Assignment 2: Discriminative and Generative Classifiers

Simone Jovon 882028

December 27, 2022

1 Task Description

The goal of the assignment is to write a handwritten digit classifier for the **MNIST database**[1]. These are composed of 70000 28x28 pixel gray-scale images of handwritten digits divided into 60000 training set and 10000 test set.

We have to train the following classifiers on the dataset:

1. SVM using **linear**, **polynomial of degree 2**, and **RBF** kernels;
2. Random forests;
3. Naive Bayes classifier where each pixel is distributed according to a Beta distribution of parameters α, β:

$$d(x, \alpha, \beta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{(\alpha-1)}(1 - x)^{(\beta-1)}$$

4. k-NN.

For SVM and random forests we can use any library we want, but we must implement the Naive Bayes and k-NN classifiers.

[1] *http://yann.lecun.com/exdb/mnist/*

2 Theoretical Background

Let's start by defining what is **supervised learning**: given a set of example input and output pairs find a function that that does a good job of predicting the output associated to a new input. Having said that, all classifiers that we will see in this report are supervised learning models.

A classifier needs to be trained on a dataset to be able to predict, in the best possible way, the output of new data (classification). On large datasets the data is split in two parts:

- **Tainig set**, to train the classifier;
- **Test set**, to test how accurate are the predictions of the classifier over new data.

During the training process the classifier don't have to fit the training date too precisely, because this can lead to bad results on new data, this problem is konwn as **overfitting**.

2.1 Support vector machine (SVM)

If a dataset can be split by a linear separator the data space is called **linear separable**, when we faced a non-linearly-separable dataset we can take two approaches:

1. use a more complex separator;
2. use a linear saparator and accept some errors.

Support vector machine performs a linear classification, but with a trick called **kernel trick** it can also performs non-linear classification.

In general, there can be multiple separators for a dataset, a natural choise, and the one adopted by SVM, is to choose the one with the largest geometric margin.

In the case of a "nearly" separable dataset we may be willing to have a separator that allows a small misclassification by making the margin condition less strict. So it can be introduced a variable C that controll how much strictly the margin must be enforced, choosing a big value of C the classifier will work very hard to correctly classify all the data set, a lower value of C instead allows more easily a misclassification of the point to achive a better margin.

On non-linearly-separable dataset if we transform the feature values of the points inside the dataset in a non-linear way, we can trandform the dataset into one that is linearly separable. To adapt the dataset to the new feature space a function $\phi(x)$ is used, the ϕ function can map the values of the dataset also to an higher-dimentional space and it can do a non-linear transformation.

Since SVM only use a dot product of the data

- $\phi(x_i) \cdot \phi(x_j)$, to train the classifier
- $\phi(x_i) \cdot \phi(u)$, to classify a new point

there is no need to compute $\phi(x)$ to every single point x, but having

$$\phi(x_i) \cdot \phi(x_j) = K(x_i, x_j)$$

we can simply define the function $K : X \times X \to \mathbb{R}, \{x_1, \dots x_n\} \subseteq X$, such function is called **kernel**.

There are various choises for kernels:

- **Linear kernel**: $K(x_i, x_j) = x_i \cdot x_j$
- **Polynomial kernel**: $K(x_i, x_j) = (1 + x_i \cdot x_j)^n$
- **Radial Basis Function**: $exp(-\frac{1}{2}\frac{\|x_i - x_j\|}{\sigma^2})$

2.2 Random forests

Decision trees can be seen as rules to perform categorization, but how to create a good decision tree that makes corrects categorization of new data? The main problem is to decide which node put in which position. This problem can be easily solved using an alghorithm like the ID3 Alghorithm, that using a measure called **Information Gain** it decides which node to put next on the tree.

Decision trees after the training can perfectly fit the given dataset but may not be so accurate to classify new data, so there may be overfitting. The Random forests solves this problem by constructing a multitude of decision tree during the training.

Before going further let's introduce a training technique called Bootstrap Aggregating (**Bagging**); given a training set it selects some random samples with replacement of the training set and it creates the associated decision trees. Having this set of decision trees, the predictions of the classifier is the average of the predictions of the various decision trees.

The training alghorithm for random forests follows the Bagging procedure but with a small difference; each time a split is to be performed, only a random subset of features is selected to create the associated decision tree.

2.3 Naive Bayes classifier

The Naive Bayes classifier have a **generative** approach instead of a **discriminative** one, so this classifier given a training set tries to create models that represents the various classes.

To classify a new data we match the date with all the models, then see which model represents better the new data.

In more formal therms, **discriminative** algorithms try to learn $p(y|x)$, instead **generative** algorithms try to model $p(x|y)$ and $p(y)$.

With a Naive Bayes classifier, to classify a new data $x = \{x_1, \dots x_n\}$ we have to compute for every class c

$$p(x, y = c) = p(x \mid y = c)p(y = c) = \left(\prod_{i=1}^{n} p\left(x_i \mid y = c\right)\right) p(y = c)$$

and choose the model that have the higher joint probability.

On our classifier each pixel is distributed according to a Beta distribution of parameters α, β

$$p(x; \alpha, \beta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1 - x)^{\beta-1}$$

with

$$\begin{aligned} \alpha &= KE[X] \\ \beta &= K(1 - E[X]) \\ K &= \frac{E[X](1 - E[X])}{\operatorname{Var}(X)} - 1 \end{aligned}$$

2.4 k-nearest neighbors (k-NN)

Let's start by defining P as the probability that a vector fall into a region $\mathcal{R}$

$$P = \int_{\mathcal{R}} p(x)dx$$

if we assume that $\mathcal{R}$ is small enough that $p(x)$ doesn't vary so much inside it, we can redefine P as

$$P = \int_{\mathcal{R}} p(x)dx \approx p(x) \int_{\mathcal{R}} dx = p(x)|\mathcal{R}|$$

After a Monte Carlo estimation of the probability $P = \frac{k}{n}$ as the proportion of vectors (k) of the sample (n) fall within $\mathcal{R}$ we have that

$$p(x) = \frac{k/n}{|\mathcal{R}|}$$

Now assume k to be constant and enlarge the window until we find enough samples, if we let the number of neighbors $k(n)$ grows to infinity more slowly than n

$$\lim_{n\to\infty} \frac{k(n)}{n} = 0$$

we have that for each point the nearest neighbor estimator converges in probability to the real density.

Having said that, the k-nearest neighbors classifier, to classify a new point, looks for k nearest neighbors on the training set and classify the new point according to the majority of the neighbors.

3 Implementation

All the code is written using python and Jupyter Notebook is also used to simply the test of the code and the results visualisation.

The following code is used to fetch the data from the dataset and randomly split it in a 60000 training set and 10000 test set:

Listing 1: Dataset fetch

```
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
X,y = fetch_openml('mnist_784', version=1, return_X_y=True)
y = y.astype(int)
X = X/255

X_train, X_test, y_train, y_test = train_test_split(X, y,
            test_size=10000, random_state=42, shuffle=True)
```

Also, the following code is used to execute a 10 way cross validation to optimize the parameters for each classifier:

Listing 2: Cross Validation

```
from sklearn.model_selection import GridSearchCV

X_train_cv = X_train[:10000]
y_train_cv = y_train[:10000]


model = Classifier()  # The instance of the classifier
parameters = {}       # The map of parametor to be tuned


tuned_model = GridSearchCV(model, parameters, cv=10, verbose=0)
tuned_model.fit(X_train_cv, y_train_cv.values.ravel())

print ("Best Score: {:.3f}".format(tuned_model.best_score_) )
print ("Best Params: ", tuned_model.best_params_)
```

For computational and time limits the dataset for cross validation has been reduced to 10000 elements.

All the source code is attached to this report.

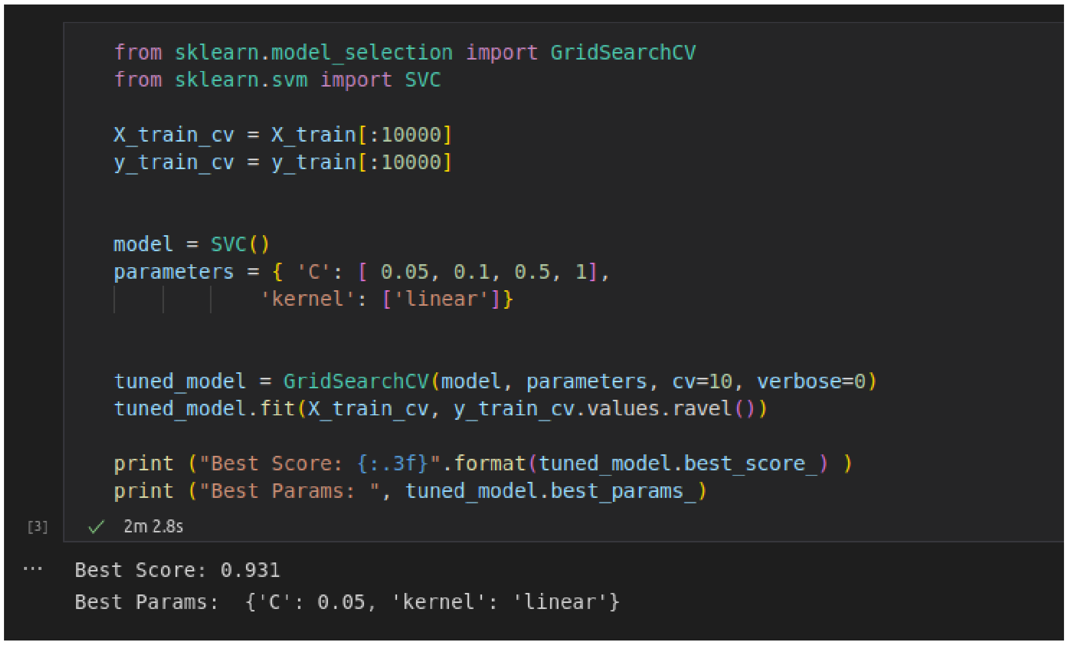
3.1 Support vector machine (SVM)

For SVM I used the **sklearn** library that implements the SVM classifier.

The cross validation is used to perform the tuning of the parameter C of the SVM classifiers.

3.1.1 Linear kernel

The cross validation for the linear kernel classifier reveals (Figure 1) that the best value of the parameter C is 0.05.



```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC

X_train_cv = X_train[:10000]
y_train_cv = y_train[:10000]


model = SVC()
parameters = { 'C': [ 0.05, 0.1, 0.5, 1],
            'kernel': ['linear']}


tuned_model = GridSearchCV(model, parameters, cv=10, verbose=0)
tuned_model.fit(X_train_cv, y_train_cv.values.ravel())

print ("Best Score: {:.3f}".format(tuned_model.best_score_) )
print ("Best Params: ", tuned_model.best_params_)
[3]  ✓ 2m 2.8s
...  Best Score: 0.931
     Best Params:  {'C': 0.05, 'kernel': 'linear'}
```

Figure 1: Linear kernel cross validation

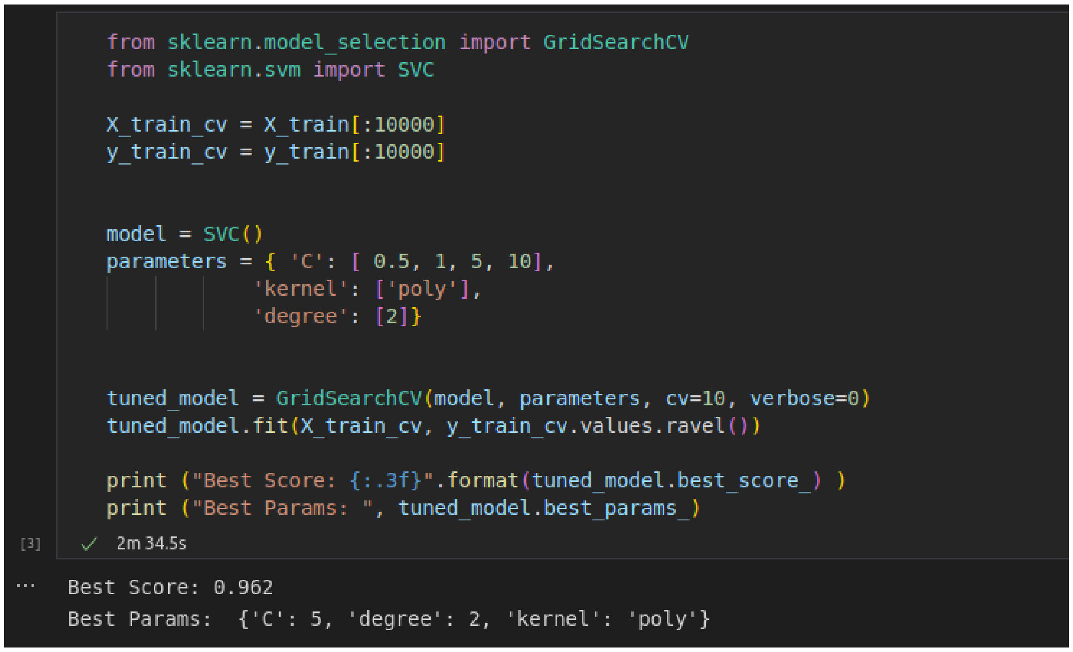
After that we can proceed with the training and test of the classifier (Listing 3).

Listing 3: SVM linear kernel

```
from sklearn import svm

#Create a svm Classifier with linear kernel
clf = svm.SVC(C=0.05,kernel='linear') #

#Train the model using the training sets
clf.fit(X_train, y_train)

#Predict the response for test dataset
y_pred = clf.predict(X_test)
```

3.1.2 Polynomial kernel

The cross validation for the polynomial kernel classifier reveals (Figure 2) that the best value of the parameter C is 5.



```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC

X_train_cv = X_train[:10000]
y_train_cv = y_train[:10000]


model = SVC()
parameters = { 'C': [ 0.5, 1, 5, 10],
            'kernel': ['poly'],
            'degree': [2]}


tuned_model = GridSearchCV(model, parameters, cv=10, verbose=0)
tuned_model.fit(X_train_cv, y_train_cv.values.ravel())

print ("Best Score: {:.3f}".format(tuned_model.best_score_) )
print ("Best Params: ", tuned_model.best_params_)
```

[3] ✓ 2m 34.5s

```
Best Score: 0.962
Best Params:  {'C': 5, 'degree': 2, 'kernel': 'poly'}
```

Figure 2: Polynomial kernel cross validation

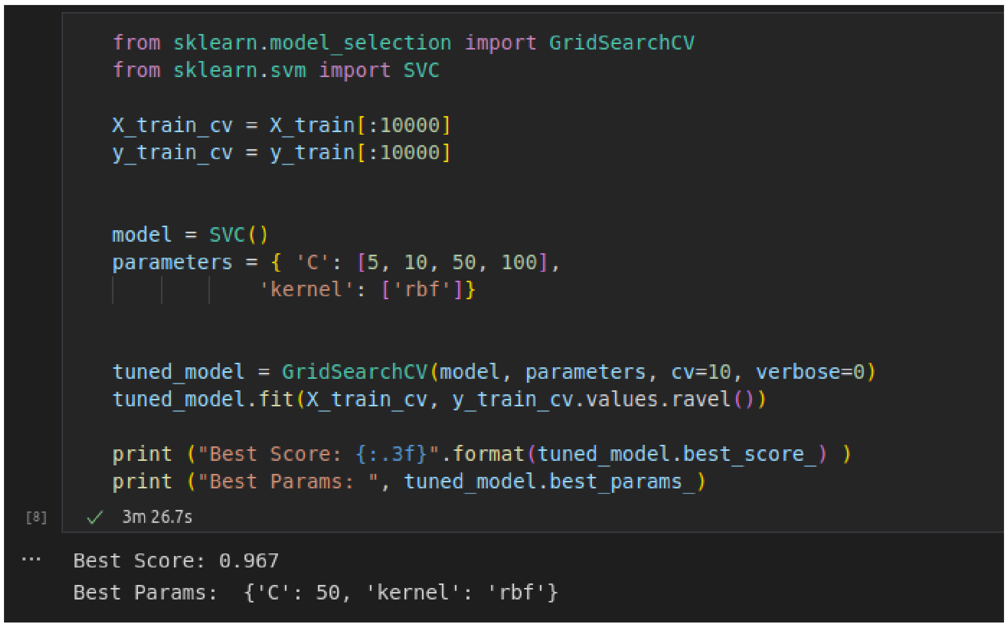
After that we can proceed with the training and test of the classifier (Listing 4).

Listing 4: SVM polynomial kernel

```
from sklearn import svm

#Create a svm Classifier with polynomial kernel
clf = svm.SVC(C=5,kernel='poly', degree=2)  # degree 2

#Train the model using the training sets
clf.fit(X_train, y_train)

#Predict the response for test dataset
y_pred = clf.predict(X_test)
```

3.1.3 Radial Basis Function

The cross validation for the RBF kernel classifier reveals (Figure 3) that the best value of the parameter C is 50.



```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC

X_train_cv = X_train[:10000]
y_train_cv = y_train[:10000]


model = SVC()
parameters = { 'C': [5, 10, 50, 100],
            'kernel': ['rbf']}


tuned_model = GridSearchCV(model, parameters, cv=10, verbose=0)
tuned_model.fit(X_train_cv, y_train_cv.values.ravel())

print ("Best Score: {:.3f}".format(tuned_model.best_score_) )
print ("Best Params: ", tuned_model.best_params_)
```

```
[8]  ✓ 3m 26.7s
```

```
Best Score: 0.967
Best Params:  {'C': 50, 'kernel': 'rbf'}
```

Figure 3: RBF kernel cross validation

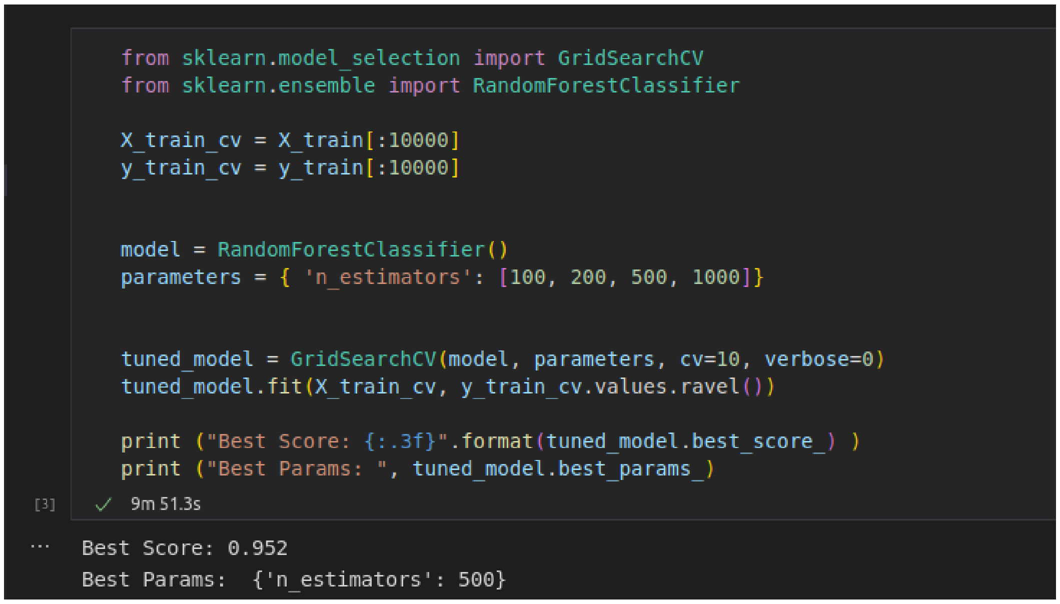
After that we can proceed with the training and test of the classifier (Listing 4).

Listing 5: SVM RBF kernel

```
from sklearn import svm

#Create a svm Classifier RBF kernel
clf = svm.SVC(C=50,kernel='rbf')

#Train the model using the training sets
clf.fit(X_train, y_train)

#Predict the response for test dataset
y_pred = clf.predict(X_test)
```

3.2 Random forests

For Random forests I used the **sklearn** library that implements the random forests classifier. The cross validation is used to perform the tuning of the number of trees in the forest and it reveals (Figure 4) that the optimum number of trees in the forest is 500.



```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier

X_train_cv = X_train[:10000]
y_train_cv = y_train[:10000]


model = RandomForestClassifier()
parameters = { 'n_estimators': [100, 200, 500, 1000]}


tuned_model = GridSearchCV(model, parameters, cv=10, verbose=0)
tuned_model.fit(X_train_cv, y_train_cv.values.ravel())

print ("Best Score: {:.3f}".format(tuned_model.best_score_) )
print ("Best Params: ", tuned_model.best_params_)
```

[3] ✓ 9m 51.3s

```
Best Score: 0.952
Best Params:  {'n_estimators': 500}
```

Figure 4: RBF kernel cross validation

So we can proceed with the training and test of the classifier (Listing 6).

Listing 6: Random forests

```
from sklearn import svm

#Create a random forests Classifier
clf=RandomForestClassifier(n_estimators=500)

#Train the model using the training sets
clf.fit(X_train, y_train)

#Predict the response for test dataset
y_pred = clf.predict(X_test)
```

3.3 Naive Bayes classifier

For the Naive Bayes classifier we have to implement it by ourself.

The `fit(self, X_train, y_train)` method(Listing 7) creates the models for every class of the training set ($[0 \dots 9]$) using the beta distribution; it's calculate the means and the variances of the features of the data on the training set. After that it calculates all the parameters needed to define the beta distribution, that models the class, and to calculates the joint pribabilities.

Listing 7: Naive Bayes fit method

```
def fit(self, X_train, y_train):
    self.X_train = X_train
    self.y_train = y_train
    self.alphas = [None] * 10
    self.betas = [None] * 10
    self.p_class = [None] * 10

    for c in range(10):
        x_class_n = self.X_train[self.y_train == c]

        means = np.mean(x_class_n, axis=0)
        variances = np.var(x_class_n, axis=0)

        ks = ((means * (1 - means)) / variances) - 1
        alphas = ks * means
        betas = ks * (1 - means)

        n_y_class_c = len(self.y_train[self.y_train == c])

        p_class = n_y_class_c / len(self.y_train)

        self.alphas[c] = np.array(alphas)
        self.betas[c] = np.array(betas)
        self.p_class[c] = np.array(p_class)
```

The `predict(self, X)` method (Listing 8) retrives the parameters computed by the `fit` method for every single class and then in order to calculate the joint distributions it calculates the $p(x_i|y = c)$ probabilities using the beta distribution.

Since the beta distribution is a continuous distribution is not possible to calculate $p(x_i|y = c)$, so I decided to calculate the probability on a small neighbourhood of x_i, $p(x_i - 0.05 < x_i < x_i + 0.05|y = c)$ (I've choosen the value 0.05 after some tests). The computation of the beta probabilities in some cases can produce NaN as result, in this cases I've simply replaced the NaN values with 1 so that in the computation of the joint distribution they are ignored.

After having calculated the joint probabilities for every single class, the `predict` method selects the class with the higher joint probability.

Listing 8: Naive Bayes fit method

```
def predict(self, X):

    predict = np.array([])

    for index, row in X.iterrows():

        p = 0
        _class = None
        row = np.array(row)

        for c in range(10):

            alphas = self.alphas[c]
            betas = self.betas[c]
            p_class = self.p_class[c]

            betas_plus = beta.cdf(row+0.05, alphas, betas)
            betas_minus = beta.cdf(row-0.05, alphas, betas)
            beta_probs = betas_plus - betas_minus

            np.nan_to_num(beta_probs, copy=False, nan=1.0)
            p_temp = np.product(beta_probs) * p_class


            if p_temp >= p:
                p = p_temp
                _class = c

        predict = np.append(predict, _class)

    return predict
```

On this Naive Bayes classifier there are no parameters to be tuned so we can simply proceed with the training and test of the classifier (Label 9).

Listing 9: Naive Bayes classifier

```
    #Create a Naive Bayes Classifier
    clf = beta_NaiveBayes()

    #Train the model using the training sets
    clf.fit(X_train, y_train)

    #Predict the response for test dataset
    y_pred = clf.predict(X_test)
```

3.4 k-nearest neighbors (k-NN)

For the k-NN we have to implement it by ourself.

First of all the `fit(self, X_train, y_train)` (Listing 10) is implemented simply by saving the training set as numpy arrays.

Listing 10: k-NN fit method

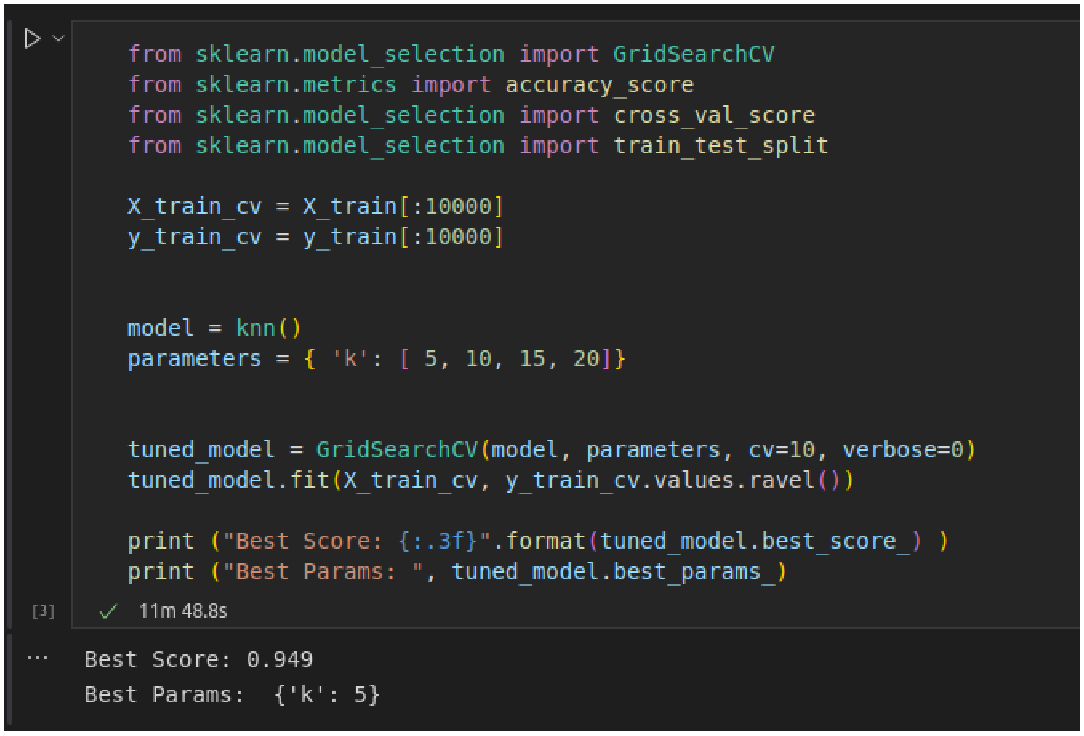
```
def fit(self, X_train, y_train):
    self.X_train = np.array(X_train)
    self.y_train = np.array(y_train)
```

Then let's define the method `predict(self, X)` (Listing 11) to classify new data; it simply iterates through every single point in input and with the help of the `__supp` private method it finds the distance to every points on the training set and selects the classes of the k nearest points. After that with the `statistics.mode` function it selects the most common class on the k selected.

Listing 11: k-NN predict method

```
def __supp(self, x):
    distances = np.linalg.norm(x - self.X_train, axis=1)
    neighbors_y = self.y_train[np.argsort(distances)[:self.k]]

    return statistics.mode(neighbors_y)


def predict(self, X):
    X = np.array(X)
    predict = np.array([])

    for x in X:
        predict = np.append(predict, self.__supp(x))

    return predict
```

The cross validation is used to perform the tuning of the number of neighbors to take in condideration and it reveals (Figure 5) that the optimum number of neighbors is 5.



```
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split

X_train_cv = X_train[:10000]
y_train_cv = y_train[:10000]


model = knn()
parameters = { 'k': [ 5, 10, 15, 20]}


tuned_model = GridSearchCV(model, parameters, cv=10, verbose=0)
tuned_model.fit(X_train_cv, y_train_cv.values.ravel())

print ("Best Score: {:.3f}".format(tuned_model.best_score_) )
print ("Best Params: ", tuned_model.best_params_)
```

```
[3]  11m 48.8s
```

```
Best Score: 0.949
Best Params:  {'k': 5}
```

Figure 5: k-NN cross validation

After having implemented the k-NN classifier we can proceed with the training and test of the classifier (Listing 12).

Listing 12: k-NN

```
#Create a k-NN Classifier
clf = knn(k=5)

#Train the model using the training sets
clf.fit(X_train, y_train)

#Predict the response for test dataset
y_pred = clf.predict(X_test)
```

4 Considerations

In the table 1 is shown a comparison between all the classifier on this report.

As we can see all the SVM classifiers have similar computational (time) requirements on cross validation, training and test time. The polynomial and RBF kernel classifiers seem to have an higher accuracy on the prediction on the training set with ,as already said, similar computational requirements than the linear kernel classifier.

The random forests classifiers seems to have slightly less accuracy than the SVM classifiers and a similar training time, the prediction time instead is way smaller than the SVM ones. I've also made a test reducing to 200 the number of trees in the forest to see how the performance changes; as expected, because of the cross validation, the accuracy is smaller but by a very small amount, 0.9681 for the 200 trees forests againt 0.9686 for the 500 trees forest. However on the performance side the training time decreases significally to 57 seconds, against the 2 minutes and 22 seconds of the 500 trees forest; so for this reason it may be convenient to use the classifier with the 200 trees forest.

Classifier	Accuracy	CV time	Training time	Test time
SVM linear	0.9429	2m 2s	2m 8s	41s
SVM poly	0.9802	2m 34s	1m 52s	31s
SVM RBF	0.9820	3m 26s	2m 22s	54s
Random forests	0.9686	9m 51s	2m 20s	1.3s
Naive Bayes classifier	0.8399	-	0.4s	43s
k-NN	0.9719	11m 49s	0.8s	27m 12s

Table 1: Classifiers comparison