

# Introduction to Natural Language Processing 01

## Lab 03

### Introduction

*The project is a continuation of what we started on the second lab. You will train a logistic regression classifier on manually extracted features.*

We need to import plotting, data management, machine learning and mathematic libraries.

```
In [1]: from typing import Tuple, Any

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from sklearn.model_selection import train_test_split
from datasets import load_dataset

import torch
from torch import nn, Tensor

sns.set_theme()
%matplotlib inline
```

### Features

For every given text, we want to generate a vector with the features seen in class.

(6 points) Code the following features:

- 1 if "no" appears in the document, 0 otherwise.
- The count of first and second pronouns in the document.
- 1 if "I" is in the document, 0 otherwise.
- Log(word count in the document).
- Number of words in the document which are in the positive lexicon.
- Number of words in the document which are in the negative lexicon.
- [Bonus] Add another feature of your choice.

In order to achieve this, we first need to apply some preprocessing over our data to ease the feature extraction.

```
In [2]: from string import punctuation
import re

def preprocessing(data : str) -> str:
    """
    Preprocess the data by removing punctuation, converting to lowercase and removing extra spaces
    The function does not remove the "-" dash symbol

    For example : son-in-law * son in law

    :param data: string data to preprocess
    :return: preprocessed string data
    """
    data = data.replace("<br />", "")
    toremove = punctuation.replace("-", "")
    translator = str.maketrans(toremove, ' ' * len(toremove))
    res = data.lower().translate(translator).strip()
    res = re.sub(r'\s+', ' ', res)
    return res
```

Then we need to have a dictionary that can give a "sentiment" value, VADER will do the trick.

```
In [3]: def lexicon_dict(path : str = "vader_lexicon.txt") -> dict:
    """
    Open the vader_lexicon.txt file and construct the dictionary.
    Implementation is inspired by the one in the vaderSentiment package.

    :return: Dictionary containing the words in the lexicon as keys and their sentiment as values.

    lex_dict = {}
    with open(path, encoding='utf-8') as file:
        content = file.read()

        for line in content.rstrip('\n').split('\n'):
            word, measure = line.strip().split('\t')[0:2]
            lex_dict[word] = float(measure)

    return lex_dict
```

Now let's implement the features vector function, which takes a piece of text and returns its feature vector. **We also added a bonus feature ("I" count).**

```
In [4]: def features_vector(text : str, use_bonus : bool, lex_dict : dict) -> np.ndarray:
    """
    Return the features vector of the text with each component representing the following features:
    - 1 if "no" appears in the document, 0 otherwise.
    - The count of first and second pronouns in the document.
    - 1 if "I" is in the document, 0 otherwise.
    - Log(word count in the document).
    - Number of words in the document which are in the positive lexicon.
    - Number of words in the document which are in the negative lexicon.

    Assuming that basic preprocessing has been done on the text (punctuation removed and lowercase).

    :param text: The text to extract the features from.
    :param dict: A sentiment analysis lexicon
    :return: A numpy array of shape (6,) containing the features.
    """
    splitted_text = list(filter(lambda s : s != '', text.split()))

    no_present = int('f' no ' in f' (text) ) # Faster than splitted
    pronouns_count = sum((word in ('I', 'you') for word in splitted_text))
    exclamation_present = int('f' ' in f' (text) )
    log_word_count = np.log(len(splitted_text))

    # Check every word appearance in the lexicon and get its sentiment value
    sentiment_array = np.vectorize(lambda x: lex_dict.get(x, 0.0))(text.split())
    positive_count = np.sum(sentiment_array >= 0.05)
    negative_count = np.sum(sentiment_array <= -0.05)

    if use_bonus: # Bonus Feature
        bonus_exclamation_count = text.count("!")
        return np.array([
            no_present,
            pronouns_count,
            exclamation_present,
            log_word_count,
            positive_count,
            negative_count,
            bonus_exclamation_count])
    return np.array([
        no_present,
        pronouns_count,
        exclamation_present,
        log_word_count,
        positive_count,
        negative_count])
```

Let's test it on a piece of example text

```
In [5]: text = "I loved this excellent movie"
text = preprocessing(text)
```

We get the following feature vector (bonus ignored)

```
In [6]: features_vector(text, False, lexicon_dict())

Out[6]: array([0.         , 1.         , 0.         , 1.60943791, 2.         ,
        0.         ])
```

### Logistic regression classifier

We need to freeze the random seed to get this experiment to be deterministic.

```
In [7]: RANDOM_SEED = 42
np.random.seed(seed=RANDOM_SEED) # Set the seed for reproducibility
torch.manual_seed(RANDOM_SEED)
```

```
Out[7]: <torch._C.Generator at 0x111353766f0>
```

(3 points) Adapt the code by adding your feature extractor and train a classifier.

For training, don't use the test set as validation. Instead, split the training set into a training and a validation set (use 10 to 20% of the training set as validation).

Here is a function to get the preprocessed dataset

```
In [8]: def get_processed_data(dataset_name : str, split : str) -> tuple(torch.Tensor, torch.Tensor):
    """
    Load a split from the hugging face dataset and preprocess it.

    :param dataset_name: Name of a hugging face dataset (e.g. "imdb")
    :param split: The split to load (e.g. "train" or "test")
    :return: A tuple containing the feature vectors and the labels as torch tensors.
    """

    # Load data with hugging face dataset
    train_data = load_dataset('imdb', split=split)
    train_data = pd.DataFrame(train_data)

    # Apply preprocessing
    train_data["text"] = train_data["text"].apply(preprocessing)

    lex_dict = lexicon_dict()

    # Transform to feature vector
    X = torch.tensor(np.vectorize(
        lambda text: features_vector(text, False, lex_dict), signature='(n)' )(np.array(train_data["text"]).t
        dtype=torch.float32))
    y = torch.tensor(train_data['label']).to_numpy(), dtype=torch.float32).reshape(-1, 1)

    return X, y
```

Next, let's save them into variables.

```
In [9]: X_train, y_train = get_processed_data("imdb", "train")

Downloading readme:   0% | 0.00/7.59k [00:00<?, 7B/s]
Found cached dataset imdb (C:/Users/Tom/.cache/huggingface/datasets/imdb/plain_text/1.0.0/d613c88cf8fa3bab83b4d
ed3713f1f74830d1100e171db75bbddb80b3345c9c0)

In [10]: X_test, y_test = get_processed_data("imdb", "test")

Found cached dataset imdb (C:/Users/Tom/.cache/huggingface/datasets/imdb/plain_text/1.0.0/d613c88cf8fa3bab83b4d
ed3713f1f74830d1100e171db75bbddb80b3345c9c0)

In [11]: X_train, X_valid, y_train, y_valid = train_test_split(
    X_train,
    y_train,
    test_size=0.2,
    stratify=y_train,
    random_state=RANDOM_SEED,
)
```

Here is the LogisticRegression class, inspired from the teacher's implemtation, that has been adapted to our needs. It allows us to instantiate a model that can be manipulated easily.

```
In [12]: class LogisticRegression(nn.Module):
    """A logistic regression implementation"""

    def __init__(self, input_dim: int, nb_classes: int) -> None:
        """
        Args:
            input_dim: the dimension of the input features.
            nb_classes: the number of classes to predict.

        super().__init__()
        # output_layer = nn.Sigmoid() if nb_classes == 1 else nn.Softmax()
        self.classifier = torch.nn.Sequential(
            nn.Linear(input_dim, nb_classes),
            # output_layer,
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Args:
            x: the input tensor.

        Returns:
            The output of activation function.

        """
        return self.classifier(x)
```

We instantiate our model and choose its loss function and optimizer, we use the same as the teacher's implementation on binary classification.

```
In [13]: model = LogisticRegression(X_train.shape[1], 1)
criterion = nn.BCEWithLogitsLoss()
# Stochastic gradient descent
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, weight_decay=0.5)
```

Now, let's define a train function, and trigger the training for 1000 epochs. The training function will output its loss each 100 epochs.

```
In [14]: %time

n_epochs = 1000

def train(verbose=True):
    """
    Performs gradient descent using the specified optimizer
    and loss criterion.

    Args:
        verbose (bool, optional): If True, prints the training
        loss for every 100th epoch. Default is True.

    Returns:
        train_losses (list): A list of training losses for each epoch.
        test_losses (list): A list of validation losses for each epoch.
    """
    train_losses = []
    test_losses = []

    # Training loop
    for epoch in range(n_epochs):
        # Setting all gradients to zero.
        optimizer.zero_grad()

        # Sending the whole training set through the model.
        predictions = model(X_train)
        # Computing the loss.
        loss = criterion(predictions, y_train)
        train_losses.append(loss.item())
        if verbose and epoch % 100 == 0:
            print(loss)

        # Computing the gradients and gradient descent.
        loss.backward()
        optimizer.step()

        # When computing the validation loss, we do not want to update the weights.
        # torch.no_grad tells PyTorch to not save the necessary data used for
        # gradient descent.
        with torch.no_grad():
            predictions = model(X_valid)
            loss = criterion(predictions, y_valid)
            test_losses.append(loss)
        return train_losses, test_losses

# Keeping an eye on the losses
train_losses, test_losses = train()

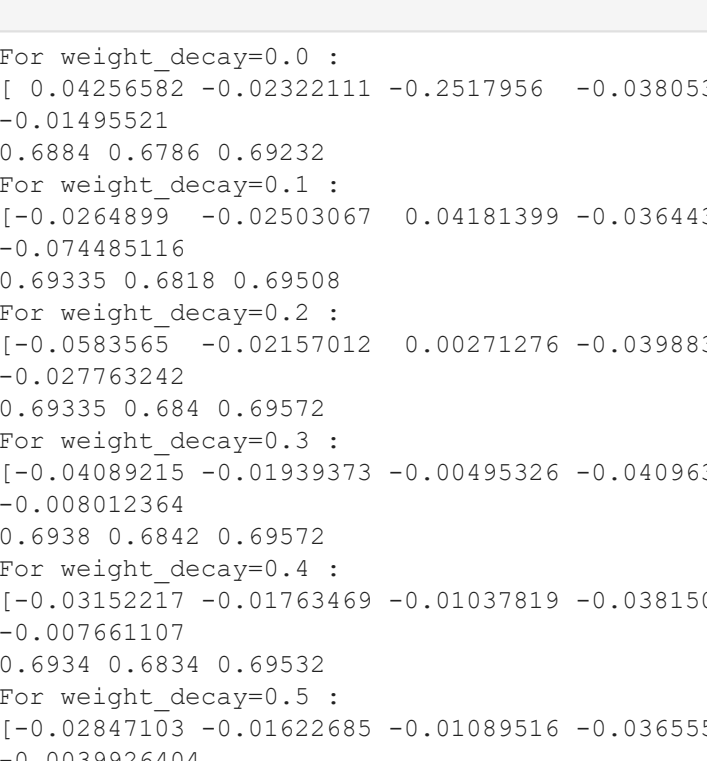
CPU times: total: 0 ns
Wall time: 0 ns
tensor(1.2303, grad_fn=<BinaryCrossEntropyWithLogitsBackward0>)
tensor(0.6148, grad_fn=<BinaryCrossEntropyWithLogitsBackward0>)
tensor(0.6066, grad_fn=<BinaryCrossEntropyWithLogitsBackward0>)
tensor(0.6054, grad_fn=<BinaryCrossEntropyWithLogitsBackward0>)
tensor(0.6050, grad_fn=<BinaryCrossEntropyWithLogitsBackward0>)
tensor(0.6048, grad_fn=<BinaryCrossEntropyWithLogitsBackward0>)
tensor(0.6047, grad_fn=<BinaryCrossEntropyWithLogitsBackward0>)
tensor(0.6046, grad_fn=<BinaryCrossEntropyWithLogitsBackward0>)
tensor(0.6046, grad_fn=<BinaryCrossEntropyWithLogitsBackward0>)
tensor(0.6045, grad_fn=<BinaryCrossEntropyWithLogitsBackward0>)
```

(1 point) Evaluate your classifier in terms of accuracy for the training, validation, and test set.

We need to evaluate the trained model on each dataset, first let's view the training curves for the training and test losses.

```
In [15]: # Checking the losses
plt.plot(np.arange(len(train_losses)), train_losses, label="Training loss")
plt.plot(np.arange(len(test_losses)), test_losses, label="Test loss")
plt.legend()

Out[15]: <matplotlib.legend.Legend at 0x1111317c370>
```



Besides, we implement a test function, which outputs the accuracy for the train, test and validation datasets, using the trained model. Next we trigger it.

```
In [16]: def test():
    """
    Runs a test on train, valid and test datasets.

    """
    # Computing the accuracy for our 3 splits.
    with torch.no_grad():
        p_train = torch.sigmoid(model(X_train))
        p_train = np.round(p_train.numpy())
        training_accuracy = np.mean(p_train == y_train.numpy())
        p_valid = torch.sigmoid(model(X_valid))
        p_valid = np.round(p_valid.numpy())
        valid_accuracy = np.mean(p_valid == y_valid.numpy())
        p_test = torch.sigmoid(model(X_test))
        p_test = np.round(p_test.numpy())
        test_accuracy = np.mean(p_test == y_test.numpy())

    print(training_accuracy, valid_accuracy, test_accuracy)

test()

0.6934 0.6838 0.69552
```

The test accuracy, which is the most important, is drawing near 70% accuracy.

(1 point) Look at the weights of your classifier. Which features seems to play most for both classes?

Let's view the different model internal parameters.

```
In [17]: for name, param in model.named_parameters():
    print(f"{name} : {param}")

classifier.0.weight : Parameter containing:
tensor([[-0.0279, -0.0162, -0.0124, -0.0358, 0.1257, -0.1584]],
        requires_grad=True)
classifier.0.bias : Parameter containing:
tensor([-0.0073], requires_grad=True)
```

Here, we implement a fancy bar plot displaying what are the most prominent weights.

```
In [18]: def plot_model_parameters(model : LogisticRegression) -> None:
    """
    Plots the features importance of features used in the Logistic Regression model

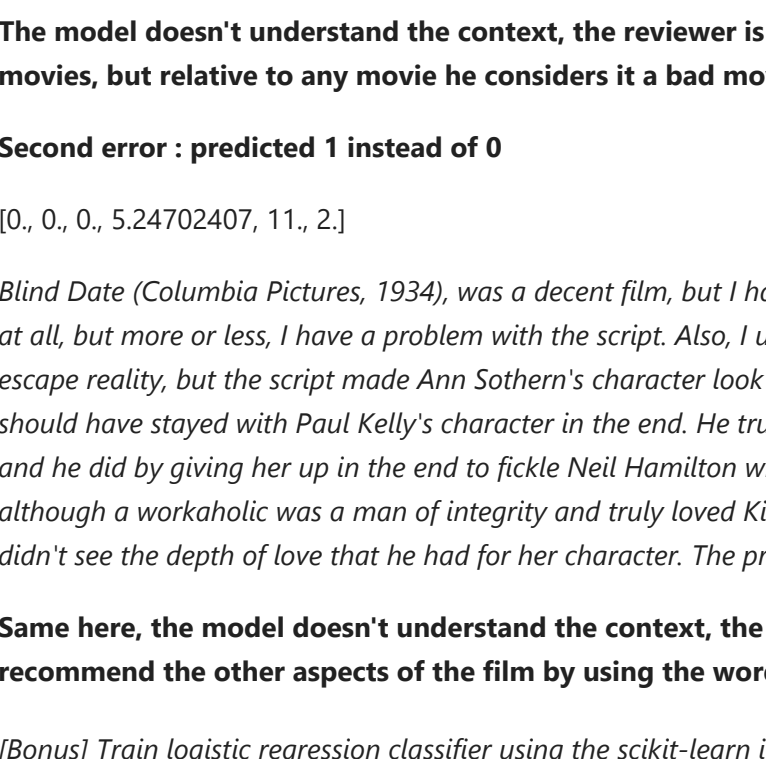
    :param model: Logistic Regression object used to train the model
    :return: None (A plot will be displayed)
    """

    Y = []
    for param in model.parameters():
        for weight in param:
            break

    Y = Y[0]
    X = list(range(len(Y)))
    fig, ax = plt.subplots()

    plt.xlabel("Features")
    plt.ylabel("Weights")
    plt.title("Feature Importance")
    ax.set_xticks(np.arange(6), labels=["no_present", "exclamation_present", "log_word_count", "sentiment_array", "positive_count", "negative_count"])
    plt.xticks(rotation=45)
    plt.bar(X, Y)
    plt.show()
```

```
In [19]: plot_model_parameters(model)
```



As we can see, the greatest weights are both word counting features. Positive and negative word count. They affect both classes.

[Bonus] The parameter `weightdecay` of the SGD optimizer corresponds to the L2 penalty. Try playing with this value and explain how it influence the model's weights.

We will iterate over [0.0, 1.0] - 10 times to check the accuracy and model weight evolution.

```
In [20]: for i in range(0, 10 + 1):
    model = LogisticRegression(X_train.shape[1], 1)
    criterion = nn.BCEWithLogitsLoss() # Binary cross entropy
    # Stochastic gradient descent

    weight_decay = i / 10.0

    optimizer = torch.optim.SGD(model.parameters(), lr=0.01, weight_decay=weight_decay)
    train(False)
    print(f"for weight_decay={weight_decay} :")
    for param in model.parameters():
        for weight in param:
            print(weight.detach().numpy())
            break

    test()

For weight_decay=0.0 :
0.04256582 -0.02322111 -0.2517956 -0.03805369 0.15725332 -0.19587041]
-0.01495521
0.6884 0.6786 0.69232
For weight_decay=0.1 :
[-0.0264899 -0.02503067 0.04181399 -0.03644374 0.14796579 -0.1852033 ]
-0.07485116
0.69335 0.6818 0.69508
For weight_decay=0.2 :
[-0.058365 -0.02157012 0.00271276 -0.03988392 0.14153051 -0.17614892]
-0.027763242
0.69235 0.684 0.69572
0.6935 0.684 0.69572
For weight_decay=0.3 :
[-0.04089215 -0.01939373 -0.00495326 -0.0409639 0.13577758 -0.1696832 ]
-0.08012364
0.6938 0.6842 0.69572
For weight_decay=0.4 :
[-0.03525217 -0.01763469 -0.01037819 -0.03815006 0.13047422 -0.1638469 ]
-0.00766107
0.6934 0.6834 0.69532
For weight_decay=0.5 :
[-0.02847103 -0.01622685 -0.01089516 -0.03655519 0.12572795 -0.1583502 ]
-0.039926404
0.6935 0.6838 0.69548
For weight_decay=0.6 :
[-0.02584398 -0.01506301 -0.01050031 -0.03427789 0.12138501 -0.15335774]
-0.004899669
0.6926 0.6842 0.69472
For weight_decay=0.7 :
[-0.02249279 -0.01408009 -0.00893397 -0.03246563 0.11742976 -0.14878292]
-0.005160769
0.6926 0.6842 0.69472
For weight_decay=0.8 :
[-0.02039512 -0.01321185 -0.00821514 -0.03080598 0.11379849 -0.14451817]
-0.0052521317
0.69235 0.6848 0.69498
For weight_decay=0.9 :
[-0.01856066 -0.01245014 -0.00742422 -0.02938523 0.11045057 -0.14054684]
-0.005008611
0.69215 0.6848 0.69508
For weight_decay=1.0 :
[-0.01713477 -0.01177222 -0.00683577 -0.02807334 0.10734612 -0.13683279]
-0.0048213834
0.69235 0.6836 0.69532
```

Increasing weight decay lead to smaller model weights and it doesn't affect accuracy.

(1 point) Take two wrongly classified samples in the test set and try explaining why the model was wrong.

We need to use a new and fresh model and train it.

```
In [21]: model = LogisticRegression(X_train.shape[1], 1)
criterion = nn.BCEWithLogitsLoss() # Binary cross entropy
# Stochastic gradient descent

weight_decay = i / 10.0

optimizer = torch.optim.SGD(model.parameters(), lr=0.01, weight_decay=1.0)
train(False); # To avoid print
```

Now that its trained we can pick two wrongly classified examples and analyse them.

```
In [22]: test_set = load_dataset("imdb", split="test")
test_set = pd.DataFrame(test_set)

predicted = np.round(model(X_test).detach().numpy()[1,:]).astype(int)

test_set["predicted"] = predicted
wrongs = test_set[test_set["label"] != test_set["predicted"]].head(2)

for index, data in wrongs.iterrows():
    text, label, predicted = data
    print(features_vector(text, False, lexicon_dict()))
    print(text, label, predicted)

Found cached dataset imdb (C:/Users/Tom/.cache/huggingface/datasets/imdb/plain_text/1.0.0/d613c88cf8fa3bab83b4d
ed3713f1f74830d1100e171db75bbddb80b3345c9c0)
0.         0.         0.         15.         15.         ]
Without the entertainment value of a rental, especially if you like action movies. This one features the usual ca
r races, fights with the great Van Damme kick style, shooting battles with the 40 shell load shotgun, and even
terraser, style bombs. All of this is entertaining and competently handled but there is nothing that really bit
we you away if you've seen your share before.<br /><br />The plot is made interesting by the inclusion of a rab
bit, which is clever but hardly profound. Many of the characters are heavily stereotyped -- the angry veterans,
the terrified alien aliens, the crooked cops, the indifferent feds, the bitchy tough lady the station head, the c
rooked politician, the fat federale who looks like he was typecast as the Mexican in a Hollywood movie from the
1940s. All passably acted but again nothing special.<br /><br />I thought the main villains were pretty well do
ne and fairly well acted. By the end of the movie you certainly knew who the good guys were and weren't. There
was an emotional lift as the really bad ones got their just deserts. Very simplistic, but then you weren't expe
cting Hamlet, right? The only thing I found really annoying was the constant cuts to VDS authors during the la
st fight scene.<br /><br />Not bad. Not good. Passable 4, 0 -1
[0.         0.         0.         5.37989735 3.         5.         ]
Its a totally average film with a few semi-interesting action sequences that make the plot seem a little better
and remind the viewer of the classic van dam films. parts of the plot don't make sense and seem to be added in to
u se up time. the end plot is that of a very basic type that doesn't leave the viewer guessing and any twists
are obvious from the beginning. the end scene with the flask backs don't make sense as they are added in and seem
to have little relevance to the history of van dam's character. not really worth watching again, bit disappointe
d in the end production, even though it is apparent it was shot on a low budget certain shots and sections in t
he film are of poor directed quality 0 -1
```

First error : predicted 1 instead of 0

[0, 3, 1, 4, 8978398, 13, 1]

First off let me say, if you haven't enjoyed a Van Damme movie since bloodsport, you probably will not like this movie. Most of these movies may not have the best plots or best actors but I enjoy these kinds of movies for what they are. This movie is much better than any of the movies the other action guys (Sagal and Dolph) have thought about putting out the past few years. Van Damme is good in the movie, the movie is only worth watching to Van Damme fans. It is not as good as Wake of Death (which i highly recommend to anyone of likes Van Damme) or In hell but, in my opinion it's worth watching. It has the same type of feel to it as Nowhere to Run. Good fun stuff!

The model doesn't understand the context, the reviewer is a hard Van Damme fan, he compares the movie from other Van Damme movies, but relative to any movie he considers it a bad movie.

Second error : predicted 1 instead of 0

[0, 0, 0, 5, 24702407, 11, 2]

Blind Date (Columbia Pictures, 1934) was a decent film, but I have a few issues with this film. First of all, I don't fault the actors in this film at all, but more or less, I have a problem with the script. Also, I understand that this film was made in the 1930's and people were looking to escape reality, but the script made Ann Sothern's character look weak. She kept going back and forth between suitors and I felt as though she should have stayed with Paul Kelly's character in the end. He truly did care about her and her family and would have done anything for her and he did by giving her up in the end to kickle Neil Hamilton who in my opinion was only out for a good time. Paul Kelly's character, although a workaholic was a man of integrity and truly loved Kitty (Ann Sothern) as opposed to Neil Hamilton, while he did like her a lot, I didn't see the depth of love that he had for her character. The production values were great, but the script could have used a little work.

Same here, the model doesn't understand the context, the reviewer explains the plot and critiques it. However he tries to still recommend the other aspects of the film by using the words "decent", "great", but overall dislikes the movie.

[Bonus] Train logistic regression classifier using the scikit-learn implementation. How does it compare with the PyTorch version?

```
In [23]: from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

logistic_reg = LogisticRegression()
```

```
In [24]: logistic_reg.fit(X_train, y_train.ravel())

Out[24]: LogisticRegression()
```

```
In [25]: preds = logistic_reg.predict(X_valid)

In [26]: accuracy_score(y_valid, preds)

Out[26]: 0.6816
```

The accuracy 0.68 is the same as with the torch Logistic Regression compared with a really consise code, thanks to the sklearn's API.