3_Build_a_confusion_matrix

October 7, 2021

1 Exercise: Build a confusion matrix

In this exercise we go into more detail on measuring the perfomance of classification models, using the concepts of *imbalanced datasets*, *accuracy* and *confusion matrices*.

1.1 Data visualization

Our new dataset represents different classes of objects found in snow.

Let's start this exercise by loading in and having a look at our data:

[ ]:
```
import pandas

#Import the data from the .csv file
dataset = pandas.read_csv('snow_objects.csv', delimiter="\t")

#Let's have a look at the data
dataset
```

[ ]:

	size	roughness	color	motion	label
0	50.959361	1.318226	green	0.054290	tree
1	60.008521	0.554291	brown	0.000000	tree
2	20.530772	1.097752	white	1.380464	tree
3	28.092138	0.966482	grey	0.650528	tree
4	48.344211	0.799093	grey	0.000000	tree
…	…	…	…	…	…
2195	1.918175	1.182234	white	0.000000	animal
2196	1.000694	1.332152	black	4.041097	animal
2197	2.331485	0.734561	brown	0.961486	animal
2198	1.786560	0.707935	black	0.000000	animal
2199	1.518813	1.447957	brown	0.000000	animal

[2200 rows x 5 columns]

1.2 Data Exploration

We can see that the dataset has both continuous (`size`, `roughness`, `motion`) and categorical data (`color` and `label`). Let's do some quick data exploration and see what different label classes we have and their respective counts:

[ ]:
```
import graphings # custom graphing code. See our GitHub repo for details

# Plot a histogram with counts for each label
graphings.multiple_histogram(dataset, label_x="label", label_group="label",␣
↪title="Label distribution")
```

The histogram above makes it very easy to understand both the labels we have in the dataset and their distribution.

One important information to notice is that this is an *imbalanced dataset*: classes are not represented in the same proportion (we have 4 times more rocks and trees than animals, for example).

This is relevant as imbalanced sets are very common "in the wild", and in the future we will learn how to address that to build better models.

We can do the same analisys for the `color` feature:

[ ]:
```
# Plot a histogram with counts for each label
graphings.multiple_histogram(dataset, label_x="color", label_group="color",␣
↪title="Color distribution")
```

We can notice that:

- We have `8` different color categories.
- The `color` feature is also heavily imbalanced.
- Out plotting algorithm is not smart enough to assign the correct colors to their respective names.

Let's see what we can find about the other features:

[ ]:
```
graphings.box_and_whisker(dataset, label_y="size", title='Boxplot of "size"␣
↪feature')
```

Above we can see that the majority of our samples are relatively small, with sizes ranging from `0` to `70`, but we have a few much bigger outliers.

Let's take a look at the `roughness` feature:

[ ]:
```
graphings.box_and_whisker(dataset, label_y="roughness", title='Boxplot of␣
↪"roughness" feature')
```

There's not a lot of variation here: values for `roughness` range from `0` to a little over `2`, with most samples having values close to the mean.

Finally, let's plot the `motion` feature:

[ ]:
```
graphings.box_and_whisker(dataset, label_y="motion", title='Boxplot of "motion"␣
↪feature')
```

Most objects seem to be either static or moving very slowly. There is a smaller number of objects moving faster, with a couple of outliers over `10`.

From the data above one could assume that the smaller and faster objects are likely hikers and animals, whereas the bigger, more static elements are trees and rocks.

1.3 Building a classification model

Let's build and train a classification model using a random forest, to predict the class of an object based on the features in our dataset:

[ ]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

# Split the dataset in an 70/30 train/test ratio.
train, test = train_test_split(dataset, test_size=0.3, random_state=2)
print(train.shape)
print(test.shape)
```

```
(1540, 5)
(660, 5)
```

Now we can train our model, using the `train` dataset we've just created:

[ ]:
```
# Create the model
model = RandomForestClassifier(n_estimators=1, random_state=1,verbose=False)

# Define which features are to be used (leave color out for now)
features= ["size", "roughness", "motion"]

# Train the model
model.fit(train[features], train.label)

print("Model trained!")
```

```
Model trained!
```

1.4 Assessing our model

We can now use our newly trained model to make predictions using the *test* set.

By comparing the values predicted to the actual labels (also called *true* values), we can measure the model's performance using different *metrics*.

Accuracy, for example, is the simply number of correctly predicted labels out of all predictions performed:

```
Accuracy = Correct Predictions / Total Predictions
```

Let's see how this can be done in code:

[ ]:
```
# Import a function that measures a models accuracy
from sklearn.metrics import accuracy_score

# Calculate the model's accuracy on the TEST set
actual = test.label
predictions = model.predict(test[features])
```

```
# Return accuracy as a fraction
acc = accuracy_score(actual, predictions)

# Return accuracy as a number of correct predictions
acc_norm = accuracy_score(actual, predictions, normalize=False)

print(f"The random forest model's accuracy on the test set is {acc:.4f}.")
print(f"It correctly predicted {acc_norm} labels in {len(test.label)}␣
 ↪predictions.")
```

```
The random forest model's accuracy on the test set is 0.8924.
It correctly predicted 589 labels in 660 predictions.
```

Our model **seems** to be doing quite well!

That intuition, however, can be misleading:

- Accuracy does not take into account the **wrong** predictions made by the model
- It's also not very good at painting a clear picture in *class-imbalanced datasets*, like ours, where the number of possible classes is not evenly distributed (recall that we have 800 trees, 800 rocks, but only 200 animals)

1.5 Building a confusion matrix

A *confusion matrix* is a table where we compare the actual labels to what the model predicted. It gives us a more detailed understanding of how the model is doing and where it's getting thigs right or missing.

This is one of the ways we can do that in code:

[ ]:
```
# sklearn has a very convenient utility to build confusion matrices
from sklearn.metrics import confusion_matrix

# Build and print our confusion matrix, using the actual vaues and predictions
# from the test set, calculated in previous cells
cm = confusion_matrix(actual, predictions, normalize=None)

print("Confusion matrix for the test set:")
print(cm)
```

```
Confusion matrix for the test set:
[[ 28  38   0   0]
 [ 30 103   1   0]
 [  0   1 217   1]
 [  0   0   0 241]]
```

While the matrix above can be useful in calculations, it's not very helpful or intuitive.

Let's add a plot with lables and colors to turn that data into actual insights:

[ ]:
```
# We use plotly to create plots and charts
import plotly.figure_factory as ff

# Create the list of unique labels in the test set, to use in our plot
# I.e., ['animal', 'hiker', 'rock', 'tree']
x=y= sorted(list(test["label"].unique()))
print(x)
print(y)
```

```
['animal', 'hiker', 'rock', 'tree']
['animal', 'hiker', 'rock', 'tree']
```

[ ]:
```
# Plot the matrix above as a heatmap with annotations (values) in its cells
fig = ff.create_annotated_heatmap(cm, x, y)

# Set titles and ordering
fig.update_layout(  title_text="<b>Confusion matrix</b>",
                    yaxis = dict(categoryorder = "category descending"))

fig.add_annotation(dict(font=dict(color="black",size=14),
                        x=0.5,
                        y=-0.15,
                        showarrow=False,
                        text="Predicted label",
                        xref="paper",
                        yref="paper"))

fig.add_annotation(dict(font=dict(color="black",size=14),
                        x=-0.15,
                        y=0.5,
                        showarrow=False,
                        text="Actual label",
                        textangle=-90,
                        xref="paper",
                        yref="paper"))

# We need margins so the titles fit
fig.update_layout(margin=dict(t=80, r=20, l=100, b=50))
fig['data'][0]['showscale'] = True
fig.show()
```

Notice that the plot has the **Actual labels** on the `y-axis` and **Predicted labels** on the `x-axis`, as defined by the `confusion_matrix` function call.

Let's go over the cells in the **top row**, left to right, to understand what the plot is telling us:

- The first cell means we had `28` animals (the real value) and our model predicted we had `28` animals (the prediction). Those are all TPs or *true positives*.

However,

- The second cell indicates the model **incorrectly** classified `38` animals as hikers (thus `38` FNs or *false negatives* in that cell). That doesn't look good!
- Third and fourth cells are correct: they each have `0` animals classified as rocks or trees (thus `0` *false negatives*).

Now, onto the **second row** from the top:

- First cell has `30` hikers classified as animals; In this case we have `30` FPs or *false positives* in that cell.
- In the second cell we have `103` hikers correctly classified (TPs or *true positives*)
- In the third cell we have `1` hiker classified as a rock! (`1` FN)
- Finally, in the fourth cell we have `0` hikers classified as trees (meaning `0` FNs)

If we go over the same process for the remaining rows, we can see that the model is generally accurate, but only because we have so many rocks and trees in our set and because it does well on those classes.

When it comes to hikers and animals the model gets confused (it shows a high number of FPs and FNs), but because these classes are less represented in the dataset the *accuracy score* remains high.

1.6 Summary

In this exercise we talked about the following concepts:

- *Imbalanced datasets*, where features or classes can be represented by a disproportionate number of samples.
- *Accuracy* as a metric to assess model performance, and its shortcomings.
- How to generate, plot and interpret *confusion matrices*, to get a better understanding of how a classification model is performing.