Machine Learning I

Mohamed Hussien

ML In Practice

Titanic Dataset

Dataset Columns:

- **PassengerId:** A unique number corresponding to every passenger.
- **Name:** The full name of the passenger.
- **Sex:** The gender of the passenger as male or female.
- **Age:** The age of the passenger as an integer.
- **Pclass:** The class in which the passenger was traveling: first, second, or third.
- **SibSP:** The number of siblings and spouse of the passenger (0 if the passenger is traveling alone).
- **Parch:** The number of parents and children of the passenger (0 if the passenger is traveling alone).
- **Ticket:** The ticket number.
- **Fare:** The fare the passenger paid in British pounds.
- **Cabin:** The cabin in which the passenger was traveling.
- **Embarked:** The port in which the passenger embarked. The three options are ‘C’ forCherburg, ‘Q’ for Queenston, and ‘S’ for Southampton.
- **Survived:** Information if the passenger survived (1) or not (0).

End-to-End Example

Use colab to open this github notebook:

“s7s/machine_learning_1/ML_in_practice/End_to_end_example.ipynb”

Import Dataset Using Pandas

Import Dataset Using Pandas

Pandas can load using two objects: DataFrame and Series

They are essentially the same thing, except that the Series is used for datasets of only one column, and the DataFrame is used for datasets of more than one column.

Use pandas to import “titanic.csv”

Import Dataset Using Pandas

Pandas can load using two objects: DataFrame and Series

They are essentially the same thing, except that the Series is used for datasets of only one column, and the DataFrame is used for datasets of more than one column.

Use pandas to import “titanic.csv”

```
raw_data = pd.read_csv('./titanic.csv')
```

Using Pandas To Explore Dataset

Length function

```
len(raw_data)
Output: 891
```

Columns property

```
raw_data.columns
Output: Index(['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp', 'Parch',
       'Ticket', 'Fare', 'Cabin', 'Embarked'], dtype='object')
```

Using Pandas To Explore Dataset

Explore specific column

```
raw_data['Survived']
Output:
0      0
1      1
2      1
3      1
4      0
      ..
886    0
887    1
888    0
889    1
890    0
Name: Survived, Length: 891, dtype: int64
```

Explore or make a new DataFrame of more than one column

```
raw_data[['Name', 'Age']]
```

Check how many passengers survived

```
sum(raw_data['Survived'])
Output: 342
```

Using Pandas To Clean Dataset

Missing Values Problem

isna() method

```
raw_data.isna().sum()
Output:
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
```

We have 891 samples

Using Pandas To Clean Dataset

Missing Values Problem [Dropping Columns]

As you noticed “Cabin” column has 687 missing value out of 891

For cases like that it is preferred to drop all column

Use **drop()** method to drop the cabin column

Using Pandas To Clean Dataset

Missing Values Problem [Dropping Columns]

As you noticed “Cabin” column has 687 missing value out of 891

For cases like that it is preferred to drop all column

Use **drop()** method to drop the cabin column

```
clean_data = raw_data.drop('Cabin', axis=1)
```

The arguments to the *drop* function are

- the name of the column we want to drop, and
- the *axis* parameter, which is 1 when we want to drop a column, and 0 when we want to drop a row.

Using Pandas To Clean Dataset

Missing Values Problem [Filling Missing Data]

As you noticed “Age” column have only 177 missing value out of 891

We can fill the missing values with the median value using **fillna()** method

Using Pandas To Clean Dataset

Missing Values Problem [Filling Missing Data]

As you noticed “Age” column have only 177 missing value out of 891

We can fill the missing values with the median value using **fillna()** method

```
median_age = clean_data["Age"].median()   #A
clean_data["Age"] = clean_data["Age"].fillna(median_age)  #B
```

Using Pandas To Clean Dataset

Missing Values Problem [Filling Missing Data]

As you noticed “Age” column have only 177 missing value out of 891

We can fill the missing values with the median value using **fillna()** method

```
median_age = clean_data["Age"].median()   #A
clean_data["Age"] = clean_data["Age"].fillna(median_age)  #B
```

For “Embarked” column we have only 2 missing values out of 891

There is no median as it is categorical column, So we can fill it with “U” for unknown

Using Pandas To Clean Dataset

Missing Values Problem [Filling Missing Data]

As you noticed “Age” column have only 177 missing value out of 891

We can fill the missing values with the median value using **fillna()** method

```
median_age = clean_data["Age"].median()   #A
clean_data["Age"] = clean_data["Age"].fillna(median_age)  #B
```

For “Embarked” column we have only 2 missing values out of 891

There is no median as it is categorical column, So we can fill it with “U” for unknown

```
clean_data["Embarked"] = clean_data["Embarked"].fillna(‘U’)
```

Using Pandas To View Dataset

Use the name **.head()** method to view the dataset

	PassengerId	Survived	Pclass	Name	Sex	SibSp	Parch	Ticket	Embarked	Centered_Normalized_Fare	Centered_Normalized_Age
0	1	0	3	Braund, Mr. Owen Harris	male	1	0	A/5 21171	S	-0.971698	-0.442427
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	1	0	PC 17599	C	-0.721729	-0.044516
2	3	1	3	Heikkinen, Miss. Laina	female	0	0	STON/O2. 3101282	S	-0.969063	-0.342950
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	1	0	113803	S	-0.792711	-0.119125
4	5	0	3	Allen, Mr. William Henry	male	0	0	373450	S	-0.968575	-0.119125
...	...	...	...	...	...	...	...	...	...	...	...
886	887	0	2	Montvila, Rev. Juozas	male	0	0	211536	S	-0.949251	-0.318080
887	888	1	1	Graham, Miss. Margaret Edith	female	0	0	112053	S	-0.882888	-0.517036
888	889	0	3	Johnston, Miss. Catherine Helen "Carrie"	female	1	2	W./C. 6607	S	-0.908457	-0.293211
889	890	1	1	Behr, Mr. Karl Howell	male	0	0	111369	C	-0.882888	-0.342950
890	891	0	3	Dooley, Mr. Patrick	male	0	0	370376	Q	-0.969746	-0.193733

Save Dataset Using Pandas

We can save dataset using **.to_csv()** method

Save Dataset Using Pandas

We can save dataset using **.to_csv()** method

```
clean_data.to_csv('./clean_titanic_data.csv', index=None)
```

Feature Engineering

Turning categorical data into numerical data [One-hot Encoding]

Machine learning models deals only with numbers.

Why not only turn categories into a sequence of numbers 0,1,2,3,4,..?

For “Embarked” column if we replace “c” with 0, “q” with 1 and “s” with 2.
We tell the model that people are more likely to survive either with this embarking order “s”, “q”, “c”
Or this embarking order “c”, “q”, “c”. And that are not all possibilities.

Attaching numbers to the features implicitly organizes them for the model, and we don’t want that.
We want the model to have more freedom over how to organize the features.

For that we use **One-hot Encoding**.

Feature Engineering

Turning categorical data into numerical data [One-hot Encoding]

Feature Engineering

Turning categorical data into numerical data [One-hot Encoding]

	embarked
Passenger 1	Q
Passenger 2	S
Passenger 3	C
Passenger 4	S

→

	embarked_c	embarked_q	embarked_s
Passenger 1	0	1	0
Passenger 2	0	0	1
Passenger 3	1	0	0
Passenger 4	0	0	1

Feature Engineering

Turning categorical data into numerical data [One-hot Encoding]

	embarked
Passenger 1	Q
Passenger 2	S
Passenger 3	C
Passenger 4	S

➔

	embarked_c	embarked_q	embarked_s
Passenger 1	0	1	0
Passenger 2	0	0	1
Passenger 3	1	0	0
Passenger 4	0	0	1

Use pandas method **.get_dummies()** to get the one hot encoding of “embarked”, “pclass” and “gender”

Use pandas method **.drop()** to remove the old columns and method **.concat()** to add the new columns

Feature Engineering

Turning categorical data into numerical data [One-hot Encoding]

	embarked
Passenger 1	Q
Passenger 2	S
Passenger 3	C
Passenger 4	S

→

	embarked_c	embarked_q	embarked_s
Passenger 1	0	1	0
Passenger 2	0	0	1
Passenger 3	1	0	0
Passenger 4	0	0	1

Use pandas method **.get_dummies()** to get the one hot encoding of “embarked”, “pclass” and “gender”

Use pandas method **.drop()** to remove the old columns and method **.concat()** to add the new columns

```
embarked_columns = pandas.get_dummies(preprocessed_data["Embarked"], prefix="Embarked")
sex_columns = pandas.get_dummies(preprocessed_data["Sex"], prefix="Sex")
pclass_columns = pandas.get_dummies(preprocessed_data["Pclass"], prefix="Pclass")
preprocessed_data = pandas.concat([preprocessed_data, gender_columns], axis=1)
preprocessed_data = pandas.concat([preprocessed_data, embarked_columns], axis=1)
preprocessed_data = pandas.concat([preprocessed_data, pclass_columns], axis=1)
preprocessed_data = preprocessed_data.drop(['Sex', 'Embarked', ‘Pclass’], axis=1)
```

Feature Engineering

Turning numerical data into categorical data [Binning]

Why can we make that?

Let’s take “Age” column as an example, it has two possibilities:
- The older the passenger is, the more likely they are to survive.
- The older the passenger is, the less likely they are to survive.

But, what if the relationship between age and survival is not as straightforward?
What if the best possibility of survival is when the passenger is between 20 and 30?

Not all models smart enough to learn that itself.

Here comes **Binning**.

Feature Engineering

Turning numerical data into categorical data [Binning]

We can turn the age column into the following:

- From 0 to 10 years old.
- From 11 to 20 years old.
- From 21 to 30 years old.
- From 31 to 40 years old.
- From 41 to 50 years old.
- From 51 to 60 years old.
- From 61 to 70 years old.
- From 71 to 80 years old.
- 81 years old or older.

Use **.cut()** method to make bins from the age column

Feature Engineering

Turning numerical data into categorical data [Binning]

We can turn the age column into the following:

- From 0 to 10 years old.
- From 11 to 20 years old.
- From 21 to 30 years old.
- From 31 to 40 years old.
- From 41 to 50 years old.
- From 51 to 60 years old.
- From 61 to 70 years old.
- From 71 to 80 years old.
- 81 years old or older.

Use **.cut()** method to make bins from the age column

```
bins = [0, 10, 20, 30, 40, 50, 60, 70, 80]
categorized_age = pd.cut(preprocessed_data['Age'], bins)
preprocessed_data['Categorized_age'] = categorized_age
preprocessed_data = preprocessed_data.drop(["Age"], axis=1)
```

Feature Engineering

Drop Unnecessary features

- **PassengerId:** A unique number corresponding to every passenger.
- **Name:** The full name of the passenger.
- **Sex:** The gender of the passenger as male or female.
- **Age:** The age of the passenger as an integer.
- **Pclass:** The class in which the passenger was traveling.
- **SibSP:** The number of siblings and spouse of the passenger.
- **Parch:** The number of parents and children of the passenger.
- **Ticket:** The ticket number.
- **Fare:** The fare the passenger paid in British pounds.
- **Cabin:** The cabin in which the passenger was traveling.
- **Embarked:** The port in which the passenger embarked.
- **Survived:** Information if the passenger survived (1) or not (0).

Feature Engineering

Drop Unnecessary features

- **PassengerId:** A unique number corresponding to every passenger.
- **Name:** The full name of the passenger.
- **Sex:** The gender of the passenger as male or female.
- **Age:** The age of the passenger as an integer.
- **Pclass:** The class in which the passenger was traveling.
- **SibSP:** The number of siblings and spouse of the passenger.
- **Parch:** The number of parents and children of the passenger.
- **Ticket:** The ticket number.
- **Fare:** The fare the passenger paid in British pounds.
- **Cabin:** The cabin in which the passenger was traveling.
- **Embarked:** The port in which the passenger embarked.
- **Survived:** Information if the passenger survived (1) or not (0).

Training the model with unnecessary features may cause overfitting

Feature Engineering

Drop Unnecessary features

- **PassengerId:** A unique number corresponding to every passenger.
- **Name:** The full name of the passenger.
- **Sex:** The gender of the passenger as male or female.
- **Age:** The age of the passenger as an integer.
- **Pclass:** The class in which the passenger was traveling.
- **SibSP:** The number of siblings and spouse of the passenger.
- **Parch:** The number of parents and children of the passenger.
- **Ticket:** The ticket number.
- **Fare:** The fare the passenger paid in British pounds.
- **Cabin:** The cabin in which the passenger was traveling.
- **Embarked:** The port in which the passenger embarked.
- **Survived:** Information if the passenger survived (1) or not (0).

Training the model with unnecessary features may cause overfitting

Which features do you propose to drop?

Use .drop() method to drop 'Name', 'Ticket', 'PassengerId’ columns

Data Splitting

Data Splitting

Firstly, we need to separate the features from the labels.

Data Splitting

Firstly, we need to separate the features from the labels.

```
features = data.drop(["Survived"], axis=1)
labels = data["Survived"]
```

Data Splitting

Firstly, we need to separate the features from the labels.

```
features = data.drop(["Survived"], axis=1)
labels = data["Survived"]
```

Secondly, we need to split data into 60% training, 20% validation, 20% testing.

We have to do that through two stages using sklearn method **train_test_split()**

Data Splitting

Firstly, we need to separate the features from the labels.

```
features = data.drop(["Survived"], axis=1)
labels = data["Survived"]
```

Secondly, we need to split data into 60% training, 20% validation, 20% testing.

We have to do that through two stages using sklearn method **train_test_split()**

```
features_train, features_validation_test, labels_train, labels_validation_test =
       train_test_split(features, labels, test_size=0.4)

features_validation, features_test, labels_validation, labels_test =
       train_test_split(features_validation_test, labels_validation_test, test_size=0.5)
```

Training Several Models

Training Several Models

We will train six different models

- Logistic Regression
- Decision Tree
- SVM
- Random Forest
- Gradient Boost
- AdaBoost

Training Several Models

We will train six different models

- Logistic Regression
- Decision Tree
- SVM
- Random Forest
- Gradient Boost
- AdaBoost

```
from sklearn.linear_model import LogisticRegression
lr_model = LogisticRegression()
lr_model.fit(features_train, labels_train)

from sklearn.tree import DecisionTreeClassifier
dt_model = DecisionTreeClassifier()
dt_model.fit(features_train, labels_train)

from sklearn.svm import SVC
svm_model = SVC()
svm_model.fit(features_train, labels_train)
```

```
from sklearn.ensemble import RandomForestClassifier
rf_model = RandomForestClassifier()
rf_model.fit(features_train, labels_train)

from sklearn.ensemble import GradientBoostingClassifier
gb_model = GradientBoostingClassifier()
gb_model.fit(features_train, labels_train)

from sklearn.ensemble import AdaBoostClassifier
ab_model = AdaBoostClassifier()
ab_model.fit(features_train, labels_train)
```

Moldes Evaluation

Moldes Evaluation

We will use two evaluation metrics using validation set:

- Accuracy

- F1-score

Moldes Evaluation

We will use two evaluation metrics using validation set:

- Accuracy

- F1-score

```
print("Scores of the models")
print("Logistic regression:", lr_model.score(features_validation, labels_validation))
```

```
from sklearn.metrics import f1_score

print("F1-scores of the models:")

lr_predicted_labels = lr_model.predict(features_validation)
print("Logistic regression:", f1_score(labels_validation, lr_predicted_labels))
```

Testing The Model

Testing The Model

After comparing the models using the validation set, we have chosen the gradient boosted tree as the best model for this dataset.

But to see if we really did well, or if we accidentally overfitted, we need to give this model its final test. We need to test the model in the test set that we haven't touched yet.

Testing The Model

After comparing the models using the validation set, we have chosen the gradient boosted tree as the best model for this dataset.

But to see if we really did well, or if we accidentally overfitted, we need to give this model its final test. We need to test the model in the test set that we haven't touched yet.

```
gb_model.score(features_test, labels_test)
Output:
0.8324022346368715
```

```
gb_predicted_test_labels = gb_model.predict(features_test)
f1_score(labels_test, gb_predicted_test_labels)
Output:
0.8026315789473685
```

Testing The Model

After comparing the models using the validation set, we have chosen the gradient boosted tree as the best model for this dataset.

But to see if we really did well, or if we accidentally overfitted, we need to give this model its final test. We need to test the model in the test set that we haven’t touched yet.

```
gb_model.score(features_test, labels_test)
Output:
0.8324022346368715
```

```
gb_predicted_test_labels = gb_model.predict(features_test)
f1_score(labels_test, gb_predicted_test_labels)
Output:
0.8026315789473685
```

We could do better if we tune the hyperparameters.. Here comes the **Grid Search**.

Tuning HyperParameters [Grid Search]

Tuning HyperParameters [Grid Search]

Let’s try to enhance the poor SVM
(69% accuracy, 42% F1-Score)

SVM is very powerful. So maybe the bad performance due to hyperparameters.

Use sklearn method **GridSearchCV()** to train the svm with the following hyperparameters:
Kernel: rbf
C: 0.01, 0.1, 1, 10, 100
gamma: 0.01, 0.1, 1, 10, 100

Tuning HyperParameters [Grid Search]

Let’s try to enhance the poor SVM (69% accuracy, 42% F1-Score)

SVM is very powerful. So maybe the bad performance due to hyperparameters.

Use sklearn method **GridSearchCV()** to train the svm with the following hyperparameters:
Kernel: rbf
C: 0.01, 0.1, 1, 10, 100
gamma: 0.01, 0.1, 1, 10, 100

```
svm_parameters = {'kernel': ['rbf'],
                  'C': [0.01, 0.1, 1 , 10, 100],
                  'gamma': [0.01, 0.1, 1, 10, 100]
                 } #A
svm = SVC()  #B

svm_gs = GridSearchCV(estimator = svm,
                      param_grid = svm_parameters)  #C

svm_gs.fit(features_train, labels_train)  #D
```

Tuning HyperParameters [Grid Search]

Let's try to enhance the poor SVM
(69% accuracy, 42% F1-Score)

SVM is very powerful. So maybe the bad performance due to hyperparameters.

Use sklearn method **GridSearchCV()** to train the svm with the following hyperparameters:
Kernel: rbf
C: 0.01, 0.1, 1, 10, 100
gamma: 0.01, 0.1, 1, 10, 100

```
svm_parameters = {'kernel': ['rbf'],
                  'C': [0.01, 0.1, 1 , 10, 100],
                  'gamma': [0.01, 0.1, 1, 10, 100]
                 } #A
svm = SVC()  #B

svm_gs = GridSearchCV(estimator = svm,
                      param_grid = svm_parameters)  #C

svm_gs.fit(features_train, labels_train)  #D
```

```
svm_winner = svm_gs.best_estimator_
svm_winner.score(features_validation, labels_validation)
Output:
0.7303370786516854
```

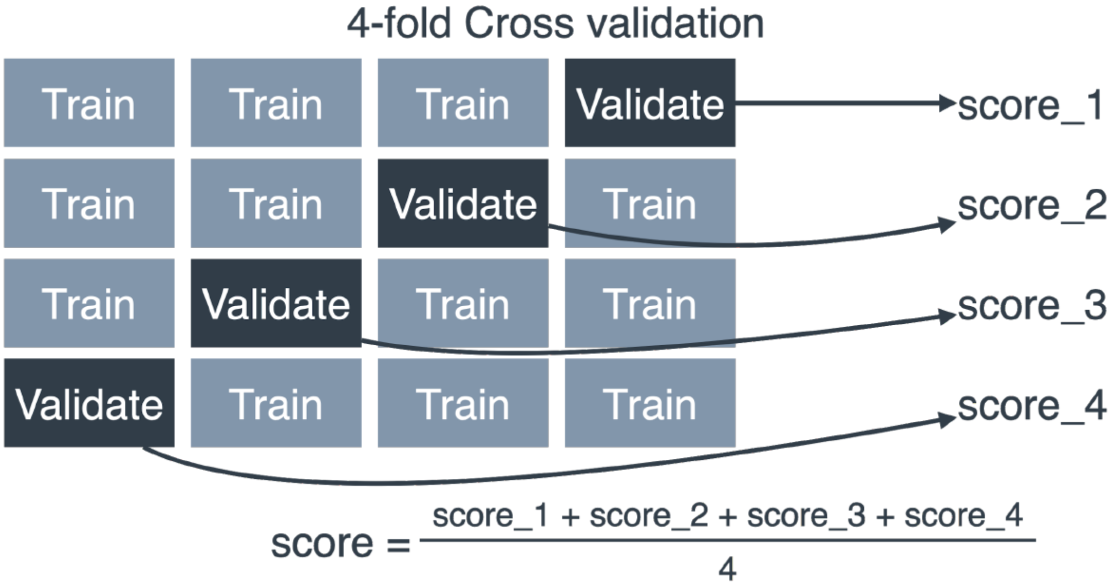
Tuning HyperParameters [Grid Search]

Tuning HyperParameters [Grid Search]

Grid Search uses the concept of the K-Fold for evaluating the model.

Tuning HyperParameters [Grid Search]

Grid Search uses the concept of the K-Fold for evaluating the model.

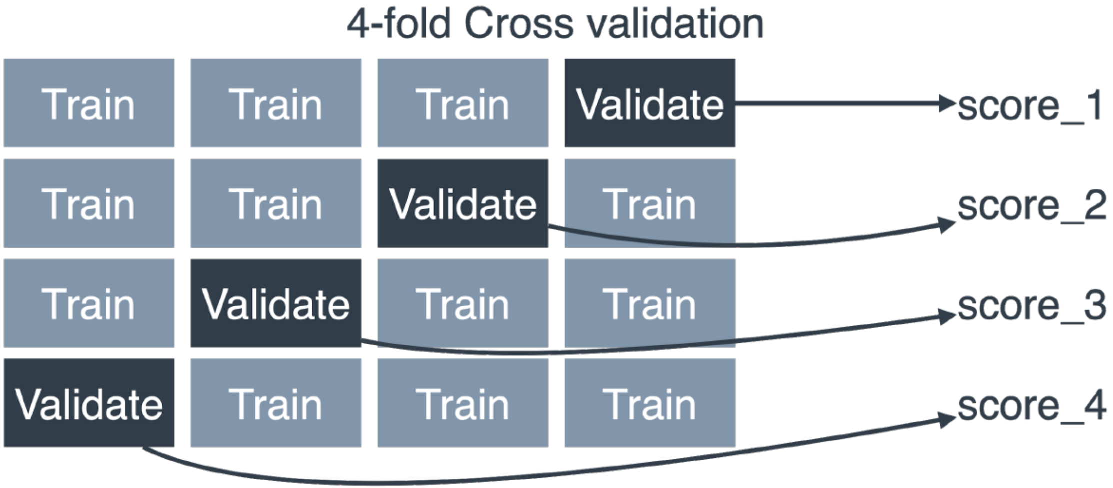


Tuning HyperParameters [Grid Search]

Grid Search uses the concept of the K-Fold for evaluating the model.

Note that the rule of K-Fold is to evaluate the model but the final model is trained over all the dataset.

To view the K-Fold models results within the grid search use **svm_gs.cv_results_**



$$\text{score} = \frac{\text{score\_1} + \text{score\_2} + \text{score\_3} + \text{score\_4}}{4}$$

Different topics

Different topics

- Stratification
- Target Transformation
- Extra Trees

Feedback

Thank You