Published in Towards Data Science

Ler Wei Han Follow
Dec 8, 2019 · 5 min read · Listen

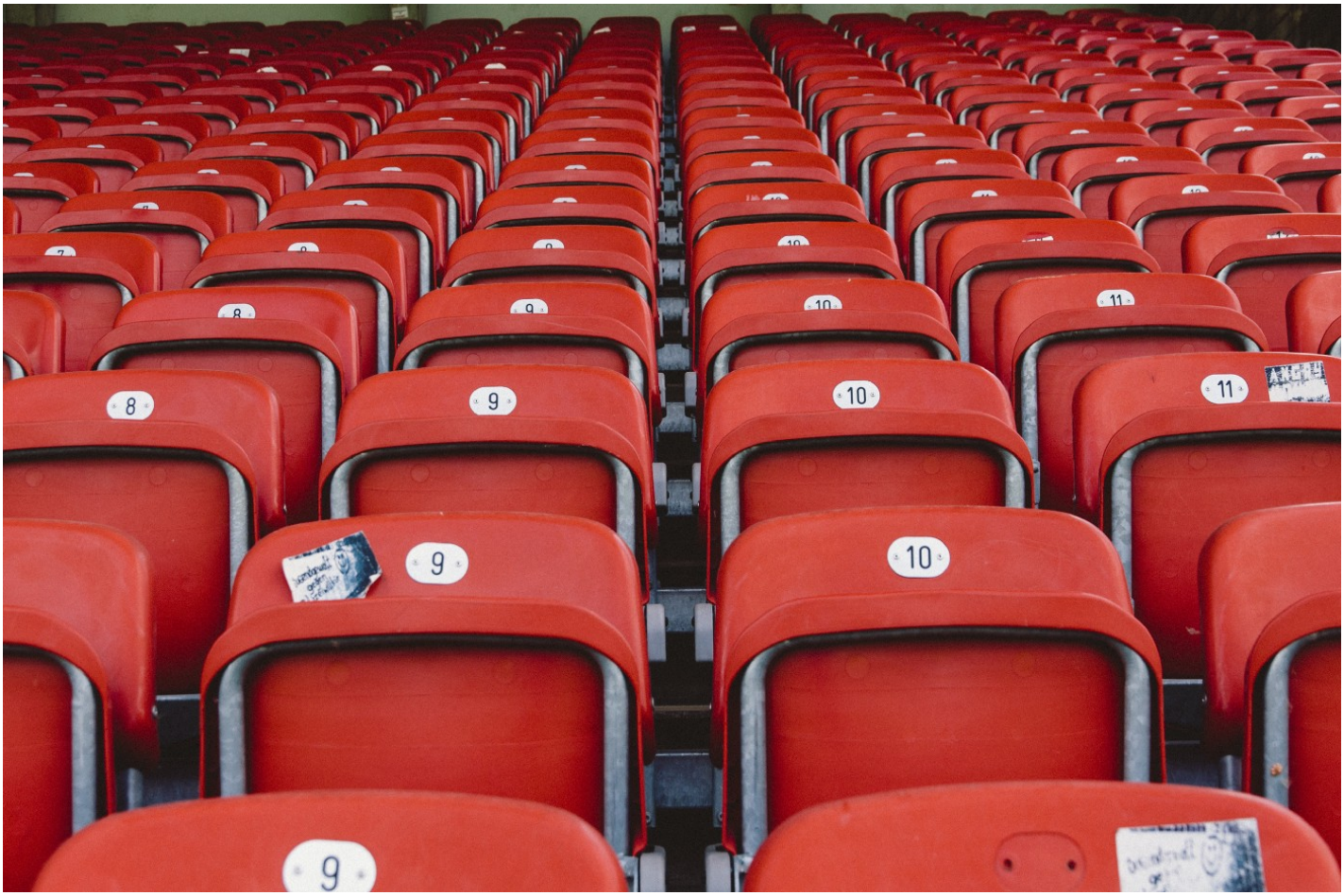


Photo by Simon Basler on Unsplash

Manipulating machine learning results with random state

Understanding the effect of random states on model outcome

Tuning hyperparameters, performing the right kind of feature engineering, feature selection etc are all part of the data science flow for building your machine learning model. Hours are spent tweaking and modifying each part of the process to improve the outcome of our model.

However, there is one argument nested within the most popular functions in data science that can be altered to change your machine learning results.

.....and it has nothing to do with domain knowledge or any of the engineering you have done on your data.

Random State

manipulation to the random permutation of the training data and the model seed, anyone can artificially improve their results.

In this article, I would like to gently highlight an often overlooked component of most data science projects — **random state** and how it affects our model outputs in machine learning.

So how does random state affect the classifier output?

To show how this affects the prediction result, I will be using Kaggle's famous Titanic dataset to predict the survival of the passengers.

Using the train dataset, I have applied some bare minimum data cleaning and feature engineering just to get the data good enough for training. I will be using a typical grid search cross validation with the xgboost classifier model for this example.

Training data that I will be using:

```
In [3]: X_train.head()
```

Out[3]:

PassengerId	Pclass	Name	Sex	Age	SibSp	Parch	Fare	Embarked
1	3	11	1	22.0	1	0	7.2500	3
2	1	12	0	38.0	1	0	71.2833	1
3	3	8	0	26.0	0	0	7.9250	3
4	1	12	0	35.0	1	0	53.1000	3
5	3	11	1	35.0	0	0	8.0500	3

Data.ipynb hosted with ❤ by GitHub view raw

Using grid search to find the optimal xgboost hyperparameters, I got the best parameters for the model.

```
In [12]: #performing grid search to find best hyperparameter
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from xgboost import XGBClassifier
XGB = XGBClassifier(random_state=0)
```

```
grid_search_XGB = GridSearchCV(XGB,
                               param_grid=parameter_grid,
                               cv=cross_validation,
                                n_jobs=-1,
                                verbose=0)

grid_search_XGB.fit(X_train, Y_train)
print('Best score: {}'.format(grid_search_XGB.best_score_))
print('Best parameters: {}'.format(grid_search_XGB.best_params_))
```

```
Best score: 0.8249158249158249
Best parameters: {'colsample_bytree': 1.0, 'gamma': 0.5, 'max_depth': 4, 'min_child_weight': 1, 'subsample': 0.8}
```

RandomState1.ipynb hosted with ❤ by GitHub view raw

Based on the cross validation result, my best performance achieved is 82.49% and the best parameters are:

```
'colsample_bytree': 1.0, 'gamma': 0.5, 'max_depth': 4, 'min_child_weight': 1, 'subsample': 0.8
```

This process is a staple to many machine learning projects: search through a range of hyperparameters to get the best averaged cross validation result. At this time, the work is considered done.

After all, cross validation should be robust to randomness…. right?

Not quite

For data science tutorials or showcases of results in Kaggle kernels, the notebook would have ended right there. However I would like pull back the previous workflow to show how the result differs with different random states.

In [13]:

```
for i in np.arange(0,5):
    XGB = XGBClassifier(random_state=i)
    parameter_grid = {'min_child_weight': [1, 5, 10],
        'gamma': [0.5, 1, 1.5, 2, 5],
        'subsample': [0.6, 0.8, 1.0],
        'colsample_bytree': [0.6, 0.8, 1.0],
        'max_depth': [3, 4, 5]}

    cross_validation = StratifiedKFold(n_splits=5,random_state=0,shuffle=True)

    grid_search_XGB = GridSearchCV(XGB,
                           param_grid=parameter_grid,
                           cv=cross_validation,
                            n_jobs=-1,
                            verbose=0)

    grid_search_XGB.fit(X_train, Y_train)
    print('Xgboost Random state:{}'.format(i),'Best score: {}'.format(grid_search_XGB.best_score_))
```

```
Xgboost Random state:0 Best score: 0.835016835016835
Xgboost Random state:1 Best score: 0.8361391694725028
Xgboost Random state:2 Best score: 0.8383838383838383
Xgboost Random state:3 Best score: 0.8361391694725028
Xgboost Random state:4 Best score: 0.8372615039281706
```

RandomState2.ipynb hosted with ❤ by GitHub view raw

Lets change the cross validation random state as well:

```
        grid_search_XGB.fit(X_train, Y_train)
        print('Xgboost Random state:{}'.format(i),'CV Random state:{}'.format(j),'Best score: {}'.format(grid_search_XGB.best_s
```

```
Xgboost Random state:0 CV Random state:0 Best score: 0.8249158249158249
Xgboost Random state:0 CV Random state:1 Best score: 0.8305274971941639
Xgboost Random state:0 CV Random state:2 Best score: 0.8395061728395061
Xgboost Random state:0 CV Random state:3 Best score: 0.8417508417508418
Xgboost Random state:0 CV Random state:4 Best score: 0.8439955106621774
Xgboost Random state:1 CV Random state:0 Best score: 0.8271604938271605
Xgboost Random state:1 CV Random state:1 Best score: 0.8249158249158249
Xgboost Random state:1 CV Random state:2 Best score: 0.8406285072951739
Xgboost Random state:1 CV Random state:3 Best score: 0.8395061728395061
Xgboost Random state:1 CV Random state:4 Best score: 0.8428731762065096
Xgboost Random state:2 CV Random state:0 Best score: 0.8271604938271605
Xgboost Random state:2 CV Random state:1 Best score: 0.8226711560044894
Xgboost Random state:2 CV Random state:2 Best score: 0.8395061728395061
Xgboost Random state:2 CV Random state:3 Best score: 0.8383838383838383
Xgboost Random state:2 CV Random state:4 Best score: 0.8428731762065096
Xgboost Random state:3 CV Random state:0 Best score: 0.8260381593714927
Xgboost Random state:3 CV Random state:1 Best score: 0.8237934904601572
Xgboost Random state:3 CV Random state:2 Best score: 0.8406285072951739
Xgboost Random state:3 CV Random state:3 Best score: 0.8417508417508418
Xgboost Random state:3 CV Random state:4 Best score: 0.8417508417508418
Xgboost Random state:4 CV Random state:0 Best score: 0.8305274971941639
Xgboost Random state:4 CV Random state:1 Best score: 0.8271604938271605
Xgboost Random state:4 CV Random state:2 Best score: 0.8484848484848485
Xgboost Random state:4 CV Random state:3 Best score: 0.8406285072951739
Xgboost Random state:4 CV Random state:4 Best score: 0.8428731762065096
```

All the results returned are different. With 5 different random states for the xgboost classifier and the cross validation split, the grid search run produces 25 different best performance results.

Having multiple results stems from the fact that the data and the algorithm we use have a random component that can affect the output.

However this creates a huge doubt in the data science process as we make changes to our model all the time.

For each of the changes that I make, I would compare the results across different runs to validate the improvements. E.g *Changing 'a' improves model by 2%, adding 'b' to the features improves it by a further 3%.*

With the variation of results as shown above, it makes me wonder if my feature engineering actually contributed to a better result **or the improvement was all down to chance**.

Maybe a different random state would make my results v

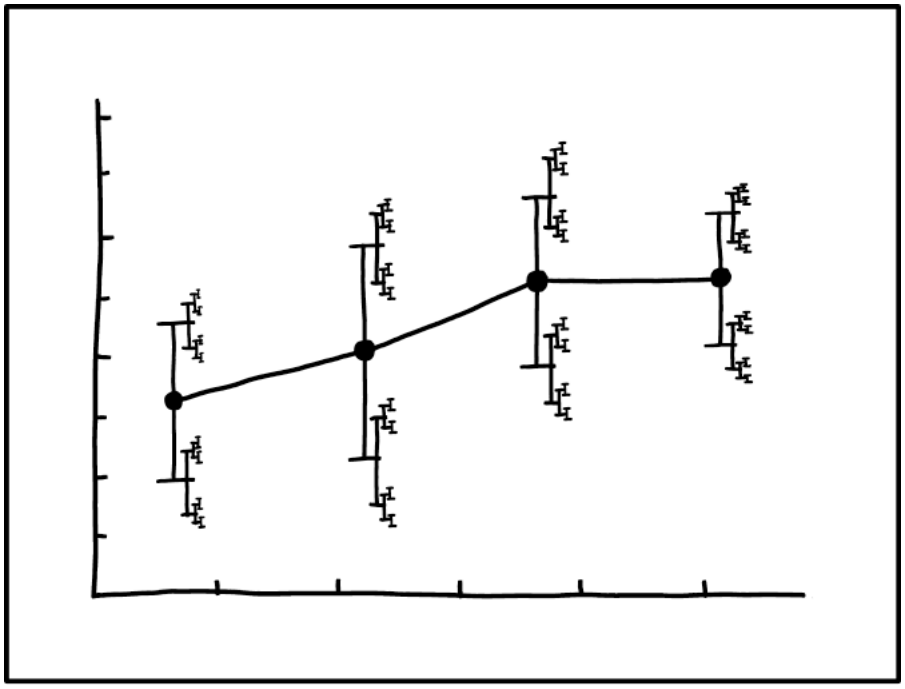
my initial run.

Which result do I present then?

It is tantalizing to present the best of the model results, since the random seed is fixed and the results are reproducible.

It seems absurd that an improvement of 2% to the result can be attributed to just a different random state. It seems like on a good day with the correct seed, we get a better result.

How should we tackle this uncertainty?



https://xkcd.com/2110/

There are many ways to tackle this issue in a data science pipeline. This is by no means a complete list, but just some of the practices I carry out on my work.

1) Fix the random state from the start

Commit to a fixed random state for everything or better yet, fix a global random seed so that randomness does not come into play. Treat it as an immutable variable in your process and not something to be tinkered with.

Alternatively,

2) Use the prediction results as an interval

Since the results vary across a range, you can opt to report the cross validation results as a range. Repeat the run with different seeds in

3) Reduce imbalance/randomness in data split

One of the ways to reduce the effect of a random split on your data is to make sure the split does not affect the composition of the data too much.

Stratify your data to reduce randomness. Stratifying your data ensures that the data for your train test split/oob error/cross validation has the same ratio of survivors/non-survivors in the train and test set respectively. The splits are made by preserving the percentage of each class will then reduce the magnitude of how a random shuffle affects the outcome. Stratification can even be done on multiple columns.

It is certainly important to note that variation in performance should not be drastic despite randomization in the data.

If the accuracy results vary wildly with the seed, it probably means that the **model is not robust** and you should consider improving your methods to better fit the data. Most of the time it does not matter, but when margins are very close, it would be tempting to consider all variables that can be used to improve model performance, including random state.

Hopefully this article has managed to highlight how randomness affects our models and a couple of ways to mitigate its effect.

Here is the github repo of the code I have used, all of which can be reproduced. Thanks for reading!