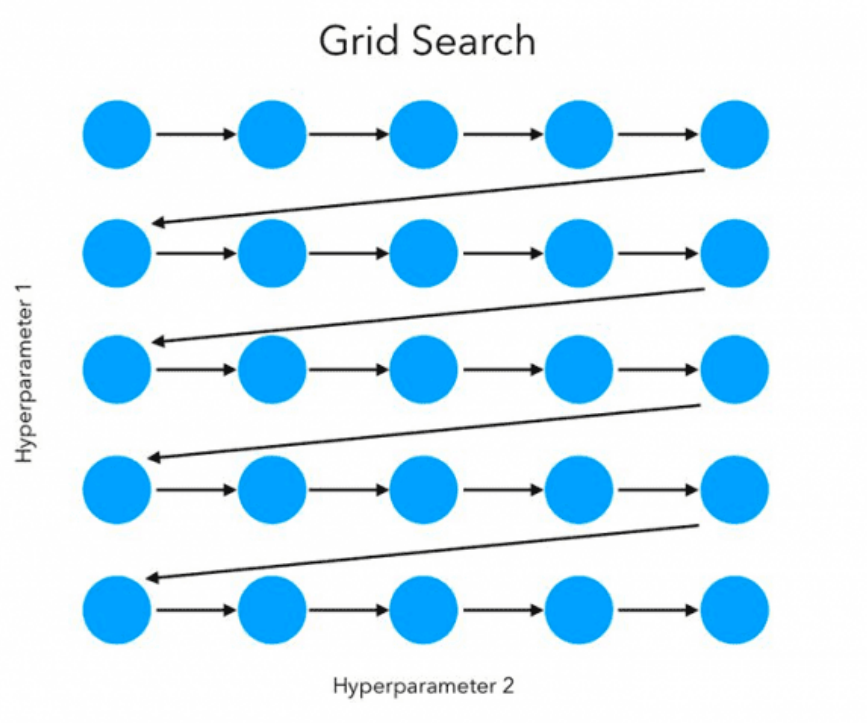
O3

SWMAL-01 Group 13



Group members
Christoffer Regnar Mølck - 202009347
Martin Teit Bruun Andersen - 202109592
Morten Kierkegaard Erichsen - 201807571

17. april 2024

1 07 CNN

1.1 What are CNNs?

Convolutional neural network's (CNN's) were invented in the 80's by Kunihiko Fukushima. His findings are inspired by Hubel and Wiesel's work on cats visual corties neurons that had small invdividual visual fields. This inspiration was the begining for feature extraction, pooling layers, recognition and classification. Kunihiko Fukushima's full article on Neocognitron can be read here: https://www.rctn.org/bruno/public/papers/Fukushima1980.pdf

Today CNN's are most commenly known for its use in image analysis. However, it is also commonly used for analysing stock prices and audio data.

Looking at image analysis, the CNN works by taking in a series of layers. Each of these layers looks at different features in the image. These features come in different sizes. One filter could be looking at vertical lines, while another could be finding eye features. However, the whole thing starts by a convulotion operation. This operation is like a sliding filter of a giving size. The size is normally a 3x3 square, however as mentioned it is user decided. The filter is called a kernel. Inside the kernel it reads the individual pixels within its square. And depending on the RPG color value it returns a number input (from 0-255) in the kernel square. Each complete kernel run over the input layer and creates a feature map. If an example were to be made on the MINST numbers, one of these created feature maps could be identifying black values in the input layer. From the convolution, each layer is pooled to a smaller layer size. Pooling layers downsample the convolution created feature maps. And on and on... However, for each convolution and pooling one has to be aware of overfitting the CNN model.

In the CNN code below, generated by Google Gemini, we have inserted a visual learning curve, that visulizes how the loss is throughout the epochs. Our CNN's input layer is a 28 by 28 by 1. Through our convolution we then take and expand the last layer, creating 32 channels. Each of these channels is then pooled through a 2 by 2 filter reducing the convolution layer from 26 by 26 by 32 to 13 by 13 by 32. Our second convolution layer then takes on our pooled 32 layers and expands it to 64 layers with a smaller size. Before the data is flatten our second convolution layer is then pooled to its last size. When the data is flatteren we see, from the model.summary(), that the value is 1600. This value represents the total amount pixels our last pool contained (5*5*64 = 1600).

The flattern data is then held on to the first dense layer of 128 feature regonizors, meaning that this dense layer has 128 neurons/categories. Afterwards this layer is then broken down to our input size (mean we have 10 different input images).

1.2 Conclusion

As a conclusion to the assignment the group tried out multiple amount of convolution layers and decided on including the one returning the best score for us. High score is commented in the bottom of the code. We found that for each convolution layer we added the worse the CNN performed leaving us with two since one was to little. We found that increasing the amount of feature maps in the first "Conv2D"would lead to increased step time when running the evalidation of the model. This makes sense due to the increased amount of features the model would look for.

1.3 Our model

```
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt

# Define the model (replace with your final CNN architecture)
model = keras.Sequential([
  # First convolutional layer
  layers.Conv2D(32, kernel_size=(3, 3), activation="relu", input_shape=(28, 28,
      ↪ 1)),
  layers.MaxPooling2D(pool_size=(2, 2)),

  # Second convolutional layer
  layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
  layers.MaxPooling2D(pool_size=(2, 2)),

  layers.Flatten(),
  layers.Dense(128, activation="relu"),
  layers.Dense(10, activation="softmax")
])

print(model.summary())

# Compile the model
model.compile(optimizer="adam", loss="categorical_crossentropy",
    ↪ metrics=["accuracy"])

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Preprocess the data
x_train = x_train.astype("float32") / 255.0
```

```
x_test = x_test.astype("float32") / 255.0
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)
y_train = keras.utils.to_categorical(y_train, num_classes=10)
y_test = keras.utils.to_categorical(y_test, num_classes=10)

# Store training history
history = model.fit(x_train, y_train, epochs=10, validation_data=(x_test,
    ↪ y_test))

# Plot the learning curve
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(loc='upper right')
plt.show()

# Evaluate the model on test data
loss, accuracy = model.evaluate(x_test, y_test)
print("Test accuracy:", accuracy)

# Highest score run 99.33
```

```
Model: "sequential_25"
```

Layer (type)	Output Shape	Param #
conv2d_66 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_66 (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_67 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_67 (MaxPooling2D)	(None, 5, 5, 64)	0
flatten_25 (Flatten)	(None, 1600)	0
dense_52 (Dense)	(None, 128)	204,928
dense_53 (Dense)	(None, 10)	1,290

```
Total params: 225,034 (879.04 KB)

Trainable params: 225,034 (879.04 KB)

Non-trainable params: 0 (0.00 B)
```

```
None
Epoch 1/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.9173 - loss: 0.2741 - val_accuracy: 0.9872 - val_loss: 0.0385
Epoch 2/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.9874 - loss: 0.0405 - val_accuracy: 0.9884 - val_loss: 0.0364
Epoch 3/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.9914 - loss: 0.0269 - val_accuracy: 0.9915 - val_loss: 0.0294
Epoch 4/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.9934 - loss: 0.0206 - val_accuracy: 0.9921 - val_loss: 0.0236
Epoch 5/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.9962 - loss: 0.0123 - val_accuracy: 0.9924 - val_loss: 0.0256
Epoch 6/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.9965 - loss: 0.0108 - val_accuracy: 0.9898 - val_loss: 0.0323
Epoch 7/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.9970 - loss: 0.0088 - val_accuracy: 0.9922 - val_loss: 0.0320
Epoch 8/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.9973 - loss: 0.0075 - val_accuracy: 0.9883 - val_loss: 0.0462
Epoch 9/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.9977 - loss: 0.0075 - val_accuracy: 0.9874 - val_loss: 0.0461
Epoch 10/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.9980 - loss: 0.0056 - val_accuracy: 0.9928 - val_loss: 0.0322
```


```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9910 - loss: 0.0416
Test accuracy: 0.9927999973297119
```

Figur 1.1: CNN model output

2 Generlization error

2.1 Qa) On Generalization Error

Training error is the error we get when we compare the training data to the model prediction, while generalization error is what we get when we compare the model to previously unseen data.

As a model can be very well fitted to data it has seen before, but poorly regularized. There are two types, under and over fitting. Under fitting, is when the model fits poorly to both the training and unseen data. While over fitting is when the model fits very well to the training data but poorly to any unseen data. Ideally the model should be general enough to fit both somewhat equally well. This point is called optimal capacity.

The generalization gap is the gap between the training error and the generalization error. Capacity is simply a term to express a models complexity or its capacity to handle complex data relations.

2.2 Qb A MSE-Epoch/Error Plot

The code performs polynomial regression using SGD, demonstrating how the model learns over time by plotting the RMSE against epochs. An epoch represents one complete pass through the entire training dataset, allowing the model to update its weights.

mse_train and mse_val refer to the mean squared error on the training and validation datasets, respectively, offering a measure of how well the model predicts compared to the actual values, with lower values indicating better performance.

The generation of the array is good for visualizing the model's learning progress, highlighting the balance between fitting the training data well (low mse_train) and generalizing effectively to new data (low mse_val). Tracking these metrics helps in identifying the best model configuration that avoids overfitting while maintaining good predictive accuracy.

The code has been changed slightly, iteration count was changed from 0 ti 1000 and the -float(inf) to 0.0 as it otherwise gave an error and did not run.

(Initial suggestion by ChatGPT, text changed significantly)

2.3 Qc) Early Stopping

I would record the current validation score, if it does not increase in the next 5 iterations i would break out of the loop. so:

```
if mse_val > mse_val_highest
  i++
  if i == 5
      break
else i = 0
```

2.4 Qd) Explain the Polynomial RMSE-Capacity plot

The x-axis shows the model complexity, indicated by the polynomial degree. The y-axis is the prediction error's standard deviation. in other words how much the data deviates from the prediction.

The training RMSE declines as model capacity increases because a higher degree polynomial can better fit the training data (it has more complexity). However, the validation RMSE decreases only to a point before it starts to increase, suggesting the model is over fitting. it's learning noise so the generalization becomes much worse.

Increasing the polynomial degree to 10 leads to higher validation RMSE, indicating more severe over fitting. The optimal model strikes a balance, fitting the training data well without capturing excessive noise, and thus generalizes better to new data.

3 Gridsearch

3.1 Qa Explain GridSearchCV

Cell to starts by loading the data using the load LoadAndSetupData function defined in cell one. This just loads a specific dataset depending on a passed in string.

The model is then instantiated and the parameters we want to grid search over are created. In this case we want to search through the kernal type, and the C values to find the best model.

the grid search is then run on the new model using the tuning parameters to tell it waht to search over.

The result is reported using the function full report that is facade that runs a few other functions to create a report on the result of the gridsearch. The report can then be used to create a new model with the found optimal parameters.

3.2 Qb Hyperparameter Grid Search using an SDG classifier

I repeat a lot of the code from Qa but using the SDG model instead, and using a few more randomly selected parameters for tuning.

```
model = SGDClassifier() #Instantiate mdoel with default params.

tuning_parameters_SGD = {
    'loss': ['hinge', 'modified_huber', 'squared_hinge', 'squared_error'],
    'penalty': ['l2', 'l1', 'elasticnet'],
    'alpha': [0.0001, 0.001, 0.01, 0.1],
    'learning_rate': ['constant', 'optimal', 'invscaling', 'adaptive'],
    'eta0': [0.0001, 0.001, 0.01, 0.1],
    'max_iter': [1000, 2000, 3000],
    'tol': [1e-3, 1e-4, 1e-5]
}

CV = 5
VERBOSE = 0

# Run GridSearchCV for the model
grid_tuned_SGD = GridSearchCV(model,
                          tuning_parameters_SGD,
```

```
                        cv=CV,
                        scoring='f1_micro',
                        verbose=VERBOSE,
                        n_jobs=-1)

start = time()
grid_tuned_SGD.fit(X_train, y_train)
t = time() - start

# Report result
b0, m0 = FullReport(grid_tuned_SGD, X_test, y_test, t)
print('OK(grid-search)')
```

this takes a few minute to run, but nothing too bad. Outputs:

```
Detailed classification report:
   The model is trained on the full development set.
   The scores are computed on the full evaluation set.

              precision recall f1-score support

           0 1.00 1.00 1.00 16
           1 0.94 0.89 0.91 18
           2 0.83 0.91 0.87 11

    accuracy 0.93 45
   macro avg 0.92 0.93 0.93 45
weighted avg 0.94 0.93 0.93 45


CTOR for best model: SGDClassifier(alpha=0.1, eta0=0.0001, loss='modified_huber', max_iter=3000,
              penalty='l1', tol=0.0001)

best: dat=iris, score=1.00000,
    ↪ model=SGDClassifier(alpha=0.1,eta0=0.0001,learning_rate='optimal',loss='modified_huber',
    ↪ max_iter=3000,penalty='l1',tol=0.0001)
```

3.3 Qc Hyperparameter Random Search using an SDG classifier

So the randomsearch essentialy uses a grid as well, however it just picks random points on that grid and test them. n_iter defines how many tests it runs. so an n_iter of 20 runs 20 tests and it returns the best one of those 20. This is obviously less accurate then gridsearch that brute forces every combination, however in my small test here, the scores ended up being the same as the hyper parameters did not have a significant effect on the scoring. So in this case, a shorter more random search works as well as grid search does. This is however rarely true when applied to real data with more complex models.

3.4 Qd MNIST Search Quest II

We have chosen to use HistGradientBoostingClassifier as our model, as it can be very accurate within its dataset and is well suited for training on large datasets.

This type of model which uses boosted trees also does not need extra data preprocessing which is convenient. This type of model however performs very poorly outside the extremes of its training data.

We started running a girdsearch, but due to the complexity of the dataset it took far to long to complete. So we changed the search to random with 20 iterations hoping that would be sufficient to find a good model.

sadly we are running out of time and had to do only a single iteration as running the boosted trees takes quite a while. I am sure we could have gotten a much better result with some tweaking. Still a score of 0.95 is quite good for a first try.

we ended up with a score:

```
best: dat=mnist, score=0.95196,
    ↪ model=HistGradientBoostingClassifier(l2_regularization=0.1,learning_rate=0.01,
    ↪ loss='log_loss',max_bins=255,max_iter=200,max_leaf_nodes=31)
```

4 Regulizers

4.1 Qa) The Penalty Factor

Code:

```
def Omega(w):
return np.linalg.norm(w[1:])**2
```

Explanation: The penalty factor is low when the weights are small (numerically), and large when the weights are large aswell (also numerically).

4.2 Qb) Explain the Ridge Plot

Explanation The α in the 'Ridge' model is a constant that multiplies the objective function:

$$||y - Xw||_2^2 + \alpha \cdot ||w||_2^2$$

in an attempt to minimize it. In the case of $\alpha = 0$, it looks as if there's a lot of overfitting, since the model being used in linear. The same with $\alpha = 1e - 5$, a bit less overfitting, but still significant. With $\alpha = 1$, however, the fit looks to be underfitted, since J actually getz penalized by the entire $||w||_2^2$ value. To determine whether or not $\alpha = 1$ indeed is a good fit, one would have to check the model score.

With $\alpha = 0$, the fitting is the same as the regular least squares method, sine the penalty term simply becomes 0. The SKLearn actually advises not to use the Ridge function with $\alpha = 0$, for "Numerical reasons". Source: .

4.3 Qc Explain the Ridge, Lasso and ElasticNet Regularization Methods

Explanation

First, looking at the formulas, from the Scikit-learn documentation:

Ridge:

$$||y - Xw||_2^2 + \alpha \cdot ||w||_2^2$$

Lasso:

$$(1/(2 \cdot n_{samples})) \cdot ||y - Xw||_2^2 + \alpha \cdot ||w||_1$$

ElasticNet:

$$1/(2n_{samples}) \cdot ||y - Xw||_2^2 + alpha \cdot l1_{ratio} \cdot ||w||_1 + 0\ 5 \cdot \alpha \cdot (1 - l1_{ratio}) \cdot ||w||_2^2$$

It is quite apparant that the main difference are as follows:

(note: coefficients = weights)

Ridge uses the L2 norm, and no constant depending on the number of samples

Lasso uses the L1 norm, and is dependant on the number of samples

Elasticnet is also dependant on the number of samples, while using both the L2 and L1 norms. Furthermore, it specifies a(n) $L1_{ratio}$, that gives the norm L2 and L1 their own form of weight. It is also clear, that if one sets the $L1_{ratio} = 1$, one would get the Lasso regulizer.

The α value in Ridge controls how much the coefficients "shrink", with a larger α shrinking the coefficients closer to zero. With a larger α, the model becomes better at handling co-linearity, since the coefficients become smaller and smaller.

Lasso estimates "sparse coefficients", which are coefficients that are non-zero, which is useful since that reduces the number of features on which a solution is dependant. In other words, it "prefers"solutions with fewer non-zero coefficients, meaning that under some circumstances, it will be able to return the exact values of the coefficients on which the model is dependant.

Being dependant on both L2 and L1, ElasticNet allows for a combination-ish of Lasso and Ridge, since it will then include some of the sparse, non-zero coefficients, while also maintaining the regularization properties of Ridge (second term in the formula).

4.4 Qd) Regularization and Overfitting

Regularization is useful in regards to reducing a models tendency to overfit, since the model gets penalized by some value, making it impossible for the model to fit too closely to the data. If theres no penalty, the model wont be able to generalize for unseen data, resulting in overfitting.

the tug-of-war is kind of a balancing act between the MSE and the regularization Ω. The MSE aims to minimize the error between what the model predicts, and what the actual target values are, pushing for a better fit of the training data. The regularization term Ω "wants"to keep the model parameters small, putting shackles on the model, preventing it from overfitting (and reduces its complexity).

When $\alpha = 1$ the regularization is much stronger, which can be beneficial for small training sets, since fewer data points means less certainty (large data good, small data bad, ish). But, as we saw in the Ridge model in Qa, with $\alpha = 1$, the model doesnt really capture any behaviour in the middle section of the data, making it underfitted. Therefore its important to tune the α value carefully.

Optional answer: Scaling would be of help, since scaling the data down to some range, eg. [-1:1], typically helps the model converge quicker, preventing the regularization term from being dominated by larger scales, eg [-1e5:1e5]. By standardizing the data, $\mu = 0$ and $\sigma = 1$, the data becomes more comparable, and as with scaling, prevents the regularization term from being dominated by larger values. By normalizing the data, each data-point would contribute equally to the regularization term, meaning no outliers (eg. faulty data) would contribute more heavily when the model converges.

4.5 Qe) Regularization Methods for Neural Networks

The explanation part is covered in the previous questions.

The following code is written by ChatGPT, which loads a housing dataset from Boston, and trains a model using Ridge, Lasso, ElasticNet and no regularization:

Code:

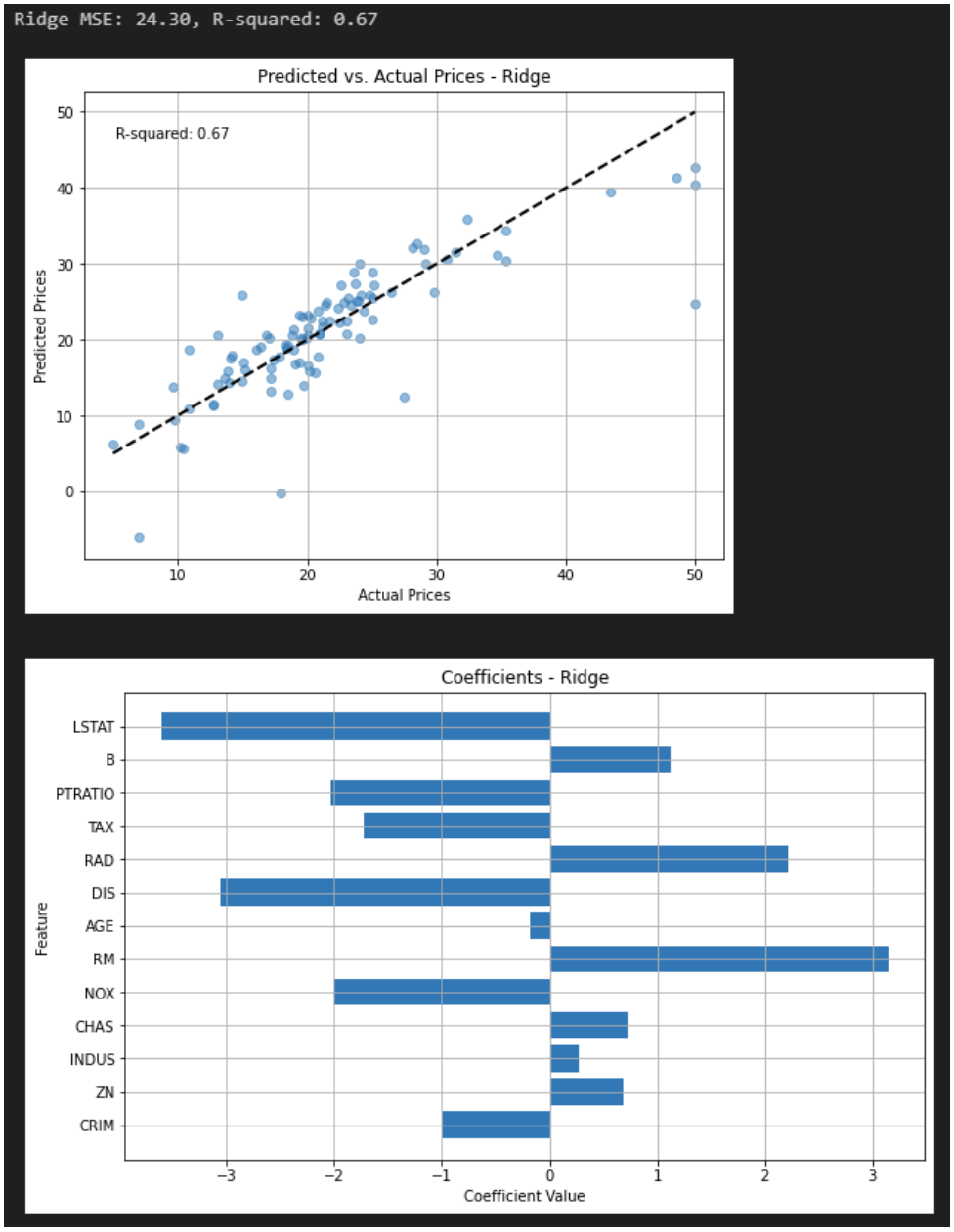
```
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.neural_network import MLPRegressor
from sklearn.linear_model import Ridge, Lasso, ElasticNet
from sklearn.metrics import mean_squared_error

# Load the Boston Housing dataset
data = load_boston()
X, y = data.data, data.target

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    ↪ random_state=42)

# Define pipelines for preprocessing and model training
pipelines = {
    "Ridge": Pipeline([
        ("scaler", StandardScaler()),
        ("regressor", Ridge(alpha=0.5)) # Ridge (L2) regularization
    ]),
    "Lasso": Pipeline([
        ("scaler", StandardScaler()),
        ("regressor", Lasso(alpha=0.5)) # Lasso (L1) regularization
    ]),
    "ElasticNet": Pipeline([
        ("scaler", StandardScaler()),
        ("regressor", ElasticNet(alpha=0.5, l1_ratio=0.5)) # ElasticNet
            ↪ regularization
    ]),
    "No Regularization": Pipeline([
        ("scaler", StandardScaler()),
        ("regressor", MLPRegressor(hidden_layer_sizes=(100,), activation='relu',
            ↪ solver='adam', max_iter=1000)) # Neural network without
            ↪ regularization
    ])
```

```
}

import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import r2_score

# Define function to plot predicted vs. actual prices
def plot_predicted_vs_actual(y_test, y_pred, title, r2_score):
    plt.figure(figsize=(8, 6))
    plt.scatter(y_test, y_pred, alpha=0.5)
    plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'k--',
        ↪ lw=2)
    plt.xlabel('Actual Prices')
    plt.ylabel('Predicted Prices')
    plt.title(title)
    plt.text(0.05, 0.9, f'R-squared: {r2_score:.2f}',
        ↪ transform=plt.gca().transAxes)
    plt.grid(True)
    plt.show()

# Define function to plot coefficients of regularized models
def plot_coefficients(coefficients, feature_names, title):
    plt.figure(figsize=(10, 6))
    plt.barh(np.arange(len(feature_names)), coefficients,
        ↪ tick_label=feature_names)
    plt.xlabel('Coefficient Value')
    plt.ylabel('Feature')
    plt.title(title)
    plt.grid(True)
    plt.show()

# Train and evaluate models
for name, pipeline in pipelines.items():
    pipeline.fit(X_train, y_train)
    y_pred = pipeline.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)
    print(f"{name} MSE: {mse:.2f}, R-squared: {r2:.2f}")

    # Plot predicted vs. actual prices
    plot_predicted_vs_actual(y_test, y_pred, f"Predicted vs. Actual Prices -
        ↪ {name}", r2)

    # If it's a regularized model, plot coefficients
    if name != "No Regularization":
        feature_names = data.feature_names
        if hasattr(pipeline.named_steps['regressor'], 'coef_'):
```

```
            coefficients = pipeline.named_steps['regressor'].coef_
        elif hasattr(pipeline.named_steps['regressor'], 'feature_importances_'):
            coefficients = pipeline.named_steps['regressor'].feature_importances_
        plot_coefficients(coefficients, feature_names, f"Coefficients - {name}")
```

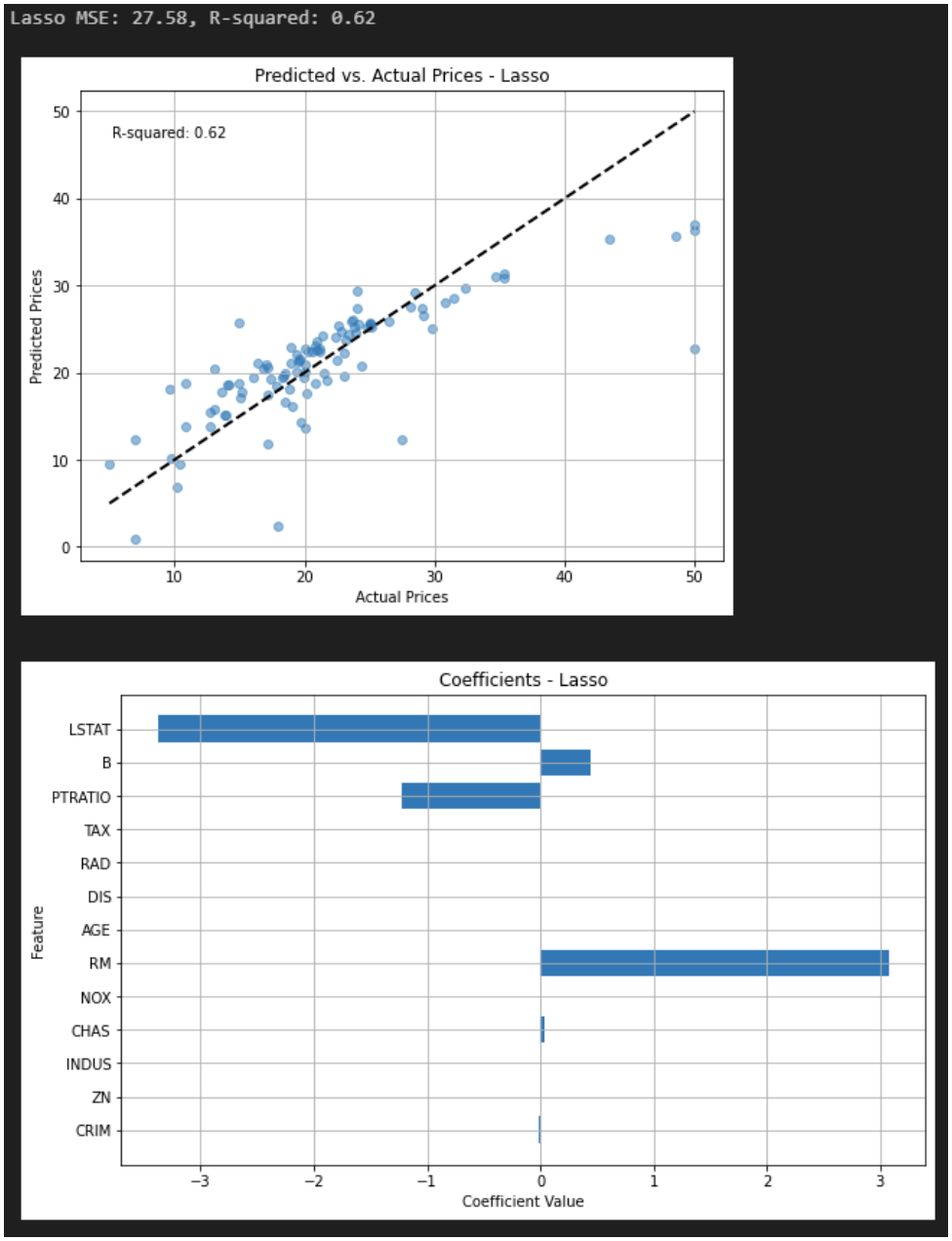
Which outputs different plots with R^2 and MSE values, as well as a graphic representation of the values of the different coefficients:

4.5.1 Ridge model



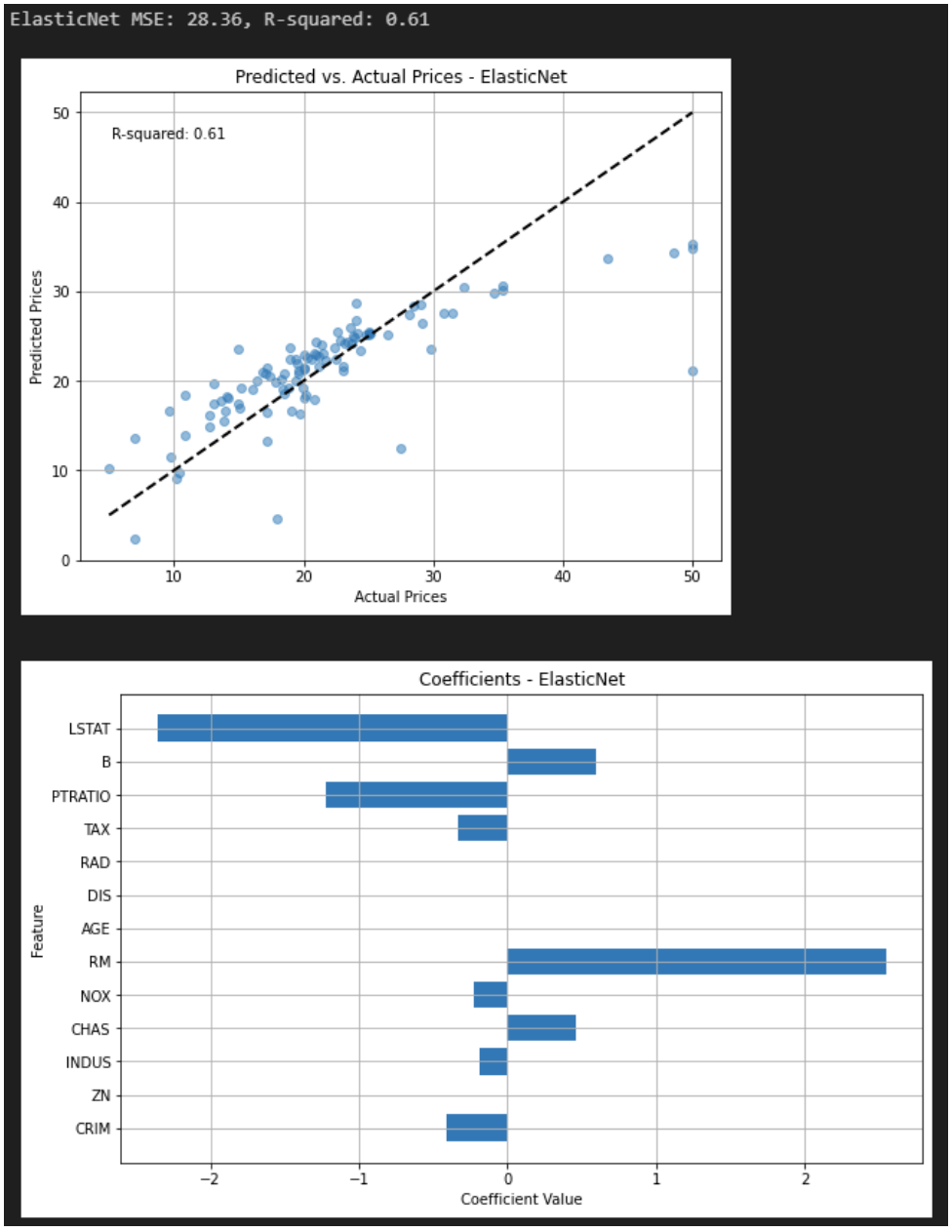
Figur 4.1: Ridge model plots

4.5.2 Lasso model



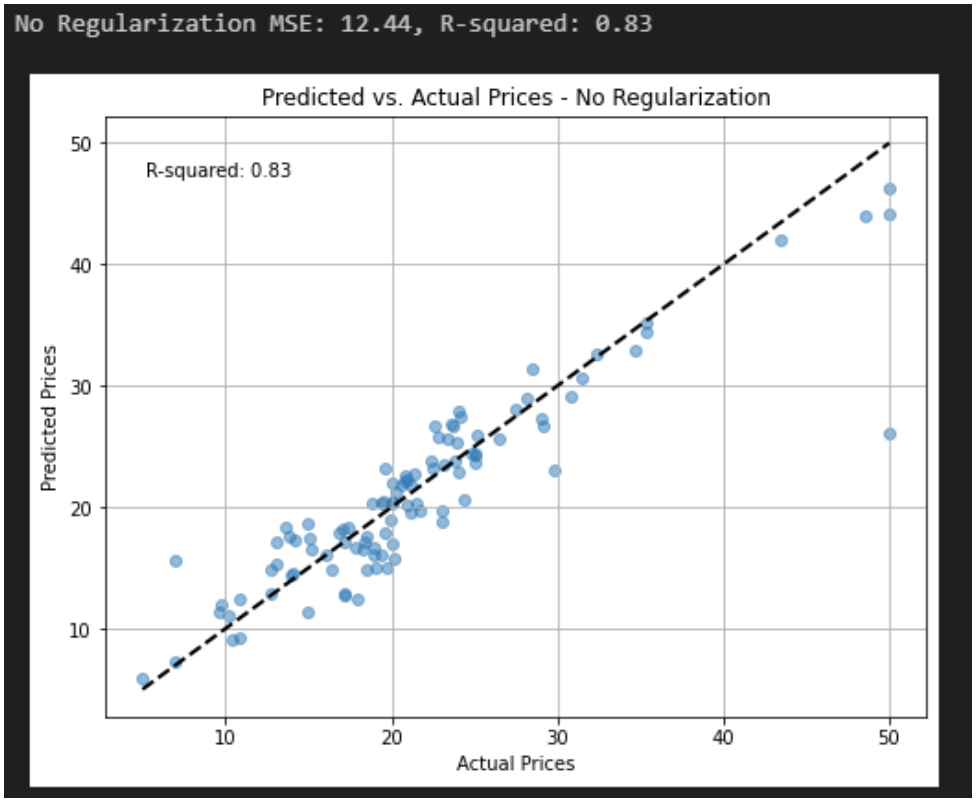
Figur 4.2: Lasso model plots

4.5.3 ElasticNet



Figur 4.3: ElasticNet model plots

4.5.4 No regularization, regular MLPRegressor



Figur 4.4: Regular MLP regression

Explanation: We see that with the same value of $\alpha = 0\,5$, the models vary differently - mostly Ridge when comparing the regularization methods. This does make sense, since it punishes the least, compared to Lasso and ElasticNet (also apparent in its lower MSE value). The regular MLPRegressor also shows the highest R^2 value, which it also should, since there is no punishment included in that model.