Institute of Computer Science, Silesian University of Technology

Team of Micro informatics and Theory of Digital Automata

Academic year:	Type of study*: SSI/NSI/NSM	Subject (BIAI):	Group	Section
2024	**NSI**	**BIAI**	**BDiIS**	**2**

Names of section members:	Supervisor:	
Martyna Szczekocka **Rafał Kowalczuk** **Tymoteusz Drzał**	OA/JP/KT/GD/BSz/GB	**GB**

Final Report

Project Topic:

Mnist fashion

Date of handing: DD/MM/YYYY	25/05/2024

1. Short Introduction Presenting the Project Topic

The purpose of this project is to evaluate the performance of various machine learning algorithms and neural network architectures on the Fashion-MNIST dataset. Fashion-MNIST, also known as MNIST Fashion, is a dataset designed for benchmarking machine learning algorithms in image recognition tasks. It was created as a more challenging alternative to the original MNIST dataset, which consists of handwritten digits. Fashion-MNIST contains 70,000 grayscale images of 10 different categories of clothing and accessories, such as t-shirts, pants, coats, dresses, and bags. Each image is 28x28 pixels in size. The dataset is divided into a training set of 60,000 images and a test set of 10,000 images.

In this project, we aim to compare the effectiveness of several classification algorithms, including traditional machine learning methods like Decision Trees and k-Nearest Neighbors, as well as more advanced techniques such as Multi-Layer Perceptron (MLP) and Convolutional Neural Networks (CNN). By analyzing the performance, accuracy, and training times of these models, we can determine which approaches are best suited for the task of fashion item recognition and what trade-offs exist between different methods.

2. Analysis of the Task

a. Possible Approaches to Solve the Problem with Its Pros/Cons, Detailed Description of Selected Methodology

To tackle the task of classifying images in the Fashion-MNIST dataset, we considered several machine learning algorithms and neural network architectures, as well as various tools, frameworks, and libraries for data handling and model implementation. Below is an analysis of each approach, including its advantages and disadvantages, followed by a detailed description of our selected methodology.

Tools/Frameworks/Libraries

1. Pandas

Pandas is a powerful data manipulation library that provides flexible data structures for data analysis and preprocessing.

Pros:

Ease of Data Handling: Simplifies loading, cleaning, and manipulating datasets.

Rich Support for Data Operations: Offers comprehensive functions for filtering, grouping, merging, and reshaping data.

Integration with Other Libraries: Seamlessly integrates with other scientific libraries like NumPy and Matplotlib.

Cons:

Memory Consumption: Can be inefficient with memory usage, especially for large datasets.

Performance with Very Large Datasets: May suffer from performance issues when handling extremely large datasets.

2. Scikit-learn

Scikit-learn is a widely used machine learning library that provides simple and efficient tools for data mining and data analysis.

Pros:

Wide Range of Algorithms: Includes numerous machine learning algorithms for classification, regression, clustering, and more.

Documentation and Community Support: Well-documented with a large, active community for support.

Ease of Integration and Use: Designed for ease of use with a consistent API, integrates well with other Python libraries.

Cons:

Scalability Limitations: Not ideal for very large-scale data or big data applications.

Lack of Support for Deep Learning Models: Primarily focused on traditional machine learning algorithms, with limited support for deep learning.

3. TensorFlow/Keras

TensorFlow is an open-source platform for machine learning, and Keras is its high-level API designed to make building and training deep learning models straightforward.

Pros:

User-Friendly and High-Level API: Intuitive and user-friendly, making it easy to build and prototype deep learning models.

Scalability and Flexibility: Highly scalable, suitable for a range of platforms from mobile devices to distributed cloud computing.

Cons:

Resource-Intensive: Requires significant computational resources, particularly for training large models.

Complexity for Advanced Optimization: Advanced optimizations and fine-tuning can be complex and require deeper understanding.

After evaluating the various approaches, we decided to use the following models for our experiments:

- DecisionTreeClassifier: Chosen for its interpretability and ease of use.
- KNeighborsClassifier: Selected due to its simplicity and effectiveness in certain conditions.
- MLPClassifier: Opted for its versatility in handling complex tasks with a customizable architecture.
- Convolutional Neural Network (CNN): Chosen for its proven effectiveness in image recognition tasks.

1. MLPClassifier (Multi-Layer Perceptron)

MLPClassifier is a feedforward artificial neural network model implemented in Scikit-learn.

Pros:

- Versatility and Customization Options: Capable of handling complex tasks with customizable architecture (number of layers, nodes, etc.).

Cons:

- Computational Complexity: Training can be computationally expensive.
- Risk of Overfitting: Requires careful tuning of hyperparameters to avoid overfitting.

2. DecisionTreeClassifier

DecisionTreeClassifier is a simple and interpretable model that uses a tree structure to make decisions based on feature values.

Pros:

- Interpretability: Easy to understand and visualize, making it suitable for exploratory data analysis.
- No Need for Complex Data Preprocessing: Can handle categorical features and missing values directly.

Cons:

- Tendency to Overfit: Prone to creating overly complex trees that fit the training data too closely.
- Instability: Small changes in data can result in significantly different trees.

3. KNeighborsClassifier

KNeighborsClassifier is a straightforward and intuitive algorithm based on the distance between data points.

Pros:

- Simple and Intuitive: Easy to understand and implement.
- Effective in Specific Conditions: Works well for small datasets and when the relationship between features is complex and nonlinear.

Cons:

- Sensitive to Improperly Scaled Features: Performance can be affected by the scale of input features.
- "Curse of Dimensionality": Becomes less effective as the number of dimensions increases.

4. Convolutional Neural Network (CNN)

Convolutional Neural Networks are deep learning models particularly well-suited for image recognition tasks.

Pros:

- Highly Effective for Image Recognition and Processing: Capable of automatically learning hierarchical feature representations from images.
- State-of-the-Art Performance: Often achieves superior accuracy on image classification tasks.

Cons:

- Require a Large Amount of Labeled Training Data: Needs substantial datasets to perform well.
- Computationally Intensive: Demands significant computational power for training and inference.

b) presentation of possible/available datasets and detailed description of the chosen ones

The Fashion-MNIST dataset consists of 70,000 grayscale images of 10 different categories of clothing and accessories, such as t-shirts, pants, coats, dresses, and bags. Each image is 28x28 pixels in size. The dataset is divided into a training set of 60,000 images and a test set of 10,000 images. Pandas: Enables the use of structures such as DataFrame and Series. Allows data retrieval from various sources, such as databases and CSV files. Supports data cleaning, transformation, and aggregation processes.

c. analysis of available tools/frameworks/libraries suitable for task solution; detailed presentation of selected tools/approach,

For this project, we selected the following tools based on their strengths and suitability for the task:

Pandas: Used for data loading, cleaning, and preprocessing. Pandas was chosen for its flexibility and powerful data manipulation capabilities, which are essential for preparing the Fashion-MNIST dataset for analysis.

Scikit-learn: Utilized for implementing traditional machine learning models such as DecisionTreeClassifier, KNeighborsClassifier, and MLPClassifier. Scikit-learn was selected due to its wide range of algorithms, ease of use, and strong community support.

TensorFlow/Keras: TensorFlow, along with Keras, was used for building and training Convolutional Neural Networks (CNNs). We chose TensorFlow/Keras for their user-friendly API, scalability, and effectiveness in deep learning tasks. These features make them ideal for addressing the image recognition challenges posed by the Fashion-MNIST dataset.

Our approach involved using Pandas for initial data handling and preprocessing, Scikit-learn for implementing and evaluating traditional machine learning models, and TensorFlow/Keras for constructing and training deep learning models. This combination of tools allowed us to leverage the strengths of each library to build effective and efficient solutions for classifying images in the Fashion-MNIST dataset.

3. Internal and External Specification of the Software Solution

a. Classes/Objects/Methods/Scripts/Functions etc. – Depending on the Approach to the Project Solution

The software solution for this project involves several key components, including classes, functions, and methods, which are outlined below.

Classes and Models:

- MLPClassifier: A feedforward artificial neural network model from Scikit-learn.
- KNeighborsClassifier: A k-nearest neighbors algorithm for classification tasks from Scikit-learn.
- DecisionTreeClassifier: A decision tree algorithm for classification tasks from Scikit-learn.
- Sequential (TensorFlow/Keras): A sequential neural network model used to construct the Convolutional Neural Network (CNN).

Functions:

save_results(data, append=False)

Description: Saves the results of model evaluations to a file.

Parameters:

data: The data to be saved.

append: Boolean flag to append data to the file if set to True; otherwise, overwrites the file.

Usage: save_results(scores, append=True)

scale(data)

Description: Scales the input data to a range between 0 and 1.

Parameters:

data: The data to be scaled.

Usage: x_train = scale(train_set.drop('label', axis=1))

fit_predict_score(model, x_train, y_train, x_test, y_test, verbose=False, showtime=False)

Description: Trains the model, makes predictions, and calculates accuracy scores for both training and test datasets.

Parameters:

model: The machine learning model to be trained and evaluated.

x_train: Training data features.

y_train: Training data labels.

x_test: Test data features.

y_test: Test data labels.

verbose: Boolean flag to print detailed information if set to True.

showtime: Boolean flag to print training time if set to True.

Returns: Dictionary containing train_score, test_score, and train_time.

Usage: scores[name] = fit_predict_score(model, x_train, y_train, x_test, y_test, verbose=True, showtime=True)

main()

Description: The main function that orchestrates the data preparation, model configuration, training, evaluation, and results saving.

Usage: if __name__ == '__main__': main()

b. Data Structures etc.

Pandas DataFrame

Used for loading and manipulating the Fashion-MNIST dataset.

Example:

train_set = pd.read_csv('data/fashion-mnist_train.csv')

x_train = scale(train_set.drop('label', axis=1))

y_train = train_set['label']

NumPy Arrays

Used for efficient numerical computations and transformations of data.

Example:

```
x_train = x_train.values.reshape(x_train.shape[0], 28, 28)
```

Dictionary

Used to store and organize model evaluation results.

Example:

```
scores = {}

scores[name] = fit_predict_score(model, x_train, y_train, x_test, y_test, verbose=True, showtime=True)
```

Sequential Model (TensorFlow/Keras)

Used to define and compile the CNN architecture.

Example:

```
cnn = Sequential(layers=[layers.Input((28, 28, 1)),
                         layers.Conv2D(32, (3, 3), activation='relu'),
                         layers.MaxPooling2D((2, 2)),
                         layers.Flatten(),
```

```
        layers.Dense(64, activation='relu'),
        layers.Dense(10, activation='softmax')], name='cnn_1c-3d_e8')
cnn.compile(optimizer='Adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])
```

4. Experiments

a. Experimental Background

Description of Experiments

The primary objective of our experiments was to evaluate and compare the performance of different machine learning algorithms and neural network architectures on the Fashion-MNIST dataset. The experiments were designed to test the following models:

- DecisionTreeClassifier
- KNeighborsClassifier
- MLPClassifier (Multi-Layer Perceptron)
- Convolutional Neural Network (CNN)

Parameters/Conditions Subject to Change

For each model, we varied several parameters to observe their effects on model performance. The key parameters and conditions changed during the experiments were:

MLPClassifier:

hidden_layer_sizes: Number and size of hidden layers.

solver: Optimization algorithm (adam vs. sgd).

alpha: Regularization term to prevent overfitting.

max_iter: Maximum number of iterations for training.

KNeighborsClassifier:

n_neighbors: Number of nearest neighbors considered for classification.

DecisionTreeClassifier:

criterion: Function to measure the quality of a split (gini vs. entropy).

max_depth: Maximum depth of the tree.

CNN:

epochs: Number of training epochs.

Conv2D layers: Number of convolutional layers.

Dense layers: Number of dense (fully connected) layers.

Method of Results Presentation

The results were presented using the following metrics:

- Train Score: Accuracy on the training set.
- Test Score: Accuracy on the test set.
- Train Time: Time taken to train the model.

b. Presentation of Experiments with Detailed Analysis and Description of the Results with Partial Conclusions Possible to Be Drawn Based on the Presented Part of Experiments

MLPClassifier (hidden_layer_size)

Results:


```
"MLPClassifier(alpha=0.01, hidden_layer_sizes=(60, 80, 60))": {
    "train_score": 0.9813333333333333,
    "test_score": 0.8824,
    "train_time": 419.00302624702454
},
```

```
"MLPClassifier(alpha=0.01, hidden_layer_sizes=(80,))": {
    "train_score": 0.9681333333333333,
    "test_score": 0.886,
    "train_time": 316.3913562297821
},
```

```
"MLPClassifier(alpha=0.01, hidden_layer_sizes=(80, 60, 60, 60, 80))": {
    "train_score": 0.9719666666666666,
    "test_score": 0.8909,
    "train_time": 322.88012647628784
},
```


Configuration: Different hidden layer sizes such as (60, 80, 60), (80,), and (80, 60, 60, 80).

Observation: The results were comparable across different configurations. Increasing the number of hidden layers did not significantly affect the training time.

Conclusion: Even a single hidden layer is sufficient to achieve good accuracy while minimizing the risk of overfitting.

MLPClassifier (solver)

Results:


```
"MLPClassifier(alpha=0.01, hidden_layer_sizes=(80, 60, 60, 60, 80))": {
    "train_score": 0.9719666666666666,
    "test_score": 0.8909,
    "train_time": 322.88012647628784
},
```

```
"MLPClassifier(alpha=0.01, hidden_layer_sizes=(80, 60, 60, 60, 80), solver='sgd')": {
    "train_score": 0.959,
    "test_score": 0.884,
    "train_time": 375.32812571525574
},
```


Configuration: Comparison between adam and sgd solvers.

Observation: Using sgd resulted in lower accuracy and longer training times compared to adam.

Conclusion: The adam solver performed better for the given dataset.

MLPClassifier (alpha)

Results:


```
"MLPClassifier(alpha=0.1, hidden_layer_sizes=(80, 60, 60, 60, 80))": {
    "train_score": 0.9560666666666666,
    "test_score": 0.8958,
    "train_time": 354.42098021507263
},
```

```
"MLPClassifier(alpha=0.01, hidden_layer_sizes=(80, 60, 60, 60, 80))": {
    "train_score": 0.9719666666666666,
    "test_score": 0.8909,
    "train_time": 322.88012647628784
},
```

```
"MLPClassifier(alpha=0.001, hidden_layer_sizes=(80, 60, 60, 60, 80))": {
    "train_score": 0.9770333333333333,
    "test_score": 0.886,
    "train_time": 472.73176431655884
},
```


Configuration: Different alpha values such as 0.1, 0.01, and 0.001.

Observation: Higher alpha values helped in reducing overfitting but increased training time.

Conclusion: A higher alpha value (0.1) provided the best trade-off between accuracy and generalization.

MLPClassifier (max_iter)

Results:


```
"MLPClassifier(alpha=0.01, hidden_layer_sizes=(60, 80, 60), max_iter=5)": {
    "train_score": 0.8845,
    "test_score": 0.8725,
    "train_time": 9.217031478881836
},
```

```
 "MLPClassifier(alpha=0.01, hidden_layer_sizes=(60, 80, 60), max_iter=2)": {
     "train_score": 0.8623166666666666,
     "test_score": 0.8591,
     "train_time": 3.846675395965576
 },
```

```
"MLPClassifier(alpha=0.01, hidden_layer_sizes=(60, 80, 60), max_iter=10)": {
    "train_score": 0.8953333333333333,
    "test_score": 0.8777,
    "train_time": 17.873986959457397
},
```


Configuration: Different maximum iterations such as 5, 10, and 2.

Observation: Increasing the number of iterations improved accuracy but also increased training time.

Conclusion: The best results were achieved with max_iter = 10, but the performance gain was marginal beyond 5 iterations.

KNeighborsClassifier (n_neighbors)

Results:


```
"KNeighborsClassifier(n_neighbors=3)": {
    "train_score": 0.9193833333333333,
    "test_score": 0.8584,
    "train_time": 40.24733805656433
},
```

```
"KNeighborsClassifier()": {
    "train_score": 0.8993333333333333,
    "test_score": 0.8592,
    "train_time": 36.7211275100708
},
```

```
"KNeighborsClassifier(n_neighbors=15)": {
    "train_score": 0.8671833333333333,
    "test_score": 0.8516,
    "train_time": 36.773380517959595
},
```

```
"KNeighborsClassifier(n_neighbors=23)": {
    "train_score": 0.8578666666666667,
    "test_score": 0.8438,
    "train_time": 37.430967569351196
},
```


Configuration: Different values for n_neighbors such as 3, 5, 15, and 23.

Observation: The default value (5) provided the best accuracy. Too few neighbors led to overfitting, while too many led to underfitting.

Conclusion: n_neighbors = 5 was optimal for the dataset.

DecisionTreeClassifier (max_depth)

Results:


```
"DecisionTreeClassifier()": {
    "train_score": 1.0,
    "test_score": 0.7958,
    "train_time": 35.07290816307068
}
```

```
"DecisionTreeClassifier(max_depth=80)": {
    "train_score": 1.0,
    "test_score": 0.7979,
    "train_time": 36.20009136199951
}
```

```
"DecisionTreeClassifier(max_depth=35)": {
    "train_score": 0.9988666666666667,
    "test_score": 0.7991,
    "train_time": 32.82619261741638
}
```

```
"DecisionTreeClassifier(max_depth=28)": {
    "train_score": 0.99705,
    "test_score": 0.8001,
    "train_time": 30.24880623817444
}
```


Configuration: Different maximum depths such as None, 80, 35, and 28.

Observation: Limiting the tree depth reduced overfitting and improved test accuracy.

Conclusion: A max_depth of 28 provided a good balance between train and test accuracy.

DecisionTreeClassifier (criterion)

Results:


```
"DecisionTreeClassifier(max_depth=28)": {
    "train_score": 0.99705,
    "test_score": 0.8001,
    "train_time": 30.24880623817444
}
```

```
"DecisionTreeClassifier(criterion='entropy', max_depth=28)": {
    "train_score": 1.0,
    "test_score": 0.8123,
    "train_time": 29.004814386367798
}
```


Configuration: Comparison between gini and entropy criteria.

Observation: The entropy criterion resulted in higher accuracy on test data but also higher train accuracy, indicating potential overfitting.

Conclusion: The gini criterion was more stable for generalization.

CNN (epochs)

Results:


```
"cnn_1c-2d_e5": {
    "train_score": 0.9454666376113892,
    "test_score": 0.9187999963760376,
    "train_time": 35.12720584869385
}
```

```
"cnn_1c-2d_e2": {
    "train_score": 0.9086499810218811,
    "test_score": 0.8985999822616577,
    "train_time": 15.977484941482544
}
```

```
"cnn_1c-2d_e8": {
    "train_score": 0.9657333493232727,
    "test_score": 0.9193999767303467,
    "train_time": 54.96539926528931
}
```


Configuration: Different number of epochs such as 2, 5, and 8.

Observation: Accuracy improved with more epochs but at the cost of increased training time.

Conclusion: 8 epochs provided the best accuracy without overfitting.

CNN (convolutional layers)

Results:


```
  "cnn_1c-2d_e8": {
      "train_score": 0.9657333493232727,
      "test_score": 0.9193999767303467,
      "train_time": 54.96539926528931
  }
```

```
"cnn_2c-2d_e8": {
    "train_score": 0.9410666823387146,
    "test_score": 0.911899983882904,
    "train_time": 48.74108004570007
}
```

```
"cnn_3c-2d_e8": {
    "train_score": 0.8935166597366333,
    "test_score": 0.880299985408783,
    "train_time": 52.725521087646484
}
```


Configuration: Varying the number of convolutional layers.

Observation: One convolutional layer provided the best result. Adding more layers did not significantly improve accuracy and sometimes degraded it.

Conclusion: A single convolutional layer was sufficient for the task.

CNN (dense layers)

Results:


```
    "cnn_1c-3d_e8": {
        "train_score": 0.9625333547592163,
        "test_score": 0.920199990272522,
        "train_time": 56.03071665763855
    }
}{
    "cnn_1c-1d_e8": {
        "train_score": 0.9364333152770996,
        "test_score": 0.9106000065803528,
        "train_time": 30.752292156219482
    }
```

```
 "cnn_1c-2d_e8": {
     "train_score": 0.9657333493232727,
     "test_score": 0.9193999767303467,
     "train_time": 54.96539926528931
 }
```


Configuration: Varying the number of dense layers.

Observation: Three dense layers achieved the best result, but the accuracy improvement was within the margin of error.

Conclusion: Adding dense layers increased training time significantly without substantial accuracy gains.

5. Summary, Overall Conclusions, Possible Improvements, Future Work etc.

Summary

In this project, we evaluated the performance of various machine learning algorithms and neural network architectures on the Fashion-MNIST dataset. Our goal was to determine which methods are best suited for the task of fashion item recognition and to identify the trade-offs between different approaches.

Convolutional Neural Network (CNN):

Performance: Achieved the highest accuracy on the test data (0.9201).

Generalization: Demonstrated good generalization with a small difference between training and test accuracy.

Training Time: Required significant computational resources and longer training times.

KNeighborsClassifier:

Performance: Provided reasonable accuracy with the optimal number of neighbors (n_neighbors = 5).

Generalization: Showed good generalization with minimal overfitting.

Simplicity: Simple and intuitive algorithm, effective for the given dataset.

DecisionTreeClassifier:

Performance: Performed the worst with high overfitting and instability.

Interpretability: Easy to understand and visualize, but prone to creating overly complex trees.

Suitability: Not ideal for this dataset due to its tendency to overfit.

MLPClassifier (Multi-Layer Perceptron):

Performance: Showed good accuracy but required careful tuning of hyperparameters to avoid overfitting.

Training Time: Computationally expensive with longer training times compared to other models.

Versatility: Capable of handling complex tasks with customizable architecture.

Overall Comparison of Models

Best Accuracy on Test Data:

CNN: Achieved the highest test accuracy (0.9201).

KNeighborsClassifier and CNN: Showed the smallest difference between training and test accuracy, indicating good generalization.

DecisionTreeClassifier: Performed the worst with high overfitting.

MLPClassifier: Showed good accuracy but required longer training times

Possible Improvements

Hyperparameter Optimization:

Conduct a more comprehensive hyperparameter search to further improve model performance and generalization.

Data Augmentation:

Implement data augmentation techniques to increase the diversity of the training dataset and improve the robustness of the models.

Ensemble Methods:

Explore ensemble methods that combine multiple models to enhance overall performance and reduce overfitting.

Regularization Techniques:

Apply advanced regularization techniques, such as dropout and L2 regularization, to prevent overfitting, particularly in neural network models

Future Work

Exploring Other Neural Network Architectures:

Investigate other deep learning architectures, such as Recurrent Neural Networks (RNNs) and Generative Adversarial Networks (GANs), to assess their suitability for fashion item recognition.

Testing on Other Datasets:

Validate the models on other image recognition datasets to evaluate their generalizability and robustness across different types of data.

Implementing Transfer Learning:

Utilize pre-trained models through transfer learning to leverage existing knowledge and potentially improve model performance with reduced training times.

Optimizing Computational Resources:

Optimize the use of computational resources, such as using distributed training techniques and hardware accelerators (e.g., GPUs and TPUs), to handle larger models and datasets more efficiently.

Utilizing Ensemble Methods:

Explore the use of ensemble methods, such as Random Forests, to improve model performance and robustness. Random Forests, which use multiple decision trees to make predictions, can help reduce overfitting and increase the accuracy of classification tasks.

By addressing these potential improvements and future work, we aim to enhance the effectiveness and efficiency of machine learning and deep learning models for image recognition tasks in the fashion domain.

6. References – list of sources used during the work on the project.

- https://pandas.pydata.org/docs/
- https://scikit-learn.org/stable/
- https://hyperskill.org/tracks/28
- https://docs.python.org/3.10/
- https://www.ibm.com/docs/pl/spss-statistics/saas?topic=networks-multilayer-perceptron
- https://www.datacamp.com/tutorial/k-nearest-neighbor-classification-scikit-learn

- https://www.datacamp.com/tutorial/decision-tree-classification-python
- https://towardsdatascience.com/grid-search-for-model-tuning-3319b259367e

7. Link to the shared cloud/GoogleDrive/OneDrive/Git/etc. where all the files are placed.

https://github.com/MSzczekocka/FashonMnist

https://drive.google.com/drive/folders/1eqomqqylt5qrzSDb8hzlnNNEbjAA6u5y