Voice Commands with WordNet (VCW) Manual

Baron Khan

June 6, 2018

Contents

1 Introduction

Voice Commands with WordNet (VCW) is a Java library for processing user commands, and can be used to add voice commands to an application.[1] VCW takes as input the user's utterance (as a string) and maps it to an action within the application. All processing is near-instantaneous on modern processors (including phones and tablets) and does not require an Internet connection. VCW can be used to add voice commands to video games, embedded systems, or any piece of software.

VCW uses Princeton University's WordNet database to find relationships between words, in order to map countless variations of commands to actions within the application without having to hard-code any phrases. VCW is compatible with any Java 8 project, as well as Android applications supporting Java 8.

[1] https://github.com/BaronKhan/voice-commands-with-wordnet

2 Getting Started with VCW

2.1 Building the Library

VCW can be built into a Java archive (JAR) using the `build.sh` Bash script in the source repository.[2] The source files can also be included directly in your project.

2.2 Adding Dependencies

As well as adding the VCW Java archive file to your project, the following dependencies are required for VCW to work, and need to be added separately to your Java project as dependencies:

- Commons Compress 1.15
- Java WordNet Interface (`https://projects.csail.mit.edu/jwi/`)
- Stanford POS Tagger (`https://nlp.stanford.edu/software/tagger.shtml#Download`)
- WS4J (`https://code.google.com/archive/p/ws4j/downloads`)

2.3 Loading the Models

VCW requires access to a local WordNet database and a Part-of-Speech (POS) tagging model. Please download the following files and include them in your project's working directory:

- `english-left3words-distsim.tagger` (`https://nlp.stanford.edu/software/tagger.shtml`)
- `wn3.1.dict/dict/` (`http://wordnetcode.princeton.edu/wn3.1.dict.tar.gz`)

2.4 Speech-To-Text Engine

VCW **does not** transform the user's utterance from audio to a string. It simply processes the string and maps the intent to an action in the application. An external speech-to-text engine is required if the user's input is not already a string. There are many open-source APIs that do this, such as Google's Speech-to-Text API or PocketSphinx.

3 Using VCW

This section describes how to use the library in a Java project, and provides examples of adding commands for use in a video game. Once finished, the system should be able to map the following commands to an intent for attacking a sword without having to hard-code any phrases:

- ”attack with a sword”
- ”hit with something sharp”
- ”use a blade to fight”

[2] https://github.com/BaronKhan/voice-commands-with-wordnet/tree/master/build

- ”launch an assault using the sword”
- ”obliterate the enemy with a pointy weapon”
- and much more...

3.1 Setting Up the Library

Before using any methods in the library, load the POS tagging model and WordNet database (as a URL).

```
VoiceProcess.loadTagger("english-left3words-distsim.tagger");   // Load POS tagging model
private URL url = new URL("file", null, "wn3.1.dict/dict/");    // Load WordNet database
```

3.2 Entities

VCW assumes that every user intent specifies the action to be performed, and optionally the target and the context of the action. For example, in the phrase, ”attack the enemy with a sword”, the action is ”attack”, while the target is ”enemy” and the context is ”sword”.

In VCW, an `Entity` is anything that can be either a target or a context. Java classes should inherit from the `Entity` class, and call its constructor using `super()`, which takes as input the name of the entity. Entities are also grouped into context types using the `setContext()` method. Listing 1 shows an example of a sword entity that will be used.

Listing 1: Example of a Sword Entity

```
public class Sword extends Entity {
    public Sword() {
        super("sword");
        setContext("weapon");
    }
}
```

Here, since a sword is a weapon, its context type is set to ”weapon”. The context type is used to index the table in a `ContextActionMap`.

3.3 Creating a Context-Action Map

VCW’s system for creating voice commands is based on mapping actions and contexts (i.e. entities for which the action is applied with) to methods that will be executed.

Create a `ContextActionMap` that contains a table with each cell containing a an instance of `Action`. The table is indexed using actions along the top (as strings) and context types along the left. Table 1 shows an example of this for a game.

Table 1: A table mapping actions to context types

	attack	**heal**	**move**
default	Attack	Heal	Move
weapon	AtkWithWeapon		
potion		HealWithPotion	

This table contains the actions, ”attack”, ”heal” and ”move”, and the context types, ”weapon” and ”potion”. For example, if the user executes an ”attack” action with the context type being ”weapon” (e.g. if they say, ”attack with a sword”), then the `AtkWithWeapon` Action class will be invoked (see section 3.5 for more on `Actions`). If no context is specified, the ”default” action is invoked (e.g. `Attack`). Blank cells indicate that the action with that context type is not compatible (for example, you cannot ”attack” with a ”potion”).

A `ContextActionMap` Java class can easily be generated from a CSV file using the included Python script, `generateTable.py`. Simply having Table 1 in a CSV file and running the following command:

```
python generateTable.py game-table.csv GameContextActionMap
```

will generate `GameContextActionMap.java` which you can add to your project, as in Listing 2.

Listing 2: Example of a ContextActionMap

```java
public class GameContextActionMap extends ContextActionMap {
    public GameContextActionMap(GlobalState state) {
        super(state);
        setActionList(               "attack",            "heal",               "move");
        addDefaultContextActions(    new Attack(),        new Heal(),           new Move());
        addContextActions("weapon",  new AtkWithWeapon(), null,                 null);
        addContextActions("potion",  null,                new HealWithPotion(), null);
    }
}
```

3.4 Creating a GlobalState

A `GlobalState` is an object used for passing the state of the application around the system. The `GlobalState` should be able to access different objects and data that may be required to execute different actions. In this case, let’s create a `GameState` object that contains our game logic. If an action decreases the health of an enemy (e.g. attacking with a sword), then the enemy’s health should be accessible from this class.

```java
public class GameState extends GlobalState {
    //...
}
```

3.5 Creating Actions

An `Action` is a wrapper that is placed in the cells of a `ContextActionMap`, as shown in Listing 2 of Section 3.3. Each `Action` should override the `execute` method, which takes as input your `GlobalState` object (to access data and objects) and the current target if specified (otherwise this is set to the default target). Listing 3 shows an example of an `Action` for attacking with a weapon.

Listing 3: Example of an Action

```java
public class AtkWithWeapon extends Action {
    public String execute(GlobalState state, Entity currentTarget) {
        //Insert code to attack with weapon (e.g. decrease enemy health)
        //Access current context using Action.getCurrentContext()
        return "Return a response to the user.";
    }
}
```

3.6 Processing a Voice Command

Finally, the user's utterance can now be processed. In the `GlobalState` class (e.g. in this example, the `GameState` class), add a `VoiceProcess` class, and associate it to your `GlobalState` and `ContextActionMap`. Once the `VoiceProcess` class is created, the WordNet database (that was loaded in Section 3.1) must be added to it using the `addDictionary` method.

```java
public class GameState extends GlobalState {
    public GameState() {
        private ContextActionMap mMap = new GameContextActionMap(this);
        private VoiceProcess mVoiceProcess = new VoiceProcess(this, mMap);
        mCommandProcess.addDictionary(url);
    }
```

Next, the possible targets and contexts need to be added to the `ContextActionMap`. These possible targets/ need to be dynamically updated as the state changes (e.g. if the enemy is no longer present, it need to be removed)

```java
mMap.addPossibleContexts(Arrays.asList(new Sword(), new Potion()));
mMap.addPossibleTargets(Arrays.asList(new Enemy()));
```

A default target can also be set. This is the target that is used if no target is specified in the user's intent. It can be set using `mMap.setDefaultTarget(target)` method. You can also set the default target to the user, using `mMap.setDefaultTargetToSelf()`.

You can now process the user's utterance and return the text response (to be displayed to the user) as the action executes. If the input is not understood, then the string returned is, ”Intent not understood”, and no changes are made to the state.

```java
String input = "attack with a sword";
String output = mVoiceProcess.processInput(input);  //GameState is updated
System.out.println(output);
```

3.7 Adding More Commands

To add more commands to your application, simply create new `Action` wrappers and update the fields in the `ContextActionMap`.

3.8 More Examples

More examples can be found in the library's repository, in the `examples` directory.[3] The examples include adding voice commands to a game, a video conferencing application, and a cooking application.

[3]https://github.com/BaronKhan/voice-commands-with-wordnet/tree/master/examples