Probase: A Probabilistic Knowledgebase

Version 0.10

Contents

Chapter 1

Knowledgebase and Our Mental World

1.1 Introduction

Our goal is to open the mental world of human beings to machines. To this end, we create a knowledgebase, called Probase. We hope that by injecting certain "general knowledge" into computing machines can gain a better understanding of human communication.

An important question is, what does the word "understand" mean here? Consider the following example. For human beings, when we see "25 Oct 1881", we recognize it as a date, although most of us do not know what it is about. However, if we are given a little more context, say the date is embedded in the following piece of short text "Pablo Picasso, 25 Oct 1881, Spain", most of us would have guessed (correctly) that the date represents Pablo Picasso's birthday. We are able to do this because we possess certain knowledge, and in this case, "one of the most important dates associated with a person is his birthday."

As another example, consider a problem in natural language processing. Humans do not find sentences such as "animals other than dogs such as cats" ambiguous, but machine parsing can lead to two possible understandings: "cats are animals" or "cats are dogs." Common sense tells us that cats cannot be dogs, which renders the second parsing improbable.

It turns out what we need in order to act like a human in the above two examples is nothing more than knowledge about concepts (e.g., persons and animals) and the ability to conceptualize (e.g., cats are animals). This is not a coincidence. Psychologist Gregory Murphy began his highly acclaimed book with the statement *"Concepts are the glue that holds our mental world together"* [8]. Nature magazine book review pointed out *"Without concepts, there would be no mental world in the first place"* [2]. Doubtless to say, having concepts and the ability to conceptualize is one of the defining characteristics of humanity. The question is then: How do we pass human concepts to machines, and how do we enable machines to conceptualize?

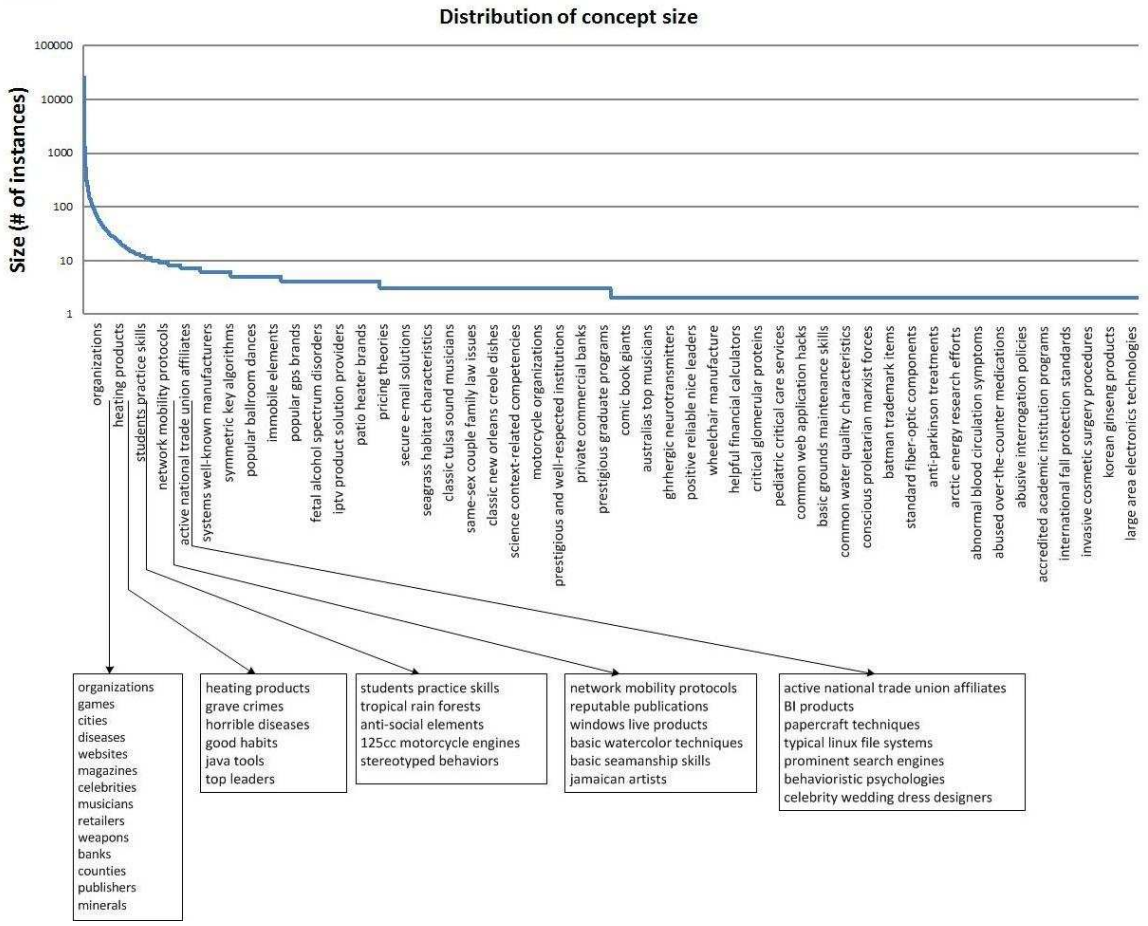

Figure 1.1: Distribution of the millions of concepts in Probase

1.2 The Probase Taxonomy

We build Probase [14, 10], a knowledgebase harnessed from billions of web pages.

Figure 1.2 is a snippet of Probase, which consists of concepts (e.g. emerging markets), instances (e.g., China), attributes and values (e.g., China's population is 1.3 billion), and relationships (e.g., emerging markets, as a concept, is closely related to newly industrialized countries), all of which are automatically derived in an unsupervised manner.

Probase is unique because it has a huge number of concepts and also a probabilistic nature. Our mental world contains many concepts about worldly facts, and Probase tries to duplicate them. The core taxonomy of Probase alone contains above 2.7 million concepts. Figure 1.1 shows their distribution. The Y axis is the number of instances each concept contains (logarithmic scale), and on the X axis are the 2.7 million concepts ordered by their size. In contrast, existing knowledgebases have far fewer concepts (Freebase [3] contains no more than 2,000 concepts, and Cyc [7] has about 120,000 concepts), which fall short of modeling our mental world. As we can see in Figure 1.1, besides popular concepts such

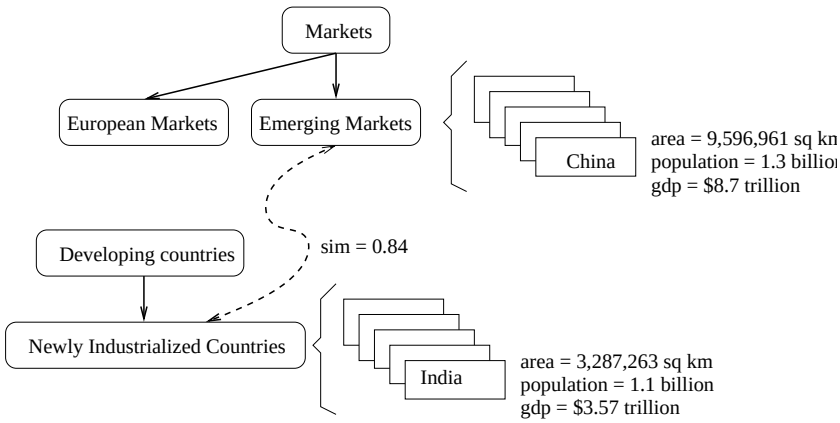

Figure 1.2: A Snippet of Probase's core taxonomy

as "cities" and "musicians", which are included by almost every general purpose taxonomy, Probase has millions of long tail concepts such as "anti-parkinson treatments", "celebrity wedding dress designers" and "basic watercolor techniques", which cannot be found in Freebase or Cyc. Besides concepts, Probase also has a large data space (each concept contains a set of instances or sub-concepts), a large attribute space (each concept is described by a set of attributes), and a large relationship space (e.g.,"locatedIn", "friendOf", "mayorOf", as well as relationships that are not easily named, such as the relationship between *apple* and *Newton*.)

We make a bold claim that Probase is a knowledgebase about concepts in our mental world because the concepts in Probase are harnessed from billions of web pages authored by millions of people (see Section 3). With such a rich concept space, Probase has much better chance to understand text in natural language (see Section 4). Indeed, we studied 2 years' worth of Microsoft's Bing search log, and found that 85% of the searches contain concepts and/or instances that exist in Probase. It means Probase can be a powerful tool to interpret user intention behind search [13, 11].

Another feature of Probase is that it is probabilistic, which means every claim in Probase is associated with some probabilities that model the claim's correctness, typicality, ambiguity, and other characteristics. The probabilities are derived from evidences found in web data, search log data, and other existing taxonomies. For example, for typicality (between concepts and instances), Probase contains the following probabilities:

- $P(C = company|I = apple)$: How likely people will think of the concept "company" when they see the word "apple".
- $P(I = steve\ jobs|C = ceo)$: How likely "steve jobs" comes into mind when people think about the concept "ceo".

Probase also has typicality scores for concepts and attributes. Another impor-

tant score in Probase is the similarity between any two concepts y_1 and y_2 (e.g., *celebrity* and *famous politicians*), which is defined as:

$$sim(y_1, y_2) = \lambda \cdot S_e(y_1, y_2) + (1 - \lambda) \cdot S_a(y_1, y_2)$$

where $\lambda \in [0, 1]$, $S_e(y_1, y_2)$ measures the similarity between the instances of y_1 and y_2, and $S_a(y_1, y_2)$ measures the similarity between the properties (attributes) of y_1 and y_2. Thus Probase can tell that *natural disasters* and *politicians* are very different concepts, *endangered species* and *tropical rainforest plants* have certain relationships, while *countries* and *nations* are almost the same concepts.

As we will see in Section 4, these probabilities serve as priors and likelihoods for Bayesian reasoning on top of Probase. In addition, the probabilistic nature of Probase also enables it to incorporate data of varied quality from heterogeneous sources. Probase regards external data as evidences, which are used to update Probase's existing beliefs. This is accomplished by the probabilistic reasoning and integration layer.

1.3 Building Probase

The Probase taxonomy is constructed in the following steps.

- We use an iterative learning approach to construct the core taxonomy [14], which consists of the isA relationship, from a web corpus of 1.68 billion web pages. The core taxonomy has 2.7 million concepts and 16 million instances.
- We find properties (attributes) for concepts [11] from the web. For every triple (subject, attribute, value), we also find, automatically, what questions the triple can be used to answer. For example, (China, population, 1.3 billion) can be used to answer the question "How many people live in China?", although the word "population" does not appear in the question.
- We use an unsupervised bootstrapping algorithm [12] to extract a massive amount of relationships among millions of instances simultaneously by repeatedly scanning the web corpus.
- We use a probabilistic data integration mechanism [6] to incorporate existing structured data, such as Freebase, IMDB, Amazon, etc. into Probase.

We briefly describe the work of constructing the core taxonomy, which consists of the 2.7 million concepts and the isA relationships among them. The isA relationships are harvested by using the so called Hearst linguistic patterns [5]. For example, a sentence that contains "... artists such as Pablo Picasso ..." can be considered as an evidence for the claim that *artist* is a hypernym of *Pablo Picasso*. Much work [4, 9] has used Hearst patterns to obtain isA patterns, but most focus

on a much limited concept space. For example, they do not differentiate the concept of *basic watercolor techniques* and the concept of *techniques*, which means on the one hand, a lot of concrete and useful information is lost, and on the other hand, some big concepts (such as *techniques*) are too vague to be useful.

Because of the irregularities and idiosyncrasies in natural languages, finding hyponyms and hypernyms are not always straight-forward. The example we have given before, *... **animals** other than dogs such as cats ...* can lead to two possible interpretations: (cats isA dogs) and (cats isA animals). Syntactically, both interpretations are correct. Unfortunately, many sentences lead to confusion, for example, in *... representatives in North America, Europe, the Middle East, Australia, Mexico, Brazil, Japan, and other **countries**...*, although *Australia* and *Middle East* appear side-by-side, one is a country but the other is not. These indicate that it is sometimes impossible to correctly identify hypernym-hyponym relationships if the parser does not have certain knowledge or common sense to begin with.

In our approach, we combine knowledge acquisition and knowledge serving: as we accumulate knowledge in the extraction process, we also use the knowledge we already extracted to improve the quality of extraction.

Specifically, we use a bootstrapping process to find hypernym-hyponym pairs. Let Γ be the set of hypernym-hyponym pairs that we have discovered. For each $(y \texttt{ isA } x) \in \Gamma$, we also keep a count $n(y \texttt{ isA } x)$, which indicates how many times we have seen $(y \texttt{ isA } x)$ in the text. Initially, Γ is empty. Then, we enlarge Γ by adding newly discovered pairs for which we are sure about their correctness. As we search for hypernym-hyponym pairs, we use Γ to help identify valid ones. Since Γ is being expanded in the process, our power to identify more valid pairs also increases, and more pairs will be added into Γ. The process terminates when Γ cannot be further enlarged.

It is important to understand that having a pair $(y \texttt{ isA } x)$ in Γ does not mean it is *true* that x and y have a hypernym-hyponym relationship. All it means is that we have certain evidences for such a claim, although the evidence itself might be wrong or we might have interpreted the evidence incorrectly. For example, for sentence *... **animals** other than dogs such as cats ...*, we first identify two possible hypernyms $\{animals, dogs\}$ in syntactic extraction. Next, we use Γ to help us decide which one is correct. There are at least four cases:

1. Γ does not yet contain any information about animals, dogs, cats, etc;
2. We have $(cats \texttt{ isA } animals) \in \Gamma$;
3. We have $(cats \texttt{ isA } dogs) \in \Gamma$;
4. both 2) and 3) are true.

One question is, how can 3) be true? The answer is that the data is noisy. There may exist a well-formed sentence *Dogs such as cats ...* that poses no ambiguity, which adds the claim $(cats \texttt{ isA } dogs)$ into Γ.

Clearly, for the sentence at hand, we cannot reject the claim $(cats \texttt{ isA } dogs)$ immediately, even if the other possible interpretation $(cats \texttt{ isA } animals)$ is already in Γ. If we do so, there is a possibility that we would have rejected the more probable interpretation of $(dogs \texttt{ isA } animals)$ if we have but once encountered an erroneous statement *Dogs such as cats* before. As we mentioned, the fact that a pair $(y \texttt{ isA } x)$ is in Γ, does not necessarily mean it is true that x and y have the hypernym-hyponym relationship. It only means we have encountered certain evidence for that claim, and the strength of the evidence is represented by $n(y \texttt{ isA } x)$. We use these information to devise a Bayesian learning approach for hypernym detection [14].

1.4 Probase for Conceptualization

In this section, we describe applications that use Probase to understand text in natural languages.

Short Text Conceptualization

Understandign short text (e.g., web search, tweets, anchor texts) is important to many applications. Much work has been devoted to topic discovery from text. Statistical approaches such as topic models [1] treat text as a bag of words in vector space, and discover "latent topics" from the text. But finding latent topics is not tantamount to understanding. A latent topic is represented by a set of words, consisting of no explicit concepts. Furthermore, for short texts that do not contain much statistics or signals, bag-of-words approaches typically do not work.

Probase enables machines to conceptualize from a set of words by performing Bayesian analysis based on the typicality and other probabilities described in Section 2. For example, given the word "India," the machine will form concepts such as *country* or *region*. Given two words, "India" and "China," the top concepts may shift to *Asian country* or *developing country*, etc. Given yet another word, "Brazil," the top concepts may change to *BRIC* or *emerging market*, etc. Using the same mechanism, the top concept formed by the machine when given the word "office" is *buildings*. When given one more word, say, "Xbox", the top concept becomes *Microsoft products*. Besides generalizing from instances to concepts, machine can also form concepts from descriptions. For example, given words "body," "smell" and "color," the concept of *wine* comes into the system. Certainly, instances and descriptions may mix, for example, we conceptualize {"apple," "headquarter"} to *company*, but {"apple," "smell," "color"} to *fruit*.

We cluster twitter messages based on conceptual signals provided by Probase [10]. The results outperform all existing approaches.

Understanding Web Tables

There are billions of tables on the Web, and they contain much valuable information. Tables are also relatively well structured, which means they are easier to understand than text in natural languages. With the help of Probase, we are able to unlock the information in such tables, and the information, once understood, is used to enrich Probase [11].

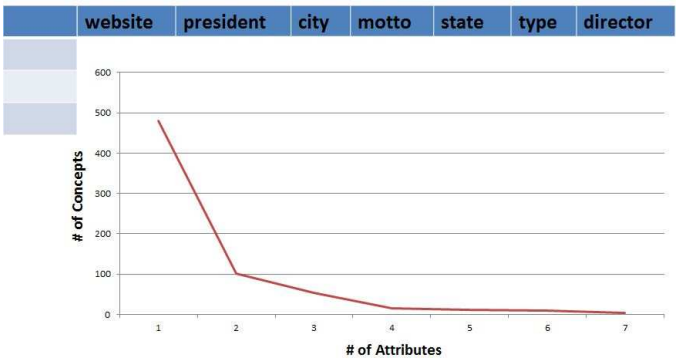

Figure 1.3: How Probase Understands a Table

Given a table, such as the one shows in Figure 1.3 which contains 4 rows, and the top row contains columns with names {*website, president, city, motto, state, type, director*}, how does Probase understand what it is about?

The way Probase understands a table is similar to how humans understand it. When given the name of the first column, *website*, the machine forms about 500 concepts. However, as it sees more and more concepts, the number of valid concepts falls sharply. After seeing all the names, Probase is pretty sure that the table is about *universities, institutes* or *schools*. In fact, the first column of the table, which is hidden in Figure 1.3, contains names such as *Stanford, ULCA, Berkeley*, which can also be conceptualize into *universities*. Thus, the process of understanding a table is a two dimensional short text conceptualization one. If the two resulting sets of concepts converge, then the machine can be pretty sure that it gets it right.

Web Search

As we have mentioned, 85% of Web searches contain concepts and/or instances that can be found in Probase. This gives Probase a good advantage to interpret the intention of the user. consider the following search queries:

1. ACM fellows working on semantic web
2. database conferences in asian cities
3. highest mountain in US

4. winter vacation destinations except florida

The user intention of each of the above queries is clear. However, current search engines cannot deliver good results as they find exact, word-for-word matches for phrases such as "database conferences", "asian cities", "ACM fellows", "mountains in US". Furthermore, they do not know that in order to find the "highest mountain", all we need to do is to apply the max aggregate on the *elevation* or *altitute* attribute of the mountain concept, and that "except florida" means the other 49 states in the US.

Probase, on the other hand, not only interprets the query correctly, but is able to rewrite the query and answer the query. Take the query "ACM fellows working on semantic web" for example. In Probase, "ACM fellows" is a class, and it contains a list of people who are ACM fellows. Combining with other keywords "working on semantic web", Probase can rewrite the query into multiple queries and then merge their results. This, however, introduces a new challenge: there are hundreds of ACM fellows. The problem is even more severe for query "database conferences in asian cities" because there are two concepts "database conferences" and "asian cities", which means it may produce hundreds or even thousands of combinations. To tackle these problems, we build a word association index, which provides the frequency of co-occurrence for every pair of keywords/instances. For example, we know *VLDB* and *Hong Kong* has much higher co-occurrence frequency than *VLDB* and *Shanghai*, which avoids sending *VLDB Shanghai* to the search engine.

1.5 Challenges

Many challenges remain. For example, the concepts in Probase consists of noun phrases only. Given a sentence "Japan invaded China", Probase conceptualizes it into *Asian countries*, as the verb *invaded* is ignored. Currently, we are introducing actions as concepts into Probase, such that the above sentence can be conceptualized into *war* or *WW II*, and an action such as "go to school" can be conceptualized into *education*. Instead of relying on semantic role labeling techniques, the actions are mined from billions of documents as well, based on the current knowledge in Probase. Furthermore, it will be interesting to see how Probase can be used to support general purpose Q/A and machine reading. Also, the uncertain data and the probabilistic platform can become a testbed for various probabilistic reasoning mechanisms, including the Bayesian logic network and the Markov logic network, etc.

1.6 Homepage of the Probase Project

Probase Homepage: http://research.microsoft.com/probase/

Chapter 2

Probase Package

2.1 System Requirements

To run our package smoothly, your environment must meet the following requirements:

	Minimum Requirements	*Recommended*
Operating System:	x64-based platform	Windows Server 2008 R2
Processor:	x64 processor	Multi-core CPU
Memory:	24GB	$\geq$32GB
.Net Framework:	.net framework 4.0 full	.net framework 4.0 full

2.2 Storage Structure

Figure 2.1 shows the storage structure of Probase package. This will help you more easily understand the APIs and return results.

2.3 What are included in the package

There are several parts in the package.

- lib
  - ProbasePackage.dll
  - SingularPluralConverter.dll
  - Trinity.dll
- TrinityAndIndex

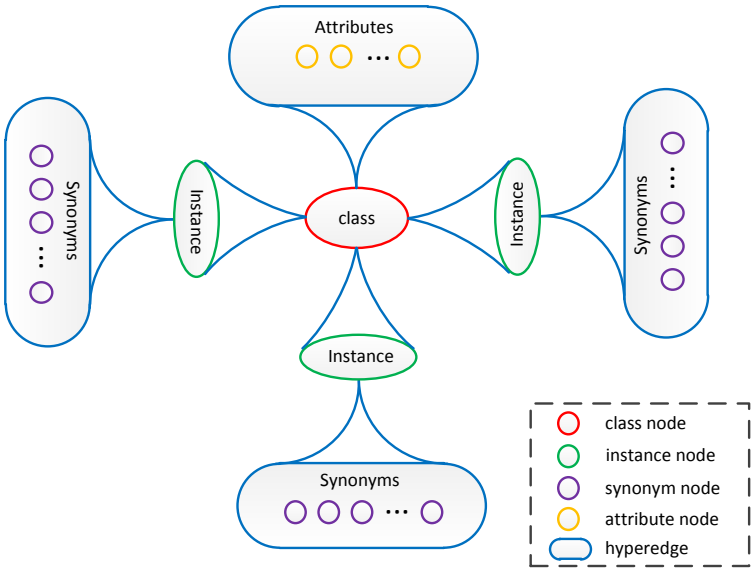

Figure 2.1: Storage Structure

The "lib" folder contains three libraries which you need to add as references to your own project: ProbasePackage.dll, Trinity.dll and SingularPluralConverter.dll.

The "TrinityAndIndex" folder contains trinity storage files and necessary indexes. DON'T CHANGE ANYTHING IN THIS FOLDER.

Besides, "Readme.txt" tells you how to quickly set up the package in your project, "sample.cs" is a sample to show how to easily consume our APIs, and 'Probase-Manual.pdf' contains more details.

2.4 Quick Setup

Before you could call the APIs, some basic operations are needed to initialize the package.

First of all, add ProbasePackage.dll, Trinity.dll and SingularPluralConverter.dll to the References of your solution.

Second, make sure that the platform of your solution is based on **.net framework 4.0 full** and **x64**(You may need to change it in Project → Properties → Build → Platform target).

Finally, be sure to set ProbasePackage.Configuration.TrinityRoot and run ProbasePackage.ProbaseAPI.Initialize() before referring to the APIs. The *TrinityRoot* should be set to the path of the folder named "TrinityAndIndex".

Please note that:

- **The initialization only needs to be run once at the beginning of the pro-**

gram.

- **The initialization may take about 2 $\sim$ 3 minutes and occupy 21 GB memory.**

2.5 Recommended Basic Access APIs

2.5.1 ProbaseAPI.GetClassByInstance Method

Syntax

```
Dictionary<string, List<NameScorePair>> GetClassByInstance(
    string instance, int topN, Boolean exactMatch)
```

Function

Return the class name and corresponding probability for a given instance, ranking by the probability of the class-instance pair.

Parameters

instance

Type: string

The name of the given instance.

topN

Type: int

It indicates that the API will return topN of the final results. If you set -1, all results will be returned.

exactMatch

Type: Boolean

Whether the instance is exactly what the user want to find. If exactMatch is set to false, the API will also consider other forms of the instance name, such as singular, plural, case-insensitive, etc.

Return values

Return a dictionary which stores the possible instance names(the different forms of the given instance) and the corresponding topN classes together with their possibilities. If you set *exactMatch* as *true*, the return dictionary should contain only one key.

Examples

```
using System;
using System.Collections.Generic;
using ProbasePackage;

class Program
{
    static void Main()
    {
        ProbasePackage.Configuration.TrinityRoot =
            @"D:\ProbasePackage\Ver0.05\TrinityAndIndex";
        ProbasePackage.ProbaseAPI.Initialize();
        Dictionary<string, List<ProbasePackage.NameScorePair>>
           returnDic =
            ProbasePackage.ProbaseAPI.GetClassByInstance ("china"
               , 5, false);
        if (returnDic.Count == 0)
            Console.WriteLine("No items found!\n");
        else
        {
            foreach (string instanceName in returnDic.Keys)
            {
                Console.WriteLine("PossibleInstanceName: " +
                   instanceName);
                List< ProbasePackage.NameScorePair > returnList =
       returnDic[instanceName];
                for (int i = 0; i < returnList.Count; i++)
                    Console.WriteLine(returnList[i].name + '\t' +
                        returnList[i].score.ToString("F6"));
                Console.WriteLine("Total Count: " + returnList.
                   Count + '\n');
            }
        }
    }
}
/* This example produces output similar to the following:
 * PossibleInstanceName: China
 * country 0.374875
 * developing country      0.053206
 * emerging market 0.043337
 * nation  0.038474
 * emerging economy        0.028939
 * Total Count: 5
 *
 * PossibleInstanceName: china
 * item    0.205567
```

```
 * fragile item    0.047109
 * service 0.032120
 * material        0.025696
 * artifact        0.019272
 * Total Count: 5
 */
```

2.5.2 ProbaseAPI.GetOneClassSetByInstance Method

Syntax

```
List<NameScorePair> GetOneClassSetByInstance(
    string synonym, int topN, Boolean exactMatch)
```

Function

Return the class name and corresponding probability for a given instance, merging the results of all the possible forms of the input instance names.

Parameters

synonym

Type: string

The name of the given instance.

topN

Type: int

It indicates that the API will return topN of the final results. If you set -1, all results will be returned.

exactMatch

Type: Boolean

Whether the instance is exactly what the user want to find. If exactMatch is set to false, the API will also consider other forms of the instance name, such as singular, plural, case-insensitive, etc.

Return values

Return a list which stores the topN class names and the corresponding probabilities.

Examples

```
using System;
using System.Collections.Generic;
using ProbasePackage;

class Program
{
    static void Main()
    {
        ProbasePackage.Configuration.TrinityRoot =
            @"D:\ProbasePackage\Ver0.05\TrinityAndIndex";
        ProbasePackage.ProbaseAPI.Initialize();

        List<ProbasePackage.NameScorePair> returnList1 =
                        ProbasePackage.ProbaseAPI.
                           GetOneClassSetByInstance("china", 25,
                           false);
        if (returnList1.Count == 0)
                        Console.WriteLine("No items found!\n");
        else
        {
                        for (int i = 0; i < returnList1.Count; i
                           ++)
            Console.WriteLine(returnList1[i].name + '\t' +
                returnList1[i].score);
            Console.WriteLine("Total Count: " + returnList1.
                                Count + '\n');
        }
    }
}
/* country 0.577058259665566
 * nation  0.0717328789008414
 * market  0.0670465438278837
 * economy 0.0481946959207583
 * asian country   0.0369581425071893
 * developing country      0.0238577058259666
 * place   0.0231654063265523
 * emerging market 0.0214080306741932
 * emerging economy        0.0165619341782937
 * area    0.0126211524123975
 * state   0.0111832996059218
 * Others  0.0103312386835659
 * developing nation       0.00958568537650442
 * large country   0.00862711683885398
 * region  0.00804132495473426
 * power   0.00766854830120354
```

```
 * government      0.00655021834061135
 * culture 0.00495260411119395
 * jurisdictions   0.00426030461177974
 * emerging country        0.0042070508041325
 * society 0.0042070508041325
 * communist country       0.00404728938119076
 * case    0.00399403557354351
 * east asian country      0.00388752795824901
 * location        0.00372776653530727
 * Total Count: 25
 */
```

2.5.3 ProbaseAPI.GetInstanceByClass Method

Syntax

```
Dictionary<string, List<EntityInfo>> GetInstanceByClass(
    string className, int topN, Boolean exactMatch)
```

Function

Return all the instances and their information under a given class, ranking by the probability of the class-instance pair.

Parameters

className

Type: string

The name of the given class.

topN

Type: int

It indicates that the API will return topN of the final results.If you set -1, all results will be returned.

exactMatch

Type: Boolean

Whether the className is exactly what the user want to find. If exactMatch is set to false, the API will also consider other forms of the class name, such as singular, plural, case-insensitive, etc.

Return values

Return a dictionary which stores the possible class names and the corresponding topN instances together with their information. Information includes name, total frequency, probability, synonym under the instance, relevant score of the instance-synonym pair and the frequency of the synonym. If you set *exactMatch* as *true*, the return dictionary should contain only one key.

The structures of EntityInfo and EntityFrequencyScore are shown as below.

```
public class EntityInfo
{
    public String name;
    public int totalFrequency;
    public double probability;
    public List<EntityFrequencyScore> SynonymInfo;
}
public class EntityFrequencyScore
{
    private string entityName;
    private string score;
    private int frequency;
}
```

Examples

```
using System;
using System.Collections.Generic;
using ProbasePackage;

class Program
{
    static void Main()
    {
        ProbasePackage.Configuration.TrinityRoot =
            @"D:\ProbasePackage\Ver0.05\TrinityAndIndex";
        ProbasePackage.ProbaseAPI.Initialize();

        Dictionary<string, List< ProbasePackage.EntityInfo>>
           returnDic2 =
        ProbasePackage.ProbaseAPI.GetInstanceByClass("Company", 3
           , false);
        if (returnDic2.Count == 0)
            Console.WriteLine("No items found!\n");
        else
        {
            foreach (string className in returnDic2.Keys)
```

```
            {
                Console.WriteLine("PossibleClassName: " +
                   className);
                List< ProbasePackage.EntityInfo> returnList2 =
                   returnDic2[className];
                for (int i = 0; i < returnList2.Count; i++)
                {
                    Console.WriteLine("EntityName: " +
                       returnList2[i].name
                        + "\tProb: " + returnList2[i].probability
                           .ToString("F6")
                        + '\n' + "Synonym: ");
                    foreach(ProbasePackage.EntityFrequencyScore
                       efs in returnList2[i].SynonymInfo)
                    {
                        Console.WriteLine('\t' + efs.
                           GetEntityName());
                    }
                }
                Console.WriteLine("Total Count: " + returnList2.
                   Count + '\n');
            }
        }
    }
}
/* This example produces output similar to the following:
 * PossibleClassName: company
 * EntityName: microsoft   Prob: 0.017736
 * Synonym:
 *        Microsoft
 *        Micro Soft
 *        Microsoft Corp.
 *         MSFT
 *        Microsoft Australia
 *        Microsoft Network
 *        computer giant
 *        computer giant NCR
 *        Microsoft Corp
 *        e Microsoft Corp.
 *        software giant Microsoft
 *        Microsoft Hardware
 *        Microsoft battle
 *        Microsoft Canada
 *        Microsoft Services
 *        Microsoft Microsoft
 *        47 Microsoft
 * EntityName: ibm Prob: 0.016365
```

```
 * Synonym:
 *        Barrister Global Services Network
 *        IBM
 *        IBM Corp.
 *        IBM IBM
 *        International Business Machines Corp.
 *        International Business Machines
 *        I.B.M.
 *        International Business Machines Inc.
 *        IBM unit
 *        IBM Corporation
 *        International Business Machines Corp
 *        IBM Canada Ltd
 *        International Business Machines Corporation
 *        N.Y.-based IBM Corp.
 *        IBM Daksh
 *        IBM Korea
 *        RedHatand IBM
 *        NCR IBM
 * EntityName: google      Prob: 0.009924
 * Synonym:
 *        Google
 *        Google Inc.
 *        Google Inc
 *        Google TV
 *        Google Checkout
 *        GoogleEarth
 *        Microsoftand Google
 * Total Count: 3
 */
```

2.5.4 ProbaseAPI.GetClassByAttribute Method

Syntax

```
Dictionary<string, List<NameScorePair>> GetClassByAttribute(
    string attribute, int topN, Boolean exactMatch)
```

Function

Return the class name and corresponding probability for a given attribute, ranking by the probability of the class-attribute pair.

Parameters

attribute

Type: string

The name of the given attribute.

topN

Type: int

It indicates that the API will return topN of the final results. If you set -1, all results will be returned.

exactMatch

Type: Boolean

Whether the attribute is exactly what the user want to find. If exactMatch is set to false, the API will also consider other forms of the attribute name, such as singular, plural, case-insensitive, etc.

Return values

Return a dictionary which stores the possible attribute names and the corresponding topN classes together with their possibilities. If you set *exactMatch* as *true*, the return dictionary should contain only one key.

Examples

```
using System;
using System.Collections.Generic;
using ProbasePackage;

class Program
{
    static void Main()
    {
        ProbasePackage.Configuration.TrinityRoot =
            @"D:\ProbasePackage\Ver0.05\TrinityAndIndex";
        ProbasePackage.ProbaseAPI.Initialize();
        Dictionary<string, List< ProbasePackage.NameScorePair>>
           returnDic3 =
            ProbasePackage.ProbaseAPI.GetClassByAttribute ("
               capitals", 10, false);
        if (returnDic3.Count == 0)
            Console.WriteLine("No items found!\n");
        else
        {
            foreach (string attributeName in returnDic3.Keys)
            {
```

```
                Console.WriteLine("PossibleAttributeName: " +
                   attributeName);
                List<ProbasePackage.NameScorePair> returnList3 =
                    returnDic3[attributeName];
                for (int i = 0; i < returnList3.Count; i++)
                    Console.WriteLine(returnList3[i].name + '\t'
                       +
                        returnList3[i].score.ToString("F6"));
                Console.WriteLine("Total Count: " + returnList3.
                   Count + '\n');
            }
        }
    }
}
/* This example produces output similar to the following:
 * PossibleAttributeName: capital
 * country 0.370836
 * state   0.199370
 * city    0.031611
 * province        0.028995
 * nation  0.025538
 * region  0.023659
 * jurisdictions   0.016739
 * island  0.015646
 * municipality    0.011606
 * european country        0.011013
 * Total Count: 10
 */
```

2.5.5 ProbaseAPI.GetAttributeByClass Method

Syntax

```
Dictionary<string, List<NameScorePair>> GetAttributeByClass(
    string className, int topN, Boolean exactMatch)
```

Function

Return the attribute name and corresponding probability for a given class, ranking by the probability of the class-attribute pair.

Parameters

className

Type: string

The name of the given class.

topN

Type: int

It indicates that the API will return topN of the final results. If you set -1, all results will be returned.

exactMatch

Type: Boolean

Whether the instance is exactly what the user want to find. If exactMatch is set to false, the API will also consider other forms of the class name, such as singular, plural, case-insensitive, etc.

Return values

Return a dictionary which stores the possible class names and the corresponding topN attributes together with their possibilities. If you set *exactMatch* as *true*, the return dictionary should contain only one key.

Examples

```
using System;
using System.Collections.Generic;
using ProbasePackage;

class Program
{
    static void Main()
    {
        ProbasePackage.Configuration.TrinityRoot =
            @"D:\ProbasePackage\Ver0.05\TrinityAndIndex";
        ProbasePackage.ProbaseAPI.Initialize();
        Dictionary<string, List< ProbasePackage.NameScorePair>>
           returnDic4 =
            ProbasePackage.ProbaseAPI.GetAttributeByClass ("
               superstar", 5, true);
        if (returnDic4.Count == 0)
            Console.WriteLine("No items found!\n");
        else
        {
            foreach (string className in returnDic4.Keys)
            {
                Console.WriteLine("PossibleClassName: " +
                   className);
                List< ProbasePackage.NameScorePair> returnList4 =
```

```
returnDic4[className];
                for (int i = 0; i < returnList4.Count; i++)
                    Console.WriteLine(returnList4[i].name + '\t'
                       +
                        returnList4[i].score.ToString("F6"));
                Console.WriteLine("Total Count: " + returnList4.
                   Count + '\n');
            }
        }
    }
}
/* This example produces output similar to the following:
 * PossibleClassName: superstar
 * name    0.262051
 * birth date      0.067213
 * birth place     0.067213
 * weight  0.063113
 * trainer 0.054592
 * Total Count: 5
 */
```

2.6 Recommended Conceptualization APIs

2.6.1 ConceptualizationAPI.GetClassByMultipleInput Method

Syntax

```
List<NameScorePair> GetClassByMultipleInput(
    List<string> inputList, string inputType, int topN)
```

Function

Return the possible class name and corresponding probability for multiple given inputs, which may be instances or attributes. The final result are ranked by the probability of the class.

Parameters

inputList

Type: List of string

The list which stores the given inputs.

inputType

Type: string

The type of the inputs, it can be either Configuration.InputType_Synonym or Configuration.InputType_Attribute.

topN

Type: int

It indicates that the API will return topN of the final results.

Return values

Return a List which stores the topN possible class names and the possibilities.

Examples

```
using System;
using System.Collections.Generic;
using ProbasePackage;

class Program
{
    static void Main()
{
    ProbasePackage.Configuration.TrinityRoot =
    @"D:\ProbasePackage\Ver0.05\TrinityAndIndex";
        ProbasePackage.ProbaseAPI.Initialize();
        List<string> inputList1 = new List<string>();
        inputList1.Add("position");
        inputList1.Add("name");
        inputList1.Add("club");
        List<ProbasePackage.NameScorePair> returnList1 =
           ProbasePackage.ConceptualizationAPI.
              GetClassByMultipleInput(inputList1,
            ProbasePackage.Configuration.InputType_Attribute, 15)
               ;
        if (returnList1.Count == 0)
            Console.WriteLine("No items found!\n");
        else
        {
            for (int i = 0; i < returnList1.Count; i++)
                Console.WriteLine(returnList1[i].name + '\t' +
                    returnList1[i].score);
            Console.WriteLine("Total Count: " + returnList1.Count
                + '\n');
        }
    }
}
```

```
/* This example produces output similar to the following:
 * star player     0.672179929480976
 * athlete 0.317130799533957
 * well-known veteran      0.00359861631611971
 * fulham's key player     0.0018203108476972
 * league manager  0.000998627612134618
 * modern-day player       0.000618290093948136
 * united's younger player 0.000432108371542334
 * mobile wallpaper theme  0.000378627951057557
 * head coach      0.000377953208955085
 * lower-tier name 0.000374530056866337
 * hall-of-fame talent     0.000318185068759354
 * produced major leaguers 0.000248980483570343
 * streetball legend       0.000165232766553
 * high profile player     0.000130992589320023
 * second-round draft pick 0.000127981662440261
 * Total Count: 15
 */
```

2.6.2 ConceptualizationAPI.GetClassByMultipleInputCaseInsensitive Method

Syntax

```
List<NameScorePair> GetClassByMultipleInputCaseInsensitive(
    List<string> inputList, string inputType, int topN)
```

Function

Return the possible class name and corresponding probability for multiple given inputs, which may be instances or attributes. The final result are ranked by the probability of the class. This function will consider all the possible lower and upper forms of the inputs and merge the results to give one list.

Parameters

inputList

Type: List of string

The list which stores the given inputs.

inputType

Type: string

The type of the inputs, it can be either Configuration.InputType_Synonym or Configuration.InputType_Attribute.

topN

Type: int

It indicates that the API will return topN of the final results.

Return values

Return a List which stores the topN possible class names and the possibilities.

Examples

```
using System;
using System.Collections.Generic;
using ProbasePackage;

class Program
{
    static void Main()
    {
        ProbasePackage.Configuration.TrinityRoot =
            @"D:\ProbasePackage\Ver0.05\TrinityAndIndex";
        ProbasePackage.ProbaseAPI.Initialize();
        List<string> inputList2 = new List<string>();
        inputList2.Add("google");
        inputList2.Add("Apple");
        inputList2.Add("microsoft");
        inputList2.Add("INTEL");
        List<ProbasePackage.NameScorePair> returnList2 =
       ProbasePackage.ConceptualizationAPI.
          GetClassByMultipleInputCaseInsensitive(
            inputList2, ProbasePackage.Configuration.
               InputType_Synonym, 10);
        if (returnList2.Count == 0)
            Console.WriteLine("No items found!\n");
        else
        {
            for (int i = 0; i < returnList2.Count; i++)
                Console.WriteLine(returnList2[i].name + '\t' +
                    returnList2[i].score);
            Console.WriteLine("Total Count: " + returnList2.Count
                + '\n');
        }
    }
}
```

```
/* This example produces output similar to the following:
 * disruptive firm 0.158174975323777
 * forum few big one       0.158174975323777
 * technology giant        0.133904947694563
 * technology company      0.109847283031224
 * tech company    0.0607165358168951
 * tech giant      0.0592425104562462
 * tech bellwether 0.0540850403978788
 * large technology company        0.0528877232306868
 * technology behemoth     0.0304228352238069
 * tech firm       0.021747665366258
 * Total Count: 10
 */
```

Chapter 3

WCF Service

3.1 Quick Setup

In this section, we first introduce the basic usage of ProbasePackage Wcf Service.

- New a project or open an existing project in Visual Studio.
- Right-click References in the Solution Explorer and select "Add Service Reference".
- Type in http://msradb029/probase/ProbaseService.svc in Address, click Go and rename the reference to "ProbaseService", see Figure 3.1.
- Add the following service namespace to your program.

```
using YourNameSpace.ProbaseService;
```

3.2 How to call the APIs

The following codes are the typical examples to use the APIs in the service.

```
        ProbaseServiceClient probaseService = new
           ProbaseServiceClient();
        //call differnt APIs provided by the service
        probaseService.GetAttributeByClass("country", 20, false);
        probaseService.Close();
```

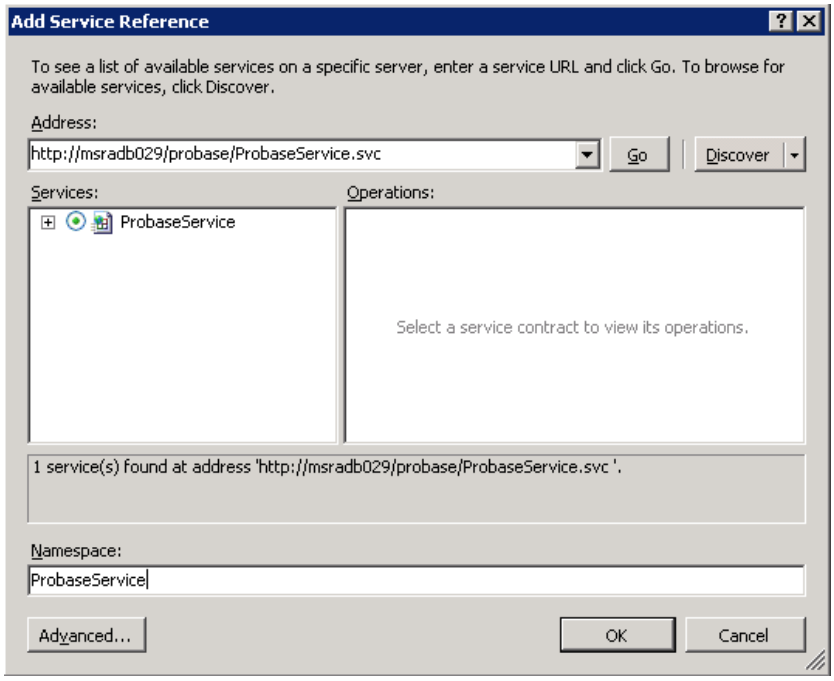

Figure 3.1: Add Probase's WCF Service

3.3 Recommended Basic Access APIs

3.3.1 GetClassByInstance Method

Syntax

```
Dictionary<string, NameScorePair[]> GetClassByInstance(
string instanceName, int topN, Boolean exactMatch)
```

Function

Return the class name and corresponding probability for a given instance, ranking by the probability of the class-instance pair.

Parameters

instanceName

Type: string

The name of the given instance.

topN

Type: int

It indicates that the API will return topN of the final results. If you set -1, all results will be returned.

exactMatch

Type: Boolean

Whether the instance is exactly what the user want to find. If exactMatch is set to false, the API will also consider other forms of the instance name, such as singular, plural, case-insensitive, etc.

Return values

Return a dictionary which stores the possible instance names and the corresponding topN classes together with their possibilities. If you set *exactMatch* as *true*, the return dictionary should contain only one key.

Examples

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using ProbasePackageTest.ProbaseService;

class Program
{
    static void Main()
    {
            ProbaseServiceClient probaseService = new
               ProbaseServiceClient();
            Dictionary<string, NameScorePair[]> returnDic =
               probaseService.GetClassByInstance("china", 5,
               false);
            if (returnDic.Count == 0)
                Console.WriteLine("No items found!\n");
            else
            {
                foreach (string instanceName in returnDic.Keys)
                {
                    Console.WriteLine("PossibleInstanceName: " +
                       instanceName);
                    NameScorePair[] returnList = returnDic[
                       instanceName];
```

```
                    for (int i = 0; i < returnList.Count(); i++)
                        Console.WriteLine(returnList[i].name + '\
                            t' +
                             returnList[i].score.ToString("F6"));
                    Console.WriteLine("Total Count: " +
                        returnList.Count() + '\n');
                }
            }
            probaseService.Close();

        }
    }
    /* This example produces output similar to the following:
     * PossibleInstanceName: China
     * country 0.374875
     * developing country      0.053206
     * emerging market 0.043337
     * nation  0.038474
     * emerging economy        0.028939
     * Total Count: 5
     *
     * PossibleInstanceName: china
     * item    0.205567
     * fragile item    0.047109
     * service 0.032120
     * material        0.025696
     * artifact        0.019272
     * Total Count: 5
     */
```

3.3.2 GetInstanceByClass Method

Syntax

```
Dictionary<string, EntityInfo[]> GetInstanceByClass(
string className, int topN, Boolean exactMatch)
```

Function

Return all the instances and their information under a given class, ranking by the probability of the class-instance pair.

Parameters

className

Type: string

The name of the given class.

topN

Type: int

It indicates that the API will return topN of the final results. If you set -1, all results will be returned.

exactMatch

Type: Boolean

Whether the className is exactly what the user want to find. If exactMatch is set to false, the API will also consider other forms of the class name, such as singular, plural, case-insensitive, etc.

Return values

Return a dictionary which stores the possible class names and the corresponding topN instances together with their information. Information includes name, total frequency, probability, synonym under the instance, relevant score of the instance-synonym pair and the frequency of the synonym. If you set *exactMatch* as *true*, the return dictionary should contain only one key.

The structures of EntityInfo and EntityFrequencyScore are shown as below.

```
public class EntityInfo
{
    public String name;
    public int totalFrequency;
    public double probability;
public List<EntityFrequencyScore> SynonymInfo;
}
public class EntityFrequencyScore
{
    private string entityName;
    private string score;
    private int frequency;
}
```

Examples

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Text;
using ProbasePackageTest.ProbaseService;

class Program
{
    static void Main()
    {
            ProbaseServiceClient probaseService = new
               ProbaseServiceClient();
            Dictionary<string, EntityInfo[]> returnDic2 =
               probaseService.GetInstanceByClass("Company", 3,
               false);
            if (returnDic2.Count == 0)
                Console.WriteLine("No items found!\n");
            else
            {
                foreach (string className in returnDic2.Keys)
                {
                    Console.WriteLine("PossibleClassName: " +
                       className);
                    EntityInfo[] returnList2 = returnDic2[
                       className];
                    for (int i = 0; i < returnList2.Count(); i++)
                    {
                        Console.WriteLine("EntityName: " +
                           returnList2[i].name
                            + "\tProb: " + returnList2[i].
                               probability.ToString("F6")
                            + '\n' + "Synonym: ");
                        foreach (EntityFrequencyScore efs in
                           returnList2[i].SynonymInfo)
                        {
                            Console.WriteLine('\t' + efs.
                               entityName);
                        }
                    }
                    Console.WriteLine("Total Count: " +
                       returnList2.Count() + '\n');
                }
            }

            probaseService.Close();
    }
}
/* This example produces output similar to the following:
 * PossibleClassName: company
 * EntityName: microsoft   Prob: 0.017736
```

```
* Synonym:
*        Microsoft
*        Micro Soft
*        Microsoft Corp.
*         MSFT
*        Microsoft Australia
*        Microsoft Network
*        computer giant
*        computer giant NCR
*        Microsoft Corp
*        e Microsoft Corp.
*        software giant Microsoft
*        Microsoft Hardware
*        Microsoft battle
*        Microsoft Canada
*        Microsoft Services
*        Microsoft Microsoft
*        47 Microsoft
* EntityName: ibm Prob: 0.016365
* Synonym:
*        Barrister Global Services Network
*        IBM
*        IBM Corp.
*        IBM IBM
*        International Business Machines Corp.
*        International Business Machines
*        I.B.M.
*        International Business Machines Inc.
*        IBM unit
*        IBM Corporation
*        International Business Machines Corp
*        IBM Canada Ltd
*        International Business Machines Corporation
*        N.Y.-based IBM Corp.
*        IBM Daksh
*        IBM Korea
*        RedHatand IBM
*        NCR IBM
* EntityName: google      Prob: 0.009924
* Synonym:
*        Google
*        Google Inc.
*        Google Inc
*        Google TV
*        Google Checkout
*        GoogleEarth
*        Microsoftand Google
```

```
 * Total Count: 3
 */
```

3.3.3 GetClassByAttribute Method

Syntax

```
Dictionary<string, NameScorePair[]> GetClassByAttribute(
string attributeName, int topN, Boolean exactMatch)
```

Function

Return the class name and corresponding probability for a given attribute, ranking by the probability of the class-attribute pair.

Parameters

attributeName

Type: string

The name of the given attribute.

topN

Type: int

It indicates that the API will return topN of the final results. If you set -1, all results will be returned.

exactMatch

Type: Boolean

Whether the attribute is exactly what the user want to find. If exactMatch is set to false, the API will also consider other forms of the attribute name, such as singular, plural, case-insensitive, etc.

Return values

Return a dictionary which stores the possible attribute names and the corresponding topN classes together with their possibilities. If you set *exactMatch* as *true*, the return dictionary should contain only one key.

Examples

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using ProbasePackageTest.ProbaseService;

class Program
{
    static void Main()
    {
            ProbaseServiceClient probaseService = new
               ProbaseServiceClient();
            Dictionary<string, NameScorePair[]> returnDic3 =
               probaseService.GetClassByAttribute("capitals", 10,
                false);
            if (returnDic3.Count == 0)
                Console.WriteLine("No items found!\n");
            else
            {
                foreach (string attributeName in returnDic3.Keys)
                {
                    Console.WriteLine("PossibleAttributeName: " +
                        attributeName);
                    NameScorePair[] returnList3 = returnDic3[
                       attributeName];
                    for (int i = 0; i < returnList3.Count(); i++)
                        Console.WriteLine(returnList3[i].name +
                           '\t' +
                            returnList3[i].score.ToString("F6"));
                    Console.WriteLine("Total Count: " +
                       returnList3.Count() + '\n');
                }
            }

            probaseService.Close();
    }
}
/* This example produces output similar to the following:
 * PossibleAttributeName: capital
 * country 0.370836
 * state   0.199370
 * city    0.031611
 * province        0.028995
 * nation  0.025538
 * region  0.023659
 * jurisdictions   0.016739
 * island  0.015646
```

```
 * municipality    0.011606
 * european country        0.011013
 * Total Count: 10
 */
```

3.3.4 GetAttributeByClass Method

Syntax

```
Dictionary<string, NameScorePair[]> GetAttributeByClass(
string className, int topN, Boolean exactMatch)
```

Function

Return the attribute name and corresponding probability for a given class, ranking by the probability of the class-attribute pair.

Parameters

className

Type: string

The name of the given class.

topN

Type: int

It indicates that the API will return topN of the final results. If you set -1, all results will be returned.

exactMatch

Type: Boolean

Whether the instance is exactly what the user want to find. If exactMatch is set to false, the API will also consider other forms of the class name, such as singular, plural, case-insensitive, etc.

Return values

Return a dictionary which stores the possible class names and the corresponding topN attributes together with their possibilities. If you set *exactMatch* as *true*, the return dictionary should contain only one key.

Examples

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using ProbasePackageTest.ProbaseService;

class Program
{
    static void Main()
{
            ProbaseServiceClient probaseService = new
               ProbaseServiceClient();
            Dictionary<string, NameScorePair[]> returnDic4 =
               probaseService.GetAttributeByClass("superstar", 5,
                true);
            if (returnDic4.Count == 0)
                Console.WriteLine("No items found!\n");
            else
            {
                foreach (string className in returnDic4.Keys)
                {
                    Console.WriteLine("PossibleClassName: " +
                       className);
                    NameScorePair[] returnList4 = returnDic4[
                       className];
                    for (int i = 0; i < returnList4.Count(); i++)
                        Console.WriteLine(returnList4[i].name +
                           '\t' +
                            returnList4[i].score.ToString("F6"));
                    Console.WriteLine("Total Count: " +
                       returnList4.Count() + '\n');
                }
            }
            probaseService.Close();
    }
}
/* This example produces output similar to the following:
 * PossibleClassName: superstar
 * name    0.262051
 * birth date      0.067213
 * birth place     0.067213
 * weight  0.063113
 * trainer 0.054592
 * Total Count: 5
 */
```

3.4 Recommended Conceptualization APIs

3.4.1 GetClassByMultipleInput Method

Syntax

```
NameScorePair[] GetClassByMultipleInput(
string[] inputList, string inputType, int topN)
```

Function

Return the possible class name and corresponding probability for multiple given inputs, which may be instances or attributes. The final result are ranked by the probability of the class.

Parameters

inputList

Type: string[]

The list which stores the given inputs.

inputType

Type: string

The type of the inputs, it can be either ”Synonym” or ”Attribute”.

topN

Type: int

It indicates that the API will return topN of the final results.

Return values

Return a List which stores the topN possible class names and the possibilities.

Examples

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using ProbasePackageTest.ProbaseService;

class Program
{
```

```
    static void Main()
{
            ProbaseServiceClient probaseService = new
               ProbaseServiceClient();
            string[] inputList1 = {"position", "name", "club"};
            NameScorePair[] returnList1 = probaseService.
               GetClassByMultipleInput(inputList1, "Attribute",
               15);
            if (returnList1.Count() == 0)
                Console.WriteLine("No items found!\n");
            else
            {
                for (int i = 0; i < returnList1.Count(); i++)
                    Console.WriteLine(returnList1[i].name + '\t'
                       +
                        returnList1[i].score);
                Console.WriteLine("Total Count: " + returnList1.
                   Count() + '\n');
            }

            probaseService.Close();
    }
}
/* This example produces output similar to the following:
 * star player     0.672179929480976
 * athlete 0.317130799533957
 * well-known veteran      0.00359861631611971
 * fulham's key player     0.0018203108476972
 * league manager  0.000998627612134618
 * modern-day player       0.000618290093948136
 * united's younger player 0.000432108371542334
 * mobile wallpaper theme  0.000378627951057557
 * head coach      0.000377953208955085
 * lower-tier name 0.000374530056866337
 * hall-of-fame talent     0.000318185068759354
 * produced major leaguers 0.000248980483570343
 * streetball legend       0.000165232766553
 * high profile player     0.000130992589320023
 * second-round draft pick 0.000127981662440261
 * Total Count: 15
 */
```

3.4.2 GetClassByMultipleInputCaseInsensitive Method

Syntax

```
NameScorePair[] GetClassByMultipleInputCaseInsensitive(
string[] inputList, string inputType, int topN)
```

Function

Return the possible class name and corresponding probability for multiple given inputs, which may be instances or attributes. The final result are ranked by the probability of the class. This function will consider all the possible lower and upper forms of the inputs and merge the results to give one list.

Parameters

inputList

Type: string[]

The list which stores the given inputs.

inputType

Type: string

The type of the inputs, it can be either ”Synonym” or ”Attribute”.

topN

Type: int

It indicates that the API will return topN of the final results.

Return values

Return a List which stores the topN possible class names and the possibilities.

Examples

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using ProbasePackageTest.ProbaseService;

class Program
{
    static void Main()
{
            ProbaseServiceClient probaseService = new
               ProbaseServiceClient();
```

```
            string[] inputList1 = {"Google", "apple", "microsoft"
               , "Intel"};
            NameScorePair[] returnList1 = probaseService.
               GetClassByMultipleInputCaseInsensitive(inputList1,
                "Synonym", 10);
            if (returnList1.Count() == 0)
                Console.WriteLine("No items found!\n");
            else
            {
                for (int i = 0; i < returnList1.Count(); i++)
                    Console.WriteLine(returnList1[i].name + '\t'
                       +
                        returnList1[i].score);
                Console.WriteLine("Total Count: " + returnList1.
                   Count() + '\n');
            }

            probaseService.Close();
    }
}
/* This example produces output similar to the following:
 * disruptive firm 0.158174975323777
 * forum few big one       0.158174975323777
 * technology giant        0.133904947694563
 * technology company      0.109847283031224
 * tech company    0.0607165358168951
 * tech giant      0.0592425104562462
 * tech bellwether 0.0540850403978788
 * large technology company        0.0528877232306868
 * technology behemoth     0.0304228352238069
 * tech firm       0.021747665366258
 * Total Count: 10
 */
```

Chapter 4

Web Interface

4.1 How to use

Please go to http://msradb029/probaseweb

You can see two columns at the top of the window.

Basic Access is used to show basic data of ProbasePackage, see Figure 4.1.

Conceptualization shows the conceptualization result of multiple instances or attributes, see Figure 4.2.

4.2 Recommended Basic Access APIs

4.2.1 GetInstanceByClass

Function

Given a class, return all the instances and their information, including probability, synonyms and relevant score, ranking by the probability of the class-instance pair.

OutputFormat

See Figure 4.3.

4.2.2 GetClassByInstance

Function

Return the class name and corresponding probability for a given instance, ranking by the probability of the class-instance pair.

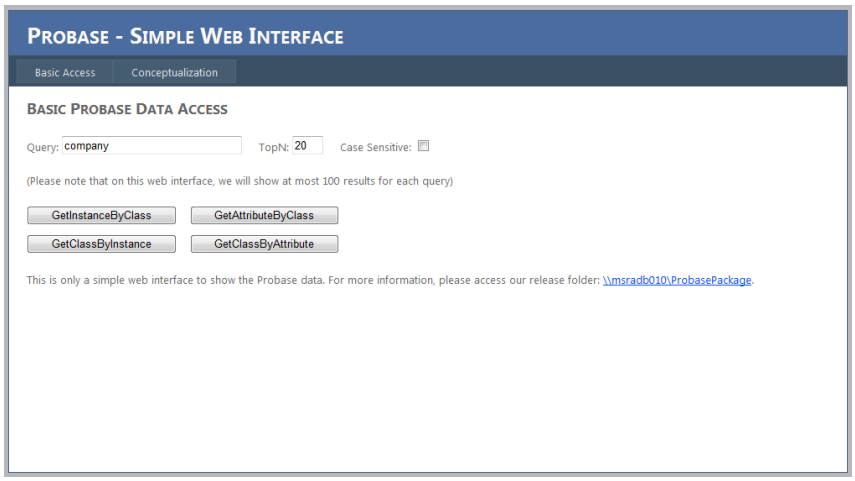

Figure 4.1: Probase Basic Access Interface

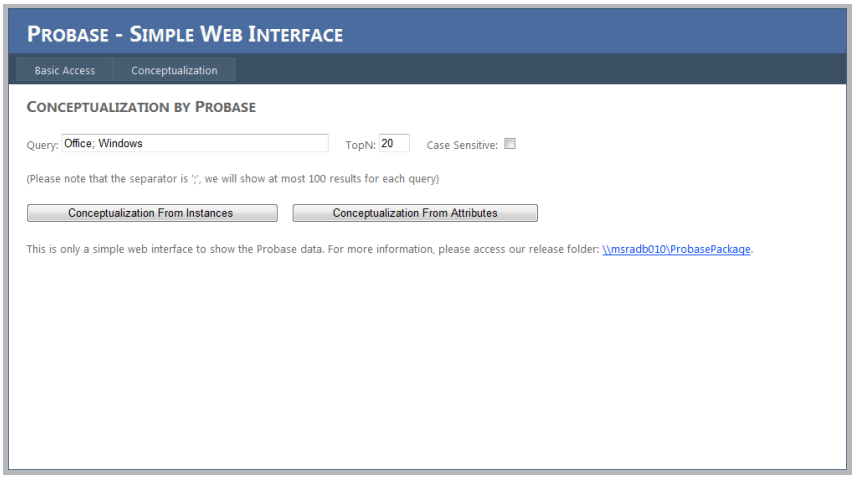

Figure 4.2: Conceptualization Interface

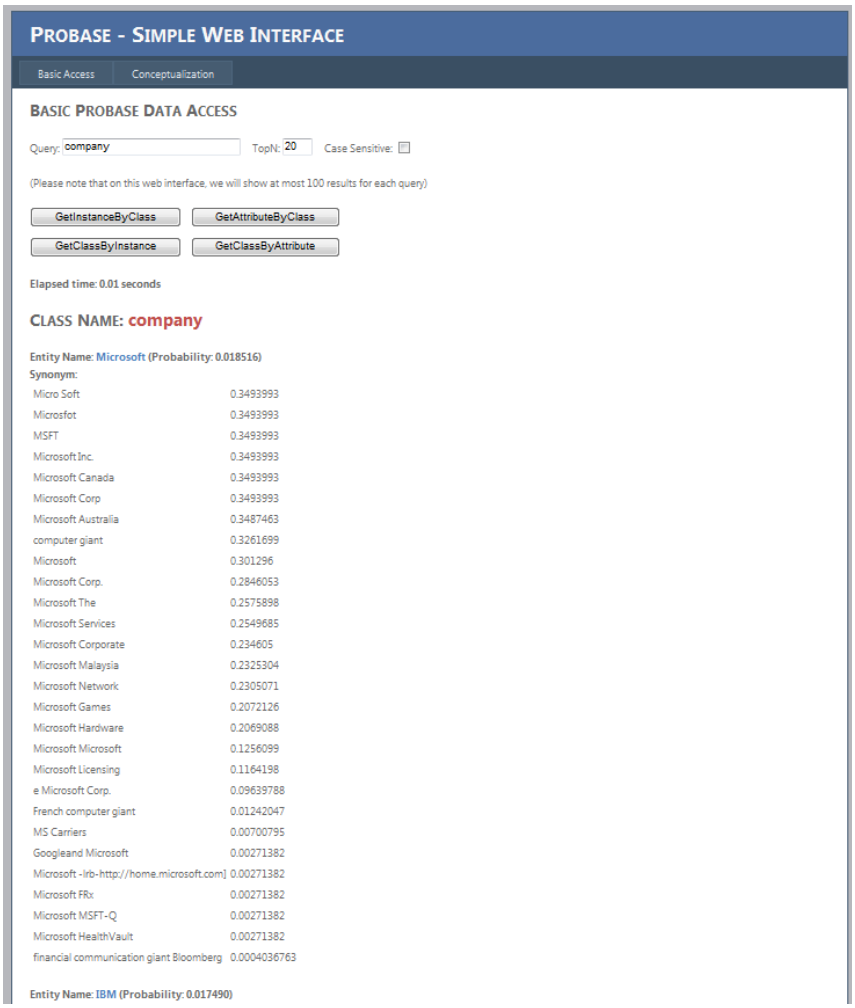

Figure 4.3: Results of GetInstanceByClass

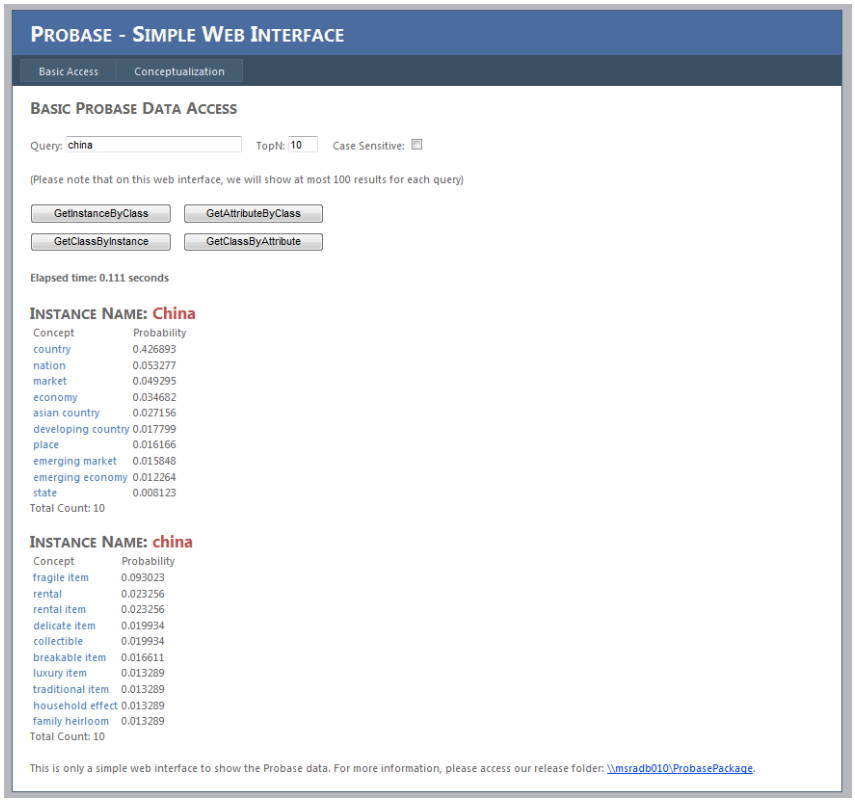

Figure 4.4: Results of GetClassByInstance

OutputFormat

See Figure 4.4.

4.2.3 GetAttributeByClass

Function

Return the attribute name and corresponding probability for a given class, ranking by the probability of the class-attribute pair.

OutputFormat

The format is similar to Figure 4.4.

4.2.4 GetClassByAttribute

Function

Return the class name and corresponding probability for a given attribute, ranking by the probability of the class-attribute pair.

OutputFormat

The format is similar to Figure 4.4.

4.3 Recommended Conceptualization APIs

4.3.1 Conceptualization From Instances

Function

Return the possible class name and corresponding probability for multiple given instances. The final results are ranked by the probability of the class.

Input Tips

Input ';' as the separator of multiple instances.

OutputFormat

See Figure 4.5.

4.3.2 Conceptualization From Attributes

Function

Return the possible class name and corresponding probability for multiple given attributes. The final results are ranked by the probability of the class.

Input Tips

Input ';' as the separator of multiple instances.

OutputFormat

See Figure 4.6.

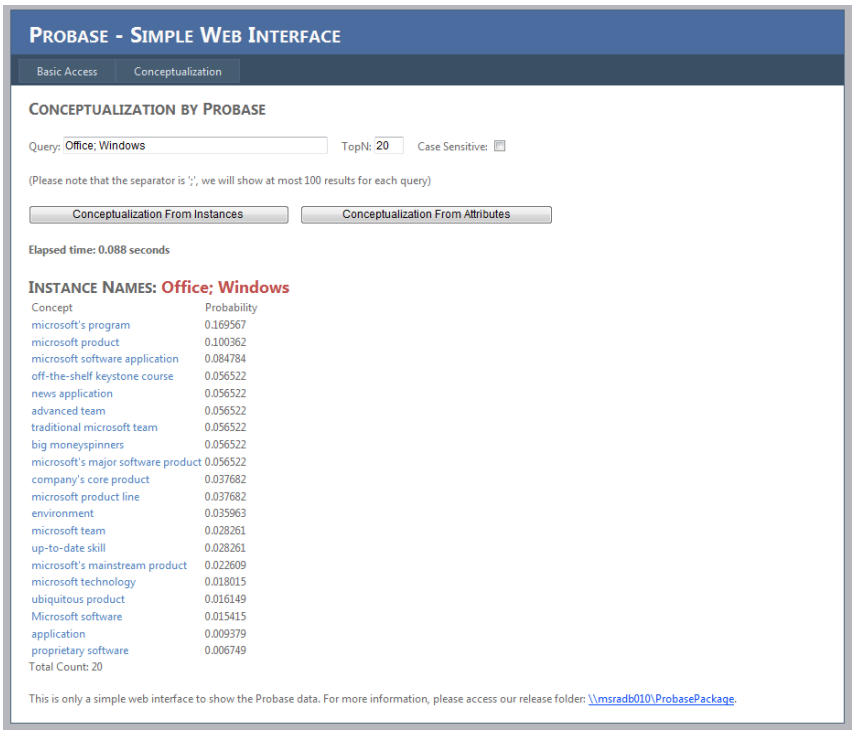

Figure 4.5: Results of Conceptualization From Instances

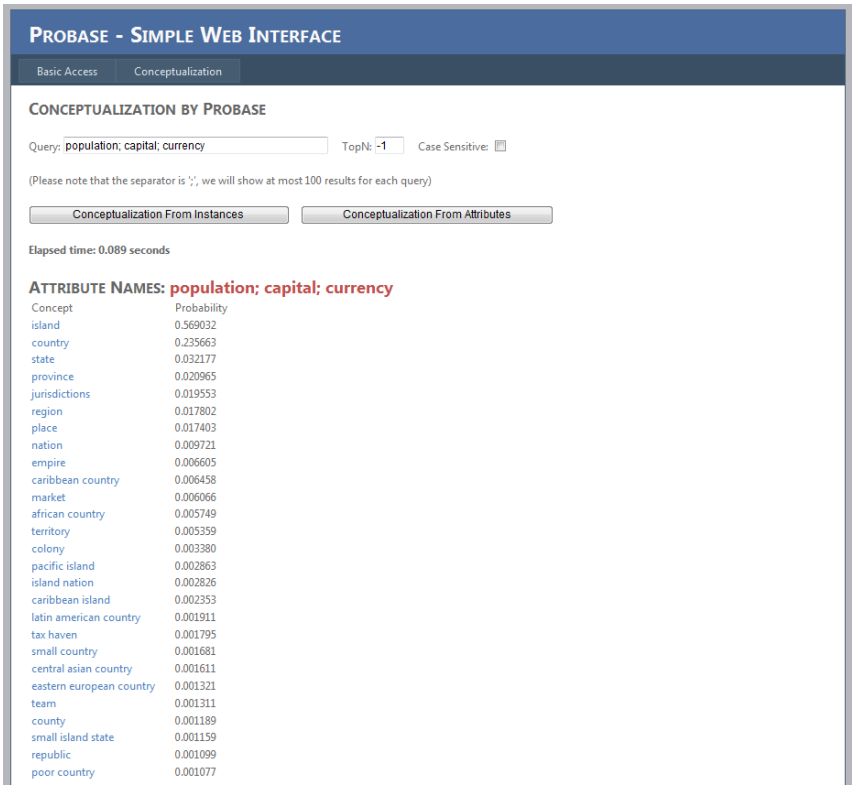

Figure 4.6: Results of Conceptualization From Attributes

Bibliography

[1] D. Blei and J. Lafferty. *Topic Models*, chapter: Topic Models. Taylor and Francis, 2009. (in Press).

[2] P. Bloom. Glue for the mental world. *Nature*, (421):212–213, Jan 2003.

[3] K. D. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD*, 2008.

[4] S. A. Caraballo. Automatic construction of a hypernym-labeled noun hierarchy from text. In *ACL*, 1999.

[5] M. A. Hearst. Automatic acquisition of hyponyms from large text corpora. In *COLING*, pages 539–545, 1992.

[6] T. Lee, Z. Wang, H. Wang, and S. won Hwang. Web scale taxonomy cleansing. Technical Report MSR-TR-2011-30, Microsoft Research, 2011.

[7] D. B. Lenat and R. V. Guha. *Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project*. Addison-Wesley, 1989.

[8] G. L. Murphy. *The big book of concepts*. The MIT Press, 2004.

[9] M. Pasca. Turning web text and search queries into factual knowledge: Hierarchical class attribute extraction. In *AAAI*, pages 1225–1230, 2008.

[10] Y. Song, H. Wang, Z. Wang, and H. Li. Short text conceptualization using a probabilistic knowledgebase. In *IJCAI*, 2011.

[11] J. Wang, B. Shao, H. Wang, and K. Q. Zhu. Understanding tables on the web. Technical Report MSR-TR-2011-29, Microsoft Research, 2010.

[12] P. Wang, H. Li, H. Wang, and K. Q. Zhu. Taxonomy assisted massive relationship extraction from the web. 2010.

[13] Y. Wang, H. Li, H. Wang, and K. Q. Zhu. Toward topic search on the web. Technical Report MSR-TR-2011-28, Microsoft Research, 2010.

[14] W. Wu, H. Li, H. Wang, and K. Zhu. Towards a probabilistic taxonomy of many concepts. Technical Report MSR-TR-2011-25, Microsoft Research, 2011.