Yiğit Yalın
22002178
Group 8
29 November 2022

EEE 485 - Final Report

Diamond Price Estimator

Introduction

Diamond is a valuable mineral with an active market. According to OEC, “diamonds were the world's 33rd most trading product with a total trade of $72.4B in 2020” [1]. Due to the activity of the market, the diamond price estimation becomes a need, and it might not always be possible to consult an expert. This project aims to cover the need for an expert for the diamond price estimation using a dataset of diamond prices with some of their associated exterior properties.

Task

The main objective of this project is to create three models to predict the diamond price given the exterior properties of the diamonds using three different algorithms. This is a regression problem and the proposed algorithms for the solution of this task are ridge regression, neural networks and support vector regression. Further discussion about the algorithms will be in the next sections of this report.

Dataset

The dataset was prepared by Greg Mills (MBA ’07) under the supervision of Phillip E. Pfeifer, Alumni Research Professor of Business Administration. The original source of the dataset was deleted from GitHub. However, the PyCaret repository [2] on GitHub contains a copy of this dataset. The dataset is in .csv format which is loaded easily using the Pandas package in Python. The dataset contains 6,000 instances. The features are the carat weight, cut, color, clarity, polish, symmetry and the grading agency.

	Carat Weight	Cut	Color	Clarity	Polish	Symmetry	Report	Price
2	1.1	Ideal	H	SI1	VG	EX	GIA	5169
3	0.83	Ideal	H	VS1	ID	ID	AGSL	3470
4	0.85	Ideal	H	SI1	EX	EX	GIA	3183
5	0.91	Ideal	E	SI1	VG	VG	GIA	4370
6	0.83	Ideal	G	SI1	EX	EX	GIA	3171
7	1.53	Ideal	E	SI1	ID	ID	AGSL	12791
8	1	Very Good	D	SI1	VG	G	GIA	5747
9	1.5	Fair	F	SI1	VG	VG	GIA	10450
10	2.11	Ideal	H	SI1	VG	VG	GIA	18609

Figure 1: First 10 Rows of the Dataset

Programming Environment

The main programming language of this project is Python. The Python version is 3.10. PyCharm is used as the IDE since the IDE makes it easier for the developers to form a project structure. Git is the main tool for the version control and all the project code is held in a repository on Github.

Two Python libraries are used extensively in this project. These libraries are Numpy and Pandas. The Numpy library is used for efficient matrix operations, and is the core library of this project since it provides all the functionality to implement ridge regression, neural networks and support vector regression. The Pandas library is the main tool in this project to preprocess the data. In addition to these libraries, the Matplotlib library is used for visualization. Also, the standard library of Python is extensively used, especially for type hints and creating abstract base classes.

Jupyter notebooks provide a nice user interface for IPython and are used to test the implementations. The data loading, data preprocessing, visualization and model training are done on Jupyter notebooks since it allows the users to run the code cell by cell.

Data Preprocessing

The data consist of seven features. The Carat Weight feature is the weight of the diamond in metric carats. The Cut, Color, Clarity, Polish, Symmetry features indicate the desirability of the cut, color, clarity and polish of the diamond, respectively. The Report feature indicates the grading agency that reports the quality of the diamonds. Among these features, the Carat Weight feature is the only numerical one. Since the ridge regression is affected by the scale of the variables, this feature is normalized. Let x_{ij} denote the jth feature of the ith data row, N denote the total number of data rows, μ_j denote the mean of the jth feature, σ_j denote the standard deviation of the jth feature, and z_{ij} denote the normalized version of x_{ij}. Then, the normalization is done as follows:

$$\mu_j = \frac{\sum_{i=1}^{N} x_{ij}}{N}$$

$$\sigma_j = \sqrt{\frac{\sum_{i=1}^{N} \left(x_{ij} - \mu_j \right)^2}{N}}$$

$$z_{ij} = \frac{x_{ij} - \mu_j}{\sigma_j}$$

By applying this transformation to the Carat Weight feature, we obtain a feature vector as shown in the figure below.

	Carat Weight
0	-0.493004
1	-1.060593
2	-1.018549
3	-0.892418
4	-1.060593
5	0.410934
6	-0.703222
7	0.347869
9	-0.598113

Figure 2: The Carat Weight Feature for the First Ten Rows of the Dataset

The other six features are categorical, and these features are one-hot encoded. Since one-hot encoding increases the number of dimensions of the feature space a lot, one column for each feature is dropped since it does not reduce the information. After applying one-hot encoding to the dataset, we obtain a new feature set consisting of 22 sparse binary features and 1 normalized numerical feature. The first 10 rows of the first 10 features of the preprocessed DataFrame are shown in the figure below.

	Carat Weight	Cut_Good	Cut_Ideal	Cut_Signature-Ideal	Cut_Very Good	Color_E	Color_F	Color_G	Color_H	Color_I
0	-0.493004	0	1	0	0	0	0	0	1	0
1	-1.060593	0	1	0	0	0	0	0	1	0
2	-1.018549	0	1	0	0	0	0	0	1	0
3	-0.892418	0	1	0	0	1	0	0	0	0
4	-1.060593	0	1	0	0	0	0	1	0	0
5	0.410934	0	1	0	0	1	0	0	0	0
6	-0.703222	0	0	0	1	0	0	0	0	0
7	0.347869	0	0	0	0	0	1	0	0	0
9	-0.598113	0	0	0	1	1	0	0	0	0

Figure 3: The First 10 Rows and Columns of the Preprocessed Dataset

Detailed Description of the Proposed Methods

The three algorithms proposed for the solution of the task are ridge regression, neural networks and support vector regression. The following sections will discuss each one of these algorithms in detail.

Ridge Regression

The first method that has been implemented is ridge regression. Ridge regression is a regularized linear regression. Let X denote the feature matrix, w denote the weight vector of the model, b denote the bias of the model, and λ denote the regularization coefficient. Then, the model and the loss function of the ridge regression are given as follows:

$$y = Xw + b$$

$$L_{ridge} = (y - Xw)^T (y - Xw) + \lambda w^T w$$

This loss function differs from the loss function of linear regression in that there is a regularization term in this function. In the implementation, setting λ to zero is allowed to obtain unregularized results. There is a closed form solution to this equation. The equation and its derivation is as follows:

$$\frac{\partial L_{ridge}}{w} = 0 \Rightarrow -2y^T X + 2w^T X^T X + 2\lambda w^T = 0$$

$$\Rightarrow w = (X^T X + \lambda I)^{-1} X^T y$$

Though $X^T X + \lambda I$ is always invertible, the closed form solution may not be the best option to use to determine the model parameters. This is mostly because of the costly nature of matrix operations. Taking the inverse of a matrix is computationally expensive, and this makes determining the weights via this equation impossible in limited time for huge datasets. Instead, the gradient descent algorithm will be used to minimize this loss function. The gradient descent algorithm uses the first derivatives to converge to a local minima of the loss function. The gradients of this loss function with respect to the weight and bias is given as follows:

$$\nabla_w L_{ridge} = \frac{2X^T(Xw + b - Y) + 2\lambda w}{N}$$

$$\nabla_b L_{ridge} = \frac{2}{N} \sum X^T(Xw + b - Y)$$

Given this equations and the learning rate α, which is the size of the gradient descent steps, the update equations are given as follows:

$$w \leftarrow w - \alpha \nabla_w L_{ridge}$$

$$b \leftarrow b - \alpha \nabla_b L_{ridge}$$

This linear algorithm is chosen because of the simplistic nature of the dataset. However, even though linear models are rather simple models which cannot capture complex nonlinear models, there is a chance to overfit if the data follows even a simpler model and it has linear correlation among its features.

Neural Network

The second method is neural networks. This algorithm is chosen since it is capable of fitting very complex functions. The neural networks are composed of layers, where the number of layers and the number of neurons in each layer are hyperparameters that require some effort to tune. After each of these layers, nonlinear activation functions are applied to the layer outputs to increase the generalizing capabilities of the network. By stacking many layers with many neurons, neural networks can fit very complex functions. Though this capability of fitting very complex functions is very desirable, this makes the algorithm very hard to optimize. In this project, the algorithm to optimize the neural network weights is stochastic gradient descent (SGD) as it is the common practice to use SGD and its sophisticated variants to train neural networks. The update equations for the neural network weights are identical to the

ones of the ridge regression, where the gradients are applied throughout the network starting from the deep layers to the input layer iteratively. The algorithm to find the gradients for hidden layer weights is called backpropagation, and is an application of the chain rule from calculus. For the diamond price prediction task, the hidden layers activations is chosen as the Rectified Linear Units (ReLU) to introduce nonlinearity and the output layer activation is chosen as linear, which is actually the identity function $f(x) = x$, since this is a regression task.

Neural networks are chosen to cover the case that the data has some complex patterns to capture and they are able to capture more complex patterns compared to ridge regression and support vector regressors.

Support Vector Regression

The third method that will be implemented in this project is support vector regression. This algorithm is chosen because it is a very efficient algorithm along with its capability of fitting nonlinear functions via kernel trick. SVM tries to find a linear function

$$f(x) = x^T\beta + b$$

and ensure that it is as flat as possible [3]. This is formulated as a convex optimization problem to minimize:

$$J(\beta) = \frac{1}{2}\beta^T\beta$$

subject to all residuals having a value less than ϵ [3]. It is possible that there is no function that satisfies this constraint. Slack variables ξ_i, ξ_i^* are introduced to deal with this problem. Introducing slack variables to the objective function gives the primal formula which is given as follows:

$$J(\beta) = \frac{1}{2}\beta^T\beta + C\sum_{i=1}^{N}\left(\xi_i + \xi_i^*\right)$$

$$\textit{subject to}:$$

$$\forall i : y_i - \left(x_i^T\beta + b\right) \leq \varepsilon + \xi_i$$

$$\forall i: \left(x_i^T\beta + b\right) - y_i \geq \varepsilon + \xi_i^*$$

$$\forall i : \xi_i^* \geq 0$$

$$\forall i: \xi_i \geq 0$$

which leads to the linear ϵ insensitive loss [3]. The linear insensitive loss is given by the formula

$$L_\varepsilon = \begin{cases} 0, & |y - f(x)| \leq \varepsilon \\ |y - f(x)| - \varepsilon & |y - f(x)| > \varepsilon \end{cases}$$

This optimization problem is computationally simpler in its Lagrange dual formulation [3]. Constructing a Lagrangian function from the primal function by introducing multipliers for each observation x_i leads to the dual formula:

$$L(\alpha) = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\left(\alpha_i - \alpha_i^*\right)\left(\alpha_j - \alpha_j^*\right)x_i^T x_j + \varepsilon\sum_{i=1}^{N}\left(\alpha_i + \alpha_i^*\right) + \sum_{i=1}^{N} y_i\left(\alpha_i^* - \alpha_i\right)$$

subject to the following constraints [3]:

$$\sum_{i=1}^{N}\left(\alpha_i - \alpha_i^*\right) = 0$$

$$\forall i: 0 \le \alpha_i \le C$$

$$\forall i: 0 \le \alpha_i^* \le C$$

The Karush-Kuhn-Tucker (KKT) complementarity conditions are optimization constraints required to obtain optimal solutions [3]. The optimal solution is obtained through Sequential Minimal Optimization algorithm [4]. SMO algorithm iteratively updates the α's until the entire training set obeys the KKT conditions within the threshold ϵ, which is typically set to 10^{-3} [4].

The β parameter can be completely described as a linear combination of the training observations using the following equation [3]:

$$\beta = \sum\left(\alpha_i - \alpha_i^*\right)\left(x_i^T x_i\right) + b$$

This algorithm is selected because the dataset contains many categorical features which will be one-hot encoded. One-hot encoding generates many new features and leads to a very sparse dataset. The SMO algorithm, which is explained above briefly, works fast and leads to accurate results on sparse datasets [4]. Also, using Kernel trick, a support vector regressor can capture nonlinear relationships. For this project, the support vector regression algorithm is implemented and tested with linear and radial basis function kernels.

Final Results & Discussion

Support vector regression algorithm is also implemented in addition to the neural networks and ridge regression, which were implemented before the first report. The first version of neural networks were modified. I added training with the momentum option to the neural networks. Also, the preliminary results were removed because they lacked hyperparameter tuning. Now the parameters are tuned on a grid using 5-fold cross validation. The values obtained are averaged of the 5-fold training and validation metrics. The 5-fold cross validation is done after 10% of the whole dataset is splitted for the test dataset.

Ridge Regression Results & Discussion

For ridge regression, there are three important hyperparameters to tune which are the learning rate, the regularization coefficient and the batch size. The learning rate takes values from the set {0.01, 0.005, 0.001, 0.0005}, the regularization coefficient takes values from the set {0.01, 0.001, 0.0001, 0}, and the batch size takes values from the set {32, 64}. The hyperparameter grid is the cartesian product of these three sets. The histogram of the models based on their MSE metric on the validation set is as follows:

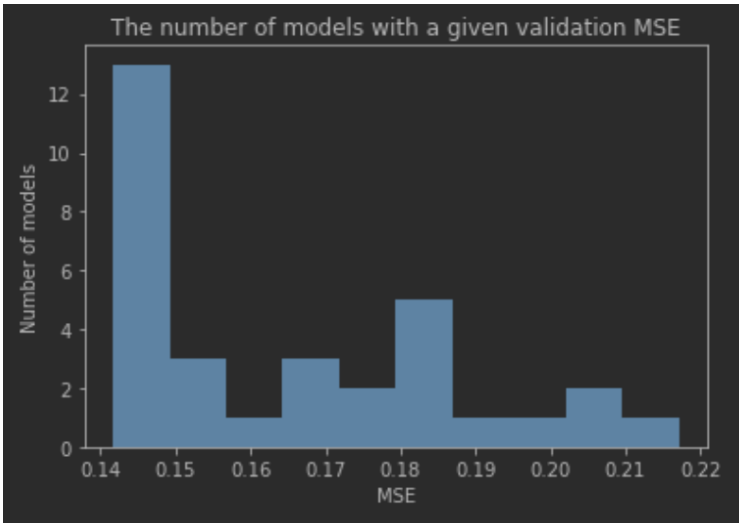


Figure 4: The histogram of the ridge regression models based on their MSE metric

The average of the MSE scores over all of the models trained is approximately 0.165. For the lower learning rates in the hyperparameter space (0.001, 0.0005), the models are converging too slowly, and thus performing worse than the models trained with higher learning rates (0.01, 0.005). The average of the MSE scores over all of the models trained using a learning rate of 0.001 or 0.0005 is approximately 0.184, whereas the ones trained with higher learning rates have an average MSE of 0.146. However, this kind of a relation does not exist between the regularization coefficient and the MSE metric. The average MSE value over the models trained with the two higher regularization coefficients is approximately the same as the models trained with the two lower regularization coefficients. Nevertheless, the best performing hyperparameters with respect to the validation MAPE metric have the following hyperparameters:

- Learning rate: 0.01
- Regularization coefficient: 0.01
- Batch size: 64

The model with these hyperparameters have the following metric scores:

```
train_MSE 0.16690228680482844 valid_MSE 0.16826969201363326
train_MAE 0.25528346168372146 valid_MAE 0.2536348429874038
train_MAPE 2.088003390918025  valid_MAPE 1.50715105022763
train_R2 0.8542625335475865   valid_R2 0.833064866522053
```

Figure 5: The train and validation scores of the best-performing ridge regression model

It might also be worth noting that models trained with mini batches of size 64 perform slightly worse compared to the models trained with mini batches of size 32 in terms of the MSE metric.

The following figure shows the train set and test set plots versus epoch numbers:

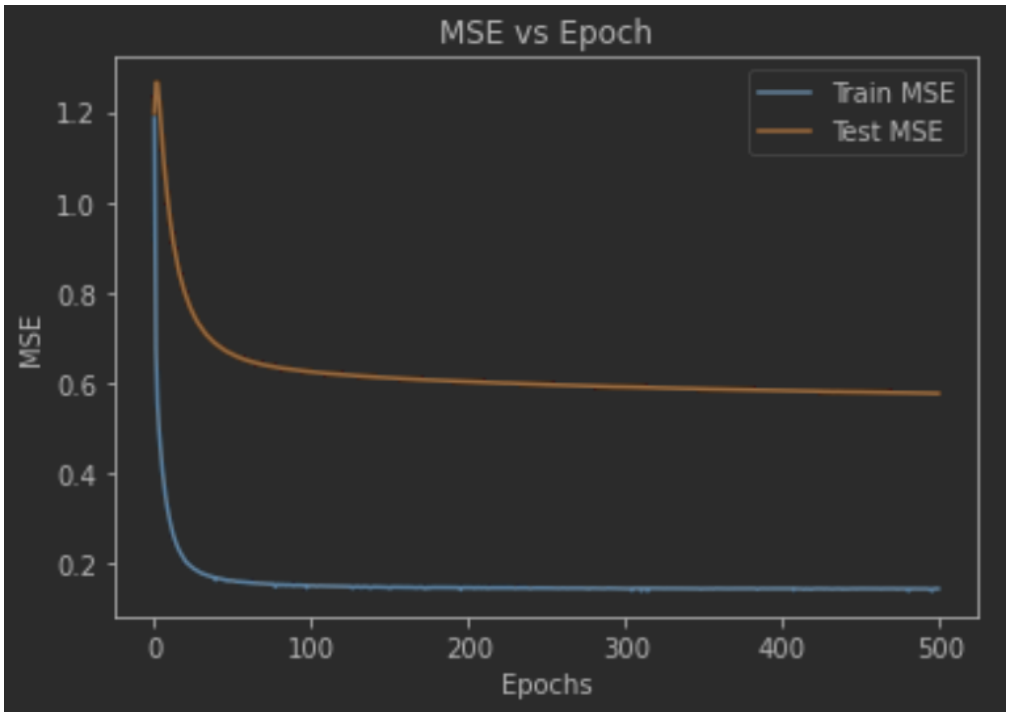


Figure 6: The training and test MSE versus epochs for ridge regression

We see that the test error is much higher than the training error. This might be an indicator of overfitting and can be tuning the regularization coefficient. Larger regularization coefficients decrease the model complexity, and thus, might solve the overfitting problem.

Neural Networks Results

For the hyperparameter tuning of neural networks, there are four important hyperparameters. These are the number of neurons, the learning rate, the momentum and the batch size. The number of neurons take values from the set {[32, 1], [64, 1], [32, 32, 1], [64, 64, 1]}, where the numbers in the lists denote the number of neurons in each layer of an MLP, the learning rate takes values from the set {0.01, 0.001, 0.0001, 0}, the momentum takes values from the set {0.85, 0.95}, and the batch size takes values from the set {32, 64}. The histogram of the models based on their MSE metric on the validation set is as follows:

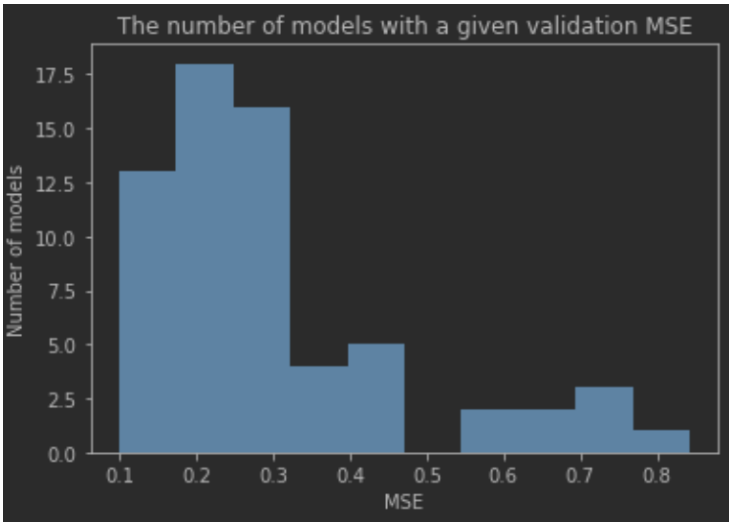


Figure 7: The histogram of the neural network models based on their MSE metric

We can observe that most of the ridge regression models had mean squared error of 0.300, approximately. Also, the standard deviation of the validation MSEs of all the models trained are 0.17. Recall that this number was 0.02 for the ridge regression. We can deduce that the neural network performance is much more dependent on the hyperparameters compared to the ridge regression as the error varies so much. The best performing neural network model with respect to the MAPE is trained using the following hyperparameters:

- The number of neurons: [64, 64, 1]
- Learning rate: 0.01
- Momentum: 0.85
- Batch size: 32

The model with these hyperparameters have the following metric scores:

```
train_MSE 0.09611180463463619  valid_MSE 0.09839532513156464
train_MAE 0.1643162377818863   valid_MAE 0.16349986456628834
train_MAPE 1.271877502520959   valid_MAPE 0.8987030178740392
train_R2 0.9322010903377901    valid_R2 0.9034182603120198
```

Figure 8: The train and validation scores of the best-performing neural network model

Though the train and validation accuracies are close to each other, it is suspicious that a neural network with two layers with 64 neurons each is not overfitting to a relatively simple dataset. Actually, when we retrain a neural network with the parameters above and plot the MSE loss with respect to the epoch number, we see the following:

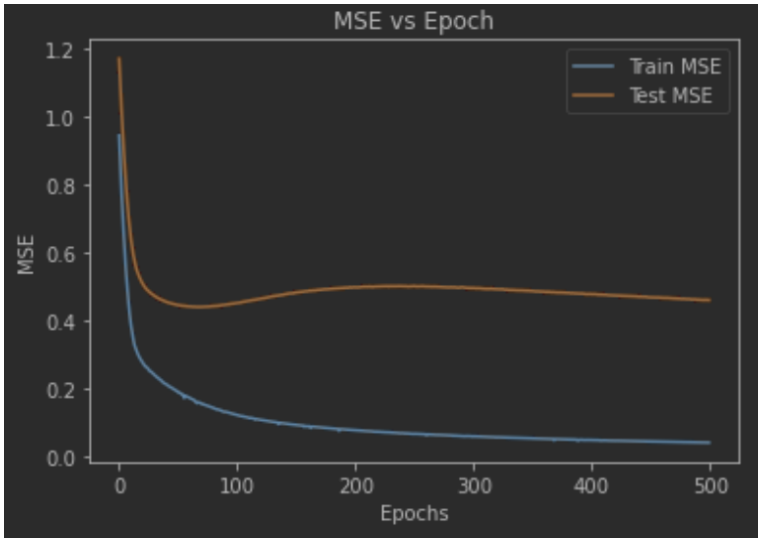


Figure 9: The training and test MSE versus epochs for neural network

We observe that the train MSE is much lower than the test MSE. We can deduce that the model does overfit the training data and cannot generalize well on the test samples. Regularization methods like l2 regularization of dropout might help the model to capture the patterns in the data instead of the noise.

Support Vector Regressor Results

For the hyperparameter tuning of the support vector regression algorithm, the grid is defined differently than the others. The following figure shows the hyperparameter combinations that are used to train models, where the values in each tuple in order correspond to C, epsilon, tolerance, kernel_type, gamma. The gamma value is the parameter of the radial basis function and is ignored if the kernel_type is linear.

```
hyperparameters = [
    (0.1, 0.001, 0.01, 'linear', 0.5),
    (0.1, 0.001, 0.01, 'rbf', 0.5),
    (0.1, 0.001, 0.001, 'linear', 0.5),
    (0.1, 0.001, 0.001, 'rbf', 0.5),
    (0.1, 0.001, 0.001, 'rbf', 1),
    (0.1, 0.01, 0.01, 'linear', 0.5),
    (0.1, 0.01, 0.001, 'rbf', 0.5),
    (0.2, 0.001, 0.001, 'rbf', 0.5),
    (0.2, 0.001, 0.001, 'rbf', 1),
    (0.2, 0.01, 0.001, 'linear', 1),
    (0.2, 0.01, 0.001, 'rbf', 1),
]
```

Figure 10: The hyperparameter grid that is searched for hyperparameter tuning. Left to right: C, epsilon, tolerance, kernel_type, gamma. The last value is ignored if the kernel type is linear.

The reason for restricting the hyperparameter space is that the support vector regressor class iterations take too much time. The histogram of the models based on their MSE metric on the validation set is as follows:

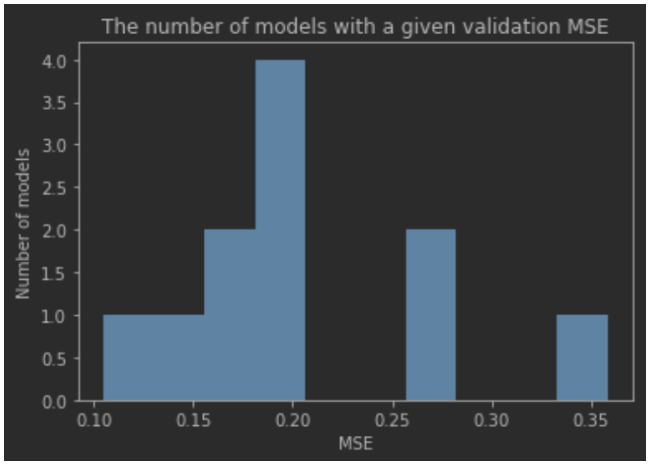


Figure 11: The histogram of the support vector regression models based on their MSE metric

The best performing neural network model with respect to the MAPE metric is trained using the following hyperparameters:

- C: 0.1
- epsilon: 0.001
- tolerance: 0.01
- kernel_type: ‘rbf’
- gamma: 0.5

```
train_MSE 0.14276249038194772   valid_MSE 0.15583602384769502
train_MAE 0.11432254888429307   valid_MAE 0.13050372816277242
train_MAPE 0.33210614648742265  valid_MAPE 0.4254734949820529
train_R2 0.8854287307363491     valid_R2 0.8775522863972913
```

Figure 12: The train and validation scores of the best-performing support vector regression model

Encountered Challenges:

All of the three regression algorithms proposed are not implemented along with their hyperparameter tuning. The hardest obstacle was to implement the support vector regressor and the SMO, which is the training algorithm of the SVO. Another hard part of the project was to implement a neural network with an arbitrary number of layers. The backpropagation of the gradient throughout the layers requires much attention since any error in this step either causes an error, which is probably a dimension mismatch in matrix multiplication, or a silent bug, which cannot be detected by the compiler/interpreter but diminishes the model performance a lot. Another challenge during the project was that the SVR takes so long, and

thus, the hyperparameter tuning times is longer compared to the other models. One of the most challenging parts in this project was the overfitting. Most of the models overfitted to some extent, and required more hyperparameter tuning. However, since hyperparameter tuning with k-fold cross validation requires k times the total number of hyperparameter combinations models trained, it becomes so hard to deal with.

Gantt Chart

The following Gantt chart was prepared to manage the project.

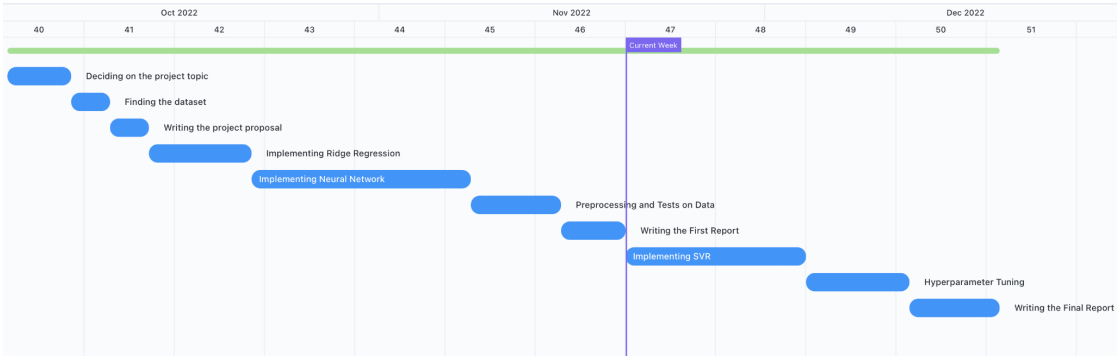


Figure 13: Gantt Chart that shows the project workflow

Conclusion

The main objective of this project was to create a model for diamond price prediction. We achieved some promising results. We obtained many models that attained decent scores in terms of MSE, MAPE, MAE, and R2. However, these models need some refinement. For example, adding regularization to the neural network would increase the performance. Also, separation of the test set should be more clever. It should take the target values into account so that both the test and train split contain similar densities of the target values. Apart from these, the solutions here provide nice baselines and implementations of strong models that can capture complex patterns, such as neural networks and SVR with radial basis function kernels. These models can provide accurate predictions after more hyperparameter tuning and more training epochs.

References

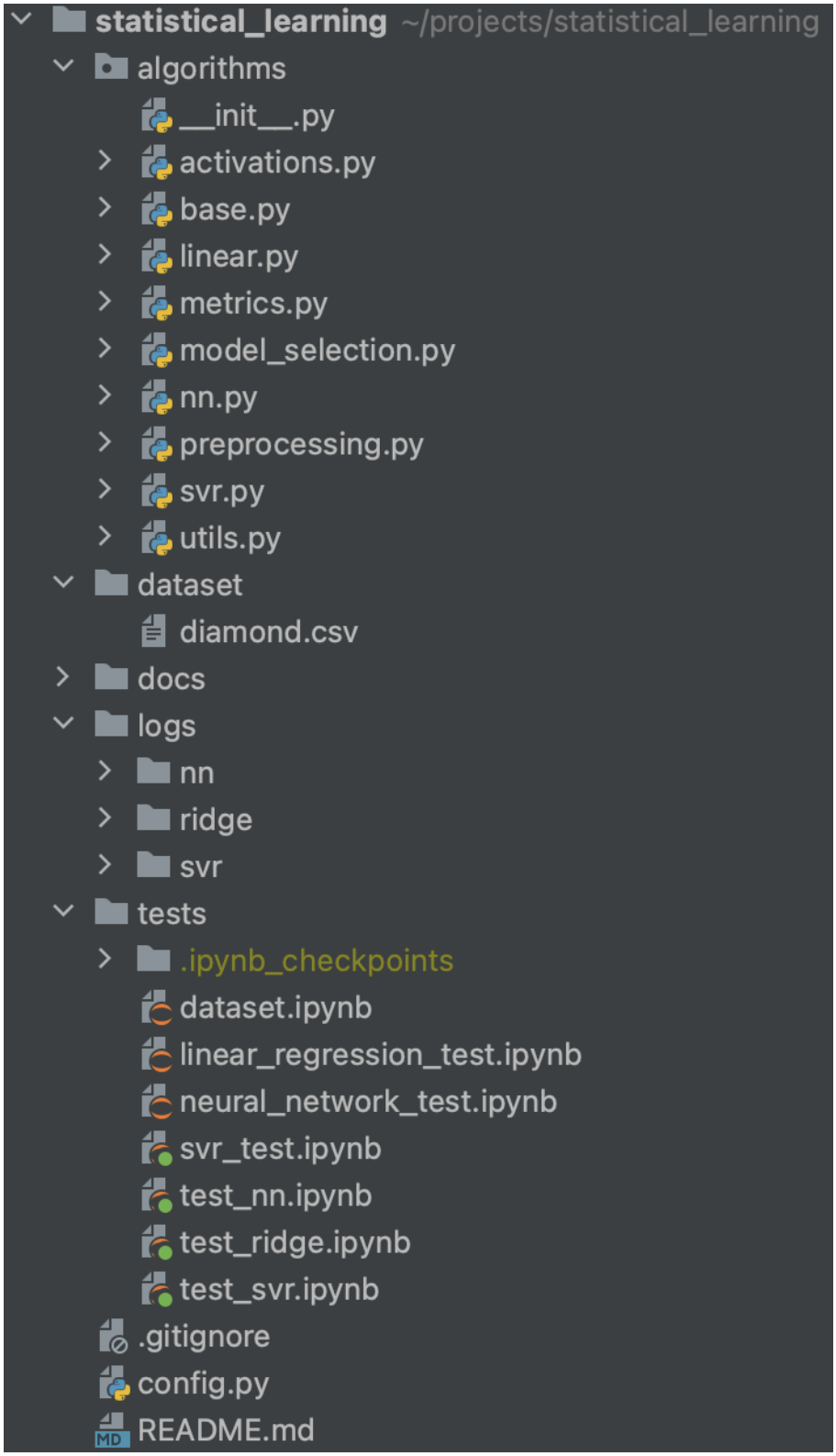
[1] The Observatory of Economic Complexity, “Diamonds,” oec.world/en.https://oec.world/en/profile/hs/diamonds (accessed October 16, 2022).

[2] M. Ali, ‘PyCaret: An open source, low-code machine learning library in Python’, April 2020.

[3] “Understanding Support Vector Machine Regression,” *Understanding Support Vector Machine Regression - MATLAB & Simulink*. [Online]. Available: https://www.mathworks.com/help/stats/understanding-support-vector-machine-regression.html. [Accessed: 20-Nov-2022].

[4] J. C. Platt, “Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines.”

Appendix A: The Project Structure



Appendix B: The Code

```
# activations.py

import numpy as np

from .base import Activation


class Linear(Activation):
    """
    The linear activation function.
    Implements f(x) = x.
    """
    def __call__(self, V):
        return V

    def backward(self, V, dZ):
        return dZ


class ReLU(Activation):
    """
    The Rectified Linear Unit (ReLU) activation function.
    Implements f(x) = x for x > 0 and 0 otherwise.
    """
    def __call__(self, V):
        return np.maximum(0, V)

    def backward(self, V, dZ):
        return np.where(V > 0, dZ, 0)

# base.py

from abc import ABC, abstractmethod
from copy import deepcopy

import numpy as np


class Activation(ABC):
    """
    Abstract Base Class for activation functions.
    """
    @abstractmethod
    def __call__(self, V):
        """
        Implements the activation function for the forward pass.
        :param V: The induced local field of the network layer
        :return: The output Z of the activation function
        """
```

```
        pass

    @abstractmethod
    def backward(self, V, dZ):
        """
        Implements the derivative of the activation function for the backward pass.
        :param V: The induced local field of the network layer after the forward pass
        :param dZ: The derivative of the layer output
        :return: the gradient of the cost with respect to V
        """
        pass

    def __repr__(self) -> str:
        """
        The string representation of the activation
        :return: the string representation
        """
        return str(self.__class__).strip('>').split(' ')[-1].strip("'")

    def __str__(self) -> str:
        """
        The string representation of the activation
        :return: the string representation
        """
        return repr(self)


class Model(ABC):
    @abstractmethod
    def __call__(self, *args, **kwargs):
        """
        Predicts the target vector given the feature matrix
        :param X: the feature matrix
        :return: predictions
        """
        pass

    @abstractmethod
    def fit(self, *args, **kwargs):
        """
        Trains the model given a dataset
        """
        pass

    @abstractmethod
    def predict(self, X: np.ndarray) -> np.ndarray:
        """
        Predicts the target vector given the feature matrix
        :param X: the feature matrix
        :return: predictions
        """
```

```
        pass

    def copy(self) -> 'Model':
        """
        Creates of a deepcopy of the model.
        Wraps Python's copy.deepcopy function
        :return: a copy of the model
        """
        return deepcopy(self)


# linear.py

from collections import defaultdict
from numbers import Number
from typing import Callable, Tuple, Union

from tqdm import tqdm
import numpy as np

from . import utils
from .base import Model
from .metrics import mse
from .model_selection import DEFAULT_METRICS


def check_dimensions(X=None, y=None) -> Tuple[Union[None, np.ndarray], Union[None, np.ndarray]]:
    """
    Checks the dimensions of the feature matrix and the target vector.
    :param X: the feature matrix
    :param y: the target vector
    :return: the feature matrix and the target vector
    """
    if X is not None and X.ndim != 2:
        raise ValueError('The number of dimensions of the feature matrix has to be 2.')
    if y is not None:
        if y.ndim == 1:
            y = y.reshape((-1, 1))
        elif y.ndim != 2:
            raise ValueError('The shape of the target matrix has to be (n, 1) or (n,),'
                             ' where n is the number of the training samples')
    return X, y


class LinearRegression(Model):
    """
    Linear regression model that fits the parameters using gradient descent.
    Only MSE is supported for gradient descent updates.

    Attributes:
        b : np.ndarray
```

```
        the bias vector of the model
    W : np.ndarray
        the weight matrix of the model
    alpha : Union[Number, Callable[[int], Number]]
        the learning rate of the model
    lambda_ : float
        the regularization parameter of the model
"""
def __init__(self, alpha: Union[Number, Callable[[int], Number]] = 1e-2, lambda_: float = 0):
    """
    The init method of the LinearRegression model
    :param alpha: the learning rate
    :param lambda_: the regularization constant
    :param normalize_features:
    """
    self.alpha = alpha
    self.lambda_ = lambda_
    self._b = None
    self._W = None
    if self.lambda_ < 0:
        raise ValueError('lambda must satisfy >= 0')

@property
def b(self) -> np.ndarray:
    """
    The y-intercept of the model
    :return: the y-intercept
    """
    return self._b

@property
def W(self) -> np.ndarray:
    """
    The weight matrix of the model
    :return: the weight matrix
    """
    return self._W

def __call__(self, X: np.ndarray) -> np.ndarray:
    """
    Wraps the model's predict method
    :param X: the feature matrix
    :return: predictions
    """
    return self.predict(X)

def __repr__(self) -> str:
    """
    Returns the initialization signature of the instance
    :return: the string representation
    """
```

```
    return f'LinearRegression(alpha={self.alpha}, lambda_={self.lambda_})'

def __str__(self) -> str:
    """
    Calls the repr method of the class
    :return: the string representation
    """
    return repr(self)

def initialize_parameters(self, in_features: int):
    self._b, self._W = utils.initialize_parameters(b_shape=(1, 1), W_shape=(in_features, 1))

def fit(self,
        X: np.ndarray,
        y: np.ndarray,
        X_valid,
        y_valid,
        epochs: int = None,
        batch_size: int = 32,
        min_delta: float = 1e-7,
        patience: int = 50,
        shuffle: bool = True,
        cold_start: bool = False) -> dict:
    """
    Calculates the weights and bias of the model using the gradient descent algorithm
    :param X: the feature matrix
    :param y: the target vector
    :param max_iter: maximum number of iterations
    :param tolerance: the tolerance for MSE loss below which the parameters are acceptable
    :param cold_start: whether to reinitialize the weights before training
    :return: the regressor itself
    """
    X = np.asarray(X)
    y = np.asarray(y)

    X, y = check_dimensions(X, y)

    if cold_start or self.b is None or self.W is None:
        self.initialize_parameters(X.shape[-1])

    n_batches = len(X) // batch_size
    history = defaultdict(list)

    n_no_improvement = 0
    for iteration in (progress_bar := tqdm(range(epochs))):
        alpha = self._get_alpha(iteration)
        train_indices = np.random.permutation(len(X)) if shuffle else np.arange(len(X))
        batch_average_metrics = defaultdict(list)
        for batch in range(n_batches):
            batch_indices = train_indices[batch * batch_size: (batch + 1) * batch_size]
            X_batch = X[batch_indices]
```

```
            y_batch = y[batch_indices]
            grad_b, grad_W = self._calculate_gradients(X_batch, y_batch)
            y_batch_pred = self.predict(X_batch)
            for metric, fn in DEFAULT_METRICS.items():
                batch_average_metrics[metric].append(fn(y_batch, y_batch_pred))
            self._b -= alpha * grad_b
            self._W -= alpha * grad_W
        train_avg_losses = {metric: np.mean(batch_average_metrics[metric])
                            for metric in DEFAULT_METRICS.keys()}
        valid_avg_losses = {metric: np.mean(fn(y_valid, self.predict(X_valid)))
                            for metric, fn in DEFAULT_METRICS.items()}

        for metric in DEFAULT_METRICS.keys():
            history[f'train_{metric}'].append(train_avg_losses[metric])
            history[f'valid_{metric}'].append(valid_avg_losses[metric])

        progress_bar.set_description_str(f'alpha={alpha}, lambda={self.lambda_},
batch_size={batch_size}')
        progress_bar.set_postfix_str(f'train_mse={train_avg_losses["MSE"]:.7f}, '
                                     f'valid_mse={valid_avg_losses["MSE"]:.7f}')

        if iteration > 2 and history['valid_MSE'][-2] - history['valid_MSE'][-1] < min_delta:
            n_no_improvement += 1
            if n_no_improvement > patience:
                break
        else:
            n_no_improvement = 0
    return history

  def predict(self, X: np.ndarray) -> np.ndarray:
    if self.b is None or self.W is None:
        raise RuntimeError('The model is not fit.')
    X = np.asarray(X)
    X, _ = check_dimensions(X)
    return self.b + X @ self.W

  def _get_alpha(self, iteration: int) -> float:
    """
    Calculates the learning rate.
    Returns the alpha parameter of the model if it is a float.
    Calls the alpha of the model with the current iteration number if it is a callable,
    :param iteration: the iteration number
    :return: learning rate
    """
    if isinstance(self.alpha, Number):
        alpha = self.alpha
    elif isinstance(self.alpha, Callable):
        alpha = self.alpha(iteration)
    else:
        raise ValueError('The alpha parameters must be of type float or Callable[[int], int]')
    return alpha
```

```
    def _calculate_gradients(self, X: np.ndarray, y: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
        """
        Calculate the gradients for MSE loss with current bias and weights
        :param X: the feature matrix
        :param y: the target vector
        :return: the gradients of the bias and the weights
        """
        y_pred = self.b + X @ self.W
        grad_b = 2 * np.sum(y_pred - y) / len(y)
        grad_W = 2 * (X.T @ (y_pred - y) + self.lambda_ * self.W) / len(y)
        return grad_b, grad_W

# metrics.py

import numpy as np


def mse(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    """
    Calculates the mean squared error given two vectors
    :param y_true: the true targets
    :param y_pred: the predicted targets
    :return: the mean squared error
    """
    y_true = np.asarray(y_true)
    y_pred = np.asarray(y_pred)
    return np.sum(np.square(y_true - y_pred)) / len(y_true)


def mae(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    """
    Calculates the mean absolute error given two vectors
    :param y_true: the true targets
    :param y_pred: the predicted targets
    :return: the mean absolute error
    """
    y_true = np.asarray(y_true)
    y_pred = np.asarray(y_pred)
    return np.sum(np.abs(y_true - y_pred)) / len(y_true)


def mape(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    """
    Calculates the mean absolute percentage error given two vectors
    :param y_true: the true targets
    :param y_pred: the predicted targets
    :return: the mean absolute percentage error
    """
    y_true = np.asarray(y_true)
    y_pred = np.asarray(y_pred)
```

```
  return np.sum(np.abs(np.divide(y_true - y_pred, y_true))) / len(y_pred)


def r2(y_true: np.ndarray, y_pred: np.ndarray) -> float:
  """
  Calculates the r2 score give two vectors
  :param y_true: the true targets
  :param y_pred: the predicted targets
  :return: the coefficient of determination
  """
  r = np.corrcoef(y_true.squeeze(), y_pred.squeeze())[0, 1]
  return r ** 2


# model_selection.py

from collections import namedtuple
from typing import Callable, Dict, List, Literal, Tuple, Union

import numpy as np
import pandas as pd

from .base import Model
from . import metrics as m


DEFAULT_METRICS = {
  'MSE': m.mse,
  'MAE': m.mae,
  'MAPE': m.mape,
  'R2': m.r2,
}


class KFoldGridSearch:
  def __init__(self, k, model, param_grid):
    self.k = k
    self.model = model
    self.param_grid = list(param_grid)

  def cv(self,
      X: np.ndarray,
      y: np.ndarray,
      shuffle: bool = True,
      **fit_params) -> list:
    """
    Apply k-fold cross validation with given dataset
    :param X: the feature matrix
    :param y: the target vector
    :param shuffle: whether to shuffle the data before cross validation
    :param fit_params: the keyword arguments to pass to the model's fit method
```

```
        :return: the KFold instance itself
        """
        X = np.asarray(X)
        y = np.asarray(y)
        indices = np.random.permutation(len(y)) if shuffle else np.arange(len(y))
        fold_size = len(y) // self.k
        scores = []
        for params in self.param_grid:
            model = self.model(*params)
            fold_scores = []
            for fold in range(self.k):
                valid_indices = indices[fold * fold_size: (fold + 1) * fold_size]
                train_indices = indices[~np.isin(indices, valid_indices)]

                X_train, y_train = X[train_indices], y[train_indices]
                X_valid, y_valid = X[valid_indices], y[valid_indices]
                history = model.fit(X_train, y_train, X_valid, y_valid, cold_start=True, **fit_params)
                fold_scores.append(history)
            scores.append(history)
        return scores


# nn.py

from collections import defaultdict, namedtuple
from itertools import product
from pathlib import Path
import json

from .model_selection import DEFAULT_METRICS

from tqdm import tqdm
import h5py
import numpy as np


def relu(z):
  return np.maximum(0, z)


def relu_backward(z):
  return np.where(z > 0, 1, 0)


# Weight and gradient containers for readability
RecurrentLayerWeights = namedtuple('RecurrentLayerWeights', ['b', 'Wx', 'Wh'])
RecurrentLayerGradients = RecurrentLayerWeights

FullyConnectedLayerWeights = namedtuple('FullyConnectedLayerWeights', ['b', 'W'])
FullyConnectedLayerGradients = FullyConnectedLayerWeights
```

```
class NeuralNetwork:
  def __init__(self, n_neurons):
     self.input_units = None              # will be determined when while training
     self.n_neurons = n_neurons             # number of neurons in the layers of MLP
     self.perceptron = None               # MLP layers

  def initialize_layers(self):
     """Initializes the weights and the hidden state of the recurrent layer"""
     rng = np.random.default_rng()

     # initialize MLP weights
     self.perceptron = []
     for i, n in enumerate(self.n_neurons):
        n_prev = self.n_neurons[i - 1] if i != 0 else self.input_units
        bound = np.sqrt(6 / (n + n_prev))
        b = rng.uniform(-bound, bound, size=(n, 1))
        W = rng.uniform(-bound, bound, size=(n, n_prev))

        layer = FullyConnectedLayerWeights(b, W)
        self.perceptron.append(layer)

  def __call__(self, X):
     """Inference mode forward pass through the network"""
     Z = X.T
     for layer in self.perceptron[:-1]:
        V = layer.W @ Z + layer.b
        Z = relu(V)
     V = self.perceptron[-1].W @ Z + self.perceptron[-1].b
     Z = V
     return Z.T

  def forward(self, X, y):
     """
     Training mode forward pass though the network.
     Caches the pre- and post-activation outputs of the layers
     """
     Z = X
     V_cache, Z_cache = [], []
     for layer in self.perceptron[:-1]:
        V = layer.W @ Z + layer.b
        Z = relu(V)

        V_cache.append(V)
        Z_cache.append(Z)

     V = self.perceptron[-1].W @ Z + self.perceptron[-1].b
     Z = V

     V_cache.append(V)
     Z_cache.append(Z)
```

```
    # compute training metrics
    J = {metric: fn(y.T, Z.T) for metric, fn in DEFAULT_METRICS.items()}
    return J, V_cache, Z_cache

def backward(self, X, y, V_cache, Z_cache):
    """Training mode backward pass through the network"""
    # calculate the gradients for the MLP
    gradients = []

    delta = (2 / Z_cache[-1].shape[1]) * np.subtract(Z_cache[-1], y)
    db = np.mean(delta, axis=1, keepdims=True)
    dW = delta @ Z_cache[-2].T / Z_cache[-2].shape[-1]
    gradients.append(FullyConnectedLayerGradients(db, dW))

    for i in reversed(range(1, len(self.perceptron) - 1)):
        delta = (self.perceptron[i + 1].W.T @ delta) * relu_backward(V_cache[i])
        db = np.mean(delta, axis=1, keepdims=True)
        dW = delta @ Z_cache[i - 1].T / Z_cache[i - 1].shape[-1]
        gradients.append(FullyConnectedLayerGradients(db, dW))

    delta = (self.perceptron[1].W.T @ delta) * relu_backward(V_cache[0])
    db = np.mean(delta, axis=1, keepdims=True)
    dW = (delta @ X.T) / X.shape[-1]
    gradients.append(FullyConnectedLayerGradients(db, dW))

    gradients.reverse()  # reverse the gradient list to get the correct order
    return gradients

def step(self, X, y):
    """Combines a forward and a backward pass through the network"""
    J, V_cache, Z_cache = self.forward(X, y)
    gradients = self.backward(X, y, V_cache, Z_cache)
    return J, gradients

def fit(self,
        X,
        y,
        X_valid,
        y_valid,
        alpha=0.1,
        momentum=0.85,
        epochs=50,
        batch_size=32,
        patience=5,
        min_delta=1e-7,
        shuffle=True,
        cold_start=False
        ):
    """
    Train the neural network
```

```
    :param X: the training features
    :param y: the training labels
    :param alpha: the learning rate
    :param momentum: the momentum
    :param epochs: the number of training epochs
    :param batch_size: the size of the training batches
    :param unfold: number of time steps to backpropagate in time
    :param shuffle: whether to shuffle the data each epoch
    :param X_valid: the validation features
    :param y_valid: the validation labels
    :param cold_start: whether to reinitialize the weights before training
    :return:
    """
    if cold_start or self.perceptron is None:
        self.input_units = X.shape[-1]
        self.initialize_layers()

    n_batches = len(X) // batch_size
    history = defaultdict(list)

    delta_weights_prev = None
    n_no_improvement = 0
    for epoch in (progress_bar := tqdm(range(epochs))):
        # obtain the shuffled indices for training and validation
        train_indices = np.random.permutation(len(X)) if shuffle else np.arange(len(X))
        batch_avg_losses = []
        for batch in range(n_batches):
            # get the batch data using the shuffled indices
            batch_indices = train_indices[batch * batch_size: (batch + 1) * batch_size]
            X_batch = X[batch_indices].T
            y_batch = y[batch_indices].T

            # forward and backward passes through the network
            J, gradients = self.step(X_batch, y_batch)

            batch_avg_losses.append(J)

            # calculate the updates given previous updates and gradients
            delta_weights = self.calculate_updates(delta_weights_prev, gradients, momentum)
            self.apply_updates(delta_weights, alpha)

            # save previous updates for momentum
            delta_weights_prev = delta_weights

        # test model performance on validation dataset
        valid_avg_losses = {metric: np.mean(fn(y_valid, self(X_valid)))
                            for metric, fn in DEFAULT_METRICS.items()}
        train_avg_losses = {metric: np.mean([loss[metric] for loss in batch_avg_losses])
                            for metric in DEFAULT_METRICS.keys()}

        # log training and validation metrics
```

```
            for metric in DEFAULT_METRICS.keys():
                history[f'train_{metric}'].append(train_avg_losses[metric])
                history[f'valid_{metric}'].append(valid_avg_losses[metric])

            progress_bar.set_description_str(f'n_neurons={"-".join([str(i) for i in self.n_neurons])}, '
                                             f'alpha={alpha}, momentum={momentum}, batch_size={batch_size}')
            progress_bar.set_postfix_str(f'train_mse={train_avg_losses["MSE"]:.7f}, '
                                         f'valid_mse={valid_avg_losses["MSE"]:.7f}')

            # stop the training if there is no improvement in last "tolerance" episodes
            if epoch > 2 and history['valid_MSE'][-2] - history['valid_MSE'][-1] < min_delta:
                n_no_improvement += 1
                if n_no_improvement > patience:
                    break
            else:
                n_no_improvement = 0
        return history

    @staticmethod
    def calculate_updates(delta_weights_prev, gradients, momentum):
        """Calculate the weight updates with momentum given previous updates and gradients"""
        delta_weights = [
            FullyConnectedLayerWeights(*[momentum * delta_w_prev + (1 - momentum) * grads
                                         for delta_w_prev, grads in zip(delta_weights_prev_, gradients_)])
            for delta_weights_prev_, gradients_ in zip(delta_weights_prev, gradients)
        ] if delta_weights_prev else gradients
        return delta_weights

    def apply_updates(self, delta_weights, alpha):
        """Apply the calculated updates to the network weights"""
        self.perceptron = [
            FullyConnectedLayerWeights(*[w - alpha * delta for w, delta in zip(layer, delta_weights_)])
            for layer, delta_weights_ in zip(self.perceptron, delta_weights)
        ]


# preprocessing.py

from pathlib import Path
from typing import Iterable, Tuple, Union

import numpy as np
import pandas as pd

import config


def load_dataset(path: Union[str, Path] = config.DATASET_PATH) -> pd.DataFrame:
    """
    Loads  dataset
```

```
    :param path: the file path of the dataset
    :return: the dataset
    """
    return pd.read_csv(path)


def split_dataset(data: pd.DataFrame,
                  target: str = config.TARGET,
                  test_split: float = 0.1,
                  seed: int = 42
                  ) -> Tuple[pd.DataFrame, pd.DataFrame, pd.DataFrame, pd.DataFrame]:
    """
    Splits the dataset
    :param data: the dataset
    :param target: the target column
    :param test_split: the fraction of the test data
    :param seed: the random seed
    :return: the train and test datasets
    """
    test_data = data.sample(frac=test_split, random_state=seed)
    train_data = data.drop(test_data.index)
    return train_data.drop(target, axis=1), train_data[[target]], \
           test_data.drop(target, axis=1), test_data[[target]],


def encode_categorical(data: pd.DataFrame,
                       columns: Union[Iterable[str], None] = None,
                       drop_first: bool = True) -> pd.DataFrame:
    """
    One-hot encodes the categorical data
    :param data: the dataset
    :param columns: the columns to encode. The columns are inferred if set to None
    :param drop_first: whether to drop the first categorical level
    :return:
    """
    if columns is not None:
        columns = list(columns)
    else:
        if config.TARGET in data.columns:
            data = data.drop(config.TARGET, axis=1)
        columns = list(data.select_dtypes(include=object).columns)
    return pd.get_dummies(data, columns=columns, drop_first=drop_first)


def normalize_columns(data: pd.DataFrame,
                      data_test: Union[None, pd.DataFrame] = None,
                      columns: Union[Iterable[str], None] = None
                      ) -> Union[pd.DataFrame, Tuple[pd.DataFrame, Union[None, pd.DataFrame]]]:
    """
    Normalize the features
    :param data: the train dataset
```

```
    :param data_test: the test dataset
    :param columns: the columns to normalize
    :return: the normalized feature matrix
    """
    data = data.copy()
    if columns is not None:
        columns = list(columns)
    else:
        if config.TARGET in data.columns:
            data = data.drop(config.TARGET, axis=1)
        columns = list(data.select_dtypes(exclude=object).columns)
    mean = data[columns].mean(axis=0)
    std = data[columns].std(axis=0)
    data[columns] = (data[columns] - mean) / std
    if data_test is not None:
        data_test[columns] = (data_test[columns] - mean) / std
    return data, data_test


def load_and_preprocess_dataset(path: Union[str, Path] = config.DATASET_PATH,
                                target: str = config.TARGET,
                                test_split: float = 0.1,
                                seed: int = 42,
                                normalize_features: bool = True,
                                normalize_target: bool = True,
                                encode: bool = True,
                                drop_first: bool = True
                                ) -> Tuple[pd.DataFrame, pd.DataFrame, pd.DataFrame, pd.DataFrame]:
    """
    :param path: the dataset filepath
    :param target: the target column
    :param test_split: the fraction of the data split for the test
    :param seed: the random seed
    :param normalize_features: whether to normalize features
    :param normalize_target: whether to normalize target
    :param encode: whether to one-hot encode the variables
    :param drop_first: whether to drop one feature column for each categorical variable
    :return: the dataset
    """
    data = load_dataset(path)
    if test_split == 0:
        X_train, y_train, X_test, y_test = data.drop(target, axis=1), data[[target]], None, None
    else:
        X_train, y_train, X_test, y_test = split_dataset(data, target, test_split, seed)
    if normalize_features:
        X_train, X_test = normalize_columns(X_train, X_test)
    if normalize_target:
        y_train, y_test = normalize_columns(y_train, y_test, columns=[config.TARGET])
    if encode:
        X_train = encode_categorical(X_train, drop_first=drop_first)
        X_test = encode_categorical(X_test, drop_first=drop_first)
```

```
    train_features = set(X_train.columns)
    test_features = set(X_test.columns)
    difference = list(train_features - test_features)
    X_test[difference] = np.zeros((len(X_test), len(difference)))
  return X_train, y_train, X_test, y_test

# svr.py

from collections import defaultdict

from tqdm import tqdm
import numpy as np

from algorithms import metrics
from algorithms.model_selection import DEFAULT_METRICS


def rbf(X, Y=None, gamma=1):
  X_norm = np.sum(X ** 2, axis=-1)
  if Y is None:
    Y = X
    Y_norm = X_norm
  else:
    Y_norm = np.sum(Y ** 2, axis=-1)
  return np.exp(-gamma * (X_norm[:, np.newaxis] + Y_norm[np.newaxis, :] - 2 * X @ Y.T))


class SupportVectorRegressor:
  def __init__(self,
               X,
               y,
               C,
               epsilon,
               tolerance,
               kernel_type,
               gamma):
    self.X = np.asarray(X)
    self.y = np.asarray(y)

    self.C = C
    self.epsilon = epsilon
    self.tolerance = tolerance
    self.kernel_type = kernel_type.lower()
    self.gamma = gamma
    self.kernel = None

    self.b = 0
    self.W = np.zeros(X.shape[1])
    self.alpha = np.zeros(X.shape[0])

    self.calculate_kernel()
```

```
  def calculate_kernel(self):
    if self.kernel_type == 'linear':
      self.kernel = self.X @ self.X.T
    elif self.kernel_type == 'rbf':
      self.kernel = rbf(X=self.X, gamma=self.gamma)
    else:
      raise NotImplementedError('Only linear and rbf kernels are implemented.')

  def calculate_error(self, i):
    return self.b + np.expand_dims(self.alpha, axis=0) @ np.expand_dims(self.kernel[i], axis=1) -
self.y[i]

  def check_KKT_condition_violations(self, i):
    Ai = self.alpha[i]
    Ei = self.calculate_error(i)

    violates = Ai == 0 and not (-self.epsilon <= Ei + self.tolerance and Ei <= self.epsilon + self.tolerance)
    violates = violates or ((-self.C < Ai < 0) and Ei != self.epsilon)
    violates = violates or (0 < Ai < self.C and Ei != -self.epsilon)
    violates = violates or (Ai == -self.C and not Ei >= self.epsilon - self.tolerance)
    violates = violates or (Ai == self.C and not Ei <= self.epsilon - self.tolerance)
    return violates

  def fit(self, X_valid, y_valid, max_iterations):
    history = defaultdict(list)

    for _ in (progress_bar := tqdm(range(max_iterations))):
      n_changes = 0
      for i in range(len(self.X)):
        if self.check_KKT_condition_violations(i):
          possible_j = np.setdiff1d(np.arange(len(self.X)), [i])
          j = np.random.choice(possible_j, size=1).item()
          n_changes += self.update_alpha(i, j)

      train_avg_losses = {metric: np.mean(fn(self.y, self.predict(self.X)))
                  for metric, fn in DEFAULT_METRICS.items()}
      valid_avg_losses = {metric: np.mean(fn(y_valid, self.predict(X_valid)))
                  for metric, fn in DEFAULT_METRICS.items()}

      for metric in DEFAULT_METRICS.keys():
        history[f'train_{metric}'].append(train_avg_losses[metric])
        history[f'valid_{metric}'].append(valid_avg_losses[metric])

      progress_bar.set_description_str(f'C={self.C}, epsilon={self.epsilon}, tolerance={self.tolerance}')
      progress_bar.set_postfix_str(f'train_mse={train_avg_losses["MSE"]:.7f}, '
                    f'valid_mse={valid_avg_losses["MSE"]:.7f}, '
                    f'number of changes: {n_changes}')
      if n_changes == 0:
        break
```

```
    return history

def predict(self, X):
    if self.kernel_type == 'linear':
        return self.b + X @ self.W.reshape(-1, 1)
    elif self.kernel_type == 'rbf':
        return (self.alpha @ rbf(self.X, X, self.gamma)).reshape(-1, 1) + self.b
    else:
        raise NotImplementedError('Only linear and rbf kernels are implemented.')

def update_alpha(self, i, j):
    Ei = self.calculate_error(i)
    Ej = self.calculate_error(j)

    L = np.max([-self.C, self.alpha[i] + self.alpha[j] - self.C])
    H = np.min([self.C, self.alpha[i] + self.alpha[j] + self.C])

    eta = self.kernel[i][i] - 2 * self.kernel[i][j] + self.kernel[j][j]

    if L == H or eta <= 0:
        return False

    Aj_updated_positive = (Ei - Ej + 2 * self.epsilon) / eta + self.alpha[j]
    Aj_updated_zero = (Ei - Ej) / eta + self.alpha[j]
    Aj_updated_negative = (Ei - Ej - 2 * self.epsilon) / eta + self.alpha[j]

    if self.alpha[i] + self.alpha[j] <= -self.C:
        Aj_updated = Aj_updated_zero

    elif -self.C < self.alpha[i] + self.alpha[j] < 0:
        if Aj_updated_positive < self.alpha[i] + self.alpha[j]:
            Aj_updated = Aj_updated_positive
        elif Aj_updated_zero <= self.alpha[i] + self.alpha[j]:
            Aj_updated = self.alpha[i] + self.alpha[j]
        elif self.alpha[i] + self.alpha[j] < Aj_updated_zero < 0:
            Aj_updated = Aj_updated_zero
        elif 0 < Aj_updated_negative:
            Aj_updated = Aj_updated_negative
        else:
            Aj_updated = 0

    elif self.alpha[i] + self.alpha[j] == 0:
        if Aj_updated_positive <= L:
            Aj_updated = L
        elif L < Aj_updated_positive < 0:
            Aj_updated = Aj_updated_positive
        elif 0 < Aj_updated_negative:
            Aj_updated = Aj_updated_negative
        else:
            Aj_updated = 0
```

```
        elif 0 < self.alpha[i] + self.alpha[j] < self.C:
            if Aj_updated_positive < 0:
                Aj_updated = Aj_updated_positive
            elif Aj_updated_zero <= 0:
                Aj_updated = 0
            elif 0 < Aj_updated_zero < self.alpha[i] + self.alpha[j]:
                Aj_updated = Aj_updated_zero
            elif self.alpha[i] + self.alpha[j] < Aj_updated_negative:
                Aj_updated = Aj_updated_negative
            else:
                Aj_updated = self.alpha[i] + self.alpha[j]

        else:
            Aj_updated = Aj_updated_zero

        Aj_updated = np.clip(Aj_updated, L, H)
        Ai_updated = self.alpha[i] + self.alpha[j] - Aj_updated

        dAi = Ai_updated - self.alpha[i]
        dAj = Aj_updated - self.alpha[j]

        if self.kernel_type == 'linear':
            self.W += np.ravel(dAi * self.X[i] + dAj * self.X[j])

        bi = self.b - dAi * self.kernel[i][i] - dAj * self.kernel[i][j] - Ei
        bj = self.b - dAi * self.kernel[i][j] - dAj * self.kernel[j][j] - Ej

        self.b = (bi + bj) / 2
        if 0 < Ai_updated < self.C:
            self.b = bi
        if 0 < Aj_updated < self.C:
            self.b = bj

        self.alpha[i] = Ai_updated
        self.alpha[j] = Aj_updated
        return True


# utils.py

from typing import Tuple

import numpy as np


def initialize_parameters(b_shape: Tuple, W_shape: Tuple) -> Tuple[np.ndarray, np.ndarray]:
    """
    Generates random weights from uniform distribution between 0 and 1
    :param b_shape: shape of the bias vector
    :param W_shape: shape of the weight matrix
    :return: the bias vector and the weight matrix
```

```
    """
    rng = np.random.default_rng()
    b = rng.uniform(-1, 1, b_shape)
    W = rng.uniform(-1, 1, W_shape)
    return b, W

# linear_regression_test.ipynb

from itertools import product
import pickle

import numpy as np

from algorithms.linear import LinearRegression
from algorithms.preprocessing import load_and_preprocess_dataset
import config
#%%
ridge_logs_dir = config.LOGS_DIR / 'ridge'
ridge_logs_dir.mkdir(exist_ok=True)
#%%
X, y, X_test, y_test = load_and_preprocess_dataset()
#%%
X = np.asarray(X)
y = np.asarray(y)
X_test = np.asarray(X_test)
y_test = np.asarray(y_test)
#%%
alphas = [1e-2, 5e-3, 1e-3, 5e-4]
#%%
lambdas = [1e-2, 1e-3, 1e-4, 0]
#%%
batch_sizes = [32, 64]
#%%
indices = np.random.permutation(len(y))
fold_size = len(indices) // 5
scores = []
for alpha, lambda_, batch_size in product(alphas, lambdas, batch_sizes):
    model = LinearRegression(alpha, lambda_)
    fold_scores = []
    for fold in range(5):
        valid_indices = indices[fold * fold_size: (fold + 1) * fold_size]
        train_indices = indices[~np.isin(indices, valid_indices)]

        X_train, y_train = X[train_indices], y[train_indices]
        X_valid, y_valid = X[valid_indices], y[valid_indices]

        history = model.fit(X_train, y_train, X_valid, y_valid, epochs=500,
                            batch_size=batch_size, cold_start=True,
                            patience=50, min_delta=1e-3)
        fold_scores.append(history)
    with open(ridge_logs_dir / f'alpha_{alpha}-lambda_{lambda_}-batch_size_{batch_size}.pkl', 'wb') as f:
```

```
        pickle.dump(fold_scores, f)

# neural_networks_test.ipynb

from itertools import product
import pickle

import numpy as np

from algorithms.nn import NeuralNetwork
from algorithms.preprocessing import load_and_preprocess_dataset

import config
#%%
nn_logs_dir = config.LOGS_DIR / 'nn'
nn_logs_dir.mkdir(exist_ok=True)
#%%
X, y, X_test, y_test = load_and_preprocess_dataset()
#%%
X = np.asarray(X)
y = np.asarray(y)
X_test = np.asarray(X_test)
y_test = np.asarray(y_test)
#%%
layers_list = [
  [32, 1],
  [64, 1],
  [32, 32, 1],
  [64, 64, 1],
]
#%%
alphas = [1e-2, 5e-3, 1e-3, 5e-4]
#%%
momentums = [0.85, 0.95]
#%%
batch_sizes = [32, 64]
#%%
indices = np.random.permutation(len(y))
fold_size = len(indices) // 5
scores = []
for layers, alpha, momentum, batch_size in product(layers_list, alphas, momentums, batch_sizes):
  model = NeuralNetwork(layers)
  fold_scores = []
  for fold in range(5):
      valid_indices = indices[fold * fold_size: (fold + 1) * fold_size]
      train_indices = indices[~np.isin(indices, valid_indices)]

      X_train, y_train = X[train_indices], y[train_indices]
      X_valid, y_valid = X[valid_indices], y[valid_indices]

      history = model.fit(X_train, y_train, X_valid, y_valid, alpha=alpha,
```

```
                    batch_size=batch_size, epochs=500, cold_start=True,
                    patience=50, min_delta=1e-3)
        fold_scores.append(history)
    with open(nn_logs_dir /
f'alpha_{alpha}-momentum_{momentum}-batch_size_{batch_size}-layers_{"_".join([str(x) for x in
layers])}.pkl', 'wb') as f:
        pickle.dump(fold_scores, f)
#%%


# svr_test.ipynb

from itertools import product
import pickle

import numpy as np

from algorithms.svr import SupportVectorRegressor
from algorithms.preprocessing import load_and_preprocess_dataset

import config
#%%
svr_logs_dir = config.LOGS_DIR / 'svr'
svr_logs_dir.mkdir(exist_ok=True)
#%%
X, y, X_test, y_test = load_and_preprocess_dataset()
#%%
X = np.asarray(X)
y = np.asarray(y)
X_test = np.asarray(X_test)
y_test = np.asarray(y_test)
#%%
hyperparameters = [
    (0.1, 0.001, 0.01, 'linear', 0.5),
    (0.1, 0.001, 0.01, 'rbf', 0.5),
    (0.1, 0.001, 0.001, 'linear', 0.5),
    (0.1, 0.001, 0.001, 'rbf', 0.5),
    (0.1, 0.001, 0.001, 'rbf', 1),
    (0.1, 0.01, 0.01, 'linear', 0.5),
    (0.1, 0.01, 0.001, 'rbf', 0.5),
    (0.2, 0.001, 0.001, 'rbf', 0.5),
    (0.2, 0.001, 0.001, 'rbf', 1),
    (0.2, 0.01, 0.001, 'linear', 1),
    (0.2, 0.01, 0.001, 'rbf', 1),
]
#%%
indices = np.random.permutation(len(y))
fold_size = len(indices) // 5
scores = []

for C, epsilon, tolerance, kernel_type, gamma in hyperparameters:
```

```
  fold_scores = []
  for fold in range(5):
      valid_indices = indices[fold * fold_size: (fold + 1) * fold_size]
      train_indices = indices[~np.isin(indices, valid_indices)]

      X_train, y_train = X[train_indices], y[train_indices]
      X_valid, y_valid = X[valid_indices], y[valid_indices]

      model = SupportVectorRegressor(X_train, y_train, C, epsilon, tolerance, kernel_type=kernel_type,
gamma=gamma)
      history = model.fit(X_valid, y_valid, max_iterations=250)
      fold_scores.append(history)

  with open(svr_logs_dir /
f'C_{C}-epsilon_{epsilon}-tolerance_{tolerance}-kernel_type_{kernel_type}-gamma_{gamma}.pkl',
'wb') as f:
      pickle.dump(fold_scores, f)
#%%

# test_ridge.ipynb

from collections import defaultdict
from itertools import product
import pickle

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from algorithms.linear import LinearRegression
from algorithms.preprocessing import load_and_preprocess_dataset
from algorithms.model_selection import DEFAULT_METRICS
import config
#%%
X_train, y_train, X_test, y_test = load_and_preprocess_dataset()
#%%
ridge_logs_dir = config.LOGS_DIR / 'ridge'
nn_logs_dir = config.LOGS_DIR / 'nn'
svr_logs_dir = config.LOGS_DIR / 'svr'
#%%
ridge_alphas = [1e-2, 5e-3, 1e-3, 5e-4]
ridge_lambdas = [1e-2, 1e-3, 1e-4, 0]
ridge_batch_sizes = [32, 64]

ridge_hyperparams = list(product(ridge_alphas, ridge_lambdas, ridge_batch_sizes))
#%%
ridge_scores = []
for alpha, lambda_, batch_size in ridge_hyperparams:
  with open(ridge_logs_dir / f'alpha_{alpha}-lambda_{lambda_}-batch_size_{batch_size}.pkl', 'rb') as f:
      ridge_histories = pickle.load(f)
  ridge_scores.append(ridge_histories)
```

```
#%%
ridge_last_scores = []

for model in ridge_scores:
  model_last_scores = defaultdict(int)
  for fold in model:
      for metric in DEFAULT_METRICS.keys():
          model_last_scores[f'train_{metric}'] += fold[f'train_{metric}'][-1] / len(model)
          model_last_scores[f'valid_{metric}'] += fold[f'valid_{metric}'][-1] / len(model)
  ridge_last_scores.append(model_last_scores)
#%%
valid_ridge = {metric: [ls[f'valid_{metric}'] for ls in ridge_last_scores] for metric in
DEFAULT_METRICS.keys()}
#%%
plt.figure()
plt.title('The number of models with a given validation MSE')
plt.xlabel('MSE')
plt.ylabel('Number of models')
plt.hist(valid_ridge['MSE'])
plt.show()
#%%
print(np.mean(valid_ridge['MSE']))
#%%
ridge_best_score = np.inf
ridge_best_model_index = -1

for i, model in enumerate(ridge_last_scores):
  if model['valid_MAPE'] < ridge_best_score:
      ridge_best_model_index = i
      ridge_best_score = model['valid_MAPE']
#%%
for k, v in ridge_last_scores[ridge_best_model_index].items():
  if k.startswith('train'):
      print(k, v)
#%%
for k, v in ridge_last_scores[ridge_best_model_index].items():
  if k.startswith('valid'):
      print(k, v)
#%%
print(np.mean(valid_ridge['MSE']), np.std(valid_ridge['MSE']))
#%%
mask = [int(lambda_ > 5e-4) for alpha, lambda_, batch_size in ridge_hyperparams]
print(np.sum(np.asarray(valid_ridge['MSE']) * np.asarray(mask)) / np.sum(mask))
#%%
mask = [int(batch_size == 64) for alpha, lambda_, batch_size in ridge_hyperparams]
print(np.sum(np.asarray(valid_ridge['MSE']) * np.asarray(mask)) / np.sum(mask))
#%%
mask = [int(batch_size == 32) for alpha, lambda_, batch_size in ridge_hyperparams]
print(np.sum(np.asarray(valid_ridge['MSE']) * np.asarray(mask)) / np.sum(mask))
#%%
best_model_params = ridge_hyperparams[ridge_best_model_index]
```

```
print(best_model_params)
#%%
best_model = LinearRegression(alpha=0.001, lambda_=0.001)
history = best_model.fit(X_train, y_train, X_test, y_test, epochs=500)
#%%
h = pd.DataFrame.from_dict(history)[['train_MSE', 'valid_MSE']]
h = h.set_axis(['Train MSE', 'Test MSE'], axis=1)
h.plot()
plt.xlabel('Epochs')
plt.ylabel('MSE')
plt.title('MSE vs Epoch')
plt.show()
#%%


# test_nn.ipynb

from collections import defaultdict
from itertools import product
import pickle

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from algorithms.nn import NeuralNetwork
from algorithms.preprocessing import load_and_preprocess_dataset
from algorithms.model_selection import DEFAULT_METRICS
import config
#%%
X_train, y_train, X_test, y_test = load_and_preprocess_dataset()
X_train = np.asarray(X_train)
y_train = np.asarray(y_train)
X_test = np.asarray(X_test)
y_test = np.asarray(y_test)
#%%
ridge_logs_dir = config.LOGS_DIR / 'ridge'
nn_logs_dir = config.LOGS_DIR / 'nn'
svr_logs_dir = config.LOGS_DIR / 'svr'
#%%
nn_layers_list = [
  [32, 1],
  [64, 1],
  [32, 32, 1],
  [64, 64, 1],
]
nn_alphas = [1e-2, 5e-3, 1e-3, 5e-4]
nn_momentums = [0.85, 0.95]
nn_batch_sizes = [32, 64]

nn_hyperparams = list(product(nn_layers_list, nn_alphas, nn_momentums, nn_batch_sizes))
```

```
#%%
nn_scores = []
for layers, alpha, momentum, batch_size in nn_hyperparams:
  with open(nn_logs_dir /
f'alpha_{alpha}-momentum_{momentum}-batch_size_{batch_size}-layers_{"_".join([str(x) for x in
layers])}.pkl', 'rb') as f:
      nn_histories = pickle.load(f)
  nn_scores.append(nn_histories)
#%%
nn_last_scores = []

for model in nn_scores:
  model_last_scores = defaultdict(int)
  for fold in model:
      for metric in DEFAULT_METRICS.keys():
          model_last_scores[f'train_{metric}'] += fold[f'train_{metric}'][-1] / len(model)
          model_last_scores[f'valid_{metric}'] += fold[f'valid_{metric}'][-1] / len(model)
  nn_last_scores.append(model_last_scores)
#%%
nn_best_score = np.inf
nn_best_model_index = -1

for i, model in enumerate(nn_last_scores):
  if model['valid_MAPE'] < nn_best_score:
      nn_best_model_index = i
      nn_best_score = model['valid_MAPE']
#%%
valid_nn = {metric: [ls[f'valid_{metric}'] for ls in nn_last_scores] for metric in
DEFAULT_METRICS.keys()}
#%%
print(np.mean(valid_nn['MSE']), np.std(valid_nn['MSE']))
#%%
plt.figure()
plt.title('The number of models with a given validation MSE')
plt.xlabel('MSE')
plt.ylabel('Number of models')
plt.hist(valid_nn['MSE'])
plt.show()
#%%
nn_hyperparams[nn_best_model_index]
#%%
for k, v in nn_last_scores[nn_best_model_index].items():
  if k.startswith('train'):
      print(k, v)
#%%
for k, v in nn_last_scores[nn_best_model_index].items():
  if k.startswith('valid'):
      print(k, v)
#%%
best_model = NeuralNetwork(n_neurons=[64, 64, 1])
```

```
history = best_model.fit(X_train, y_train, X_test, y_test, alpha=0.01, batch_size=32,
momentum=0.85,epochs=500, patience=50, min_delta=1e-4)
#%%
h = pd.DataFrame.from_dict(history)[['train_MSE', 'valid_MSE']]
h = h.set_axis(['Train MSE', 'Test MSE'], axis=1)
h.plot()
plt.xlabel('Epochs')
plt.ylabel('MSE')
plt.title('MSE vs Epoch')
plt.show()
#%%

# test_svr.ipynb

from collections import defaultdict
from itertools import product
import pickle

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from algorithms.svr import SupportVectorRegressor
from algorithms.preprocessing import load_and_preprocess_dataset
from algorithms.model_selection import DEFAULT_METRICS
import config
#%%
X_train, y_train, X_test, y_test = load_and_preprocess_dataset()
#%%
X_train = np.asarray(X_train)
y_train = np.asarray(y_train)
X_test = np.asarray(X_test)
y_test = np.asarray(y_test)
#%%
ridge_logs_dir = config.LOGS_DIR / 'ridge'
nn_logs_dir = config.LOGS_DIR / 'nn'
svr_logs_dir = config.LOGS_DIR / 'svr'
#%%
hyperparameters = [
  (0.1, 0.001, 0.01, 'linear', 0.5),
  (0.1, 0.001, 0.01, 'rbf', 0.5),
  (0.1, 0.001, 0.001, 'linear', 0.5),
  (0.1, 0.001, 0.001, 'rbf', 0.5),
  (0.1, 0.001, 0.001, 'rbf', 1),
  (0.1, 0.01, 0.01, 'linear', 0.5),
  (0.1, 0.01, 0.001, 'rbf', 0.5),
  (0.2, 0.001, 0.001, 'rbf', 0.5),
  (0.2, 0.001, 0.001, 'rbf', 1),
  (0.2, 0.01, 0.001, 'linear', 1),
  (0.2, 0.01, 0.001, 'rbf', 1),
]
```

```
#%%
svr_scores = []
for C, epsilon, tolerance, kernel_type, gamma in hyperparameters:
  with open(svr_logs_dir /
f'C_{C}-epsilon_{epsilon}-tolerance_{tolerance}-kernel_type_{kernel_type}-gamma_{gamma}.pkl',
'rb') as f:
      ridge_histories = pickle.load(f)
  svr_scores.append(ridge_histories)
#%%
svr_last_scores = []

for model in svr_scores:
  model_last_scores = defaultdict(int)
  for fold in model:
      for metric in DEFAULT_METRICS.keys():
          model_last_scores[f'train_{metric}'] += fold[f'train_{metric}'][-1] / len(model)
          model_last_scores[f'valid_{metric}'] += fold[f'valid_{metric}'][-1] / len(model)
  svr_last_scores.append(model_last_scores)
#%%
valid_svr = {metric: [ls[f'valid_{metric}'] for ls in svr_last_scores] for metric in
DEFAULT_METRICS.keys()}
#%%
plt.figure()
plt.title('The number of models with a given validation MSE')
plt.xlabel('MSE')
plt.ylabel('Number of models')
plt.hist(valid_svr['MSE'])
plt.show()
#%%
print(np.mean(valid_svr['MSE']))
#%%
svr_best_score = np.inf
svr_best_model_index = -1

for i, model in enumerate(svr_last_scores):
  if model['valid_MAPE'] < svr_best_score:
      svr_best_model_index = i
      svr_best_score = model['valid_MAPE']
#%%
for k, v in svr_last_scores[svr_best_model_index].items():
  if k.startswith('train'):
      print(k, v)
#%%
for k, v in svr_last_scores[svr_best_model_index].items():
  if k.startswith('valid'):
      print(k, v)
#%%
print(np.mean(valid_svr['MSE']), np.std(valid_svr['MSE']))
#%%
best_model_params = hyperparameters[svr_best_model_index]
print(best_model_params)
```

```
#%%
best_model = SupportVectorRegressor(X_train, y_train, C=0.1, epsilon=0.001, tolerance=0.01,
kernel_type='rbf', gamma=0.5)
history = best_model.fit(X_test, y_test, max_iterations=250)
#%%
h = pd.DataFrame.from_dict(history)[['train_MSE', 'valid_MSE']].iloc[:50]
h = h.set_axis(['Train MSE', 'Test MSE'], axis=1)
h.plot()
plt.xlabel('Epochs')
plt.ylabel('MSE')
plt.title('MSE vs Epoch')
plt.show()
#%%
history['valid_MSE']
```