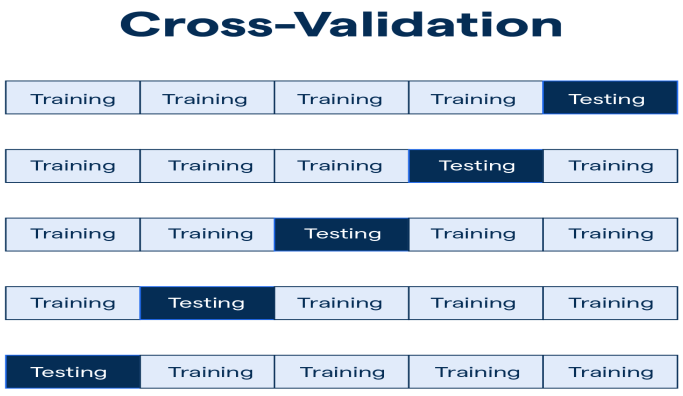
Exam Prep

All the stuff needed in the exam with code and answers, I pray I don't fail this.., First look at this

Cross validation

So what is cross validation?

It's a Machine learning technique used to divide the model into multiple different parts to train the ML model on the data it has not seen yet, an example of this is below



Simple Cross validation use train-test method before doing anything

```
cross_validation = cross_val_score(pipeline, X_train, y_train, cv=5, scoring='accuracy')
```

be aware the cross validation score sometime uses the negative mean squared error, so the result will be negative, solve it with -np.mean when wanting to make them positive, ***cv=5 is 5 folds cross validation***

Can be with negative mean squared error

```
cv_scores = cross_val_score(model, X, y, cv=7, scoring='neg_mean_squared_error')
```

What should it output and display?

```
# Display cross-validation results
print("Cross-Validation MSE (mean squared error):", -cv_scores)
print("Average CV MSE:", -cv_scores.mean())
```

If the scoring='***accuracy***' the -minus shouldn't be used

What does the score mean? ***// answer later***

Grid search

It's an algorithm in machine learning, used to try every single model so it can get the best model with the best hyperparameters.

```
following number of trees [10, 20,40, 50, 100,150, 200,250, 300, 350,450,500]
```

We fill the parameters here, they wanted the above number of trees aka the number of estimators

```
param_grid = {
    'n_estimators': [10, 20,40, 50, 100,150, 200,250, 300, 350,450,500]  # Different number of trees
}
```

They wanted us to use RandomForestRegressor model, you can use other models if requested, they will be below

```
rf = RandomForestRegressor(random_state=42)
```

We fill our GridSearchCV (Cross Validation) with data, this is 5 folds CV, it uses a RandomForestRegressor()

```
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, cv=5, scoring='neg_mean_squared_error', n_jobs=1)
```

We fit our data we split using train-test split

```
grid_search.fit(X_train, y_train)
```

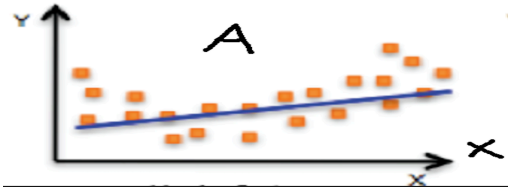
The values you can and should get from grid search

```
# Get the best parameters and best score
best_params = grid_search.best_params_
best_score = grid_search.best_score_

print(f"Best Parameters: {best_params}")
print(f"Best Cross-Validation Score (Negative MSE):
{best_score}")

best_rf = grid_search.best_estimator_
```

*The **best_estimator_** makes **best_rf (best RandomForestRegressor)** have the best parameters so you can use it to train data*

Underfitting

Underfitting is when our model does not capture the relationships between the input and output variables in ML, in diagrams happens when our model regression line does this



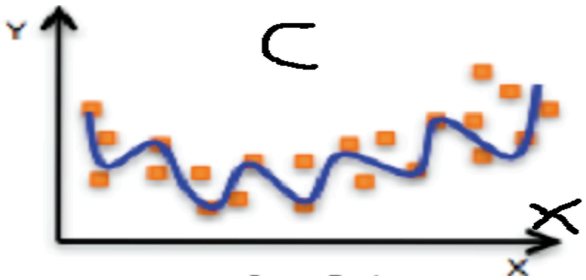
The above is a prime example of underfitting in linear regression,

A question : my data is under-fitted, what are the 3 possible ways to solve it?

- Increase the training data
- decrease the regularisation
- Try a more complex model, or another model

Overfitting

Happens when a model memorises the data so much that it cannot predict anything other than the training data provided, good on training data, poor on new test data in ML, ***in the regression model example*** **:**



A question : my data is overfitted, give 3 solutions on what to do

- Increase the training data
- Increase the regularisation, reduce the number of features
- Try a simpler model
- Tip: use cross-validation to detect possible overfitting

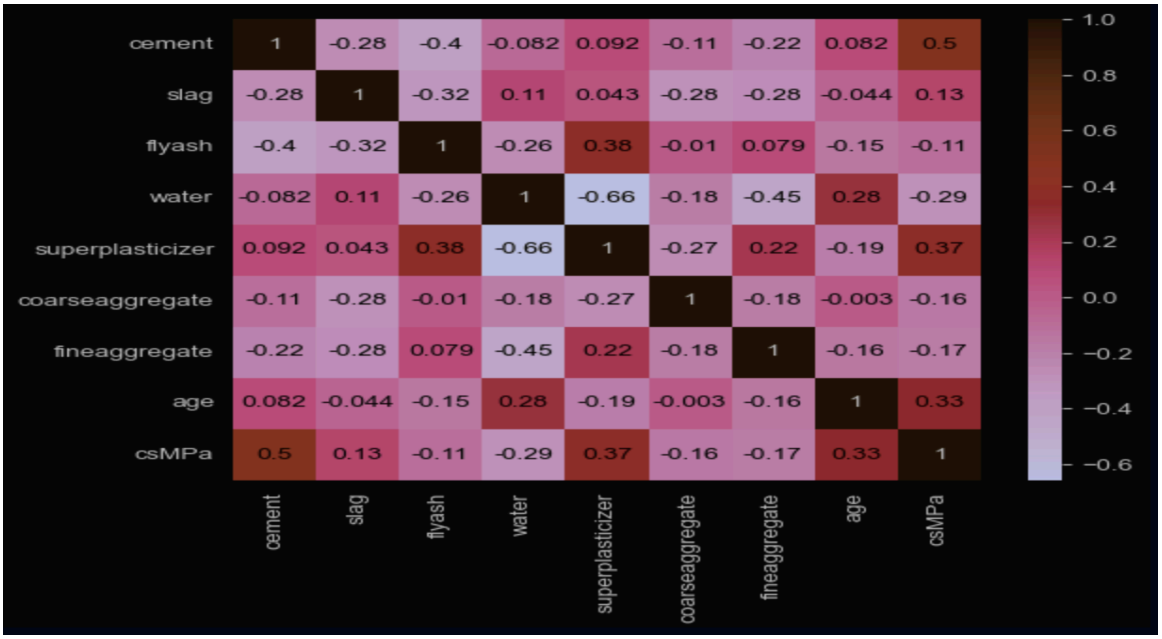
Correlation

A **correlation** is when there's a relationship between 2 or more variables For example how the change of a variable affects the other variables,

Here a below example of us displaying the heatmap of the correlation between all the variables in the concrete_data data frame from the .csv

sns.heatmap(concrete_data.corr(), annot=True)

The above code will display the following ***heatmap matrix***



Now we can see the **correlation** between the variables, in the question's case we want to get the variables with the highest correlation with csMPa, so the result is : { cement, age, superplasticizer } that's how we get them, using the corr and heatmap, look at the rows and columns of it

Know how to show a correlation from data

Use the **.corr()** like above and look for the matrix which one has the highest numbers has the highest correlation, the 1's are the data columns with each other so skip them, unless you saw 2 different column names

Histograms

A histogram is a diagram used to show the amount of times a data appears in the dataset, basically the occurrence of stuff

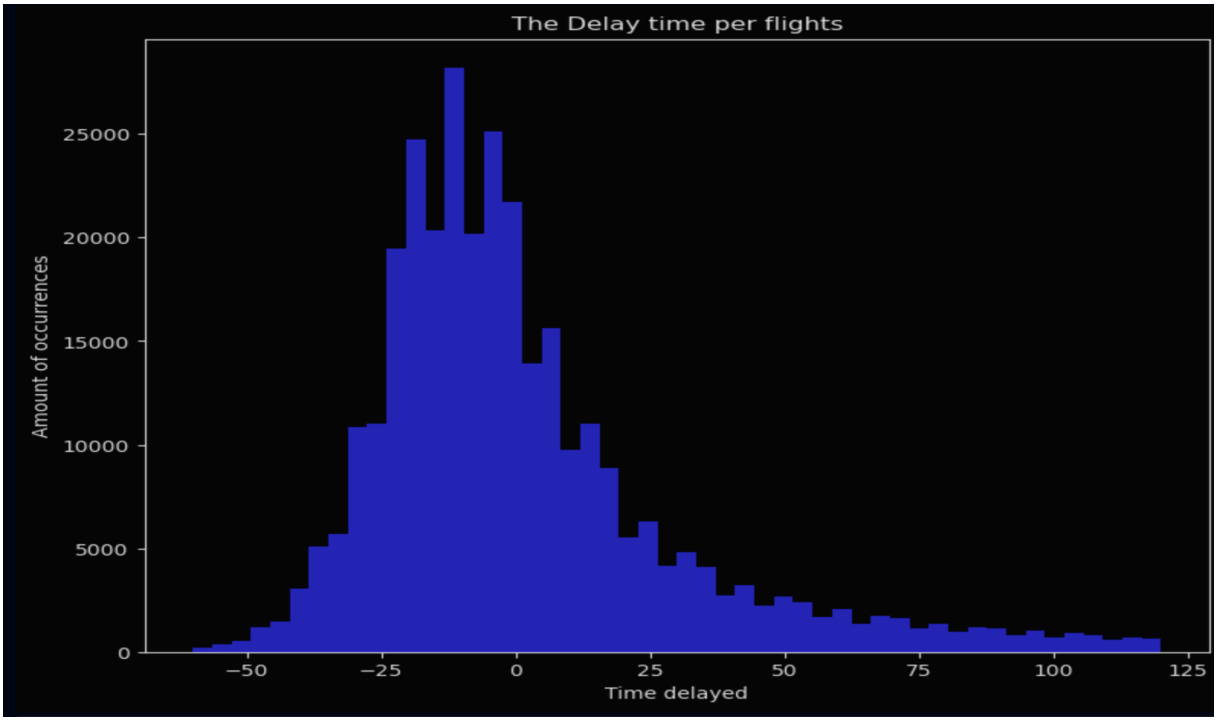
plt.hist() # is used to show data in histogram, you give it the x and it will show it's occurrences, below is what you can do in the ( )

```
hist(x: Any,
     bins: Any = None,
     range: Any = None,
     density: bool = False,
     weights: Any = None,
     cumulative: bool = False,
     bottom: Any = None,
     histtype: str = 'bar',
     align: str = 'mid',
     orientation: str = 'vertical',
     rwidth: Any = None,
     log: bool = False,
     color: Any = None,
     label: Any = None,
     stacked: bool = False,
     *,
     data: Any = None,
     **kwargs: Any) -> Any
```

In the below example we show the amount of delays per minute

plt.hist(merged1['arr_delay'], bins=50, color='blue')

The above code will display this, from the arr_delay from flights.csv



As seen like we explained it shows a histogram, **btw it may not exactly be like this**, i used **plt.figure(figsize=(9, 7))** and gave it a label for x and y and gave it a title, you have to do that ***yourself***

How to plot figures

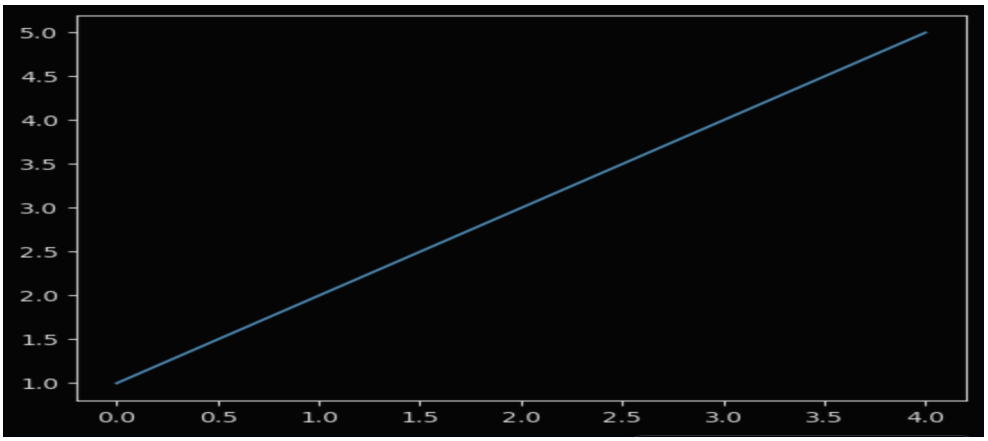
plt.plot() # used to show data in a linear way, these are the ( ) params

```
plot(*args: Any,
     scalex: bool = True,
     scaley: bool = True,
     data: Any = None,
     **kwargs: Any) -> Any
```

A simple code example :

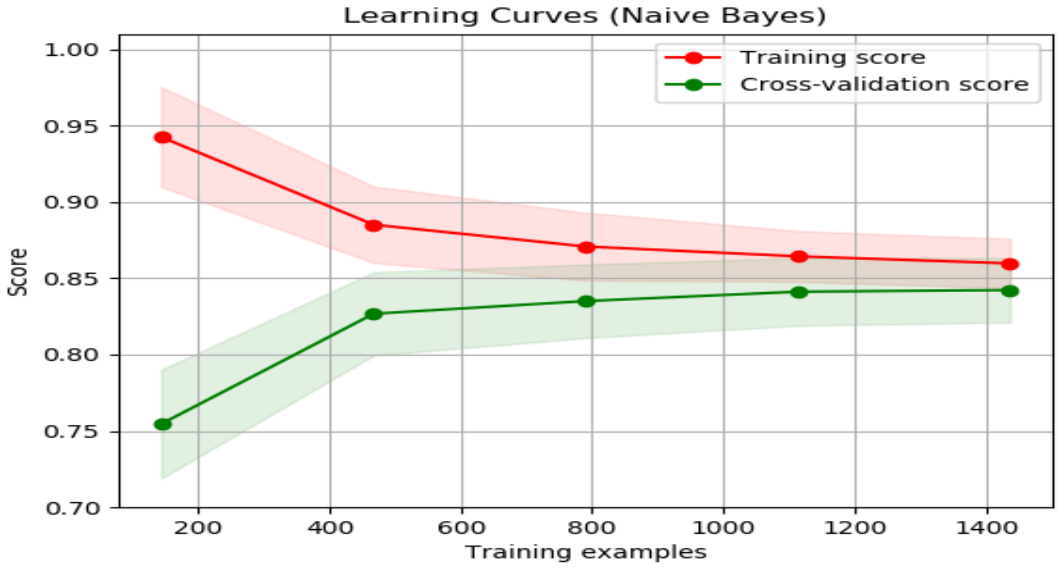
numbers1 = [1, 2, 3, 4,5]
plt.plot(numbers1)
plt.show()

Will produce the below photo



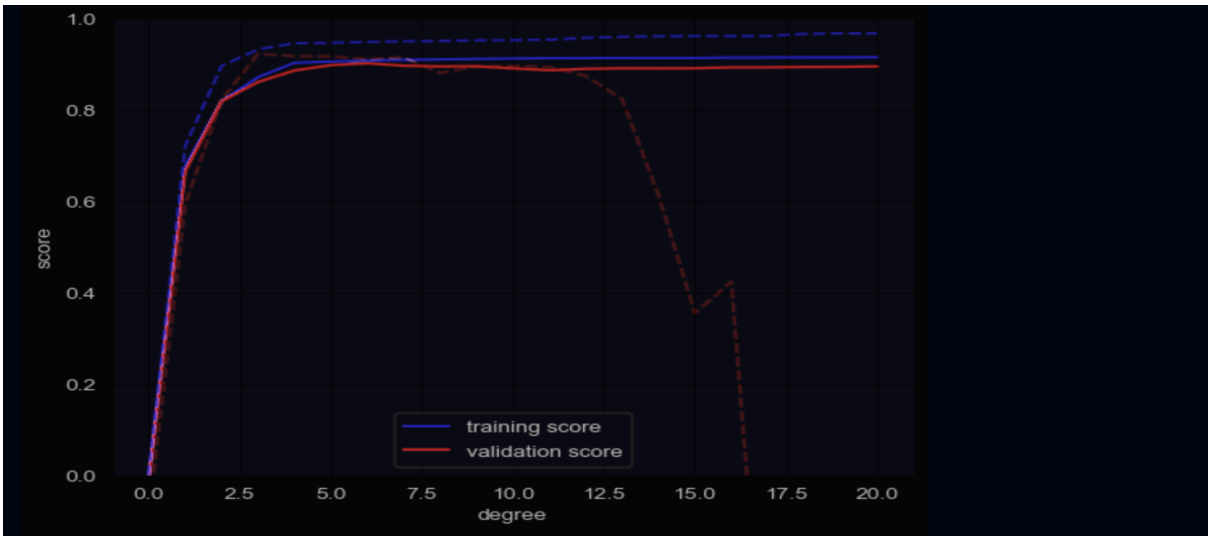
Learning curves

Basically shows how the model learns the data



Training curves and losses

Let's take this example : we have the dashed lines with curves that have a score that goes down after we use a more complex mode, and we have a normal line that has data that has consistent score, aka no loss, what's the difference between them? The one with the loss is the smaller data set, the one with the consistent lines is a large data set



Confusion matrix

First split the data using the train test split, after that if you use a model or a pipeline or anything to ***fit the train data***, predict with that model a value that you name y_pred using the X_test, use the y_pred and y_test in your confusion matrix, you can use them in other stuff too

```
y_pred = pipeline_dt.predict(X_test)
print("Confusion matrix", confusion_matrix(y_test, y_pred1))
```

Classification Report

The same as above in Confusion Matrix

```
y_pred = pipeline_dt.predict(X_test)
print("classification report", classification_report(y_test,y_pred1))
```

Decision trees

There's 2 types of Decision Trees.

- DecisionTreeRegressor
- DecisionTreeClassifier

DecisionTreeRegressor

First do the train-test split on data you defined

Define the model

```
regressor = DecisionTreeRegressor(random_state=42)
```

Fit the model

```
regressor.fit(X_train, y_train)
```

Predict the y variable to use in evaluating the model

```
y_pred = regressor.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

You need to visualise the actual vs predicted

DecisionTreeClassifier

First do the train-test split on data

After that define the model, random_state so random data don't change, can be done with scaler, check Pipelines

```
DTree =  DecisionTreeClassifier(random_state=42)
```

Perform cross validation, and calculate the accuracy

```
scores_dt = cross_val_score(DTree, X_train, y_train,
cv=5, scoring='accuracy')
```

Fit the model

```
DTree.fit(X_train, y_train)
```

Evaluate on the test set : aka get y_pred

```
y_pred_dt = DTree.predict(X_test)
```

The data you can get from the decision tree, test accuracy, classification report, and confusion matrix

```
print(f"Test Accuracy: {DTree.score(X_test,
y_test):.4f}")
print("Classification Report:\n",
classification_report(y_test, y_pred_dt))
#other important metrics
print("Confusion Matrix:\n", confusion_matrix(y_test,
y_pred_dt))
```

Learn how to apply grid search with different values for the same models.

This was already done down in GridSearch and some params GridParam, you can also see Grid Search above in titles

Neural networks

Implement regression models

Just click the link

Implement classification models

Just click to be redirected to the code

Difference between regression and classification models

- Regression model
  They deal with numbers and it's about predicting the futures based on the given numerical data, like will the stock market go up or how much prices of groceries will be next year

- Classification model
  They deal with categories, they classify data, for example like does the patient have cancer or no, or it sees an animal image and decides is this animal a cat or a dog

How to evaluate your model

```
model.evaluate(x_test_norm,y_test_onehot)
```

You get the x_test_norm and y_test_onehot from the following

- ***x_test_norm*** (MinMaxScaler() normalised data) :

```
scaler = MinMaxScaler()
x_train_norm = scaler.fit_transform(X_train)
```

- ***y_test_onehot*** (use ***to_categorical()***), if data isn't a DataFrame :

```
y_test_onehot = to_categorical(y_test)
```

If the data is a DataFrame ***pd.get_dummies()***

```
df = pd.DataFrame({'color': ['red', 'blue', 'green'], 'size': ['S', 'M', 'L']})
df_encoded = pd.get_dummies(df, columns=['color', 'size'])
```

How to decide loss functions

You decide the loss function based on the data you want to predict, if you want to predict categories like cat or dog or values like future price

If the data in you want to predict is :

- If the data is in Categories *(Classification)*, like cats or dogs like 0 to 9 image prediction : `loss='categorical_crossentropy'`

- If the data is Regression like you want to predict the price in future or MPG aka a Number, use: `loss='mean_squared_error'`

Example :

```
## EXERCISE 2

The goal is to predict the quality class of wine, based on features
that come from a chemical analysis of the wine.
```

Answer : Use `loss='categorical_crossentropy'` because you want to predict the **QUALITY CLASS**, aka ***a category from 0 to 2***

```
## EXERCISE 3

In this exercise we will predict the house price, based on
some characteristics of the house.
```

Answer : Use `loss='mean_squared_error'` because you want to predict the **HOUSE PRICE**, aka a numerical value, not a category

K-means clustering

KMeans Clustering has almost the same steps as all of them, but Standardize the features you want to use using StandardScaler()

Don’t do train-test split unless required from the prof

Define the data you want to cluster

```
# Convert to a Pandas DataFrame for easier exploration
X = pd.DataFrame(wine.data, columns=wine.feature_names)
y = pd.Series(wine.target)
```

```
# Standardize the features (important for KMeans since
it's distance-based)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Perform K-Means clustering
kmeans = KMeans(n_clusters=3, random_state=42)
clusters = kmeans.fit_predict(X_scaled)
```

The same output

```
# Compare the K-Means clusters to the actual wine classes
print("Confusion Matrix:\n", confusion_matrix(y, clusters))
print("\nClassification Report:\n", classification_report(y,
clusters))
```

If asked to visualise the clusters

```
#3c) Visualize the clusters (using PCA to reduce to 2 dimensions for
plotting)
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

plt.figure(figsize=(8,6))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=clusters, cmap='viridis',
marker='o', edgecolor='k')
plt.title('K-Means Clustering of Wine Dataset (PCA-reduced data)')
plt.xlabel('PCA 1')
plt.ylabel('PCA 2')
plt.colorbar(label='Cluster')
plt.show()
```

RandomForest

We have RandomForestRegressor and RandomForestClassifier

RandomForestRegressor()

```
# Features (input) and target variable
X = df.drop(columns=['csMPa'])
y = df['csMPa']

# Split the data into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Initialize and fit the RandomForestRegressor
rf_regressor = RandomForestRegressor(n_estimators=100, random_state=42)
rf_regressor.fit(X_train, y_train)
```

```
# Make predictions on the test set
y_pred = rf_regressor.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

```
print(f"Mean Squared Error: {mse}")
print(f"R² Score: {r2}")
```

```
# Visualize actual vs predicted
plt.scatter(y_test, y_pred, color='blue')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'k--',
lw=2)
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.title('RandomForestRegressor: Actual vs Predicted')
plt.show()
```

RandomForestRegressor can be used with GridSearchCV, look above to find it

RandomForestClassifier()

Do the train-test split for the data you use in cross validation

```
# Create a pipeline with scaling and Random Forest
rf = RandomForestClassifier(n_estimators=100, random_state=42) # Random
Forest classifier

# Perform cross-validation
scores_rf = cross_val_score(rf, X_train, y_train, cv=5, scoring='accuracy')
```

The random forest

```
print(f"\nRandom Forest:")
print(f"Mean Cross-Validation Accuracy: {scores_rf.mean():.4f}")
```

Fitting the model and getting our predicted data to use down

```
# Fit the model to the training set
rf.fit(X_train, y_train)

# Evaluate on the test set
y_pred_rf = rf.predict(X_test)
```

Data we want to get

```
print(f"Test Accuracy: {rf.score(X_test, y_test):.4f}")
print("Classification Report:\n", classification_report(y_test, y_pred_rf))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_rf))
```

Random Important Stuff

train-test split

with random_state and the training size of 80%

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
train_size=0.8, random_state=42)
```

The **stratify=y** makes sure that the *random data are picked equally* from each column, for the percentage of the split to be the same as the data

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42, stratify=y)
```

Pipelines

Scaling and Random Forest with 100 estimators

```
# Create a pipeline with scaling and Random Forest
pipeline_rf = Pipeline([
    ('scaler', StandardScaler()),
    ('random_forest_classifier',
RandomForestClassifier(n_estimators=100,
random_state=42))
])
```

Scaling and Decision Tree Classifier pipeline

```
# Create a pipeline with scaling and Decision Tree
pipeline_dt = Pipeline([
    ('scaler', StandardScaler()),
# Step 1: StandardScaler for scaling features
    ('decision_tree',
DecisionTreeClassifier(random_state=42))
# Step 2: Decision Tree Classifier
])
```

Create a *PolynomialRegression pipeline*, create the variables with ***PolynomialFeatures*** and feed them into the ***LinearRegression***

```
def PolynomialRegression(degree=2, **kwargs):
    return make_pipeline(PolynomialFeatures(degree),
LinearRegression(**kwargs))
```

Can also be done this way, recommended

```
polynomial_regression = Pipeline([
    ("polynomialfeatures", PolynomialFeatures()),
    ("linearregression", LinearRegression())
])
```

If asked to use a ***standard scaler***.

```
polynomial_regression = Pipeline([
    ("standard_scaler", StandardScaler()),
    ("poly_features", PolynomialFeatures(degree=2)),
    ("linearregression", LinearRegression())
])
```

btw be aware the cross validation score uses the negative mean squared error

A Pipeline with Scaler and Polynomial features and Ridge

```
model_pip =Pipeline([(
     'standard_scaler',StandardScaler()),
    ('poly_features', PolynomialFeatures(degree=2)),
    ('ridge_regression',Ridge())
])
```

Grid params

They wanted to test the polynomial degrees from 1 to 10, and for Ridge regression they wanted to try ridge factors of (1.0, 0.5, 0.1) for a GridSearchCV with ***PolynomialFeatures*** and ***Ridge Regression*** Pipeline

```
param_grid = {
    'poly_features__degree': np.arange(1, 10),
# Test polynomial degrees from 1 to 9
    'ridge_regression__alpha': [1.0, 0.5, 0.1]
# Ridge factors to test, if no Ridge() don't add it
}
```

They wanted to use this for a GridSearchCV with RandomForestRegressor model, this one is without Pipeline

```
param_grid = {
   'n_estimators': [10, 20,40, 50, 100,150, 200,250, 300,
350,450,500]   }
```

Output values for each model

Grid Search

```
# Get the best parameters and best score
best_params = grid_search.best_params_
best_score = grid_search.best_score_

print(f"Best Parameters: {best_params}")
print(f"Best Cross-Validation Score (Negative MSE):
{best_score}")

# Train the RandomForestRegressor using the optimal
number of trees
best_rf = grid_search.best_estimator_
```

Confusion matrix

```
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_dt))
```

Evaluating the model

```
# Make predictions on the test set
y_pred = best_rf.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Test Set Mean Squared Error: {mse}")
print(f"Test Set R² Score: {r2}")
```

Decision Tree Classifier

- Cross Validation Scores for Decision

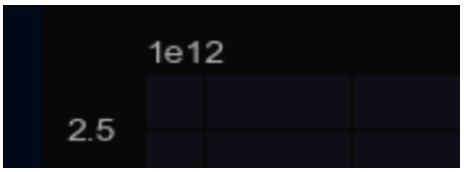
```
# Create a pipeline without scaling and Decision Tree
DTree =  DecisionTreeClassifier(random_state=42)

# Perform cross-validation
scores_dt = cross_val_score(DTree, X_train, y_train, cv=5,
scoring='accuracy')

print(f"Mean Cross-Validation Accuracy: {scores_dt.mean():.4f}")

print(f"Test Accuracy: {DTree.score(X_test, y_test):.4f}")

print("Classification Report:\n", classification_report(y_test,
y_pred_dt))

#other important metrics
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_dt))
```

Plot y-axis limit (plt.ylim)

```
plt.ylim(0, max(-np.mean(train_scores, axis=1).max(),
-np.mean(validation_scores, axis=1).max()) + 1000)
```

Or **plt.ylim(0, 1000)** in case you see the 1e12, or any big number at the top left

Not normal top left



Normal top left

the Actual vs Predicted question relationships

Creates a scatter plot with the data

```
plt.scatter(y_test, y_pred, color='blue')
```

Visualises a line for going for the data, predicted on y axis, x axis is actual

```
plt.plot([y_test.min(), y_test.max()], [y_test.min(),
y_test.max()], 'k--', lw=2)
```

Gaussian basis functions

Since i got that we don't need to create something similar from prof Saja you can use the following code if needed

```
from sklearn.base import BaseEstimator, TransformerMixin

class GaussianFeatures(BaseEstimator, TransformerMixin):
    """Uniformly spaced Gaussian features for one-dimensional input"""

    def __init__(self, N, width_factor=2.0):
        self.N = N
        self.width_factor = width_factor

    @staticmethod
    def _gauss_basis(x, y, width, axis=None):
        arg = (x - y) / width
        return np.exp(-0.5 * np.sum(arg ** 2, axis))

    def fit(self, X, y=None):
        # create N centers spread along the data range
        self.centers_ = np.linspace(X.min(), X.max(), self.N)
        self.width_ = self.width_factor * (self.centers_[1] -
self.centers_[0])
        return self

    def transform(self, X):
        return self._gauss_basis(X[:, :, np.newaxis], self.centers_,
                                 self.width_, axis=1)


gauss_model = make_pipeline(GaussianFeatures(20),
                            LinearRegression())
gauss_model.fit(x[:, np.newaxis], y)
yfit = gauss_model.predict(xfit[:, np.newaxis])

plt.scatter(x, y)
plt.plot(xfit, yfit)
plt.xlim(0, 10);
```

GridSearchCV with DecisionTreeClassifier params

They wanted a maximum tree depth from range 3 to 10, and split criterion equal to gini or entropy

```
grid_param = {'criterion' : ['gini', 'entropy'],
              'max_depth' : list(range(3,11))}
```

Normalising (Normalizing) data

We use the MinMaxScaler() to normalize the data

Why? : to make sure the data contribute equally to the model

A small example of us normalising the X_train data

```
scaler = MinMaxScaler()
x_train_norm = scaler.fit_transform(X_train)
```

Some Important metrics

```
acc = accuracy_score(y_true=y_tst, y_pred=y_tst_pred)
prec = precision_score(y_true=y_tst, y_pred=y_tst_pred, average='weighted')
rec = recall_score(y_true=y_tst, y_pred=y_tst_pred, average='weighted')
f1 = f1_score(y_true=y_tst, y_pred=y_tst_pred, average='weighted')
sll4 = precision_recall_fscore_support(y_true=y_tst, y_pred=y_tst_pred)
```

Important to consider stuff

The target column that you WANT TO PREDICT should be y
The features column that you WILL PREDICT WITH should be X

Make sure to check for the plot if it gives a weird number like **1e12**, use **plt.ylim()** and limit the plot height to a decent degree

- Transform cement feature into ***numpy array***
  - X = df[['cement']].to_numpy()

- ***Flatten a 2d array***, make 2d array to 1d array
  - X.ravel()

- To ***make evenly spaced numbers***, makes them into 2d array
  - X_test = np.linspace(X.min(), X.max(), 500)[:, None]

- **Scale the input using min_max scaling** (why do we do this?)
  - To make all the features contribute equally to the model

- Split the data in a train (85%) and test the dataset (15%) (why do we do this?)
  - A : to check overfitting. And to see the model performance

- Put the output in the right format. What is the name of the format?
  - X_train, X_test, y_train, y_test

IMPORTANT, do at least the exercises even if you cheat from the solution because it's impossible to understand them all, just make sure you understand whatever in here

At the end of this guide, pray for me to pass and goodluck

All the libraries seen in the exercises

All Libraries

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, cross_val_score,
validation_curve, GridSearchCV, RandomizedSearchCV
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor,
plot_tree
from sklearn.ensemble import RandomForestClassifier,
RandomForestRegressor
from sklearn.preprocessing import StandardScaler, MinMaxScaler,
PolynomialFeatures
from sklearn.pipeline import Pipeline
from sklearn.pipeline import make_pipeline
from sklearn.metrics import (classification_report, confusion_matrix,
ConfusionMatrixDisplay, accuracy_score,
                             precision_score, recall_score, f1_score,
precision_recall_fscore_support,
                             mean_squared_error, r2_score, silhouette_score)
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from tensorflow.keras import Model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
import math
%matplotlib inline
```

Classification

In Classification we want to categorise the data into values, like for example if animal is a cat or dog based on some stats or a photo

Let's start the example :

All the needed libs

```
import matplotlib.pyplot as plt
from IPython.core.display import display_png
from intake.source.cache import display
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.datasets import mnist
from tensorflow.keras import Model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.optimizers import Adam
from plot_loss import plot_loss # own function in plot_loss.py
```

The plot loss function, you can make it in plot_loss.py in the folder

```
import matplotlib.pyplot as plt

def plot_loss(history):
   plt.plot(history.history['loss'])
   if 'val_loss' in history.history:
       plt.plot(history.history['val_loss'])
       plt.legend(['Train', 'Val'], loc='upper left')
   plt.title('Model loss')
   plt.ylabel('Loss')
   plt.xlabel('Epoch')
   plt.show()
```

let's load the dataset, IMPORTANT this dataset is already split up so you don't need to do the train-test split, seen in the exam-prep

```
# Load the training and test dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
# 60000 training images with 28x28 pixels
print(x_train.shape)
print(y_train.shape)
```

We get the following results
(60000, 28, 28) : X_train and X_test are 3d arrays
(60000,)

Now what's the thing with this data?

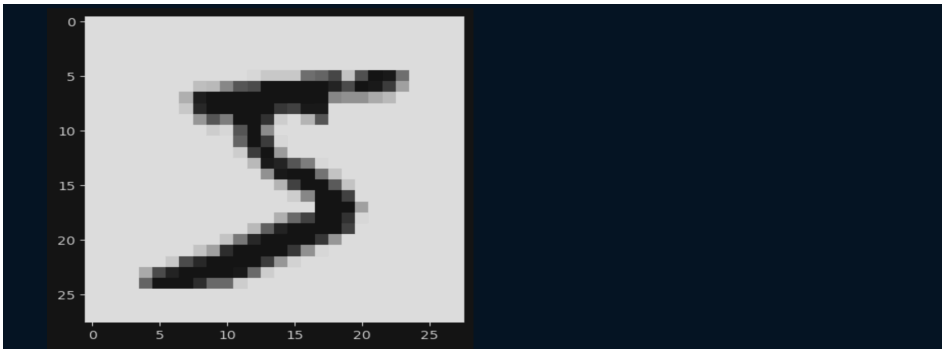
It's like a numerical representation of numbers in binary or something

Let's dive deeper into this :

The y value of the first image :
print(y_train[0]) # should output 5, it's the value of the x_train multi numbers

print(x_train[0]) # will output a big array 28 x 28 of some numbers that cannot be understood

plt.imshow(x_train[0], cmap='gray', vmin=0, vmax=255) # this will show those numbers of 28 x 28 into an image



Step 2 : Transforming the data

transform the data into 1d array to fit the input layer of our model

```
x_train = x_train.reshape(( -1, 784 ))
x_test = x_test.reshape(( -1, 784 ))
print(x_train.shape) # will output (60000, 784)
```

Why 784? Because 28 x 28 = 784, and check the top of this page to understand where 28 x 28 comes from, using .shape

2.1 : After that we normalize the data using MinMaxScaler, why? To make sure all the data contribute equally to the model by bringing them to a similar scale, like from [0,1]

scaler = MinMaxScaler()
x_train_norm = scaler.fit_transform(x_train)
x_test_norm = scaler.fit_transform(x_test)

2.2 : we use one-hot encoding for the output aka the (y) this means that the model will output 10 probabilities, (prob that a digit is 0 and prob that a digit is 1)

use **.to_categorical( )** when the class labels are integers, like the y column is numbers and not text,
print(y_test, y_train) # outputs [7 2 1 ... 4 5 6] [5 0 4 ... 5 6 8]

When the column names are in text use **pd.get_dummies()**
Example of pd.get_dummies( ) from gpt, we get one-hot encoding of columns color and size
df = pd.DataFrame({'color': ['red', 'blue', 'green'], 'size': ['S', 'M', 'L']})
df_encoded = pd.get_dummies(df, columns=['color', 'size'])

Now lets do the code

```
# one-hot encoding for output
y_train_onehot = to_categorical(y_train)
y_test_onehot = to_categorical(y_test)
print(y_train_onehot.shape)
print(y_train_onehot[0])
print(y_train.shape)
```

(60000, 10)
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
(60000,)

Now the most important part after we made our data ready

Part 3 : Define the model

We define the inputs : the input has 784 nodes, because 28 x 28 = 784

```
inputs = Input(shape=(784,))
```

We choose 2 hidden layers with 128 and 64 nodes and relu activation function, we can try other stuff but in the exam if not specified use these params

```
x = Dense(128, activation='relu')(inputs)
x = Dense(64, activation='relu')(x) # if 3 hidden layers do this again down
```

Output layer has 10 nodes, the same as nodes amount from the one-hot encoding, print(y_train_onehot.shape) (60000, 10) # nodes on right, btw the activation is ***softmax*** since we have 10 probabilities and the sum of them must be 1

```
outputs = Dense(10, activation='softmax')(x)
```

We Construct the model with our inputs and outputs we did above

```
model = Model(inputs, outputs, name='MNIST')
```

Print a summary of the model
model.summary()

Compile the model with prams, we want Adam optimizer with a learning rate of 0.001, *the loss is categorical_crossentropy because it's used for classification* using one-hot encoding as output used together with softmax activation function, we define accuracy metric for Classifications

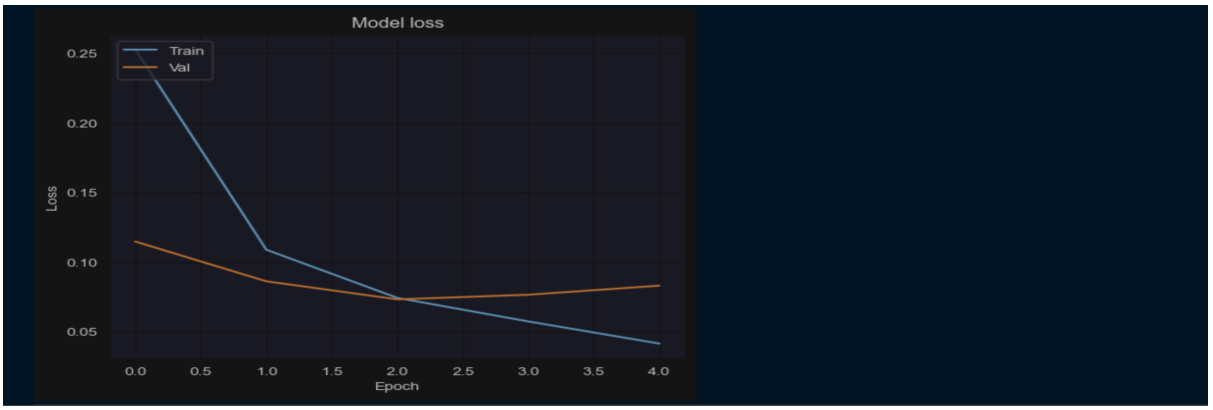
```
model.compile(
    optimizer=Adam(learning_rate=0.001),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

Step 4 : Train the model

We train the model using our training input data and training target we created in step 2, we train the model for 50 epochs with a batch size of 32 (we can try other values but if not defined in the exam use the below code), but important to take 10% of the training dataset for validation

```
history = model.fit(
    x_train_norm, # training input
    y_train_onehot, # training targets
    epochs=50,
    batch_size=32,
    validation_split=0.1, # 10% of the training data
)
# plot loss function this is in a separate plot_loss.py
file, the code to create it is above
plot_loss(history)
```

This is the resulting model



What does this model mean?

The training loss becomes lower : means the model is learning the data better with each epoch, in the first one there's a sharp drop means the model learned the data quickly

The validation loss drops a little and rises a little at epoch 2, the model has a possible overfit from epoch 3, meaning model starts to memorise data

Model learns good at the beginning but memorises the data later on

Output meaning

 # Epoch 1/5
Training Loss: 0.2504 - Indicates the initial error of the model on training data.

Training Accuracy: 92.64% - Shows the model correctly predicted 92.64% of the training data.

Validation Loss: 0.1093 - The model's error on validation data, lower than training loss (good generalization).

Validation Accuracy: 97.02% - Accuracy on validation data, indicating strong initial generalization.

 # Epoch 2/5
Training Loss: 0.1045 - Training loss has decreased, meaning the model is learning well.

Training Accuracy: 96.86% - Improved accuracy on the training data.

Validation Loss: 0.0874 - Validation loss decreased, indicating good generalization continues.

Validation Accuracy: 97.38% - Slight improvement in validation accuracy.

 # Epoch 3/5
Training Loss: 0.0718 - Further decrease in training loss.

Training Accuracy: 97.78% - Training accuracy continues to improve.

Validation Loss: 0.0861 - Validation loss is stable, showing model generalization.

Validation Accuracy: 97.43% - Small increase in validation accuracy.

 # Epoch 4/5
Training Loss: 0.0552 - Training loss continues to decrease, the model is learning.

Training Accuracy: 98.26% - Model achieves 98.26% accuracy on training data.

Validation Loss: 0.0909 - Validation loss has slightly increased, hinting at possible overfitting.

Validation Accuracy: 97.40% - Validation accuracy is stable.

 # Epoch 5/5
Training Loss: 0.0434 - Training loss decreases further, model is optimising well.

Training Accuracy: 98.61% - Improved training accuracy.

Validation Loss: 0.0805 - Validation loss decreases again, indicating improved generalisation.

Validation Accuracy: 97.95% - Validation accuracy improves to a strong 97.95%.

Step 5 : Evaluate the Model

```
# evaluate
model.evaluate(x_test_norm,y_test_onehot)
```

Outputs
[0.08251411467790604, 0.9761000275611877]

What do they mean? 0.0825 is the loss, 0.9761 is the accuracy

Can be written :

- Loss : 0.0825, NO percentage
- Accuracy : 97.61%, The Model correctly predicted 97.61% of test samples

Step 6 : Use the model to predict

Now after all that hell we can start using the model to predict our data, we want to use predict the first 2 numbers in the normalised data

```
# predict
predicted = model.predict(x_test_norm[:2])
print(predicted)
# what it should be
print(y_test[:2])
```

[[7.7462653e-10 2.1532708e-08 1.0148794e-06 1.6397362e-05
3.5830769e-11 1.2618911e-07 7.8251218e-13 9.9998045e-01
5.1869907e-08 1.8649442e-06]
[1.3941837e-12 1.3949767e-06 9.9999785e-01 6.5808183e-07
1.8889152e-16 5.5563715e-10 5.9035909e-10 1.7148238e-11
1.2394306e-09 1.5150514e-14]]
[7 2]

Our first to numbers are indeed 7 and 2

Regression

We use the Regression to predict values, like the price of a home based on it's equipment and the future price of groceries, in this example we will be looking at the fuel efficiency of miles per gallon MPG, this is a continuous variable, in contrast with a Classification model where a class is predicted

Importing the libs

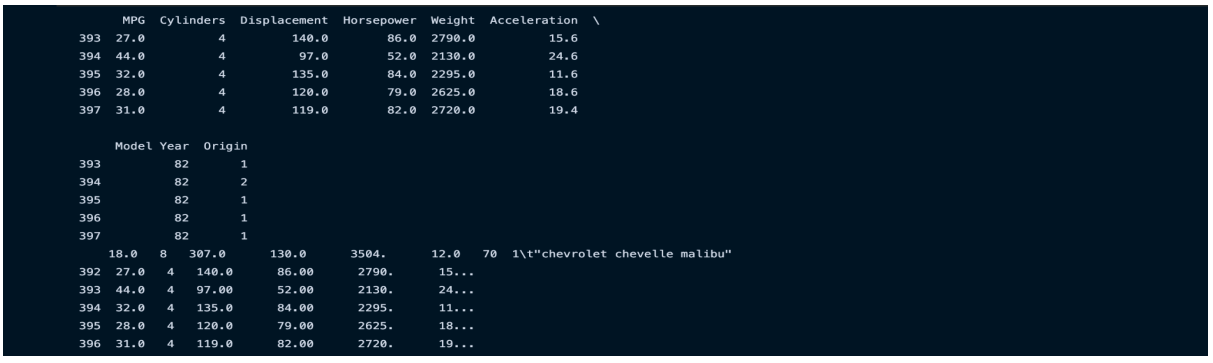
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras import Model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.optimizers import Adam
from plot_loss import plot_loss # own function in plot_loss.py
```

Step 1 : Loading the dataset

We read the dataset using pandas, we give column names because this one doesn't have any, prob..

```
column_names = ['MPG', 'Cylinders', 'Displacement', 'Horsepower',
'Weight', 'Acceleration', 'Model Year', 'Origin']
dataset = pd.read_csv("../data/auto-mpg.csv", names=column_names,
na_values='?', sep=' ', comment='\t', skipinitialspace=True)
```

Difference with data that has labels and the same data without



```
      MPG  Cylinders  Displacement  Horsepower  Weight  Acceleration  \
393  27.0          4         140.0        86.0  2790.0          15.6
394  44.0          4          97.0        52.0  2130.0          24.6
395  32.0          4         135.0        84.0  2295.0          11.6
396  28.0          4         120.0        79.0  2625.0          18.6
397  31.0          4         119.0        82.0  2720.0          19.4

     Model Year  Origin
393          82       1
394          82       2
395          82       1
396          82       1
397          82       1
    18.0   8   307.0      130.0      3504.      12.0   70  1\t"chevrolet chevelle malibu"
392  27.0   4   140.0       86.00      2790.      15...
393  44.0   4   97.00       52.00      2130.      24...
394  32.0   4   135.0       84.00      2295.      11...
395  28.0   4   120.0       79.00      2625.      18...
396  31.0   4   119.0       82.00      2720.      19...
```

Step 2 : data cleaning and normalisation

Check if the data has any na values, do it in both Classification and Regression

dataset.isna().sum() # to check which columns has how many NA's

In the above example we have 6 NA's in horsepower but it's an important column so we drop the rows and not the column, (you decide on which)

dataset = dataset.dropna()

We separate data into predictors and feature we want to predict (MPG)

```
x = dataset[["Cylinders","Displacement", "Horsepower",
"Weight", "Acceleration", "Model Year"]]
y = dataset[['MPG']]
```

x.describe() *gets the count, mean, standard deviation, min value, max value, 25%, 50%, 75%*

After that we normalise the inputs using the MinMaxScaler() it is good practice to always do this so features get the same scale

scaler = MinMaxScaler()
x_norm = scaler.fit_transform(x)

After that we split the data using the train-test split, we use 10% test data because it was specified, you can whatever specified

```
x_train,x_test,y_train,y_test =
train_test_split(x_norm,y,test_size=0.1)
```

Step 3 : Define the model

They wanted an input layer with *6 nodes* *(the same as predictors count)* we choose 2 hidden layers with 64 nodes and a relu activation function, the output layer has ***1 node*** (the same as column we want to predict count), for regression output a ***linear activation function*** is used, we choose ***Adam as optimizer*** with learning rate of 0.001, the loss is MSE which we often **only** use for regression models, we define a mean_absolute_precentage_error metric bcs it's regression

```
# input layer 6 nodes
inputs = Input(shape=(6,))

# 2 hidden layer
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)

# output layer
outputs = Dense(1, activation='linear')(x)

# construct the model to use
model = Model(inputs, outputs, name='auto-mpg')

# print a summary
model.summary()

# compile the model
model.compile(
    optimizer=Adam(learning_rate=0.001),
    loss='mean_squared_error',
    metrics=['mean_absolute_percentage_error']
)
```

Step 4 : Train the model

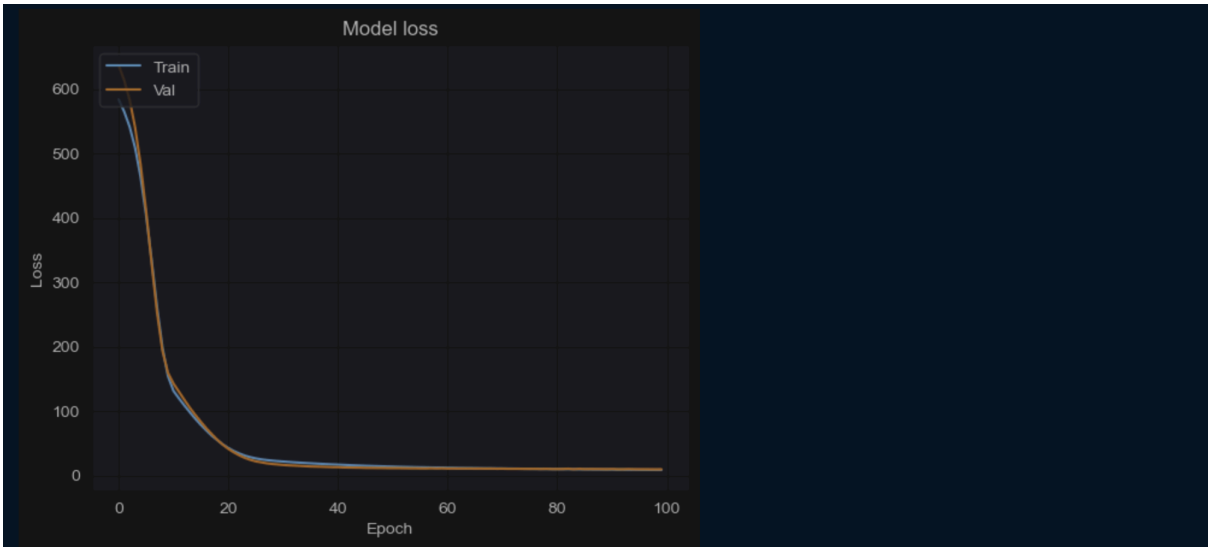
We train the mode for 100 epochs with a batch size of 32 we take 10% of the training dataset for validation data

```
history = model.fit(
    x_train, # training input
    y_train, # training targets
    epochs=100,
    batch_size=32,
    validation_split=0.1,
)
```

Plot the loss function

```
plot_loss(history)
```

the model we trained, using the function we have from the function that was created for us



Step 5 : evaluate the model

```
model.evaluate(x_test,y_test)
```

Will output : loss: 9.5111 - mean_absolute_percentage_error: 10.3670

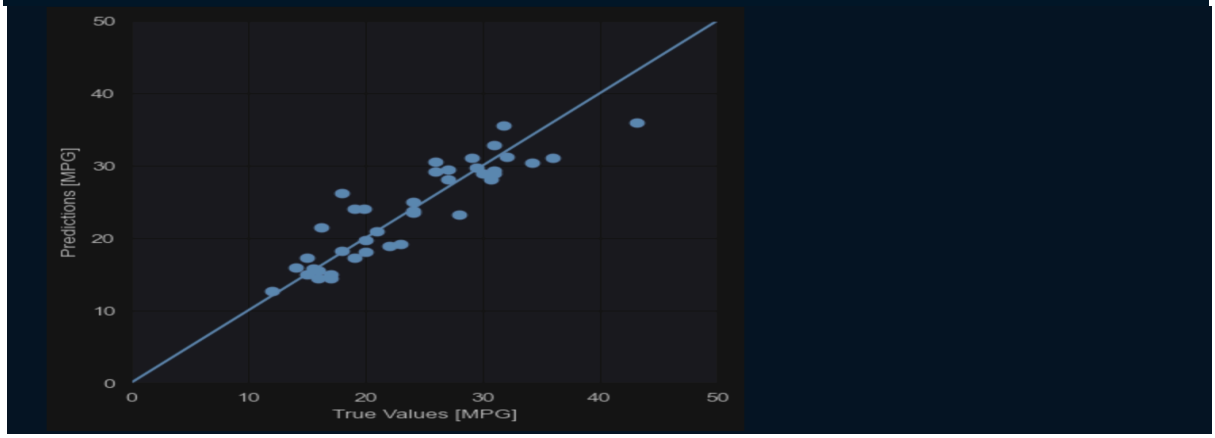
Step 6 : Use the model to predict

```
y_predicted = model.predict(x_test)
```

Final step : Plotting

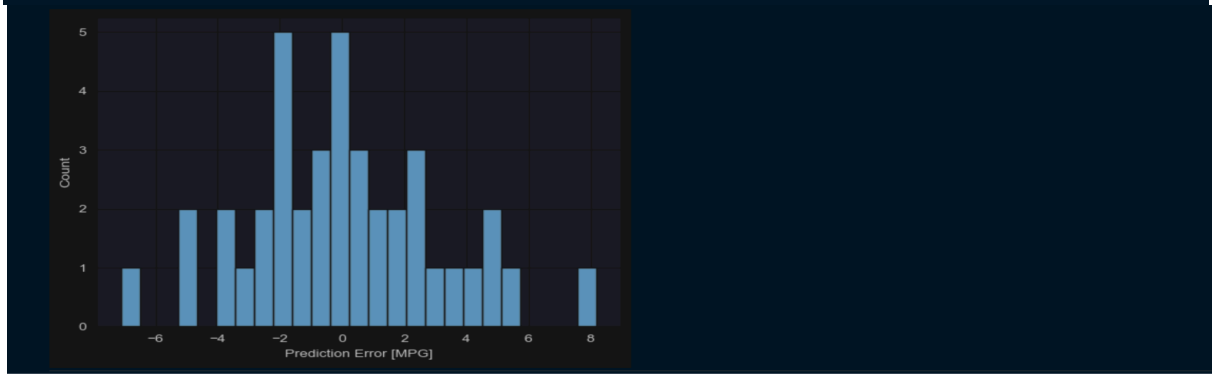
Plot the predicted values against the true value

```
a = plt.axes(aspect='equal')
plt.scatter(y_test, y_predicted)
plt.xlabel('True Values [MPG]')
plt.ylabel('Predictions [MPG]')
lims = [0, 50]
plt.xlim(lims)
plt.ylim(lims)
plt.plot(lims, lims)
```



We also plot the error distribution

```
error = y_predicted - y_test
plt.hist(error, bins=25)
plt.xlabel('Prediction Error [MPG]')
plt.ylabel('Count')
```



Neural Network Exercises

Exercise 1 :

This is an unfamiliar question with math, it's basic but they didn't teach it.

A very small NN has input layer with ***2 nodes*** and output layer with ***1 node***, there's ***no*** hidden nodes, the activation function used by the output is ***sigmoid***, use numpy arrays and python math module to calculate the output of the integration and activation function of the output node which of the 2 inputs has the biggest influence on the output?

This is basically pure math formula, if they drop this in the exam they want to catch us lacking, but i have it just in case

```
nodes = np.array([1,1])
weights = ([0.3, -0.1])
bias = -0.7

integration = np.sum(nodes * weights) + bias

activation = 1 / (1 + math.exp(-integration))


print(integration, activation)
```

-0.5 0.3775406687981454

Since the first input has a larger weight and is positive, it has a greater influence on the output compared to the second input.

Exercise 2 Classification :

The goal is to predict the quality class of wine, based on features that come from a chemical analysis of the wine.

So what we know now is that it's a classification and not regression since we want to predict a class which is a categorical variable

Load the data

We load the data and put the data and target in their places

```
wine = load_wine()
wine_x = load_wine().data
wine_y = load_wine().target
```

We make them into a DataFrame with names and a Series

```
wine_x = pd.DataFrame(wine_x, columns=wine.feature_names)
wine_y = pd.Series(wine_y)
```

Print the shape of each one to see what are you dealing with

```
print(wine_x.shape)# ( 178, 13)
print(wine_y.shape) # (178,) only 1 col, not df
```

Get the amount of null variables for each feature

```
print(wine_x.isna().sum())
```

Get the amount of classes and how many values are in them

```
print(wine_y.value_counts()) # 3 classes
```

Scale the input using min_max scaling (why do we do this?).

```
scaler = MinMaxScaler()

scaled_x = scaler.fit_transform(wine_x)
```

To make the data contribute to the model equally and uniformly.

Split the data in train 85% and test 15% (why do we do this?).

```
X_train, X_test, y_train, y_test = train_test_split(wine_x,
wine_y, random_state=1, train_size=0.85)
```

Put the output in the right format (what is the right format ?) :
the one-hot encoding for y_train and y_test

```
y_train_onehot = to_categorical(y_train)
y_test_onehot = to_categorical(y_test)

print(y_train_onehot.shape)
```

Define the model

First the inputs, they need to be the same as the shape of X columns in this case there's 13 columns in the wine_x

```
inputs = Input(shape=(13,))
```

We defined 2 hidden layers, they can be the same
Output : 3 nodes we want to predict, 3 categories

```
x = Dense(128, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)

outputs = Dense(3, activation='softmax')(x)
```

```
model = Model(inputs, outputs, name='WINE')
```

```
model.summary()


model.compile(
    optimizer=Adam(learning_rate=0.001),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

Train the model

```
model.fit(
    X_train,
    y_train_onehot,
    epochs=50,
    batch_size=32,
    validation_split=0.1
)
```

Evaluate the model

```
model.evaluate(X_test, y_test_onehot)
```

Predict with the model

Focus on here because it's important to not make this bad

```
There is a new wine with the following characteristics for the
features: 12, 6.5, 3, 25, 100, 2.5, 4, 0.5, 2, 8, 1, 3, 500. Predict
the quality for this wine. What should be the shape of the predict
parameter? Can we feed in the feature values right away? Which
quality class is predicted? How sure is the prediction?
```

So what do you see?

The features for the new wine we want to predict, **it's important to make it a 2D ARRAY**

new_wine = [[12, 6.5, 3, 25, 100, 2.5, 4, 0.5, 2, 8, 1, 3, 500]]

After making sure the values are in a 2D ARRAY, you need to transform them using MinMaxScaler(), we ONLY use .transform here to predict new data, NO fit_transform, that's why the data **needs** to be a 2d array

```
new_wine_scaled = scaler.transform(new_wine)

predicted = model.predict(new_wine_scaled)

print(predicted)
```

Ex 3 Regression :

Focus on what we want to predict

```
In this exercise we will predict the house price, based
on some characteristics of the house.
```

They want to predict a ***house price***, not a category, so regression

Import the data :

```
houses_x = fetch_california_housing().data
houses_y = fetch_california_housing().target
fetch_california_housing().DESCR
```

Data Preparation

Print the shape of houses_x and houses_y

```
# shapes
print(houses_x.shape) # (20640, 8)
print(houses_y.shape) # (20640,)
```

Check how many null values

```
np.isnan(houses_x).sum() # no null values
```

Scale the data using MinMaxScaler()

```
# scale
scaler = MinMaxScaler()

x_scaled = scaler.fit_transform(houses_x)
```

Split the Data into train-test

```
# split
X_train, X_test, y_train, y_test = train_test_split(x_scaled, houses_y,
random_state=42, train_size=0.90)
```

Define the model with hidden layers and output layer and inputs

```
inputs = Input(shape=(8,)) # same as count of columns in houses_x

x = Dense(128, activation='relu')(inputs) # hidden layer 1
x = Dense(64, activation='relu')(x) # hidden layer 2

outputs = Dense(1, activation='linear')(x) # output layer

model = Model(inputs, outputs, name='houses')


model.summary()

model.compile(
   optimizer=Adam(learning_rate=0.001),
   loss='mean_squared_error',
   metrics=['mean_absolute_percentage_error']

)

# Total params: 9,473
```

Train the model on the TRAIN data, epochs and batch_size, from you own

```
history = model.fit(
   X_train,
   y_train,
   epochs=50,
   batch_size=32,
   validation_split=0.1
)

plot_loss(history)
```

Evaluate the model on the TEST data

```
# evaluate

model.evaluate(X_test, y_test)
```

Use the model to predict a house with features: 3,30,5,1,1500,4,38,-125

```
house_pr = [[3,30,5,1,1500,4,38,-125]]

house_pr_scaled = scaler.transform(house_pr)

predict = model.predict(house_pr_scaled)

print(predict)
```