NHL STENDEN
hogeschool

Applied

Text Classification

Guest lecture by Saul Johnson

Who am I?

I'm Saul, a final-year information security Ph.D. candidate at Teesside University in the north-east of England.

I'm also Head of Software Engineering at BreachLock B.V. an Amsterdam-based offensive security firm.

GitHub: @lambdacasserole
Twitter: @lambdacasserole
Website: https://sauljohnson.com/
Linkedin: https://www.linkedin.com/in/sauljohnson/

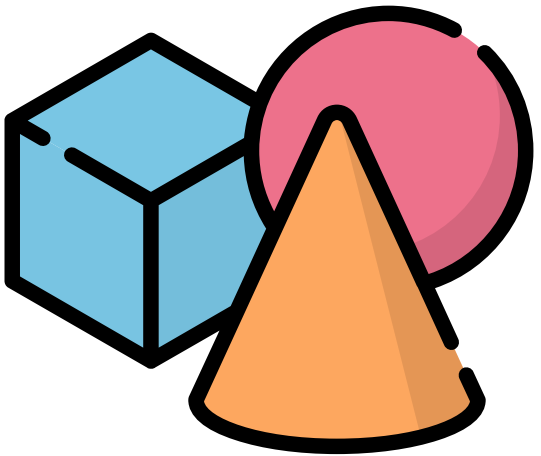
NHL STENDEN hogeschool

Text classification? What's that?

- Text classification refers to the task of automatically sorting text inputs into different categories.

- For example, if we're given one big folder full of text files containing recipes—some for main course dishes and some for desserts—can we automatically sort these files into 2 different folders?

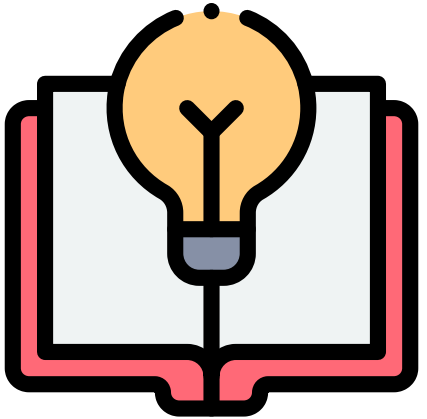
- Sure, we could look for key words in the text and sort like that, but a much more flexible and reliable approach is to deploy machine learning!



What are we going to cover?

By the end of this session, we'll aim to:

- **Understand** what text classification is and how it's useful.
- **Know** how to apply Python and SciKit Learn to text classification problems.
- **Be familiar in-depth** with the Naive Bayes classifier, and be aware of support vector machine and random forest classifiers.
- **Be able to** deploy each of these models to solve real-world text classification problems, and evaluate their accuracy.
- **Know** how to source and process training data, and where to access further resources and reading.



Tools and Technologies

We're going to use a very restrained set of tools and technologies today, and keep things highly applied (just the essential theory).

• Our programming language of choice will be Python.

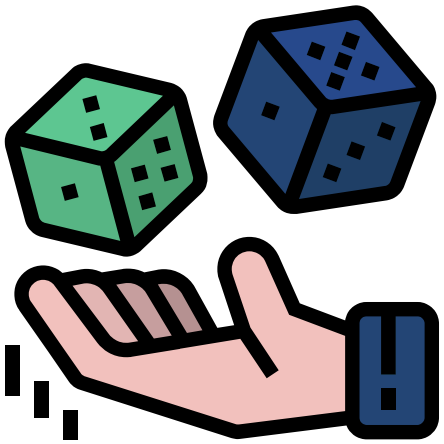
• We'll be using the SciKit Learn library for our text classification models it also provides a ton of useful tools and helper functions to make training and interacting with text classification models easier!

NHL STENDEN
hogeschool

Naive Bayes Classifiers

Examining a text classification model in-depth...

Naive Bayes: A Rapid Primer!



One of the easiest text classification models to understand in-depth is called "Naive Bayes". It has a strange name (we'll come back to it later) but a number of advantages:

• It's fast, and can be trained and queried very quickly

• It has surprisingly good performance in a whole bunch of different applications. From automatically detecting the language a document is written in to sorting positive from negative movie reviews, Naive Bayes has you covered!

It does have cons, which we'll discuss later on.

Case Study: Spam or not? [1/5]

Let's take a look at how we can train a Naive Bayes model to filter spam from non-spam messages in our email inboxes! This is a super useful and common deployment of text classification models.

Let's imagine we have 20 spam and 15 non-spam messages in our training data set...

Non-spam x15

Spam x20

Case Study: Spam or not? [2/5]

Even without looking inside these messages, we already know useful information! We receive more spam than non-spam. We therefore have a starting probability that a new, ***unseen*** message that arrives in our inbox is spam! We call these probabilities our ***prior probabilities (just "priors" for short)***.

$$P(Spam) = \frac{count(Spam)}{count(Spam) + count(Nonspam)} = \frac{20}{20 + 15} = 0.571$$

P(Non-spam) = 0.429
(1 – 0.571)

P(Spam) = 0.571

NHL STENDEN hogeschool

Case Study: Spam or not? [3/5]

Now, let's actually get into the contents of the messages themselves. Let's build a ***frequency distribution*** of how often each word occurs in spam messages vs non-spam messages. For simplicity, we'll pretend that each message is made up of just 5 different words: "hey", "beer", "cash", "friend" and "pharmacy"

"hey" occurs **8** times
"beer" occurs **7** times
"cash" occurs **2** times
"pharmacy" occurs **0** times

Non-spam word frequencies

"hey" occurs **3** times
"beer" occurs **1** time
"cash" occurs **9** times
"pharmacy" occurs **8** times

Spam word frequencies

NHL STENDEN hogeschool

Case Study: Spam or not? [3/5]

Now, let's actually get into the contents of the messages themselves. Let's build a ***frequency distribution*** of how often each word occurs in spam messages vs non-spam messages. For simplicity, we'll pretend that each message is made up of just 4 different words: "hey", "beer", "cash" and "pharmacy"

"hey" occurs **8** times
"beer" occurs **7** times
"cash" occurs **2** times
"pharmacy" occurs **0** times

Non-spam word frequencies

Total words in all non-spam:
8 + 7 + 2 + 0 = 17

"hey" occurs **3** times
"beer" occurs **1** times
"cash" occurs **9** times
"pharmacy" occurs **8** times

Spam word frequencies

Total words in all spam:
3 + 1 + 9 + 8 = 21

NHL STENDEN hogeschool

Case Study: Spam or not? [3/5]

Now, let's actually get into the contents of the messages themselves. Let's build a ***frequency distribution*** of how often each word occurs in spam messages vs non-spam messages. For simplicity, we'll pretend that each message is made up of just 4 different words: "hey", "beer", "cash" and "pharmacy"

P("hey" | Non-spam) = 8 ÷ 17 = **0.47**
P("beer" | Non-spam) = 7 ÷ 17 = **0.41**
P("cash" | Non-spam) = 2 ÷ 17 = **0.18**
P("pharmacy" | Non-spam) = 0 ÷ 17 = **0**

Non-spam word **probabilities**

P("hey" | Spam) = 3 ÷ 21 = **0.14**
P("beer" | Spam) = 1 ÷ 21 = **0.05**
P("cash" | Spam) = 9 ÷ 21 = **0.43**
P("pharmacy" | Spam) = 8 ÷ 21 = **0.38**

Spam word **probabilities**

Case Study: Spam or not? [4/5]

Now, let's imagine we get a new ***unseen*** message in our inbox. It's just arrived! Now we need to classify it! Here's what it says:

"Hey! Cash? Beer!"

Further reading:
Laplace smoothing. What is it? Why is it useful?

Let's do the maths…

Assuming this is non-spam:
prior (**0.429**) × hey (**0.47**) × cash (**0.18**) × beer (**0.41**) = **0.01488**

Assuming this is spam:
prior (**0.571**) × hey (**0.14**) × cash (**0.43**) × beer (**0.05**) = **0.00171**

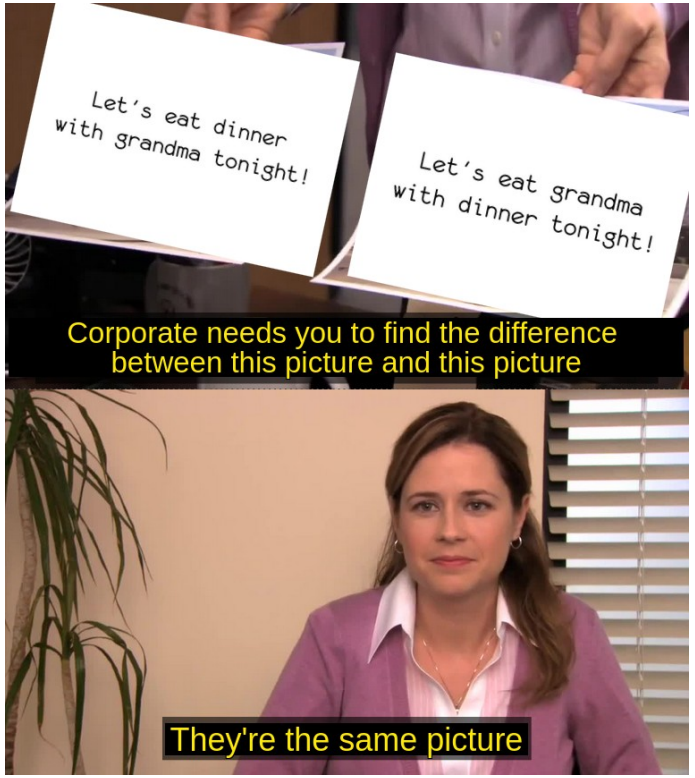
As the non-spam score is higher than the spam score, we classify it as...

Case Study: Spam or not? [5/5]

Not spam!

NHL STENDEN hogeschool

Why "Naive" Bayes?



Naive Bayes is named "naive" because it is completely ignorant of (i.e. *naive to*) word order. It treats text as an unordered "bag of words" which makes it fast and simple, but very unsuitable for classification based on language structures.

To a Naive Bayes model:

*"Let's eat **dinner** with **grandma** tonight!"*

Is the same thing as:

*"Let's eat **grandma** with **dinner** tonight!"*

NHL STENDEN
hogeschool

Two Alternative Models

*A **very high-level** overview of two other types of classifier!*

Support Vector Machines (SVMs)

Support vector machines are another machine learning model usable for text classification.

• To apply an SVM to our spam classification problem, we could situate text inputs at coordinates in feature space corresponding to their word frequencies.

• The SVM then finds the best way to draw a border (also called a *hyperplane*) separating the two classes.

• An unseen input can then be classified based on which side of the border it lands on!

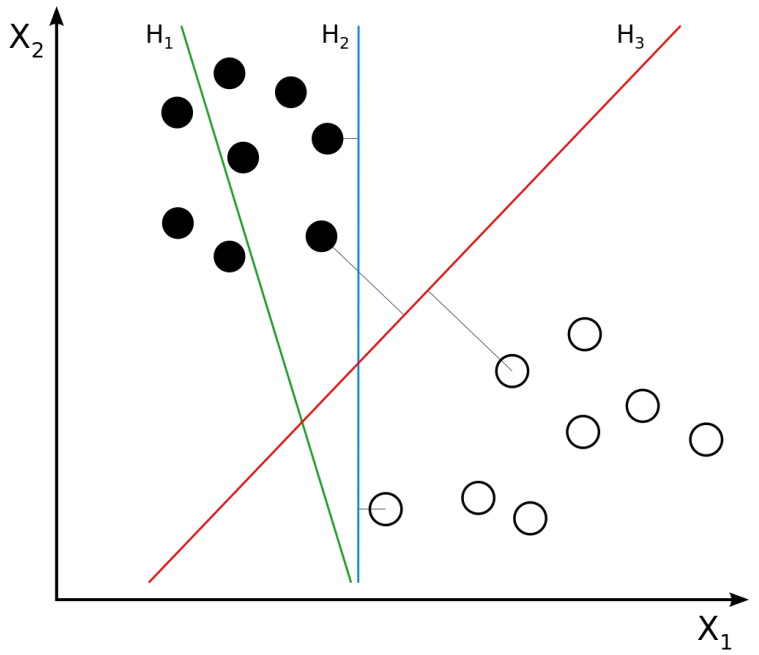


Figure from: https://commons.wikimedia.org/wiki/File:Svm_separating_hyperplanes_(SVG).svg

Random Forest (RF) Classifiers

Random forest classifiers can also be used in machine learning for text classification.

• To apply random forests in this context, we would take word frequencies as variables and construct a whole bunch (100s-1000s) of random decision trees from them.

• When we want to classify an unseen input, we run it through each decision tree and see which decision the majority of trees give as output.

• Whichever class the majority of trees decide on, that's our classification!

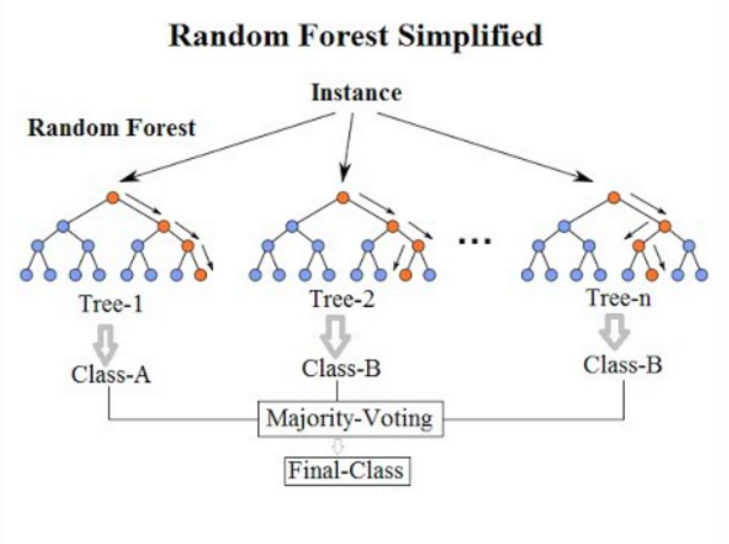


Figure from: https://commons.wikimedia.org/wiki/File:Random_forest_diagram_complete.png

NHL STENDEN hogeschool

NHL STENDEN
hogeschool

Data Acquisition

Let's go hunting for some data!

Kaggle: An Invaluable Resource!

You may have heard of a website called *Kaggle* before. It's a service run by Google that hosts publicly-available datasets that are ready to use in machine learning applications!

For training text classification models, it's a treasure-trove of ready-made datasets. It's completely free to sign up for an account and you should definitely do so it you haven't already!

Let's grab a labelled, ready to use SMS spam dataset from Kaggle now.

kaggle

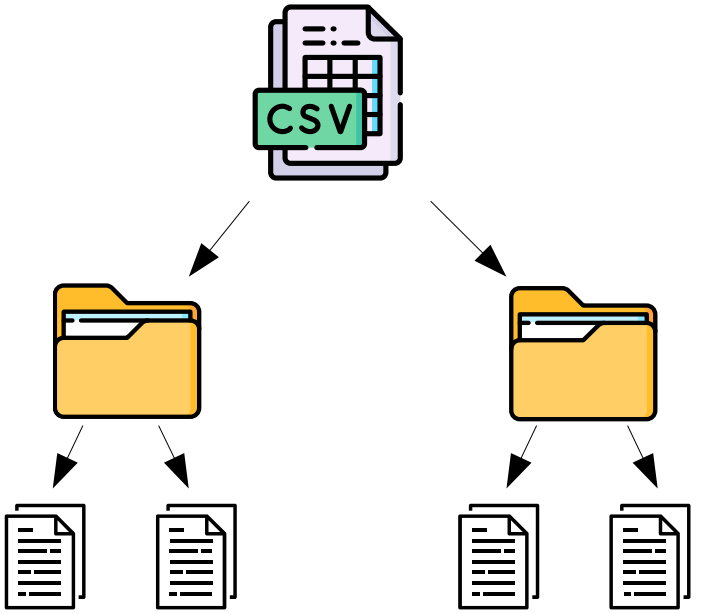
NHL STENDEN hogeschool

But... this is a CSV file!

Right now, the data is all together in one big CSV (comma-separated values) file. Let's split each training input into separate plain text files so we can take advantage of some of SciKit Learn's helper functions!

We're going to use Python's built-in CSV reading functionality

Another great choice for reading CSV files (and Excel files and more!) is a library called *Pandas*. This library is much more powerful than Python's in-built functionality, but for simplicity's sake we'll stick with the standard library approach for now.



NHL STENDEN
hogeschool

Time to write some code!

Let's build a text classification solution from the ground up!

But first! A note on GitHub!

All the source code from our work today will be hosted on my GitHub account here!

GitHub (owned by Microsoft) is a very popular platform for software developers, machine learning engineers, data science practitioners etc. to share their open-source code!

GitHub has a fantastic student developer pack with a ton of free giveaways. You can sign up here with your NHL Stenden email if you're interested:

https://education.github.com/pack

Prepping our data...

First, we use SciKit's load_files function, specifying a directory containing sub-folder.

We then split our data into testing and training sets, reserving a specific portion of data (in this case 20%) for testing our model and determining its accuracy.

```
# Load files in data directory, taking subdirectories as classes.
dataset = load_files(
    './data',
    load_content=True,
    encoding='UTF-8',
    decode_error='replace',
)

# Split into testing and training data.
x_train, x_test, y_train, y_test = train_test_split(
    dataset.data,
    dataset.target,
    test_size=0.2, # 20% test data.
)
```

Building a Pipeline

Then, we'll choose a classifier. Let's go with MultinomialNB (Naive Bayes) for now, but notice that SVM and RF classifiers are also shown here (but commented out).

We then create a *pipeline* around this classifier.

But what is a pipeline?

```
# Choose classifier (uncomment the lines below to select).
classifier = MultinomialNB() # Multinomial Naive Bayes.
# classifier = SGDClassifier() # Linear SVM.
# classifier = RandomForestClassifier() # Random forest.

# Create pipeline.
pipeline = Pipeline([
    ('vectorizer', CountVectorizer()), # Convert to token count matrix.
    ('tfidf', TfidfTransformer()), # TFIDF normalization.
    ('classifier', classifier)
])

# Actually train  model.
pipeline.fit(x_train, y_train)
```

Building a Pipeline

A pipeline is a very concise and powerful way to encode a machine learning workflow using SciKit Learn.

Here, we use CountVectorizer to convert words to frequencies, a TfidfTransformer to perform TFIDF normalization and finally, we drop the result into our chosen classifier and call pipeline.fit() to train the model!

```
# Choose classifier (uncomment the lines below to select).
classifier = MultinomialNB() # Multinomial Naive Bayes.
# classifier = SGDClassifier() # Linear SVM.
# classifier = RandomForestClassifier() # Random forest.

# Create pipeline.
pipeline = Pipeline([
    ('vectorizer', CountVectorizer()), # Convert to token count matrix.
    ('tfidf', TfidfTransformer()), # TFIDF normalization.
    ('classifier', classifier)
])

# Actually train  model.
pipeline.fit(x_train, y_train)
```

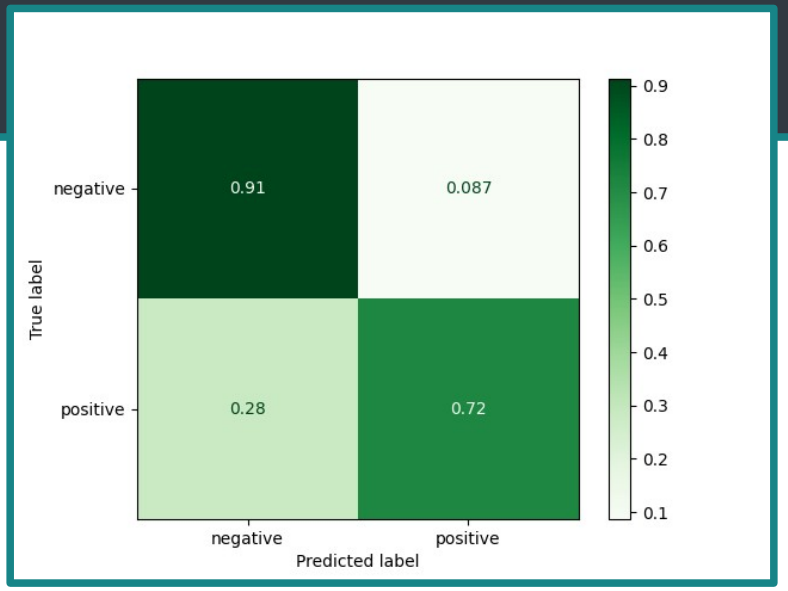
Further reading:
TFIDF normalization. What is it? Why is it useful?

NHL STENDEN hogeschool

Evaluating our work...

```
# Plot confusion matrix as image using matplotlib and show it.
plot_confusion_matrix(pipeline, x_test, y_test,
    display_labels=dataset.target_names,
    cmap=plt.cm.Greens,
    normalize='true')
plt.show()
```

Now we'll plot a ***confusion matrix***, which will give an at-a-glance overview of how well our model performs, broken down by true positives, true negatives, false positives and false negatives.



NHL STENDEN hogeschool

Saving our trained model for later!

```
# Pickle trained model to file for later use.
joblib.dump(pipeline, './classifier.pickle')
```

Now we've trained our model, let's save it for later so we don't have to re-train it every time we want to use it!

To do this, we'll save the Python object straight to disk. This process of taking a Python object, turning it into 1s and 0s and saving it to a file is called ***serialization***.

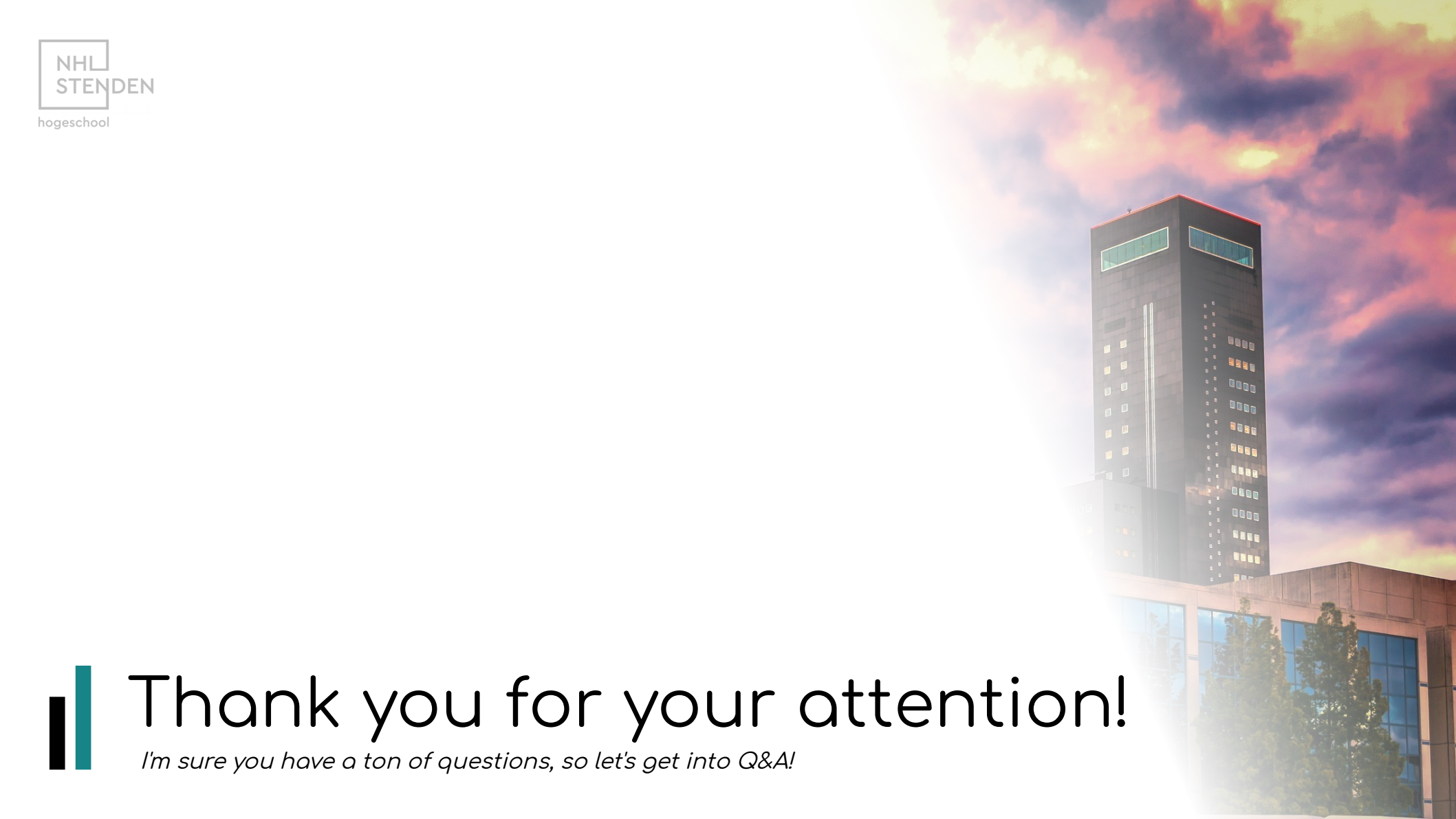
Python contains a serilization library called 'pickle', so we sometimes colloquially refer to serialization in Python as ***pickling***. We use a different library here, however, called ***joblib*** as it can be slightly faster for certain data science applications.

Interacting with our model

```
import os

from joblib import load


# List directories to get classes. Filter dotfiles and sort in alphabetical order.
classes = list(filter(lambda file: not file.startswith('.'), os.listdir('./data')))
classes.sort()

# Load trained model.
pipeline = load('./classifier.pickle')

# Interact with classifier.
print('Query the trained text classification model interactively (CTRL+C to exit):')
while True:
    result = pipeline.predict([input('input> ').strip()])
    print(classes[result[0]])
```

Now we can write another program to load our pickled pipeine from disk and interactively feed it data! Let's go ahead and write it now!

NHL STENDEN hogeschool



Thank you for your attention!

I'm sure you have a ton of questions, so let's get into Q&A!