

Foundation of Artificial Intelligence

Assignement 2

Davide Vitagliano

Academic Year 2023/2024

Sommario

[1. Requirements 1](#_Toc153731285)

[2. Introduction 1](#_Toc153731286)

[3. Designing the solution 2](#_Toc153731287)

[3.1. Support Vector Machine 2](#_Toc153731288)

[3.1.1. Formalization 2](#_Toc153731289)

[3.1.2. Code explanation 3](#_Toc153731290)

[3.1.3. Conclusion 5](#_Toc153731291)

[3.2. Random Forest 7](#_Toc153731292)

[3.2.1. Formalization 7](#_Toc153731293)

[3.2.2. Code explanation 8](#_Toc153731294)

[3.2.3. Conclusion 9](#_Toc153731295)

[3.3. K Nearest Neighbor 11](#_Toc153731296)

[3.3.1. Formalization 11](#_Toc153731297)

[3.3.2. Code Explanation 11](#_Toc153731298)

[3.3.3. Conclusion 12](#_Toc153731299)

[3.4. Naive Bayes 13](#_Toc153731300)

[3.4.1. Formalization 13](#_Toc153731301)

[3.4.2. Code Explanation 13](#_Toc153731302)

[3.4.3. Conclusion 14](#_Toc153731303)

[4. Comparison 15](#_Toc153731304)

# Requirements

Write a handwritten digit classifier for the MNIST database. These are composed of 70000 28x28 pixel gray-scale images of handwritten digits divided into 60000 training set and 10000 test set.

Train the following classifiers on the dataset and use 10 fold cross validation to optimize the hyperparameters:

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

With

* K-NN;

For this assignment we used GPU acceleration to increase the performance of the training process using NVIDIA CUDA Toolkit.

# Introduction

The first thing to do during the training process is applying feature selection for dimensionality reduction. This is a process of selecting a subset of the original features that are most relevant and informative for the classification. The main benefits of feature selection are:

* Reducing computational cost: fewer features means less time and space required for training and testing models;
* Improving model performance: removing irrelevant or redundant features can reduce noise and overfitting, and increase accuracy and generalization.

In our case we use Principal Components Analysis to reduce the dimensionality of the dataset by maximizing the variance of each dimension.

Hyperparameter tuning is the process of finding the optimal set of hyperparameters for a classifier. Hyperparameters cannot be learned from the training data because they aggressively increase the capacity of a model and can push the loss function to an undesired minimum, causing overfitting to the data, as opposed to correctly mapping the richness of the structure in the data.

We perform 10 fold cross validation in the process of hyperparameter tuning for each classifier. Cross validation is a statistical method used to estimate the skill of the models; it consists of 3 phases:

1. Data splitting: the entire training set is divided into 10 equal parts;
2. Model training and validation: the model is trained on 9 of these folds and validated on the remaining one. This phase is repeated 10 times, each with a different fold used for validation;
3. Performance evaluation: the model’s performance is evaluated on each of the 10 folds and the final performance score is the average of the 10 scores.

We use the accuracy score to evaluate the quality of the different classifiers. The accuracy is the fraction of prediction that a model got right. It can also be calculated in terms of positives and negatives predictions:

Where TP = True Positives, TN = True Negatives, FP = False Positives, and FN = False Negatives.

# Designing the solution

## Support Vector Machine

### Formalization

Support vector machines are a type of supervised learning method that can be used for classification, regression and outlier detection.

Some of the main features of SVMs are:

* They use a kernel function to map the input data into a higher dimensional space, where a linear decision boundary can be found;
* They try to maximize the margin between the decision boundary and the nearest training points of each class, which are called support vectors;
* They can handle both linear and non-linear problems by choosing different kernel functions, such as polynomial, radial basis function, or custom kernels;
* They have a regularization parameter C that controls the trade-off between the complexity of the model and the error on the training data.

There are several hyperparameters that we need to tune:

* **C**: This is the regularization parameter, also known as the cost parameter. This tells the SVM optimization how much we want to avoid misclassifying each training example. A smaller value of C creates a wider margin, which may allow more misclassifications. A larger C creates a narrower margin and thus may reduce the number of misclassifications;
* **Kernel**: This specifies the kernel type to be used in the algorithm. In our case it will be 'linear', 'poly' or 'rbf';
* **Degree**: This is the degree of the polynomial kernel function ('poly') and is ignored by all other kernels. It essentially controls the complexity of the model;
* **Gamma**: This defines how far the influence of a single training example reaches, with low values meaning 'far' and high values meaning 'close'. It can be seen as the inverse of the radius of influence of samples selected by the model as support vectors;
* **Coef0**: This is the independent term in the kernel function. It is only significant in 'poly' and 'sigmoid'. If gamma is 0, then coef0 controls the bias effect. Otherwise, the larger gamma is, the higher the bias and the lower the variance;
* **Shrinking**: This is a heuristic method used to speed up the training process. It identifies and removes some of the constraints that are not likely to change the final solution, thereby reducing the size of the problem.
* **Probability**: When this hyperparameter is set to True, the SVM enables the calculation of probability estimates. This is done by fitting an additional logistic regression model on the decision function’s scores. This process involves an internal 5 fold cross validation and can slow down the training time.

### Code explanation

This code defines two classes, Classifier and LinearSvc, for training a machine learning model using the ATOM library and the MNIST dataset.

Follows the detailed explanation of each method of the two classes:

The first part is common for every class of the SVM family:

* *Classifier* class: This is the base class for creating a classifier.
  + Initialization (*\_\_init\_\_* method):
    - It loads the MNIST dataset using the *fetch\_openml* function, which fetches datasets from the OpenML repository. The data is then normalized by dividing each feature by 255.
    - An *ATOMClassifier* object is created with the loaded data. This object acts as a wrapper for the whole machine learning pipeline, helping to rapidly find a good model.
  + Train (*train* method):
    - It applies Principal Component Analysis for feature selection.
* *LinearSVC, PolySVC* and *RbfSVC* class: This class inherits from the *Classifier* class and overrides the *train* method.
  + Train (*train* method):
    - it first calls the *train* method of the *Classifier* class to perform feature selection;
    - Then, it uses the *run* method of the *ATOMClassifier* object to train a SVM model;
    - The *run* method takes several parameters, including the type of model to train (SVM), the metric to optimize (accuracy), the number of trials for hyperparameter tuning (10 fold cross validation) and the hyperparameters for the SVM model (*est\_params*);
    - The *est\_params* parameter specifies the hyperparameters for the SVM model;
    - The *ht\_params* parameter specifies the distributions for the hyperparameters that will be tuned during training;
    - Finally, it evaluates the trained model and stores the results in *self.results*.

Now it follows the differences between between the classes:

* *LinearSVC* class:
  + Train (*train* method):
    - In this case, the *est\_params­* tells that the model is using a linear kernel, shrinking heuristic and no probability estimates;
    - In this case, the *ht\_params* tells that the model is using a float distribution for the C parameter;
* *PolySVC* class:
  + Train (*train* method):
    - In this case, the *est\_params­* tells that the model is using a polynomial kernel of second degree, a gamma set to “scale”, shrinking heuristic and no probability estimates;
    - In this case, the *ht\_params* tells that the model is using a float distribution for the C parameter and the coef0 parameter;
* *RbfSVC* class:
  + Train (*train* method):
    - In this case, the *est\_params­* tells that the model is using a Radial Basis Function kernel, a gamma set to “scale”, shrinking heuristic and no probability estimates;
    - In this case, the *ht\_params* tells that the model is using a float distribution for the C parameter parameter;

Gamma set to “scale” means that

### Conclusion

<< ================== ATOM ================== >>

Algorithm task: multiclass classification.

Parallel processing with 12 cores.

GPU training enabled.

Execution engine: cuml.

Parallelization backend: loky

Dataset stats ==================== >>

Shape: (70000, 785)

Train set size: 60000

Test set size: 10000

-------------------------------------

Memory: 439.60 MB

Scaled: False

Outlier values: 415414 (0.9%)

Fitting FeatureSelector...

Performing feature selection ...

--> Feature pixel13 was removed due to collinearity with another feature.

--> Feature pixel14 was removed due to collinearity with another feature.

--> Feature pixel15 was removed due to collinearity with another feature.

--> Feature pixel646 was removed due to collinearity with another feature.

--> Applying Principal Component Analysis...

--> Scaling features...

--> Keeping 779 components.

--> Explained variance ratio: 1.0

Training ========================= >>

Models: LinearSVM

Metric: accuracy

Running hyperparameter tuning for SupportVectorMachine...

| **Trial** | **C** | **Accuracy** | **Time per Trial** |
| --- | --- | --- | --- |
| 0 | 0.0178 | 0.9437 | 02m:37s |
| 1 | 0.1449 | 0.9343 | 03m:05s |
| 2 | 0.001 | 0.9411 | 03m:28s |
| 3 | 0.0081 | 0.9412 | 02m:39s |
| 4 | 0.0028 | 0.9406 | 02m:58s |
| 5 | 0.0019 | 0.9438 | 03m:28s |
| 6 | 0.0036 | 0.9415 | 02m:56s |
| 7 | 0.0109 | 0.9433 | 02m:43s |
| 8 | 0.0155 | 0.9404 | 02m:46s |
| 9 | 0.0413 | 0.9382 | 02m:45s |

Hyperparameter tuning ---------------------------

Best trial --> 5

Best parameters:

--> C: 0.0019

Best evaluation --> accuracy: 0.9438

Time elapsed: 29m:25s

Fit ---------------------------------------------

Train evaluation --> accuracy: 0.9536

Test evaluation --> accuracy: 0.9423

Time elapsed: 06m:39s

-------------------------------------------------

Total time: 36m:04s

Final results ==================== >>

Total time: 36m:05s

-------------------------------------

SupportVectorMachine --> accuracy: 0.9423

## Random Forest

### Formalization

A Random Forest Classifier is a machine learning algorithm that operates by constructing a multitude of decision trees at training time. It’s a meta estimator that fits several decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control overfitting. For classification tasks, the output of the random forest is the class selected by most trees.

The prediction is made based on the majority vote from the multitude of decision trees in the forest. This approach can handle high dimensional datasets like MNIST effectively, and it also provides a measure of feature importance, which can be insightful for understanding which pixels contribute most to the digit classification.

There are several hyperparameters that we need to tune:

* **Number of Estimators**: This refers to the number of trees in the forest. Each tree is built on a subset of data and features. The higher the number of trees, the better the performance, but it can make the code slower;
* **Max Depth**: This is the maximum depth of the tree. It is defined as the longest path between the root node and the leaf node;
* **Criterion**: This is the function to measure the quality of a split. Supported criteria the Gini impurity and the information gain;
* **Min Samples Split**: This specifies the minimum number of samples required to split an internal node;
* **Min Samples Leaf**: This specifies the minimum number of samples required to be at a leaf node;
* **Max Features**: This is the number of features to consider when looking for the best split;
* **Bootstrap**: This is a resampling technique used to estimate statistics on a population by sampling a dataset with replacement. It can be used to estimate summary statistics such as the mean or standard deviation;
* **Max Samples**: This hyperparameter helps to choose the maximum number of samples from the training dataset to train each individual tree;
* **CCP alpha**: Cost complexity pruning provides another option to control the size of a tree. In Decision Tree Classifier, this pruning technique is parameterized by the cost complexity parameter. Greater values of *ccp\_alpha* increase the number of nodes pruned.

### Code explanation

Also in the case of Random Forest the code defines two classes, *Classifier* and *RandForest*, for training a machine learning model using the ATOM library and the MNIST dataset.

Follows the detailed explanation of each method of the two classes:

* *Classifier* class: This is the base class for creating a classifier.
  + Initialization (*\_\_init\_\_* method):
    - It loads the MNIST dataset using the *fetch\_openml* function, which fetches datasets from the OpenML repository. The data is then normalized by dividing each feature by 255.
    - An *ATOMClassifier* object is created with the loaded data. This object acts as a wrapper for the whole machine learning pipeline, helping to rapidly find a good model.
  + Train (*train* method):
    - It applies Principal Component Analysis for feature selection.
* *RandForest* class: This class inherits from the *Classifier* class and overrides the *train* method.
  + Train (*train* method):
    - it first calls the *train* method of the *Classifier* class to perform feature selection;
    - Then, it uses the *run* method of the *ATOMClassifier* object to train a RF model;
    - The *run* method takes several parameters, including the type of model to train (RF), the metric to optimize (accuracy), the number of trials for hyperparameter tuning (10 fold cross validation) and the hyperparameters for the RF model (*est\_params*);
    - The *est\_params* parameter specifies the hyperparameters for the RF model. In this case, it’s not capping the max depth. It’s using a min samples split of 2, a mmin samples leaf of 1 and “sqrt” as max features. It’s also using bootstrap technique and the number of rows in the dataset as max samples;
    - The *ht\_params* parameter specifies the distributions for the hyperparameters that will be tuned during training. In this case, it’s using an int distribution for the number of estimators, a categorical distribution for the criterion and a float distribution for ccp alpha;
    - Finally, it evaluates the trained model and stores the results in *self.results*.

### Conclusion

<< ================== ATOM ================== >>

Algorithm task: multiclass classification.

Parallel processing with 12 cores.

GPU training enabled.

Execution engine: cuml.

Parallelization backend: loky

Dataset stats ==================== >>

Shape: (70000, 785)

Train set size: 60000

Test set size: 10000

-------------------------------------

Memory: 439.60 MB

Scaled: False

Outlier values: 415414 (0.9%)

Fitting FeatureSelector...

Performing feature selection ...

--> Feature pixel13 was removed due to collinearity with another feature.

--> Feature pixel14 was removed due to collinearity with another feature.

--> Feature pixel15 was removed due to collinearity with another feature.

--> Feature pixel646 was removed due to collinearity with another feature.

--> Applying Principal Component Analysis...

--> Scaling features...

--> Keeping 779 components.

--> Explained variance ratio: 1.0

Training ========================= >>

Models: RF

Metric: accuracy

Running hyperparameter tuning for RandomForest...

| **Trial** | **n\_estimators** | **Criterion** | **ccp\_alpha** | **Accuracy** | **Time per Trial** |
| --- | --- | --- | --- | --- | --- |
| 0 | 420 | gini | 0.01 | 0.7744 | 02m:12s |
| 1 | 150 | entropy | 0.01 | 0.8438 | 01m:38s |
| 2 | 400 | gini | 0.025 | 0.4143 | 02m:08s |
| 3 | 210 | gini | 0.025 | 0.3703 | 01m:32s |
| 4 | 420 | gini | 0.005 | 0.8232 | 02m:10s |
| 5 | 810 | gini | 0.025 | 0.3647 | 03m:28s |
| 6 | 880 | gini | 0.0 | 0.9302 | 03m:28s |
| 7 | 170 | gini | 0.015 | 0.6523 | 01m:21s |
| 8 | 960 | entropy | 0.01 | 0.8521 | 06m:08s |
| 9 | 690 | gini | 0.03 | 0.3334 | 03m:01s |

Hyperparameter tuning ---------------------------

Best trial --> 6

Best parameters:

--> n\_estimators: 880

--> criterion: gini

--> ccp\_alpha: 0.0

Best evaluation --> accuracy: 0.9302

Time elapsed: 27m:07s

Fit ---------------------------------------------

Train evaluation --> accuracy: 1.0

Test evaluation --> accuracy: 0.9295

Time elapsed: 03m:37s

-------------------------------------------------

Total time: 30m:44s

Final results ==================== >>

Total time: 30m:48s

-------------------------------------

RandomForest --> accuracy: 0.9295

## K Nearest Neighbor

### Formalization

The k Nearest Neighbors algorithm is a simple, easy-to-implement supervised machine learning algorithm that can be mainly used to solve classification problems.

Some of the key points of KNN are:

* It’s a non-parametric method, which means it makes no underlying assumptions about the distribution of data;
* For classification problems, a class label is assigned based on a majority vote, the label that is most frequently represented around a given data point is used;
* The KNN algorithm is also part of a family of “lazy learning” models, meaning that it only stores a training dataset versus undergoing a training stage. This also means that all the computation occurs when a classification or prediction is being made;

When using the kNN classifier with the MNIST dataset, the algorithm classifies a new image based on the labels of the k most similar images in the training set. Similarity is typically measured using a distance metric, such as Euclidean distance. A small k value can make the classifier sensitive to noise, while a large k value can make it computationally expensive.

The only hyperparameter we need to tune is k.

### Code Explanation

Follows the detailed explanation of each method of the class *Knn*:

* Initialization (*\_\_init\_\_* method):
  + It initializes the class with the number of neighbors to consider in the KNN algorithm;
  + It also loads the MNIST dataset, normalizes it, performs feature selection using PCA;
  + It splits the data into training and testing sets.
* Training (*fit* method):
  + It performs 10-fold cross-validation on the training set;
  + For each fold, it computes the pairwise distances between the validation set and the training set, finds the k-nearest neighbors for each sample in the validation set;
  + It makes predictions by taking the most common label among the neighbors for each sample;
  + it computes the accuracy of the predictions on the validation set.
* Prediction (*predict* method):
  + It computes the pairwise distances between the test set and the training set, finds the k-nearest neighbors for each sample in the test set;
  + It makes predictions by taking the most common label among the neighbors for each sample;
  + It computes the accuracy of the predictions on the test set.

### Conclusion

Fitting FeatureSelector...

Performing feature selection ...

--> Feature pixel646 was removed due to collinearity with another feature.

--> Feature pixel616 was removed due to collinearity with another feature.

--> Applying Principal Component Analysis...

--> Scaling features...

--> Keeping 781 components.

--> Explained variance ratio: 1.0

Training Fold 0 time: 8.707385671998054 seconds

| **Fold** | **Training Time (seconds)** | **Accuracy** |
| --- | --- | --- |
| 0 | 8.707 | 0.9702 |
| 1 | 8.275 | 0.9692 |
| 2 | 7.843 | 0.9702 |
| 3 | 8.005 | 0.9712 |
| 4 | 8.414 | 0.9702 |
| 5 | 8.247 | 0.9717 |
| 6 | 8.022 | 0.9713 |
| 7 | 8.282 | 0.9735 |
| 8 | 8.583 | 0.9695 |
| 9 | 8.498 | 0.9715 |

Best Training Cross-Validation Accuracy: 0.9735

Test Set time: 15.6139 seconds

Test Set Accuracy: 0.9728

## Naive Bayes

### Formalization

The Naive Bayes classifier is based on applying Bayes’ theorem with the “naive” assumption of conditional independence between every pair of features given the value of the class variable.

The Beta distribution can be used as the prior distribution for the parameters of the Bernoulli distribution when the features are binary. The Beta distribution is defined on the interval [0, 1], making it a suitable choice for modeling the parameters of a Bernoulli distribution, which represent probabilities.

The classifier calculates the parameters (alpha and beta) of the Beta distribution for each class in the training set. These parameters are then used to calculate the likelihood of a given feature value given a class. The class with the highest posterior probability is then chosen as the prediction for a given input.

### Code Explanation

Follows the detailed explanation of each method of the class *Knn*:

* Initialization (*\_\_init\_\_* method):
  + It initializes the class with the number of neighbors to consider in the KNN algorithm;
  + It also loads the MNIST dataset, normalizes it, performs feature selection using PCA;
  + It splits the data into training and testing sets.
* Training (*fit* method):
  + It calculates the mean and variance for each class in the training set;
  + then it uses these to calculate the alpha and beta parameters for the Beta distribution.
* Prediction (*predict / validate* method):
  + This method calculates the posterior probabilities for each class given the input data, then returns the class with the highest posterior probability.
* Cross Validation (*cross\_validate* method):
  + This method performs k fold cross validation on the training data. It splits the training data into k folds;
  + for each fold, it fits the model on the training portion and validates it on the validation portion;
  + It keeps track of the accuracy for each fold and the parameters that yield the best accuracy.

### Conclusion

Fitting FeatureSelector...

Performing feature selection ...

--> Feature pixel646 was removed due to collinearity with another feature.

--> Feature pixel616 was removed due to collinearity with another feature.

--> Applying Principal Component Analysis...

--> Scaling features...

--> Keeping 781 components.

--> Explained variance ratio: 1.0

| **Fold** | **Training Time (seconds)** | **Accuracy** |
| --- | --- | --- |
| 0 | 0.183 | 0.1013 |
| 1 | 0.121 | 0.0938 |
| 2 | 0.122 | 0.0968 |
| 3 | 0.121 | 0.0992 |
| 4 | 0.121 | 0.1032 |
| 5 | 0.120 | 0.0962 |
| 6 | 0.124 | 0.1002 |
| 7 | 0.123 | 0.1012 |
| 8 | 0.123 | 0.0995 |
| 9 | 0.122 | 0.0993 |

Best Cross-Validation Accuracy: 0.1032

Testing time: 0.1477 seconds

Test Accuracy: 0.0959

# Comparison