MNIST FASHION

Tymoteusz Drzał, Rafał Kowalczuk, Martyna Szczekocka

INTRODUCTION

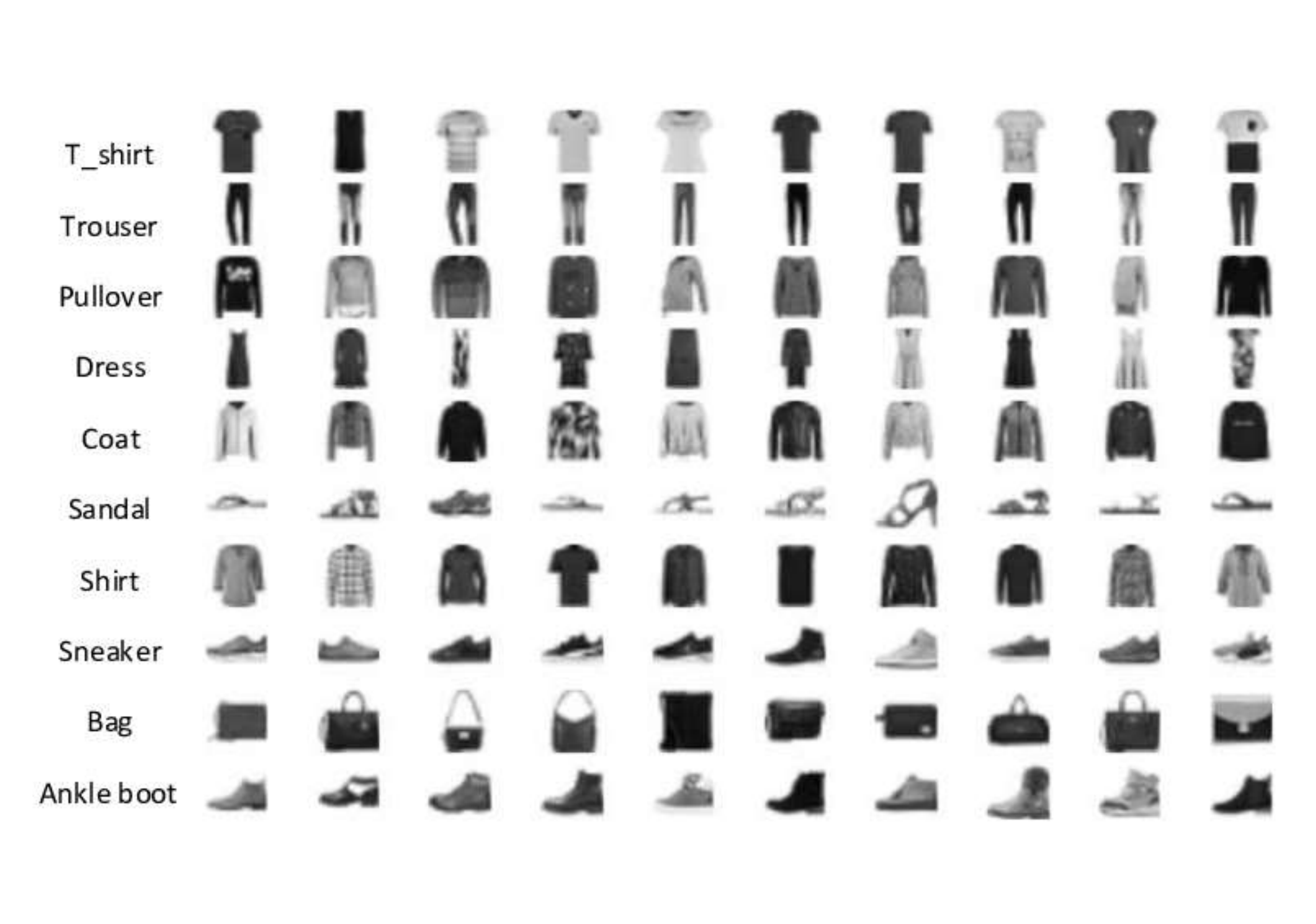
MNIST Fashion, also known as Fashion-MNIST, is a dataset designed to benchmark machine learning algorithms in image recognition tasks. It was created as a more challenging alternative to the original MNIST dataset, which consists of handwritten digits.

CONTENTS OF

- 70,000 black and white images depicting 10 categories of clothing and accessories, such as shirts, pants, coats, dresses, bags, etc.

- Each image is 28x28 pixels in size.

 - The collection is divided into a training set (60,000 images) and a test set (10,000 images)



PANDAS

- Allows the use of structures ie: DataFrame and Series,Allows retrieval of data from various sources, e.g.: database, CSVSupports cleaning, transformation and aggregation process

PANDAS

PROS

- **Ease of data handling -** makes it easy to load, process and analyze data from various sources, offering intuitive structures Rich support for data operations - the library offers a wide range of functions for data cleaning, selection, filtering, aggregation and data transformation
- **Integration with other libraries -** integrates perfectly with many other libraries used in data analysis and machine learning, such as Scikit-learn.

CONS

- **Memory consumption -** It can be inefficient in terms of memory consumption, especially when handling very large data sets.
- **Performance with very large data -** Although Pandas is efficient at processing and analyzing moderately sized data, it can encounter performance problems with very large sets where specialized data processing tools may be more appropriate.

SCIKIT-LEARN

Includes a wide range of algorithms for classification, regression, clustering, etc.

- Provides tools for model validation and feature selection

SCIKIT-LEARN

PROS

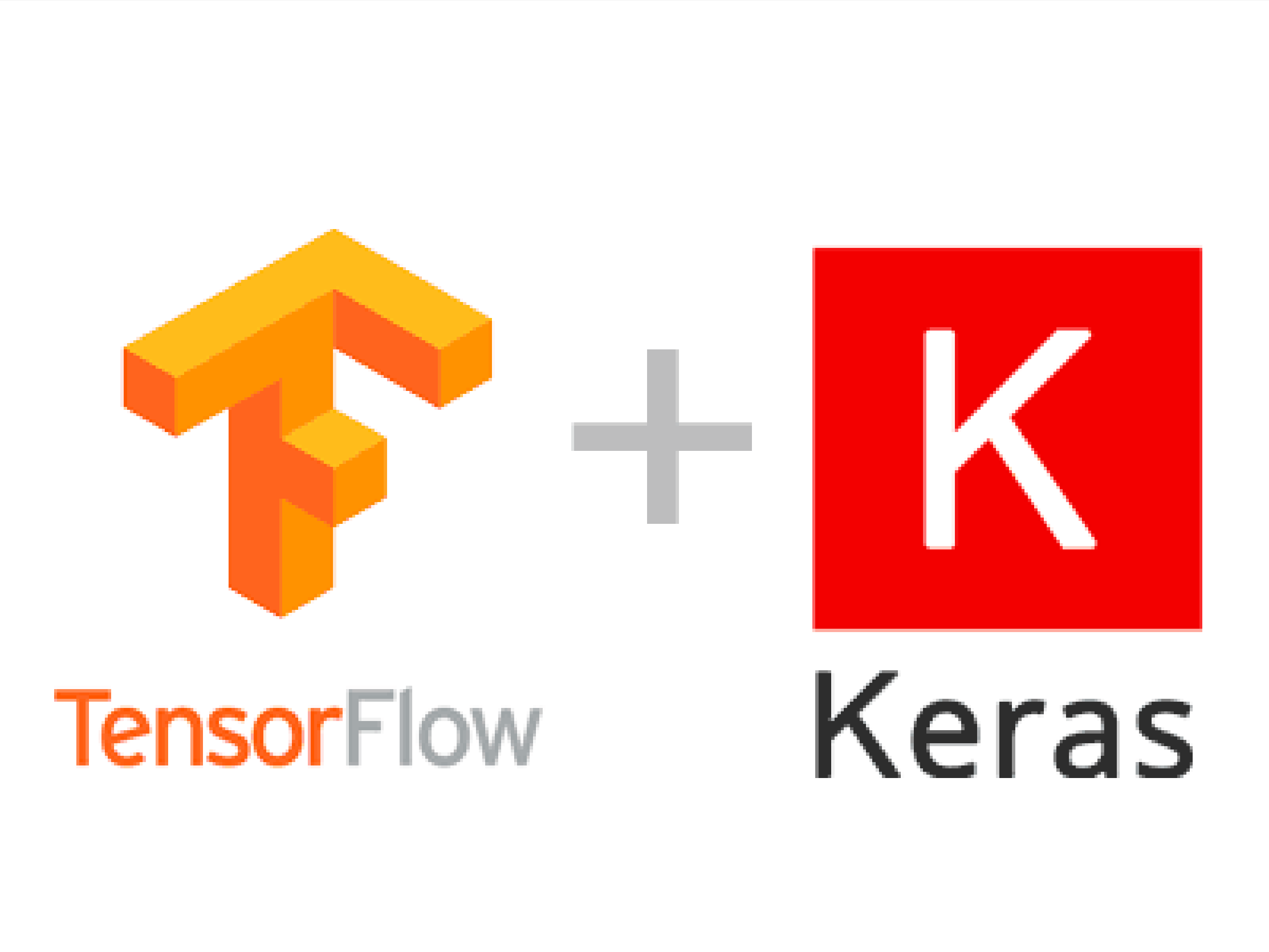
- **Wide range of algorithms -** offers an extensive collection of machine learning algorithms that are ready to use, including classification, regression, clustering and dimensionality reduction.
- **Documentation and community support -** has well-written documentation. In addition, the library has a large and active user community.
- **Ease of integration and use -** the library is designed with ease of use in mind, offering a consistent API that is well integrated with other popular Python scientific libraries such as NumPy, SciPy and Pandas.

CONS

- **Limitations in scalability -** not the best choice for very large data sets. More scalable tools may be needed for Big Data applications.
- **Lack of support for deep learning models**

TENSORFLOW/KERAS

- TensorFlow is an open-source library for numerical computation, particularly well-suited for large-scale Machine Learning and deep learning tasks. Is used for implementing and deploying machine learning applications.
- The Keras module within TensorFlow, known as tf.keras, is a high-level API that makes it easy to build and train deep learning models. Originally an independent framework, Keras is designed to be user-friendly, modular, and extensible. It provides essential abstractions and building blocks for developing deep learning models, including a wide range of pre-built layers, optimizers, and tools to facilitate the easy construction, training, and evaluation of neural networks. With tf.keras, users can quickly prototype and experiment with different models, making it accessible for beginners while still flexible enough to support complex research.



TENSORFLOW/KERAS

PROS

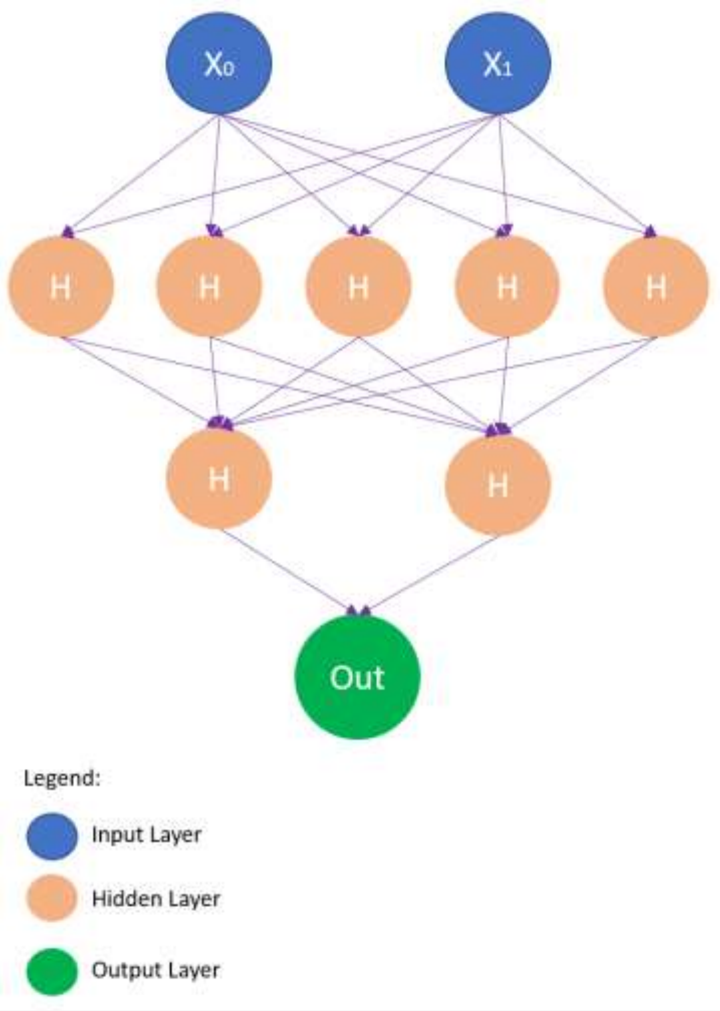
- **User-Friendly and High-Level API:** intuitive and user-friendly interface. It allows for quick prototyping and experimentation, making it accessible for beginners and efficient for professionals.
- **Scalability and Flexibility. I**s highly scalable, allowing models to be deployed on various platforms, from mobile devices to distributed cloud computing setups. Its flexibility enables it to handle different types of neural networks, from simple to complex, and adapt to a wide range of tasks.

CONS

- **Resource-Intensive**: TensorFlow, especially with Keras, can be resource-intensive, requiring significant computational resources, particularly for training large models. This can be a limitation for users without access to high-performance computing resources.
- Understanding its lower-level features and optimizing models for performance might require a deeper understanding of the framework.

MLPCLASSIFIER

- **Description**:
- MLP (multilayer perceptron) based classifier. It works on the principle of feedforward neural networks. The model consists of one input layer, one or more hidden layers and one output layer. During the learning phase, MLPClassifier adjusts the weights using a back propagation algorithm, and optimizing the loss function
- **Applications**: Classification where it is necessary to distinguish between two or more classes of labels
- **Pros**: Versatility and customizability.
- **Cons**: Computational complexity, risk of overfitting, need to select hyperparameters



KNEIGHBORSCLASSIFIER

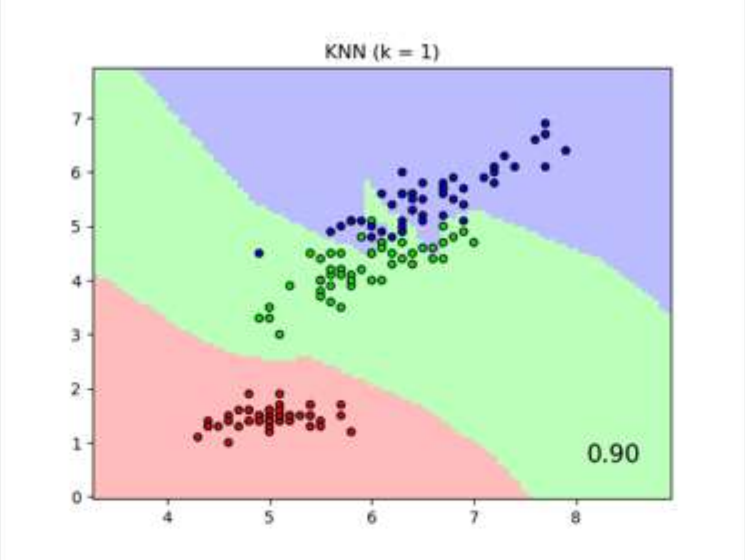
Description:

Neighborhood-based classification algorithm.This algorithm classifies objects based on the voting of their nearest neighbors.An object is assigned to the class that is most frequent among its k nearest neighbors. Neighbors are selected based on their distance in the feature space, which can be calculated using metrics.

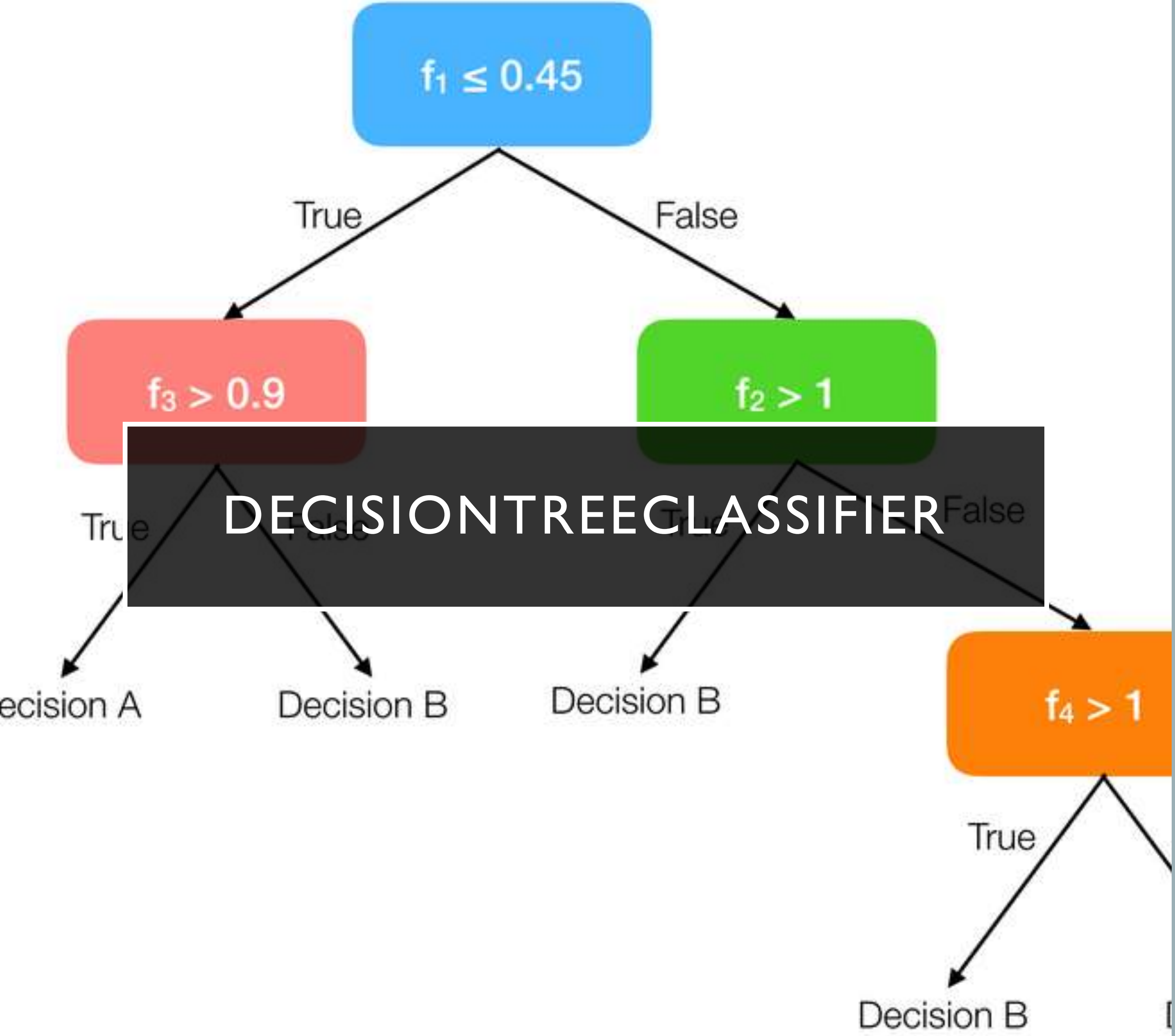
Applications:When boundaries between classes are not clearly defined and where the data is non-linear in nature

Pros: Simple and intuitive, effective (under certain conditions)

Cons: Susceptibility to improperly scaled features, "curse of dimensionality"



DECISIONTREECLASSIFIER



Description:

A machine learning algorithm that creates a model in the form of a decision tree. It works by dividing a set of data into smaller and smaller subsets based on feature values. At each node of the tree, the algorithm selects the feature that most efficiently separates the data into classes, and the process continues recursively until certain stopping criteria are reached,.

Applications: By being relatively easy to understand and visualize, are popular in exploratory data analysis and in situations where model interpretability is important

Pros: Interpretability, does not require complex data processing

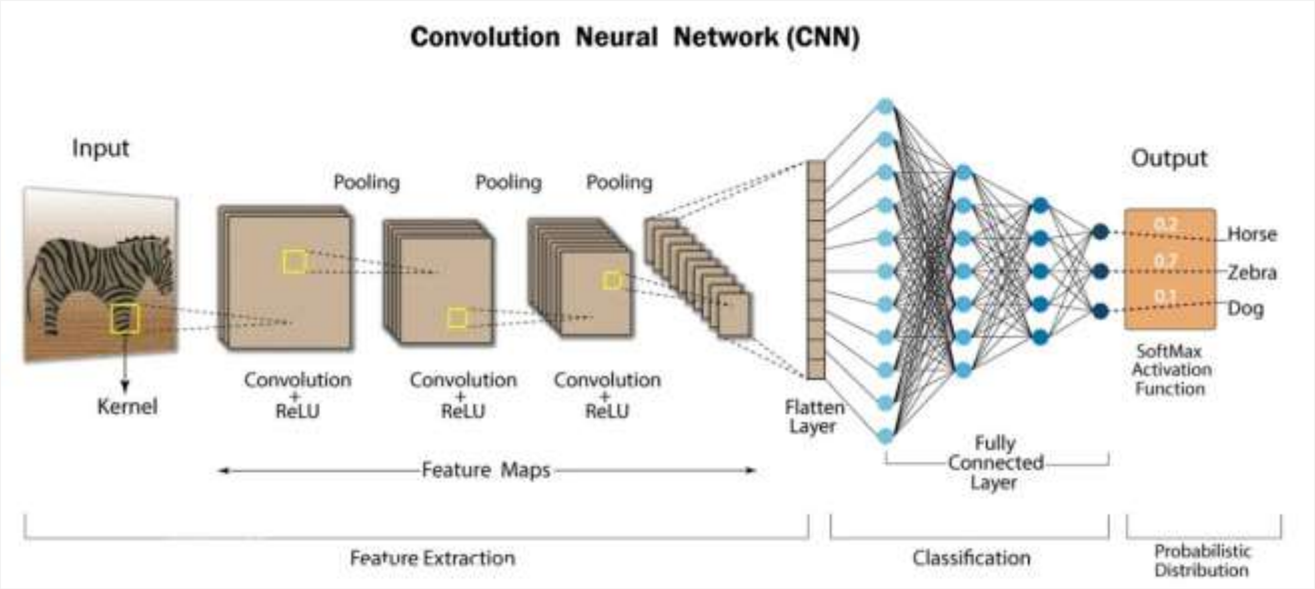
Cons:Tendency to overfitting, instability

CONVOLUTIONAL NEURAL NETWORK

- Description:
Deep learning model designed for processing structured grid-like data, such as images. It consists of multiple layers, including convolutional layers that apply filters to extract features, pooling layers that reduce dimensionality, and fully connected layers that interpret features for classification or other tasks. CNNs excel in visual recognition due to their ability to automatically learn spatial hierarchies of features from data, making them widely used in image and video recognition tasks.

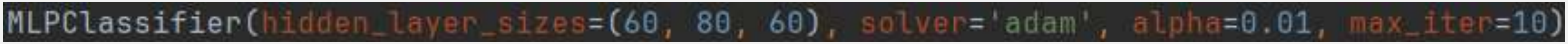
- Advantages: highly effective for image recognition and processing due to their ability to automatically learn and detect features hierarchically. Ideal for tasks like object detection, facial recognition, and image classification.

- Disadvantages: CNNs require a large amount of labeled training data to perform well, especially for deep networks. Additionally, they are computationally intensive, requiring significant computational resources for training and deployment.



MLPCLASSIFIER - PARAMETERS

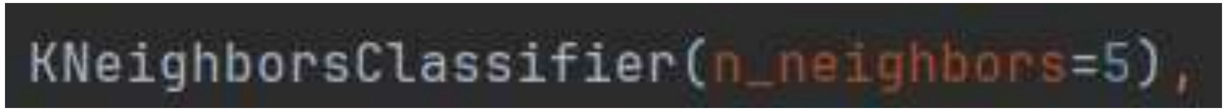
- **hidden_layer_sizes -** determines the number and size of hidden layers in a neural network. Networks that are too small may not be able to learn adequately from the data, while networks that are too large may lead to overlearning.
- **solver -** specifies the algorithm used to optimize the network weights in the learning process.
- *adam*: recommended for most cases and copes well with large data sets,-
- *sgd*: allows more detailed adjustment of the learning
- **processalpha -** helps prevent overlearning by limiting large values of weights in the model. Lower values correspond to better fitting, and higher values generalize
- **max_iter -** amount of learning iteration.



```
MLPClassifier(hidden_layer_sizes=(60, 80, 60), solver='adam', alpha=0.01, max_iter=10)
```

KNEIGHBORSCLASSIFIER - PARAMETERS

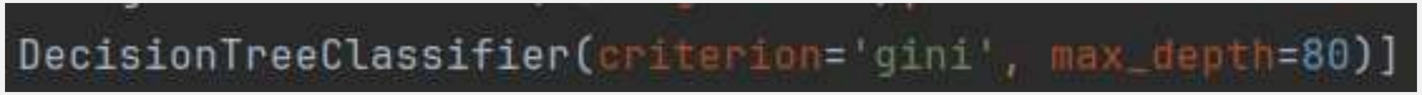
n_neighbors - specifies the number of nearest neighbors to be considered for class assignment. It directly affects the results of classification: too few neighbors can lead to a model that is overly sensitive to noise in the data (overfitting), while too many can cause overgeneralization,



```
KNeighborsClassifier(n_neighbors=5),
```

DECISIONTREECLASSIFIER - PARAMETERS

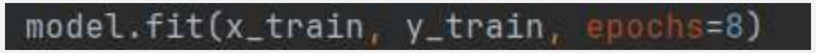
- **criterion -** a function that is used to evaluate the quality of a split in a decision tree.
- *gini* - evaluates how often an element randomly selected from a set will be misclassified,
- *entropy* - uses a measure of information entropy to evaluate disorder in a data set. Gini typically runs faster, and entropy can lead to more balanced
- **treesmax_depth-** specifies the maximum depth of the decision tree. Limiting this parameter is one way to control the risk of overfitting, as it prevents the creation of overly complex models that literally "memorize" the training data.



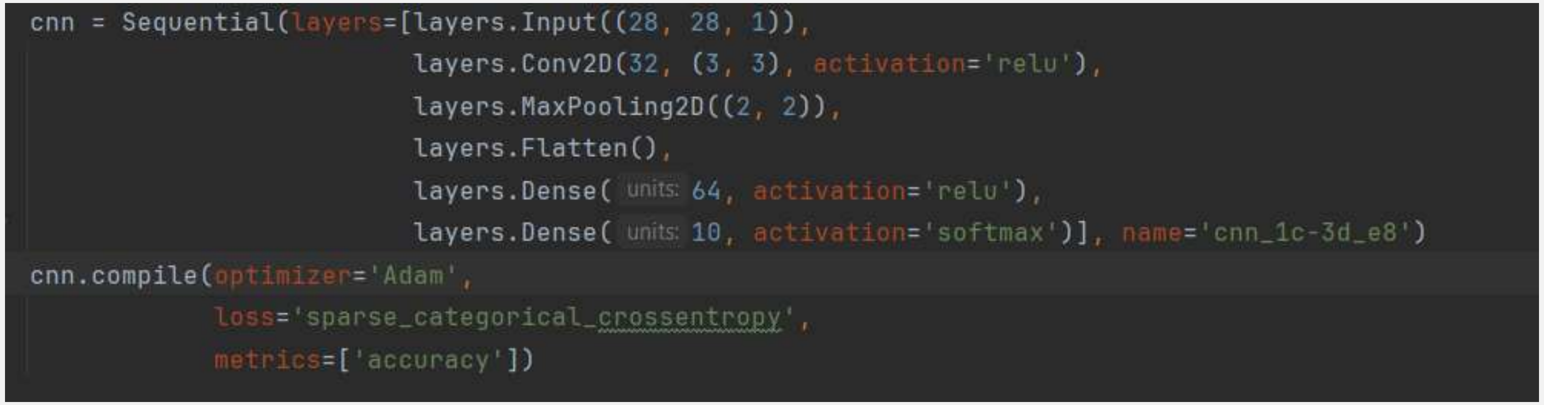
```
DecisionTreeClassifier(criterion='gini', max_depth=80)]
```

CONVOLUTIONAL NEURAL NETWORK- PARAMETERS

- **epochs** – Increasing the number of epochs allows the model to make more passes through the training data, which can improve accuracy. Too many epochs can lead to overfitting and increase training time
- **Amount of layers.Conv2D -** Increased amount of layers allows the network to capture more complex features, leading to potentially better performance on intricate tasks. Decreased Layers can speed up training and reduce the computational requirements.
- **Amount of layers.Dense -** Adding more dense layers can improving the model's capacity for tasks like classification. This can also lead to overfitting, particularly if the dataset is not large enough. Reducing the number of Dense layers simplifies the model, which can make it less prone to overfitting and faster to train.



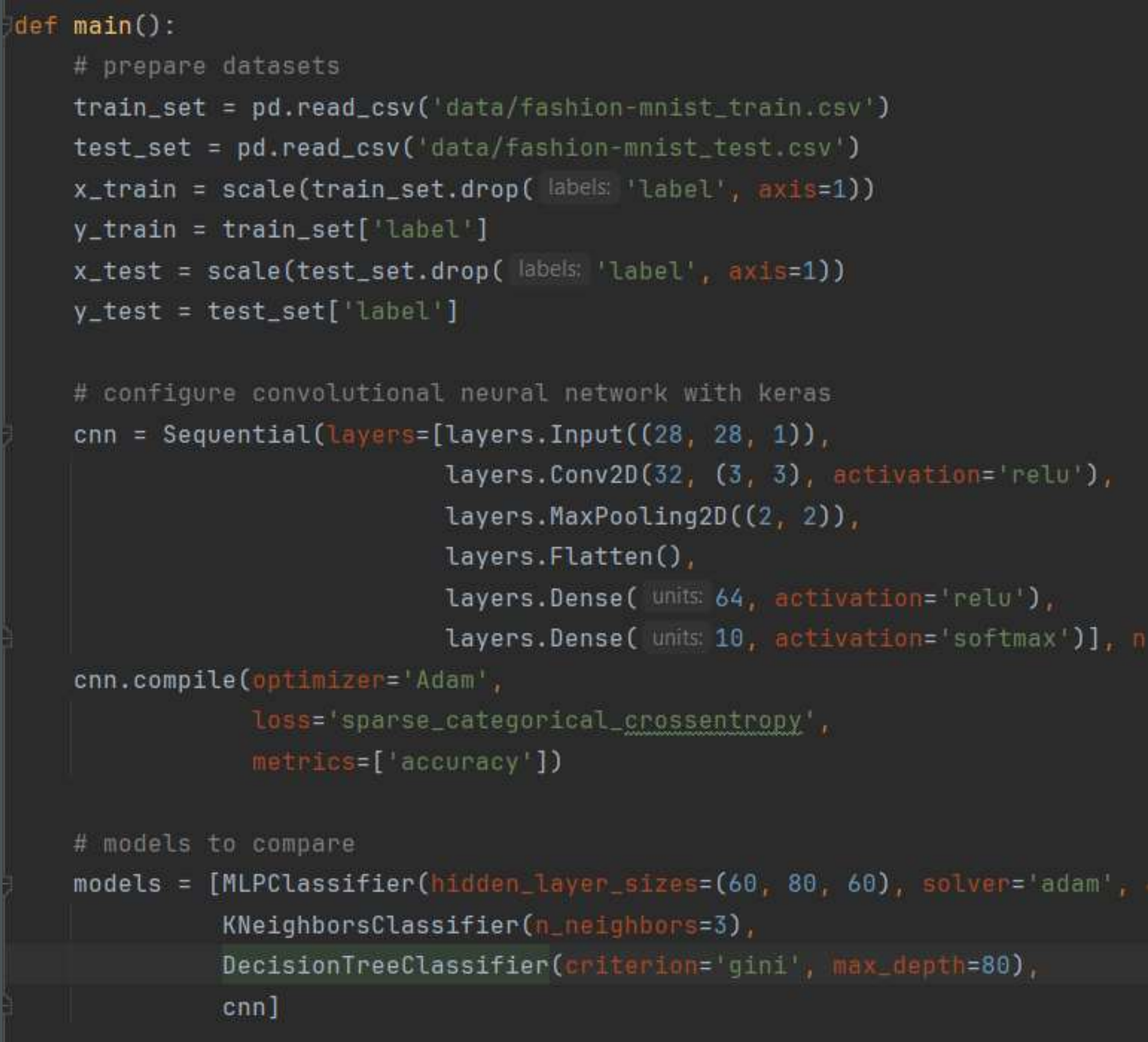
```
model.fit(x_train, y_train, epochs=8)
```



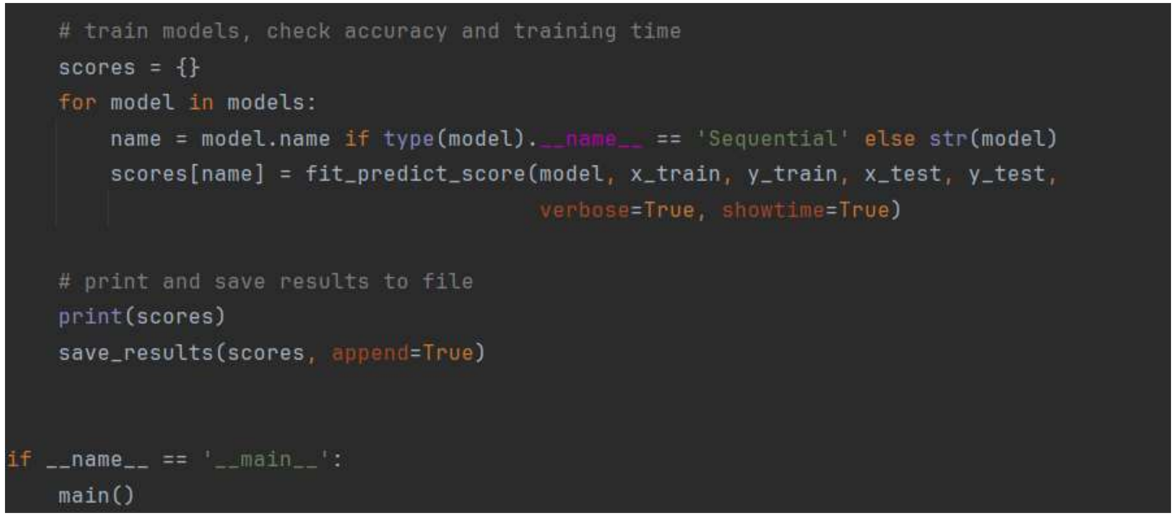
```
cnn = Sequential(layers=[layers.Input((28, 28, 1)),
                         layers.Conv2D(32, (3, 3), activation='relu'),
                         layers.MaxPooling2D((2, 2)),
                         layers.Flatten(),
                         layers.Dense( units: 64, activation='relu'),
                         layers.Dense( units: 10, activation='softmax')], name='cnn_1c-3d_e8')
cnn.compile(optimizer='Adam',
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy'])
```

IMPLEMENTATION

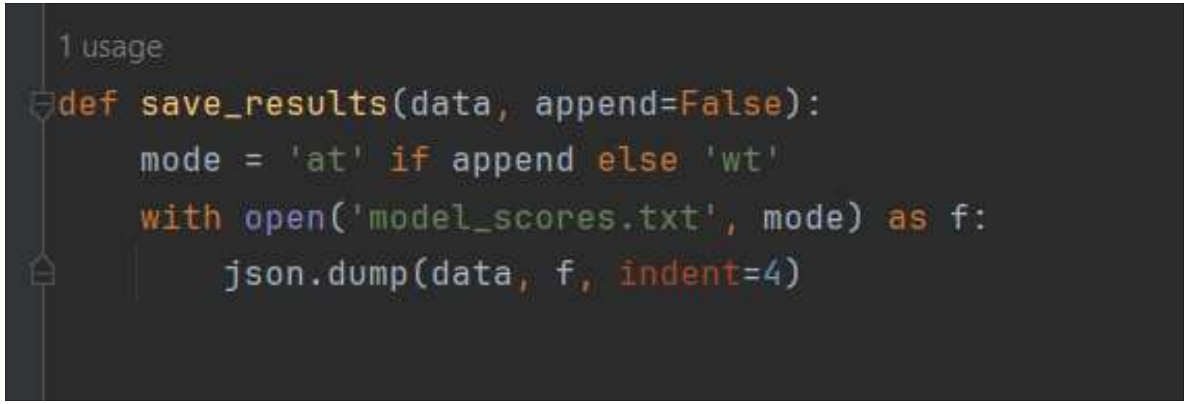
- Load data from csv file using pandas library (training and test data are in separate files),
- x_train - test data,
- y_train - labels for test data,
- x_test - test data,
- y_test - labels for test data,
- cnn - configuration of convolutional neural network
- models - models used



```
def main():
    # prepare datasets
    train_set = pd.read_csv('data/fashion-mnist_train.csv')
    test_set = pd.read_csv('data/fashion-mnist_test.csv')
    x_train = scale(train_set.drop( labels: 'label', axis=1))
    y_train = train_set['label']
    x_test = scale(test_set.drop( labels: 'label', axis=1))
    y_test = test_set['label']

    # configure convolutional neural network with keras
    cnn = Sequential(layers=[layers.Input((28, 28, 1)),
                             layers.Conv2D(32, (3, 3), activation='relu'),
                             layers.MaxPooling2D((2, 2)),
                             layers.Flatten(),
                             layers.Dense( units: 64, activation='relu'),
                             layers.Dense( units: 10, activation='softmax')], n
    cnn.compile(optimizer='Adam',
                loss='sparse_categorical_crossentropy',
                metrics=['accuracy'])

    # models to compare
    models = [MLPClassifier(hidden_layer_sizes=(60, 80, 60), solver='adam', 
              KNeighborsClassifier(n_neighbors=3),
              DecisionTreeClassifier(criterion='gini', max_depth=80),
              cnn]
```



```
    # train models, check accuracy and training time
    scores = {}
    for model in models:
        name = model.name if type(model).__name__ == 'Sequential' else str(model)
        scores[name] = fit_predict_score(model, x_train, y_train, x_test, y_test,
                                         verbose=True, showtime=True)

    # print and save results to file
    print(scores)
    save_results(scores, append=True)


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



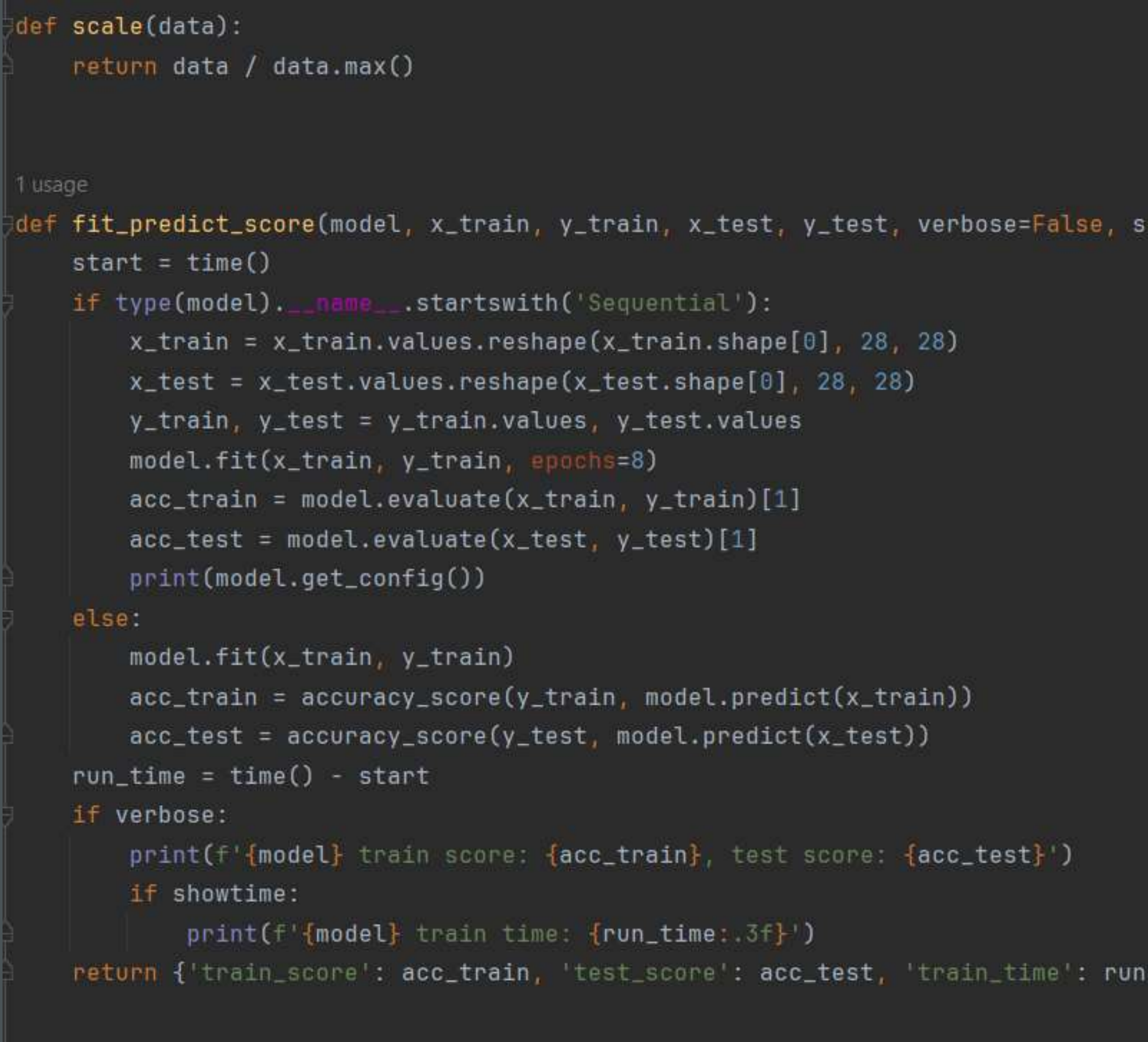
```
1 usage
def save_results(data, append=False):
    mode = 'at' if append else 'wt'
    with open('model_scores.txt', mode) as f:
        json.dump(data, f, indent=4)
```

IMPLEMENTATION

Storage result data with configuration parameters in file in JSON format

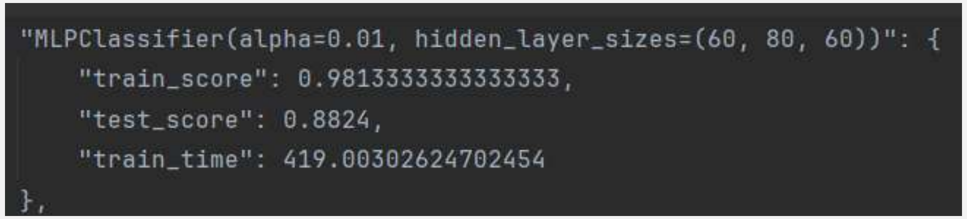
IMPLEMENTATION

- **scale -** scaling the value,
- **fit_predict_score**:
- If statement preparing data for particular models needs.
- **- model.fit** – model learning,
  - acc_train – model performance on training data,
  - acc_test – model performance on test data,
  - run_time – learning time

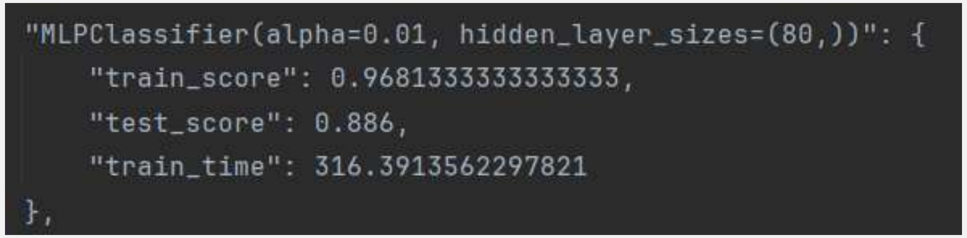


```python
def scale(data):
    return data / data.max()


def fit_predict_score(model, x_train, y_train, x_test, y_test, verbose=False, s
    start = time()
    if type(model).__name__.startswith('Sequential'):
        x_train = x_train.values.reshape(x_train.shape[0], 28, 28)
        x_test = x_test.values.reshape(x_test.shape[0], 28, 28)
        y_train, y_test = y_train.values, y_test.values
        model.fit(x_train, y_train, epochs=8)
        acc_train = model.evaluate(x_train, y_train)[1]
        acc_test = model.evaluate(x_test, y_test)[1]
        print(model.get_config())
    else:
        model.fit(x_train, y_train)
        acc_train = accuracy_score(y_train, model.predict(x_train))
        acc_test = accuracy_score(y_test, model.predict(x_test))
    run_time = time() - start
    if verbose:
        print(f'{model} train score: {acc_train}, test score: {acc_test}')
        if showtime:
            print(f'{model} train time: {run_time:.3f}')
    return {'train_score': acc_train, 'test_score': acc_test, 'train_time': run
```

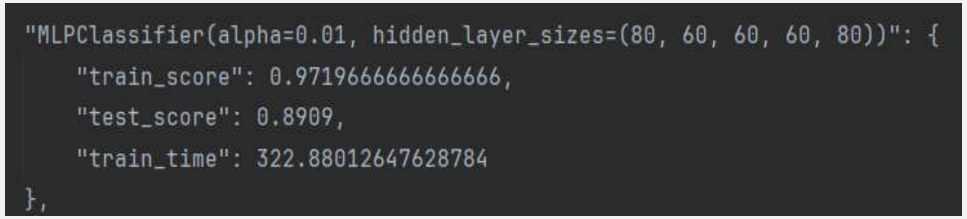
ANALYSIS OF RESULTS



```
"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
},
```

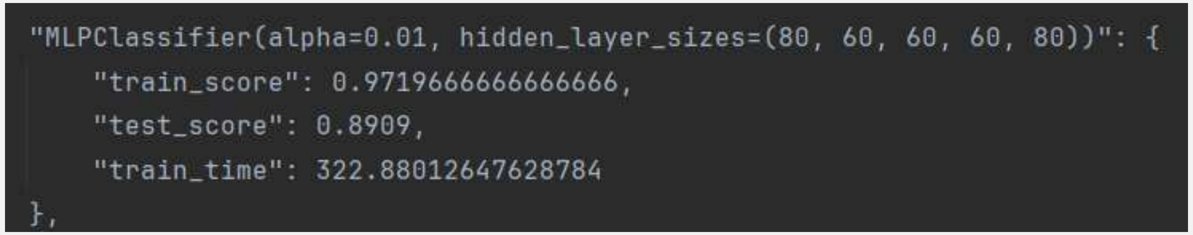


```
"MLPClassifier(alpha=0.01, hidden_layer_sizes=(80, 60, 60, 60, 80))": {
    "train_score": 0.9719666666666666,
    "test_score": 0.8909,
    "train_time": 322.88012647628784
},
```

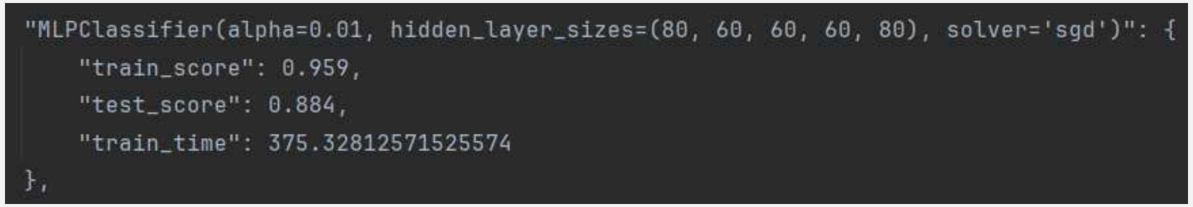
MLPClassifier (hidden_layer_size)
The results, when changing the parameter, are comparable to each other. Increasing the number of layers does not necessarily go hand in hand with an increase in the training time of the model.

It can be concluded that even the number of hidden layers equal to one is sufficient because the performance on the test data is comparable, and it has the least chance of over-fitting the data.

ANALYSIS OF 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
},
```

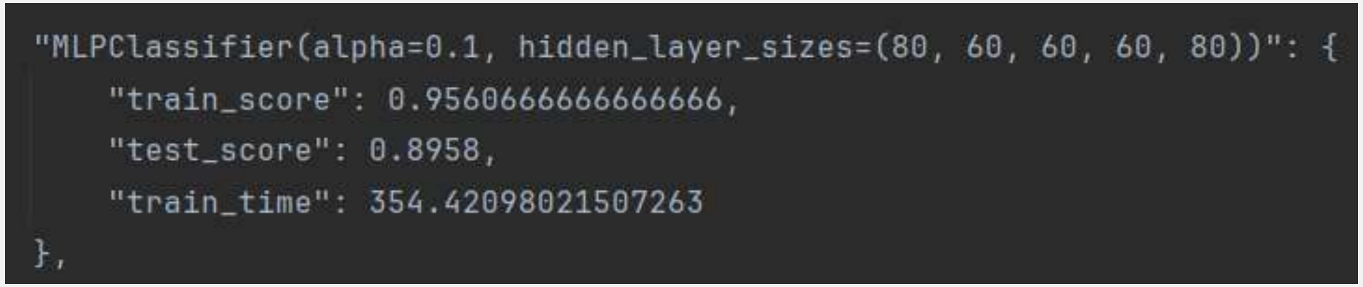


```
"MLPClassifier(alpha=0.01, hidden_layer_sizes=(80, 60, 60, 60, 80), solver='sgd')": {
    "train_score": 0.959,
    "test_score": 0.884,
    "train_time": 375.32812571525574
},
```

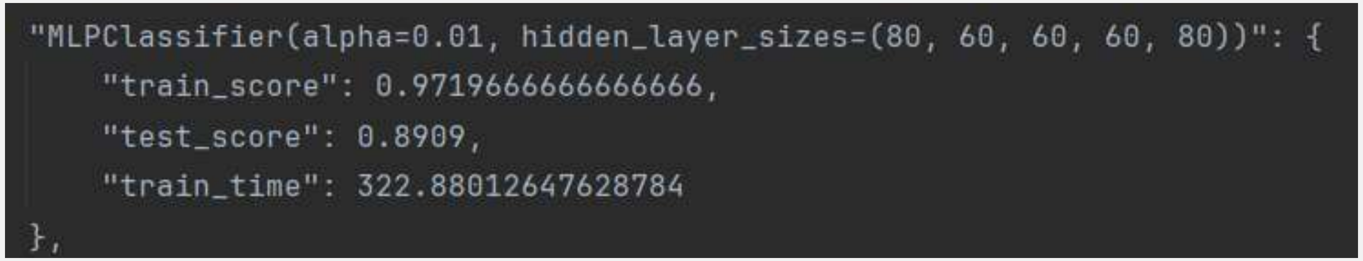
MLPClassifier (solver)

Changing the model from 'adam' to 'sgd' results in decreased accuracy and increased model training time. For test subjects, a more generic solver works better.

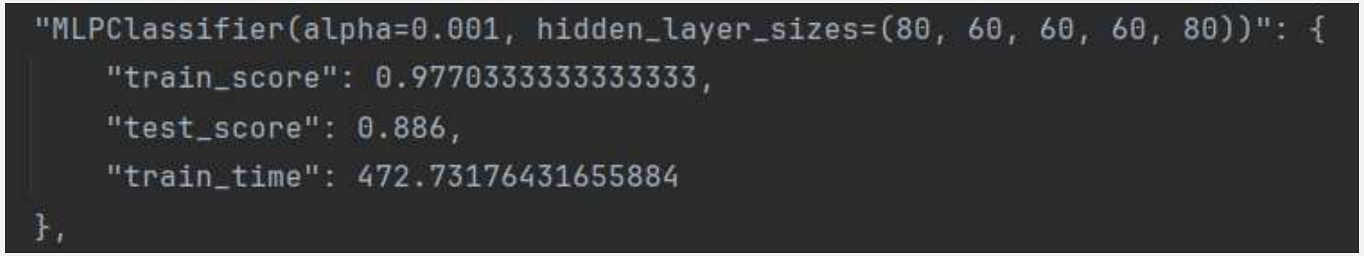
ANALYSIS OF RESULTS



```
"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
},
```

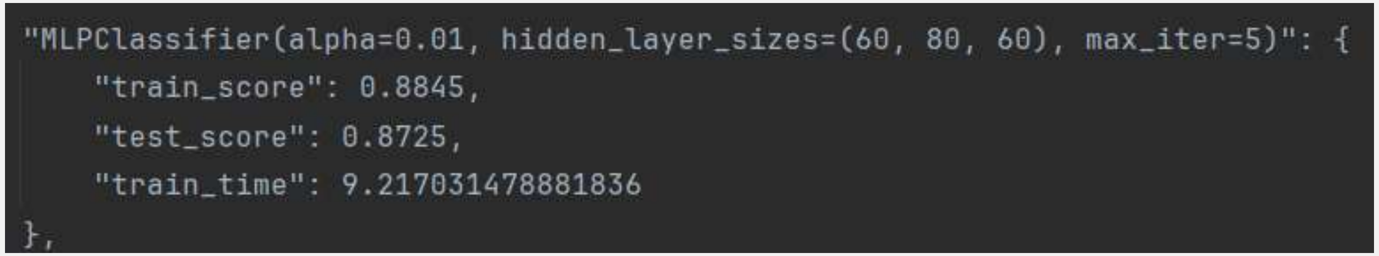


```
"MLPClassifier(alpha=0.001, hidden_layer_sizes=(80, 60, 60, 60, 80))": {
    "train_score": 0.9770333333333333,
    "test_score": 0.886,
    "train_time": 472.73176431655884
},
```

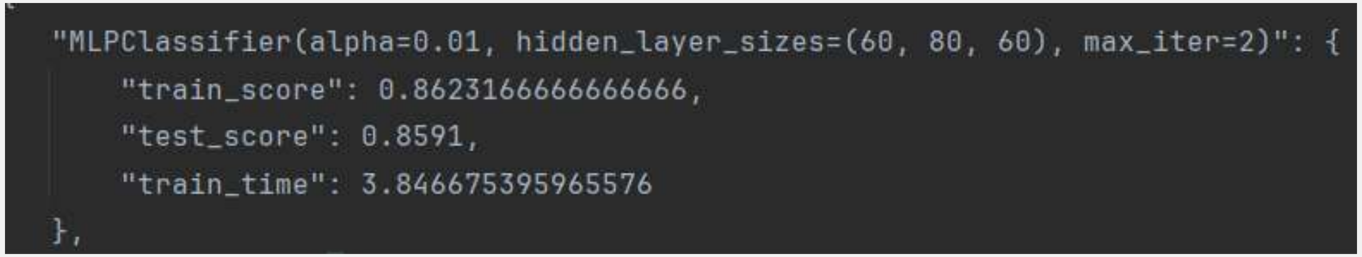
MLPClassifier (alpha)

The best result was achieved by the highest alpha value, which means that it is beneficial to limit over-adaptation to the test data. The highest value of the alpha parameter further increased the learning time of the model.

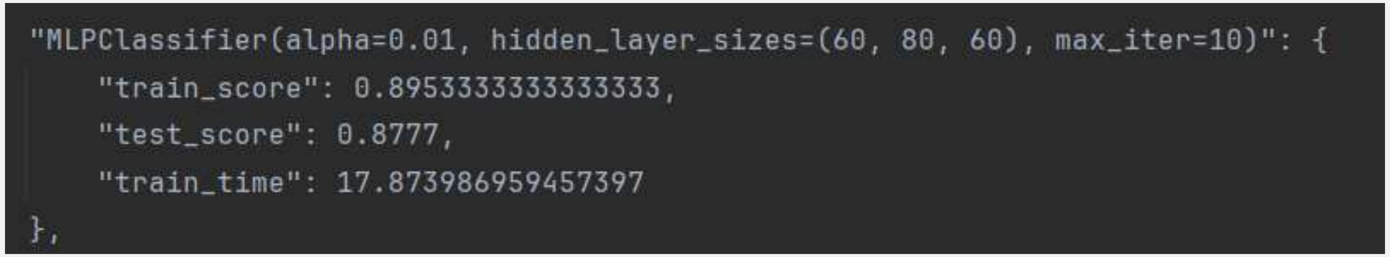
ANALYSIS OF RESULTS



```
"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
},
```

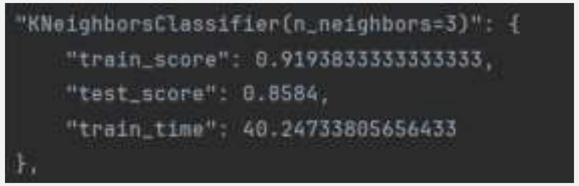


```
"MLPClassifier(alpha=0.01, hidden_layer_sizes=(60, 80, 60), max_iter=10)": {
    "train_score": 0.8953333333333333,
    "test_score": 0.8777,
    "train_time": 17.873986959457397
},
```

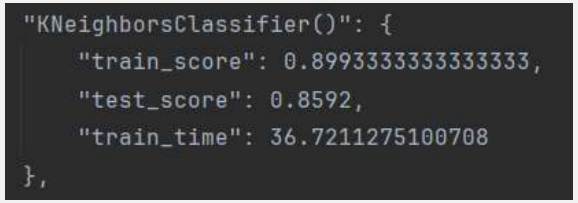
MLPClassifier (max_iter)

The best result for the MLPClassifier was achieved with max_iter = 10. However, the difference in performance between max_iter = 5 and max_iter = 10 was not significant. The computation time for max_iter = 10 was twice as long compared to max_iter = 5.

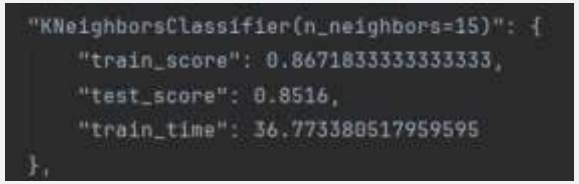
ANALYSIS OF RESULTS



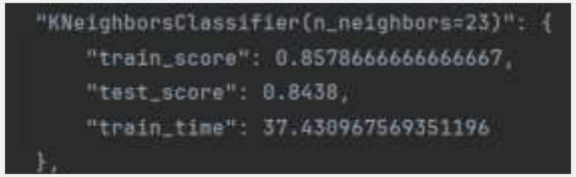
```
"KNeighborsClassifier(n_neighbors=3)": {
    "train_score": 0.9193833333333333,
    "test_score": 0.8584,
    "train_time": 40.24733805656433
},
```



```
"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
},
```

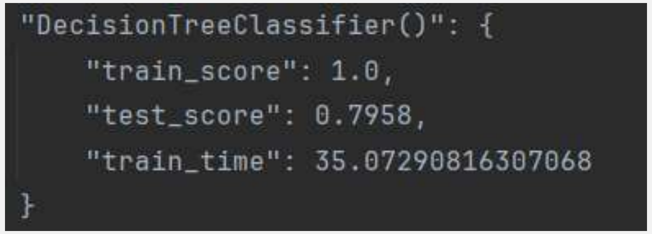


```
"KNeighborsClassifier(n_neighbors=23)": {
    "train_score": 0.8578666666666667,
    "test_score": 0.8438,
    "train_time": 37.430967569351196
},
```

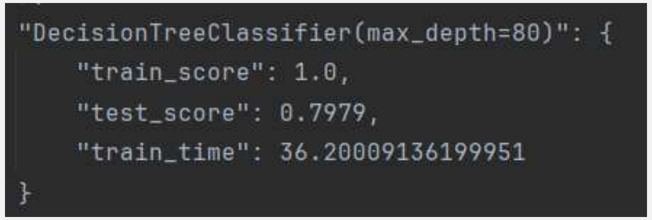
KNeighborsClassifier(n_neighbors)

The best result was obtained with the default value of the parameter, i.e. 5. Too few neighbors can lead to overfitting. However, with the received values of train_score, it is rather possible to exclude such an option. There is no major change in learning times.

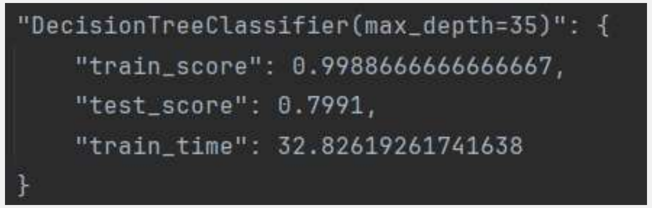
ANALYSIS OF RESULTS



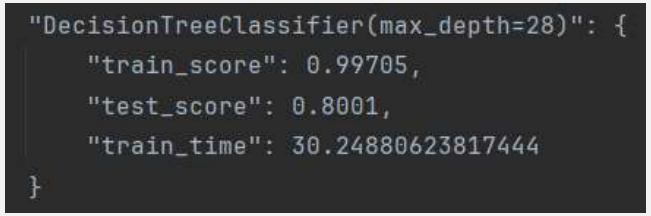
```
"DecisionTreeClassifier()": {
    "train_score": 1.0,
    "test_score": 0.7958,
    "train_time": 35.07290816307068
}
```



```
"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
}
```

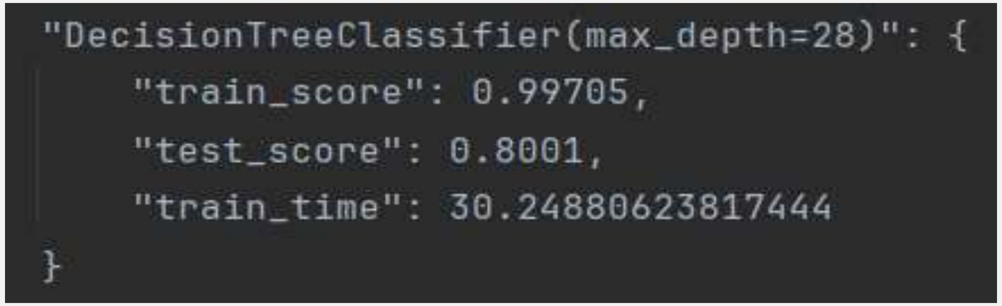


```
"DecisionTreeClassifier(max_depth=28)": {
    "train_score": 0.99705,
    "test_score": 0.8001,
    "train_time": 30.24880623817444
}
```

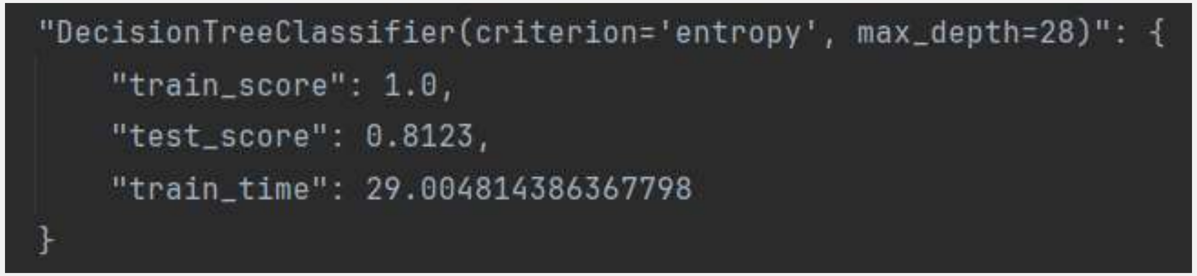
DecisionTreeClassifier(max_depth)

Limiting the max_depth value had a positive effect on the results. The values of None, 80 and 35 resulted in an accuracy for the training data close to 1, which indicates overtraining of the model.

ANALYSIS OF RESULTS



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

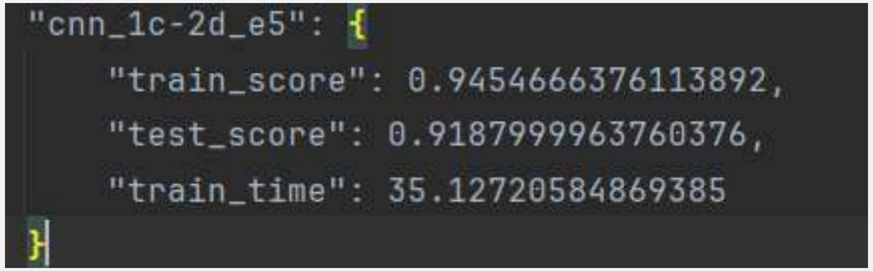


```
"DecisionTreeClassifier(criterion='entropy', max_depth=28)": {
    "train_score": 1.0,
    "test_score": 0.8123,
    "train_time": 29.004814386367798
}
```

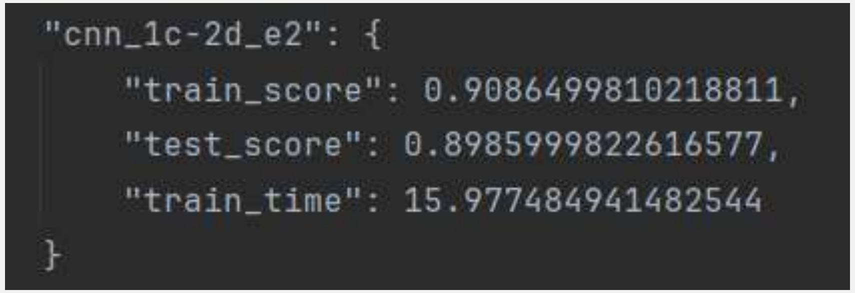
DecisionTreeClassifier(criterion)

The use of the entropy function instead of gini results in an increase in accuracy on the test data but also increases the train_score again

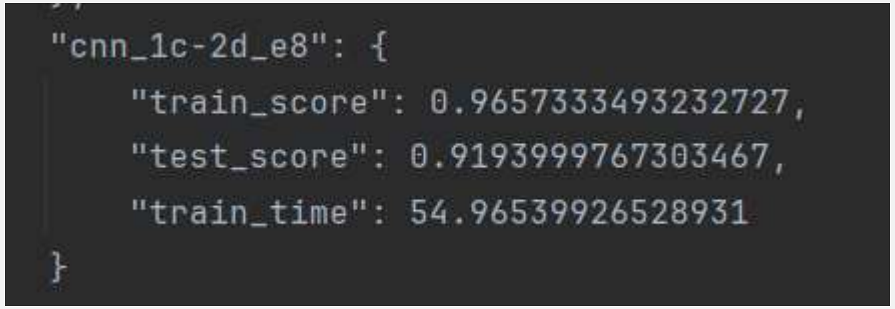
ANALYSIS OF RESULTS



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



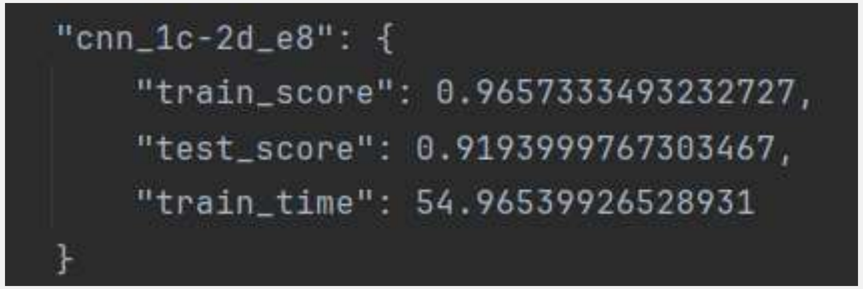
```
"cnn_1c-2d_e2": {
    "train_score": 0.9086499810218811,
    "test_score": 0.8985999822616577,
    "train_time": 15.977484941482544
}
```



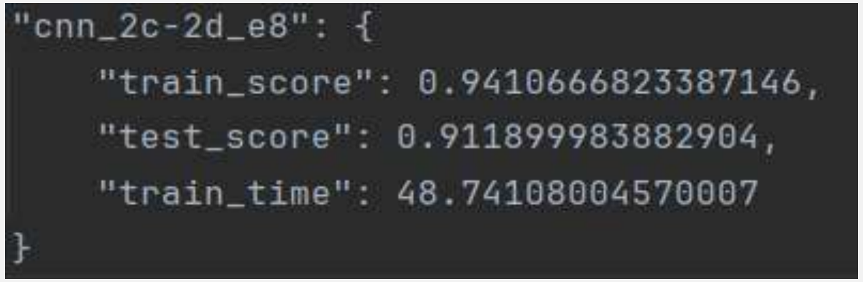
```
"cnn_1c-2d_e8": {
    "train_score": 0.9657333493232727,
    "test_score": 0.9193999767303467,
    "train_time": 54.96539926528931
}
```

The highest accuracy on the test data was achieved with 8 epochs. As the number of epochs increases, one can observe an improvement in accuracy but also a significant increase in time. The difference in accuracy between the test and training data is not significant, which suggests that overfitting is not occurring.

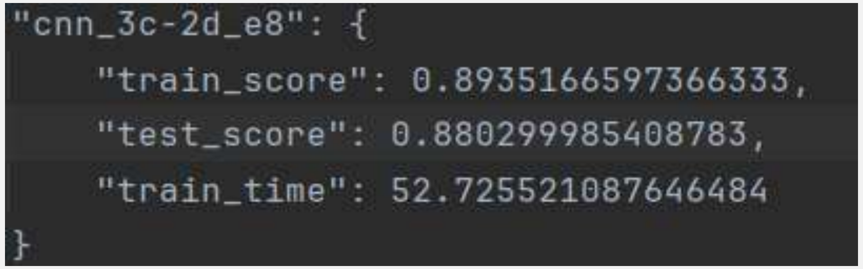
ANALYSIS OF RESULTS



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



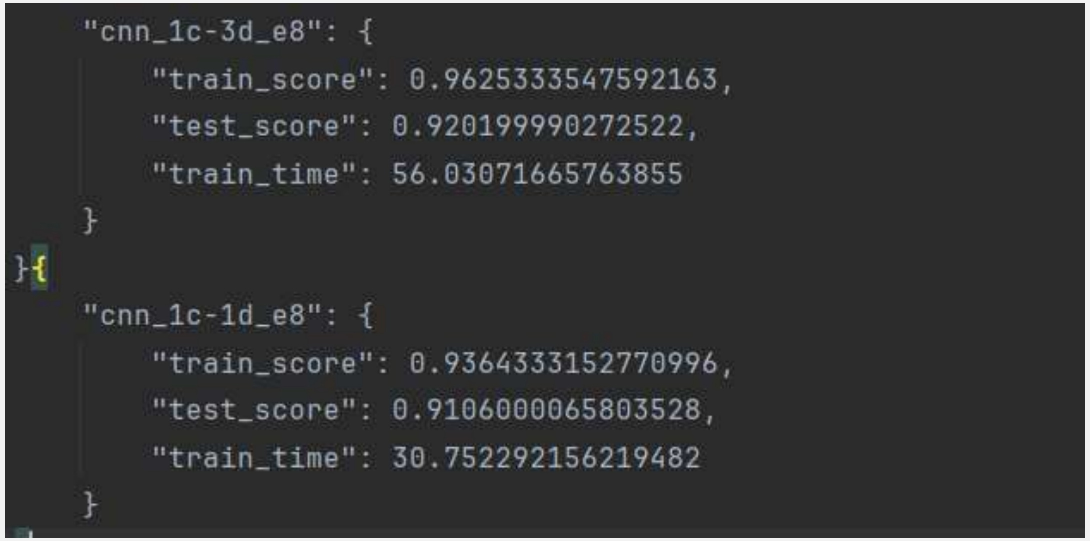
```
"cnn_2c-2d_e8": {
    "train_score": 0.9410666823387146,
    "test_score": 0.911899983882904,
    "train_time": 48.74108004570007
}
```



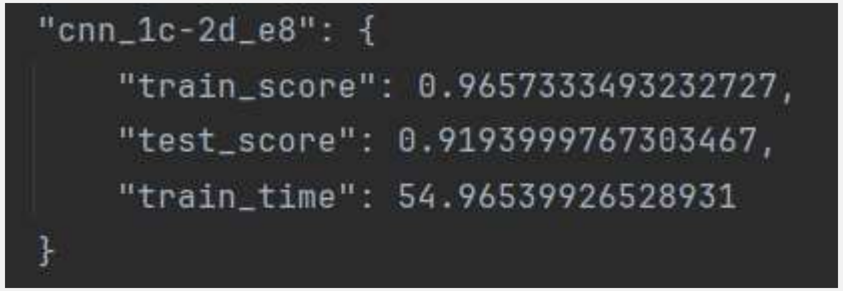
```
"cnn_3c-2d_e8": {
    "train_score": 0.8935166597366333,
    "test_score": 0.880299985408783,
    "train_time": 52.725521087646484
}
```

The best result was achieved by the model with a single convolutional layer. As the number of layers increased, a decrease in accuracy was observed. This could be due to a mismatch between the convolutional layer's parameters and the provided data. In all reults training time was simmilar.

ANALYSIS OF RESULTS



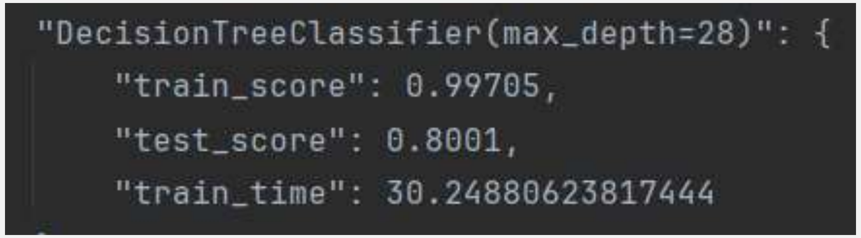
```
    "cnn_1c-3d_e8": {
        "train_score": 0.9625333547592163,
        "test_score": 0.920199990272522,
        "train_time": 56.03071665763855
    }
}{
    "cnn_1c-1d_e8": {
        "train_score": 0.9364333152770996,
        "test_score": 0.9106000065803528,
        "train_time": 30.752292156219482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



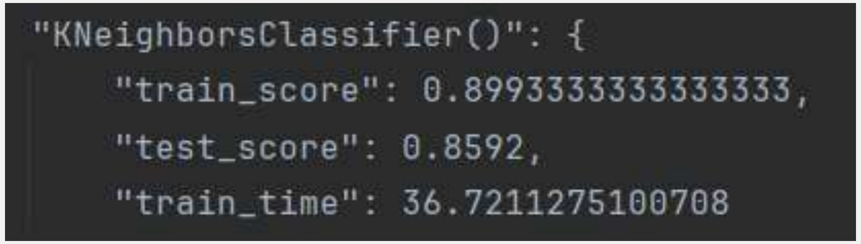
```
"cnn_1c-2d_e8": {
    "train_score": 0.9657333493232727,
    "test_score": 0.9193999767303467,
    "train_time": 54.96539926528931
}
```

The network with 3 dense layers achieved the best result. However, it's worth noting that the accuracy in all cases was within the margin of error. A significant increase in training time was observed between 1 and 2 dense layers. This could be because, with just one layer, we only have the output layer, which differs in its nature from the other dense layers.

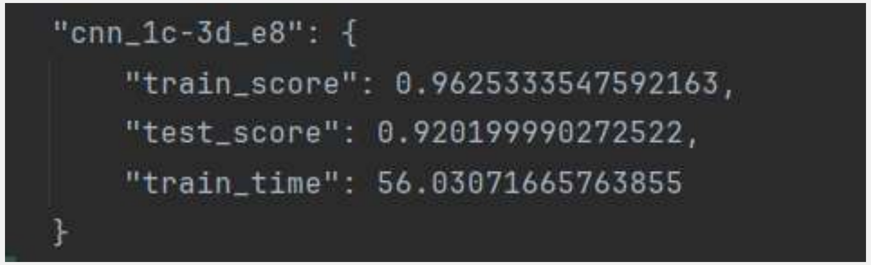
ANALYSIS OF RESULTS



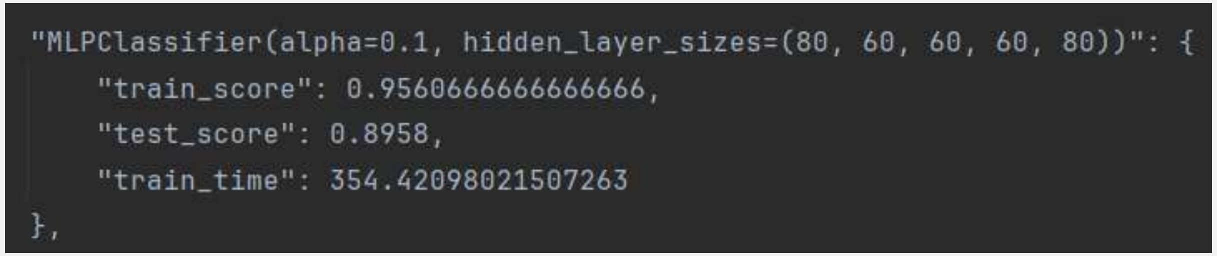
```
"DecisionTreeClassifier(max_depth=28)": {
    "train_score": 0.99705,
    "test_score": 0.8001,
    "train_time": 30.24880623817444
```



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



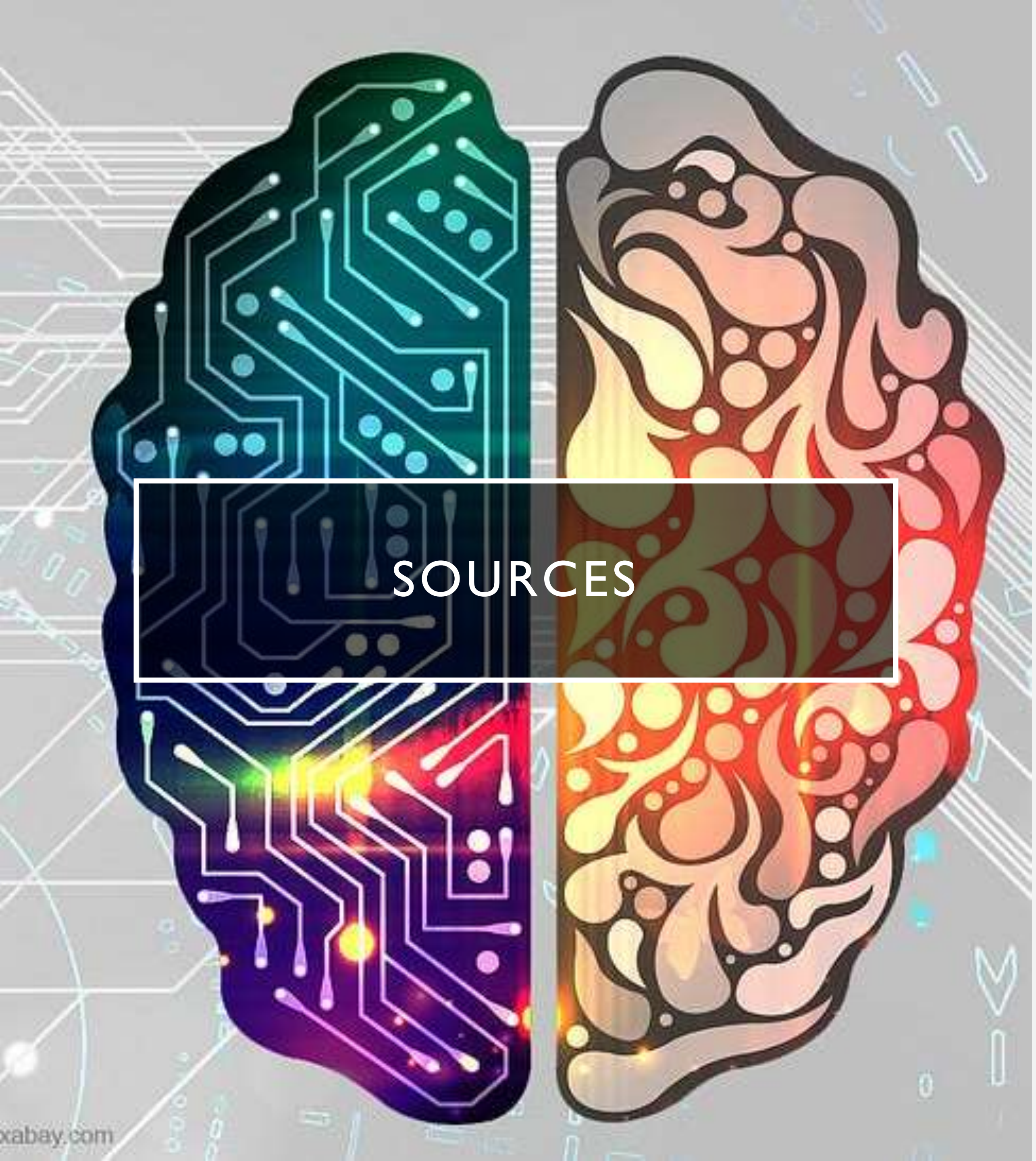
```
"cnn_1c-3d_e8": {
    "train_score": 0.9625333547592163,
    "test_score": 0.920199990272522,
    "train_time": 56.03071665763855
}
```



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

The highest accuracy on test data was achieved by the CNN model (0.9201).The KNeighborsClassifier and CNN model have the smallest difference between accuracy between test and training data.The worst performance was achieved by the DecisonTreeClassifier model. Its accuracy value for the test data is close to 1 which indicates too much fit to the data and makes it an average model for the given set.

The DecisonTreeClassifier and KNeighborsClassifier models have similar training times.The CNN model shows a 2-fold higher training time, while MLPClassifier shows a 10-fold



SOURCES

- 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