CHAPTER 1

Explore train data

You will work with another Kaggle competition called "Store Item Demand Forecasting Challenge". In this competition, you are given 5 years of store-item sales data, and asked to predict 3 months of sales for 50 different items in 10 different stores.

To begin, let's explore the train data for this competition. For the faster performance, you will work with a subset of the train data containing only a single month history.

Your initial goal is to read the input data and take the first look at it.

```python
# Import pandas
import pandas as pd

# Read train data
train = pd.read_csv('train.csv')

# Look at the shape of the data
print('Train shape:', train.shape)

# Look at the head() of the data
print(train.head())
```

Explore test data

Having looked at the train data, let's explore the test data in the "Store Item Demand Forecasting Challenge". Remember, that the test dataset generally contains one column less than the train one.

This column, together with the output format, is presented in the sample submission file. Before making any progress in the competition, you should get familiar with the expected output.

That is why, let's look at the columns of the test dataset and compare it to the train columns. Additionally, let's explore the format of the sample submission. The `train` DataFrame is available in your workspace.

```python
import pandas as pd

# Read the test data
test = pd.read_csv('test.csv')

# Print train and test columns
print('Train columns:', train.columns.tolist())
print('Test columns:', test.columns.tolist())
```

Determine a problem type

You will keep working on the Store Item Demand Forecasting Challenge. Recall that you are given a history of store-item sales data, and asked to predict 3 months of the future sales.

Before building a model, you should determine the `problem type` you are addressing. The goal of this exercise is to look at the distribution of the target variable, and select the correct problem type you will be building a model for.

The train DataFrame is already available in your workspace. It has the target variable column called "sales". Also, `matplotlib.pyplot` is already imported as `plt`.

Train a simple model

As you determined, you are dealing with a regression problem. So, now you're ready to build a model for a subsequent submission. But now, instead of building the simplest Linear Regression model as in the slides, let's build an out-of-box Random Forest model.

You will use the `RandomForestRegressor` class from the scikit-learn library.

Your objective is to train a Random Forest model with default parameters on the "store" and "item" features.

```python
import pandas as pd
from sklearn.ensemble import RandomForestRegressor

# Read the train data
train = pd.read_csv('train.csv')

# Create a Random Forest object
rf = RandomForestRegressor()

# Train a model
rf.fit(X=train[['store', 'item']], y=train['sales'])
```

Prepare a submission

You've already built a model on the training data from the Kaggle Store Item Demand Forecasting Challenge. Now, it's time to make predictions on the test data and create a submission file in the specified format.

Your goal is to read the test data, make predictions, and save these in the format specified in the "sample_submission.csv" file. The `rf` object you created in the previous exercise is available in your workspace.

Note that starting from now and for the rest of the course, `pandas` library will be always imported for you and could be accessed as `pd`.

```python
# Read test and sample submission data
test = pd.read_csv('test.csv')
sample_submission = pd.read_csv('sample_submission.csv')

# Show the head() of the sample_submission
print(sample_submission.head())
```

To determine whether a model is overfitting, say we are using the mean squared error (MSE, the lower the better) as the validation technique, when there is a huge gap between the train MSE and the validation MSE; then the model is overfitting.

Train XGBoost models

Every Machine Learning method could potentially overfit. You will see it on this example with `XGBoost`. Again, you are working with the Store Item Demand Forecasting Challenge. The train DataFrame is available in your workspace.

Firstly, let's train multiple XGBoost models with different sets of hyperparameters using XGBoost's learning API. The single hyperparameter you will change is:

`max_depth` - maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit.

```python
import xgboost as xgb

# Create DMatrix on train data
dtrain = xgb.DMatrix(data=train[['store', 'item']],
                     label=train['sales'])

# Define xgboost parameters
params = {'objective': 'reg:linear',
          'max_depth': 2,
          'verbosity': 0}

# Train xgboost model
xg_depth_2 = xgb.train(params=params, dtrain=dtrain)
```

Explore overfitting XGBoost

Having trained 3 XGBoost models with different maximum depths, you will now evaluate their quality. For this purpose, you will measure the quality of each model on both the train data and the test data. As you know by now, the train data is the data models have been trained on. The test data is the next month sales data that models have never seen before.

The goal of this exercise is to determine whether any of the models trained is overfitting. To measure the quality of the models you will use `Mean Squared Error (MSE)`. It's available in `sklearn.metrics` as `mean_squared_error()` function that takes two arguments: true values and predicted values.

train and test DataFrames together with 3 models trained (`xg_depth_2`, `xg_depth_8`, `xg_depth_15`) are available in your workspace.

```python
from sklearn.metrics import mean_squared_error

dtrain = xgb.DMatrix(data=train[['store', 'item']])
dtest = xgb.DMatrix(data=test[['store', 'item']])

# For each of 3 trained models
for model in [xg_depth_2, xg_depth_8, xg_depth_15]:
    # Make predictions
    train_pred = model.predict(dtrain)
    test_pred = model.predict(dtest)

    # Calculate metrics
    mse_train = mean_squared_error(train['sales'], train_pred)
    mse_test = mean_squared_error(test['sales'], test_pred)
    print('MSE Train: {:.3f}. MSE Test: {:.3f}'.format(mse_train, mse_test))
```

LINK FOR CHAPTER 1: https://github.com/TheophilusEmma/Kaggle_Comp._notes/blob/main/chapter1-Intro.pdf

CHAPTER 2

1. Understand the problem

In the previous chapter, we got acquainted with what Machine Learning competition actually looks like, and had an overview of the general competition process. Now it's time to start solving the problems!

1. Solution workflow

Before proceeding, let's take a look at the broad scheme that we'll be using throughout the subsequent chapters. Let's call it a 'solution workflow'. Typically it consists of four major stages. First, `we start by understanding the problem and the competition metric.`

1. EDA

Then we need to make some `EDA (exploratory data analysis) in order to see and understand the data we're working with`.

1. Local validation strategy

The next very important step is to establish the `local validation strategy`. We already know that its goal is to prevent overfitting.

1. Modeling

Finally, the longest part of the competition is Modeling, which includes `continuous improvements of the solution`. In this chapter, we will talk about the first three blocks. The third and fourth chapters are entirely devoted to Modeling.

> part1: Understand the problem

To understand the problem we need to perform the following steps. `Determine the data type we will be dealing with`. Is it the usual tabular data? Or maybe we're given time series data. Or it's unstructured data like images.Or text, and so on. It could be even a mix of multiple data types. In this course, we mostly concentrate on the tabular data and time series. No worries, the general solution workflow is the same for any data type.

The `next step is to determine the problem type`. We've talked about it a little in the previous chapter. Here we should select between `classification`, `regression`, `ranking` and so on.

Lastly, we should get familiar with the `metric being optimized`. As we already know, `every competition has a single metric`. It is used by Kaggle to evaluate the submissions and to determine the best performing solution.

> > Metric definition

Generally, the majority of the metrics can be found in the sklearn.metrics library. However, there are some special competition metrics that are not available in scikit-learn. In such cases, we have to create metrics manually. Suppose we're solving the competition problem with Root Mean Squared Logarithmic Error as an evaluation metric. This metric is not implemented in scikit-learn. Its formula is presented on the slide. N is the number of observations in the test set, y is the actual value, y hat is the predicted value. So, it is a usual Root Mean Squared Error in a logarithmic scale. In this situation, we have to define a custom function that takes as input the true and predicted values, and outputs the metric value. Firstly, we compute squares under the sum using numpy log and power methods. Finally, we get the square root of the mean over all the observations, and return the result.

The main takeaway from this lesson is that before building any models, we should perform some preliminary steps to understand the data and the problem we're facing. So, let's practice with other problem types and metrics!

Understand the problem type

As you've just seen, the first step of the solution workflow is to skim through the problem statement. Your goal now is to determine data types available as well as the problem type for the Avito Demand Prediction Challenge. The evaluation metric in this competition is the `Root Mean Squared Error`. The problem definition is presented below.

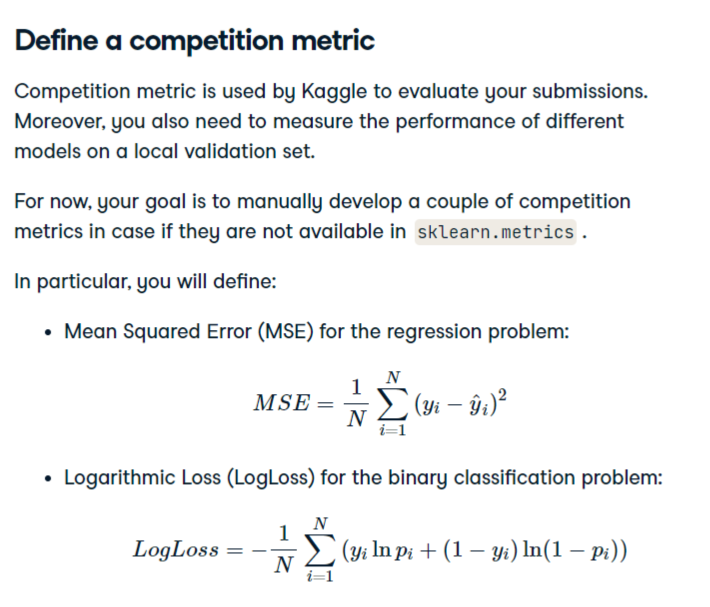
In this Kaggle competition, Avito is challenging you to predict demand for an online advertisement based on its full description (price, title, images, etc.), its context (geo position, similar ads already posted) and historical demand for similar ads in the past.

What problem type are you facing, and what data do you have at your disposal?

`ANSWER`

This is a regression problem with tabular, time series, image and text data

---




```python
import numpy as np

# Import MSE from sklearn
from sklearn.metrics import mean_squared_error

# Define your own MSE function
def own_mse(y_true, y_pred):
        # Raise differences to the power of 2
    squares = np.power(y_true - y_pred, 2)
    # Find mean over all observations
    err = np.mean(squares)
    return err

print('Sklearn MSE: {:.5f}. '.format(mean_squared_error(y_regression_true, y_regression_pred)))
print('Your MSE: {:.5f}. '.format(own_mse(y_regression_true, y_regression_pred)))
```

```python
import numpy as np

# Import log_loss from sklearn
from sklearn.metrics import log_loss

# Define your own LogLoss function
def own_logloss(y_true, prob_pred):
        # Find loss for each observation
    terms = y_true * np.log(prob_pred) + (1 - y_true) * np.log(1 - prob_pred)
    # Find mean over all observations
    err = np.mean(terms)
    return -err

print('Sklearn LogLoss: {:.5f}'.format(log_loss(y_classification_true, y_classification_pred)))
print('Your LogLoss: {:.5f}'.format(own_logloss(y_classification_true, y_classification_pred)))
```

Initial EDA

Now we know how to figure out what problem we're addressing, and how to use the appropriate metric. The next step is to look at the data and find interesting patterns in it using Exploratory Data Analysis (EDA for short).

Goals of EDA

EDA has multiple goals. To start with, we could get the size of the train and test data. It will give us an idea of how much resources we need for the competition and what models we could use. Then we could investigate the properties of the target variable. For example, there could be a high class imbalance in the classification problem, or a skewed distribution in the regression problem. Similarly, we could look at the properties of the features. Finding some peculiarities and dependencies between features and target variable is always useful. Also, EDA is a good place to start in order to generate some ideas and future hypotheses on feature engineering.

EDA statistics

As mentioned in the slides, you'll work with New York City taxi fare prediction data. You'll start with finding some basic statistics about the data. Then you'll move forward to plot some dependencies and generate hypotheses on them.

The `train` and `test` DataFrames are already available in your workspace.

```python
# Shapes of train and test data
print('Train shape:', train.shape)
print('Test shape:', test.shape)

# Train head()
print(train.head())

# Describe the target variable
print(train.fare_amount.describe())

# Train distribution of passengers within rides
print(train.passenger_count.value_counts())
```

EDA plots I

After generating a couple of basic statistics, it's time to come up with and validate some ideas about the data dependencies. Again, the train DataFrame from the taxi competition is already available in your workspace.

To begin with, let's make a `scatterplot` plotting the relationship between the fare amount and the distance of the ride. Intuitively, the longer the ride, the higher its price.

To get the distance in kilometers between two geo-coordinates, you will use Haversine distance. Its calculation is available with the `haversine_distance()` function defined for you. The function expects train DataFrame as input.

```python
# Calculate the ride distance
train['distance_km'] = haversine_distance(train)

# Draw a scatterplot
plt.scatter(x=train['fare_amount'], y=train['distance_km'], alpha=0.5)
plt.xlabel('Fare amount')
plt.ylabel('Distance, km')
plt.title('Fare amount based on the distance')

# Limit on the distance
plt.ylim(0, 50)
plt.show()
```

EDA plots II

Another idea that comes to mind is that the price of a ride could change during the day.

Your goal is to plot the median fare amount for each hour of the day as a simple line plot. The hour feature is calculated for you. Don't worry if you do not know how to work with the date features. We will explore them in the chapter on Feature Engineering.

```python
# Create hour feature
train['pickup_datetime'] = pd.to_datetime(train.pickup_datetime)
train['hour'] = train.pickup_datetime.dt.hour

# Find median fare_amount for each hour
hour_price = train.groupby('hour', as_index=False)['fare_amount'].median()

# Plot the line plot
plt.plot(hour_price['hour'], hour_price['fare_amount'], marker='o')
plt.xlabel('Hour of the day')
plt.ylabel('Median fare amount')
plt.title('Fare amount based on day time')
plt.xticks(range(24))
plt.show()
```

Local Validation

1. Local validation

After some preliminary steps, we come to one of the crucial parts of the solution process: local validation.

1. Motivation

Before we start, let's discuss the motivation for local validation. Recall the plot with possible overfitting to Public test data. The problem we observe here is that we can't detect the moment when our model starts overfitting by looking only at the Public Leaderboard. That's where local validation comes into play. Using only train data, we want to build some kind of an internal, or local, approximation of the model's performance on a Private test data.

1. Holdout set

The question is: how do we build such an approximation of the model's performance? The simplest way is to use a holdout set. We split all train data (in other words, all the observations we know the target variable for) into train and holdout sets.

1. Holdout set

We then build a model using only the train set and make predictions on the holdout set. So, the holdout is similar to the usual test data, but the target variable is known.

1. Holdout set

It allows to compare predictions with the actual values and gives us a fair estimate of the model's performance. However, such an approach is similar to just looking at the results on the Public Leaderboard. We always use the same data for model evaluation and could potentially overfit to it. A better idea is to use cross-validation.

1. K-fold cross-validation

The process of K-fold cross-validation is presented on the slide. We split the train data into K non-overlapping parts called 'folds' (in this case K is equal to 4).

1. K-fold cross-validation

Then train a model K times on all the data except for a single fold. Each time, we also measure the quality on this single fold the model has never seen before. K-fold cross-validation gives our model the opportunity to train on multiple train-test splits instead of using a single holdout set. This gives us a better indication of how well our model will perform on unseen data.

1. K-fold cross-validation

To apply K-fold cross-validation with scikit-learn, import it from the model_selection module. Create a KFold object with the following parameters: n_splits is the number of folds, shuffle is whether the data is sorted before splitting. Generally, it's better to always set this parameter to True. And random_state sets a seed to reproduce the same folds in any future run. Now, we need to train K models for each cross-validation split. To obtain all the splits we call the split() method of the KFold object with a train data as an argument. It returns a list of training and testing observations for each split. The observations are given as numeric indices in the train data. These indices could be used inside the loop to select training and testing folds for the corresponding cross-validation split. For pandas DataFrame it could be done using the iloc operator, for example.

1. Stratified K-fold

Another approach for cross-validation is stratified K-fold. It is the same as usual K-fold, but creates stratified folds by a target variable. These folds are made by preserving the percentage of samples for each class of this variable. As we see on the image, each fold has the same classes distribution as in the initial data. It is useful when we have a classification problem with high class imbalance in the target variable or our data size is very small.

1. Stratified K-fold

Stratified K-fold is also located in sklearn's model_selection module. It has the same parameters as the usual KFold: n_splits, shuffle and random_state. The only difference is that on top of the train data, we should also pass the target variable into the split() call in order to make a stratification.

K-fold cross-validation

You will start by getting hands-on experience in the most commonly used K-fold cross-validation.

The data you'll be working with is from the "Two sigma connect: rental listing inquiries" Kaggle competition. The competition problem is a multi-class classification of the rental listings into 3 classes: low interest, medium interest and high interest. For faster performance, you will work with a subsample consisting of 1,000 observations.

You need to implement a K-fold validation strategy and look at the sizes of each fold obtained. train DataFrame is already available in your workspace.

```python
# Import KFold
from sklearn.model_selection import KFold

# Create a KFold object
kf = KFold(n_splits=3, shuffle=True, random_state=123)

# Loop through each split
fold = 0
for train_index, test_index in kf.split(train):
    # Obtain training and testing folds
    cv_train, cv_test = train.iloc[train_index], train.iloc[test_index]
    print('Fold: {}'.format(fold))
    print('CV train shape: {}'.format(cv_train.shape))
    print('Medium interest listings in CV train: {}\n'.format(sum(cv_train.interest_level == 'medium')))
    fold += 1
```

LINK FOR CHAPTER 2: https://github.com/TheophilusEmma/Kaggle_Comp._notes/blob/main/chapter2%20(1).pdf