Janex & Eureka Guidebook

Jack Franklin

Introduction

Janex: Python

Janex-Python is the original version of Janex, which focuses on using the physical structure of words and lettering to decide similarity, rather than the semantics or meanings behind them. It also provides many basic functions needed to build a more complex AI program.

Janex: PyTorch

Janex-PyTorch is a far more advanced branch of the Janex family, as it utilises a Long Short-Term Memory Network combined with this data to fulfill intent classification requests.

Janex: NLG

Janex: NLG is by far the most powerful edition of Janex. It can rip down pieces of text into sentences, and then register which words come before and after each word, saving them to a database, and then reusing these patterns, with use of an LSTM, can predict the following words based on the trends in the data you feed to it.

Janex: Spacy

Janex: Spacy is a slightly new approach to intent classification. It uses the spacy models to convert words into numerical representations of themselves, and then use basic maths and cosine similarity calculations to determine the similarity between these words and the ones provided in the intents, using the highest match to classify inputs.

Janex: Jarvis

And Jarvis – he isn't a library, it is an AI which uses all four previous Janex versions coinciding with each other, using Janex: NLG for text generation, Janex: PyTorch for intent classification, Janex: Python for text manipulation and Janex: Spacy to understand semantic input. He can be infinitely configured to suit your needs, straight from something as simple as modifying the config.json file, and can be spoke to via Discord, Whisper, Terminal, or others.

Eureka-Py

Eureka-Py is a library which allows you to connect to the Large Language Model known as Eureka. This guide will explain how you can incorporate this Artificial Intelligence into your projects without needing to run it locally, using simple commands from the Eureka-Py library.

Before we get started:

A few of the Janex libraries may require a pre-defined intents.json file. Some Janex files, such as PyTorch, have backup systems to automatically download these files, so make sure you have a strong internet connection before usage.

However, if you would like a custom intents.json file, filled with your own commands and small-talk, you can see the structure of the intents file stored in 'Jarvis/AI' and then work on it yourself!

Janex: Python Edition

Intent Classification

The 'original' version of Janex' intent classifier worked by physically breaking down the text input and the available intents, and then matching the input with the patterns by their numeric and alphabetical composite.

While this is a very literal approach and is subsequently almost instant and lightweight, this literal method may lead to high inaccuracies with large sets of data, and so is only recommended for experimenting and/or for small basic datasets with unique words.

Whatsmore, if it identifies nouns, it will add an extra amount of similarity to that portion due to nouns being uncommon and therefore should be recognised as more important than common generic words.

However, from versions 0.80.0 and up, the method was enhanced slightly to increase accuracy. Here is how the code would look after Janex is installed via pip.

```python
from janex.intentclassifier import *

Classifier = IntentClassifier()

Classifier.set_intentsfp("intents.json")
Classifier.set_vectorsfp("vectors.json")
Classifier.set_dimensions(300)

Classifier.train_vectors()
```

```
Input = input("You: ")

classification = Classifier.classify(Input)

response = random.choice(classification["respor

print(response)
```

Other tools (Word Manipulation)

Janex also holds standalone tools which are used for larger Natural Language Processing (NLP) tasks.

This includes:

Tokenizing:

Breaking up sentences and blocks of text into words, and remembering the common patterns between them.

```
from Janex.word_manipulation import *

string = "Hello. My name is Brendon."

tokens = tokenize(string)

print(tokens)
```

Stemming:

Reducing whole words down to their root form, removing tense or extensions.

Vectorizing:

Finding numeric representations of words to be used in mathematical cosine computations, stored in a numpy array (a set of numbers, the number of numbers depends on the set dimension).

```python
from Janex.vectortoolkit import *

input_string = "Hello, my name is Sheila."

vectors = string_vectorize(input_string)

vectors = reshape_array_dimensions(vectors, 300) # To reshape the vector array

secondstring = "Hello, my name is Robert."

second_vectors = string_vectorize(secondstring)

second_vectors = reshape_array_dimensions(second_vectors, 300)

similarity = calculate_cosine_similarity(vectors, second_vectors)

print(similarity)
```

Janex: PyTorch Edition

Intent Classifier & Response Generator

Janex-PyTorch, as described before, uses the PyTorch library in order to form a feed-forward neural network. This means the input data is fed into the 3-parameter network, along with the pre-trained .pth data harvested from your intents, and then a match is predicted using the similarity.

If that's not confusing enough, here's an excerpt!

```python
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, self).__init__()
        self.l1 = nn.Linear(input_size, hidden_size)
        self.l2 = nn.Linear(hidden_size, hidden_size)
        self.l3 = nn.Linear(hidden_size, num_classes)
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out = self.l2(out)
        out = self.relu(out)
        out = self.l3(out)
        # no activation and no softmax at the end
        return out
```

But don't worry, you don't really need to touch any of that internal code. You just need to integrate the Janex syntax in a similar fashion to Python-Edition, but slightly altered to adjust to this edition's needs.

How to use? 🔗

Firstly, you need to install the Janex: PyTorch Edition library.

```
pip install JanexPT
```

Secondly, you need to import JanexPT into your code.

```
from JanexPT import *
```

Next, create an instance of the JanexPT class and define the path to your intents, thesaurus, and the name you wish to give to your AI.

```
intents_file_path, thesaurus_file_path, UIName = "intents.json", "thesaurus.json", "May" # Use which

janexpt = JanexPT(intents_file_path, thesaurus_file_path, UIName)
```

Then, you will need to train your intents data into a pth file. If you do not have a training program, it will curl install the pre-built one from this repo.

```
janexpt.trainpt()
```

Next, you need to give the program some text you wish to send to your AI, and then send it.

```
YourInput = input("You: ")
YourName = "Bob" # Whatever you wish your AI to refer to you as

classification = JanexPT.pattern_compare(YourInput, YourName)
response = JanexPT.response_compare(input_string, classification)

print(response)
```

And there we have it, the code will use a triple-layer NeuralNet to predict which class your input belongs in, and then uses the Janex library to pick the best response from those available.

This code creates an instance of JanexPT, and then with the intents file directory and thesaurus file directory defined, with also the name of your PyTorch chatbot added, you can start pattern compare.

As soon as you run the program, in terminal will appear a download progress bar, as the library has detected that you do not have the built-in training program.

As such, once it downloads the file, it will attempt to automatically run it (if it doesn't and instead shows an error, try running the train.py file yourself from within the directory and then run your main program again.)

Once this is complete, and your program is running, the chatbot should be functional and you're good to go!

Janex: NLG

Janex NLG is the only version that does not utilise intent classification for its tasks. Instead, it is designed to generate text using large blocks of text that are fed into it.

Unlike other Janex versions, this one uses multiple 3rd party libraries to function – PyTorch & Spacy – PyTorch used to create a long-short term memory, and Spacy to compute the vectors (numerical representations) of text trained into the binary file.

In simple terms, you need to create a directory for your AI to be in. Then, create a folder within this directory called, for example, 'files'. To diversify your data, I would recommend using movie scripts and/or bible txts, the more data, the more patterns are collected, and then subsequently the better the results.

Janex: NLG was never designed for permanent use, and may improve as time goes on. At the time of writing this, it has not met a usable or practical level of word generation, but I hope that changes soon.

Here is the code for training a Janex NLG model…

Training the model 🔗

First, I would recommend creating a file named 'train.py' which you would use to create the binary file.

In this file, you would write:

```
from JanexNLG.trainer import *

NLG = NLGTraining() # Create an instance of the JanexNLG training module.
NLG.set_directory("./files") # Set this to the name of a folder in the same direc
NLG.set_spacy_model("en_core_web_md") # You can set this to any Spacy model of yc
NLG.train_data() # Finally, train the data. This will save everything collected i
```

Optional GPU support:

```
NLG.set_device("cuda")
```

After potentially a few minutes, or hours, depending on both your hardware and the diversification of your text examples, the training program will have generated a binary file to store your model in. That said and done, you can then activate it.

Using the model 🔗

Once you've created the binary data, effectively teaching the AI the connections between words and sentence structures, you can then use it to generate text.

```
from JanexNLG import *

Generator = NLG("en_core_web_md", "janex.bin") # Your chosen spacy model and the
input_sentence = input("You: ")
ResponseOutput = Generator.generate_sentence(input_sentence)
print(ResponseOutput)
```

If you would like to add more data to the binary file, but do not want to train the model from scratch, you can do this with the built-in finetuning feature.

Finetuning the model 🔗

For versions > 0.0.2, a finetuning feature is available. After training your model, if you wish to add extra modifications to alter the model for a specific purpose, you can set the directory to a new folder, put these new data pieces in there, and then continue to finetune the model.

```
from JanexNLG.trainer import *

NLG = NLGTraining()
NLG.set_directory("./files_for_finetuning")
NLG.set_spacy_model("en_core_web_md")
NLG.finetune_model("janex.bin") # You've got to add your model name to this funct
```

Janex: Jarvis
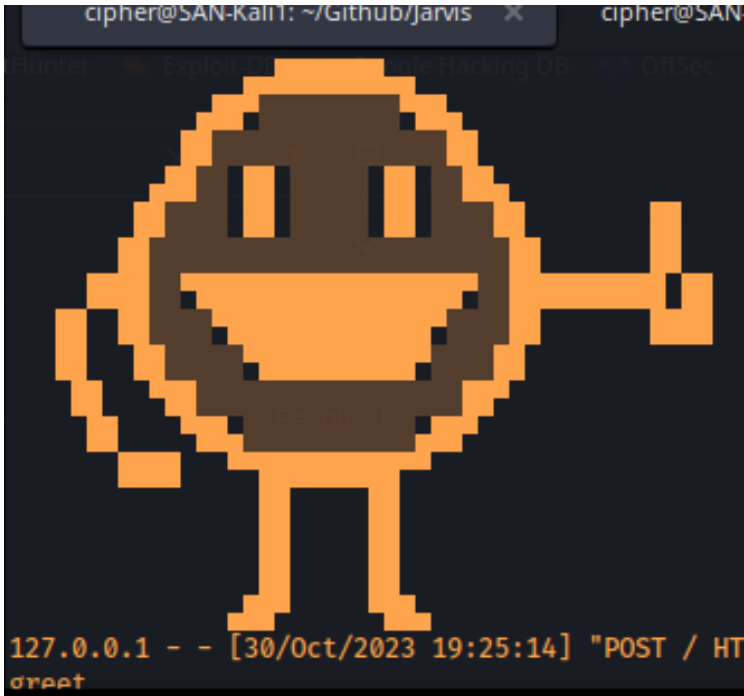
Here is Jarvis!

He listens, he speaks, he moves, and he blinks! Also he opens websites, pauses your music, talks through discord, and can be modded to do more.

13

Brief history of J.A.R.V.I.S.

When Sylvie killed He Who Remains, and the timeline began branching, Miss Minutes arrived in an alternate timeline where technology existed purely in the form of pixels!

Not really.

Jarvis was a project of mine which started in 2016, I found a lot of interest in Jarvis from Iron Man, and real life examples of Machine Learning, such as Cleverbot (Also known as Eviebot) – Using this newbie block language, I created my own virtual assistant, named 'Lyra' – who had many MP3 files uploaded into his code, and could respond to text inputs (just if and when chains)

Obviously, this interest never left me. As my AI assistant began improving, moving over to Python, to JavaScript, and then back to Python again, sometimes evolving into a speaking Windows command prompt, then later getting a menu screen, later integrating with Google's speech recognition API, starting to interact with the terminal to control the computer, and then evolving in the sense of using an actual Neural Network to determine the intent of the input.

I never made these codes public, as it was constantly breaking and was never fully efficient, until I rewrote it from the

ground-up and renamed it 'Jarvis' – after the iconic Jarvis of the Iron Man franchise, and gave him a slightly animated face to express what processes were being undertaken in the background.

Now, Jarvis can control Discord servers, control Linux and MacOS devices, open websites, run silently on low-power systems, and back himself up to Github to protect his own code, as well as maintain whatever conversation you educate it on, and also utilise the Janex: NLG library to attempt at breaking free from the conversational confines of the intent classifier.