Econ 425 Week 7

Large language models

Grigory Franguridi

UCLA Econ

USC CESR

franguri@usc.edu

Language modeling

Predicting the probability distribution of word sequences in a language

Crucial for natural language processing (NLP) applications, e.g.,

- **Text generation**: generating coherent and contextually relevant text based on previous words
- **Speech recognition**: converting spoken language into text by determining the most probable sequence of words
- **Machine translation**: translating text from one language to another by predicting the sequence of words in the target language
  - example issue: translation of gender-neutral pronouns from Turkish to English when translating the phrase "he/she is a doctor." Google Translate interprets it as "he is a doctor," while the phrase "he/she is a nurse," is translated as "she is a nurse"

Language modeling

- **Spell check and autocorrection**: suggesting corrections or completions for partially typed or misspelled words
- **Information retrieval**: Improving search engine results by understanding the context and meaning of query terms
- if interested in learning more, please refer to this Princeton course: https://www.cs.princeton.edu/courses/archive/fall22/cos597G/

N-grams: intro

- continuous sequences of words/symbols/tokens in a document. Alternatively, neighboring sequences of items in a document
- have a wide range of applications, like language models, semantic features, spelling correction, machine translation, text mining, etc.
- N is the number of words/symbols/tokens that are grouped to form a sequence,
  see example below

N-grams: example

Sentence in training data: 'The cat sat on the mat'

- **Unigram (1-gram)**: 'The', 'cat', 'sat', 'on,' 'the', "mat"
- **Bigram (2-gram)**: 'The cat', 'cat sat', 'sat on', 'on the', 'the mat'
- **Trigram (3-gram)**: 'The cat sat', 'cat sat on', 'sat on the', 'on the mat'
- **N-gram**: similarly with N consecutive items

N-grams: example

Sentence in training data: 'The cat sat on the mat'

1. tokenize the sentence into words: ["The", "cat", "sat", "on", "the", "mat"]
2. create bigrams: ["The cat", "cat sat", "sat on", "on the", "the mat"]
3. count the frequency of each bigram:
   - "The cat": 1
   - "cat sat": 1
   - "sat on": 1
   - "on the": 1
   - "the mat": 1

N-grams: example

- **goal**: predict the next word after the phrase "the cat"
- look at the bigrams starting with "The cat" and find that "The cat sat" is the only bigram with "The cat" as the prefix
- therefore, our model predicts that the most likely next word after "The cat" is "sat"
- if we have a larger corpus with more sentences, we count the frequency of each bigram in the entire corpus and use these frequencies to calculate the probabilities of the next words

N-grams: example

Sentences in training data:

1. 'the cat sat on the mat'
2. 'the cat drank milk'

Frequencies:

- "the cat": 2 (frequency in the corpus)
- "cat sat": 1
- "cat drank": 1
- "sat on": 1
- "on the": 1
- "the mat": 1

N-grams: example

- **goal**: predict the next word after "the cat"
- probability of two options ("sat" and "drank") is calculated based on the frequency of the corresponding bigram:

$$\hat{P}(\text{"sat"}|\text{"the cat"}) = \frac{\hat{P}(\text{"the cat sat"})}{\hat{P}(\text{"the cat"})} = 0.5$$

$$\hat{P}(\text{"drank"}|\text{"the cat"}) = \frac{\hat{P}(\text{"the cat drank"})}{\hat{P}(\text{"the cat"})} = 0.5$$

- hence, both "sat" and "drank" are predicted to follow "the cat" with equal probabilities

N-grams models

- **N-gram: One-hot representation of words**
- An n-gram is a contiguous sequence of n items from a given sample of text or speech.
- In the context of one-hot representation of words, each word in the vocabulary is represented as a vector with a 1 in the position corresponding to the word's index in the vocabulary and 0s in all other positions. This representation is used in language models to predict the probabilities of the next word in a sequence.

N-grams models

Text corpus: "the cat sat on the mat"

- tokenize into words: ["the", "cat", "sat", "on", "the", "mat"]
- create 1-grams (unigrams): ["the", "cat", "sat", "on", "the", "mat"]
- create a vocabulary (unique 1-grams): ["the", "cat", "sat", "on", "mat"]

N-grams models

- represent each word in the vocabulary using one-hot encoding, a vector with length the size of the vocabulary:
  - "the": [1, 0, 0, 0, 0]
  - "cat": [0, 1, 0, 0, 0]
  - "sat": [0, 0, 1, 0, 0]
  - "on": [0, 0, 0, 1, 0]
  - "mat": [0, 0, 0, 0, 1]
- these vectors can then be used as input to various ML models for tasks like text classification, sentiment analysis, etc.
- one-hot representations are very sparse (mostly zeros) and do not capture any semantic information or relationships between words

N-grams models: bigram (Markovian)

- predicts the next word in a sequence based on the previous word
- assumes that the probability of a word depends only on the *immediately preceding* word, making it a first-order Markov model
- we have seen an example before

N-gram models: left-to-right/autoregressive

- predicts the next word in a sequence based on the previous $n-1$ *words*, following a left-to-right or autoregressive approach
- generates text by predicting one word at a time, conditioning on the previous words in the sequence

Autoregressive model: example

Text corpus: "the quick brown fox jumps over the lazy dog"

- tokenize the sentence into words: ["the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"]
- create trigrams (3-grams): ["the quick brown", "quick brown fox", "brown fox jumps", "fox jumps over", "jumps over the", "over the lazy", "the lazy dog"]
- could count the frequency of each trigram in our corpus, but for this example, assume each trigram appears once

Different types of N-grams models

- **goal**: predict the next word in the sequence "the quick brown"
- look at the 3-grams starting with "the quick brown" and find that "the quick brown fox" is the only 3-gram with "the quick brown" as the prefix
- hence, our model predicts that the most likely next word after "the quick brown" is "fox"
- if we want to continue generating text, we move one word to the right and consider the next 3-gram "quick brown fox". Again, we find that "quick brown fox jumps" is the only 3-gram with "quick brown fox" as the prefix, so we predict "jumps" as the next word

Different types of N-grams models

- repeat this process to generate text in a left-to-right (autoregressive) manner: "the quick brown fox jumps over the lazy dog"
- in applications, the n-gram model would be trained on a much larger corpus, and the probabilities of the next words would be calculated based on the frequencies of the n-grams in the corpus

Drawbacks of N-grams models

- **low-frequency words/high dimension**
  - some words do not appear often enough in the training data to accurately estimate their probabilities
  - hence, the dimensionality of the one-hot representations can be very high, esp. for large vocabularies, leading to computational/storage issues
- **smoothing technique: $+1$ to all words**
  - add 1 to the count of all words (or n-grams),
  - known as add-one or Laplace smoothing
  - ensures that every word has a non-zero probability, improving generalization

Classical text representations

- **Bag-of-words (BoW)**:
  - each document is represented as a vector indicating the frequency of each word in the vocabulary
  - *does not* capture word order or context
- **Word2Vec**:
  - represents words in a continuous vector space
  - *trained* to predict a word based on its context (CBoW) or to predict the context based on a word (Skip-gram)
  - still context-independent
- **Dense Vector:**
  - represents words as dense vectors aka word embeddings
  - capture semantic information about the words, meaning that words with similar meanings have similar vectors
  - lower-dimensional, data-driven

Classical text representations

- **Compressed Contextual Information**:
- context-dependent: the same word can have *different* embeddings depending on its context
- significant advancement over Word2Vec
- useful for tasks that require understanding of the context, such as text classification and summarization

ELMo: example

Text corpus:

1. "I read a book on the history of science"
2. "I will read a book on the beach"

- "read" appears in both sentences, but it has different meanings in each context:
  - past tense in the first sentence
  - future tense in the second sentence

ELMo: example

- ELMo generates embeddings for each instance of "read" based on the entire sentence
  - In the first sentence, ELMo embedding for "read" would capture the past tense and the context related to "history of science"
  - In the second sentence, ELMo embedding for "read" would capture the future tense and the context related to "beach"
  - hence, these two embeddings (vectors) are numerically different
- these embeddings can then be used in various NLP tasks. For example, in text classification, the ELMo embeddings can help the model distinguish between sentences about different subjects (history vs. leisure activities) and understand the temporal context.
- e.g., in named entity recognition (NER), these embeddings can help the model recognize that "history of science" is a subject or field of study, while "beach" is a location

More advanced embedding: BERT

BERT (Bidirectional Encoder Representations from Transformers):

- uses the *Transformer* architecture to generate context-dependent embeddings
- trained on a large corpus of text using tasks like *masked language modeling* and next sentence prediction
- BERT embeddings capture deep *bidirectional* context (information is passed in both directions: left-to-right and right-to-left), making them highly effective for a wide range of NLP tasks

Transformers

Drawbacks of RNN :

- **vanishing/exploding Gradients**: RNNs and their variants like LSTM and GRU can struggle with long sequences due to the vanishing and exploding gradients. These issues make it difficult for the model to learn dependencies between distant words in a sequence
- **sequential processing**: RNNs process data sequentially, which means they cannot take full advantage of modern parallel computing architectures. This makes training on long sequences time-consuming

Transformers

Key component: **attention mechanism**

- allows the model to weigh the importance of different words in a sentence based on their relevance to the current word or task. This is achieved through a set of scores that determine how much *attention* should be given to each word when generating a representation for a given word
- unlike RNNs, which process words one after another, attention in transformers can process all words in a sentence simultaneously, capturing dependencies between words regardless of their distance in the sequence

Transformers

Scaled dot-product attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$

- Q (Query): roughly, embedding of current word
  - in sequence-to-sequence models, come from the decoder, representing the current state or output that the decoder is trying to generate
- K (Key): compared to queries to compute attention scores
  - in sequence-to-sequence models, come from the encoder, representing the entire input sequence
- V (Value): used to compute weighted sum based on attention scores
  - in sequence-to-sequence models, typically come from the encoder and are used to create a weighted representation of the input sequence that is passed to the decoder

Transformers

- **softmax** converts attention scores $\mathbf{z} = QK^T$ to probabilities

$$\text{softmax}(\mathbf{z}) = \frac{e^{\mathbf{z}}}{\sum_{j=1}^{d} e^{z_j}}$$

- $d =$ **dimension of key = dimension of query**
- scaling by $1/\sqrt{d}$ helps prevent softmax from having extremely small gradients when dot products are large

Transformers

Advantage: parallel computing of gradients in long sequences

- since attention processes all words in a sentence simultaneously, the gradients for all words can be computed in parallel during training
- significantly speeds up the training process compared to RNNs, which must compute gradients sequentially
- makes transformers well-suited for handling long sequences and large datasets, leading to faster training times and better scalability

Transformers: architecture

Encoder-decoder architecture:

- **encoder** processes the input sequence and generates a set of representations
- **decoder** then uses these representations, along with previous outputs, to generate the next element in the output sequence
- commonly used in tasks like machine translation, where the input and output sequences can be of different lengths

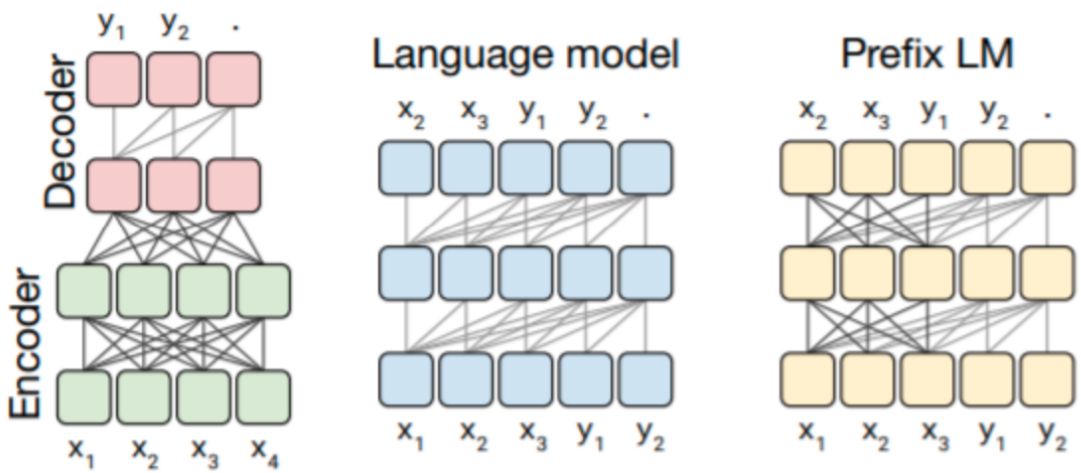
Transformer architectures

Decoder-only architecture:

- no encoder; decoder processes the input and generates the output in a single pass
- often used in language modeling tasks, where the model predicts the next word in a sequence based on the previous words
- example: GPT
- creates more *diverse* predictions; in GPT, the level of diversity is controlled by the *temperature* parameter

Transformer architectures

- Left: standard encoder-decoder architecture uses fully visible masking in the encoder and the encoder-decoder attention, with causal masking in the decoder
- Middle: single transformer layer stack and is fed the concatenation of the input and target, using a causal mask throughout
- Right: adding a prefix to a language model corresponds to allowing fully-visble masking over the input



GPT series

- Generative Pre-trained Transformer
- **decoder-only** architecture
- utilizes the core components of transformers
  - multi-head self-attention
  - positional encoding
  - residual connections
  - feedforward NNs
- processes the input and generates the output in a single pass
- example: given the input "the quick brown fox", GPT predicts the next word as "jumps" based on the context provided by the preceding words

GPT series

Advantage: generative/creative tasks

- autoregressive nature of GPT makes it particularly well-suited for generative tasks, where the goal is to produce *new and diverse* content based on a given context
- example: in story generation, given a prompt like "Once upon a time, there was a dragon", GPT can continue the story coherently and creatively, generating text like "who lived in a vast kingdom," which may *never* appear in the training corpus

GPT series vs BERT

In contrast, BERT is better at contextualized encoding

- While GPT is designed for generative tasks, BERT is optimized for tasks that require a **deep understanding of context**, such as question answering and sentiment analysis
- This is because BERT's bidirectional nature allows it to consider both the preceding and following context when encoding each token
- Example: in sentiment analysis, BERT can more accurately determine the sentiment of a sentence like "I am not happy with this product" by considering the context provided by the word "not", which changes the overall sentiment

GPT series

Extremely large number of parameters:

- 175 billion in GPT-4
- > 1 trillion in GPT-4 (estimated)

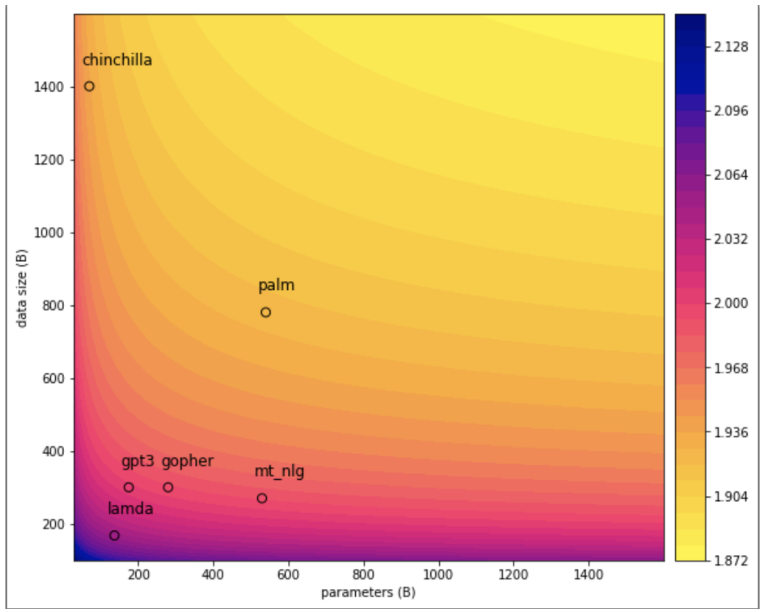
Pre-training:

- using a diverse range of internet text
- achieves great performance on a wide variety of tasks, even with little to no task-specific training data

Data size matters[1]: an empirical LLM scaling law

Let N denote model size and D denote training data size.

$$\text{LM loss } L(N, D) = \frac{A}{N^{\alpha}} + \frac{B}{D^{\beta}} + C$$

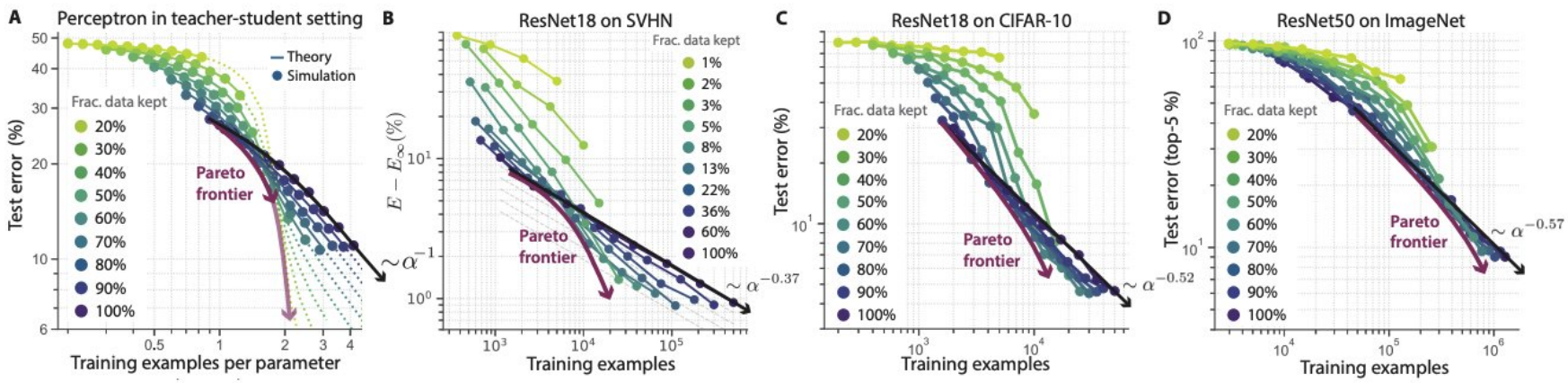


If we can leverage enough data, there is no reason to train 500B+ param models

[1] Training compute-optimal large language models by DeepMind

Data quality matters more[2]: beyond the scaling law

Removing the "easy training samples" (defined in a self-supervised manner) improves scaling law from power level to exponential level. It means that we could go from 3% to 2% error by only adding a few carefully chosen examples, rather than collecting 10× more random ones



[2]Beyond neural scaling laws: beating power law scaling via data pruning by Meta AI

2021 NeurIPS workshop on data-centric AI

- 99% model-centric papers (e.g., network architecture, algorithm design, feature engineering) v.s. 1% data-centric papers (e.g., data sourcing/augmentation/generation):

	Steel defect detection	Solar panel	Surface inspection
Baseline	76.2%	75.68%	85.05%
Model-centric	+0% (76.2%)	+0.04% (75.72%)	+0% (85.05%)
Data-centric	+16.9% (93.1%)	+3.06% (78.74%)	+0.4% (85.45%)

- Current status of data-centric AI:
  - fairly ambiguous since annotators are the source of data and ground truth
  - rapidly developing, but no active research community yet
  - absence of tools, best practices, and infrastructure for managing data in ML systems

Data-centric AI

Other examples:

(i) data sourcing/labeling/pre-processing/cleaning (1st generation)

(ii) data generation/augmentation/pruning (algorithm-based, 2nd generation)

(iii) data governance & compliance (regulation-driven, e.g., fairness and privacy)

OpenAI API

- provides access to advanced natural language processing (NLP) models developed by OpenAI, including the famous GPT series
- allows developers to integrate state-of-the-art language models into their applications, enabling a wide range of functionalities such as text generation, language translation, question answering, and more
- not free, but super cheap;
  credits for academic researchers or startups (through Microsoft)

OpenAI API

Key features:

- **Ease of use**: designed to be user-friendly, allowing developers to interact with powerful language models using simple HTTP requests
- **Scalability**: can handle requests ranging from small-scale applications to large, enterprise-level deployments
- **flexibility**: supports various use cases, including chatbots, content creation, language analysis, and more
- **Customization**: developers can fine-tune the responses of the language models to better suit their specific needs
- **Security**: OpenAI ensures data privacy and security, making it suitable for use in sensitive and commercial environments

OpenAI API: example

(need to generate an API key)

```
import openai

openai.api_key = 'your-api-key'

response = openai.Completion.create(
  model="text-davinci-003",
  prompt="Once upon a time, there was a dragon",
  max_tokens=50
)

print(response.choices[0].text.strip())
```

OpenAI API

Configuration:

- use the model 'text-davinci-003' to generate a continuation of the story prompt "once upon a time, there was a dragon"
- set the 'max_tokens' parameter to specify the maximum length of the generated text.
- output might look something like this: **'who lived in a vast kingdom. The dragon was feared by all, but it had a secret: it was actually a gentle creature who loved to read books and learn about the world.'**
- parameter: sampling temperature $\in [0, 1]$;
  higher values make the output more random, lower values make it more focused and deterministic;
  (try different temperatures after class)

NYT-API

- set of web services provided by The New York Times that allows developers to access and use various types of content from the newspaper's vast database
- covers news articles, book reviews, movie reviews, and more
- using the NYT API, developers can integrate The NYT content into their own applications, websites, or research projects
- provides access to current and historical articles, metadata, and other information, enabling developers to create rich, informative experiences for their users

NYT-API

- typically accessed through HTTP requests, and the responses are usually in JSON format
- developers need to sign up for an API key to use the services, and there are different access levels and rate limits depending on the specific API and the usage plan
- overall, a valuable resource for developers looking to incorporate high-quality news and information into their projects

Sentiment analysis

- computational task of identifying and categorizing opinions expressed in a piece of text, especially to determine whether the writer's attitude towards a particular topic, product, etc., is positive, negative, or neutral
- widely used in business and marketing for analyzing customer reviews, social media comments, and survey responses to gauge public opinion about products, services, or events

Sentiment analysis: feature extraction

Feature extraction for attitude:

- **Lexicon/rule-based methods**: rely on predefined lists of words and phrases with associated sentiment scores. Examples:
  - **AFINN**: assigns integer sentiment scores to words, ranging from -5 (very negative) to +5 (very positive)
  - **SentiWordNet**: extension of WordNet that assigns sentiment scores to WordNet synsets (sets of synonyms), allowing for more nuanced sentiment analysis
- **Contextualized embedding**: a data-driven approach, unlike lexicon-based methods
  - captures the meaning of words in context
  - models like BERT or ELMo provide word embeddings that are sensitive to the surrounding words, enabling more accurate sentiment analysis in complex sentences

Existing models: classifiers

- **SVM (Support Vector Machine)**: a popular machine learning model used for classification tasks, including sentiment analysis. SVMs can efficiently handle high-dimensional data and are effective in separating positive and negative sentiment classes
- **Naive Bayes**: a simple probabilistic classifier based on Bayes' theorem; particularly suited for sentiment analysis due to its ability to handle large feature sets and its effectiveness in text classification tasks

Sentiment analysis: example

"I love this product, but the color is not what I expected."

- using a lexicon-based method like AFINN:
  - "love" might have a positive score (+3)
  - "not" might be used to invert the sentiment of the following word
  - "expected" might have a neutral score (0)
- overall sentiment score could be calculated by **summing the scores of individual words** and considering the negation. In this case, the sentiment might be considered slightly positive due to the presence of the word "love"

Sentiment score with LLM

To perform sentiment analysis using OpenAI's API (assuming you're using the GPT model), you would typically follow these steps:

- **Set up your environment**: make sure you have the OpenAI Python package installed and your API key is set up correctly
- **prepare your prompt**: create a prompt that instructs the model to perform sentiment analysis, e.g., ask the model to rate the sentiment of a sentence on a scale from 0 to 1
- fine tuning (model weights will be updated) $\longrightarrow$ prompt tuning (model weights remain the same)
- **Call the API**: use the OpenAI API to send your prompt to the model and receive the response

Sentiment score with LLM: OpenAI API

The 'analyze_sentiment' function sends a prompt to the OpenAI API requesting to rate the sentiment of a given sentence:

```
import openai

openai.api_key = 'your-api-key'

def analyze_sentiment(text):
    response = openai.Completion.create(
        engine="text-davinci-003",
        prompt=f"Rate the sentiment of the following sentence on a scale from 0
        max_tokens=10,
        temperature=0
    )
    return response.choices[0].text.strip()

# Example usage
sentence = "I love this product, but the color is not what I expected."
sentiment_score = analyze_sentiment(sentence)
print(f"Sentiment score for '{sentence}': {sentiment_score}")
```

Sentiment score with LLM: OpenAI API

The output returned by API:

```
Sentiment score for 'I love this product, but the color is not what I expected.': 0.7
```

The score is a number between 0 and 1, where 0 represents a very negative sentiment and 1 represents a very positive sentiment