CS 696 Applied Large Language Models
Spring Semester, 2025
Doc 13 Fine Tuning 1
Feb 20, 2025

Copyright ©, All rights reserved. 2025 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/openpub/) license defines the copyright on this document.

References

UCI Machine Learning Repository
https://archive.ics.uci.edu

Imbalance Learn
https://imbalanced-learn.org/stable/user_guide.html

Finetuning Large Language Models,
Sebastian Raschka, PhD
APR 22, 2023
https://magazine.sebastianraschka.com/p/finetuning-large-language-models

Building a Large Language Model (from Scratch), Sebastian Raschka

Loading Saved Weights & Memory

```
model = GPTModel(BASE_CONFIG) # gpt2-xl (1558M)
device = torch.device("cuda")
model.to(device)
```

Maximum GPU memory allocated: 6.4 GB

```
model = GPTModel(BASE_CONFIG)
model.to(device)

model.load_state_dict(
    torch.load("model.pth", map_location=device, weights_only=True)
)
model.to(device)
model.eval();
```

Maximum GPU memory allocated: 12.8 GB

Copy Weights One By One to GPU

```
model = GPTModel(BASE_CONFIG).to(device)

state_dict = torch.load("model.pth", map_location="cpu", weights_only=True)

print_memory_usage()

# Sequentially copy weights to the model's parameters
with torch.no_grad():
    for name, param in model.named_parameters():
        if name in state_dict:
            param.copy_(state_dict[name].to(device))
        else:
            print(f"Warning: {name} not found in state_dict.")
```

Maximum GPU memory allocated: **6.7 GB**

PyTorch’s “meta" device

Tensor containing only metadata

Load a model's structure and parameters without loading the actual data into memory

- Inspecting model architecture
- Code Optimization
  - Optimize the computational graph before running actual computations

Using meta is CPU Memory limited

```
with torch.device("meta"):
    model = GPTModel(BASE_CONFIG)

model = model.to_empty(device=device)

state_dict = torch.load("model.pth", map_location=device, weights_only=True)

# Sequentially copy weights to the model's parameters
with torch.no_grad():
    for name, param in model.named_parameters():
        if name in state_dict:
            param.copy_(state_dict[name])
        else:
            print(f"Warning: {name} not found in state_dict.")
```

Maximum GPU memory allocated: 12.8 GB
-> Maximum CPU memory allocated: 1.3 GB

torch.load & mmap=True Option

torch.load

Loads the entire file in memory

mmap=true

Creates a virtual memory map of file

Data is loaded on demand

Depending on available memory, some data is loaded

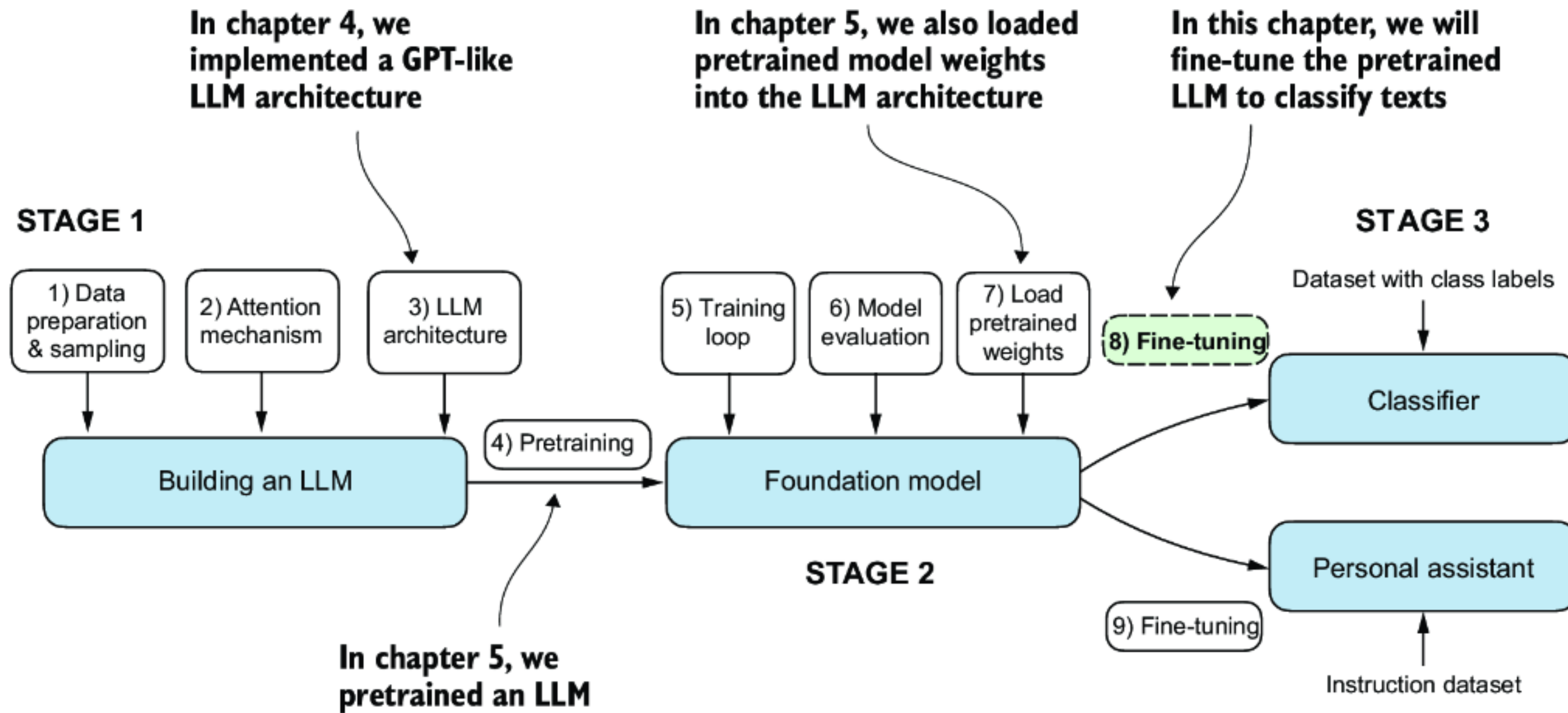
If low on CPU, Memory

```
with torch.device("meta"):
    model = GPTModel(BASE_CONFIG)

model.load_state_dict(
    torch.load("model.pth", map_location=device, weights_only=True, mmap=True),
    assign=True
)
```

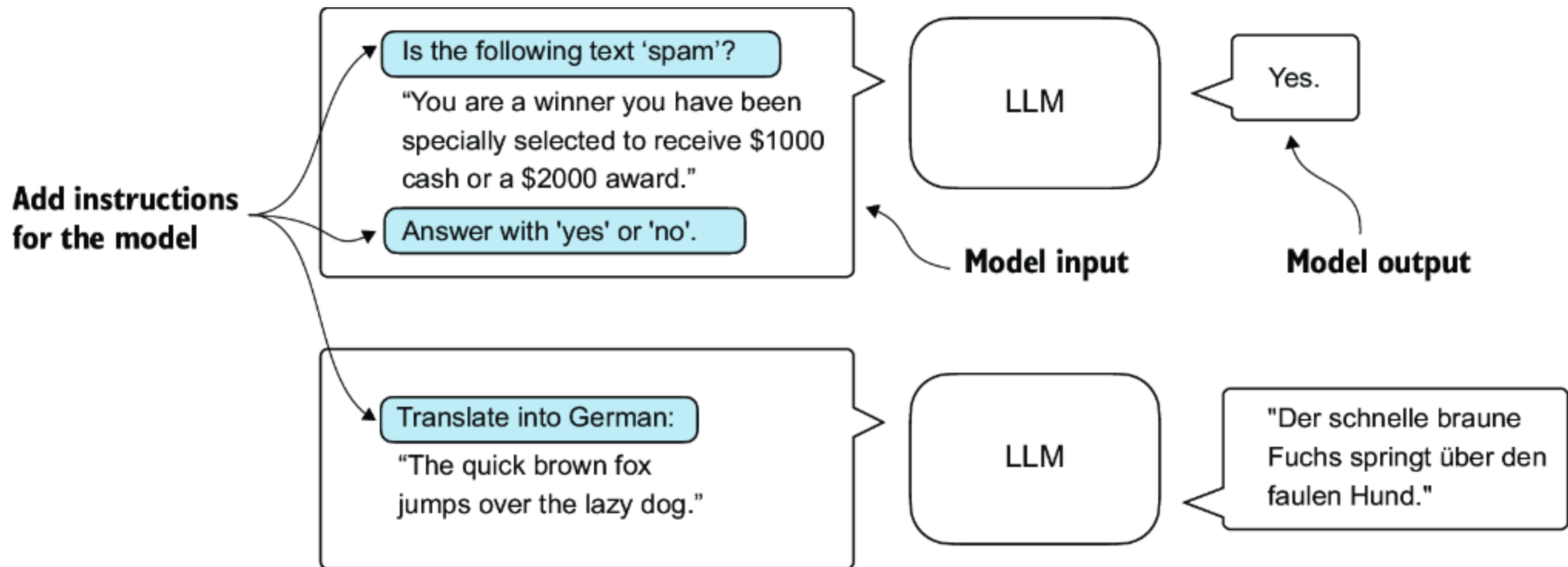
Maximum GPU memory allocated: 6.4 GB

-> Maximum CPU memory allocated: 5.9 GB

Fine Tuning

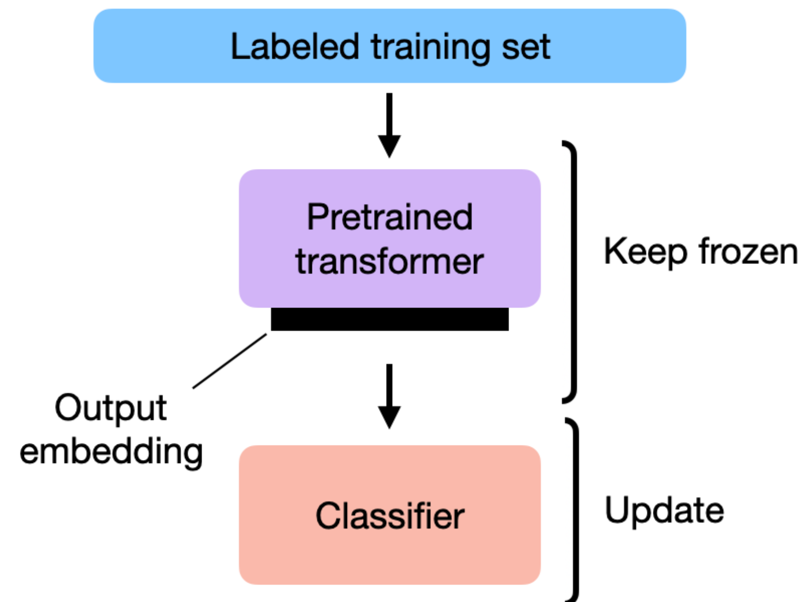


Instruction vs Classification Fine-Tuning

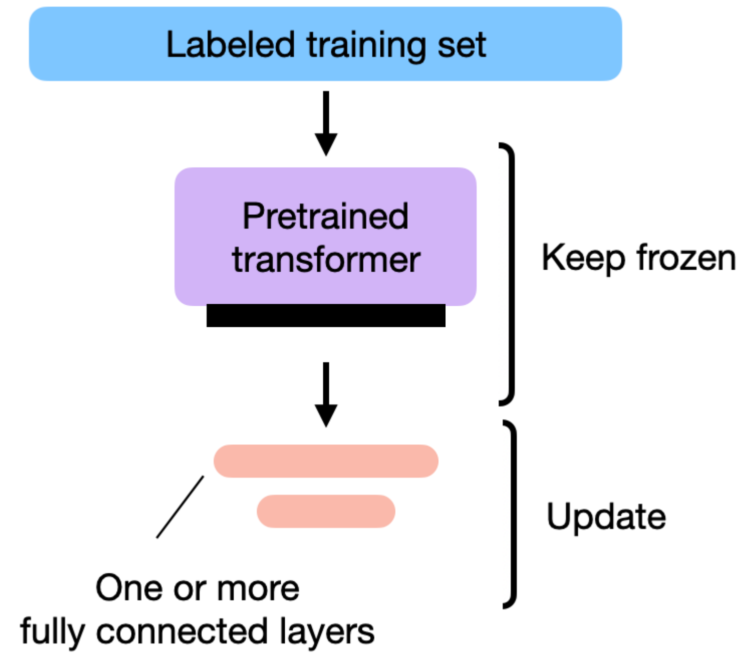


Finetuning Methods

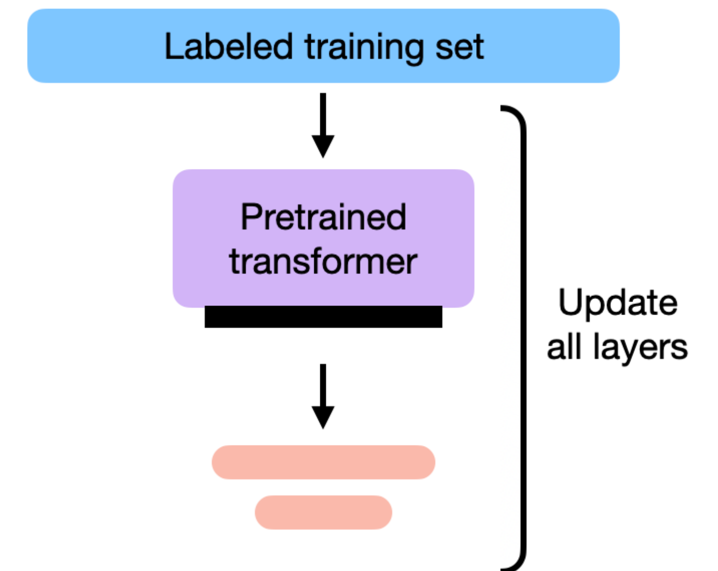
1) Feature-based approach



2) Finetuning I



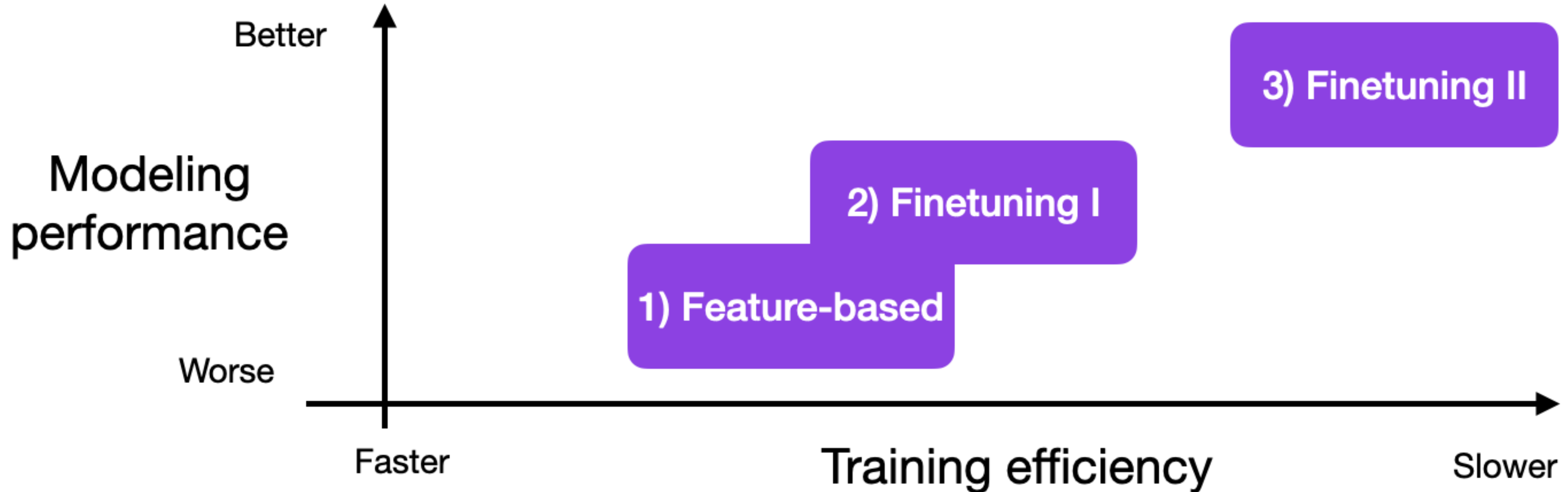
3) Finetuning II



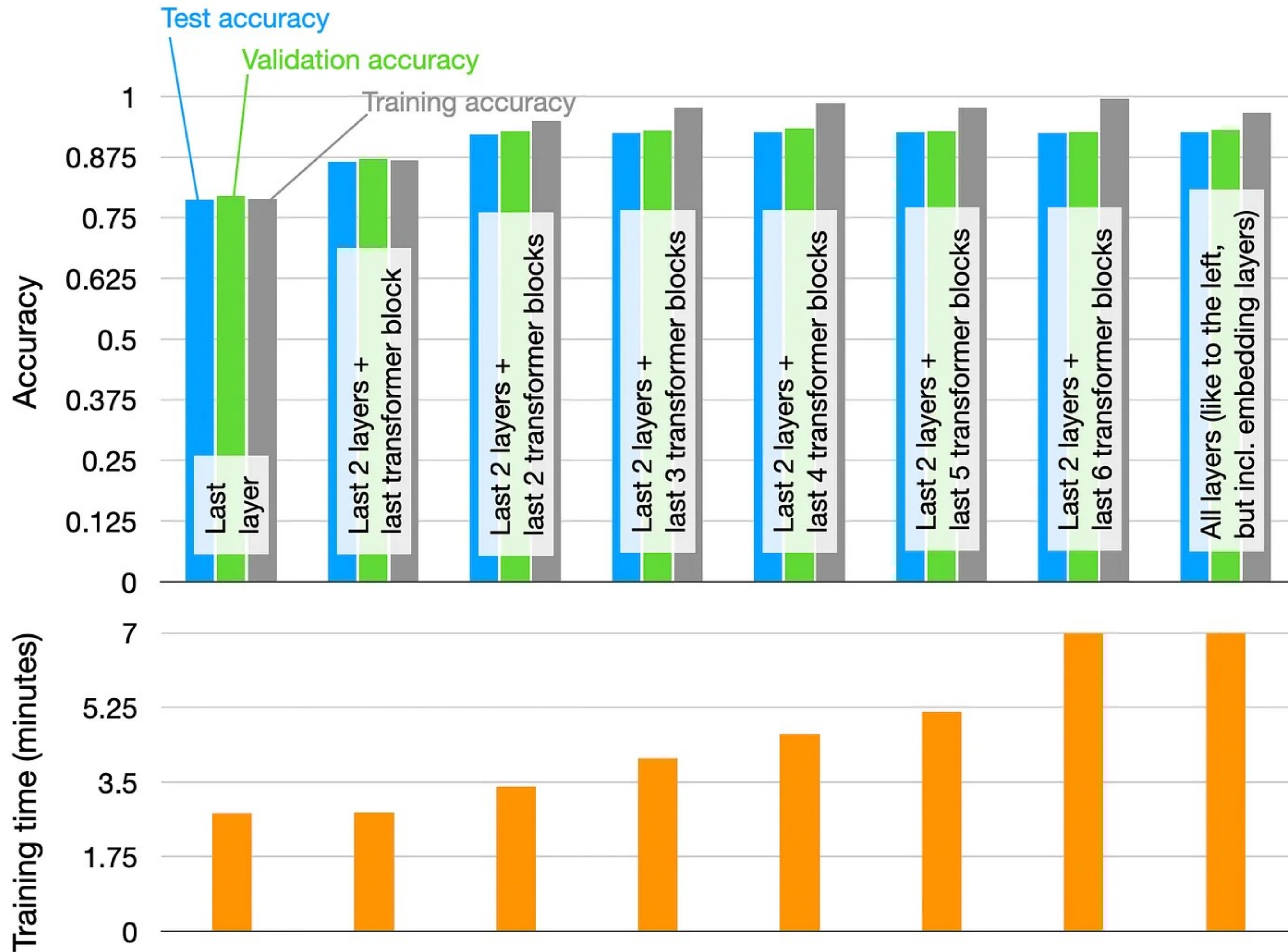
Movie Review Classifier REsults

https://github.com/rasbt/LLM-finetuning-scripts/tree/main/conventional/distilbert-movie-review

1) Feature-based approach with logistic regression: 83% test accuracy
2) Finetuning I, updating the last 2 layers: 87% accuracy
3) Finetuning II, updating all layers: 92% accuracy.

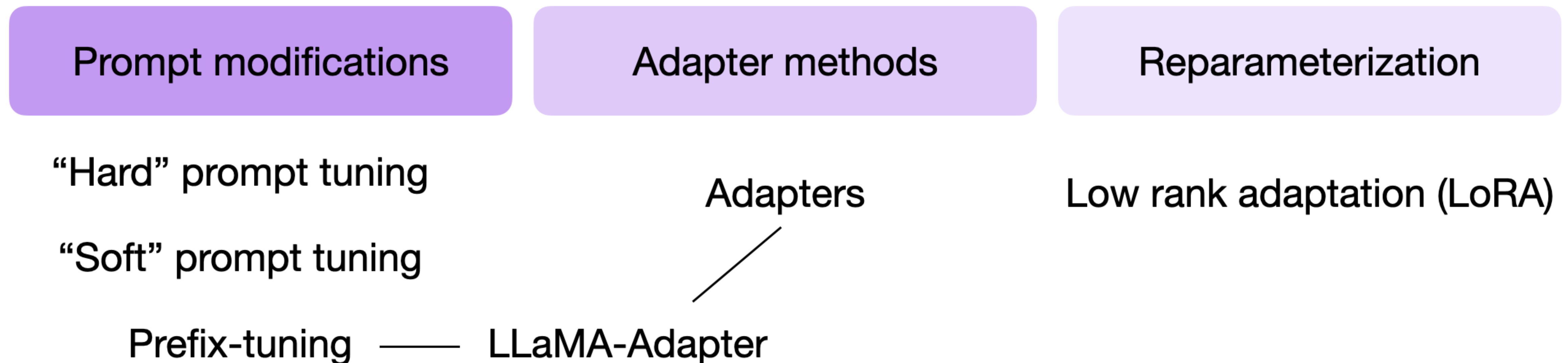


DistilBERT model finetuned on the 20k training examples



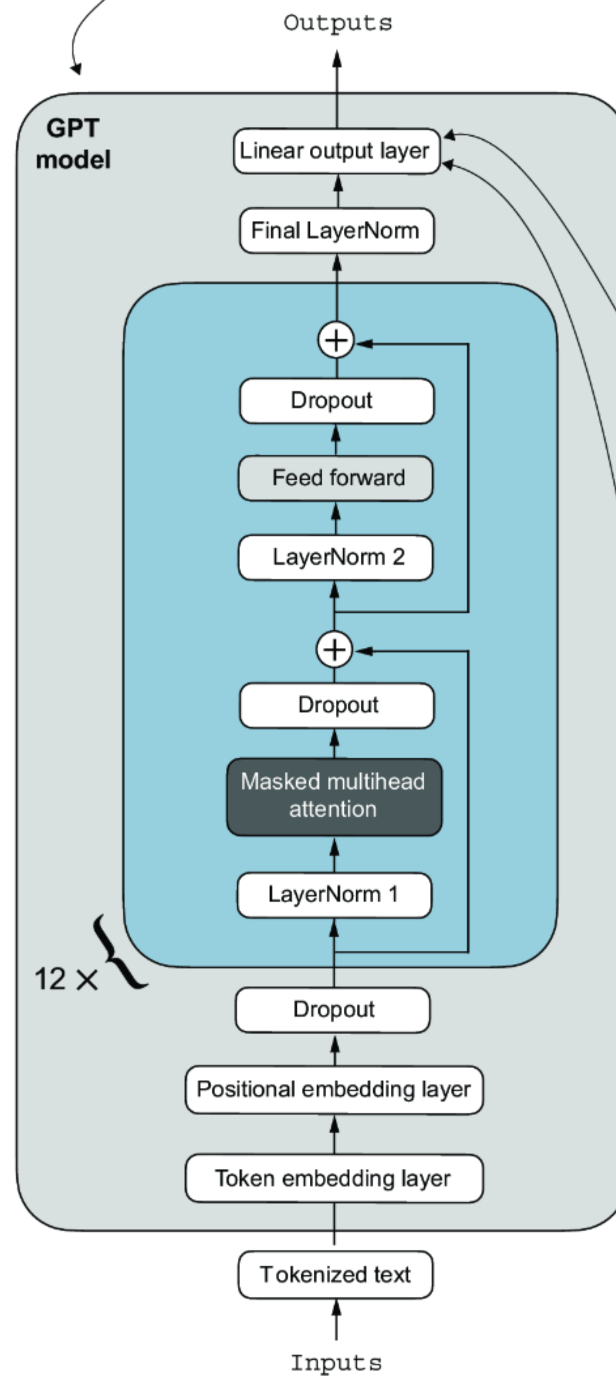
Parameter-Efficient Finetuning (PEFT)

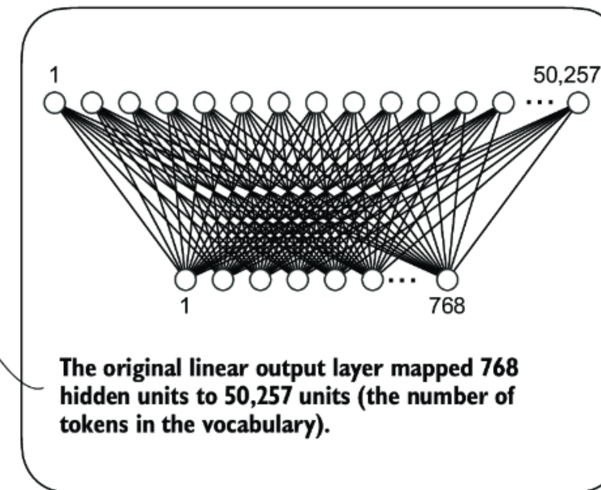
Reduced computational costs (requires fewer GPUs and GPU time);
Faster training times (finishes training faster);
Lower hardware requirements (works with smaller GPUs & less smemory);
Better modeling performance (reduces overfitting);
Less storage (majority of weights can be shared across different tasks).

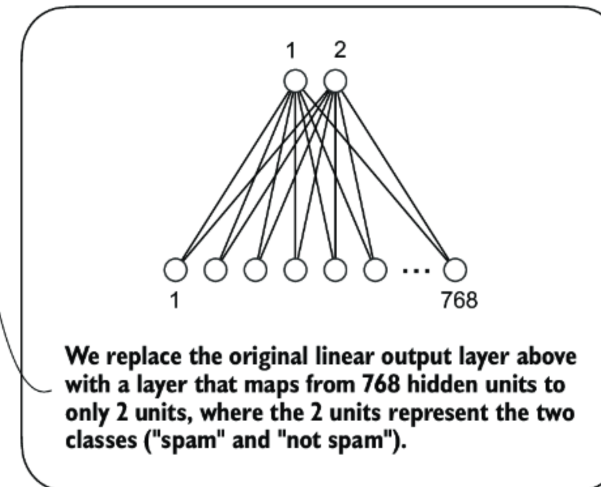


Classification Fine-Tuning

The GPT model we implemented in chapter 5 and loaded in the previous section







UCI Machine Learning Repository

https://archive.ics.uci.edu

Welcome to the UC Irvine Machine Learning Repository

We currently maintain 674 datasets as a service to the machine learning community. Here, you can donate and find datasets used by millions of people all around the world!

VIEW DATASETS | CONTRIBUTE A DATASET

Popular Datasets

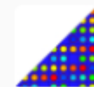
Iris
A small classic dataset from Fisher, 1936. One of the earliest known datasets used for...
Classification | 150 Instances | 4 Features

Heart Disease
4 databases: Cleveland, Hungary, Switzerland, and the VA Long Beach
Classification | 303 Instances | 13 Features

Wine Quality
Two datasets are included, related to red and white vinho verde wine samples, from th...
Classification, Regres... | 4.9K Instances | 12 Features

Adult
Predict whether annual income of an individual exceeds $50K/yr based on census dat...
Classification | 48.84K Instances | 14 Features

New Datasets



Lattice-physics (PWR fuel assembly neutronics simulation results)
This dataset encompasses lattice-physics parameters—the infinite multiplication fact...
Regression | 24K Instances | 39 Features

Gas sensor array low-concentration
This dataset contains 6 gas responses collected by a sensor array consisting of 10 m...
Classification, Regres... | 90 Instances

Twitter Geospatial Data
Seven days of geo-tagged Tweet data from the United States with exact GPS location...
Classification, Regres... | 14.26M Instances | 4 Features

CAN-MIRGU
A Comprehensive CAN Bus Attack Dataset from Moving Vehicles for Intrusion Detecti...
Classification | 48 Instances

SMS Spam Collection

Donated on 6/21/2012

The SMS Spam Collection is a public set of SMS labeled messages that have been collected for mobile phone spam research.

Dataset Characteristics
Multivariate, Text, Domain-Theory

Subject Area
Computer Science

Associated Tasks
Classification, Clustering

Feature Type
Real

 # Instances
5574

 # Features
-

Dataset Information

Additional Information

This corpus has been collected from free or free for research sources at the Internet:

-> A collection of 425 SMS spam messages was manually extracted from the Grumbletext Web site. This is a UK forum in which cell phone users make public claims about SMS spam messages, most of them without reporting the very spam message received. The identification of the text of spam messages in the claims is a very hard and time-consuming task, and it involved carefully scanning hundreds of web pages. The Grumbletext Web site is: http://www.grumbletext.co.uk/.
-> A subset of 3,375 SMS randomly chosen ham messages of the NUS SMS Corpus (NSC), which is a dataset of about 10,000 legitimate messages collected for research at the Department of Computer Science at the National University of Singapore. The messages largely originate from Singaporeans and mostly from students attending the University. These messages were collected from volunteers who were made aware that their contributions were going to be made publicly

SMS Spam Collection

Spreadsheet

	Label	Text
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...
...	...	...
5567	spam	This is the 2nd time we have tried 2 contact u...
5568	ham	Will ü b going to esplanade fr home?
5569	ham	Pity, * was in mood for that. So...any other s...
5570	ham	The guy did some bitching but I acted like i'd...
5571	ham	Rofl. Its true to its name

Label
ham 4825
spam 747

Imbalanced dataset

Handling Imbalanced Datasets

Over-sampling

Add data to the smaller sets

Under-sampling

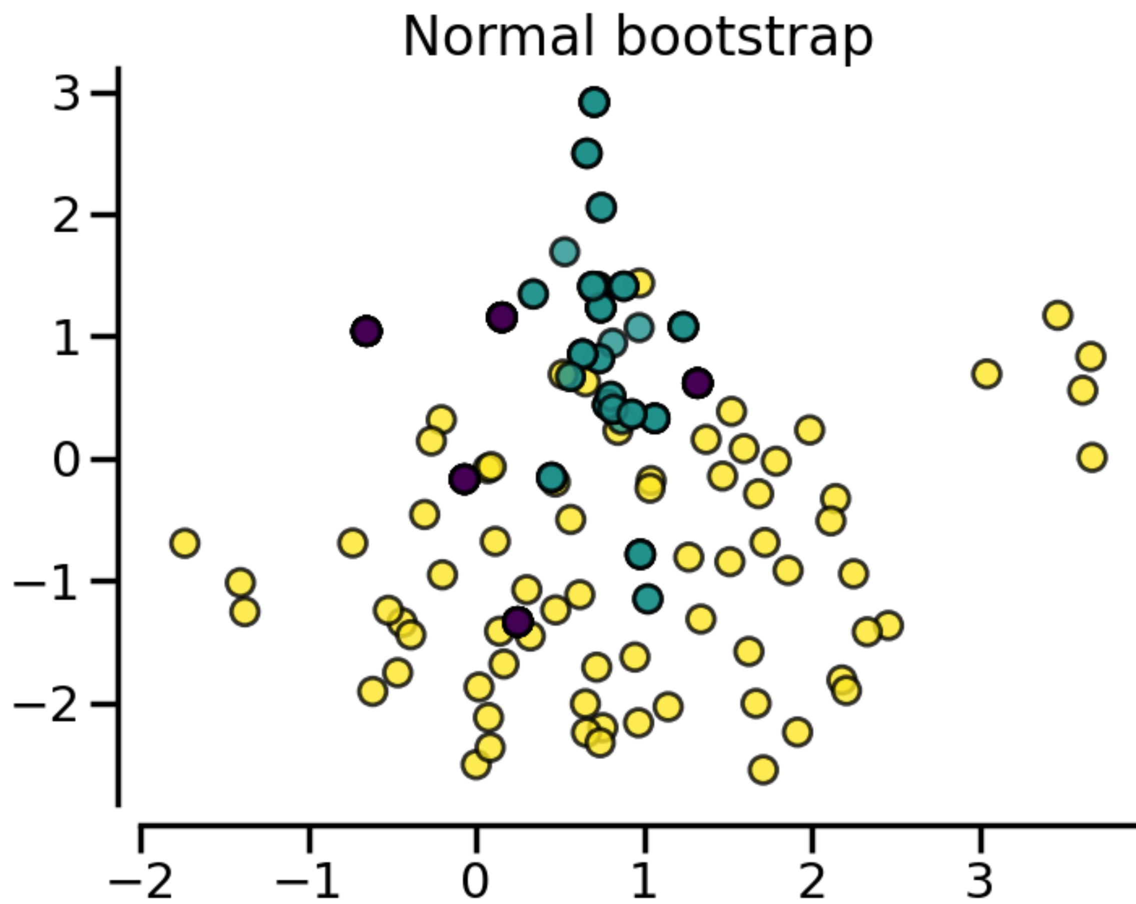
Remove data from the largest set

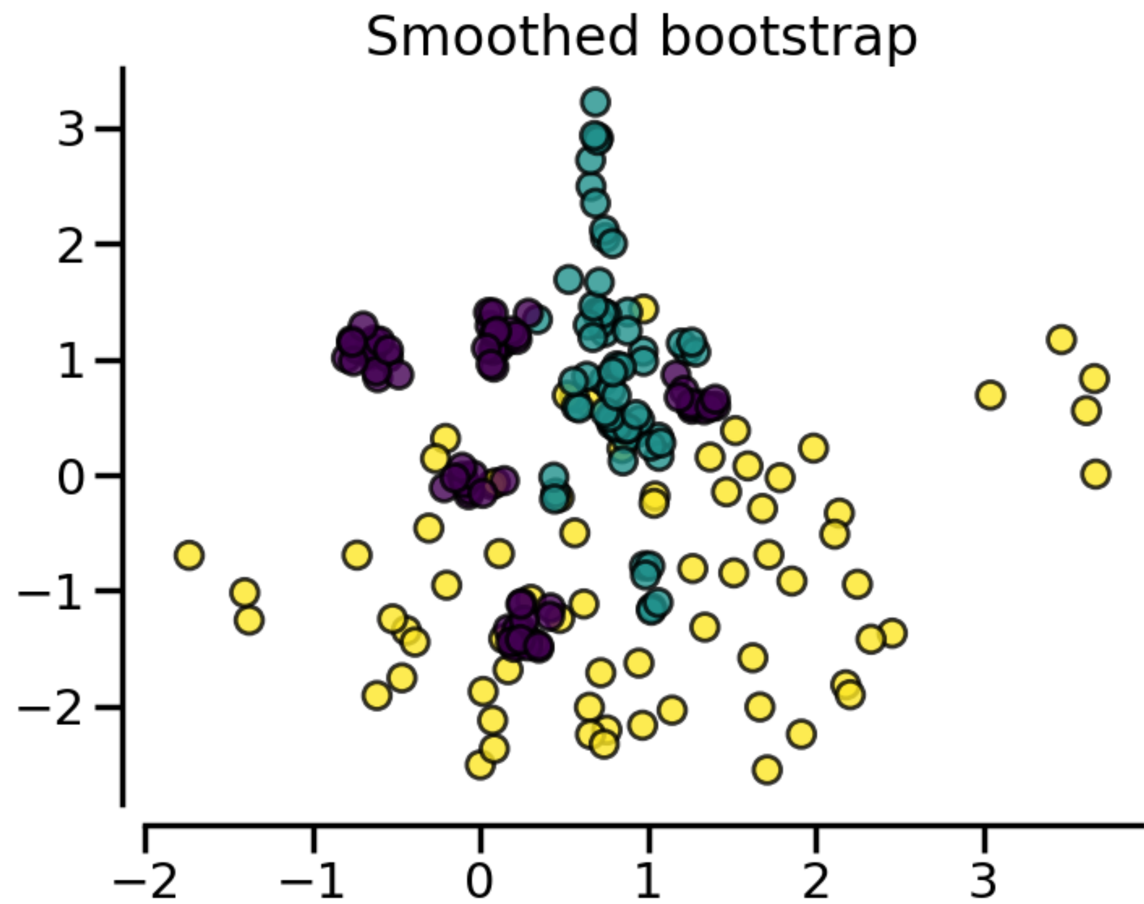
Over-sampling

Random Sampling with Replacement

Random Over-Sampling Examples (ROSE)

Resampling with RandomOverSampler

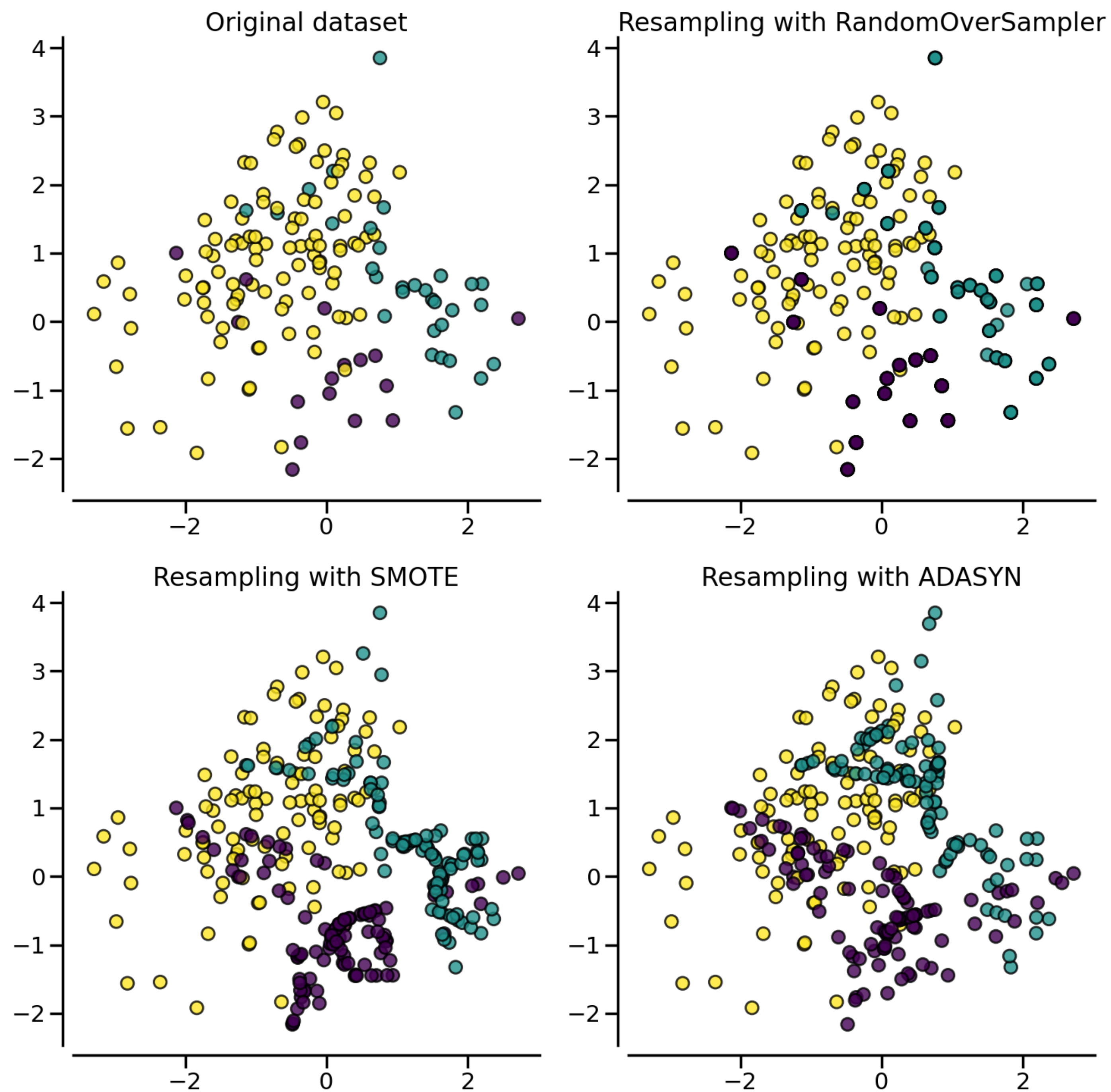




SMOTE & ADASYN

Synthetic Minority Oversampling Technique (SMOTE)

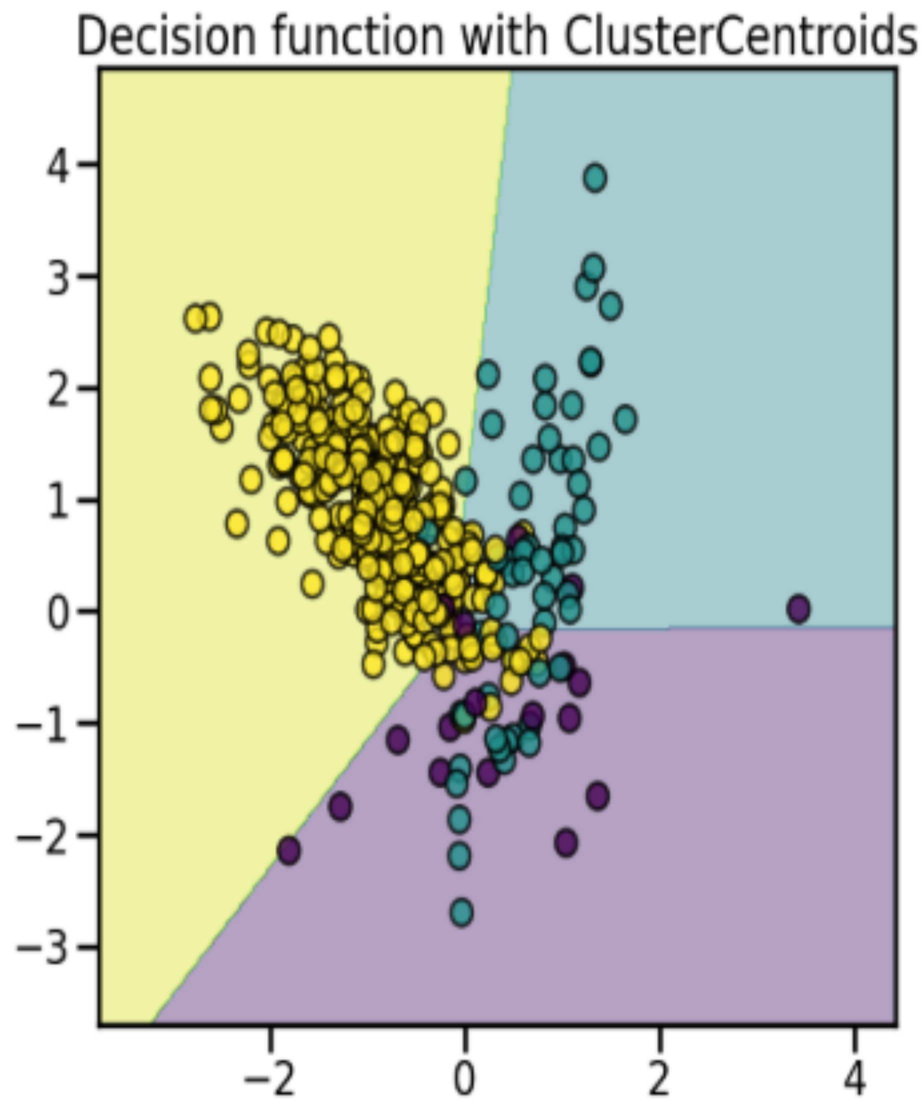
Adaptive Synthetic (ADASYN)

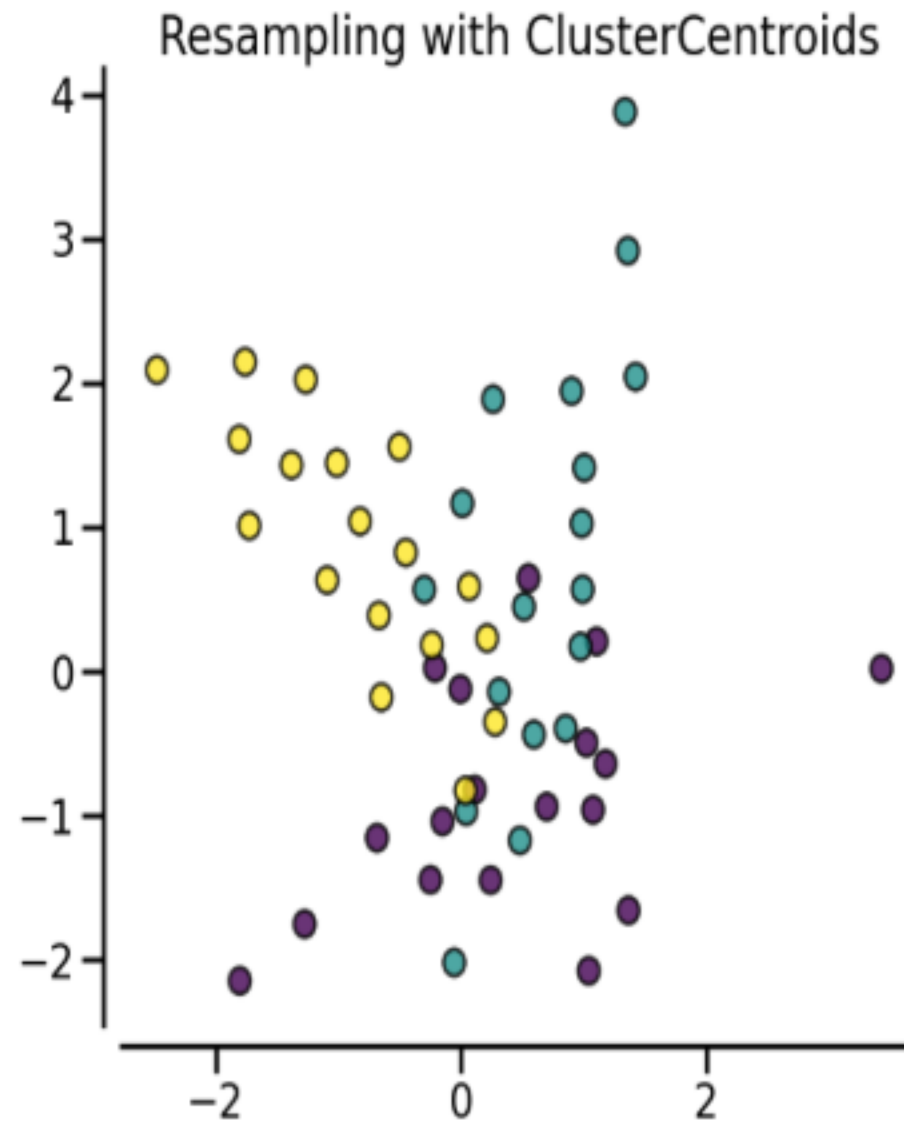


Under Sampling

Prototype generation methods

Create a smaller set by creating new data

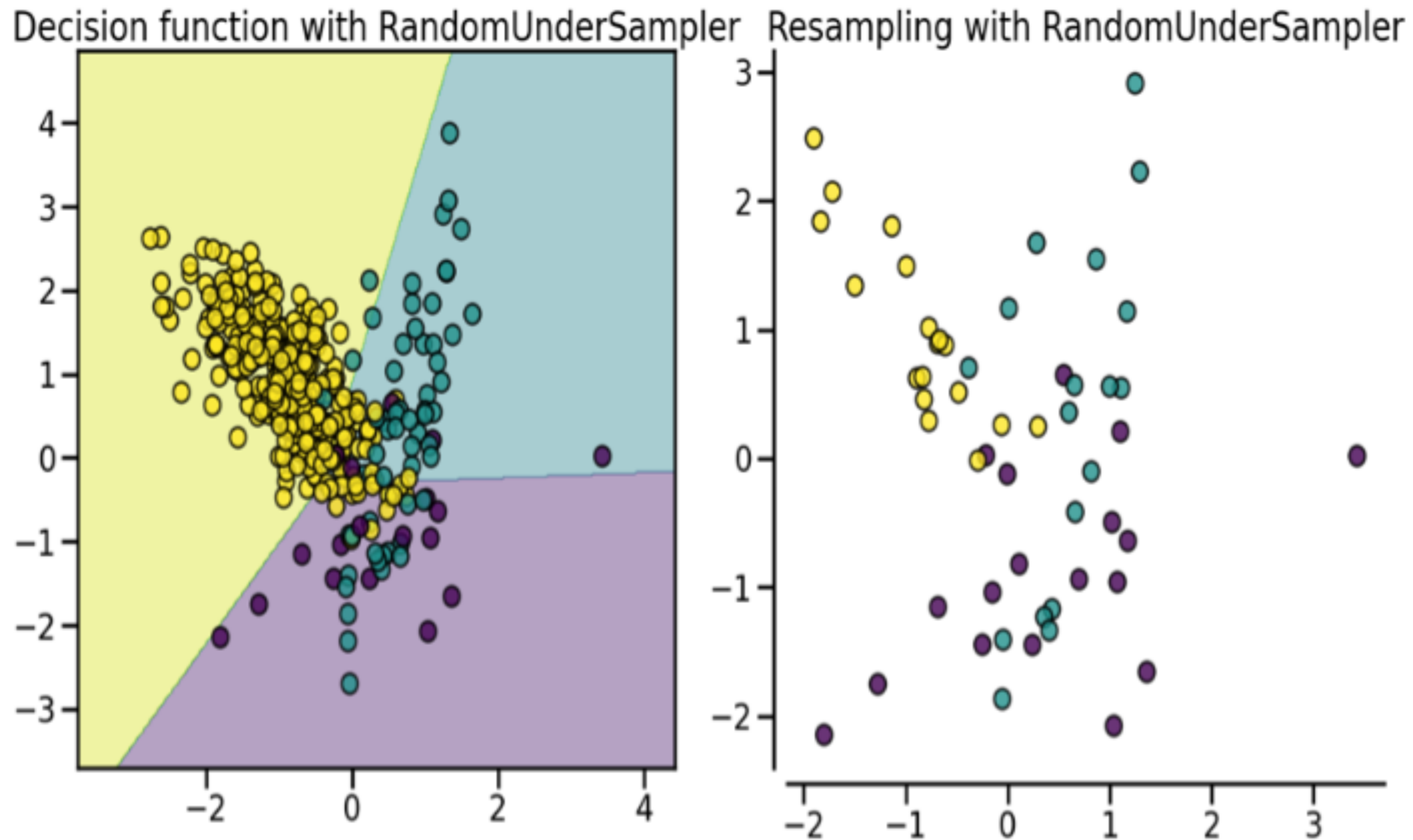




Under Sampling - Prototype selection methods

Controlled Undersampling

Random sampling



Under Sampling - Cleaning under-sampling

Clean” the feature space by removing either “noisy” observations or observations that are “too easy to classify”

Tomek’s links

Edited nearest neighbours

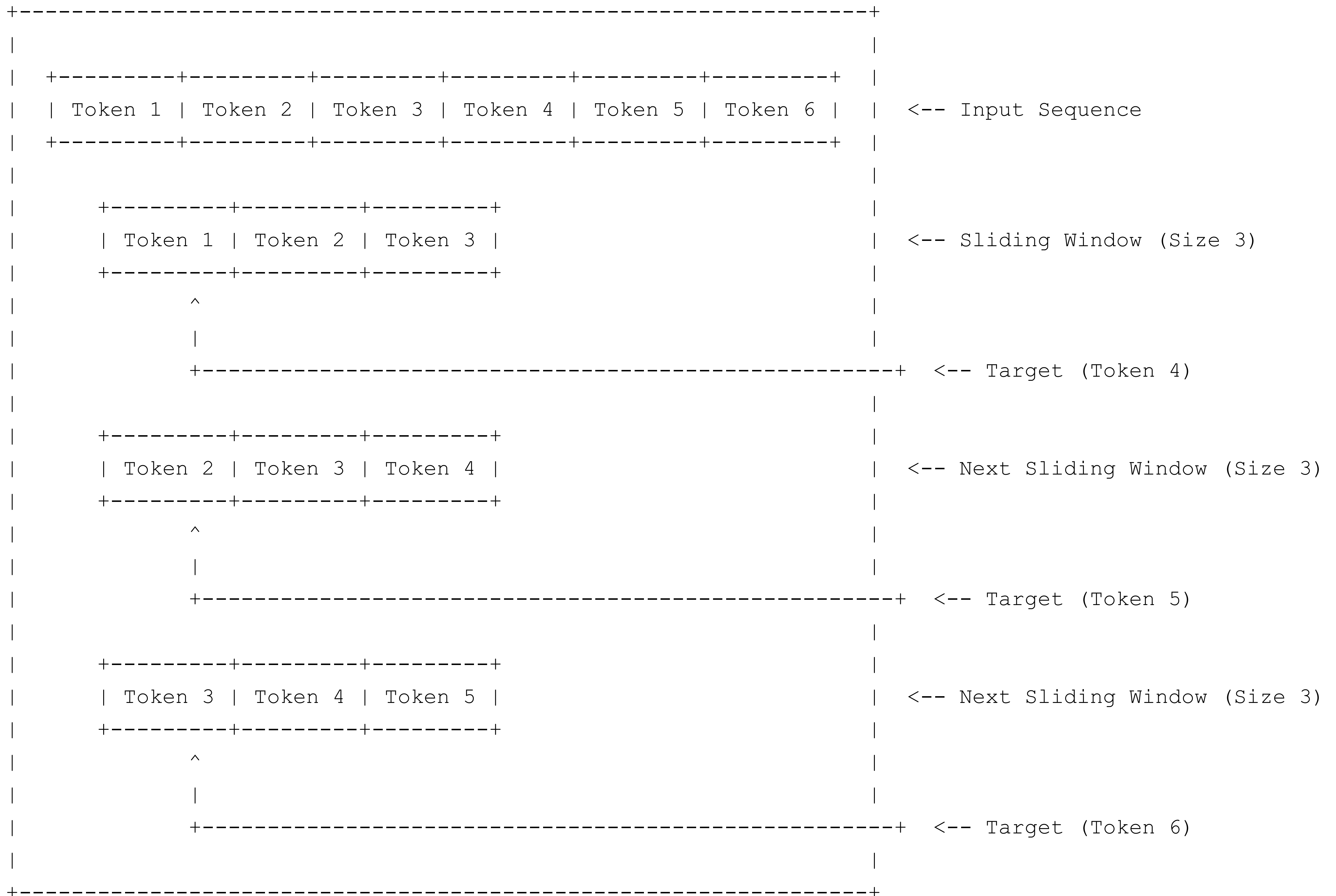
Repeated Edited Nearest Neighbours

Message Length

Oh k...i'm watching here:)

U don't know how stubborn I am. I didn't even want to go to the hospital. I kept telling Mark I'm not a weak sucker. Hospitals are for weak suckers.

Yup


Token 1
Token 2
Token 3
Token 4
Token 5
Token 6
<-- Input Sequence
Token 1
Token 2
Token 3
<-- Sliding Window (Size 3)
<-- Target (Token 4)
Token 2
Token 3
Token 4
<-- Next Sliding Window (Size 3)
<-- Target (Token 5)
Token 3
Token 4
Token 5
<-- Next Sliding Window (Size 3)
<-- Target (Token 6)

Message Length

Oh k...i'm watching here:)

U don't know how stubborn I am. I didn't even want to go to the hospital. I kept telling Mark I'm not a weak sucker. Hospitals are for weak suckers.

Yup

Either

- Trim all to the length of the shortest message
- Pad all to the length of the longest

Create balanced Dataset

Divide into

- Training 70%
- Validation 10%
- Testing 20%

Create

- Dataset
- Data Loader

Load the model

Dataset

```
import torch
from torch.utils.data import Dataset

class SpamDataset(Dataset):
    def __init__(self, csv_file, tokenizer, max_length=None,
                 pad_token_id=50256):
        self.data = pd.read_csv(csv_file)
        self.encoded_texts = [
            tokenizer.encode(text) for text in self.data["Text"]
        ]

        if max_length is None:
            self.max_length = self._longest_encoded_length()
        else:
            self.max_length = max_length
            self.encoded_texts = [
                encoded_text[:self.max_length]
                for encoded_text in self.encoded_texts
            ]

        self.encoded_texts = [
            encoded_text + [pad_token_id] * (self.max_length - len(encoded_text))
            for encoded_text in self.encoded_texts
        ]
```

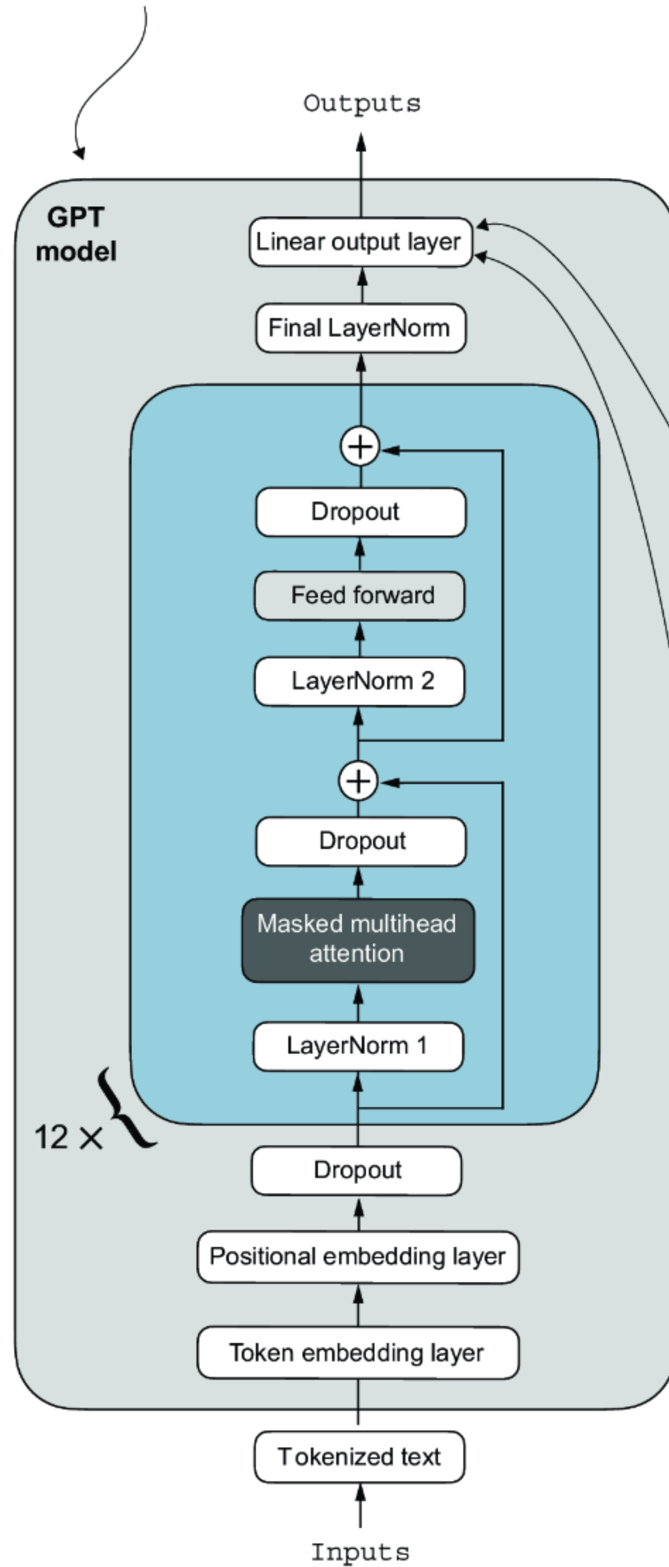
Dataset

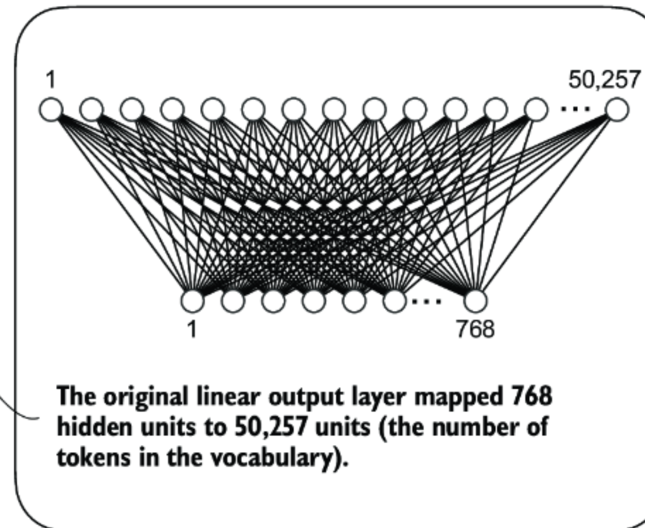
```
def __getitem__(self, index):
    encoded = self.encoded_texts[index]
    label = self.data.iloc[index]["Label"]
    return (
        torch.tensor(encoded, dtype=torch.long),
        torch.tensor(label, dtype=torch.long)
    )

def __len__(self):
    return len(self.data)

def _longest_encoded_length(self):
    max_length = 0
    for encoded_text in self.encoded_texts:
        encoded_length = len(encoded_text)
        if encoded_length > max_length:
            max_length = encoded_length
    return max_length
```

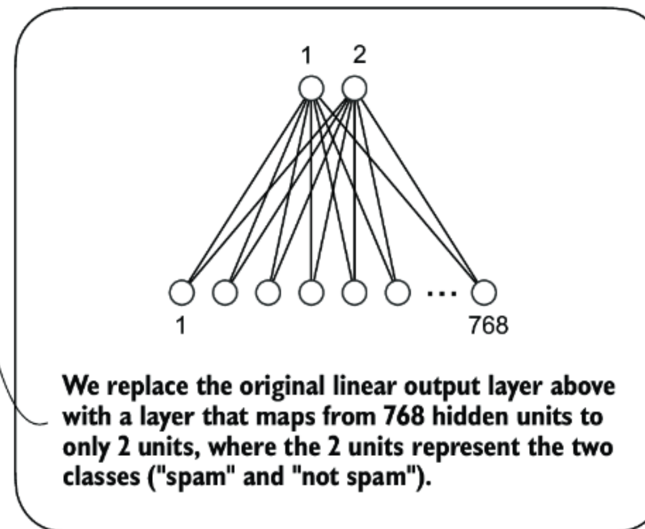
Freeze layers

Replace output layer

The GPT model we implemented in chapter 5 and loaded in the previous section







print(model)

```
GPTModel(
  (tok_emb): Embedding(50257, 768)
  (pos_emb): Embedding(1024, 768)
  (drop_emb): Dropout(p=0.0, inplace=False)
  (trf_blocks): Sequential(
    (0): TransformerBlock(
      (att): MultiHeadAttention(
        (W_query): Linear(in_features=768, out_features=768, bias=True)
        (W_key): Linear(in_features=768, out_features=768, bias=True)
        (W_value): Linear(in_features=768, out_features=768, bias=True)
        (out_proj): Linear(in_features=768, out_features=768, bias=True)
        (dropout): Dropout(p=0.0, inplace=False)
      )
      (ff): FeedForward(
        (layers): Sequential(
          (0): Linear(in_features=768, out_features=3072, bias=True)
          (1): GELU()
          (2): Linear(in_features=3072, out_features=768, bias=True)
        )
      )
      (norm1): LayerNorm()
      (norm2): LayerNorm()
      (drop_resid): Dropout(p=0.0, inplace=False)
    )
```

print(model)

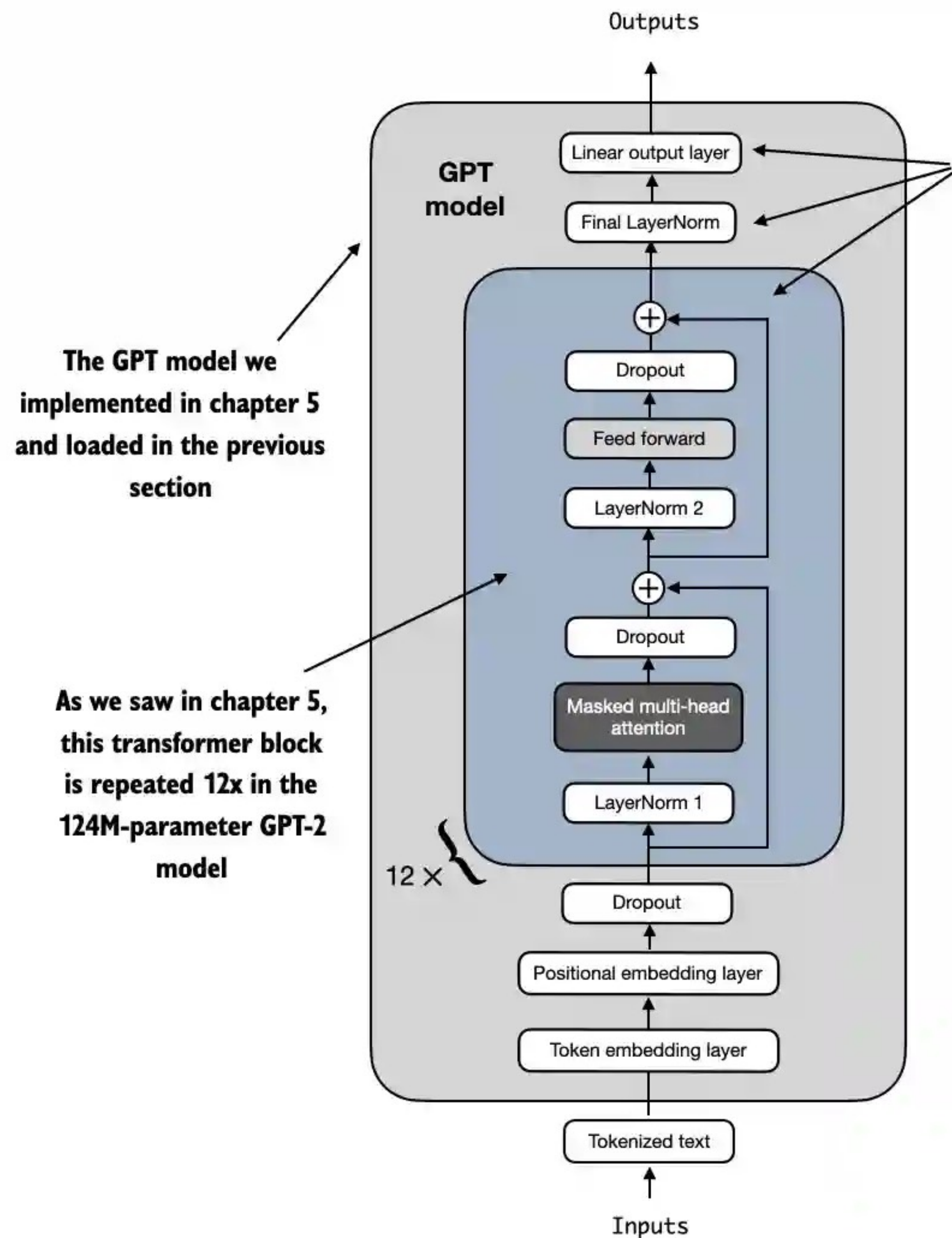
```
(1): TransformerBlock(
     (att): MultiHeadAttention(
…

(11): TransformerBlock(
     (att): MultiHeadAttention(
       (W_query): Linear(in_features=768, out_features=768, bias=True)
       (W_key): Linear(in_features=768, out_features=768, bias=True)
       (W_value): Linear(in_features=768, out_features=768, bias=True)
       (out_proj): Linear(in_features=768, out_features=768, bias=True)
       (dropout): Dropout(p=0.0, inplace=False)
     )
     (ff): FeedForward(
       (layers): Sequential(
         (0): Linear(in_features=768, out_features=3072, bias=True)
         (1): GELU()
         (2): Linear(in_features=3072, out_features=768, bias=True)
       )
     )
     (norm1): LayerNorm()
     (norm2): LayerNorm()
     (drop_resid): Dropout(p=0.0, inplace=False)
   )
 )
 (final_norm): LayerNorm()
 (out_head): Linear(in_features=768, out_features=50257, bias=False)
)
```

Feezing layers

```
for param in model.parameters():
    param.requires_grad = False
```

Replacing Outer Layer

```
torch.manual_seed(123)

num_classes = 2
model.out_head = torch.nn.Linear(
                    in_features=BASE_CONFIG["emb_dim"],
                    out_features=num_classes)
```

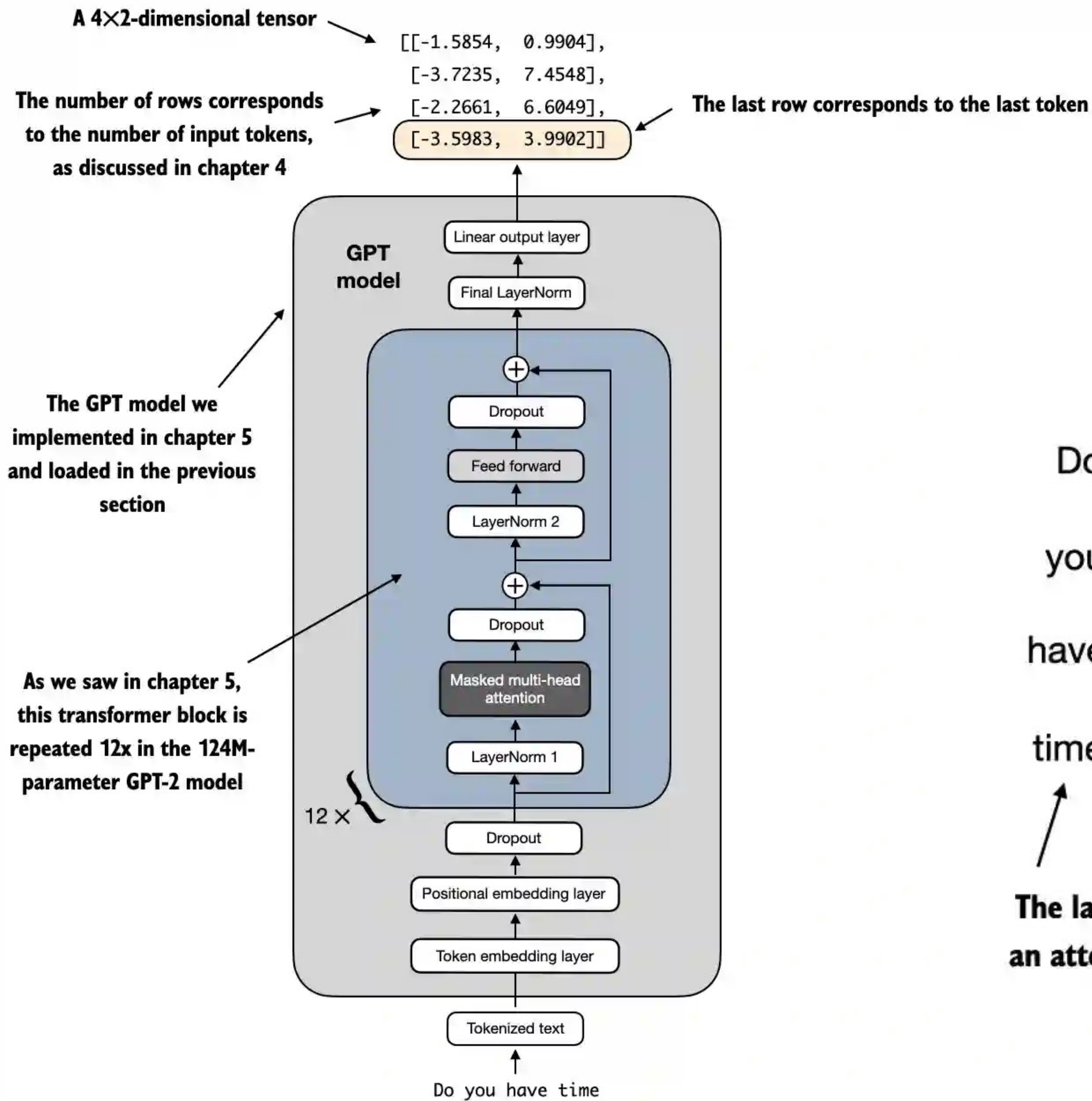



```
for param in model.trf_blocks[-1].parameters():
    param.requires_grad = True

for param in model.final_norm.parameters():
    param.requires_grad = True
```

Now Get Two Output

```
inputs = tokenizer.encode("Do you have time")
inputs = torch.tensor(inputs).unsqueeze(0)

with torch.no_grad():
    outputs = model(inputs)

print("Outputs:\n", outputs)
print("Outputs dimensions:", outputs.shape) # shape: (batch_size, num_tokens, num_classes)
```

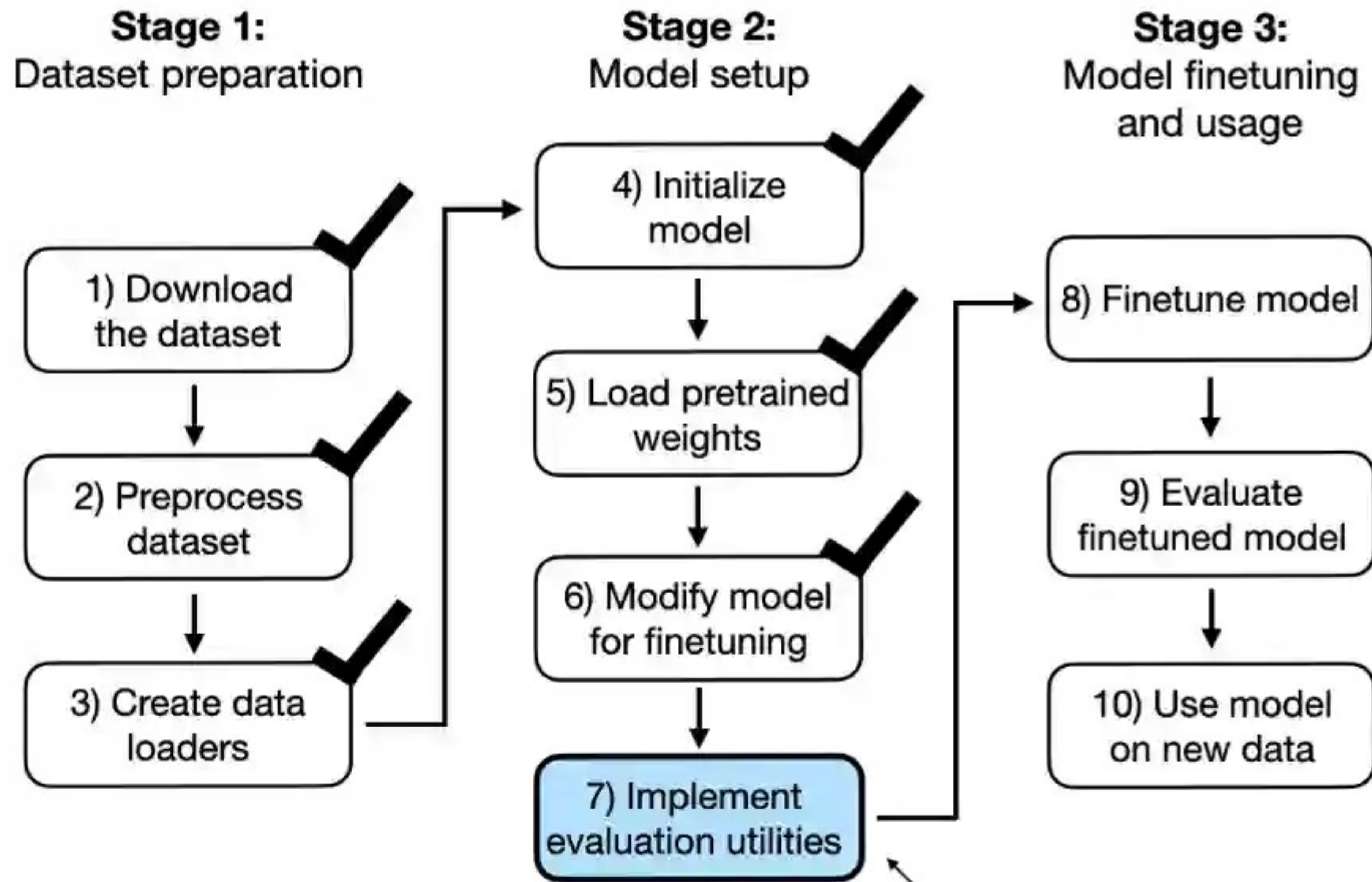
```
Outputs:
 tensor([[[-1.5854,  0.9904],
         [-3.7235,  7.4548],
         [-2.2661,  6.6049],
         [-3.5983,  3.9902]]])
Outputs dimensions: torch.Size([1, 4, 2])
```



Tokens masked out via the causal attention mask discussed in chapter 3

	Do	you	have	time
Do	1.0			
you	0.55	0.45		
have	0.38	0.30	0.32	
time	0.27	0.24	0.24	0.25

The last token is the only token with an attention score to all other tokens



In this section, we implement the utility function to calculate the classification loss and accuracy of the model