Lab Session #07

Introduction

Welcome to Lab #07: This week, we'll practice more machine learning with *scikit-learn* and start developing chatbots using two of the techniques covered in the lecture, pattern-based bots and search-based bots.

Follow-up Lab #6

Here are some solution notes for the previous lab. As always, if you have any questions, please ask in the Moodle Discussion Forum!

Solution Task #1 (TF-IDF Vectors)

Here's a sample program for this task from last week that prints out various intermediate steps.

Solution Task #2 (Search)

Here's a solution for the "re-implementation of Google"

Solution Task #3 (k-Means)

Here's an example solution for the first part (clustering the test sentences).

Task #1: k-Nearest Neighbors (kNN)

Let's experiment with the kNN algorithms we covered in the lecture: k-Nearest Neighbors (kNN) is a versatile algorithm used in machine learning for both classification and regression tasks. In classification, kNN assigns a class to a data point based on the majority class among its k closest neighbors. For regression, it predicts a value by averaging the values of its k nearest neighbors. This simplicity in principle belies its effectiveness in many practical applications.

Task #1.1: kNN Regression

The goal here is to apply what we've learned by implementing a kNN regression, as practiced in lecture Worksheet #5. First, we need to import the necessary libraries to work with our dataset and the kNN regressor using scikit-learn:

```
import numpy as np

from sklearn.neighbors import KNeighborsRegressor
```

Next, let's define our dataset. Use the samples provided in the worksheet, which we'll input as an array:

```
dataset = np.array([[135, 0, 5, 3], [90, 123, 2, 5], [159, 2, 1, 1]])
```

To set up our feature vectors (the input variables for our model), we select the first three columns of our dataset:

```
X = dataset[:, 0:3]
```

The target values (the output our model should learn to predict) come from the last column of the dataset:

```
y = dataset[:, 3]
```

Now, create a kNN regressor object with `n_neighbors=2`. This tells the model to consider the 2 nearest neighbors when making predictions:

```
clf = KNeighborsRegressor(n_neighbors=2)
```

Train the model with our dataset by fitting it to our features and target values:

```
clf.fit(X, y)
```

For the prediction step, you'll need to use your trained model to predict the output for new data. Imagine we have new test data with features [109, 5, 3]. You should predict the output for this new set of features using:

```
prediction = ...
```

Hint: Review the scikit-learn documentation on KNeighborsRegressor to find how to make predictions with your model.

A Note on Normalization

Normalization is a preprocessing step which involves scaling the features in your dataset so that they have a uniform range. This is important because features often come in various scales and units, making it challenging for many machine learning algorithms to learn effectively. Algorithms that rely on the distance between data points, like kNN, can be particularly sensitive to this issue.

Looking at our previous dataset, you'll notice the features vary widely in scale. For example, the first feature ranges from 90 to 159, while the second feature has values

close to 0 for two samples and 123 for another. This difference in scale can disproportionately influence the model's predictions.

In scikit-learn, normalization can be easily accomplished using the `StandardScaler` or `MinMaxScaler`. Here's how you might normalize our dataset with `StandardScaler`:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_normalized = scaler.fit_transform(X)
```

After normalization, each feature in your dataset will contribute equally to the distance computation, ensuring that the model's performance is not skewed by the scale of the features.

Feature normalization is not only crucial in kNN but is broadly applicable across many machine learning algorithms, especially those involving distance calculations (like SVMs and k-means clustering) and gradient descent optimization (like neural networks). It helps improve the convergence speed and the accuracy of these models, making it an essential step in the preprocessing pipeline.

Task #1.2: kNN Classification

Classification is a type of supervised learning where the goal is to predict the categorical class labels of new instances, based on past observations. In kNN classification, the class of a sample is determined by the majority vote of its k nearest neighbors in the feature space. For these experiments, we will use the *kNN classification* algorithm as discussed in the lecture. This one is also available in *scikit-learn*:

```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
```

To see how this works, let's start with some real data, using one of the example datasets available with *scikit-learn*. Here, we will be using the *wine* dataset:

```
from sklearn.datasets import load_wine
X, y = load_wine(return_X_y=True)
```

The wine dataset is a classic machine learning dataset that contains the chemical analysis of 178 wine samples from three different cultivars, featuring 13 different attributes such as alcohol content, color intensity, and phenols.

Now create the train and test data. Use scikit-learn's train_test_split helper function to split the wine dataset into a training and testing subset:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
```

Here, the data is split into 80% training data and 20% testing data.

Recall our discussion on normalization from Task #1.1. Scaling the features, as we've already learned, is crucial for the effectiveness of kNN algorithms. Apply what we discussed:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

Now you can "train" a classifier (for kNN, this simply stores the vectors with their labels):

```
clf = KNeighborsClassifier(n_neighbors=3)
clf.fit(X_train, y_train)
```

Here, "3" is *k*, the number of neighbors voting when classifying unseen data (see the documentation). Note that this is a standard pattern when creating a ML model with scikit-learn, you can use other algorithms (e.g., Naive Bayes, SVM) in the same way.

Now you can make predictions on the test data:

```
y_pred = clf.predict(X_test)
```

Evaluate the performance of your classifier

Evaluating the performance of your machine learning model is crucial, as it provides insights into how well the model is likely to perform on unseen data. This step helps us understand the model's strengths and weaknesses, guiding further improvements.

To understand the effectiveness of your kNN classifier on the wine dataset, we use several key metrics:

- **Precision** tells us the proportion of positive identifications that were actually correct. For example, if the model predicts a wine sample as belonging to Cultivar 1, precision measures how many of these predictions were accurate.
- **Recall** indicates the proportion of actual positives that were correctly identified. Using the wine dataset, it would measure how well the model is at identifying all samples of Cultivar 1.
- **F1-measure** is the harmonic mean of precision and recall, providing a single metric to assess the balance between them.
- **Accuracy** measures the overall correctness of the model across all wine samples and classes.

To compute these metrics and understand your model's performance, use the *scikit-learn* evaluation tools, leaving the actual calls to the functions for you to investigate:

```
from sklearn.metrics import classification_report, confusion_matrix

print(confusion_matrix(...))

print(classification_report(...))
```

The confusion matrix provides a visual understanding of the types of errors your model is making, showing the correct and incorrect predictions across different classes. The classification report then summarizes the precision, recall, F1-score, and accuracy for each class, giving a detailed view of how well your model performs for each type of wine.

Interpreting these outputs lets you see not just how accurately your model predicts, but also how it balances its predictions across the different classes of wine, highlighting areas for potential improvement.

Task #2: Building Your First Chatbot

One of the foundational techniques for creating chatbots involves utilizing question-answer patterns, where regular expressions match the user's input, and specific responses are generated based on these patterns. While this method has its limitations, such as difficulty in handling unexpected queries or managing complex conversations, it's an excellent starting point for understanding chatbot basics. Moreover, pattern-based bots can be effectively integrated with other chatbot development techniques discussed in our course, offering a robust foundation for more advanced applications.

We'll explore the use of the Artificial Intelligence Markup Language (AIML), a standard that, despite the shift in active development to newer platforms, remains a vital part of chatbot history and development. AIML's simplicity and accessibility make it an ideal tool for beginners.

To get started, you need one of the AIML-compatible libraries, e.g., AIML-Bot. Install the library and create your first bot:

```
import aiml_bot
bot = aiml_bot.Bot(learn="mybot.aiml")
while True:
    print(bot.respond(input("> ")))
```

You'll need an AIML file (which is in XML format) to define the question/answer patterns. Here is a first one to test:

```
<?xml version="1.0" encoding="UTF-8"?>
<aiml version="1.0.1">
  <category>
    <pattern>HELLO *</pattern>
    <template>Hi Human!</template>
  </category>
  <category>
    <pattern>HELLO TROLL</pattern>
    <template>Good one, human.</template>
  </category>
</aiml>
```

Experiment with generating <random> responses as shown in the lecture.

To avoid duplicating patterns for the same kind of interaction, you can use the <srai> tag:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<aiml version="1.0.1">
  <category>
    <pattern>WHAT IS COMP 474</pattern>
    <template>COMP 474 is the Intelligent Systems course.</template>
  </category>
  <category>
    <pattern>WHAT IS COMP 6721</pattern>
    <template>COMP 6721 is the Applied Artificial Intelligence course.</template>
```

```
  </category>
  <category>
    <pattern>DO YOU KNOW WHAT * IS</pattern>
    <template>
      <srai>WHAT IS <star/></srai>
    </template>
  </category>
</aiml>
```

As you can see, questions matching the pattern `DO YOU KNOW WHAT * IS` are now redirected to the patterns for `WHAT IS *`.

You can find various AIML files online; for example, there is a std-65% AIML file that covers "65% of all standard questions" that a bot could be asked. Try some of them out, but remember that the Python AIML implementations only support AIML v1, so AIML v2 files will not work.

Where to Go from Here

The landscape of bot development is ever-evolving, with a myriad of platforms offering diverse capabilities for creating conversational agents. As we've seen, pattern-matching remains a fundamental technique, but many platforms have extended their functionality to include more sophisticated methods.

- **Pandorabots**: Offers AIML 2.0 for advanced pattern-matching capabilities and operates on a proprietary cloud framework. More at Pandorabots.
- **Amazon Lex**: Amazon's service for building conversational interfaces into any application using voice and text. Lex benefits from deep integration with AWS's ecosystem. Learn more at Amazon Lex.
- **Google Dialogflow**: Known for its natural language understanding (NLU) capabilities, Dialogflow enables developers to incorporate pattern-matching alongside intent recognition for a more nuanced conversation flow. Details at Dialogflow.
- **Microsoft Azure Cognitive Services for Language**: This suite, which includes former QnA Maker capabilities, provides tools for building, testing, and deploying conversational AI applications, with services for language understanding, translation, and speech recognition. More information at Azure Cognitive Services for Language.
- **IBM Watson Assistant**: Leverages IBM's AI to offer a comprehensive development environment for chatbots, capable of understanding natural language input and automating interactions at scale. Visit Watson Assistant for details.

For developers interested in open-source alternatives, several projects offer a range of capabilities for chatbot development:

- Rasa: Provides tools for creating conversational AI with a strong emphasis on machine learning for understanding and generating user interactions (we will cover Rasa in one of the next labs).
- ChatterBot: An easy-to-use library for building chatbots that can learn from conversations with a simple API for pattern-based dialogue.
- Hubot: Developed by GitHub, Hubot is a customizable life embetterment robot that can be integrated into various chat services.
- Errbot: A Python-based bot designed to simplify the creation and management of chatbots for multiple platforms with a wide range of plugins.
- Will: An easy-to-extend chatbot framework that enables building capabilities on top of several messaging platforms with simple syntax.

As the field of conversational AI advances, integrating a variety of chatbot development techniques becomes essential for crafting sophisticated and responsive agents. These techniques, including pattern matching, search-based methods, grounding in real-world information, and generative models powered by deep neural networks (transformers, LLMs), each contribute unique strengths. Together, they enhance a chatbot's ability to understand and interact with users in a more human-like manner.

Pattern matching serves as a reliable foundation for addressing common queries with predetermined responses, offering quick and precise replies to frequently asked questions. Complementing this, search-based methods dynamically retrieve information from databases or the internet, providing relevant and current answers. Grounding techniques ensure that chatbots can anchor their conversations in real-world contexts, improving the relevance and appropriateness of responses. Finally, generative models, especially those leveraging LLMs, are capable of producing unique and contextually relevant replies. This enables more natural and engaging interactions by closely mimicking human conversational patterns.

In practical applications, the strategic combination of these techniques allows for the development of chatbots that are capable of efficiently managing routine interactions and engaging in complex conversations. These advanced interactions require understanding, reasoning, and creativity—qualities that are achieved by using pattern matching for rapid FAQ responses, search-based methods for in-depth inquiries, grounding for context maintenance in extended dialogues, and generative models for crafting personalized and nuanced replies.

This holistic approach to chatbot development not only fosters the creation of versatile, intelligent, and user-friendly conversational agents but also meets a wide spectrum of user needs and expectations across various scenarios, such as customer service, education, and entertainment. By leveraging the strengths of each method, developers can create chatbots that engage in simple dialogues, accurately deliver information,

comprehend context, and generate indistinguishable human-like responses, thus broadening the scope and effectiveness of real-world bots.

Task #3: Search-based Chatbots

In this task, we explore the search-based bot technique, which leverages Information Retrieval (IR) to find answers closely matching a user's query within a corpus.

To compute the similarity (*user question* vs. *corpus question* or *user question* vs. *corpus answer*), we'll use the same techniques that you already developed in previous labs: Vectorization using TF-IDF and similarity computation using the *cosine similarity*. So, your task is to write a program that:

- Takes a user's question as input and converts it into a TF-IDF vector;
- finds the closest matching question in the dataset by using cosine similarity;
- and prints out the existing answer for that question from the dataset.

For our experiments, we'll use the Amazon Q&A dataset that you can download at https://cseweb.ucsd.edu/~jmcauley/datasets.html#amazon_qa. This dataset is a rich collection of question-answer pairs across various product categories, providing a realistic setting for testing our search-based chatbot.

To start, choose a single category from the "Per-category files" list to simplify the initial experiments. After downloading the zip file, use the following code to load your chosen dataset into your program:

```
import gzip

def read_file(path):
    g = gzip.open(path, 'r')
    for l in g:
        yield eval(l)


dataset = read_file('qa_Appliances.json.gz')
```

Extract questions and their corresponding answers from the dataset, focusing solely on data entries with "question" and "answer" as key values:

```
question_list = []

answer_list = []

for i in dataset:
    question_list.append(i['question'])
    answer_list.append(i['answer'])
```

Next, convert the list of questions to an array using Numpy's asarray() function for easier manipulation:

```
import numpy as np

question_dataset = np.asarray(question_list)
```

Vectorize the natural language data into a TF-IDF matrix using TfidfVectorizer():

```
# Your TF-IDF Vectorizer code here
```

Now, prompt the user for a question:

```
input_question = input("What is your question: ")
```

After capturing the user's question, vectorize this input in the same way as the dataset to prepare it for similarity comparison. Then, find the most similar questions in the dataset by computing the cosine similarity between them and the user's question.

For example, the highest cosine similarity question for the question *"Is the blender powerful?"* (in the appliances data set) should be:

```
Similar question to user's question: Is the Genuine OEM FSP Whirlpool Kitchen Aid Blender

Rubber Seal part number 9704204 suitable for Kitchenaid blender KSB560CU1?

Answer given to the similar question: Not sure, but it did work fine for me.

FYI - It is inexpensive enough to order it and try.

If it does not fit, send it back. As an alternative you should be able to

contact the company to get a better answer to your question.
```

Experiment with not just finding the closest match but also identifying the top-*n* similar questions, enhancing the chatbot's utility and user interaction.

In the era of conversational AI, search-based bots remain a relevant technique for instances where precise, information-rich responses are required from a predefined dataset or corpus. While Large Language Models (LLMs) have revolutionized the field with their ability to generate human-like text, search-based bots offer unmatched accuracy in delivering specific information directly linked to user queries, making them indispensable for applications like customer support, technical troubleshooting, and any scenario where correctness and direct relevance take precedence over conversational fluidity.

Conclusions

In this lab, we've covered some more foundations in machine learning and then explored two key techniques in the development of chatbots: creating a pattern-based chatbot using AIML and developing a search-based chatbot leveraging IR techniques.

These exercises provided foundational knowledge in building and understanding different types of chatbot technologies.

Chatbots have significantly impacted various industries, most notably in customer support. Studies, such as those by Gartner, predict that by 2027, chatbots will become the primary customer service channel for roughly a quarter of organizations. This shift not only represents a substantial cost saving for businesses but also offers 24/7 customer service capabilities, and in many cases, more efficient resolution of customer inquiries. Beyond customer support, chatbots find applications in sectors like healthcare for patient triage, in e-commerce for personalized shopping experiences, and in education for tutoring and support.

For you, acquiring skills in bot development is not just about understanding the current landscape of conversational AI but also about preparing for a future where such technologies play a central role in the digital economy. Proficiency in creating and managing bots opens up career opportunities across a wide range of industries, where businesses seek to leverage AI to improve customer engagement, automate processes, and enhance user experiences.

While we have covered pattern-based and search-based chatbot development in this lab, the exploration doesn't end here. In subsequent lectures and labs, we will delve into the other two critical techniques of bot development: grounding and generative models. These sessions will further equip you with the skills needed to build more advanced and contextually aware chatbot systems, preparing you for the challenges and opportunities in the field of conversational AI.

That's all for this lab!