Building a Machine Learning Pipeline

Some notes to guide you during the workshop

1. A fair way to evaluate our model

In case you didn't notice, we've just evaluated our model with the exact same data it was trained with. This is, for obvious reasons, not a fair way to evaluate how well it did. In theory, the model could be simply a bit database, recording every instance it saw, returning the result of each instance as they are presented to it at test time. Would that be a good model?

The first cardinal sin of machine learning is to evaluate your model in the same data you trained it with! How can you know how well it will behave in the future if you never gave it unseen data?

The most basic way of dealing with this is to do a train-test split. You'll see it used in the cells below the note that brought you here.

2. Why is the score so high when using a decision tree?

Such a good performance in your first attempt at a problem really should raise some eyebrows.

One of the best and fastest ways to debug your data science pipeline is to inspect your models; see what they are really doing.

Decision Trees (DT) are very useful in this regard (most linear models too), that's why they are very good models for a first approach to almost any supervised ML problem. Namely, sklearn provides us with a mechanism to inspect which features the DT is looking at, and how important they are in the decision the DT is making. This is called "feature importance".

If you add the following lines to your code, right after the training of your DT, you'll be able to check what features your DT is really looking at.

```
df_imp = pd.DataFrame(
    dt.feature_importances_,
    columns=["importance"]
)
df_imp.index = COL_NAMES
df_imp.sort_values(by="importance", ascending=False, inplace=True)
```

This creates a dataframe that stores the feature importance, according to your DT, of each feature in the dataset.

If you look at the feature importance table you just built (e.g. by running the following line of code: `df_imp.head(10)`), you'll see something like this: |feature|importance| |---|---| away__suffered|0.356907 away__scored|0.227553 home__scored|0.224549 home__suffered|0.187731 away__avg_suffered_5|0.003259 away__avg_scored_5|0.000000 away__avg_suffered_3|0.000000 away__avg_scored_3|0.000000 away__delta_minus_4|0.000000 away__delta_minus_3|0.000000

Features with a higher score are considered more important by your model. This clearly shows that the model was quite shrewd in understanding that the features on the top rows are indeed very informative about the result of the game (please check the dictionary if you're not sure what they are). **In fact, they are all you need to know in order to accurately predict the result of the game!**. After all, you should be able to infer the result of a soccer game if you know how much goals the visiting and home teams scored. These should not be listed as features at all -- if you're trying to predict the result of a soccer game, you won't know the number of goals scored in advance, will you?

There is, however, another pertinent question to ask here:

If the DT figured out so fast how these "illegitimate" features are so informative about the target variable, why didn't the other models do the same?

To understand this we have to look at the data we are providing to our models. Lets look at a few statistics about the data we provided to our models. By running the line:

```
df[COL_NAMES].astype(float).describe()
```

You should be presented with an output quite similar to what we see below:

	home__val	away__val	home__scored	home__suffered	home__avg_scored_3	home__avg_suffered_3
count	3042.000000	3042.000000	3042.000000	3042.000000	3042.000000	3042.000000
mean	46780.108481	46900.460224	1.450033	1.089086	1.086675	1.280517
std	77869.625111	78471.921279	1.274920	1.113555	0.855126	0.802040
min	**0.000000**	**0.000000**	**0.000000**	**0.000000**	**0.000000**	**0.000000**
25%	0.000000	0.000000	1.000000	0.000000	0.333333	0.666667
50%	15350.000000	14950.000000	1.000000	1.000000	1.000000	1.333333
75%	26380.000000	28650.000000	2.000000	2.000000	1.666667	1.666667
max	**314700.000000**	**314700.000000**	**10.000000**	**8.000000**	**5.666667**	**5.333333**

The relevant rows for the analysis we're doing here are highlighted in bold. In particular, if you look at the minimum and maximum values for each column, you'll see that the **xxx_val** features have a very different range from the other variables!

What we see here is evidence that this discrepancy in ranges has little to no influence on the DT model, while it severely hinders the predictive capability of KNN and our Linear classifier. So much so that they couldn't even make use of the features that explicitly told the result.

Linear Model

The linear model works by multiplying the features (home__val and away__val included!) by learned coefficients. These coefficients are usually initialized to have values near to zero for all features and therein lies the problem. The maximum value of some features is so huge in this case, that even a small coefficients will make the model overshoot and give an answer that doesn't quite take in consideration any other feature. This is giving a disproportionate "feature importance" to such features, making the learning process really hard.

KNN

KNN works in a very different way, but it too is very sensitive to feature values. KNN tries to find instances that are similar, by measuring the distance between the points in it's database and the ones being predicted. To understand the impact, consider the following three instances:

instance	home__val	away__val	away__suffered	away__scored	result
#1	250 000	250 000	1	0	1
#2	550 000	550 000	1	0	1
#3	250 000	250 000	0	2	-1

In both instance #1 and #2 the visiting team lost the game by conceding one goal and scoring None, the home team won. In instance #3 the visiting team scored two goals and didn't concede a single one, in other words, the home team lost.

However, instance #1 is way closer to instance #3 than it is to instance #2 because the value of the teams is very similar(the same actually), while it is 250k units away from #2. Therefore, KNN will likely assign the wrong result when trying to predict instance #2 from instance #1 and #3.

Decision Tree

Decision trees are not very sensitive to these kinds of problems. All they do is, in broad terms, is to break up the feature space in chunks that are as uniform as possible in terms of the target variable, usually by splitting each feature range in two mutually exclusive and collectively exhaustive sub-ranges. They don't care much about the minimum and maximum values for each variable.

Fixing our previous mistakes

So, we have diagnosed two distinct problems here! Firstly, we noticed that some data being provided to ours models shouldn't be there at all, and secondly, we understood that we need to preprocess our data when working with some of models, namely the Linear Model and the KNN.

Preprocessing our data

Preprocessing your data is paramount. Different models require different types of preprocessing. For our job, making sure all our features have zero mean and a unit std will do the job.

For this, we will use a StandardScaler. Adding the following lines after your train, test split might do the job:

```
 scaler = StandardScaler()
X_tr = scaler.fit_transform(X_tr)
X_te = scaler.transform(X_te)
```

If you make no further changes, you'll notice that now your models are all preforming quite better.

Removing the "illegitimate" features

To remove the features that shouldn't really be there you should comment the line that adds them to the feature list **and remove the remaining or** on the last clause. Your COL_NAMES variable should be defined in the following way:

```
COL_NAMES = [
    x for x in df.columns if
    x in ["home__val", "away__val"] or
    "__avg_" in x or
    "delta_minus" in x
]
```

If you run your code again, you should be presented with a more realistic assessment performance of your models!

3. Predicting the future using the past

When using instances collected at different points in time we have to be extra careful about how to validate our models. We don't want to teach our model how to predict the result of a soccer game by looking at data that comes from the future, right? What use would that have?

When we split our data into train and test sets, in order to be able to "fairly evaluate our models' performance", we were a bit sloppy. Let me explain:

It might be the case that we trained our model with data from, lets say, the third game in the season, and then evaluated the same model with data from the first game of the same season. Is this really fair? Does this really resemble the situation the model will be in when trying to predict the matches this weekend? Certainly not!

The impact of this problem differs depending on the features and models we use. There are models that are really good at quite literally "memorizing" the data you fed them, and to use it in future predictions, making them perform really well at test time, only for you to discover the model is useless when put to work in the real world.

Scikit-learn has a couple of methods to help us validate our models when dealing with time series.

If you replace the train test split code with the following lines, you'll have in place the most basic for of train test split you could use while making sure you don't train your models with data from the future.

In order to put in place a suitable cross validation scheme that tackles our concerns, you should replace the old cross validation scheme with these lines of code:

```
 splitter = TimeSeriesSplit(n_splits=2)
tr_ind, te_ind = (*splitter.split(X),)[0]
X_tr, y_tr = X.loc[tr_ind], y.loc[tr_ind]
X_te, y_te = X.loc[te_ind], y.loc[te_ind]
```

Please notice that this will only work if the data in your dataframe is already ordered, which it should be, because it is being ordered at the beginning of the script.

Notice that the Linear Model has almost no decrease in performance, while the other two models had a significant decrease in performance. We've just fixed another issue in our pipeline.

4. Optimizing our models' parameters

Our models are working now with the default hyperparameter, but why use the default values if we can do better?

When optimizing our models' hyperparameters we might make hundreds or thousands of tries. This could arguably be done by finding the best parameter that, when trained on the training set, yield the best results on the test set. This, however, has a problem: In a sense, we're training the model parameters (e.g. coefficients) on the training set and the model hyperparameters in the test set, while at the same time trusting the last one to provide us with an accurate account of how well the model did. This is a problem similar to what we have before: the hyperparameters we found are the best for the test set we have, but is it proof that they are the best in new, unseen data ? Not really.

A simple way to mitigate this effect is to use k fold cross validation. This simple mechanism allows us to test our hyperparameters in a bunch of different test sets (here they are really called validation sets). We choose the set of hyperparameters that are, on average, better on them, giving us more confidence in the hyperparameters chosen. We'll also keep a separate (true) test set where we'll measure the performance of our best model.

The way to define the hyperparameters to search and to get the test set is as follows:

```
 X_tr, y_tr = X[:-1000], y[:-1000]
X_te, y_te = X[-1000:], y[-1000:]

splitter = TimeSeriesSplit(n_splits=3)

param_grid = {
    "min_samples_split": uniform(loc=2, scale=8),
    "min_samples_leaf": uniform(loc=1, scale=5),
    "max_depth":uniform(loc=3, scale=40)
}

p_sampler = ParameterSampler(
    param_grid,
    n_iter=100,
    random_state=RANDOM_STATE
)
```

This defines the parameters where the search will be made and reserves the 1000 most recent games to be used as a test set for our optimized model. We'll be using the DT model here, but you're welcome to test this approach in any of the other models.

The following bunch of code actually does the search and stores the result in a dataframe:

```
data = []
for p in p_sampler:
    params = {k:int(v) for k, v in p.items()}
    res = []
    for tr_ind, val_ind in splitter.split(X_tr):
        X_tr_fold, y_tr_fold = X_tr.iloc[tr_ind], y_tr.iloc[tr_ind]
        X_te_fold, y_te_fold = X_tr.iloc[val_ind], y_tr.iloc[val_ind]
        dt = DecisionTreeClassifier(
            random_state=RANDOM_STATE,
            **params
        )
        dt.fit(X_tr_fold, y_tr_fold)
        res.append(dt.score(X=X_te_fold, y=y_te_fold))
    dt = DecisionTreeClassifier(
        random_state=RANDOM_STATE,
        **params
    )
    dt.fit(X_tr, y_tr)
    data.append(
        {
            "score":np.mean(res),
            **params,
            "model":dt
        }
    )
df_search = pd.DataFrame(data).drop_duplicates(subset=[*param_grid,])
df_search.sort_values(by="score", inplace=True, ascending=False)
df_search.head()
```

Finally, you'll want to get your best model and see how well it really scores on the test set, which you can do using the following line of code:

```
df_search.loc[0].model.score(X_te, y_te)
```

5. Coming up with better features

There's only so much one can predict with a poor set of features. Big improvement can often be achieved in prediction tasks by engineering better features. Maybe you could look at the dataset generating scripts (*prepare.py*) and come up with better features of your one.

6. From scripts to pipeline

Our pipeline is made of a couple steps: * Scraping for data * Processing data * Training your model * Making predictions

With our current setup, we can do all this manually by running the scripts one by one. This is not, however the most scalable of solutions. Maybe we could use an ML Pipelining framework to help us with that. For this, we'll use Airflow.

Airflow is a pipelining framework started by Airbnb. It has all the essential mechanisms necessary to make ML pipelines.

Airflow pipelines are defines as DAGS, in code. In the code that was provided to you, you should see an example *dag* folder with an example. You also have a series of .py file on the code folder that when run, perform the various steps required for the pipeline.

These files scrape for data, prepare the data, and train a model (DT) with it. It's your job to setup a pipeline (DAG) to run them, in the right order, so that your model gets updated whenever new data is available. Feel free to try other models, or even combinations of them.

If you have the time you should add a new step to the pipeline that uses the model that you've just fitted to generate predictions for the upcoming games. There's a csv that contains these games already.