Text Data: Natural Language Processing

Daniel Hardt

CBS

Copenhagen
Business School
HANDELSHØJSKOLEN

Outline

Question 1

Create a dataframe by reading the file *eng1000.csv* . Use value_counts to see the counts for each value of the column, *emotion*. We will create models to predict emotions based on texts. Assign X to the *text* column and y to the *emotion* column.

```
df = pd.read_csv("eng_1000.csv")
print("value counts",df['emotion'].value_counts())
X = df['text']
y = df['emotion']
```

```
value counts anticipation    408
joy             403
sadness         129
anger            54
fear              6
Name: emotion, dtype: int64
```

Question 2

Perform a train test split and then apply *Countvectorizer* to X_train and X_test.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

# Vectorizer
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer()
X_Trainvec = cv.fit_transform(X_train)
X_Testvec = cv.transform(X_test)
```

Question 3

Build a logistic regression classifier (we suggest these settings: solver='lbfgs', multi_class='auto'). Print score for train and test data.

```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(solver='lbfgs', multi_class='auto')
lr.fit(X_Trainvec, y_train)
lr_score = lr.score(X_Trainvec, y_train)
print("Train", lr_score)
lr_score = lr.score(X_Testvec, y_test)
print( "Test",lr_score)
```

```
Train 0.9853333333333333
Test 0.504
```

Question 4

In the above model, we used unigrams (single words) only. This is the default for count_vectorizer. Try with unigrams and bigrams, and also unigrams, bigrams and trigrams. You do this by setting ngram_range for count_vectorizer. Build a logistic regression model for each of these settings and report on the results.

```
cv = CountVectorizer(ngram_range=(1,3))
X_Trainvec = cv.fit_transform(X_train)
X_Testvec = cv.transform(X_test)
lr = LogisticRegression(solver='lbfgs', multi_class='auto')
lr.fit(X_Trainvec, y_train)
lr_score = lr.score(X_Trainvec, y_train)
print("Train", lr_score)
lr_score = lr.score(X_Testvec, y_test)
print( "Test",lr_score)
```

```
Train 0.996
Test 0.508
```

Question 5

Use dummy classifier with the default settings (most frequent class), and the uniform strategy (random guessing). Print the train and test results for each, to determine some baselines for comparison.

```
from sklearn.dummy import DummyClassifier

dummy = DummyClassifier()
dummy.fit(X_Trainvec, y_train)
dummy_score = dummy.score(X_Trainvec, y_train)
print("Train", dummy_score)
dummy_score = dummy.score(X_Testvec, y_test)
print( "Test",dummy_score)

dummy = DummyClassifier(strategy="uniform")
dummy.fit(X_Trainvec, y_train)
dummy_score = dummy.score(X_Trainvec, y_train)
print("Train", dummy_score)
dummy_score = dummy.score(X_Testvec, y_test)
print( "Test",dummy_score)
```

```
Train 0.3453333333333333
Test 0.324
Train 0.196
Test 0.212
```

Question 6

Use the TfidfTransformer, to create tfidf scores instead of frequency scores. You apply the TfidfTransformer on the vectors created by CountVectorizer, using the fit_transform method just as is done with CountVectorizer. Create a logistic regression model with the data produced by TfidfTranform, and report the scores on train and test.

```
from sklearn.feature_extraction.text import TfidfTransformer
```

```
cv = CountVectorizer(ngram_range=(1,3))
cv = CountVectorizer(ngram_range=(1,1))

X_Trainvec = cv.fit_transform(X_train)
X_Testvec = cv.transform(X_test)
tfidf = TfidfTransformer()
X_TrainTfidf = tfidf.fit_transform(X_Trainvec)
X_TestTfidf = tfidf.transform(X_Testvec)
```

```
lr = LogisticRegression(solver='lbfgs', multi_class='auto',random_state=0)

lr.fit(X_TrainTfidf, y_train)
lr_score = lr.score(X_TrainTfidf, y_train)
print("Train", lr_score)
lr_score = lr.score(X_TestTfidf, y_test)
print( "Test",lr_score)
```

```
Train 0.8226666666666667
Test 0.524
```

Question 7

Create a Pipeline, consisting of Countvectorizer, TfidfTransformer, and LogisticRegression. Apply the pipeline to the training data, just as in the previous question, and report results on train and test.

```
from sklearn.pipeline import Pipeline

pipe = Pipeline(
    [
        ("vect", CountVectorizer()),
        ("tfidf", TfidfTransformer()),
        ("clf", LogisticRegression(solver='lbfgs', multi_class='auto',random_state=0))
    ]
)
pipe.fit(X_train,y_train)
print("Train", pipe.score(X_train, y_train))
print( "Test",pipe.score(X_test, y_test))
```

```
Train 0.8226666666666667
Test 0.524
```

Question 8

Use the above pipeline with GridSearchCV. Use the following choices for parameters: for CountVectorizer, use ngram ranges of (1,1), (1,2), and (1,3). For TfidfTransformer set the *use_idf* parameter to True or False. Print the best score and best parameter choices.

```
from sklearn.model_selection import GridSearchCV

parameters = {
     "vect__ngram_range": ((1, 1), (1, 2), (1,3)),
     "tfidf__use_idf": (True, False)
  }

grid_search = GridSearchCV(pipe, parameters, n_jobs=-1, verbose=1,cv=3)

grid_search.fit(X_train, y_train)
print("Best score: %0.3f" % grid_search.best_score_)
print("Best parameters set:")
best_parameters = grid_search.best_estimator_.get_params()
for param_name in sorted(parameters.keys()):
    print("\t%s: %r" % (param_name, best_parameters[param_name]))
```

```
Fitting 3 folds for each of 6 candidates, totalling 18 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  18 out of  18 | elapsed:   15.2s finished
```

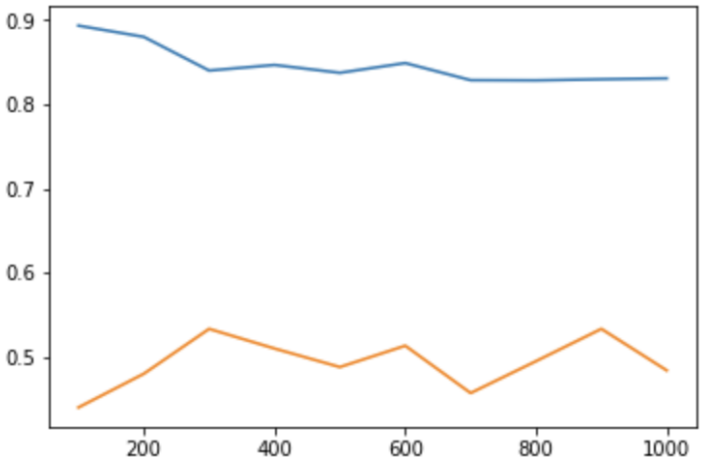
```
Best score: 0.516
Best parameters set:
        tfidf__use_idf: False
        vect__ngram_range: (1, 1)
```

Question 9

Use the pandas *sample* method to create a total of 10 samples of the eng1000 data, ranging from 100 to 1000. For each sample create a model and report train and test results. You should use the best parameters determined in the previous answer.

```
import matplotlib.pyplot as plt

def modelSteps(df, step):

    size = len(df)
    start = int(size / step)
    inc = start
    trainaccuracy=[]
    testaccuracy=[]


    for s in range(start, size + 1, inc):
        print (s)
        df_sample = df.sample(s, random_state=0)
        X = df_sample['text']
        y = df_sample['emotion']
        X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
        pipe = Pipeline(
            [
                ("vect", CountVectorizer()),
                ("tfidf", TfidfTransformer()),
                ("clf", LogisticRegression(solver='lbfgs', max_iter=1000,multi_class='auto',random_state=0))
            ]
            )

        pipe.fit(X_train,y_train)
        print("Train {:6.3f}".format( pipe.score(X_train, y_train)))
        print( "Test  {:6.3f}".format(pipe.score(X_test, y_test)))
        print("*****************")
        testaccuracy.append(pipe.score(X_test, y_test))
        trainaccuracy.append(pipe.score(X_train, y_train))

    x = list(range(start, size + 1, inc))
    plt.plot(x, trainaccuracy)
    plt.plot(x, testaccuracy)
    plt.show()


df = pd.read_csv("eng_1000.csv")
modelSteps(df,10)
```

```
100
Train  0.893
Test   0.440
*****************
200
Train  0.880
Test   0.480
*****************
300
Train  0.840
Test   0.533
*****************
400
Train  0.847
Test   0.510
*****************
500
Train  0.837
Test   0.488
*****************
600
Train  0.849
Test   0.513
*****************
700
Train  0.829
Test   0.457
*****************
800
Train  0.828
Test   0.495
*****************
900
Train  0.830
```

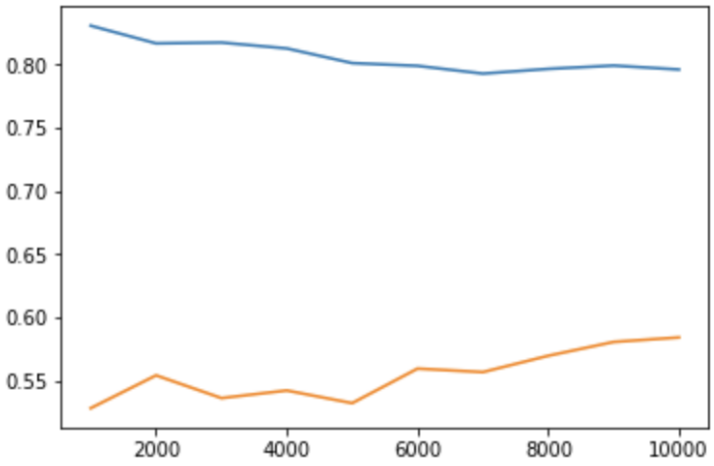

0.9
0.8
0.7
0.6
0.5
200
400
600
800
1000

Question 10

Do the same as the previous question, but use the eng10000 data, ranging from 1000 to 10000. For each of the above results, producing a line plot, with amount of data on the x axis, and accuracy score on the y axis. There should be a line for both test score and train score.

```
df = pd.read_csv("eng_10000.csv")
modelSteps(df,10)
```

```
1000
Train  0.831
Test   0.528
*****************
2000
Train  0.817
Test   0.554
*****************
3000
Train  0.817
Test   0.536
*****************
4000
Train  0.813
Test   0.542
*****************
5000
Train  0.801
Test   0.532
*****************
6000
Train  0.799
Test   0.559
*****************
7000
Train  0.793
Test   0.557
*****************
8000
Train  0.796
Test   0.570
*****************
9000
Train  0.799
```


0.80
0.75
0.70
0.65
0.60
0.55
2000
4000
6000
8000
10000

Types of Data

- Numerical Data
- Categorical Data
- Text data is different
  - Content of an email
  - A Headline
  - Text of political speeches

Can text be treated as structured data?

Making Language into Structured Data

- The solution: Dummy values for words
  - One feature for each word
  - Value is 1 if word occurs in text, 0 otherwise
  - Alternative values: number of word occurrences, or TFIDF score
  - For any text, value of most features will be 0

Exploit Big Data for Text

- Use Supervised ML for text processing
- Can we get labeled text data?
- Build Classifiers
  - Spam Detection
  - Sentiment Analysis
  - Topic Detection
  - ...
- What does this have to do with AI? Real understanding? Turing Test?

Treat text as a **Bag of Words**

Sentiment Analysis

- Is a text Positive or Negative?
- Used for Social Media Analysis
- Marketing
- Impact of new product

Movie Reviews as Data

- Online movie reviews
- Texts paired with ratings
- IMDB reviews
  - Positive: Rating of 7-10
  - Negative: Rating of 1-4

Bag of Words Processing

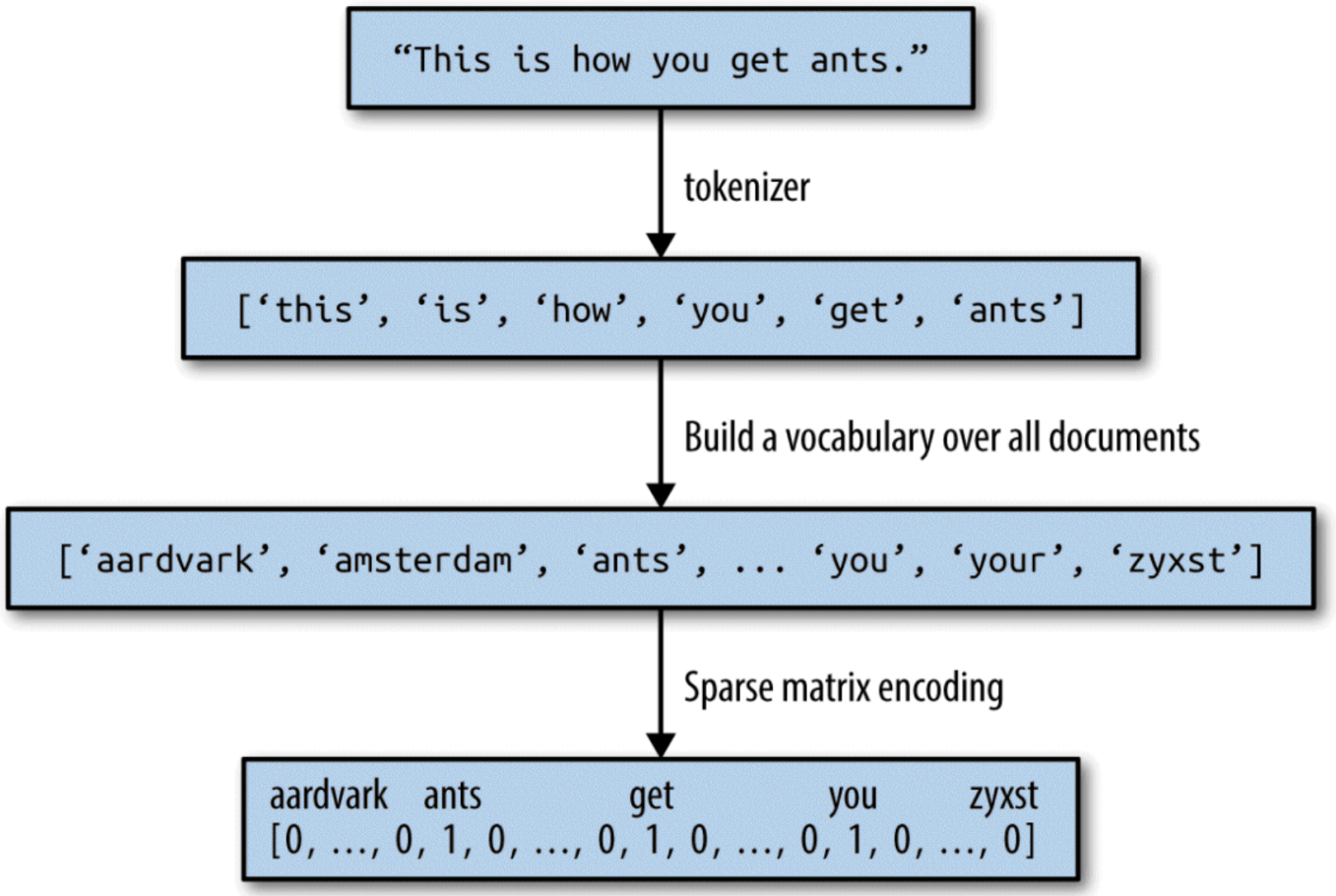


Figure 7-1. Bag-of-words processing

Bag of Words Processing

```
bards_words =["The fool doth think he is wise,",
              "but the wise man knows himself to be a fool"]
```

Bag of Words Processing

```
from sklearn.feature_extraction.text import CountVectorizer
vect = CountVectorizer()
vect.fit(bards_words)
```

Bag of Words Processing

```
Vocabulary size: 13
Vocabulary content:
 {'the': 9, 'himself': 5, 'wise': 12, 'he': 4, 'doth': 2,
'to': 11, 'knows': 7,
  'man': 8, 'fool': 3, 'is': 6, 'be': 0, 'think': 10, 'but':
1}
```

Bag of Words Processing

```
bag_of_words = vect.transform(bards_words)
print("bag_of_words: {}".format(repr(bag_of_words)))
```

Out[10]:

```
bag_of_words: <2x13 sparse matrix of type '<class
'numpy.int64'>'
```

Bag of Words Processing

```
    In[11]:
print("Dense representation of bag_of_words:\n{}".format(
    bag_of_words.toarray()))
    Out[11]:
Dense representation of bag_of_words:
[[0 0 1 1 1 0 1 0 0 1 1 0 1]
 [1 1 0 1 0 1 0 1 1 1 0 1 1]]
```

Stopwords

- Words that are “too frequent to be informative”
- Built-in Stopwords List: above, elsewhere, into, well, fifteen, . . .
- Could also discard words that appear too frequently

TF-IDF

- Words that are frequent in a document tell a lot about that document
- Words that appear in lots of documents are less interesting
- TF-IDF
  - *increases* as term frequency increases
  - *decreases* as document frequency increases

Bag of Words with More Than One Word

- nGrams

```
['The fool doth think he is wise,',
 'but the wise man knows himself to be a fool']
```

Unigrams

```
['be', 'but', 'doth', 'fool', 'he', 'himself', 'is',
'knows', 'man', 'the',
 'think', 'to', 'wise']
```

Bigrams

```
Vocabulary size: 14
Vocabulary:
['be fool', 'but the', 'doth think', 'fool doth', 'he is',
'himself to',
 'is wise', 'knows himself', 'man knows', 'the fool', 'the
wise',
 'think he', 'to be', 'wise man']
```

Ngrams

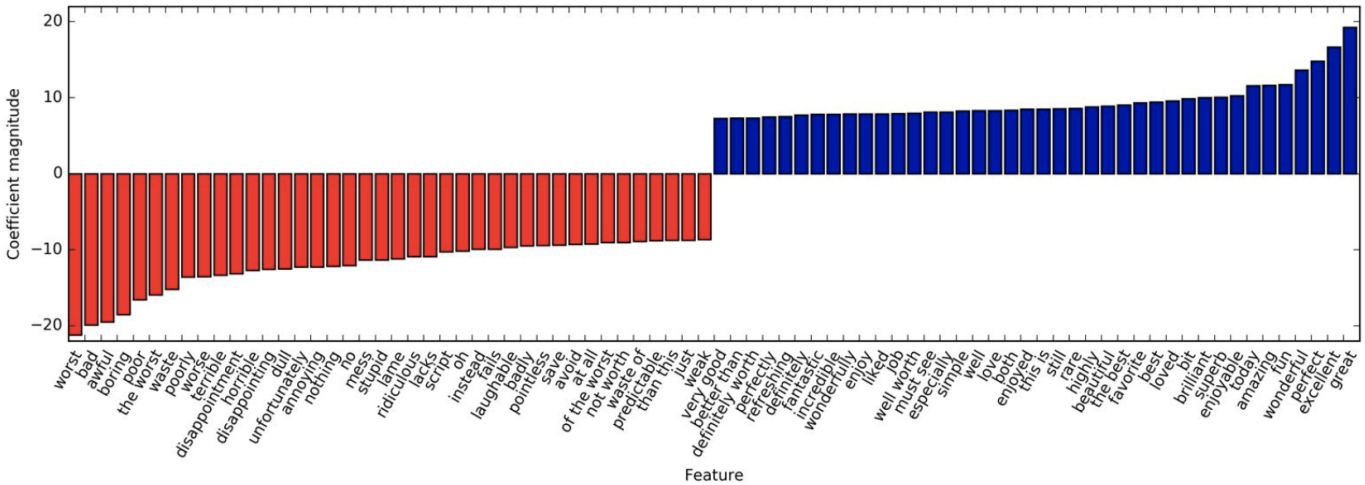


Figure 7-4. Most important features when using unigrams, bigrams, and trigrams with tf-idf rescaling

Copenhagen
Business School
HANDELSHØJSKOLEN

Language Modeling and Ngrams

Assign probability to a sequence of words

What is p(He went to the store)?

Language Modeling and Ngrams

He went to the store

- 1-grams (unigrams): He, went, to, the, store (5)
- 2-grams (bigrams): He went, went to, to the, the store (4)
- 3-grams (trigrams): He went to, went to the, to the store (3)
- 4-grams: He went to the, went to the store (2)
- 5-grams: He went to the store (1)

Language Modeling and Ngrams

Bigram approximation:

p(He went to the store) =
p(went|He) * p(to|went) *p(the|to) * p(store|the)

Trigram approximation:

p(He went to the store) =
p(to|He went) *p(the|went to) * p(store|to the)

BERT

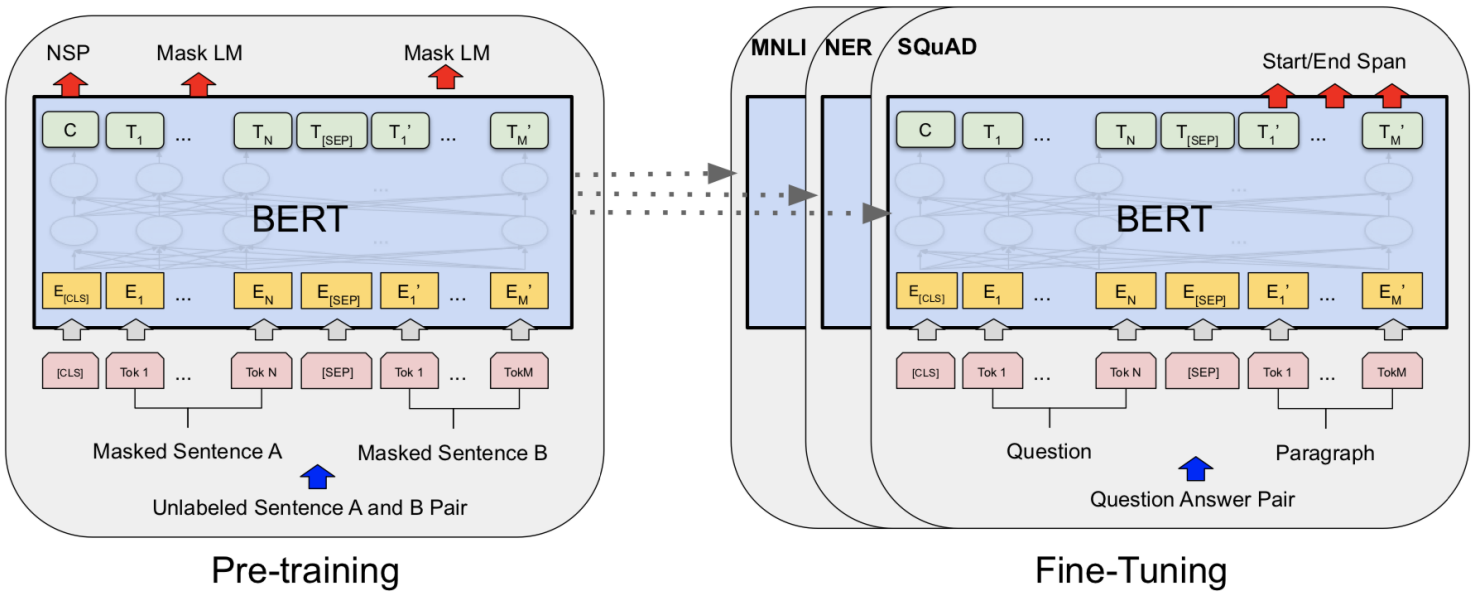


Figure 1: Overall pre-training and fine-tuning procedures for BERT. Apart from output layers, the same architectures are used in both pre-training and fine-tuning. The same pre-trained model parameters are used to initialize models for different down-stream tasks. During fine-tuning, all parameters are fine-tuned. `[CLS]` is a special symbol added in front of every input example, and `[SEP]` is a special separator token (e.g. separating questions/answers).

GPT-2

Talk to Transformer

See how a modern neural network completes your text. Type a custom snippet or try one of the examples. Learn more below.

Follow @AdamDanielKing for more neat neural networks.

Custom prompt

Copenhagen Business School is the largest business school in Scandinavia. It now has several courses that deal with AI and machine learning.

GENERATE ANOTHER

Completion

Copenhagen Business School is the largest business school in Scandinavia. It now has several courses that deal with AI and machine learning. They also offer the largest ever programme on research in AI to university students with more than 18,000 course credits (2016).

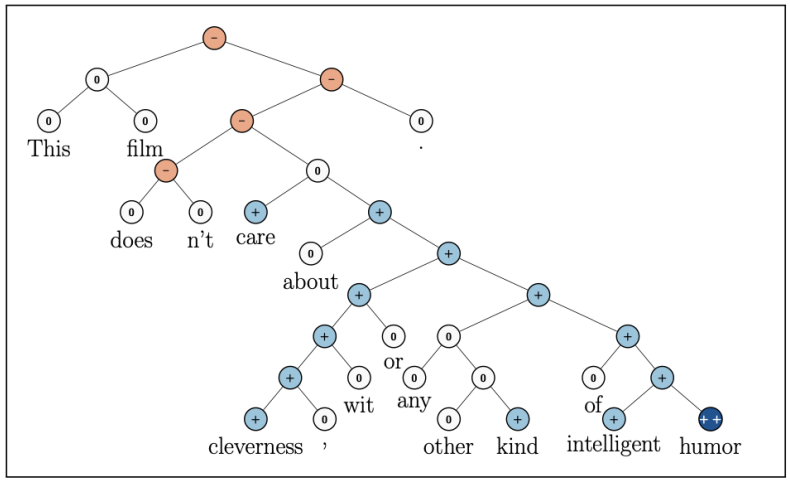
Sentiment Analysis

Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank

Richard Socher, Alex Perelygin, Jean Y. Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng and Christopher Potts
Stanford University, Stanford, CA 94305, USA
richard@socher.org,{aperelyg,jcchuang,ang}@cs.stanford.edu
{jeaneis,manning,cgpotts}@stanford.edu

Abstract

Semantic word spaces have been very useful but cannot express the meaning of longer phrases in a principled way. Further progress towards understanding compositionality in tasks such as sentiment detection requires richer supervised training and evaluation resources and more powerful models of composition. To remedy this, we introduce a Sentiment Treebank. It includes fine grained



Inference

MultiNLI

The Multi-Genre NLI Corpus

Adina Williams
Nikita Nangia
Sam Bowman
NYU

Introduction

The Multi-Genre Natural Language Inference (MultiNLI) corpus is a crowd-sourced collection of 433k sentence pairs annotated with textual entailment information. The corpus is modeled on the SNLI corpus, but differs in that covers a range of genres of spoken and written text, and supports a distinctive cross-genre generalization evaluation. The corpus served as the basis for the shared task of the RepEval 2017 Workshop at EMNLP in Copenhagen.

Inference

Examples

Premise	Label	Hypothesis
Fiction		
The Old One always comforted Ca'daan, except today.	*neutral*	Ca'daan knew the Old One very well.
Letters		
Your gift is appreciated by each and every student who will benefit from your generosity.	*neutral*	Hundreds of students will benefit from your generosity.
Telephone Speech		
yes now you know if if everybody like in August when everybody's on vacation or something we can dress a little more casual or	*contradiction*	August is a black out month for vacations in the company.
9/11 Report		
At the other end of Pennsylvania Avenue, people began to line up for a White House tour.	*entailment*	People formed a line at the end of Pennsylvania Avenue.

Question Answering

SQuAD2.0

The Stanford Question Answering Dataset

What is SQuAD?

Stanford **Qu**estion **A**nswering **D**ataset (SQuAD) is a reading comprehension dataset, consisting of questions posed by crowdworkers on a set of Wikipedia articles, where the answer to every question is a segment of text, or *span*, from the corresponding reading passage, or the question might be unanswerable.

Leaderboard

SQuAD2.0 tests the ability of a system to not only answer reading comprehension questions, but also abstain when presented with a question that cannot be answered based on the provided paragraph.

Rank	Model	EM	F1
	Human Performance *Stanford University*	86.831	89.452

Question Answering

SQuAD2.0 combines the 100,000 questions in SQuAD1.1 with over 50,000 unanswerable questions written adversarially by crowdworkers to look similar to answerable ones. To do well on SQuAD2.0, systems must not only answer questions when possible, but also determine when no answer is supported by the paragraph and abstain from answering.

Explore SQuAD2.0 and model predictions

SQuAD2.0 paper (Rajpurkar & Jia et al. '18)

SQuAD 1.1, the previous version of the SQuAD dataset, contains 100,000+ question-answer pairs on 500+ articles.

Explore SQuAD1.1 and model predictions

SQuAD1.0 paper (Rajpurkar et al. '16)

Getting Started

Question Answering

Leaderboard

SQuAD2.0 tests the ability of a system to not only answer reading comprehension questions, but also abstain when presented with a question that cannot be answered based on the provided paragraph.

Rank	Model	EM	F1
	Human Performance *Stanford University* (Rajpurkar & Jia et al. '18)	86.831	89.452
1 Jun 04, 2021	IE-Net (ensemble) *RICOH_SRCB_DML*	**90.939**	**93.214**
2 Feb 21, 2021	FPNet (ensemble) *Ant Service Intelligence Team*	90.871	93.183
3 May 16, 2021	IE-NetV2 (ensemble) *RICOH_SRCB_DML*	90.860	93.100
4 Apr 06, 2020	SA-Net on Albert (ensemble) *QIANXIN*	90.724	93.011
5 May 05, 2020	SA-Net-V2 (ensemble) *QIANXIN*	90.679	92.948
5 Apr 05, 2020	Retro-Reader (ensemble) *Shanghai Jiao Tong University* http://arxiv.org/abs/2001.09694	90.578	92.978

Devlin et al. 2019

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

Jacob Devlin **Ming-Wei Chang** **Kenton Lee** **Kristina Toutanova**
Google AI Language
{jacobdevlin,mingweichang,kentonl,kristout}@google.com

Abstract

We introduce a new language representation model called **BERT**, which stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers. Unlike recent language representation models (Peters et al., 2018a; Radford et al., 2018), BERT is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. As a result, the pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as question answering and language inference, without substantial task-specific architecture modifications.

There are two existing strategies for applying pre-trained language representations to downstream tasks: *feature-based* and *fine-tuning*. The feature-based approach, such as ELMo (Peters et al., 2018a), uses task-specific architectures that include the pre-trained representations as additional features. The fine-tuning approach, such as the Generative Pre-trained Transformer (OpenAI GPT) (Radford et al., 2018), introduces minimal task-specific parameters, and is trained on the downstream tasks by simply fine-tuning *all* pre-trained parameters. The two approaches share the same objective function during pre-training, where they use unidirectional language models to learn general language representations.

Main Points

- **B**idirectional **E**ncoder **R**epresentations from **T**ransformers.
- Pre-trained language model
- Can be fine-tuned for
  - **Single Sentence Classification:** Sentiment Analysis, Emotion Classification
  - **Sentence Pair Classification:** Question-Answering, Inference

Input-Output Representation

- Can represent a single sentence or pair of sentences
- “Sentence” can be any span of text
- Add [CLS] symbol to beginning of input sequence
- [SEP] symbol at end of first sentence
- If input is a pair of sentences, they are separated by [SEP]
- Use WordPiece embeddings

Pre-Training

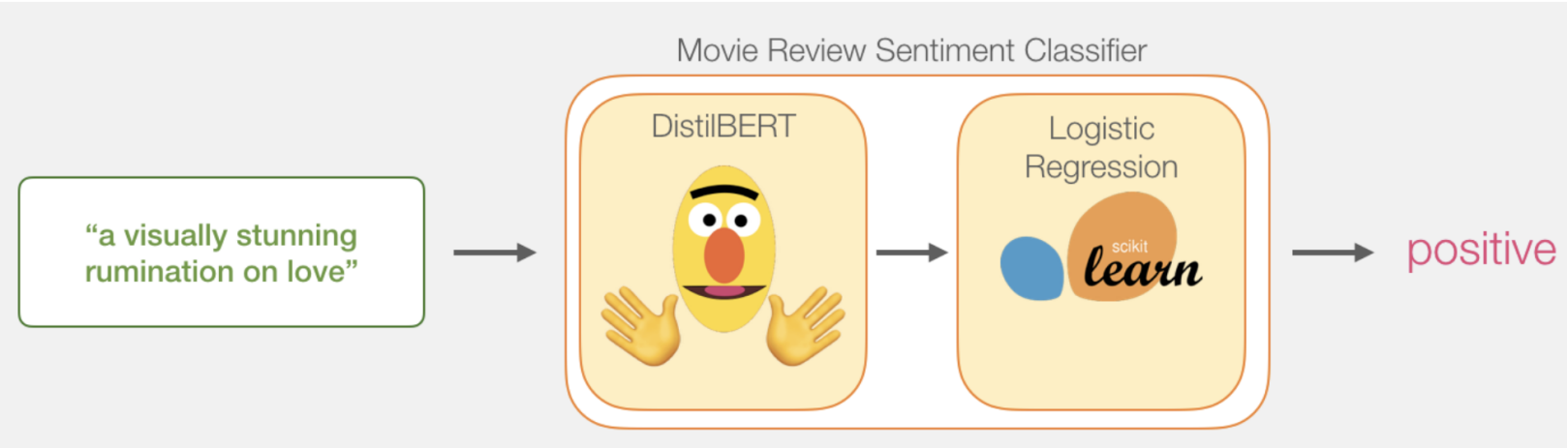
- Input is pair of sentences
- Two Tasks:
- Masked LM
  - Predict *masked* tokens, based on surrounding tokens
- Next Sentence Prediction
  - Predict whether second sentence naturally follows first one

Fine-Tuning

- Different for different tasks
- Sentence-pair tasks (Question Answering, Inference, . . .)
  - Input is sentence pairs, with same representation as training
- Single-sentence tasks: (Sentiment analysis, emotion classification, . . .)
  - Input is single sentence, ending with [SEP] symbol

System	MNLI-(m/mm)	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE	**Average**
	392k	363k	108k	67k	8.5k	5.7k	3.5k	2.5k	-
Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.8	90.4	36.0	73.3	84.9	56.8	71.0
OpenAI GPT	82.1/81.4	70.3	87.4	91.3	45.4	80.0	82.3	56.0	75.1
$\text{BERT}_{\text{BASE}}$	84.6/83.4	71.2	90.5	93.5	52.1	85.8	88.9	66.4	79.6
$\text{BERT}_{\text{LARGE}}$	**86.7/85.9**	**72.1**	**92.7**	**94.9**	**60.5**	**86.5**	**89.3**	**70.1**	**82.1**

Table 1: GLUE Test results, scored by the evaluation server (https://gluebenchmark.com/leaderboard). The number below each task denotes the number of training examples. The "Average" column is slightly different than the official GLUE score, since we exclude the problematic WNLI set.[8] BERT and OpenAI GPT are single-model, single task. F1 scores are reported for QQP and MRPC, Spearman correlations are reported for STS-B, and accuracy scores are reported for the other tasks. We exclude entries that use BERT as one of their components.


Movie Review Sentiment Classifier
"a visually stunning rumination on love"
DistilBERT
Logistic Regression
scikit
learn
positive

Two Models

- DistilBERT
- Logistic Regression

DistilBERT

- Smaller version of BERT
- Almost matches BERT performance
- Produces sentence embedding – vector of size 768

Logistic Regression

- Classifies each sentence as Positive or Negative
- Features are 768 real numbers – sentence embedding

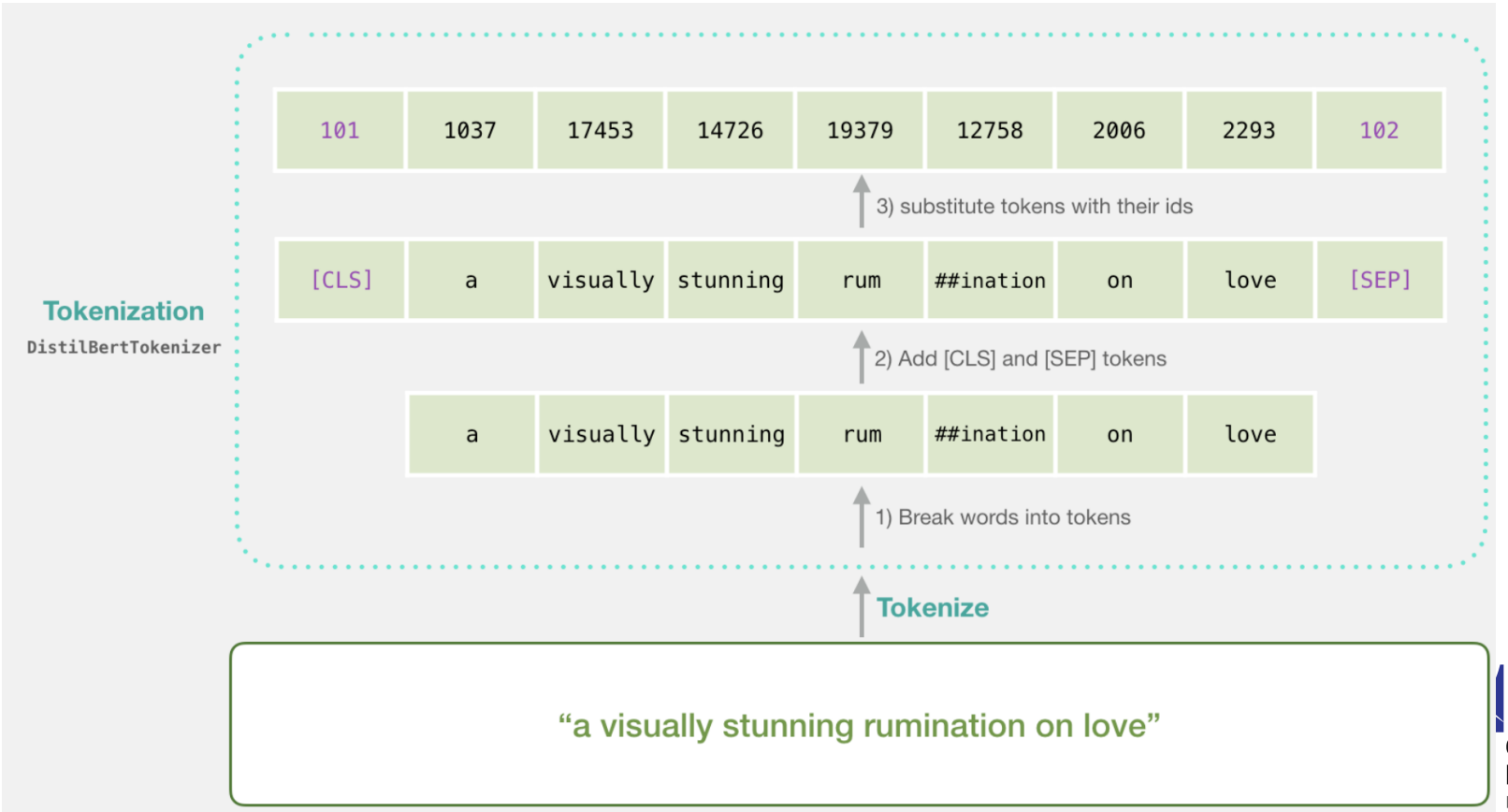
Dataset

- SST2
- Movie Review texts, classified as Positive or Negative

Model 1: Data Preparation

- Tokenization
- Padding
- Masking

```
tokenized = batch_1[0].apply((lambda x: tokenizer.encode(x, add_special_tokens=True)))
```



Padding

After tokenization, `tokenized` is a list of sentences -- each sentences is represented as a list of tokens. We want BERT to process our examples all at once (as one batch). It's just faster that way. For that reason, we need to pad all lists to the same size, so we can represent the input as one 2-d array, rather than a list of lists (of different lengths).

```
max_len = 0
for i in tokenized.values:
    if len(i) > max_len:
        max_len = len(i)

padded = np.array([i + [0]*(max_len-len(i)) for i in tokenized.values])
```

Our dataset is now in the `padded` variable, we can view its dimensions below:

```
np.array(padded).shape
```

```
(2000, 59)
```

Masking

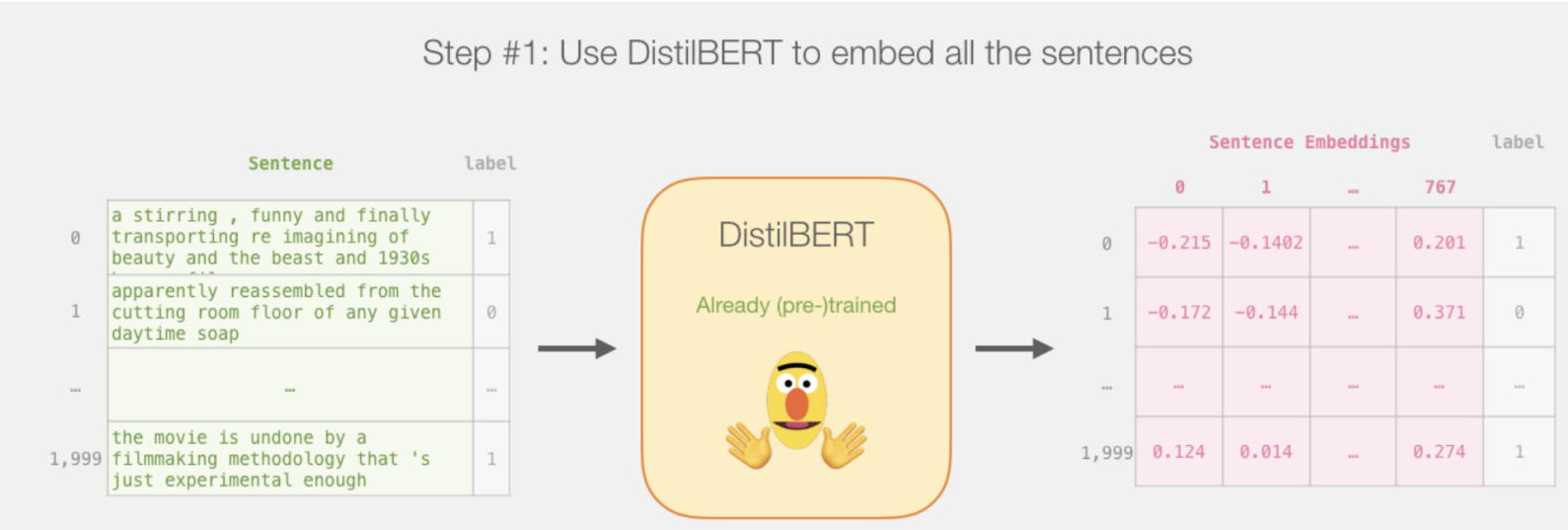
If we directly send `padded` to BERT, that would slightly confuse it. We need to create another variable to tell it to ignore (mask) the padding we've added when it's processing its input. That's what attention_mask is:

```
attention_mask = np.where(padded != 0, 1, 0)
attention_mask.shape
```

```
(2000, 59)
```

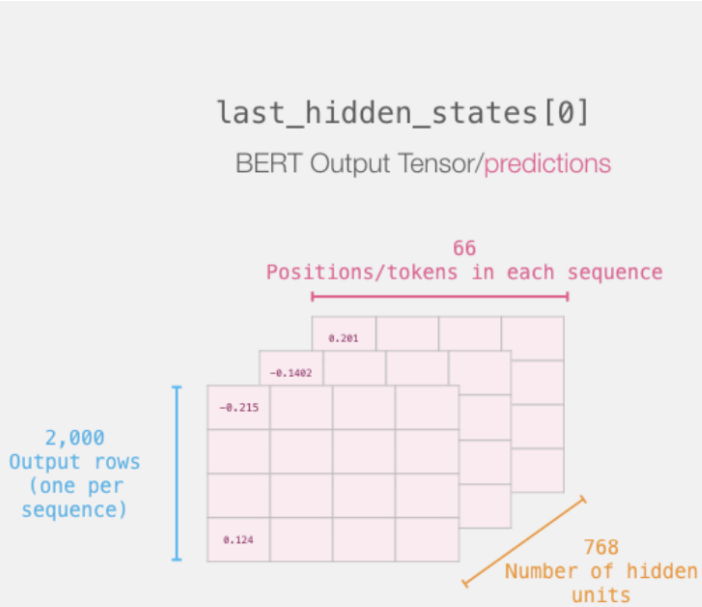
Model 2: Sentence Embeddings

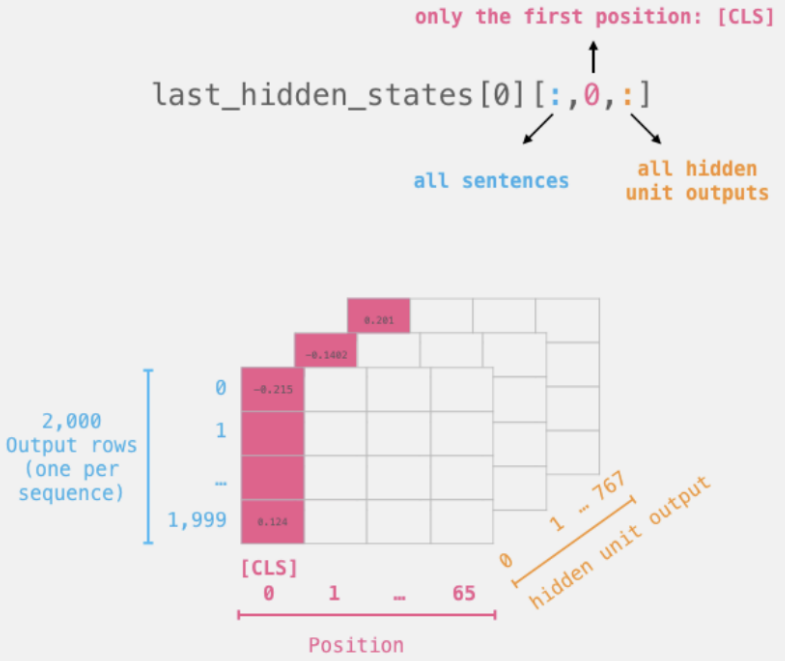
- All sentences are input to BERT
- Output of interest corresponds to first token ([CLS])


Step #1: Use DistilBERT to embed all the sentences
Sentence
label
0
a stirring , funny and finally transporting re imagining of beauty and the beast and 1930s
1
1
apparently reassembled from the cutting room floor of any given daytime soap
0
…
…
…
1,999
the movie is undone by a filmmaking methodology that 's just experimental enough
1
DistilBERT
Already (pre-)trained
Sentence Embeddings
label
0
1
…
767
0
-0.215
-0.1402
…
0.201
1
1
-0.172
-0.144
…
0.371
0
…
…
…
…
…
…
1,999
0.124
0.014
…
0.274
1

Model Produces Sentence Embeddings

```
input_ids = torch.tensor(padded)
attention_mask = torch.tensor(attention_mask)

with torch.no_grad():
    last_hidden_states = model(input_ids, attention_mask=attention_mask)
```





Copenhagen
Business School
HANDELSHØJSKOLEN

Assign Features and Labels

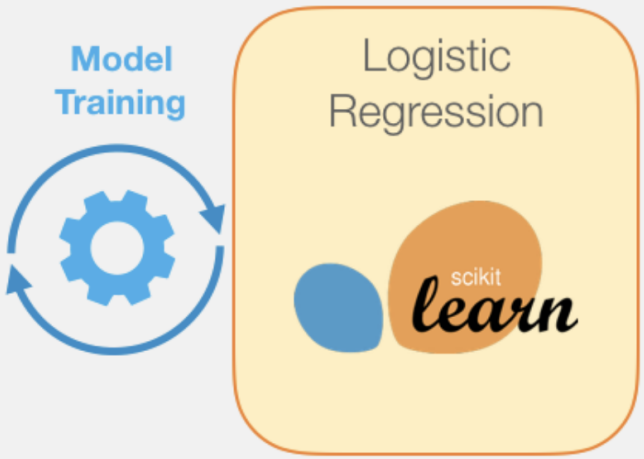
```
features = last_hidden_states[0][:,0,:].numpy()
```

```
labels = batch_1[1]
```

Logistic Regression Model

Step #3: Train the logistic regression model using the training set

	Sentence Embeddings				label
	0	1	…	767	
0	-0.215	-0.1402	…	0.201	1
1					
2					
3					
4					
5					
6					
…					
1,499					



Logistic Regression Model

```
lr_clf = LogisticRegression()
lr_clf.fit(train_features, train_labels)
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

Scoring the Model

```
lr_clf.score(test_features, test_labels)
```

```
0.824
```

```
from sklearn.dummy import DummyClassifier
clf = DummyClassifier()

scores = cross_val_score(clf, train_features, train_labels)
print("Dummy classifier score: %0.3f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
```

```
Dummy classifier score: 0.527 (+/- 0.05)
```

Best Scores

For reference, the highest accuracy score for this dataset is currently **96.8**. DistilBERT can be trained to improve its score on this task – a process called **fine-tuning** which updates BERT's weights to make it achieve a better performance in this sentence classification task (which we can call the downstream task). The fine-tuned DistilBERT turns out to achieve an accuracy score of **90.7**. The full size BERT model achieves **94.9**.