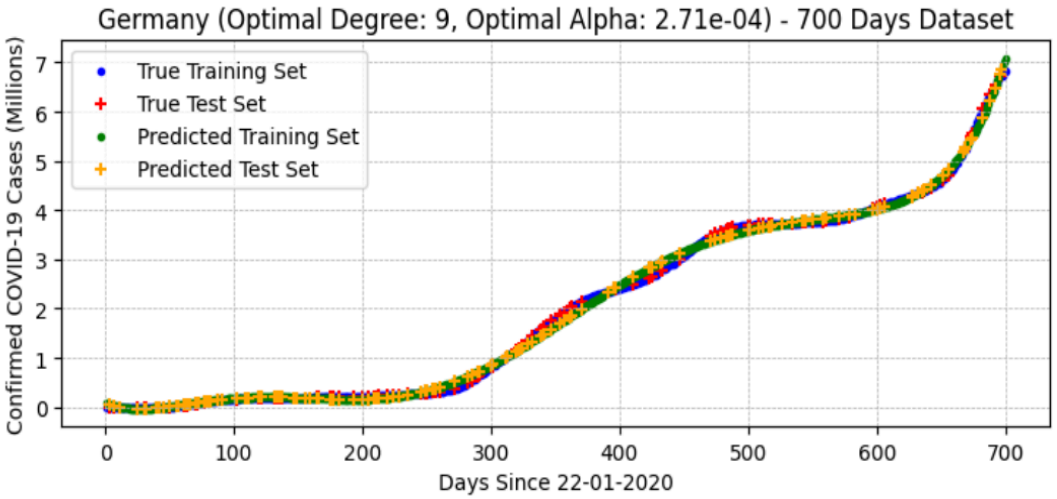
Assignment 1: Some common mistakes to avoid

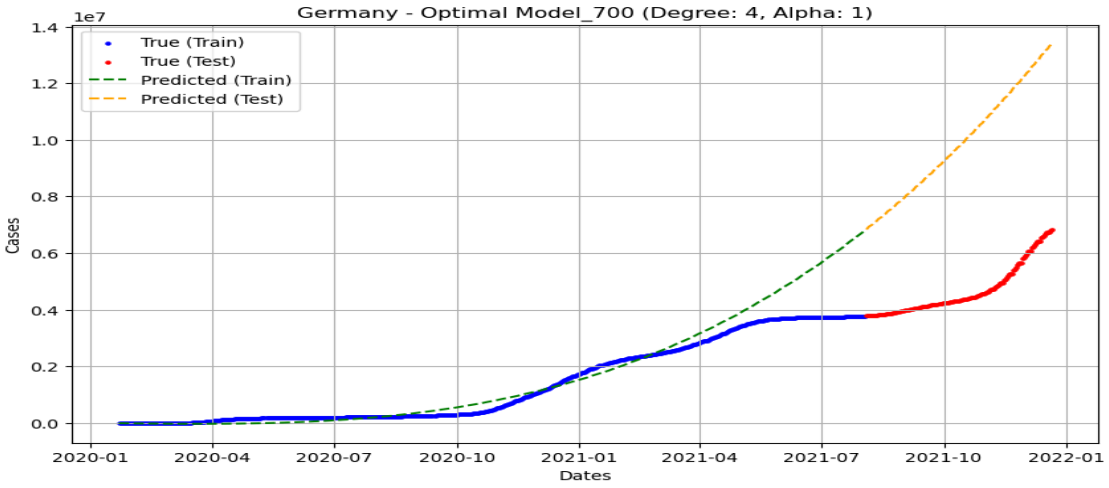
Intricacies of time series forecasting

Two common mistakes observed when handling the time series data were (1) the usage of shuffling when creating the train-test data split and (2) the usage of **Kfold** splits in the cross-validation loop.

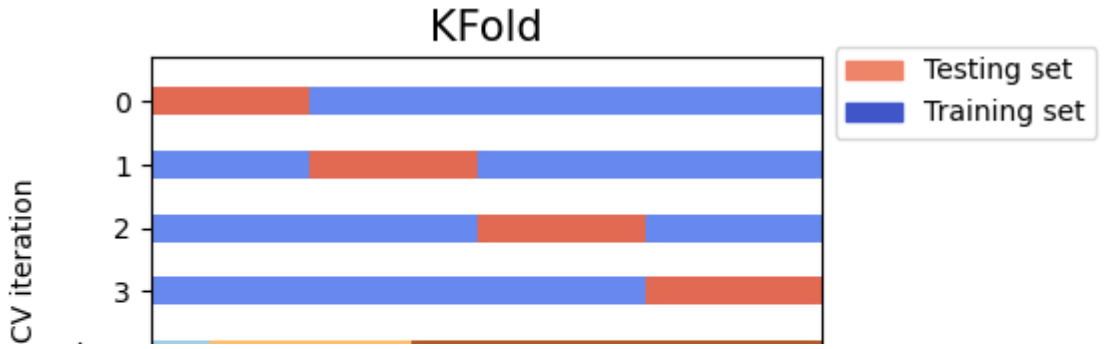
(1) When a shuffled train-test split is used, as showcased in the figure below,
The model will train on points that can be either in the future or past regarding the test set. Thus, the temporal ordering is broken, and your models can learn from future data points during training, leading to data leakage and overly optimistic performance estimates with reduced generalization. Notice how the polynomial regression model overfits Germany's covid curve.



When the test set follows the training set in time, we can achieve the correct implementation of time series forecasting, as illustrated in the figure below:

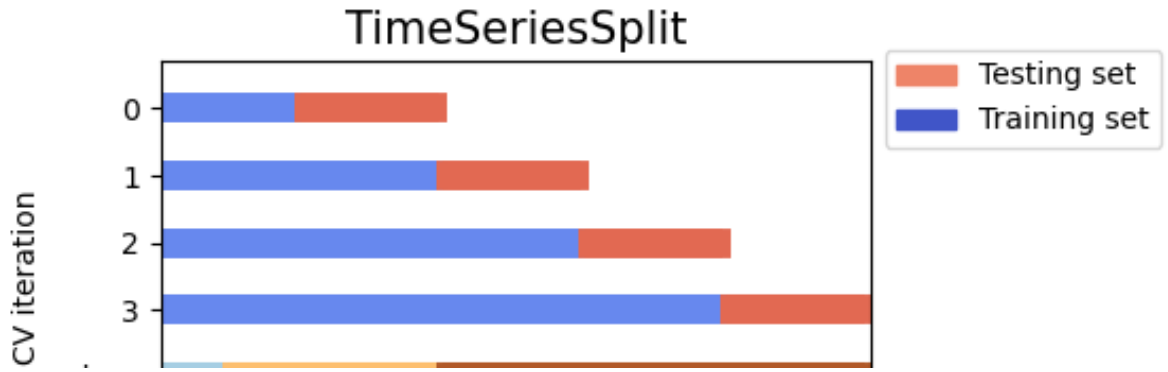


(2) Every student made a similar mistake during cross-validation. As you can observe from the figure below, taken from the sci-kit learn documentation, a KFold split was used to create the test-validation splits.



Splits with the KFold sci-kit learn object do not follow temporal ordering, and once again, the test training set will contain points occurring after the ones in the test set, leading to data leakage and biased models.

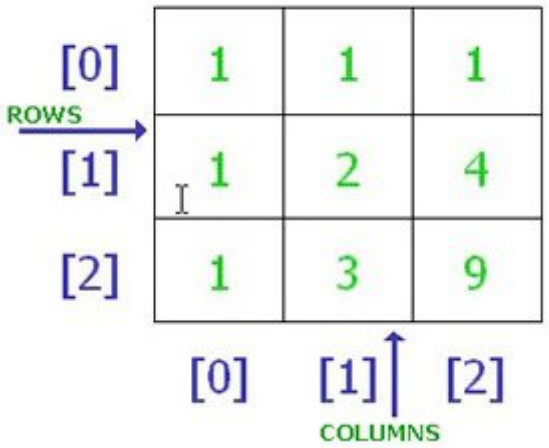
An alternative you can use is the TimeSeriesSplit sci-kit learn object, a variation of KFold fold that returns the first k folds as the train set and the remaining k+1 folds as the test set. Consequently, the training set always precedes the test set.



Feature scaling before the train test split is NOT a good idea

Another common mistake was applying feature scaling (e.g., MinMaxScaler, StandardScaler) to the entire dataset before splitting it into training and testing sets. Normalization and Standardization are column-wise operations, which means they use information from across samples to scale the data. Thus, if you apply it before splitting the dataset, the test set gets partially scaled based on the training set, causing data leakage and overly optimistic results. The standard practice is to perform feature scaling after the split by

fitting the scaler only to the training data and then transforming both the training and testing

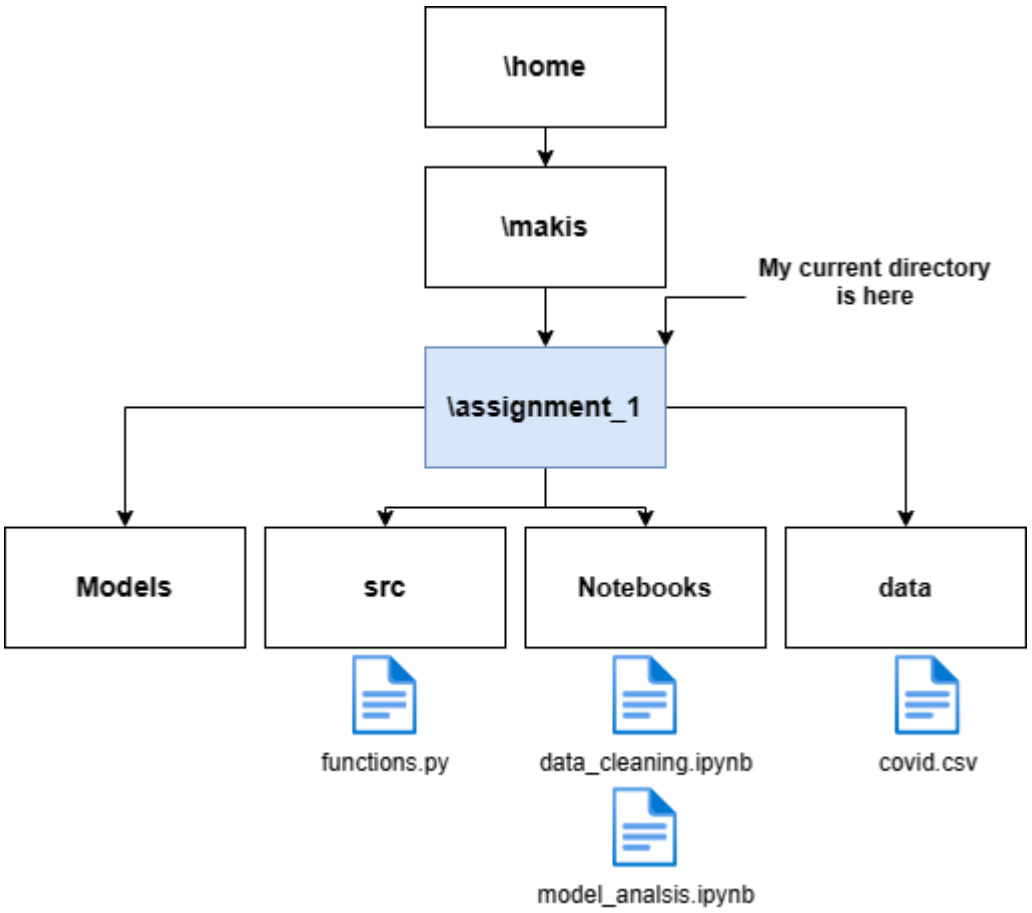


data accordingly:

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scale.transform(X_test)
```

Relative file paths are standard best practices.

A common mistake that caused many notebooks to throw an error and not run was the use of absolute file paths, which we had to change. Below is an illustration of your project directory “assignment_1” on the Linux computer of an imaginary user named “Makis”.



Absolute file paths are the full directory structure starting from the root directory ("/" for Linux or "C:" for Windows). On the other hand, relative file paths specify the path to a file or directory **relative to the current working directory**. When importing data, many students might have used an absolute path like `pd.read_csv("home\makis\assignment_1\data\covid.csv")`. But chances that I have also named my user directory makis and the code runs without a file-path error are pretty slim. It's better to use **relative to the project directory paths** *(assignment_1 in this case),* which will be the same for all users, like for example `pd.read_csv(".\assignment_1\data\covid.csv")`

For loops in Python for hyperparameter tuning and why you should avoid them

The majority of the students used a nested for loop for hyperparameter tuning to iterate through pairs of polynomial degree and alpha for ridge regression:

```
for degree in range(1,10):
        for alpha in [0.001, 0.1, 1, 10]:
                pass
```

while this is technically correct for tuning two parameters with small search spaces, it won't be able to scale when you need to search simultaneously more parameters in a larger space. Depending on how large your search space is and/or your computational power, you can consider these alternatives, which will always be a better option:

- GridSearchCV (when the search space/number of parameters is small)- which exhaustively checks all possible parameter pairs
- RandomizedSearchCV (when the search space/number of parameters is large) - which checks random groups of parameters with a specified number of iterations
- Optuna TPESampler (It has a steep learning curve, but it is by far the best solution) - which performs Bayesian hyperparameter tuning and modifies the search space based on the results of each trial

Importing your codebase

Usually, the simple Python command `from codebase import function_1, function_2` works fine to allow us to use function_1 and 2 developed in a file codebase.py in our current working file/notebook. However if this is not the case, that means that Python does not recognize “codebase.py” as a file that contains modules. You can alleviate this issue by Appending the file’s directory path to the system path:

```
import sys
sys.path.insert(0,'/path/to/codebasy.py_directory')
```

Saving and loading models to/from files

A final common mistake was the majority of saved models, while able to be loaded, were not functional. The issue stems from the fact that when the model is loaded it expects the input data to be in the shape it was trained on. e.g. if the optimal model was a 7-th degree polynomial. It will expect features in the shape (m x 7) and not in the format of the original dataset. There are two ways to circumvent this problem.

1. Create a deployment script as recommended in the assignment. This script should contain the data preprocessing steps before loading and deploying the model, making sure that the input data is appropriately formatted.
2. Alternatively, utilize a complete scikit-learn Pipeline as the model. When loaded and fitted, it automatically transforms the features as required.

Declutter your notebooks

Many libraries provide warnings as output. For example, NumPy will provide the following warning for an ill-conditioned matrix:

```
LinAlgWarning: Ill-conditioned matrix (rcond=6.12962e-22): result may not be accurate.
  return linalg.solve(A, Xy, assume_a="pos", overwrite_a=True).T
```

While helpful, if the code issuing the warning is inside a for loop, the above example statement will be printed as many times as the for loop iterations cluttering the notebook. We suggest after you finish your development and know which warning popped up, to import the warnings library and disable their printing so the final notebook looks clean:

```
import warning
warning.filter(“ignore”)
```