Product Spec: Multi-provider Completions

Description

This document describes a generalized framework to combine completions from multiple providers into a cohesive set. The framework is provider-agnostic – It does not rely on any knowledge of the constituent providers to function. Therefore, it should be straightforward to add or remove providers at any time.

Design Principles

- Completions should be ordered by the expected values that they would provide the user.
- Low value or duplicative completions should be removed to reduce the amount of information that the user needs to parse.
- Completions should be mixed in a generalizable and objective manner so that hooking up new providers in the future will be straightforward.

Completions Providers

The table below outlines the providers that can suggest completions for the user.

	A	B	C	D
1	**Provider**	**Example**	**# Tokens**	**Score**
2	Traditional	`import j$` → `import json`	Single	Frequency
3	Keywords	`im$` → `import`	Single	Probability
4	Names	`f$` → `foo`	Single	Frequency
5	Attributes	`foo.b$` → `foo.bar`	Single	Frequency
6	Import alias	`import num$` → `import numpy as np`	Multi	N/A
7	Call model	`foo.bar($)` → `foo.bar(baz)`	Multi	Confidence
8	General expressions	`fo$` → `foo.bar(baz)`	Multi	Confidence
9	Call patterns	`foo.bar($)` → `foo.bar([arg])`	Multi	Frequency
10	Arg specs	`foo.bar($)` → `foo.bar(kwarg=[int])`	Multi	Frequency
11	`dict` keys	`my_dict[$]` → `my_dict['key']`	Single	Alphabetical
12	Lexical	`de$` → `def foo(bar):`	Multi	Expected value

This table is provided purely for informational purposes. The framework described in the rest of the document does not rely on the implementation details of any individual provider.

Phases

The mixing process consists of a number of discrete phases. Each phase can be thought of as a function that takes in a completions set as input and returns a potentially modified output set. The phases must be executed in the order in which they are defined in this document.

Phase 1: Scoring

In this phase, we assign scores to each completion to be used in later phases. This is done so that we can compare completions from different providers objectively, which is necessary since any given provider may use a unique internal score that does not translate over to other providers.

We will refer to the score assigned in this phase as the ***utility score***. The utility score being used in production is calculated using a linear model and is roughly a measure of how likely the completion is to be selected. The factors that go into this model are:

- *[Todo]*

Phase 2: Sorting

In the sorting phase, we reorder the completions according to the scores computed in ⚓Phase 1: Scoring. There is no tiebreaking logic if there are multiple completions with the same score.

> ***Note:*** *We had previously "grouped" completions such that we sorted completions based on parent-child relationships as well but no longer do this because grouping resulted in a lower top-5 recall. (⚓[1] Top-5 recall from different...)*

Phase 3: Filtering

The filtering phase is responsible for paring down the completions set to increase the informational density of the returned set by removing low value and redundant completions.

There are two types of filters that are applied in this phase. ***Individual filters*** operate on single completions and can be applied one-by-one. ***Group filters*** operate on entire completions groups to determine if there are any redundancies that can be removed within the group.

Individual Filters

- *[Not implemented yet]* Remove semantically invalid completions
- *[Not implemented yet]* Remove low score completions

Group Filters

- Deduplicate completions where the insert text is the same
  - In this situation, just keep the highest ranked completion
- Deduplicate completions where the display text is the same
  - When choosing between lexical completions and snippets, keep the one that is ranked the highest and discard all others (⚓[2] Discussion in GitHub about...)
- Deduplicate completions that differ by a trailing whitespace
  - When choosing between lexical completions and keywords, keep the keyword completions (⚓[2] Discussion in GitHub about...)
- *[Not implemented yet]* Proposal for removing near duplicates
  - For each group, compare each child to the parent and remove the child if it's a near duplicate
  - The parent can never be removed, in accordance with the rule above about not changing the structure of the completions tree
    - This means that when comparing `with` and `with open`, the only completion that can be removed is

`with open`

- Remove the child if the marginal increase in value is below a certain amount
  - Example: expected number of tokens added *conditional that the parent is correct* is < 0.5
  - Pr(`with`) = 0.9, Pr(`with ^ open`) = 0.63 → Pr(`open | with`) = 0.7 → 0.7 is also the expected number of tokens, so we can keep `with open` as a completion

Phase 4: Rendering

In the rendering phase, we potentially manipulate the completions' display texts to make it more human-readable.

- For name completions that are names of functions, we convert `func_name` → `func_name(...)` or `func_name()` if there are no arguments.

- [1] Top-5 recall from different ranking strategies
- [2] Discussion in GitHub about duplicate completions