summarize_with_llama_cpp

Generated by Doxygen 1.9.4

1 Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

ArgumentParser
A simple command line argument parser 2

model_wrapper::Model
Model wrapper 9

ArgumentParser::Option
Represents a command line option with its properties and value 12

2 File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

3 Class Documentation

3.1 ArgumentParser Class Reference

A simple command line argument parser.

```
#include "argument_parser.h"
```

Classes

- struct Option

  Represents a command line option with its properties and value.

Public Member Functions

- ArgumentParser & add_flag (const std::string &name, const std::string &short_name, const std::string &description, bool is_mandatory)

  Add a flag option.
- template<typename T >
  ArgumentParser & add_option (const std::string &name, const std::string &short_name, const std::string &description, bool is_mandatory, T default_val=T{})

  Add an option that takes a value.
- ArgumentParser (std::string description="")

  Construct a new Argument Parser.
- template<typename T >
  T get_option (const std::string &name) const

  Get the value of an option.
- const std::vector< std::string > & get_positional () const

  Get positional arguments.
- bool is_provided (const std::string &name) const

  Check if an option was provided on the command line.
- void parse (int argc, char **argv)

  Parse command line arguments.
- void **print_help** () const

  Print help message to stdout.

Private Member Functions

- void collect_remaining_positional (std::size_t start_index, int argc, char **argv)

  Add all remaining arguments as positional.
- std::string generate_help () const

  Generates help text for command line usage.
- bool is_help_requested () const

  Check if help was requested.
- void process_arguments (int argc, char **argv)

  Process all command line arguments.
- void process_option (const std::string &name, int &current_index, int argc, char **argv)

  Process a single option from the command line.
- template<typename T >
  ArgumentParser & register_option (const std::string &name, const std::string &short_name, const std::string &description, bool is_flag, bool is_mandatory, T default_val)

  Internal helper method to register any type of option.
- void validate_option_name (const std::string &name, bool is_short=false) const

  Validate option names and check for duplicates.
- void validate_required_options () const

  Validate that all required options were provided.

Static Private Member Functions

- template<typename T >
  static std::any convert (const std::string &val)

  Converts a string argument to a typed value.

Private Attributes

- std::unordered_map< std::string, Option > **options**

  Maps long option names to their corresponding Option objects.
- std::vector< std::string > **positional**

  Stores positional command line arguments (not associated with any option)
- std::string **program_description**

  Description of the program to show in help message.
- std::string **program_name**

  Name of the program (extracted from argv[0])
- std::unordered_map< std::string, std::string > **short_to_long**

  Maps short option names to their corresponding long option names for quick lookup.

3.1.1 Detailed Description

A simple command line argument parser.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 ArgumentParser() `ArgumentParser::ArgumentParser ( std::string description = "" ) [explicit]`

Construct a new Argument Parser.

Parameters

description	Program description text

3.1.3 Member Function Documentation

3.1.3.1 add_flag() `ArgumentParser & ArgumentParser::add_flag (`

```
const std::string & name,
const std::string & short_name,
const std::string & description,
bool is_mandatory )
```

Add a flag option.

Parameters

name	Long option name
short_name	Short option name (single character)
description	Option description
is_mandatory	Whether the option is required

Returns

Reference to this parser for method chaining

3.1.3.2 add_option() `template<typename T >`

```
ArgumentParser & ArgumentParser::add_option (
const std::string & name,
const std::string & short_name,
const std::string & description,
bool is_mandatory,
T default_val = T{} )  [inline]
```

Add an option that takes a value.

Template Parameters

T	Type of the option value

Parameters

name	Long option name
short_name	Short option name (single character)
description	Option description

Parameters

is_mandatory	Whether the option is required
default_val	Default value if not provided

Returns

Reference to this parser for method chaining

3.1.3.3 collect_remaining_positional()

```
void ArgumentParser::collect_remaining_positional (
        std::size_t start_index,
        int argc,
        char ** argv )  [private]
```

Add all remaining arguments as positional.

Parameters

start_index	Starting index in argv
argc	Argument count
argv	Argument array

3.1.3.4 convert()

```
template<typename T >
static std::any ArgumentParser::convert (
        const std::string & val )  [inline], [static], [private]
```

Converts a string argument to a typed value.

Template Parameters

T	The target type to convert to (string, bool, int, float, double)

Parameters

val	The string value to convert

Returns

std::any The converted value wrapped in std::any

Exceptions

std::runtime_error	If T is not one of the supported types
std::invalid_argument	If the string cannot be converted to the requested type
std::out_of_range	If the converted value would be out of range for the target type

3.1.3.5 generate_help() `std::string ArgumentParser::generate_help ( ) const [private]`

Generates help text for command line usage.

Returns

Formatted help string

Creates a help message showing program usage, description, and available options in a readable format

3.1.3.6 get_option() `template<typename T >`
`T ArgumentParser::get_option (`
`const std::string & name ) const [inline]`

Get the value of an option.

Template Parameters

T	Type to cast the option value to

Parameters

name	Option name

Returns

The option value

Exceptions

std::runtime_error	if option doesn't exist
std::bad_any_cast	if type doesn't match

3.1.3.7 get_positional() `const std::vector< std::string > & ArgumentParser::get_positional ( ) const`

Get positional arguments.

Returns

Vector of positional arguments

3.1.3.8 is_help_requested() `bool ArgumentParser::is_help_requested ( ) const [private]`

Check if help was requested.

Returns

true if help option was provided

3.1.3.9 is_provided() `bool ArgumentParser::is_provided ( const std::string & name ) const`

Check if an option was provided on the command line.

Parameters

name	Option name

Returns

true if the option was provided

3.1.3.10 parse() `void ArgumentParser::parse ( int argc, char ** argv )`

Parse command line arguments.

Parameters

argc	Argument count
argv	Argument array

Exceptions

std::runtime_error	for parsing errors

3.1.3.11 process_arguments() `void ArgumentParser::process_arguments ( int argc, char ** argv ) [private]`

Process all command line arguments.

Parameters

argc	Argument count
argv	Argument array

3.1.3.12 process_option()

```
void ArgumentParser::process_option (
    const std::string & name,
    int & current_index,
    int argc,
    char ** argv )  [private]
```

Process a single option from the command line.

Parameters

name	The long name of the option being processed
current_index	Current index in argv array, may be updated if option consumes a value
argc	Number of command line arguments
argv	Array of command line arguments

Exceptions

std::runtime_error	If the option is unknown or value parsing fails
std::runtime_error	If a non-flag option is provided without a value

Handles both flag options (which don't require a value) and value options (which consume the next argument as their value). For value options, the index cuurent_index is incremented to skip the value in the main parsing loop.

3.1.3.13 register_option()

```
template<typename T >
ArgumentParser & ArgumentParser::register_option (
    const std::string & name,
    const std::string & short_name,
    const std::string & description,
    bool is_flag,
    bool is_mandatory,
    T default_val )  [inline], [private]
```

Internal helper method to register any type of option.

Template Parameters

T	The type of the option value

Parameters

name	Long option name
short_name	Short option name

Parameters

description	Option description
is_flag	Whether this option is a flag (no value)
is_mandatory	Whether this option is required
default_val	Default value

Returns

Reference to this parser for method chaining

3.1.3.14 validate_option_name()

```
void ArgumentParser::validate_option_name (
        const std::string & name,
        bool is_short = false ) const  [private]
```

Validate option names and check for duplicates.

Parameters

name	The option name to validate
is_short	Whether this is a short option name

Exceptions

std::invalid_argument	if the name is invalid or already exists

3.1.3.15 validate_required_options()

```
void ArgumentParser::validate_required_options ( ) const
[private]
```

Validate that all required options were provided.

Exceptions

std::runtime_error	if any mandatory option is missing

The documentation for this class was generated from the following file:

- include/argument_parser.h

3.2 model_wrapper::Model Class Reference

Model wrapper.

```
#include "model.h"
```

Public Member Functions

- void generate_response (const std::string &prompt, std::ostream &out)

  Generate a responde from the prompt.
- std::string get_formatted_prompt (const std::vector< llama_chat_message > &messages)

  Apply the chat template to the messages.
- Model (const std::string_view model_path, const float temperature, const int32_t number_of_gpu_layers, const std::size_t prediction_length)

  Construct a new Model object.

Private Member Functions

- llama_context_ptr create_context (const std::size_t number_of_tokens)

  Create the context.
- void **initialize_sampler** ()

  Initialize the sampler.
- std::vector< llama_token > tokenize_prompt (const std::string &prompt)

  Tokenize the prompt .

Private Attributes

- llama_model_ptr **m_model** {nullptr}
- const std::size_t **m_prediction_length**
- llama_sampler_ptr **m_sampler** {nullptr}
- const float **m_temperature**
- const llama_vocab * **m_vocab**

3.2.1 Detailed Description

Model wrapper.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 Model()

```
model_wrapper::Model::Model (
        const std::string_view model_path,
        const float temperature,
        const int32_t number_of_gpu_layers,
        const std::size_t prediction_length )
```

Construct a new Model object.

Parameters

model_path	The path to the model file in GGUF format
temperature	The temperature
number_of_gpu_layers	The number of GPU layers to use
prediction_length	The maximum number of tokens to predict

Exceptions

std::runtime_error	if the model cannot be loaded

3.2.3 Member Function Documentation

3.2.3.1 create_context() `llama_context_ptr model_wrapper::Model::create_context ( const std::size_t number_of_tokens ) [private]`

Create the context.

The context size is determined as number_of_tokens + m_prediction_length.

Parameters

number_of_tokens	The number of tokens

Returns

The context pointer

Exceptions

std::runtime_error	if the context cannot be created

3.2.3.2 generate_response() `void model_wrapper::Model::generate_response ( const std::string & prompt, std::ostream & out )`

Generate a responde from the prompt.

The model generates a response to the prompt by sampling tokens and this function writes to out each token

Parameters

prompt	The prompt to respond to
out	The output stream to write the response to

Exceptions

std::runtime_error	if the generation fails

3.2.3.3 get_formatted_prompt() `std::string model_wrapper::Model::get_formatted_prompt ( const std::vector< llama_chat_message > & messages )`

Apply the chat template to the messages.

Parameters

messages	The messages to format

Returns

The formatted prompt

Exceptions

std::runtime_error	if the chat template cannot be applied

3.2.3.4 tokenize_prompt() `std::vector< llama_token > model_wrapper::Model::tokenize_prompt ( const std::string & prompt ) [private]`

Tokenize the prompt .

Returns

The tokens

Exceptions

std::runtime_error	if the prompt cannot be tokenized

The documentation for this class was generated from the following file:

- include/model.h

3.3 ArgumentParser::Option Struct Reference

Represents a command line option with its properties and value.

Public Attributes

- std::function< std::any(const std::string &)> **converter**

Converter function to parse string argument into typed value.

- std::string **description**

  Description text shown in help message.

- bool **is_flag**

  True if option is a flag (doesn't take a value)

- bool **is_mandatory**

  True if option is required/mandatory.

- bool **is_provided** {false}

  True if option was provided on command line.

- std::string **name**

  Long name of the option without leading dashes.

- std::string **short_name**

  Short name of the option (single character) without leading dash.

- std::any **value**

  The option's value (type-erased)

3.3.1 Detailed Description

Represents a command line option with its properties and value.

The documentation for this struct was generated from the following file:

- include/argument_parser.h

4 File Documentation

4.1 argument_parser.h

```
1 ////////////////////////////////////////////////////////////////////////////////
2 // File:  argument_parser.h
3 //
4 // License:  MIT
5 //
6 // Copyright (C) 2025 Onur Ozuduru
7 //
8 // Follow Me!
9 //   github:  github.com/onurozuduru
10 ////////////////////////////////////////////////////////////////////////////////
11
12 #include <any>
13 #include <functional>
14 #include <stdexcept>
15 #include <string>
16 #include <type_traits>
17 #include <unordered_map>
18 #include <vector>
19
20 /**
21 * \brief A simple command line argument parser
22 */
23 class ArgumentParser {
24 private:
25   /**
26 * \brief Represents a command line option with its properties and value
27 */
28   struct Option {
29     /**
30 * \brief Long name of the option without leading dashes
31 */
32     std::string name;
33
34     /** \brief Short name of the option (single character) without leading dash
35 */
```

```
36      std::string short_name;
37
38      /**
39   * \brief Description text shown in help message
40   */
41      std::string description;
42
43      /**
44   * \brief True if option is a flag (doesn't take a value)
45   */
46      bool is_flag;
47
48      /**
49   * \brief True if option is required/mandatory
50   */
51      bool is_mandatory;
52
53      /**
54   * \brief The option's value (type-erased)
55   */
56      std::any value;
57
58      /**
59   * \brief Converter function to parse string argument into typed value
60   */
61      std::function<std::any(const std::string &)> converter;
62
63      /**
64   * \brief True if option was provided on command line
65   */
66      bool is_provided{false};
67    };
68
69    /**
70   * \brief Maps long option names to their corresponding Option objects
71   */
72    std::unordered_map<std::string, Option> options;
73
74    /**
75   * \brief Maps short option names to their corresponding long option names
76   * for quick lookup
77   */
78    std::unordered_map<std::string, std::string> short_to_long;
79
80    /**
81   * \brief Stores positional command line arguments (not associated with any
82   * option)
83   */
84    std::vector<std::string> positional;
85
86    /**
87   * \brief Name of the program (extracted from argv[0])
88   */
89    std::string program_name;
90
91    /**
92   * \brief Description of the program to show in help message
93   */
94    std::string program_description;
95
96    /**
97   * \brief Converts a string argument to a typed value
98   *
99   * \tparam T The target type to convert to (string, bool, int, float, double)
100  * \param val The string value to convert
101  * \return std::any The converted value wrapped in std::any
102  * \throws std::runtime_error If T is not one of the supported types
103  * \throws std::invalid_argument If the string cannot be converted to the
104  * requested type
105  * \throws std::out_of_range If the converted value would be out of range for
106  * the target type
107  */
108   template <typename T> static std::any convert(const std::string &val) {
109     if constexpr (std::is_same_v<T, std::string>) {
110       return val;
111     } else if constexpr (std::is_same_v<T, bool>) {
112       return (val == "true" || val == "1" || val == "yes");
113     } else if constexpr (std::is_same_v<T, int>) {
114       return std::stoi(val);
115     } else if constexpr (std::is_same_v<T, float>) {
116       return std::stof(val);
117     } else if constexpr (std::is_same_v<T, double>) {
118       return std::stod(val);
119     } else {
120       throw std::runtime_error("Unsupported type conversion");
121     }
122   }
```

```
123
124   /**
125 * \brief Process a single option from the command line
126 *
127 * \param name The long name of the option being processed
128 * \param current_index Current index in argv array, may be updated if option
129 * consumes a value
130 * \param argc Number of command line arguments
131 * \param argv Array of command line arguments
132 *
133 * \throws std::runtime_error If the option is unknown or value parsing fails
134 * \throws std::runtime_error If a non-flag option is provided without a value
135 *
136 * \details Handles both flag options (which don't require a value) and value
137 * options (which consume the next argument as their value).  For value
138 * options, the index cuurent_index is incremented to skip the value in the
139 * main parsing loop.
140 */
141   void process_option(const std::string &name, int &current_index, int argc,
142                       char **argv);
143
144   /**
145 * \brief Generates help text for command line usage
146 * \return Formatted help string
147 *
148 * \details Creates a help message showing program usage, description,
149 *          and available options in a readable format
150 */
151   std::string generate_help() const;
152
153   /**
154 * \brief Validate option names and check for duplicates
155 * \param name The option name to validate
156 * \param is_short Whether this is a short option name
157 * \throws std::invalid_argument if the name is invalid or already exists
158 */
159   void validate_option_name(const std::string &name,
160                             bool is_short = false) const;
161
162   /**
163 * \brief Internal helper method to register any type of option
164 *
165 * \tparam T The type of the option value
166 * \param name Long option name
167 * \param short_name Short option name
168 * \param description Option description
169 * \param is_flag Whether this option is a flag (no value)
170 * \param is_mandatory Whether this option is required
171 * \param default_val Default value
172 * \return Reference to this parser for method chaining
173 */
174   template <typename T>
175   ArgumentParser &register_option(const std::string &name,
176                                   const std::string &short_name,
177                                   const std::string &description, bool is_flag,
178                                   bool is_mandatory, T default_val) {
179     validate_option_name(name);
180     if (!short_name.empty()) {
181       validate_option_name(short_name, true);
182     }
183
184     Option opt{name,         short_name,  description, is_flag,
185                is_mandatory, default_val, convert<T>};
186     options[name] = opt;
187     if (!short_name.empty()) {
188       short_to_long[short_name] = name;
189     }
190     return *this;
191   }
192
193   /**
194 * \brief Process all command line arguments
195 * \param argc Argument count
196 * \param argv Argument array
197 */
198   void process_arguments(int argc, char **argv);
199
200   /**
201 * \brief Add all remaining arguments as positional
202 * \param start_index Starting index in argv
203 * \param argc Argument count
204 * \param argv Argument array
205 */
206   void collect_remaining_positional(std::size_t start_index, int argc,
207                                     char **argv);
208
209   /**
```

```
210 * \brief Check if help was requested
211 * \return true if help option was provided
212 */
213   bool is_help_requested() const;
214
215   /**
216 * \brief Validate that all required options were provided
217 * \throws std::runtime_error if any mandatory option is missing
218 */
219   void validate_required_options() const;
220
221 public:
222   /**
223 * \brief Construct a new Argument Parser
224 * \param description Program description text
225 */
226   explicit ArgumentParser(std::string description = "");
227
228   /**
229 * \brief Add an option that takes a value
230 * \tparam T Type of the option value
231 * \param name Long option name
232 * \param short_name Short option name (single character)
233 * \param description Option description
234 * \param is_mandatory Whether the option is required
235 * \param default_val Default value if not provided
236 * \return Reference to this parser for method chaining
237 */
238   template <typename T>
239   ArgumentParser &add_option(const std::string &name,
240                              const std::string &short_name,
241                              const std::string &description, bool is_mandatory,
242                              T default_val = T{}) {
243     return register_option<T>(name, short_name, description, false,
244                               is_mandatory, default_val);
245   }
246
247   /**
248 * \brief Add a flag option
249 * \param name Long option name
250 * \param short_name Short option name (single character)
251 * \param description Option description
252 * \param is_mandatory Whether the option is required
253 * \return Reference to this parser for method chaining
254 */
255   ArgumentParser &add_flag(const std::string &name,
256                            const std::string &short_name,
257                            const std::string &description, bool is_mandatory);
258
259   /**
260 * \brief Parse command line arguments
261 * \param argc Argument count
262 * \param argv Argument array
263 * \throw std::runtime_error for parsing errors
264 */
265   void parse(int argc, char **argv);
266
267   /**
268 * \brief Get the value of an option
269 * \tparam T Type to cast the option value to
270 * \param name Option name
271 * \return The option value
272 * \throw std::runtime_error if option doesn't exist
273 * \throw std::bad_any_cast if type doesn't match
274 */
275   template <typename T> T get_option(const std::string &name) const {
276     if (!options.contains(name)) {
277       throw std::runtime_error("Option not registered:  " + name);
278     }
279
280     return std::any_cast<T>(options.at(name).value);
281   }
282
283   /**
284 * \brief Check if an option was provided on the command line
285 * \param name Option name
286 * \return true if the option was provided
287 */
288   bool is_provided(const std::string &name) const;
289
290   /**
291 * \brief Get positional arguments
292 * \return Vector of positional arguments
293 */
294   const std::vector<std::string> &get_positional() const;
295
296   /**
```

```
297 * \brief Print help message to stdout
298 */
299   void print_help() const;
300 };
```

4.2 model.h

```
1 ////////////////////////////////////////////////////////////////////////////////
2 // File:  model.h
3 //
4 // License:  MIT
5 //
6 // Copyright (C) 2025 Onur Ozuduru
7 //
8 // Follow Me!
9 //   github:  github.com/onurozuduru
10 ////////////////////////////////////////////////////////////////////////////////
11
12 #include "llama-cpp.h"
13 #include <ostream>
14 #include <string>
15 #include <vector>
16
17 namespace model_wrapper {
18 /**
19 * \brief Model wrapper
20 */
21 class Model {
22 private:
23   const float m_temperature;
24   const std::size_t m_prediction_length;
25
26   llama_model_ptr m_model{nullptr};
27   const llama_vocab *m_vocab;
28   llama_sampler_ptr m_sampler{nullptr};
29
30   /**
31 * \brief Tokenize the prompt
32 * \p
33 * \return The tokens
34 * \throw std::runtime_error if the prompt cannot be tokenized
35 */
36   std::vector<llama_token> tokenize_prompt(const std::string &prompt);
37
38   /**
39 * \brief Initialize the sampler
40 */
41   void initialize_sampler();
42
43   /**
44 * \brief Create the context.
45 * \details The context size is determined as
46 * number_of_tokens + m_prediction_length.
47 * \param number_of_tokens The number of tokens
48 * \return The context pointer
49 * \throw std::runtime_error if the context cannot be created
50 */
51   llama_context_ptr create_context(const std::size_t number_of_tokens);
52
53 public:
54   /**
55 * \brief Construct a new Model object
56 * \param model_path The path to the model file in GGUF format
57 * \param temperature The temperature
58 * \param number_of_gpu_layers The number of GPU layers to use
59 * \param prediction_length The maximum number of tokens to predict
60 * \throw std::runtime_error if the model cannot be loaded
61 */
62   Model(const std::string_view model_path, const float temperature,
63         const int32_t number_of_gpu_layers,
64         const std::size_t prediction_length);
65
66   /**
67 * \brief Apply the chat template to the messages
68 * \param messages The messages to format
69 * \return The formatted prompt
70 * \throw std::runtime_error if the chat template cannot be applied
71 */
72   std::string
73   get_formatted_prompt(const std::vector<llama_chat_message> &messages);
74
75   /**
76 * \brief Generate a responde from the prompt
77 * \details The model generates a response to the prompt by sampling tokens
```

```
78 * and this function writes to out each token
79 * \param prompt The prompt to respond to
80 * \param out The output stream to write the response to
81 * \throw std::runtime_error if the generation fails
82 */
83   void generate_response(const std::string &prompt, std::ostream &out);
84 };
85 } // namespace model_wrapper
```

Index