Cros Validation

It's been almost 2 years since I've started learning about machine learning and it's internal structures and processes. One thing that still concerns me that no matter how careful you are with your data you can't always avoid the `bias and variance problem` and there aren't many creative solutions to it. But there is one thing that you can do and that is `Cross Validation`. Cross validation might not make bias-variance problem obsolete but it will definitely help you reduce bias-variance problem of a model. So, this article is about Cross Validation.

As always, I'm `Md. Rishat Talukder` AKA `Itvaya` (Nobody knows me by that name), Let's get started.

- LinkedIn
- YouTube
- gtihub
- Gmail
- discord

The Intuition of Bias-Variance Problem

The Intuition

Let's say you are a student. You have an `exam coming` 1 week later. But you are `under-prepared` for the exam.

So, you `study hard`. So, hard that by the exam time you cleared 80% of the syllabus. You are confident that you should at least get a moderate grade.

And You did! You got 90%. Congrats!

So, when you took the test you realised the questions from the 20% of the syllabus were not covered that much in the test and the teacher actually wrote almost all the questions from that 80% that you studied hard for.

So, here's the question for you, ***Does this mean that you are secretly brilliant or were you just lucky? If the question from the rest of the 20% of the syllabus were covered in the test, do you thing you would have got a 100%?***

The answer is clearly, `No`.

Now, imagine you took 5 more exams on the same syllabus. The questions are now more `scrambled` in the test. You average around 78%.

This time the results looks trustworthy.

Here's some crucial points:

- After taking 5 exams you get a proper view of your performance, 1 exam is not enough to get a proper view of your performance.
- You got lucky on the first test because the questions were not scrambled, on the later tests the questions were scrambled and you performed worse in some tests.

Now, keep this intuition in mind because we are going to apply this in machine learning terms.

The Bias-Variance Problem

Now, in the above example, the question was, ***Does this mean that you are secretly brilliant or were you just lucky?***

Let's get the point clear for the first exam:

- You covered `80%` of the syllabus. Syllabus is the `dataset` and 80% is the `training set`.
- You took the first test where almost all the questions were from that `80%` of the syllabus. The first test is the `training set`.
- You performed `90%` in the first test. YOU are the `MODEL`.

Now, let's rephrase the question:

- You made a model that in trained on `80%` of the dataset and then tested it on `20%` of the dataset where most of the inputs are similar to the training set.
- It leanred the 80% of the data patterns very well so, when faced with similar data it performed well. 90%.

Which is amazing but this is a classic example of a `bias` in the model.

That's is what we actually will see if we run the model on more diverse data multiple times.

And if you have a dataset that has all the other 20% of the data patterns then the model will not perform as well as it did in the first test.

So, one thing is clear, building and testing a model on a single configuration of data is not enough to get a proper view of the model performance.

So, this is where `cross-validation` comes into play.

Cross Validation

In general splitting the data into `training` and `testing` is not enough to get a proper view of the model performance.

We do that using the `train_test_split()` method.

Let's talk about this method first.

train_test_split()

Let's say you have a simple target variable `y` and a set of features `X`.

X	y
1	1
2	1
3	1
4	2
5	2
6	2
7	2
8	3
9	3
10	3

When we apply `train_test_split()` method it will suffle this. Because, as you can see the data is in order and if we take 70% as `training` set then the first 7 rows will be in the `training` set and the last 3 rows will be in the `testing` set and it will be blunder because the last 3 rows have the class of `3` and the rest of the data does not have the class of `3`.

So, no model can ever learn the patterns of `class 3` because it does not exist in the `training` set and the model will have `high variance`. I cannot even say this is high variance because if the model never learns the patterns of `class 3` then whats the point in testing it in the `testing` set?

So, to avoid this the data has to be suffled.

SO, let's say after suffling the data we have:

X	y
8	3
9	3
10	3
1	1
2	1
3	1
4	2
5	2

X	y
6	2
7	2

Now, if we take 70% of the data as the `training` set then the `training` set will have all the `3` classes and the `testing` set will have only the `2` class.

So, now the model will be able to learn the patterns of the `3` class and should also be able to perform well right?

Even though suffling the data can solve the probelm of a bad split still if you take a look at dataset after suffling then you will still see that the `training` set will have all the `3` classes but class `2` will have only 1 example in the `training` set and when we are testing the model we are `only` tesing the model on the `2` class. So, no matter how hard we try to measure the performance of the model we cannot get a proper view of the model performance.

`train_test_split()` ensures that training and testing data come from the same underlying distribution. While shuffling helps prevent catastrophic splits, a single split is still sensitive to randomness, which is why cross-validation is often preferred for reliable model evaluation.

> **Note**: `train_test_split()` has an argument called `stratify` that can be used to ensure that the `training` and `testing` sets have the same distribution of the target variable.

So, let's see the `train_test_split()` function in action.

I'll recreate the same dataset and split it into `training` and `testing` sets.

In [ ]:
```python
import numpy as np
import pandas as pd

df = pd.DataFrame({
    'Feature' : np.arange(1,11),
    'Target' : np.array([1,1,1,2,2,2,2,3,3,3])
})
df
```

Out[ ]:

	Feature	Target
0	1	1
1	2	1
2	3	1
3	4	2
4	5	2
5	6	2
6	7	2
7	8	3
8	9	3
9	10	3

Now, let's split the data into `training` and `testing` sets.

```
In [8]:
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    df[['Feature']],
    df['Target'],
    test_size=0.30,
    random_state=43
)
```

The data is split into `training` and `testing` sets. Now, let's see how it looks like.

```
In [19]:
train_set = pd.concat([X_train, y_train], axis=1)
train_set
```

Out[19]:

	Feature	Target
8	9	3
1	2	1
3	4	2
4	5	2
7	8	3
2	3	1
5	6	2

```
In [20]:
testing_set = pd.concat([X_test, y_test], axis=1)
testing_set
```

Out[20]:

	Feature	Target
0	1	1
6	7	2
9	10	3

Here you can see that the `testing` set does not have the `class 1`. SO, there will be a unbalanced evaluation of the model.

And this is when you can use a argument called `stratify` and it will ensure that the `testing` and `training` sets have the same distribution of the `target variable`.

> You can pass a column name to the `stratify` argument for it to work.

In [ ]:
```
X_train, X_test, y_train, y_test = train_test_split(
    df[['Feature']],
    df['Target'],
    test_size=0.30,
    random_state=43,
    stratify=df['Target']
)
```

Now, let's see how it looks like.

In [21]:
```
train_set = pd.concat([X_train, y_train], axis=1)
train_set
```

Out[21]:

	Feature	Target
8	9	3
1	2	1
3	4	2
4	5	2
7	8	3
2	3	1
5	6	2

In [22]:
```
testing_set = pd.concat([X_test, y_test], axis=1)
testing_set
```

Out[22]:

	Feature	Target
0	1	1
6	7	2
9	10	3

And, we see that now the `testing` set has all the `3` classes and also the training set has all the `3` classes as well.

So, this solves the problem right?

YEAH! BUUUUUUUUT.(There's always a but)

Remember the exam intuition?

In this case,

One stratified split = **One fair exam**

Because, the exam now covers all topics, has balanced questions and it is well designed. But it is still just `one exam` right?

There are a lot of things that can still go wrong, For example:

- Getting a lucky split.
  - If the test set contains `easy` samples and closely resemble the training set, the accuray looks artificially high.
- Getting a unlucky split.
  - If the test set contains `harder or noisier` samples and has a slightly different distribution the accuray looks artificially low.
- Unreliable model comparisons.
  - Fairly comparing model with different splits becomes less reliable.

So, to resolve all this issues we use `cross-validation`.

Deep in the `cross-validation`

Th concept is `simple`:

- Take the whole dataset.
- Split it into `k` parts.
- From those `k` parts take 1 part as the `testing` set and the rest as the `training` set.
- So the same for all the `k` parts.
- Repeat the process `k` times.
- Get `k` `training` and `testing` sets.

> **Note**: `k` refers to the number of splits. More formally, it's called `folds`.

More formal terminology for this is `k-fold cross-validation`.

So, let's get deeper.

`k`-fold cross-validation

Let's say you have a the previous data set.

X	y
1	1
2	1
3	1

X	y
4	2
5	2
6	2
7	2
8	3
9	3
10	3

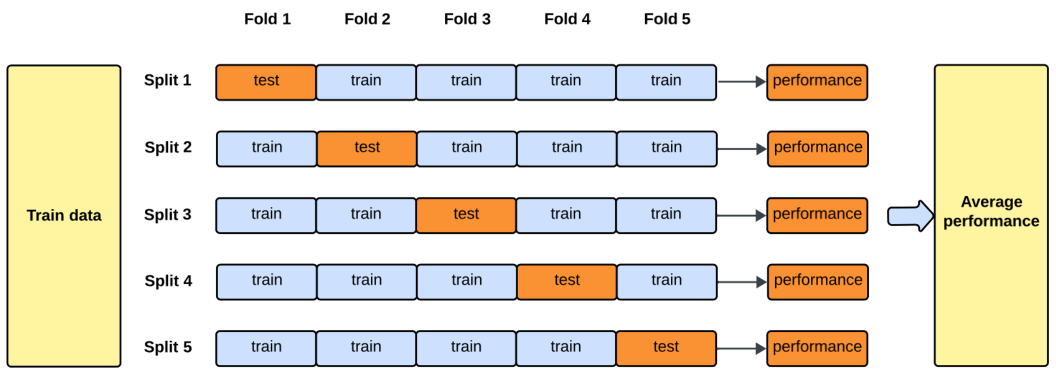
Now, the firs step is to split it into `k` parts.

Now let's say `k=3`:

So, the whole dataset will be split into 3 parts.

Fold 1		Fold 2		Fold 3	
X	y	X	y	X	y
1	1	4	2	8	3
2	1	5	2	9	3
3	1	6	2	10	3
		7	2		

Now, we take each fold as the `testing` set and the rest as the `training` set. This will be called `3 fold cross-validation`.



Above image shows an example of `5 fold cross-validation`.

This will give us a overall average accuracy of the model and help us understand if the split is good or not.

Now, if you look at the dataset we are working with it is not suffled and as a result the folds are also not suffled and this will cause the same issues of underfitting and overfitting.

So always remember. You need to shuffle the data before you split it.

So, let's see how we can do this and test it out.

In [23]:
```
df
```

Out[23]:

	Feature	Target
0	1	1
1	2	1
2	3	1
3	4	2
4	5	2
5	6	2
6	7	2
7	8	3
8	9	3
9	10	3

`Sci-kit learn` has a function called `cross_val_score()` that can do all these things for us.

In [24]:
```
from sklearn.model_selection import cross_val_score
```

We can directly pass a model instance, the features, the target variable to the `cross_val_score()` function and pass the `number` of folds. This will return the accuracy scores for each fold.

I'll be using the `k nearest neighbors` model for this example.

In [25]:
```
from sklearn.neighbors import KNeighborsClassifier

model = KNeighborsClassifier(n_neighbors=1)
```

In [27]:
```
scores = cross_val_score(
    model,
    df[['Feature']],
    df['Target'],
    cv=3
)

scores
```

Out[27]:
```
array([0.5, 1. , 1. ])
```

`cv` refers to the number of folds and we get the accuracy scores for each fold.

Adn if you look closely there something clearly wrong with `fold 1`.

The cause is we didn't shuffle the data before we split it.

By default there is no way to tell the method to suffle the data before splitting it.

But there are some other ways we can do this.

`Sklearn` give us more ways to `cross-validate`.

It classes like `KFold`, `StratifiedKFold` to give us the ability to shuffle the data before splitting it.

This can be a little hard to wrap your head around.

But first we make a `StratifiedKFold` object because we are working with a classification problem.

> StratifiedKFold splits the data into non-overlapping folds such that the labels are distributed evenly across the folds.

In [28]:
```
from sklearn.model_selection import StratifiedKFold

cv = StratifiedKFold(
    n_splits=3,
    random_state=42,
    shuffle=True
)
```

In this object we can tell how many folds we want and also tell it to shuffle the data before splitting it and also pass a random state so that we can recreate the same results.

Now, we can pass this object to tell the `cross_val_score()` function what to do and how to do it.

In [29]:
```
scores = cross_val_score(
    model,
    df[['Feature']],
    df['Target'],
    cv=cv
)

scores
```

Out[29]:
```
array([0.75      , 0.66666667, 1.        ])
```

And now we have a more reasonable result.

We can see that for each fold we get a different accuracy score but for `fold 3` we get an accuracy of `1.0` or 100%.

So, our models ovreall accuracy will be the mean of all the accuracy scores.

In [30]:
```
np.mean(scores)
```

Out[30]:
```
np.float64(0.8055555555555555)
```

That is `80.5%`.

So, Even though we can have a accuracy score of `100%` for `fold 3` we have a avg of `80.5%` accuracy throughout all the folds.

Not bad but it sure needs to be better.

And that's all I have to talk about in this article. I hope this article gave you a internal understanding of cross-validation and how to use it.

Conclusion

Cross-validation does not make a model smarter, nor does it magically improve accuracy. What it provides is something far more important: a reliable estimate of how a model is likely to perform on unseen data.

A single train-test split, even with shuffling and stratification, is still only one possible version of reality. Cross-validation reduces our dependence on luck by evaluating the model across multiple splits and averaging the results.

In practice, cross-validation should be used to compare models and tune hyperparameters, not to produce a deployable model. Once a satisfactory configuration is identified, the final step is to train a single model on the entire dataset and deploy it.

In short, stratification makes evaluation fair, cross-validation makes it reliable, and final training makes the model usable.

> Cross-validation doesn’t tell us the truth — it tells us the most honest approximation of it.

Happy Coding!!