precision-recall

April 30, 2023

1 Percision-Recall Trade-Off

Two commonly used metrics for classification are **precision** and **recall**. Conceptually, **precision** refers to the percentage of positive results which are relevant, while **recall** refers to the percentage of positive cases correctly classified. Often we face a situation where choosing between increasing the **recall** (while lowering the **precision**) or increasing the **precision** (and lowering the **recall**) becomes necessary. This notebook reads a popular CSV file containing the passenger list of Titanic and creates a **Logistic Regression** model with it. The model will help to predict if a person will survive or not by analyzing other dependent variables from the CSV file. The end goal however is to increase the **precision** of the model as much as possible, thus that all the positive predictions the model makes are correct (increased **precision**), even if it isn't able to catch all the positive predictions (lower **recall**).

[1]:
```
# Importing all the necessary libraries:

import pandas as pd
from numpy import arange, argmax
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_score, recall_score, f1_score
```

[2]:
```
# Reading the CSV file with pandas:

df = pd.read_csv('Titanic Passenger List.csv')

df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64
 7   Parch        891 non-null    int64
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object
 11  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

There are many text-based data (**object**) in the file. As it is difficult for a machine learning model to read and analyse anything but a numerical value, we are dropping some columns from the CSV file. Also, we are converting the "**Sex**" column from **strings** to **integers** as it is an important variable. After conversion there will be just two values denoting sex of the passengers: 0 for female and 1 for male.

[3]:
```
# Using pandas 'get_dummies' function to convert strings to integeres:

dummies = pd.get_dummies(df['Sex'], drop_first= True, dtype= 'int64')

# The function created a new dataframe that we need to join with the original:

df = pd.concat([df, dummies], axis= 'columns')

# Dropping all the unnecessary columns along with the original Sex column:

df = df.drop(['PassengerId', 'Name', 'Sex',
              'Ticket', 'Cabin', 'Embarked'], axis= 'columns')

df.head()
```

[3]:
```
   Survived  Pclass   Age  SibSp  Parch     Fare  male
0         0       3  22.0      1      0   7.2500     1
1         1       1  38.0      1      0  71.2833     0
2         1       3  26.0      0      0   7.9250     0
3         1       1  35.0      1      0  53.1000     0
4         0       3  35.0      0      0   8.0500     1
```

We need to see if there are any null values in the dataset as that could hinder our model.

[4]:
```
# Counting all the null values

df.isnull().sum()
```

[4]:
```
Survived      0
Pclass        0
Age         177
SibSp         0
```

```
Parch          0
Fare           0
male           0
dtype: int64
```

The age column has 177 null values and we need to get rid of them. To do that we will be replacing the null values with the mean age of all the passengers.

[5]:
```
# Calculating the mean age of the passengers:

mean_age = round(df['Age'].mean())

# Replacing the null values with the mean age:

df['Age'] = df['Age'].fillna(mean_age)

df.isnull().sum()
```

[5]:
```
Survived    0
Pclass      0
Age         0
SibSp       0
Parch       0
Fare        0
male        0
dtype: int64
```

Now we will start making our model. We will make two models, where in the first model we will use all the dependent variables from the CSV, and in the second model, we will use only: the '**Pclass**', '**Age**' and the '**male**' column. We will score the two models according to their **accuracy**, **precision**, **recall** and **f1 score**.

Note: The **F1 score** is the harmonic mean of **precision** and **recall**.

[6]:
```
# Creating the first model with all the variables:

x = df.drop('Survived', axis= 'columns').values
y = df['Survived'].values


x_train, x_test, y_train, y_test = train_test_split( x,y,
                                                     test_size= 0.3,
                                                     random_state= 5
                                                     )


model_1 = LogisticRegression()
model_1.fit(x_train, y_train)
```

```
y_pred = model_1.predict(x_test)

# Printing the scores of the first model:

print("Model 1 Scores:")
print(f" 1) Accuracy: {round(accuracy_score(y_test, y_pred), 2) * 100}%")
print(f" 2) Precision: {round(precision_score(y_test, y_pred), 2) * 100}%")
print(f" 3) Recall: {round(recall_score(y_test, y_pred), 2) * 100}%")
print(f" 4) F1_score: {round(f1_score(y_test, y_pred), 2) * 100}% \n")

# Creating the second model with only the 'Pclass', 'Age' and the 'male' column:

X = df[['Pclass', 'Age', 'male']].values

X_train, X_test, Y_train, Y_test = train_test_split( X,y,
                                                     test_size= 0.3,
                                                     random_state= 5
                                                     )


model_2 = LogisticRegression()
model_2.fit(X_train, Y_train)

Y_pred = model_2.predict(X_test)

# Printing the scores of the second model:

print("Model 2 Scores:")
print(f" 1) Accuracy: {round(accuracy_score(Y_test, Y_pred), 2) * 100}%")
print(f" 2) Precision: {round(precision_score(Y_test, Y_pred), 2) * 100}%")
print(f" 3) Recall: {round(recall_score(Y_test, Y_pred), 2) * 100}%")
print(f" 4) F1_score: {round(f1_score(Y_test, Y_pred), 2) * 100}%")
```

```
Model 1 Scores:
 1) Accuracy: 81.0%
 2) Precision: 79.0%
 3) Recall: 67.0%
 4) F1_score: 73.0%

Model 2 Scores:
 1) Accuracy: 82.0%
 2) Precision: 81.0%
 3) Recall: 67.0%
 4) F1_score: 74.0%
```

As we can see the second model has a good precision score than the first model. As our end goal is to make a model with the highest precision score, the second model should be our preferred model right away.

However, with a **Logistic Regression** model, we have an easy way of shifting between emphasizing precision and emphasizing recall. As the model dosen't just return a prediction, but it returns a probability value between 0 and 1 of all the datapoints. Typically if the value is $>=$ **0.5**, it predicts that the passenger survived, anything below it and it predicts that the passenger didn't survive. By tweaking this **threshold** of **0.5** we could increase or decrease the precision of the model. Therefore we will try to change the threshold and see if we could increase the precision of the second model further.

[7]:
```
# Getting the probability values of X_test:

Y_pred_proba = model_2.predict_proba(X_test)

Y_pred_proba[:5]
```

[7]:
```
array([[0.91360539, 0.08639461],
       [0.91360539, 0.08639461],
       [0.92420319, 0.07579681],
       [0.90417954, 0.09582046],
       [0.91799671, 0.08200329]])
```

The method "**predict_proba**" of the Sklearn library gives us two values for each data point. The first value is the probability that the datapoint is in the **0** class (**didn't survive**) and the second is the probability that the datapoint is in the **1** class (**did survive**). We only need the second column of this result. Now we need to find a suitable **threshold** that will give us the best **precision score**. In this regard, anything closer to **1** will always be the best. But it will lower the recall rate significantly. Thus, instead of finding from probability value **0 to 1**, we will find the **threshold** from **0 to 0.8**. That will give some room to **recall**.

[8]:
```
# Applying threshold tp positive probabilities to crate labels:

def to_labels(possible_probs, thresholds):
    return (possible_probs >= thresholds).astype('int')

# Defining thresholds:

thresholds = arange(0, 0.80, 0.001)

# Evaluating each threshold:

scores = [precision_score( Y_test,
                           to_labels(Y_pred_proba[:,1], t),
                           zero_division= 1
                           )
          for t in thresholds]

# Getting the best threshold:

ix = argmax(scores)
```

```
print("Result:")
print(f" 1) Best threshold = {thresholds[ix]}")
print(f" 2) Best Precision Score = {round(scores[ix], 2) * 100}%")
```

```
Result:
 1) Best threshold = 0.725
 2) Best Precision Score = 94.0%
```

We have got the best **threshold** value and it enables us to get the highest **precision score** possible. If we compare the positive probabilities from '**X__test**' to our **threshold**, we make sure that only the values equal to or greater than our **threshold** get a '**True**'.

[9]:
```
# Comparing these probability values with our threshold:

Y_pred_proba = model_2.predict_proba(X_test)[:,1] >= thresholds[ix]

Y_pred_proba[:30]
```

[9]:
```
array([False, False, False, False, False, False,  True, False,  True,
       False, False, False, False,  True, False,  True, False, False,
       False, False, False, False,  True, False, False, False, False,
       False, False, False])
```

Now we will score the second model again in terms of **precision** and **recall**. And compare it with the first time.

[10]:
```
# Score before tweaking the threshold:

print("Before increasung the threshold:")
print(f" 1) Precision: {round(precision_score(Y_test, Y_pred), 2) * 100}%")
print(f" 2) Recall: {round(recall_score(Y_test, Y_pred), 2) * 100}% \n")

# Score after tweaking the threshold:

print("After increasung the threshold:")
print(f" 1) Precision: {round(precision_score(Y_test, Y_pred_proba), 2)* 100}%")
print(f" 2) Recall: {round(recall_score(Y_test, Y_pred_proba), 2) * 100}%")
```

```
Before increasung the threshold:
 1) Precision: 81.0%
 2) Recall: 67.0%

After increasung the threshold:
 1) Precision: 94.0%
 2) Recall: 45.0%
```

As we can see the precision of the model increased quite a lot. We will now try to predict the survival of three imaginary passengers: One a female at the age of 24 travelling in 1st class, second a boy at the age of 15, also travelling in 1st class, and an old man of 60 travelling in the 2nd class. We will create a new model that will be trained on the entire dataset rather than just **X_train** and **Y_train**.

[11]:
```python
# Creating a new Logistic Regression model:

new_model = LogisticRegression()
new_model.fit(X,y)

# Using the dependent variables to predict an outcome:

''' The code goes like this:

        1st passenger -
            'Pclass' = 1,
            'Age' = 24,
            'male' = 0

        2nd passenger -
            'Pclass' = 1,
            'Age' = 15,
            'male' = 1

        3rd passenger -
            'Pclass' = 2,
            'Age' = 60,
            'male' = 1 '''

result = new_model.predict_proba([ [1,24,0],
                                   [1,15,1],
                                   [2,60,1]
                                   ])[:,1] >= thresholds[ix]

# Printing the results:

print("Did they survived?")
print(f" 1st passenger - {result[0]}")
print(f" 2nd passenger - {result[1]}")
print(f" 3rd passenger - {result[2]}")
```

```
Did they survived?
 1st passenger - True
 2nd passenger - False
 3rd passenger - False
```

As we have increased the **precision** of the model, whoever it has labelled '**Survived**' has more surety of survival now than in the normal model.

- by Sourin Das