Run all Kernel Compute not connected

Exercise: Build a simple logistic regression

```
import pandas
!pip install statsmodels
!wget https://raw.githubusercontent.com/MicrosoftDocs/mslearn-introduction-to-machine-learning
!wget https://raw.githubusercontent.com/MicrosoftDocs/mslearn-introduction-to-machine-learning

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

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

Data Exploration

The `avalanche` field is our target. A value of `1` means that an avalanche did occur at the conditions described by the features, whereas a value of `0` means no avalanche hapenned. Since our targets can only be `0` or `1` we call this a *binary classification* model.

Now let's plot the relationships between each feature and the target values. That helps us understand which features are more likely to influence the results:

```
import graphing # custom graphing code. See our GitHub repo for details

graphing.box_and_whisker(dataset, label_x="avalanche", label_y="surface_hoar", show=True)
graphing.box_and_whisker(dataset, label_x="avalanche", label_y="fresh_thickness", show=True)
graphing.box_and_whisker(dataset, label_x="avalanche", label_y="weak_layers", show=True)
graphing.box_and_whisker(dataset, label_x="avalanche", label_y="no_visitors")
```

We can notice that:

- For `fresh_thickness` the outcomes are very similar. This means that variations in their values aren't strongly correlated with the results.
- Variations in values for `weak_layers` and `no_visitors`, seem to correlate with a larger number of `avalanche` results, and thus we should assign more importance to these features.

The differences between avalanche and non-avalanche days are small and there isn't one clear driver of issues. Weak layers looks like a good starting point as it is related to the widest variation in results.

Building a simple logistic regression model

We will now built and train a model to predict the chance of an avalanche happening based **solely** on the number of weak layers of snow:

```
# Here we import a function that splits datasets according to a given ratio
from sklearn.model_selection import train_test_split
```

```
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)
```

OK, lets train our model, using the `train` dataset we've just created (notice that `weak_layers` will be the only feature used to determine the outcome):

```
import statsmodels.formula.api as smf
import graphing # custom graphing code. See our GitHub repo for details

# Perform logistic regression.
model = smf.logit("avalanche ~ weak_layers", train).fit()
```

```
print("Model trained")
```

After training, we can print a model summary with very detailed information:

```
print(model.summary())
```

Notice that the positive coefficient for `weak_layers` means that a higher value indicates a higher likelihood for an avalanche.

Using our model

We can now use our trained model to make predictions and estimate probabilities.

Let's pick the first four occurrences in our `test` set and print the probability of an avalanche for each one of them:

```
# predict to get a probability

# get first 3 samples from dataset
samples = test["weak_layers"][:4]

# use the model to get predictions as possibilities
estimated_probabilities = model.predict(samples)

# Print results for each sample
for sample, pred in zip(samples,estimated_probabilities):
    print(f"A weak_layer with value {sample} yields a {pred * 100:.2f}% chance of an avalanche
```

Let's plot out model to understand this:

```
# plot the model
# Show a graph of the result
predict = lambda x: model.predict(pandas.DataFrame({"weak_layers": x}))

graphing.line_2D([("Model", predict)],
                 x_range=[-20,40],
                 label_x="weak_layers",
                 label_y="estimated probability of an avalanche")
```

The line plots the function of the **probability** of an avalanche over the number of weak layers; Notice that the more weak layers, the more likely an avalanche will happen. This plot can look a bit confusing for two reasons.

Firstly, the curve can make predictions from negative to positive infinity, but we only have data for 0 - 10 layers:

```
print("Minimum number of weak layers:", min(train.weak_layers))
print("Maximum number of weak layers:", max(train.weak_layers))
```

This is because logistic regression models allow predictions outside the range of values they have seen, and sometimes do so quite well.

```
import numpy as np

# Get actual rates of avalanches at 0 layers
avalanche_outcomes_for_0_layers = train[train.weak_layers == 0].avalanche
print("Average rate of avalanches for 0 weak layers of snow", np.average(avalanche_outcomes_fo

# Get actual rates of avalanches at 10 layers
avalanche_outcomes_for_10_layers = train[train.weak_layers == 10].avalanche
print("Average rate of avalanches for 10 weak layers of snow", np.average(avalanche_outcomes_f
```

Our model is actually doing a good job! It's just that avalanches aren't *only* caused by weak layers of snow. If we want to do better, we probably need to think about including other information in the model.

Classification or decision thresholds

To return a binary category (`True` = "avalanche", `False` = "no avalanche") we need to define a *Classification Threshold* value. Any probability above that threshold is returned as the positive category, whereas values below it will be returned as the negative category.

Let's see what happens if set our threshold to `0.5` (meaning that our model will return `True` whenever it calculates a chance above 50% of an avalanche happening):

```
# threshold to get an absolute value
threshold = 0.5

# Add classification to the samples we used before
for sample, pred in list(zip(samples,estimated_probabilities)):
    print(f"A weak_layer with value {sample} yields a chance of {pred * 100:.2f}% of an avalan
```

Note that a `0.5` threshold is just a starting point that needs to be tuned depending on the data we're trying to classify.

Performance on test set

Now let's use our `test` dataset to perform a quick evaluation on how the model did. For now, we'll just look at how often we correctly predicted if there would be an avalanche or not

```
# Classify the mdel predictions using the threshold
predictions = model.predict(test) > threshold

# Compare the predictions to the actual outcomes in the dataset
```

```
# Compare the predictions to the actual outcomes in the dataset
accuracy = np.average(predictions == test.avalanche)

# Print the evaluation
print(f"The model correctly predicted outcomes {accuracy * 100:.2f}% of time.")
```

Avalanches are hard to predict but we're doing ok. It's hard to tell exactly what kind of mistakes our model is making, though. We'll focus on this in the next exercise.

No compute Compute not connected Viewing Kernel not connected

Next unit: Assessing a classification model

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆