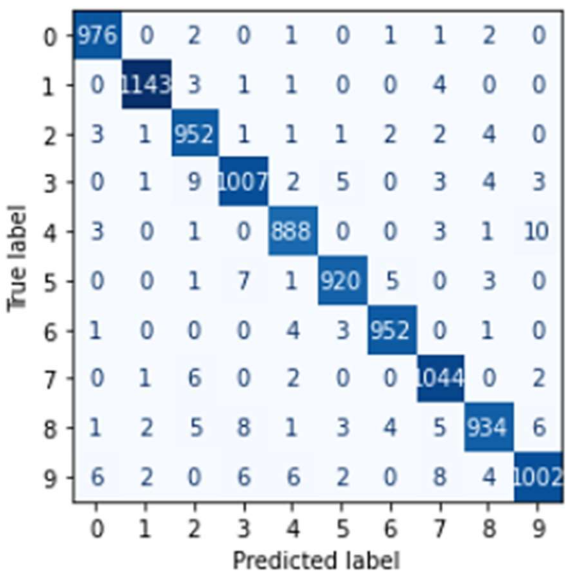
which are highly correlated that means which are somewhat linearly dependent with other features. These features contribute very less in predicting the output but increases the computational cost.

This algorithm faces the '*zero-frequency problem'* where the frequency-based probability estimate will be zero and this will get a zero when all the probabilities are multiplied. We used a *smoothing technique* to overcome this issue, as we did adding an epsilon when calculate k, but this only solve the variance equal to 0. However to solve the mean equal to 0 we just ignore the probability because we know that in that area all pixels are equal to 0 so this mean that are all black pixels. We are interested in pixels other than 0 in order to predict the number represented in the image.

CONCLUSION

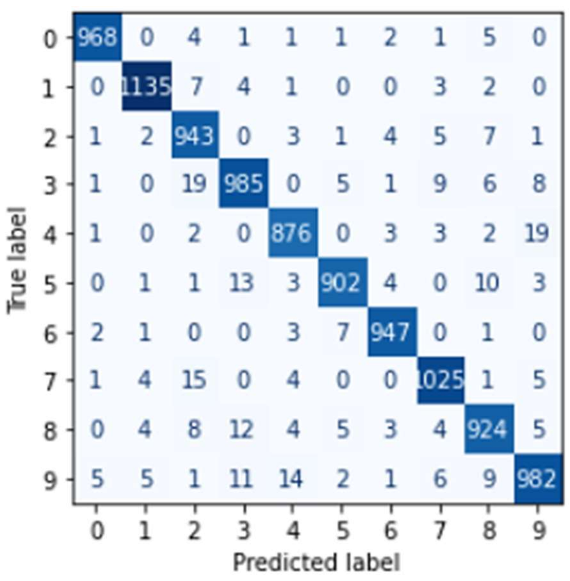
CONFUSION MATRIX:

SVM



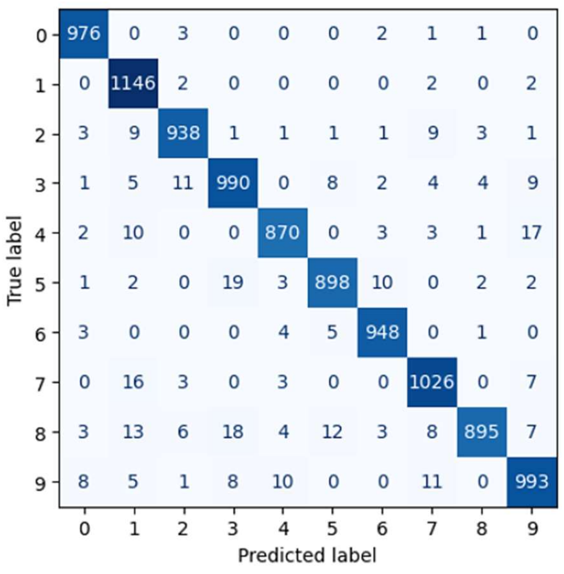
True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	976	0	2	0	1	0	1	1	2	0
1	0	1143	3	1	1	0	0	4	0	0
2	3	1	952	1	1	1	2	2	4	0
3	0	1	9	1007	2	5	0	3	4	3
4	3	0	1	0	888	0	0	3	1	10
5	0	0	1	7	1	920	5	0	3	0
6	1	0	0	0	4	3	952	0	1	0
7	0	1	6	0	2	0	0	1044	0	2
8	1	2	5	8	1	3	4	5	934	6
9	6	2	0	6	6	2	0	8	4	1002

RANDOM FOREST



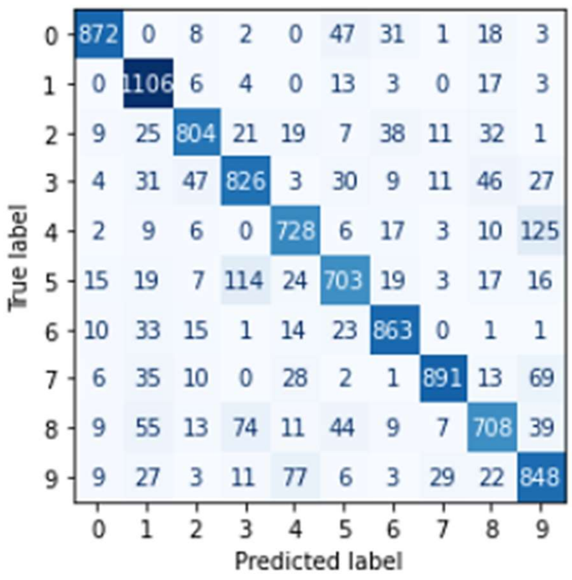
True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	968	0	4	1	1	1	2	1	5	0
1	0	1135	7	4	1	0	0	3	2	0
2	1	2	943	0	3	1	4	5	7	1
3	1	0	19	985	0	5	1	9	6	8
4	1	0	2	0	876	0	3	3	2	19
5	0	1	1	13	3	902	4	0	10	3
6	2	1	0	0	3	7	947	0	1	0
7	1	4	15	0	4	0	0	1025	1	5
8	0	4	8	12	4	5	3	4	924	5
9	5	5	1	11	14	2	1	6	9	982

KNN



True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	976	0	3	0	0	0	2	1	1	0
1	0	1146	2	0	0	0	0	2	0	2
2	3	9	938	1	1	1	1	9	3	1
3	1	5	11	990	0	8	2	4	4	9
4	2	10	0	0	870	0	3	3	1	17
5	1	2	0	19	3	898	10	0	2	2
6	3	0	0	0	4	5	948	0	1	0
7	0	16	3	0	3	0	0	1026	0	7
8	3	13	6	18	4	12	3	8	895	7
9	8	5	1	8	10	0	0	11	0	993

NAIVE BAYES



True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	872	0	8	2	0	47	31	1	18	3
1	0	1106	6	4	0	13	3	0	17	3
2	9	25	804	21	19	7	38	11	32	1
3	4	31	47	826	3	30	9	11	46	27
4	2	9	6	0	728	6	17	3	10	125
5	15	19	7	114	24	703	19	3	17	16
6	10	33	15	1	14	23	863	0	1	1
7	6	35	10	0	28	2	1	891	13	69
8	9	55	13	74	11	44	9	7	708	39
9	9	27	3	11	77	6	3	29	22	848

Putting all the correlation matrixes together we can see that the SVM model with "RBF" kernel is the best. We had already seen it with regards to accuracy, but we can see it here too. The worst model is the naive bayes classifier.

The class that was most correctly predicted in all models is label 1, in fact the corresponding diagonal value is the highest. The worst predicted class for SVM with "RBF" kernel, KNN and random forest was class 4. The only one that differs is the naive bayes classifier, where the worst were class 5 and 8.

CROSS VALIDATION:

	CROSS VALIDATION TIME	*ACCURACY*
SVM	26,34 minutes	96,7%
RANDOM FOREST	19,72 minutes	95,2%
KNN	46,80 minutes	94,6%
NAIVE BAYES	//	//

FIT-PREDICT:

	FIT TIME	*PREDICT TIME on Train*	*ACCURACY TRAIN SET*	*PREDICT TIME on Test*	*ACCURACY TEST SET*
SVM	3,24 minutes	8,32 minutes	100%	1,30 minutes	98,2 %
RANDOM FOREST	3,71 minutes	11,38 seconds	100%	1,29 seconds	96,8%
KNN	0 seconds	More than 3 hours	100%	38,44 minutes	96,8%
NAIVE BAYES	0 seconds	8,87 minutes	83,4%	1,60 minutes	83,5%

In terms of cross validation and testing accuracy, the best model is SVM with "RBF" kernel. The main reason to use an SVM with a non-linear kernel (RBF) is because the problem might not be linearly separable. Another related reason to use SVMs is if you are in a highly dimensional space. For example, SVMs have been reported to work better for text classification. But it requires a lot of time for training and to do the cross validation.

So, it is not recommended when we have a large number of training examples. In terms of computation time, however, it does not beat the KNN, which is clearly the worst.

KNN is robust to noisy training data and is effective in case of large number of training examples. The computation time is also very much as we need to compute distance of each query instance to all training samples. Its accuracy percentage is not as good as that of the SVM model, but it still predicts our dataset well.

Random Forest is nothing more than a bunch of Decision Trees combined. They can handle categorical features very well. This algorithm can handle high dimensional spaces as well as large number of training examples. Its accuracy percentage on the testing set is identical to that of the KNN, the confusion matrix is also very similar. However in terms of computation time it is clearly better than the SVM and KNN model.

If we analyze the last and only generative model, bayes naïve classifier based on Beta Distribution, we can notice that it doesn't need to do cross validation to tune hyperparameters and that the accuracy percentage is the lowest compared to all the other models, much worse. Another thing that can be noticed is that the model on the train set does not have 100% accuracy, which is present on the other models. The accuracy on the train set and test set remains practically the same. This is probably due to outliers present in the dataset that descriptive models handle quite well and generative models are more sensitive. This type of model assumes independence between the features, which we have seen not present, therefore it can lead to incorrect predictions. Moreover, the zero frequency problem does not improve the result.

In conclusion, in terms of accuracy alone, the best model is SVM with RBF kernel. If instead we are looking for a tradeoff between time and accuracy, the best model is random forest.