Contents

Acknowledgements

Thanks to my parents, my grandparents and my sister who, with their sweet and tireless support, both moral and economic, have allowed me to get here, at the end of this difficult travel, contributing to my personal training.

Thanks to prof. Di Martino, rapporteur of this thesis, as well as for the help given to me in all these years and the great knowledge that he has given me, for the availability and accuracy shown during the whole creating period. Without him this work would not have come to life!

Many thanks to my colleagues and friends Jessica, Luca and Francesco for sharing with me beautiful and ugly moments of this long journey. Without you I would have never seen this day!

Last but not least, I wanted to thank Stefania, Salvatore M. and Salvatore D. for their valuable advices and their willingness to always listen to me.

I would like to end this section with a famous assertion:

Man must persevere in the idea that the incomprehensible is understandable; otherwise he would give up on trying. (J. W. Goethe)

Chapter 1

Introduction

Cloud computing technology can be viewed as a set of software services (SaaS), platform services (PaaS) and infrastructure services (IaaS) offered to users over the net. It has the advantage of offering users scalable and elastic service capabilities, at a reduced cost, speeding up the deployment of their businesses. Indeed, in the SaaS model, a user gets the provider applications that are hosted and run in cloud infrastructure; in the PaaS model, the user gets the platform that is running in the cloud infrastructure for creating or deploying the applications; and in the IaaS model, the user gets the virtual resources in which an execution environment can be deployed to run an application.

A Cloud federation is a new paradigm where a set of Cloud providers are grouped in order to offer a wider range of resources/services and, hence, enlarge their market shares. However, Cloud federations are hindered by the lack of a standard language for describing providers' offers making it difficult for end users to discover and select an appropriate provider within a federation. In fact, service providers describe similar services with different manners (different names for services belonging categories, different names for services functionalities, etc.); such heterogeneous descriptions complicate service comparison and selection, and may create interoperability problems among multiple providers (especially when

migrating from a provider to another).

To overcome the problems stemming from heterogeneous service descriptions within a Cloud federation, this work proposes to define a common ontology for Cloud Services description. It is worth nothing that, there exists some industrial standards and research projects that treat the aforementioned problems; unfortunately, current propositions focus only on the IaaS layer. That is, they do not offer solutions to the PaaS and the SaaS layers. So this work tries to define a semantic representation of cloud services, using an ontology, that allows to represent all cloud services, regardless of their belonging provider and that allows to cloud users to easily and automatically find which cloud services use for their needs.

After search for cloud services is completed, there is another problem for users: understand how to use these services through their APIs. Indeed, it's not an easy task for users to correctly use these services: they have to read a lot of documentation pages to understand how to properly make calls to Restful APIs in order to receive expected outputs. Luckily, in this field there are already some standardization projects like OpenAPI Specifications. Formerly known as Swagger Specifications, is a powerful definition format to describe Restful APIs. The specification creates a Restful interface for easily developing and consuming an API by effectively mapping all the resources and operations associated with it. It's easy to learn, language agnostic and both, human and machine readable. So this work proposes an automatic way to obtain Swagger documentation of API by answering to specific questions about this API. In fact, despite the now undisputed Swagger diffusion, which can be considered a de facto standard at present, APIs producers hardly provide a description using such formalism, because developers should have a great knowledge of Swagger Specifications and must manually generate the OpenAPI file from their existing API codebase, which makes for a long, cumbersome process. Yet swagger, with a set of tools connected to it, undoubtedly provides benefits to both developers and consumers of these APIs. Among these

tools surely we must count:

Swagger-Codegen: can convert OpenAPI definitions into code, saving time and effort. Examples of auto-generated code include stubs for the implementation of the server-side logic, or a complete mock server that allows the client-side developers to start working with the API even before it is done. Services like APIMatic can even use use OpenAPI definitions to create complete SDKs for developers in programming languages that nobody on developer team is fluent in.

Swagger-UI: was the first tool to offer API documentation with integrated test client, so developers could see a list of available API operations and quickly send a request in their browser to test how the API responds before writing any code of their own. Basing documentation on a definition ensures that the documentation covers the entirety of an API and correctly lists all methods, parameters and responses.

Lastly, this work provides meta-informations about APIs' methods and parameters, so when a user needs to do a specific task, the methods to use and which parameters to be specified in order to achieve this task, can be suggested to him. In a hopefully way, this work would like to give a user the chance to say what he needs to do, in the most appropriate language to him, that is natural language (English, to be clear) and then all the specific calls to make, to obtain what the user asks for, are automatically found and easily feasible through Swagger-UI, inserting the needed parameters.

Chapter 2

Background Information

2.1 Cloud Computing

Introduction History has a funny way of repeating itself, or so they say. But it may come as some surprise to find this old cliché applies just as much to the history of computers as to wars, revolutions, and kings and queens. For the last three decades, one trend in computing has been loud and clear: big, centralized, mainframe systems have been "out"; personalized, power-to-the-people, do-it-yourself PCs have been "in." Before personal computers took off in the early 1980s, if your company needed sales or payroll figures calculating in a hurry, you'd most likely have bought in "data-processing" services from another company, with its own expensive computer systems, that specialized in number crunching; these days, you can do the job just as easily on your desktop with off-the-shelf software. Or can you? In a striking throwback to the 1970s, many companies are finding, once again, that buying in computer services makes more business sense than do-it-yourself. This new trend is called cloud computing and, not surprisingly, it's linked to the Internet's inexorable rise. What is cloud computing? How does it work? Let's take a closer look!

2.1.1 NIST cloud computing definition

The National Institute of Standards and Technology (NIST) defines as following cloud computing [22]:

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model promotes availability and is composed of five essential characteristics, three service models, and four deployment models.

Essential Characteristics:

On-demand self-service. A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service's provider.

Broad network access. Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, laptops, and personal digital assistants [PDAs]). *Resource pooling.* The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, network bandwidth, and virtual machines. *Rapid elasticity.* Capabilities can be rapidly and elastically provisioned, in some cases automatically, to quickly scale out and rapidly released to quickly scale in. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be purchased in

any quantity at any time. *Measured Service.* Cloud systems automatically control and optimize resource use by leveraging a metering capability3 at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

Service Models:

Cloud Software as a Service (SaaS). The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through a thin client interface such as a Web browser (e.g., Web-based email). The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited userspecific application configuration settings.

Cloud Platform as a Service (PaaS). The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly application hosting environment configurations.

Cloud Infrastructure as a Service (IaaS). The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, deployed applications, and possibly limited control of select networking components (e.g., host firewalls).

Deployment models:

Private cloud. The cloud infrastructure is operated solely for an organization. It may be managed by the organization or a third party and may exist on premise or off premise.

Community cloud. The cloud infrastructure is shared by several organizations and supports a specific community that has shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be managed by the organizations or a third party and may exist on premise or off premise.

Public cloud. The cloud infrastructure is made available to the general public or a large industry group and is owned by an organization selling cloud services.

Hybrid cloud. The cloud infrastructure is a composition of two or more clouds (private, community, or public) that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load balancing between clouds).

2.1.2 NIST cloud computing reference architecture

The NIST cloud computing reference architecture defines five major *actors: cloud consumer, cloud provider, cloud auditor, cloud broker, and cloud carrier.* Each actor is an entity (a person or an organization) that participates in a transaction or process and/or performs tasks in cloud computing. Figure 2.1 briefly lists the five major actors defined in the NIST cloud computing reference architecture. Figure 2.2 shows the interactions among the actors in the NIST cloud computing reference architecture. A cloud consumer may request cloud services from a cloud provider directly or via a cloud broker. A cloud auditor conducts independent audits and may contact the others to collect necessary information. The details will be discussed in the following sections and be presented as successive diagrams in increasing levels of detail.

Actor	Definition
Cloud Consumer	Person or organization that maintains a business relationship with, and uses service from, *Cloud Providers*.
Cloud Provider	Person, organization, or entity responsible for making a service available to *Cloud Consumers*.
Cloud Auditor	A party that can conduct independent assessment of cloud services, information system operations, performance, and security of the cloud implementation.
Cloud Broker	An entity that manages the use, performance, and delivery of cloud services, and negotiates relationships between *Cloud Providers* and *Cloud Consumers*.
Cloud Carrier	The intermediary that provides connectivity and transport of cloud services from *Cloud Providers* to *Cloud Consumers*.

Figure 2.1: Actors in Cloud Computing

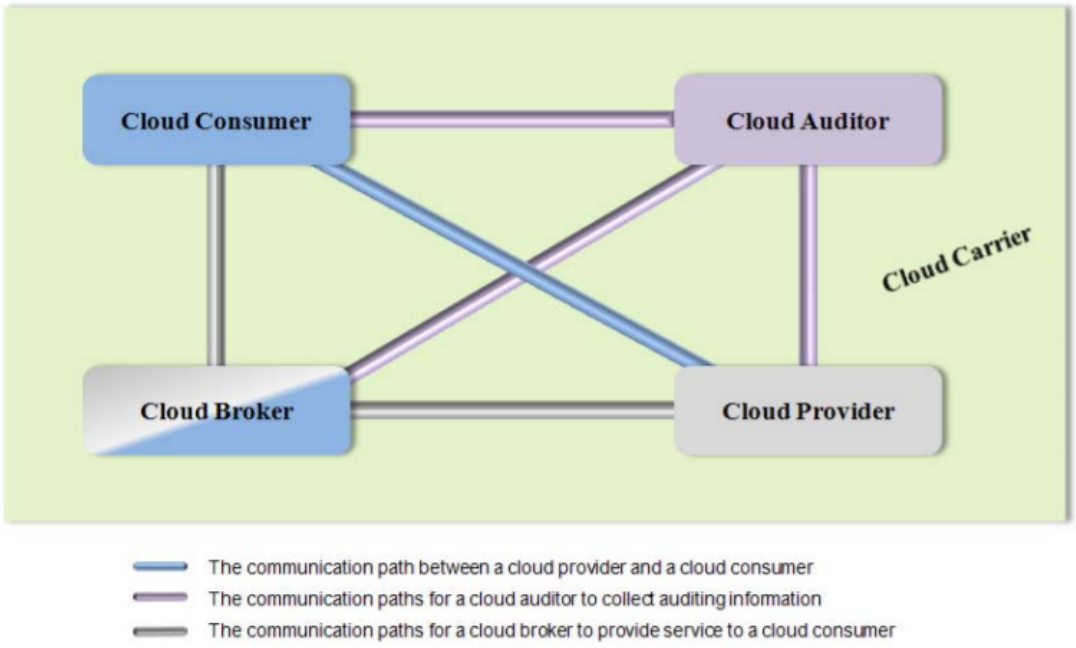


Figure 2.2: Interactions between the Actors in Cloud Computing

2.1.3 Cloud Consumer

The cloud consumer is the ultimate stakeholder that the cloud computing service is created to support. A cloud consumer represents a person or organization that maintains a business relationship with, and uses the service from, a cloud provider.

A cloud consumer browses the service catalog from a cloud provider, requests the appropriate service, sets up service contracts with the cloud provider, and uses the service. The cloud consumer may be billed for the service provisioned, and needs to arrange payments accordingly. Depending on the services requested, the activities and usage scenarios can be different among cloud consumers, as shown in Figure 2.3. Some example usage scenarios are listed in Figure 2.4.

Type	Consumer Activities	Provider Activities
SaaS	Uses application/service for business process operations.	Installs, manages, maintains, and supports the software application on a cloud infrastructure.
PaaS	Develops, tests, deploys, and manages applications hosted in a cloud environment.	Provisions and manages cloud infrastructure and middleware for the platform consumers; provides development, deployment, and administration tools to platform consumers.
IaaS	Creates/installs, manages, and monitors services for IT infrastructure operations.	Provisions and manages the physical processing, storage, networking, and the hosting environment and cloud infrastructure for IaaS consumers.

Figure 2.3: Cloud Consumer and Cloud Provider

SaaS applications are usually deployed as hosted services and are accessed via a network connecting SaaS consumers and providers. The consumers of SaaS can be organizations that provide their members with access to software applications, end users who directly use software applications, or software application administrators who configure applications for end users. SaaS consumers access and use applications on demand, and can be billed on the number of consumers or the amount of consumed services. The latter can be measured in terms of the time in use, the network bandwidth consumed, or the amount/duration of data stored.

Cloud consumers who use PaaS can employ the tools and execution resources provided by cloud providers for the purpose of developing, testing, deploying, and managing applications hosted in a cloud environment. PaaS consumers can be application developers who design and implement application software, applica-

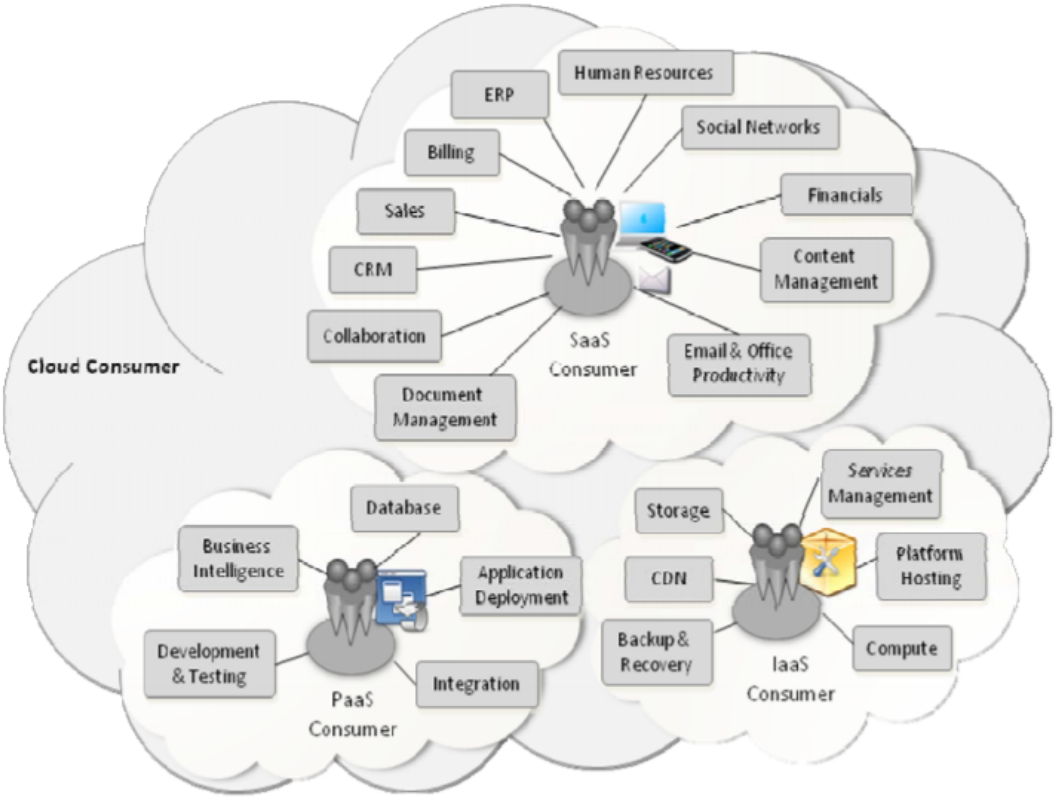


Figure 2.4: Example of Services Available to a Cloud Consumer

tion testers who run and test applications in various cloud-based environments, application deployers who publish applications into the cloud, and application administrators who configure and monitor application performance on a platform. PaaS consumers can be billed by the number of consumers, the type of resources consumed by the platform, or the duration of platform usage.

IaaS clouds provision consumers the capabilities to access virtual computers, networkaccessible storage, network infrastructure components, and other fundamental computing resources, on which consumers can deploy and run arbitrary software. The consumers of IaaS can be system developers, system administrators, and information technology (IT) managers who are interested in creating, installing, managing and monitoring services for IT infrastructure operations. IaaS consumers are provisioned with the capabilities to access these computing

resources, and are billed for the amount of resources consumed.

2.1.4 OpenCrowd Cloud Taxonomy

Some example cloud services available to a cloud consumer are listed below and shown in Figure 2.4: these categories are taken from OpenCrowd Taxonomy [48], a project that aims to provide information on the cloud computing services available all over the cloud world and create a dialog between cloud computing services providers and consumers.

SaaS services:

- Email and Office Productivity: Applications for email, word processing, spreadsheets, presentations, etc.
- Billing: Application services to manage customer billing based on usage and subscriptions to products and services.
- Customer Relationship Management (CRM): CRM applications that range from call center applications to sales force automation.
- Collaboration: Tools that allow users to collaborate in workgroups, within enterprises, and across enterprises.
- Content Management: Services for managing the production of and access to content for web-based applications.
- Document Management: Applications for managing documents, enforcing document production workflows, and providing workspaces for groups or enterprises to find and access documents.
- Financials: Applications for managing financial processes ranging from expense processing and invoicing to tax management.

- Human Resources: Software for managing human resources functions within companies.
- Sales: Applications that are specifically designed for sales functions such as pricing, commission tracking, etc.
- Social Networks: Social software that establishes and maintains a connection among users that are tied in one or more specific types of interdependency.
- Enterprise Resource Planning (ERP): Integrated computer-based system used to manage internal and external resources, including tangible assets, financial resources, materials, and human resources.

PaaS Services:

- Business Intelligence: Platforms for the creation of applications such as dashboards, reporting systems, and data analysis.
- Database: Services offering scalable relational database solutions or scalable non-SQL datastores.
- Development and Testing: Platforms for the development and testing cycles of application development, which expand and contract as needed.
- Integration: Development platforms for building integration applications in the cloud and within the enterprise.
- Application Deployment: Platforms suited for general purpose application development. These services provide databases, web application runtime environments, etc.

IaaS Services:

- Backup and Recovery: Services for backup and recovery of file systems and raw data stores on servers and desktop systems.

- Compute: Server resources for running cloud-based systems that can be dynamically provisioned and configured as needed. NIST SP 500-292 NIST Cloud Computing Reference Architecture 25

- Content Delivery Networks (CDNs): CDNs store content and files to improve the performance and cost of delivering content for web-based systems.

- Services Management: Services that manage cloud infrastructure platforms. These tools often provide features that cloud providers do not provide or specialize in managing certain application technologies.

- Storage: Massively scalable storage capacity that can be used for applications, backups, archival, and file storage.

However, these categories can be considered as macro categories, since within them it is possible to identify other smaller service categories and these ones are those that a consumer looks for. For example in compute Compute category we can find the following micro categories (and others):

- Container orchestration: Allows to easily manage, deploy and scale containerized applications.

- Autoscale: Allows to scale up/down cloud compute capacity, basing on scaling plans.

- Compute Capacity: Offers cloud compute capacity.

2.1.5 Cloud Provider

A cloud provider can be a person, an organization, or an entity responsible for making a service available to cloud consumers. A cloud provider builds the requested software/platform/ infrastructure services, manages the technical infrastructure required for providing the services, provisions the services at agreed-upon service

levels, and protects the security and privacy of the services. Cloud providers undertake different tasks for the provisioning of the various service models. For Cloud Software as a Service, the cloud provider deploys, configures, maintains, and updates the operation of the software applications on a cloud infrastructure so that the services are provisioned at the expected service levels to cloud consumers. The provider of SaaS assumes most of the responsibilities in managing and controlling the applications and the infrastructure, while the cloud consumers have limited administrative control of the applications. For Cloud Platform as a Service, the cloud provider manages the cloud infrastructure for the platform, and provisions tools and execution resources for the platform consumers to develop, test, deploy, and administer applications. Consumers have control over the applications and possibly the hosting environment settings, but cannot access the infrastructure underlying the platform including network, servers, operating systems, or storage. For Cloud Infrastructure as a Service, the cloud provider provisions the physical processing, storage, networking, and other fundamental computing resources, as well as manages the hosting environment and cloud infrastructure for IaaS consumers. Cloud consumers deploy and run applications, have more control over the hosting environment and operating systems, but do not manage or control the underlying cloud infrastructure (e.g., the physical servers, network, storage, hypervisors, etc.). The major five activities of cloud providers are: *Service Deployment, Service Orchestration, Cloud Service Management, Security and Privacy.*

2.1.6 Cloud Auditor

A cloud auditor is a party that can conduct independent assessment of cloud services, information system operations, performance, and security of a cloud implementation. A cloud auditor can evaluate the services provided by a cloud provider in terms of security controls, privacy impact, performance, etc. Auditing

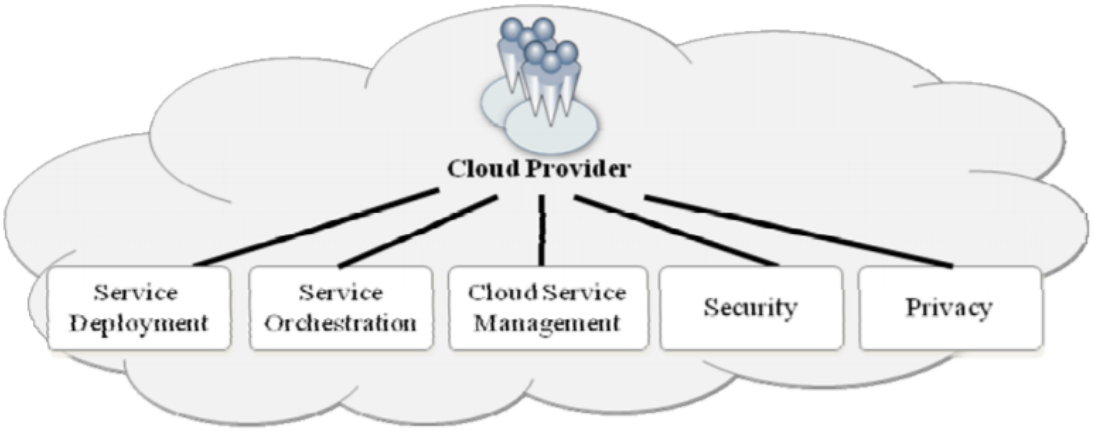


Figure 2.5: Cloud Provider: major activities

is especially important for federal agencies as "agencies should include a contractual clause enabling third parties to assess security controls of cloud providers" (by Vivek Kundra, Federal Cloud Computing Strategy, February 2011.). Security controls are the management, operational, and technical safeguards or countermeasures employed within an organizational information system to protect the confidentiality, integrity, and availability of the system and its information. For security auditing, a cloud auditor can make an assessment of the security controls in the information system to determine the extent to which the controls are implemented correctly, operating as intended, and producing the desired outcome with respect to the security requirements for the system. The security auditing should also include the verification of the compliance with regulation and security policy. Federal agencies should be aware of the privacy concerns associated with the cloud computing environment where data are stored on a server that is not owned or controlled by the federal government. Privacy impact auditing can be conducted to measure how well the cloud system conforms to a set of established privacy criteria. A privacy impact audit can help federal agencies comply with applicable privacy laws and regulations governing an individual's privacy, and to ensure confidentiality, integrity, and availability of an individual's personal information

at every stage of development and operation.

2.1.7 Cloud Broker

As cloud computing evolves, the integration of cloud services can be too complex for cloud consumers to manage. A cloud consumer may request cloud services from a cloud broker, instead of contacting a cloud provider directly. A cloud broker is an entity that manages the use, performance, and delivery of cloud services and negotiates relationships between cloud providers and cloud consumers. In general, a cloud broker can provide services in three categories:

Service Intermediation: a cloud broker enhances a given service by improving some specific capability and providing value-added services to cloud consumers. The improvement can be managing access to cloud services, identity management, performance reporting, enhanced security, etc.

Service Aggregation: a cloud broker combines and integrates multiple services into one or more new services. The broker provides data integration and ensures the secure data movement between the cloud consumer and multiple cloud providers.

Service Arbitrage: is similar to service aggregation except that the services being aggregated are not fixed. Service arbitrage means a broker has the flexibility to choose services from multiple agencies. The cloud broker, for example, can use a credit-scoring service to measure and select an agency with the best score.

2.1.8 Cloud Carrier

A cloud carrier acts as an intermediary that provides connectivity and transport of cloud services between cloud consumers and cloud providers. Cloud carriers provide access to consumers through network, telecommunication, and other access devices. For example, cloud consumers can obtain cloud services through network

access devices, such as computers, laptops, mobile phones, mobile Internet devices (MIDs), etc. The distribution of cloud services is normally provided by network and telecommunication carriers or a transport agent, where a transport agent refers to a business organization that provides physical transport of storage media such as high-capacity hard drives. Note that a cloud provider will set up service level agreements (SLAs) with a cloud carrier to provide services consistent with the level of SLAs offered to cloud consumers, and may require the cloud carrier to provide dedicated and encrypted connections between cloud consumers and cloud providers.

2.2 REST

This paragraph starts with the origins of REST, explains the alternatives to use REST and the problems these alternatives present. It also introduces the fundamental ideas of REST, the main characteristics and why to consider it.

2.2.1 Background on WEB Services

In November 1990, Sir Tim Berners-Lee, actual Director of the World Wide Web Consortium, helped by Robert Cailliau, published a formal proposal presenting what we now know as the World Wide Web. Since then, the World-Wide Web global information initiative has been using HTTP. The Hypertext Transfer Protocol (HTTP), according to the IETF specification [20], is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, object-oriented protocol which can be used for many tasks through the extension of its request methods. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred. Berners-Lee was one of the authors of the HTTP/1.0 [21] and HTTP/1.1 [20] [19] standard specification. Another author was Roy Thomas

Fielding who, at the same time the RFC 2068 [19] was being developed, honed his model down to a set of principles, properties, and constraints [50]. Here was born the term REpresentational State Transfer (REST), introduced in 2000 in his doctoral dissertation [23]. Considering that the World Wide Web uses HTTP and the relation between REST and HTTP, REST can be considered as the set of principles underlying the Web. Today's "web service" architectures have forgotten or ignore the most important feature that makes the Web successful: its simplicity. REST tries to be faithful to the Web principles. The World Wide Web Consortium (W3C) is the main international standards organization for the World Wide Web (WWW or W3). Founded and headed by Sir Tim Berners-Lee, it is constituted of member organizations which maintain full-time staff for the purpose of working together in the development of standards for the WWW. Thw W3C, through the WSA (Web Services Architecture) document [16] defines a Web service as a software system designed to support interoperable machineto-machine interaction over a network. It has an interface described in a machineprocessable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards. The WSA does not specify how Web services are implemented, and imposes no restriction on how Web services might be combined. It just describes the minimal characteristics that are common to all Web services, and a number of characteristics that are needed by many of them. The WS-I (Web Services Interoperability) organization instead mandates both SOAP and WSDL in their definition of a web service. According to the W3C there are two classes of web services, REST-compliant Web services, and arbitrary Web services (Big Web Services according to [39]). Both classes use URIs to identify resources and use Web protocols (such as HTTP and SOAP 1.2) and XML data formats for messaging. SOAP 1.2 can be used in a consitent manner with REST. To extend Web Services

capabilities there are a number of standards, protocols, specifications mostly built on top of HTTP. These are known as the WS-* stack and include WS-Notification, WS-Security, WSDL and SOAP [39]. This WS-* stack is the one that has evolved so fast as it has complicated the simplicity of the Web. The WS stack delivers interoperability for both the Remote Procedure Call (RPC) and messaging integration styles.

SOAP once stood for Simple Object Access Protocol, now it is just an acronym that according to [18] is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. It uses XML (Extensible Markup Language) technologies to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols. SOAP 1.1, which is considered as the XML-RPC's larger sibling, exchanges messages over HTTP 1.1. The SOAP messages are used as an envelope where the application encloses whatever information needs to be sent [26]. The envelope contains a header and a body. XML Schema is used to describe the structure of the SOAP message, so that SOAP engines at the two endpoints can marshall and unmarshall the message content and route it to the appropriate implementation.

The **WSDL** (Web Services Description Language) Version 2.0 [17] provides a model and an XML format for describing Web services. WSDL 2.0 enables one to separate the description of the abstract functionality offered by a service from concrete details of a service description such as "how" and "where" that functionality is offered. This specification defines a language for describing the abstract functionality of a service as well as a framework for describing the concrete details of a service description. It also defines the conformance criteria for documents in this language.

REST As a consequence of the increasing complexity of the Big Web services, the RESTful "style" has been brought as an alternative solution to implement re-

mote procedure calls across the Web. Furthermore, as cited on [39]: "The web is based on resources, but Big Web Services don't expose resources. The Web is based on URIs and links, but a typical Big Web Service exposes one URI and zero links. The Web is based on HTTP, and Big Web Services hardly use HTTP's features at all. This isn't academic hair-splitting, because it means Big Web Services don't get the benefits of resource-oriented web services. They're not addressable, cacheable, or well connected, and they don't respect any uniform interface. (Many of them are stateless, though.) They're opaque, and understanding one doesn't help you understand the next one. In practice, they also tend to have interoperability problems when serving a variety of clients." Representational state transfer (REST) is an architectural style for distributed hypermedia systems such as the World Wide Web. To implement this RESTful architecture there is a set of guidelines called ROA, Resource-Oriented Architecture which are the rules to be followed for designing RESTful web services [23]. The REST architectural style is based on the following principles [39] [14] [8]: Resources, Addressability, Statelessness, Connectedness and Uniform Interface and Cacheability.

2.2.2 Resources

Everything that a service provides is a resource (customers, cars, pictures, invoices, e-stores, etc.). A resource may be a physical object or an abstract concept. Every Resource will have at least one URI (a unique id). If a piece of information does not have a URI, it is not a resource and it is not on the Web. Two resources cannot have the same URI, but two different URIs can point to the same data at the same moment. For example, we can have an URI to identify the last version and another for the particular version. When the last version is the specific version, both URIs are pointing to the same data. Some months later, when the new version is released, the last version URI will not be the same as the specific URI

anymore. The URI should have a structure. With that it will be predictable for the client and it will be possible for him to create its own entry into the service. That is not a RESTful rule, it is a good web design rule. A resource can be represented in any format, any media type, for example: XML, XHTML, JSON or CSV representation. Each representation will have its own address but avoiding the format for example: path/page3.xml, path/page3.html and so on. The Content-Type header tells the client the representation needed to display the entity-body. On the human web, a web browser tries to display it inline, and if not possible run an external program. On the programmable web, a web service client uses this to decide which parser to apply to the entity-body.

2.2.3 Addressability

Every resource should be addressable. A resource is addressable by means of URIs so we will have as many URIs as resources we have. With this addressability we can bookmark an specific page, you can mail the URI to someone else, it allows cacheability (the first time someone requests a URI, the cache will save a copy. The next time someone request that same URI, the cache will serve the saved copy). Consider the web application Google Maps. We type "Norra Stationsgatan 60, Stockholm" and we obtain the view of the address we have just typed in. The URI of the site "http://maps.google.se/" does not change. That does not mean that Google maps is not addressable, it means that the web application that we are using is not, but instead, the web service is. If we want to send a friend this map, we need to ask the application for the specific URI and it will show us an URI with all the information that our friend needs to see the same image we are looking at. Without addressability we cannot send the map we are looking at to a friend, we need to do a screenshot or explain to him the steps he has to follow.

2.2.4 Stalessness

Every HTTP request should happen in complete isolation. It means that when the client makes an HTTP request all the information required to process that request must be present. The service should never rely on information from previous requests. If some data from the previous request is needed, the client have to send it again since the server does not keep any information about the client. This makes the system more reliable, simpler and horizontally scalable. A client does a request to a server A; the client does another request but just in that moment the server A fails. Another server will serve the request since all the information that it needs to serve it is given when doing the request. It means that a web server is replaceable easily by another, easily fail-over makes the system scalable. In a system with a load balancer (LB), the LB just has to take into account which server will serve better the request not if that server has served the requests from that user before or not. This kind of LB becomes much simpler to implement.

2.2.5 Connectedness

RESTful services should guide clients from one state to another by sending links in the representation. Representations are hypermedia, documents that contain not just data, but links to other resources. Human web is well connected but programmable web was not until now. All the resources should be connected and linked. It allows the client to discover the interface by traversing hyperlinks between each of its resource representations.

2.2.6 Uniform Interface

The client interacts via the defined HTTP methods: GET, POST, PUT, DELETE. These methods define the things that can be done to a resource.

The GET method means retrieve whatever information (in the form of an

entity) is identified by the Request-URI [20]. In the case of RESTful web services, this information identified by the URI is a representation of a resource. GET is a safe and idempotent method. Safe means that it does not change the state of the server, it only retrieves information so it should not have side effects. Idempotent means that it can be applied multiple identical requests having the same effect as a single request. Since the HTTP protocol is a stateless protocol, being prescribed as safe should also be idempotent. Being idempotent allows to safely repeat GET on resources in failure cases. If the request succeeded, and the resulting resource is returned in the message body the appropriate response status is 200 (OK). If the resource does not exist the response status is 404 (Not Found). If the request message includes a Range header field, the semantics of the GET method change to a "partial GET". It reduces unnecesasry network usage by allowing partially-retrieved entities to be completed without transferring data already held by the client.

The POST method is used to request that a new resource need to be created with the data enclosed in the request as a new subordinate of the URI given. If the action performed does not result in a resource that can be identified by an URI, the appropriate response status is 200 (OK). If a resource has been created, the response should be 201 (Created) and contain an entity which describes the status of the request and refers to the new resource, and a Location header.

The PUT method requests that the enclosed entity be stored under the supplied Request-URI [20]. If the URI refers to an already existing resource, the enclosed entity should be considered as a modified version of the one on the server. If it does not point to an existing resource, and that URI is capable of being defined as a new resource by the requesting user agent, the origin server can create the resource with that URI. If a new resource is created, the response status is 201 (Created). If an existing resource is modified, a 200 (OK) response codes should be sent to indicate successful completion of the request. If the resource could

not be created or modified, an appropriate error response should be given that reflects the nature of the problem. The main difference between POST and PUT is reflected in the different meaning of the request. The URI in a PUT request identifies the entity enclosed with the request (the client indicates in the URI the id of the resource). In contrast, the URI in a POST request identifies the resource that will handle the enclosed entity. Moreover, the PUT method is idempotent, as well as the GET method, since making more than one PUT request to a given URI should have the same effect as making only one, but the POST method is not.

The DELETE method requests that the server deletes the resource identified by the request. It is also an idempotent method as the PUT and GET methods. A successful response should be 200 (OK) if the action has been performed, if the action could not be performed, an appropriate error response should be given that reflects the nature of the problem.

2.2.7 Cacheable

The resources should be cachable whenever possible with an expiration date and time. Cacheability offers better user experience since the load on the server is decreased and with that, a better response and loading time is achieved. There are two types of caching mechanisms in HTTP: Expiration and Validation. According to RFC2616 [20], the goal of caching in HTTP/1.1 is to eliminate the need to send requests in many cases, which reduces the number of network round-trips required for many operations and, to eliminate the need to send full responses in many other cases, which reduces network bandwidth requirements. The former is done through the ”Expiration” mechanism; the latter through the ”Validation” mechanism. The responses to the POST method are not cacheable, unless the response includes appropriate Cache-Control or Expires header fields (Expiration).

The conditional GET method is intended to allow cached entities to be refreshed without requiring multiple requests or transferring data already held. The GET method becomes conditional GET if the request messages includes an Etag / If-None-Match, Last-Modified / If-Modified-Since.

Expiration

Expiration is a way for the server to say how long a requested resource stays fresh for. It is excellent for resources that change very rarely (i.e. images) or at known time.

- Expires header: the HTTP Expires header just says the date and time when the resource will become stale. It requires the server's and the client's clock to be synchronized.

- Cache-Control Header: HTTP 1.1 replaced the Expires header by this one which solves the syncronization problem from the Expires header and offers more flexibility. The cache-control header has a number of clauses that can be used to control the way the client caches the resource.

Validation

Validation allows a client to ask the server whether a cached version of a resource is still fresh.

- Last-Modified / If-Modified-Since: this header makes conditional HTTP GET possible. When doing a GET request, the Last-Modified header value tells the client the last time the representation changed. The client can keep track of this date and send it in the If-Modified-Since header of a future request. When the server receives a GET request that has the If-ModifiedSince header defined, it checks that the resource has not changed from the time specified in the mentioned header. If it has not, the server will send a response code of 304 ("Not modified") with no entity-body. If it has, then

the server will realize that given the fact that the If-Modified-Since date is previous to the one that the server internally has. Therefore, the server will send the new entity-body and will fill the Last-Modified header with the time in which the resource was lastly modified. Since Last-Modified has a one-second accuracy it is not accurate enough. Occasionally a conditional HTTP GET gives the wrong result only taking into account the If-Modified-Since header. For this reason it is also advisable to use the ETag and If-None-Match headers.

- ETag / If-None-Match: the value of ETag is used to determine if a representation has changed. If the respresentation changes, the ETag should also change. The ETag header ought to be sent in response to GET requests. The value of ETag can be used as the value of a If-None-Match header in a future conditional GET request. If the representation has not changed, the condition fails since the ETag is the same. The server sends a response code of 304 ("Not Modified") with no entity-body, so the server can save time and bandwidth not sending it. If the ETag has changed, the condition succeeds and the server sends the new entity-body. The ETag header is a second line of defense, the conditional GET main drivers are the Last-Modified and the If-Modified-Since response headers. If the representation changes twice in one second, the Last-Modified header will not change but it will do it the ETag header.

2.2.8 Why REST?

Given the principles explained in the last section, a set of benefits can be obtained following the REST architectural style. Firstly, each physical object or abstract concept at which we can refer is identified as a unique resource. This implies that each of them can be quickly accesed by just one request once they are identified.

This results in a better navigability and experience for the user and less load for the server since each resource can be accessed with just one request. Secondly, due to the statelessness principle, REST makes the system really scalable since the servers do not keep any information from any of the clients. This means that every single machine that is part of the system is able to receive any kind of request from any of the clients and generate a proper response. Other effects of this principle are that complex load balancers are not needed anymore and that the clients that are using the system when a server goes down do not notice it, since the system becomes fault-tolerant and recoverable. Furthermore, with a no RESTful web service it can be really difficult to discover the interface because the resources may not be addressable, they may not be connected between each other and the HTTP methods used in each request may not follow a standard pattern. Following the REST style, the client is able to guess how the interface is designed and go directly to the information needed using a specific URI or it may navigate through the representations using the links they contain resulting in a really good user experience. Lastly, cacheability, which can be easily achieved by using the HTTP headers properly, reduces the load on the servers and improves the response time which is translated again in a better user experience. All these benefits make REST an interesting and attractive architectural style that is worth to try out since it can really improve a web service in many ways.

2.2.9 RESTful APIs

API (application programming interface) is a set of rules and mechanisms by which one application or component interacts with the others. It seems that the name speaks for itself, but let's get deeper into details.

API can return data that you need for your application in a convenient format (e.g. JSON or XML).

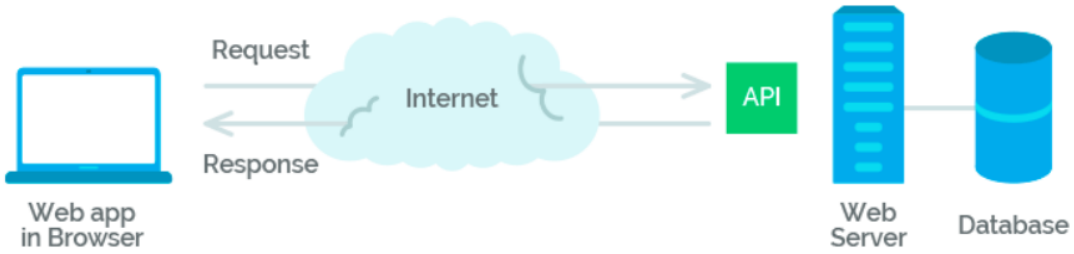


Figure 2.6: API: how it works

Let's look at an example. You are probably familiar with the web-application Github. It has its own API, with the help of which you can get information about users, their repositories and much more useful things when developing your app. You can use and manipulate this data in your project.

The example of a standard request to API looks like this:

```
➜ ~ curl https://api.github.com/orgs/MLSDev
```

Figure 2.7: Example Github Request

The request is performed with the help of CURL which is a console utility. There are also browser utilities like Postman, REST Client, etc. We can see the response in Figure 2.8.

A simple definition of RESTful API can easily explain the notion. REST is an architectural style, and RESTful is the interpretation of it. That is, if your back-end server has REST API and you make client-side requests (from a website/application) to this API, then your client is RESTful. RESTful API best practices come down to four essential operations: receiving data in a convenient format, creating new data, updating data, deleting data (you can see REST API Design in Figure 2.9).

REST relies heavily on HTTP. Infact, each operation uses its own HTTP

```
{
  "login": "MLSDev",
  "id": 1436035,
  "url": "https://api.github.com/orgs/MLSDev",
  "repos_url": "https://api.github.com/orgs/MLSDev/repos",
  "events_url": "https://api.github.com/orgs/MLSDev/events",
  "hooks_url": "https://api.github.com/orgs/MLSDev/hooks",
  "issues_url": "https://api.github.com/orgs/MLSDev/issues",
  "members_url": "https://api.github.com/orgs/MLSDev/members{/member}",
  "public_members_url": "https://api.github.com/orgs/MLSDev/public_members{/member}",
  "avatar_url": "https://avatars.githubusercontent.com/u/1436035?v=3",
  "description": "",
  "name": "MLSDev",
  "company": null,
  "blog": "http://mlsdev.com/",
  "location": "Ukraine",
  "email": "hello@mlsdev.com",
  "public_repos": 25,
  "public_gists": 0,
  "followers": 0,
  "following": 0,
  "html_url": "https://github.com/MLSDev",
  "created_at": "2012-02-14T08:35:16Z",
  "updated_at": "2015-10-29T13:48:13Z",
  "type": "Organization"
}
```


Figure 2.8: Example Github Response

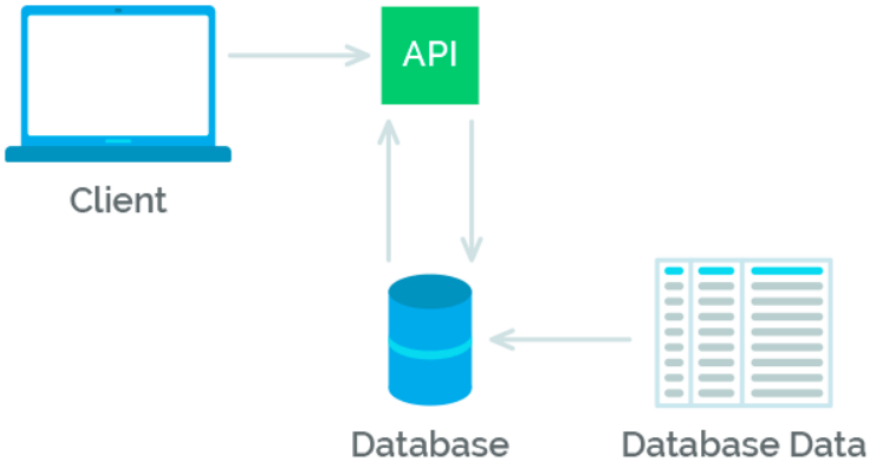


Figure 2.9: REST API Design

method: GET - getting; POST - creation; PUT - update (modification); DELETE - removal. All these methods (operations) are generally called CRUD. They man-

age data or as Wikipedia says, "create, read, update and delete" it. The fact that REST contains a single common interface for requests and databases is its great advantage. This can be viewed in the table in Figure 2.10.

Figure 2.10: Single common interface

All requests you make have their HTTP status codes. There are a lot of them and they are divided into 5 classes. The first number indicates which of them a code belongs to:

- 1xx - informational;
- 2xx - success;
- 3xx - redirection;
- 4xx - client error;
- 5xx - server error

You should always provide versioning for your REST API. For example, if the API is at the URL http://example.com/api, it is necessary to make changes to it at http://example.com/api/v1.

Today the back end is usually developed not only for web platforms, but also for mobile applications. Therefore, if you create an API without a version and change something on the server, the web will still be updated without problems. But there may be issues with mobile apps. Even if you make changes, there is no guarantee that a user will accept them on their device. You have probably seen

how many unconfirmed updates can wait their turn on the phone. So, it is very logical to support both old and new API versions.

There are two options for specifying versions. The first one, which is indicating them in the URL, has already been described. The second option is about including a version in a request header. In general, the former solution is widely criticized. There are a lot of arguments like «THIS IS BAD!!!!!», but they do not seem convincing. In fact, everyone chooses the option that works best for them.

All resources in REST are entities. They can be independent like:

- GET /users - get all users;
- GET /users/123 - get a particular user with id = 123;
- GET /posts - get all posts.

There are also dependent entities, that rely on their parent models:

- GET /users/123/projects - get all the projects that a user with id = 123 has.

The above examples show that GET implies getting the entity you request. It is which means receiving identical data while performing the same requests. A successful request returns an entity representation combined with status code 200 (OK). If there is an error you will get back code 404 (Not Found), 400 (Bad Request) or 5xx (Server Error).

Let's move to the POST method (the creation of a new entity):

- POST /users - when creating a new entity you set parameters in the request body.

The POST request is not idempotent, which means that if you send the same data in the repeat request, it will create an entity duplicate but with a different

identifier.After that you will get the result which may be status code 200 (OK), for example. Then the response will contain the data of a saved entity.

It can also return status 201 (Created), resulting in the creation of a new entity. The server can specify its address in the response body. It is recommended to indicate it in the header "Location".

The next request is PUT. It is used to update entities. When you send it the request body should contain updated entity data it refers to.

- PUT /users/123 - upgrade a user entity with id = 123. The changes need to be indicated in the parameters. If updated successfully the request returns code 200 (OK) and the representation of the updated entity.

The last request is DELETE. It is simple to understand and is used to remove a specific entity according to an identifier.

- DELETE /users/123 - delete a user with id = 123.

If the removal is successful it returns 200 (OK) together with the response body that contains information about the status of the entity. For example, when you do not remove the entity from the database but just mark it as deleted, repeated requests will always return 200 (OK) and the response body with a status. DELETE can be considered as an idempotent request. It can also return code 204 (No Content) without the response body.

If you delete the entity from the database completely, the second request should return 404 (Not Found) because that resource is already removed and no longer accessible.

There is a great and quite widespread open-source framework called Swagger.io. It makes the life of developers easier in terms of documentation, description and even initial testing.

You can see all requests for the user there, in Figure 2.11:

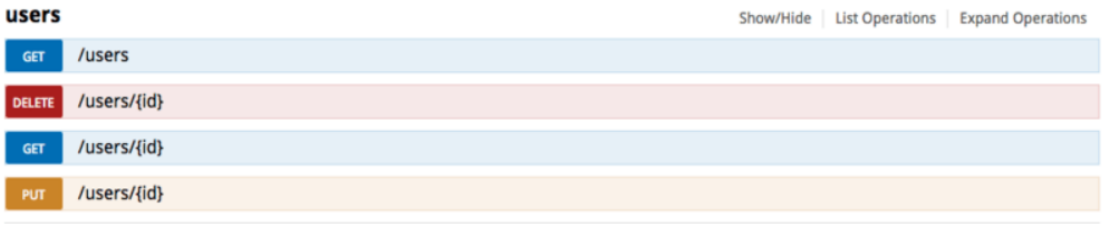


Figure 2.11: Swagger UI interface

It also provides an opportunity to view the description of each request model. You can fill the fields with data, send it and get real results 2.12.

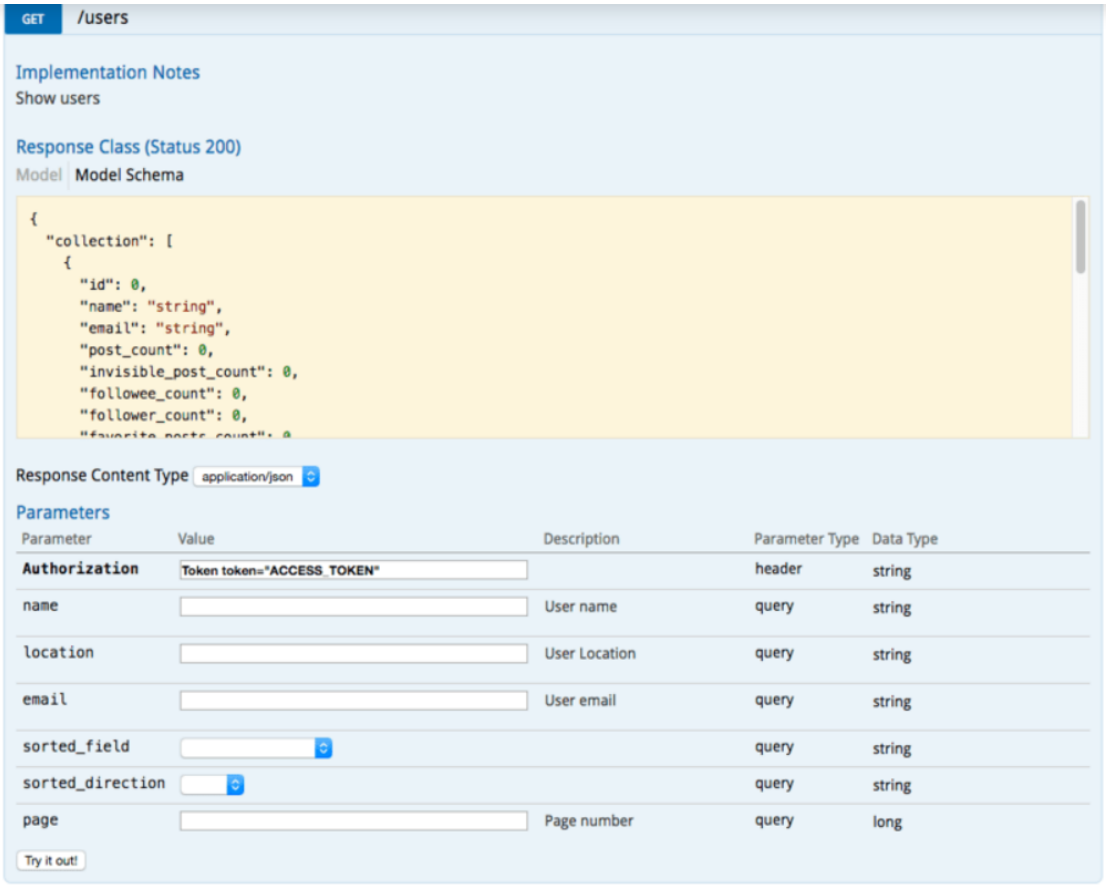


Figure 2.12: Swagger UI interface: more details

2.3 Ontology

Ontologies and semantic technologies are becoming popular again. They were a hot topic in the early 2000s, but the tools needed to implement these concepts were not yet sufficiently mature. Ontology and semantic technologies have now matured to the point where they are widely available and reasonably priced. This has potentially vast benefits for organizations that are seeking to improve the use and reuse of their structured and unstructured information and want to maximize findability and discoverability. But, "What is an ontology and why do I want one?"

2.3.1 Ontology definition

An ontology is "a defined model that organizes structured and unstructured information through entities, their properties, and the way they relate to one another." Many of you are familiar with terms like taxonomies and metadata. Think of an ontology as another way to classify content (like a taxonomy) that allows you to relate content based on the information in it as opposed to a term describing it. For example, a company could creates an ontology about its employees and consultants (see Figure 2.13). This example shows an ontology of a company, its employees, consultants, and the projects they are working on. In this example, Kat Thomas is a consultant who is working with Bob Jones on a Sales Process Redesign project. Kat works for Consult, Inc and Bob reports to Alice Reddy. You can infer a lot of information through this ontology. Since the Sales Process Redesign is about sales you can infer that Kat Thomas and Bob Jones have expertise in sales. Consult, Inc must provide expertise in this area as well. You also know that Alice Reddy is likely responsible for some aspect of sales at Widgets, Inc because her direct report is working on the Sales Process Redesign project.

There are many reasons why this is valuable. Ontologies can allow to:

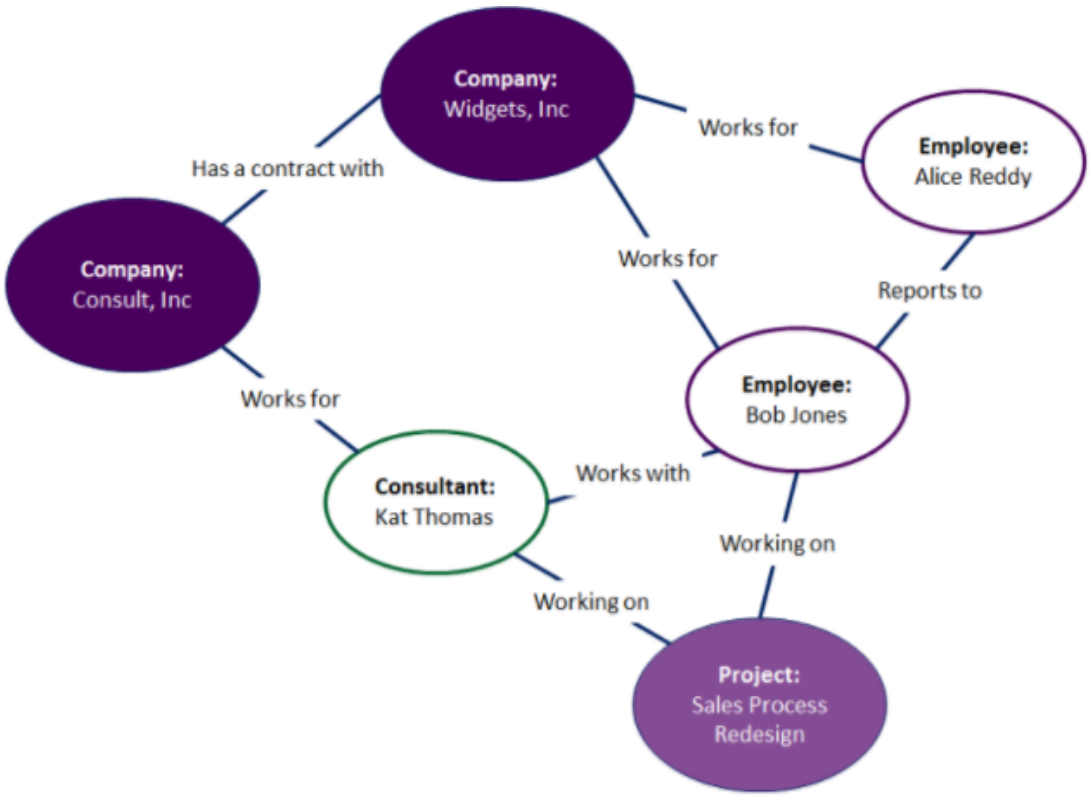


Figure 2.13: Company Ontology example

- Manage content more effectively;
- Maximize findability and discoverability of information;
- Increase the reuse of "hidden" and unknown information;
- Elevate SEO on external search engines. SEO is the set of strategies and practices aimed at increasing the visibility of a website by improving its position in the rankings of search engines.

2.3.2 Manage content more effectively

Content management is a time consuming process. It is one thing to manage metadata on a couple thousand pieces of content. What if you are managing hundreds of thousands of pieces of content? Ontologies are focused on relationships

between entities. To extend the example above, I can identify content authored by Kat Thomas or Bob Jones and associate it with Sales information because Sales Process Redesign project is about sales. I no longer need to manually tag this content as I can rely on the entities in the content and the information I have about them.

2.3.3 Improved findability

Ontologies give you new ways to find and discover content. Ontologies can power faceted search or allow people to browse through related content based on the people, places, and things that are mentioned in the text. I can see all of the deliverables created by Consultant, Inc and see all of the deliverables they have provided to my company. I can also see who works for them and who they have worked with at my company. I am navigating based on things that I understand to find relevant content and information.

Ontologies also allow for more accuracy in the way content is classified as opposed to classic metadata. For example, a piece of content on our intranet quotes Kat Thomas who at the time was an outside consultant. If we used the metadata approach her content might be tagged as consultant information. A year later Kat takes a job as head of sales. Using an ontology her recommendations would show up as recommendations from the head of sales. If I was just relying on metadata, I would have to go back and update my content to reflect her new position.

2.3.4 Greater Content Reuse

Publishers use ontologies to group content in new ways. The best example of this is the New York Times Topic Pages. All of the content related to famous people or topics are grouped on a single page. These articles appeared in the paper, but

are now reused as a single place to learn all about a specific topic.

Content reuse is not limited to content on your site. Because ontologies are standards based, other sites can use your content to augment their content. As a result, your content appears in more places and is more likely to be seen by others.

2.3.5 Improved SEO

Ontologies are machine readable. This means that search engines are able to understand the content. As a result, content based on ontologies rates higher in Google and other search engines. Wordlift is a great example of an ontology plug-in for WordPress that promises to improve SEO.

2.3.6 Web Ontology Language

The Web Ontology Language (OWL) is a family of knowledge representation languages for authoring ontologies. The Web Ontology Language (OWL) is a standard from [51], based on XML, RDF and RDFS. With OWL complex relationships and constraints can be represented in ontologies [44]. For this work, we refer to the RFD Schema.

The following OWL Lite features related to RDF Schema are included:

- Class: a class denotes a group of individuals that belong together because they share some properties.
- rdfs:subClassOf : class hierarchies may be created by making one or more statements that a class is a subclass of another class.
- rdf:Property: properties can be used to state relationships between individuals or from individuals to data values.
- rdfs:subPropertyOf : property hierarchies may be created by making one or more statements that a property is a subproperty of one or more other

properties. Both owl:ObjectProperty (relation between two instances of two classes) and owl:DatatypeProperty (relation between the instance of a class and the istance of a datatype) are subclasses of the RDF class rdf:Property.

- rdfs:domain: a domain of a property limits the individuals to which the property can be applied. If a property relates an individual to another individual, and the property has a class as one of its domains, then the individual must belong to the class.

- rdfs:range: the range of a property limits the individuals that the property may have as its value. If a property relates an individual to another individual and the property has a class as its range, then the other individual must belong to the range class.

- Individual: individuals are instances of classes, and properties may be used to relate one individual to another.

2.4 Natural Language Processing

2.4.1 Introduction

Natural language processing is a set of techniques that allows computers and people to interact. Consider the process of extracting information from some data generating process: A company wants to predict user traffic on its website so it can provide enough compute resources (server hardware) to service demand. Engineers can define the relevant information to be the amount of data requested. Because they control the data generating process, they can add logic to the website that stores every request for data as a variable. Then, they can define the unit of measurement as the amount of data requested as a byte, in turn allowing us to represent the information as integers. With an excellent representation of the

information in hand, the engineers can store it in a tabular database so analysts can make predictions based on this historical data.

Natural language processing is the application of the steps above — defining representations of information, parsing that information from the data generating process, and constructing, storing, and using data structures that store information — to information embedded in natural languages.

What makes a language natural is precisely what makes natural language processing difficult; the rules governing the representation of information in natural languages evolved without predetermination. These rules can be high level and abstract, such as how sarcasm is used to convey meaning; or quite low level, such as using the character "s" to denote plurality of nouns. Natural language processing involves identifying and exploiting these rules with code to translate unstructured language data into information with a schema. Language data may be formal and textual, such as newspaper articles, or informal and auditory, such as a recording of a telephone conversation. Language expressions from different contexts and data sources will have varying rules of grammar, syntax, and semantics. Strategies for extracting and representing information from natural languages that work in one setting often fail in others.

2.4.2 Business Uses

Companies often have access to records of natural language that contain valuable information. Product reviews or even tweets on Twitter can contain specific complaints or feature requests related to a product that can help prioritize and evaluate proposals. Online marketplaces may have item descriptions available that can help define a taxonomy of products. A digital newspaper may have an archive of online articles that can be used to build a search engine to allow users to find relevant content. Information that is representational of natural language

can also be useful for building powerful applications, such as **bots that respond to questions** or software that **translates from one language to another** or **sentiment analysis** (an example in Figure 2.14)

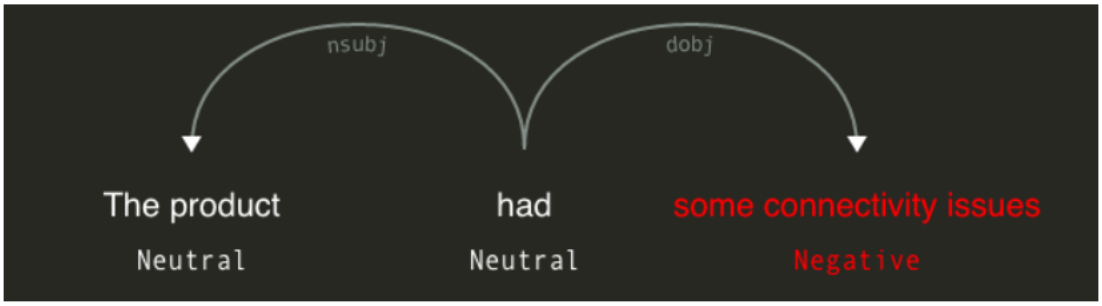


Figure 2.14: Natural language processing can be used to identify specific complaints from text.

Projects requiring natural language processing are generally organized by these sorts of challenges. Solving them usually requires to serially piece multiple subtasks together, where there may be many approaches for each subtask. The universe of natural language processing methods can be daunting, as it's highly specialized, vast, and somewhat lacking in an overarching conceptual framework. While a complete summary of natural language processing is well beyond the scope of this paragraph, this will covers some concepts that are commonly used in general purpose natural language processing work.Let's assume that we have access to textual data with which to work (not auditory, which requires the additional step of **speech recognition**).

2.4.3 Identifying Lexical Units

As natural languages are generally composed of words, an initial step of many natural language processing projects is identifying words within some raw text. The concept of a word, however, may be too restrictive or ambiguous. The strings "cats" and "cat" are different forms of the same entry in the dictionary; should they be treated equivalently? "Star Wars" has no entry in the dictionary, and

though it contains a space, we think of it a singular entity. These are the sorts of challenges involved in defining lexical units, which represent basic elements of a vocabulary.

For a given task, the researcher must define what constitutes an appropriate lexical unit. Should singular and plural forms of a word be considered to belong to the same lexical unit? Assume we are building a question answering system, and receive the following queries:

A: "Find the closest theaters to me."

B: "Find the closest theater to me."

In A, the user is implying that he wants to view multiple theaters, whereas in B, he just wants the single closest theater. Throwing away the distinction between singular and plural will degrade the quality of our application.

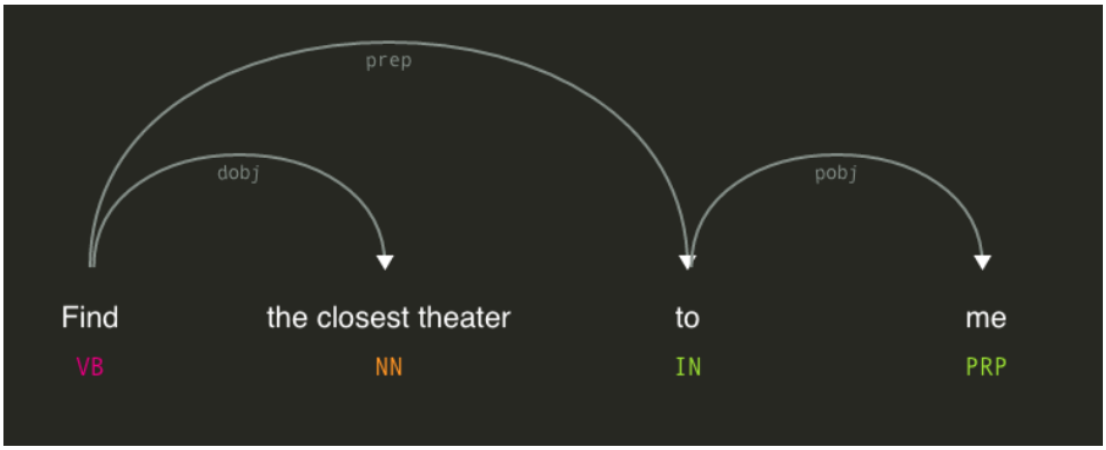


Figure 2.15: Phrase analysis example:grammatical and dependency

Alternatively, assume that we want to summarize the following product reviews:

A: "The product had some connectivity problems."

B: "The product had a connectivity problem."

Here, the distinction between "problems" and "problem" is not relevant.

While not at all exhaustive, **tokenization** and **normalization** are two com-

mon steps to parsing lexical units from natural language. Tokenization is the process of separating a sequence of characters into tokens, each of which represents an instance of a term. Normalization is the set of steps we take to condense terms into lexical units.

Tokenization algorithms can be simple and deterministic, such as separating characters into tokens every time a character is not alphanumeric. Non-alphanumeric characters could be spaces, punctuation marks, hashtags, etc. We can implement this approach with pattern matching, checking for the presence of characters or character sequences according to some rules where we define these patterns as regular expressions (often abbreviated "regex"). For example, we can denote the set of all numeric characters with regex string "0-9" or "$_:$" We can combine terms with modifiers into complex search patterns. Application of regular expressions is so widespread that implementations are available in almost all modern programming languages.

It's easy to find examples where this strategy will fail ("didn't $\neq$ ["didn", "t"]). In some applications, researchers capture these patterns with multiple complex regex queries and morphology-specific rules, and pass the text input through a finite state machine to determine the correct tokenization. Encoding an exhaustive set of rules can be difficult, and will depend on the application and the type of text data under analysis (though there are some partial rulesets that have been assembled). To adapt to a new corpus, tokenizers can be built by training statistical models on hand-tokenized text, though this approach is rarely used in practice due to the success of deterministic approaches. These models may look at sequences of character properties, such as whether the contiguous alphanumeric begins with a capital letter, and model tokenizations as a function of these properties. The rules the model learns to use will depend on the provided training corpus (Twitter, medical articles, etc.).

Another challenge is that multi-term character sequences can comprise a lex-

ical unit. These multi-term sequences could be named entities such as "New York," in which we case, we are doing **named entity recognition** or simply common phrases. One approach to this problem is to allow for some redundancy in our representation by including all n-length sets of terms (n-grams) as tokens. If we limit ourselves to unigrams and bigrams, New York would be tokenized as "New,""York,""New York." To add some sophistication instead of exhausting all n-grams, we could select the highest order n-gram representation of a set of terms subject to some condition, like whether it exists in a hard-coded dictionary (called a gazetteer) or if it is common in our dataset. Alternatively, we could employ machine learning models. If the probability of a given word, p(wordi | wordi-1), is high enough given the preceding word, we might consider the sequence a two-term lexical unit (alternatively, we could use the formulation p(wordi | wordi-1 = bigram) using labeled examples). Though smoothing can help ameliorate the problem, these language models tend to have trouble generalizing, and require some amount of transfer learning, feature engineering, determinism, or abstraction. Probabilistic n-gram models require labeled examples, machine learning algorithms, and feature extractors (the latter two are bundled in Stanford's NER software). If such models are not worth the investment, high quality pretrained models can often be used successfully on new data sets.

After tokenization, we may wish to normalize our tokens. Normalization is a set of rules that aim to reduce all instances of a lexical equivalence class to their canonical form.

This may include procedures like: *plural* -> singular (e.g. cats -> cat); *past tense* -> present tense (e.g. ran -> run); *adverb form* -> adjective form (quickly -> quick); *hyphenation* -> concatenation.

One approach to normalization is to reduce a word to its dictionary form through a process called **lemmatisation**. The lemma of a term has all the metadata contained in a dictionary entry: part of speech, definition (word sense), and

stem. Lemmatisation traditionally requires a morphological parser, in which we completely featurize some unprocessed term (tense, plurality, part of speech, etc.) based on its morphological elements (prefix, suffix, etc.). To build a parser, we create a dictionary of known stems and affixes (lexicon) with metadata about them, like possible parts of speech, enumerate the rules (morphotactics) governing how morphemes can be compiled together (plural modifier "-s" must follow the noun, for example), and finally enumerate rules (orthographic rules) that govern changes in a word under different morphological states (for instance, a past tense verb ending in "-c" must have a "k" added, such as "picnic -> picnicked"). These rules and terms are passed to a finite state machine that pass over some input, maintaining a state or set of feature values that is then updated as rules and lexicon are checked against the text (similar to how regular expressions work).

The degree of specificity in how we define the equivalence class (which morphological elements are relevant) and, thus, the degree of normalization we use and morphological metadata we extract depends on our application. In information retrieval, we often only care about the high-level semantics of some text. All morphemes, or meaning-conveying elements of the word other than the stem, can be discarded along with all morphological metadata (such as tense). Stemming algorithms usually use substitution rules, such as:

- if a given stem contains a vowel, we remove "-ing" from all tokens that contain the stem ("barking" $\rightarrow$ "bark" but "string" $\rightarrow$ "string")

Ultimately, the goals of building a preprocessing pipeline include: all relevant information is extracted; all irrelevant information is discarded; text is represented in sufficiently few equivalence classes; the data is in a useful form, such as lists of ascii-encoded strings or as abstract data structures such as sentences or documents.

The definition of relevant, sufficient, and useful depends on the requirements of the project, the strength of the development team, and availability of time and

resources. Furthermore, in many cases, the strength of a preprocessing pipeline can determine the overall success of a project. Therefore, care should be taken while building a pipeline.

After translating raw text into a string or tokenized array of lexical units, the researcher or developer may take steps to preprocess his or her text data, such as string encoding, stop word and punctuation removal, spelling correction, part-of-speech tagging, chunking, sentence segmentation, and syntax parsing. The table in Figure 2.16 illustrates some examples of the types of processing a researcher may use given a task and some raw text.

Raw Text	Processed	Steps	Task	How Pipeline Suits Task
She sells seashells by the seashore.	['she','sell','seashell','seashore']	tokenization, lemmatization, stop word removal, punctuation removal	topic modeling	We only care about high level, thematic, and semantically heavy words
John is capable.	John/PROPN is/VERB capable/ADJ ./PUNCT	tokenization, part of speech tagging	named entity recognition	We care about every word, but want to indicate the role each word plays to build a list of NER candidates
Who won? I didn't check the scores.	[u'who', u'win'], [u'i',u'do',u'not',u'check',u'score']	tokenization, lemmatization, sentence segmentation, punctuation removal, string encoding	sentiment analysis	We need all words, including negations since they can negate positive statements, but don't care about tense or word form

Figure 2.16: Types of processing by task

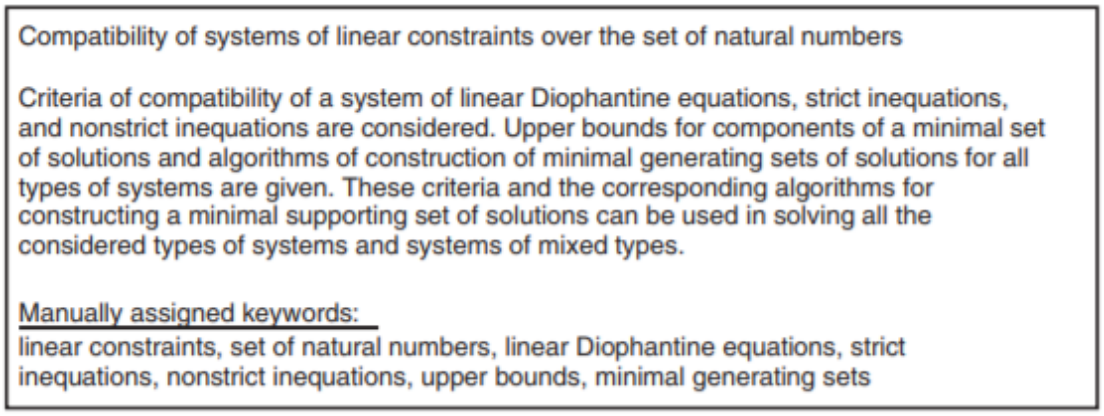
2.4.4 Keywords extraction

Keywords, which we define as a sequence of one or more words, provide a compact representation of a document's content. Ideally, keywords represent in condensed form the essential content of a document. Keywords are widely used to define queries within information retrieval (IR) systems as they are easy to define, revise, remember, and share. In comparison to mathematical signatures, keywords are independent of any corpus and can be applied across multiple corpora and IR systems. Keywords have also been applied to improve the functionality of IR systems.

Despite their utility for analysis, indexing, and retrieval, most documents do not have assigned keywords. Most existing approaches focus on the manual assignment of keywords by professional curators who may use a fixed taxonomy, or rely on the authors' judgment to provide a representative list. Research has therefore focused on methods to automatically extract keywords from documents as an aid either to suggest keywords for a professional indexer or to generate summary features for documents that would otherwise be inaccessible. While some keywords are likely to be evaluated as statistically discriminating within the corpus, keywords that occur in many documents within the corpus are not likely to be selected as statistically discriminating. Corpus-oriented methods also typically operate only on single words. This further limits the measurement of statistically discriminating words because single words are often used in multiple and different contexts. To avoid these drawbacks, we focus our interest on methods of keyword extraction that operate on individual documents. Such document-oriented methods will extract the same keywords from a document regardless of the current state of a corpus. Document-oriented methods therefore provide contextindependent document features.In the following section, we describe Rapid Automatic Keyword Extraction (RAKE), an unsupervised, domain-independent,

and language-independent method for extracting keywords from individual documents [55].

RAKE RAKE develop motivation has been to develop a keyword extraction method that is extremely efficient, operates on individual documents to enable application to dynamic collections, is easily applied to new domains, and operates well on multiple types of documents, particularly those that do not follow specific grammar conventions. Example below contains the title and text for a typical abstract, as well as its manually assigned keywords.


Compatibility of systems of linear constraints over the set of natural numbers

Criteria of compatibility of a system of linear Diophantine equations, strict inequations, and nonstrict inequations are considered. Upper bounds for components of a minimal set of solutions and algorithms of construction of minimal generating sets of solutions for all types of systems are given. These criteria and the corresponding algorithms for constructing a minimal supporting set of solutions can be used in solving all the considered types of systems and systems of mixed types.

Manually assigned keywords:
linear constraints, set of natural numbers, linear Diophantine equations, strict inequations, nonstrict inequations, upper bounds, minimal generating sets

RAKE is based on the observation that keywords frequently contain multiple words but rarely contain standard punctuation or stop words, such as the function words and, the, and of , or other words with minimal lexical meaning. Reviewing the manually assigned keywords for the abstract in th example, there is only one keyword that contains a stop word (of in set of natural numbers). Stop words are typically dropped from indexes within IR systems and not included in various text analyses as they are considered to be uninformative or meaningless. This reasoning is based on the expectation that such words are too frequently and broadly used to aid users in their analyses or search tasks. Words that do carry meaning within a document are described as content bearing and are often referred to as content words. The input parameters for RAKE comprise a list of stop words (or stoplist), a set of phrase delimiters, and a set of word delimiters. RAKE uses stop words

and phrase delimiters to partition the document text into candidate keywords, which are sequences of content words as they occur in the text. Co-occurrences of words within these candidate keywords are meaningful and allow us to identify word cooccurrence without the application of an arbitrarily sized sliding window. Word associations are thus measured in a manner that automatically adapts to the style and content of the text, enabling adaptive and fine-grained measurement of word co-occurrences that will be used to score candidate keywords.

Candidate keywords RAKE begins keyword extraction on a document by parsing its text into a set of candidate keywords. First, the document text is split into an array of words by the specified word delimiters. This array is then split into sequences of contiguous words at phrase delimiters and stop word positions. Words within a sequence are assigned the same position in the text and together are considered a candidate keyword. You can see below the candidate keywords in the order that they are parsed from the sample technical abstract shown in the example.

Compatibility – systems – linear constraints – set – natural numbers – Criteria – compatibility – system – linear Diophantine equations – strict inequations – nonstrict inequations – Upper bounds – components – minimal set – solutions – algorithms – minimal generating sets – solutions – systems – criteria – corresponding algorithms – constructing – minimal supporting set – solving – systems – systems

The candidate keyword /textitlinear Diophantine equations begins after the stop word *of* and ends with a comma. The following word *strict* begins the next candidate keyword *strict inequations*.

Keyword scores After every candidate keyword is identified and the graph of word co-occurrences, shown below,is complete a score is calculated for each candidate keyword and defined as the sum of its member word scores. There are evaluated several metrics for calculating word scores,based on the degree and

	algorithms	bounds	compatibility	components	constraints	constructing	corresponding	criteria	diophantine	equations	generating	inequations	linear	minimal	natural	nonstrict	numbers	set	sets	solving	strict	supporting	system	systems	upper
algorithms	2						1																		
bounds		1																							1
compatibility			2																						
components				1																					
constraints					1								1												
constructing						1																			
corresponding	1						1																		
criteria								2																	
diophantine									1	1			1												
equations									1	1			1												
generating											1			1					1						
inequations												2				1					1				
linear					1				1	1			2												
minimal											1			3				2	1			1			
natural															1		1								
nonstrict												1				1									
numbers															1		1								
set														2				3				1			
sets											1			1					1						
solving																				1					
strict												1									1				
supporting														1				1				1			
system																							1		
systems																								4	
upper		1																							1

frequency of word vertices in the graph: word frequency $freq(w)$, word degree $deg(w)$, and ratio of degree to frequency $\frac{deg(w)}{freq(w)}$. The metric scores for each of the content words in the sample abstract are listed below.

	algorithms	bounds	compatibility	components	constraints	constructing	corresponding	criteria	diophantine	equations	generating	inequations	linear	minimal	natural	nonstrict	numbers	set	sets	solving	strict	supporting	system	systems	upper
deg(w)	3	2	2	1	2	1	2	2	3	3	3	4	5	8	2	2	2	6	3	1	2	3	1	4	2
freq(w)	2	1	2	1	1	1	1	2	1	1	1	2	2	3	1	1	1	3	1	1	1	1	1	4	1
deg(w) / freq(w)	1.5	2	1	1	2	1	2	1	3	3	3	2	2.5	2.7	2	2	2	2	3	1	2	3	1	1	2

In summary, $deg(w)$ favors words that occur often and in longer candidate keywords; $deg(minimal)$ scores higher than $deg(systems)$. Words that occur frequently regardless of the number of words with which they co-occur are favored by $freq(w)$; $freq(systems)$ scores higher than $freq(minimal)$. Words that predominantly occur in longer candidate keywords are favored by $\frac{deg(w)}{freq(w)}$; $\frac{deg(diophantine)}{freq(diphantine)}$ scores higher than $\frac{deg(linear)}{freq(linear)}$. The score for each candidate keyword is computed

as the sum of its member word scores. Below you can see a list of each candidate keyword from the sample abstract using the metric $\frac{deg(w)}{freq(w)}$ to calculate individual word scores.

minimal generating sets (8.7), linear diophantine equations (8.5), minimal supporting set (7.7), minimal set (4.7), linear constraints (4.5), natural numbers (4), strict inequations (4), nonstrict inequations (4), upper bounds (4), corresponding algorithms (3.5), set (2), algorithms (1.5), compatibility (1), systems (1), criteria (1), system (1), components (1),constructing (1), solving (1)

Adjoining keywords Because RAKE splits candidate keywords by stop words, extracted keywords do not contain interior stop words. While RAKE has generated strong interest due to its ability to pick out highly specific terminology, an interest was also expressed in identifying keywords that contain interior stop words such as axis of evil. To find these RAKE looks for pairs of keywords that adjoin one another at least twice in the same document and in the same order. A new candidate keyword is then created as a combination of those keywords and their interior stop words. The score for the new keyword is the sum of its member keyword scores. It should be noted that relatively few of these linked keywords are extracted, which adds to their significance. Because adjoining keywords must occur twice in the same order within the document, their extraction is more common on texts that are longer than short abstracts.

Extracted keywords After candidate keywords are scored, the top T scoring candidates are selected as keywords for the document. T is computed as one-third the number of words in the graph. The sample abstract contains 28 content words, resulting in $T = 9$ keywords. Below you can see a comparison between rake extracted keywords and the manually assigned:

Extracted by RAKE	Manually assigned
minimal generating sets	minimal generating sets
linear diophantine equations	linear Diophantine equations
minimal supporting set	
minimal set	
linear constraints	linear constraints
natural numbers	
strict inequations	strict inequations
nonstrict inequations	nonstrict inequations
upper bounds	upper bounds
	set of natural numbers

2.5 Deep Learning for NLP

In order to use Deep Learning techniques for Natural Language Processing, first of all it is needed to transform words in numbers or better in vectors. The text can be pre-processed by creating a dictionary (or using a predefined one) which contains a unique identification key for each word in the **text corpus** (is a set of texts, for example a collection of text documents), unknown words and words or characters deemed insignificant are usually given a predefined "dump"-key, usually 0. Usually different grammatical versions of a word is viewed as different words. Non-word objects are usually removed from the document if they have no specific significance for the objective domain. Different capitalized variations of words may appear (e.g. New York, NEW YORK), but this problem can be avoided by ignoring the difference between upper and lower case. It is convenient to numerate a word with the value according to how frequent it occurs in the corpus or by it's importance as calculated in a TF-IDF representation (this representation will be introduced later in this section). Meaning that the 4th most frequent or important word is encoded with the key "4". Often the most frequent and infrequent words are removed as they are deemed insignificant (i.e. they do not contain any information of enough significance and may contribute to making the model too complex, resulting in overfitting). The input types used to represent

the text data can be non-sequential input or sequential input.

2.5.1 Non-sequential input

Non-sequential input is the simplest input variant which ignore the placement of words and as such ignore any spatial correlation between words (i.e. the significance of the words placement according to each other). Among these input types Bag of Words and TF-IDF (term frequency–inverse document frequency) are quite popular and will be introduced here.

Bag of Words Bag of Words is a simple approach which uses a dictionary G to create a counting vector v of the length |G|, where each element v_i refers to the number of occurrences of the word G_i in a document. This approach is often used for document classification.

An example of a Bag of Words representation:
Here we view the whole text corpus as the two "documents" listed below.

1. Ron eats potatoes. His sister stole a potato.
2. Ron ate a potato. His sister still stole a potato.

The corresponding dictionary becomes (here in the order as observed):
["Ron", "eats", "potatoes", "His", "sister", "stole", "a", "potato", "ate", "still"]
, which contains 10 distinct words. And using the indexes as given in the dictionary both documents can be represented as vectors of length 10. These vectors becomes:

1. [1,1,1,1,1,1,1,1,0,0]
2. [1,0,0,1,1,1,2,2,1,1]

TF-IDF TF-IDF (often written as tf-idf) stands for "term frequency–inverse document frequency", which is a more advanced non-sequential representation form than Bag of Words. This statistic is intended to imply how important a word is to a document based on the text corpus. The approximated importance

of a word increases proportionally to the number of times the word appears in the document and decreases based on how often it appears in the rest of the text corpus. Variations of TF-IDF are often used in search engines as a tool for ranking the relevance of a document based on a search query. The TF-IDF weight is composed of the two terms Term frequency (TF) and the Inverse Document Frequency (IDF). Term Frequency (TF) measures how frequent a term appears in a given document. Because of the variability of document lengths it is possible that a term appears more often in a longer document than a shorter more relevant document. This is incorporated by normalizing the number of appearances of a word by the total number of words in the document. The Term Frequency is defined as:

$$TF(t,d) = \frac{N(t,d)}{\sum_t N(t,d)}$$

where N(t, d) is the number of times the word with index t, in the dictionary, appears in document d. TF(t, d) is then the frequency that the word with index t, in the dictionary, appears in document d.

Inverse Document Frequency (IDF) measures how important a term is. This is done by penalizing frequent terms (e.g. "is", "of", "that", etc.) and scaling up the importance of infrequent terms. One variant of the IDF is defined as:

$$IDF(t) = \log \frac{|D|}{\sum_{d \in D} N(t,d)! = 0} = \log \frac{|D|}{|D| - \sum_{d \in D} N(t,d) == 0}$$

where D is the set of documents and |D| gives the total number of documents. The simplest TF-IDF method is defined as:

$$TF - IDF(t,d) = TF(t,d) * IDF(t)$$

A common modification of this is:

$$TF - IDF(t,d) = TF(t,d) * (1 + IDF(t))$$

which effectively gives a word that occurs in every document the TF-IDF(t, d) value equal to TF(t, d) instead of zero.

An example of a TF-IDF representation:
The text corpus is the same as in the Bag of Words example. Here we view the whole text corpus as the two "documents" listed below.

1. Ron eats potatoes. His sister stole a potato.
2. Ron ate a potato. His sister still stole a potato.

The corresponding dictionary becomes (here in the order as observed):
["Ron", "eats", "potatoes", "His", "sister", "stole", "a", "potato", "ate", "still"]
, which contains 10 distinct words. Using the indexes as given in the dictionary both documents can be represented as vectors of length 10. There are two documents so $|D| = 2$, and the first sentence contains 8 words and the second 10 words. The $TF(t,1)$ and $TF(t,2)$ vectors are then equal to the corresponding Bag of words vectors divided by the number of words in each sentence:

1. $\frac{1}{8} \times [1,1,1,1,1,1,1,1,0,0]$
2. $\frac{1}{10} \times [1,0,0,1,1,1,2,2,1,1]$

The how many documents each word appears in is given by the vector

$$[2,1,1,2,2,2,2,2,1,1,1], \tag{3.3}$$

and since $\log(1) = 0$ and $\log(2) \approx 0.69$ the IDF vector becomes

$$0.69 \times [0,1,1,0,0,0,0,0,1,1]. \tag{3.4}$$

Using the TF-IDF modification $\text{TF-IDF}(t,d) = TF(t,d) \cdot (1 + IDF(t))$ we get

1. $\frac{1}{8} \times [1,1.69,1.69,1,1,1,1,1,0,0]$
2. $\frac{1}{10} \times [1,0,0,1,1,1,2,2,1.69,1.69]$

2.5.2 Sequential Input

As the previous text representations methods does not retain word order a more advanced representation method is desirable. One approach is to represent the text data as a sequence, this retains the word order and results in insight of word placement correlation. This is done by creating a vector of smaller or equal length to the document and transforming the document into a sequence of numbers where each word is enumerated in accordance to a dictionary. Usually it is desired that all the vectors are of the same length. This is done by choosing a desired length and cutting the vectors that are too long (and removing the redundant parts, an example is removing the end of the documents that are too long) and padding 0's to the vectors that are too short so that they all are of the same length.

Utilizing this representation one can represent each word as a vector (popularly called a word vector) and not blindly assume that the worth of a word in a sentence is most effectively represented by a scalar value.

2.5.3 Word vectors

Representing each word as a vector will incorporate more complex information into the representation, which can be used to retrieve even more information from the interactions of each word and their placement relative to each other. One can either train one's own representations of each word or use pre-trained representations such as GloVe [4] and Googles word2vec [5]. The GloVe representation is introduced by [49]. The word2vec representation is based on the papers [45] [46] and [47] which introduce, evaluate and improve the Continuous Bag-ofWords Model (CBOW) and the Continuous Skip-gram Model (popularly called the Skip-gram model). The TF-IDF approach give us some idea of a word's relative importance in a given corpus, it however does not give any insight into the words semantic meaning. Word vectors have been shown to capture syntactic and semantic lin-

A word sequence example:
Assume the text corpus contain the two documents:

1. Leonardo was amazing in Inception.
2. Leonardo needs an Oscar.

Using the dictionary:

- Leonardo: 1
- was: 2
- amazing: 3
- in: 4
- Inception: 5
- needs: 6
- an: 7
- Oscar: 8

And creating word sequences of length 5 this results in the word sequences:

1. [1, 2, 3, 4, 5]
2. [1, 6, 7, 8, 0]

guistic regularities well [47] . These word vectors are quite useful as features in NLP problems.

A popular example showing that word vectors are able to capture semantic similarities between words:
Assume that the words {"king", "queen", "man", "woman"} are represented by the vectors $v_{\text{king}}, v_{\text{queen}}, v_{\text{man}}, v_{\text{woman}}$. Then the relationship

$$v_{\text{king}} - v_{\text{man}} + v_{\text{woman}} \approx v_{\text{queen}} \tag{3.5}$$

holds (this behaviour is the result of a huge training set such as the Google News dataset of about 100 billion words). Another example is the relationship:

$$v_{\text{Paris}} - v_{\text{France}} + v_{\text{Italy}} \approx v_{\text{Rome}} \tag{3.6}$$

Note: These results were reported by Mikolov *et al. (2013b).*

Asymptotically it is intuitive that one would get better theoretical results if one train domain specific word vectors instead of using pre-defined ones. Of course in many circumstances one would need lot of training data and training time to get the domain specific word vectors to outperform the pre-trained vectors (which has been trained on extremely large datasets). Methods for computing word vectors are described later in this section.

A word sequence example with word vectors:
Assume one has a weight matrix W and the following sequence vectors:

1. $[1, 2, 3]$
2. $[2, 1, 0]$

Assume W is given as:

$$W^T = \begin{pmatrix} \{0\} & \{1\} & \{2\} & \{3\} \\ 0 & 1 & 1 & 2 \\ 0 & 1 & 2 & 2 \end{pmatrix} \tag{3.7}$$

Each number in the sequence vectors refers to a row in the weight matrix W which is the corresponding word vector. The initial row with id 0 models all unknown words (usually a zero vector) and the 3 other rows contains the known word representations. Using this weight matrix the word sequences can be modelled as matrices:

1. $\begin{bmatrix} 1 & 1 & 2 \\ 1 & 2 & 2 \end{bmatrix}$
2. $\begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & 0 \end{bmatrix}$

Here each word in the sequence is replaced by the corresponding word vector, as a column (eg. the first word in a sequence is represented by column 1 and the n'th word in a sequence is represented by column n).

The weight matrix is kept for efficient representations as it would take too much memory to at all times represent each occurrence of a word as a vector when the vectors become large.

2.5.4 More advanced non-sequential input

More advanced non-sequential input representations like document vectors and word clusters use the word vector representations to compute their values. These

representations are introduced here.

Document vectors [40] introduced the concept of computing the average of the word vectors representing a sentence or document and use it as the input to a classification/regression model. This averaged vector is dubbed a Document vector and can be used to represent documents in a more semantically rich way than standard BoW and TF-IDF representations and is much cheaper to use for training than sequential input. [33] used Document vectors as the input neural network models and efficiently trained a model with good results.

Word clusters - Bag of Clusters In word clusters the most similar words are paired together in clusters. Meaning that instead of counting words the Bag of Clusters representation counts occurrences of each cluster. Word clusters can easily be trained by using Support vector clustering (SVC) (a clustering method using support vector machines introduced by [12]) to create word clusters (dubbed Bag of Clusters).

2.5.5 Methods to extend the vocabulary

Ways of extending the vocabulary, such as N-grams and Skip-grams, are introduced in this section.

N-grams N-grams can be useful to represent phrases with unique meaning as single items. An example would be "Air Canada", as its meaning cannot easily be combined from the meanings of the separate items "Canada" and "Air". N-gram's are continuous sequences of n items from a given sequence (usually from text or speech) which can be used to extract such phrases.

N-grams are especially nice to use to boost a tf-idf representation as it can weigh important phrases with high values and unimportant phrases with low val-

A n-gram example:
Assume the sentence:

"The Shawshank Redemption is quite excellent."

The 1-gram and 2-gram representations of the sentence:

- 1-grams (uni-grams):
  { The, Shawshank, Redemption, is, quite, excellent }
- 2-grams (bi-grams):
  { The Shawshank, Shawshank Redemption, Redemption is, is quite, quite excellent }

ues. Any redundant phrases can be removed by only keeping the values over a certain threshold or the vocabulary with the k-highest values.

Skip-gram Skip-gram modelling is a generalization of n-grams which handles data sparsity better than classic n-grams as shown by [30] (i.e. in cases where the vocabulary is too small it can be extended with skip-grams to improve performance of the models used). Skip-grams gives the ability to skip words (items) in a sequence. A k-skip-n-gram contains all the sub-sequences of n words where each word is a distance k or less from the previous one.

An skip-gram example:
Assume the sentence:

"The potatoes are the real victims here!"

The resulting 1-skip-2-gram representation:

{ The potatoes, The are, potatoes are, potatoes the, are the, are real, the real, the victims, real victims, real here, victims here }

2.5.6 Naive Bayes Classifier

The simplest solutions are usually the most powerful ones, and Naive Bayes is a good proof of that. In spite of the great advances of the Machine Learning

in the last years, it has proven to not only be simple but also fast, accurate and reliable. It has been successfully used for many purposes, but it works particularly well with natural language processing (NLP) problems. Naive Bayes is a family of probabilistic algorithms that take advantage of probability theory and Bayes' Theorem to predict the category of a sample. They are probabilistic, which means that they calculate the probability of each category for a given sample, and then output the category with the highest one. The way they get these probabilities is by using Bayes' Theorem, which describes the probability of a feature, based on prior knowledge of conditions that might be related to that feature. In the case of sentence classification, we don't even have numeric features. We just have text. We need to somehow convert this text into numbers that we can do calculations on. So we use word frequencies. That is, we ignore word order and sentence construction, treating every document as a set of the words it contains. Our features will be the counts of each of these words. Even though it may seem too simplistic an approach, it works surprisingly well.

Bayes' Theorem Now we need to transform the probability we want to calculate into something that can be calculated using word frequencies. For this, we will use some basic properties of probabilities, and Bayes' Theorem. Bayes' Theorem is useful when working with conditional probabilities, because it provides us with a way to reverse them:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \rightarrow P(B|A) = \frac{P(A|B)P(B)}{P(A)}$$

Suppose that A is our category and B is the sentence that must be classfied. To calculate $P(A|B)$, we can count how many times the sentence B appears in the A category $P(A|B)$, divide it by the total P(A). There's a problem though: the sentence B doesn't appear in our training set, so this probability is zero. Unless

every sentence that we want to classify appears in our training set, the model won't be very useful. So here comes the Naive part: we assume that every word in a sentence is independent of the other ones. This means that we're no longer looking at entire sentences, but rather at individual words. This assumption is very strong but super useful. It's what makes this model work well with little data or data that may be mislabeled. And now, all of these individual words actually show up several times in our training set, and we can calculate them. The final step is just to calculate every probability and see which one turns out to be larger. Calculating a probability is just counting in our training set. In summary:

- Very simple, but effective probabilistic classfier:

$$p(y|x_1, ..., x_n) = \frac{p(y, x_1, ..., x_n)}{p(x_1, ..., x_n)} = \frac{p(x_1, ..., x_n|y)p(y)}{p(x_1, ..., x_n)}$$

  Class prior probability p(y) is just the frequency of the class in the training data.

- Naive Bayes Assumption to calculate $p(y|x_1, ..., x_n)$:

$$p(x_1, ..., x_n|y) = \prod_{i=1}^{n} p(x_i|y)$$

  Each observed variable is assumed to be independent of each other given the class.

- Denominator can be ignored since the data are given and the same across all y, we are interested in:

$$arg\max_y(y|x_1, ..., x_n) = arg\max_y p(y, x_1, ..., x_n)p(y)p(x_1, ..., x_n) =$$

$$arg\max_y(x_1, ..., x_n|y)p(y)$$

There are many things that can be done to improve this basic model. These techniques allow Naive Bayes to perform at the same level as more advanced methods. Some of these techniques are:

- Removing stopwords: these are common words that don't really add anything to the categorization, such as a, able, either, else, ever and so on.
- Lemmatizing words: this is grouping together different in sections of the same word. For example: selection, selections, selected, and so on would be grouped together and counted as more appearances of the same word.
- Using n-grams: instead of counting single words, we could count sequences of words.
- Using TF-IDF: instead of just counting frequency we could do something more advanced like also penalizing words that appear frequently in most of the samples.

2.5.7 Multinomial Naive Bayes

The best approach to characterize text documents is using term frequency-inverse document frequency (tf-idf(t, d)) (shown in previous section). This can be used to compute the maximum-likelihood estimate based on the training data to estimate the class-conditional probabilities in the multinomial model:

$$P(x_i|y_j) = \frac{\sum tf - idf(x_i, d \in y_j) + a}{\sum N_{d \in y_j} + aV}$$

where:

- $\sum tf - idf(x_i, d \in y_j)$ is the sum of raw term frequencies-inverse document frequencies of word x_i from all documents in the training sample that belong to class y_j.

- $\sum N_{d \in y_j}$ is the sum of all term frequencies in the training sample for class y_j.
- "a" is an additive smoothing parameter (a=1 for Laplace smoothing).
- "V" the size of the vocabulary (number of different words in the training set). The class-conditional probability of encountering the text x can be calculated as the product from the likelihoods of the individual words (under the naive assumption of conditional independence).

2.5.8 Neural Networks

Neural networks consists of multiple neurons in multiple layers, each neuron is modelled in the same way, usually with the same parameters in each layer. Each neuron is activated to a certain degree based on the input given. A single neuron model is illustrated in Figure 2.17. The neuron activation function is defined as:

$$y = f(x) = \sigma\left(\sum_{i}^{K} w_i x_i\right) = \sigma\left(\mathbf{w}^T(x)\right)$$

where $\mathbf{x}$ is the input vector and $\mathbf{w}$ is the weight vector. f is usually a non-linear activation function that maps the vector $\mathbf{x}$ to the scalar output y. Often a bias is added by setting $x_0 = 1$ (where the corresponding w_0 acts as the bias).

An activation function is said to be activated if its output is non-zero. It is also said to have a strong activation if the output is relatively high and have a weak activation if its output is relatively small. An activation function is desired to be non-linear, continuously differentiable and monotonic, it is further desired that the function $f(x) \approx x$ when x approaches 0. The activation functions are desired to be non-linear as this is a feature needed for the neural network to be an universal approximator [15]. Continuously differentiable activation functions are necessary for gradient-based optimization methods. Monotone activation func-

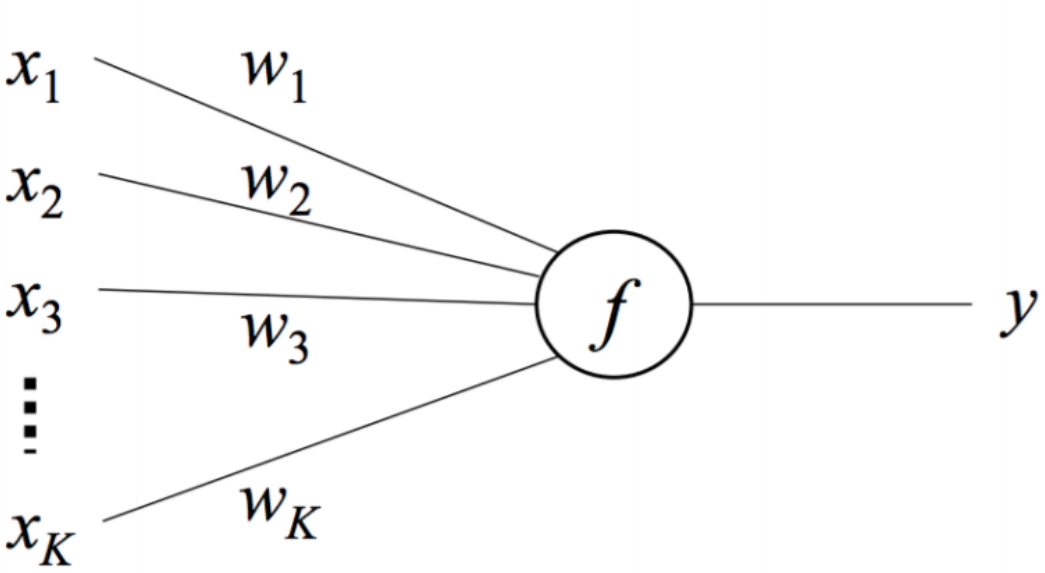


Figure 2.17: This figure shows an illustration of a single neuron model

tions guarantees a convex error surface of a single-layer model [58]. And $f(x) \approx x$ when x approaches 0 the networks can train more efficiently (if this is not satisfied weights must be initialized with care [56]. There are many activation functions to choose from. The sigmoid, hyperbolic tangent and the ReLU activation functions are introduced and defined in this section. Some classification and regression functions used in the final layer of neural networks are also introduced.

Activation functions

Sigmoidal functions The most common form of activation functions are the sigmoidal functions which are monotonically increasing functions that asymptotically approaches some value as the input approaches $\pm\infty$. The most common sigmoidal functions are the standard logistic function (usually referred to as the sigmoid function) and the hyperbolic tangent. The logistic sigmoid, motivated somewhat by the biological neurons, is defined as:

$$f(x) = \frac{1}{1 + e^{-x}} \in [0, 1)$$

The hyperbolic tangent which approximates the logistic sigmoids behaviour and normalize the data between -1 and 1 is defined as:

$$f(x) = tanh(x) \in (-1, 1)$$

Both sigmoidal functions are plotted in the figure 2.18.

The hyperbolic tanget is better for training models efficiently with back-propagation (this is an optimization method used for neural networks and which will be introduced later in this section). [42] motivates that sigmoidal functions that are symmetric around the origin are preferred because they on average produce outputs close to zero which results in a faster convergence. Both sigmoidal functions however face the **vanishing gradient problem**. The vanishing gradient problem comes from how neural networks are trained with back-propagation. The error signal computed in an n-layer model consists of n gradients in the range of $-1, 1$ for hyperbolic and 0,1 for sigmoidal multiplied together, this results in a small (vanishing) error signal and in turn results in slow training of the model.

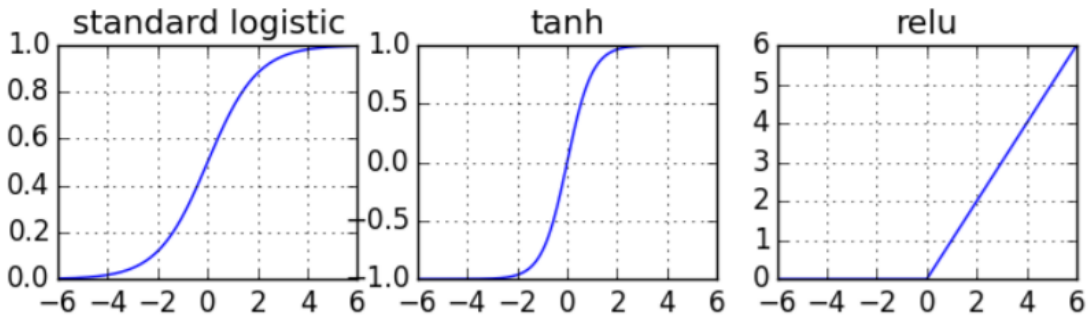


Figure 2.18: Plots of the standard logistic function, the hyperbolic tangent

Rectifier (ReLU) Another activation function of interest is the rectifier function, also known as the ramp function, which as of 2015 is the most popular activation function for deep neural network according to [43]. A unit using the rectifier activation function is called a rectifier linear unit, ReLU. The activation

function is often referenced as ReLU in deep learning applications/programming. The rectifier function is defined as:

$$f(x) = max(0, x)$$

The rectifier function is plotted in figure 2.18. [28] argues that it is more biologically plausible than the logistic sigmoid and that it is more efficient to train than the hyperbolic tangent. [28] also shows that the rectifier function is "remarkably" adapted to sentiment analysis in text-based tasks. Further motivations for the Rectifier functions is that it results in sparse activations (on an average only about 50 % of the ReLU units are activated) and that it isn't afflicted by the vanishing gradient. It is however not differentiable close to 0, a way around this is the smooth approximation of the rectifier function called the **softplus** function. The softplus function is defined as:

$$f(x) = \ln 1 + e^x$$

The softplus function however doesn't induce the sparsity that the ReLU function does. Other variants of the Rectifier have been tailored for various specific deep learning tasks.

Final Layer

For the final layer in the neural networks probabilistic classification functions, such as the softmax function, or regression functions, such as the linear activation function, are preferred depending on if it is a classification or regression problem. The softmax and linear activation functions are described in this section, along with a mention about how support vector machines might surpass their performance.

Softmax When the classification problem isn't binary but contains multiple class the softmax function is usually used, it is however also viable for the binary case. This is a generalization of the logistic function and is just another name for a multinomial classification model when one assumes that there exists no hierarchy among the classes. The softmax function is nice as it gives an approximation of the probability that a class is the correct one. The simplest approach is to simply choose the class with the highest probability, and ignore the rest. But since it is a probabilistic function it can also be used for a generative model. The softmax scores (probabilities) are computed by the normalizing function:

$$\sigma_j = P(y = j/\mathbf{x}) = \frac{e^{\mathbf{x}^T \mathbf{w}_j}}{\sum_k^K e^{\mathbf{x}^T \mathbf{w}_k}}$$

$j \in \{0, 1, ..., K\}$ and K is the number of classes. $P(y = j\|\mathbf{x}$ is the probability of class j being the correct class of the K classes given the observations $\mathbf{x}$, and the weights $\mathbf{w}_j$.

Linear The linear activation layer is just a simple linear regression model trained on the incoming features. Given input features $x_i \in X$ and weights $w_i \in W$ where $i \in 1, ..., n$ the output of the linear layer is given as:

$$y = f(x) = X^T W = \sum_i^n w_i x_i$$

SVM However [57] demonstrate a small but consistent advantage in classification problems of replacing the final softmax layer with a linear support vector machine (SVM). These findings could also imply that the SVM extension for regression, the Support Vector Regression (SVR) model, could outperform the final linear activation layer in regression problems.

Weight initialization

At initialization it is desirable that the weights are close to the center of the possible values of it's domain, so that the activation function operates in the domain where it is approximately linear and the gradients are close to their potential maximums. [29] recommends drawing the initial weights from the uniform distribution. The width of the uniform distribution sampled from depends on the activation function and the variable n, which is the sum of n_{in}, the number of input values given to the hidden layer that the weights belong to, and n_{out}, the number of hidden units in the hidden layer. This normalizing of the uniform distribution is meant to fulfill the objective of maintaining activation variances and back-propagated gradient variances. For the standard logistic function draw from:

$$U\left[-4x\frac{\sqrt{6}}{\sqrt{n}}, 4x\frac{\sqrt{6}}{\sqrt{n}}\right]$$

For the hyperbolic tangent draw from:

$$U\left[-6x\frac{\sqrt{6}}{\sqrt{n}}, 6x\frac{\sqrt{6}}{\sqrt{n}}\right]$$

The scaling values 4 and 6 corresponds to the width of the area that the standard logistic and hyperbolic tangents haven't yet reached their maximum values, as $|x|$ values greater than 4 (logistic function) and 6 (hyperbolic tangent) results in $f(x)$ reaching it's min or max value. For the rectifier function, ReLU, [31] recommends sampling the weights from the normal distribution $N(0, \sigma^2)$, with variance 0.01. If the layer is very large (wide) using a variance of 0.001 may induce better performance/results.

2.5.9 Multilayer Perceptron (MLP)

The Multilayer perceptron (MLP) model is a basic neural network that consists of multiple neurons. The MLP model with 1 hidden layer is illustrated in Figure 2.19 and consists of an input layer, a hidden layer and an output layer. The input layer consists of the input vector $\mathbf{x} = \{x_1, ..., x_K\}$, with K input variables. The hidden layer consists of the hidden vector $\mathbf{h} = \{h_1, ..., h_N\}$, with N neurons (where each neuron behave just as the single neuron model). And the output layer consists of the output vector $\mathbf{y} = \{y_1, ..., y_M\}$, with M neurons. Every element in the input layer is connected to every element in the hidden layer, where element w_{ki} of weight matrix $W(K \times N)$ indicates the weight associated with input element k and hidden element i. The same connection structure is also present between the hidden layer and the output layer (i.e. that every element in the hidden layer is connected to every element in the output layer), and here element w'_{ij} of weight matrix $W'(N \times M)$ indicates the weight associated with hidden element i and output element j. The output of hidden element h_i is given by the equation:

$$h_i = f(u_i) = f\left(\sum_{k=1}^{K} w_{ki} x_k\right) \forall i \in \{1, 2, ..., N\}$$

where u_i is the input of the activation function of hidden element h_i . And the output values y_j of the output layer y are given by the equation:

$$y_j = g(v_j) = g\left(\sum_{i=1}^{N} w'_{ij} h_i\right) \forall j \in \{1, 2, ..., M\}$$

where v_j is the input sent to some activation function g that computes the value for y_j . The weights W,W' are trained by using Stochastic Gradient Descent as it is a computationally efficient alternative to standard optimization methods. The full optimization method is called back-propagation.

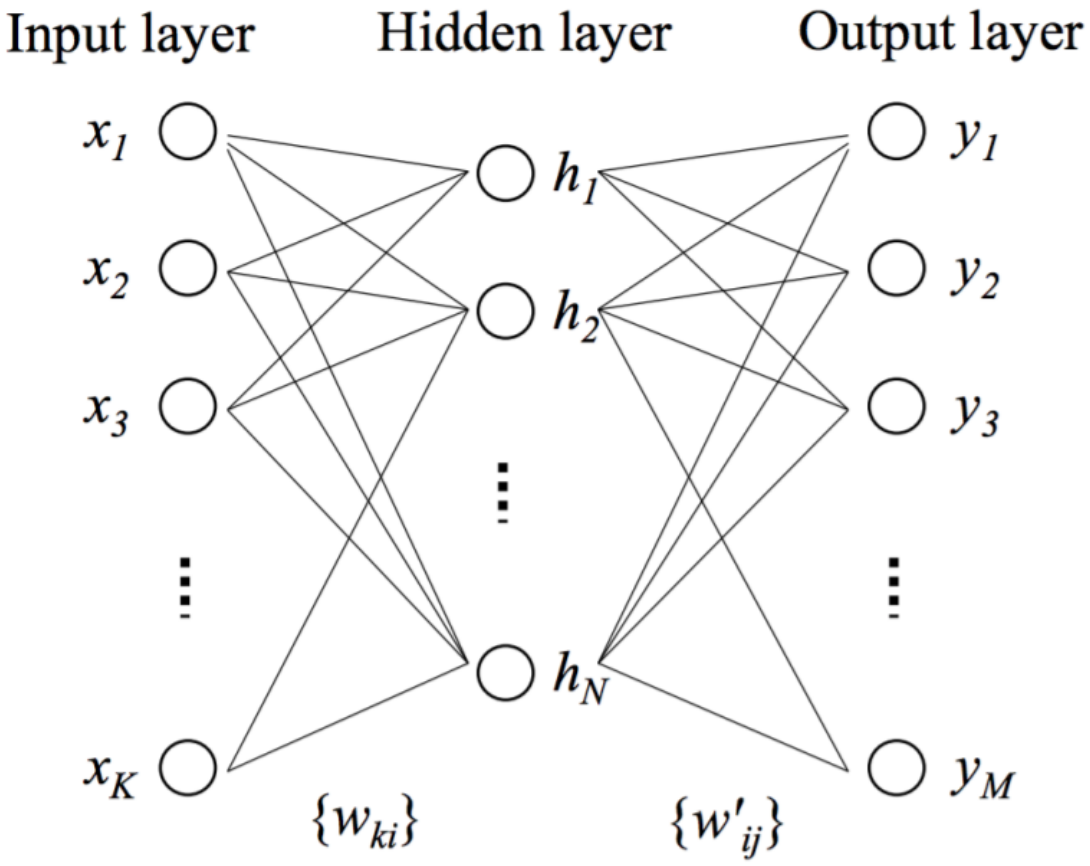


Figure 2.19: The MLP model with one hidden layer

2.5.10 Back-Propagation

Back-propagation is the method used to train neural networks. Back-propagation is an abbreviation of "backward propagation errors", i.e. the final classification/value error gets propagated backwards in the network in order to update the weights. An example of using back-propagation on the MLP model with one hidden layer: we want to train the MLP model by updating the weights W and W' . We first find the gradient of the chosen loss function E with respect to W' . Using the chain rule we get:

$$\frac{\partial E}{\partial w'_{ij}} = \frac{\partial E}{\partial v_j} \frac{\partial v_j}{\partial w'_{ij}}$$

where:

$$\frac{\partial v_j}{\partial w'_{ij}} = h_i$$

$\frac{\partial E}{\partial v_j}$ can be rewritten:

$$\frac{\partial E}{\partial v_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial v_j}$$

which makes it easier to compute the gradient when the loss and activation functions are known. Thus the gradient can be written as:

$$\frac{\partial E}{\partial w'_{ij}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial v_j}h_i \forall w'_{ij} \in W'$$

Next we find the gradient of the loss function E with respect to W . Using the chain rule we get:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial h_i}\frac{\partial h_i}{\partial u_i}\frac{\partial E}{\partial w_{ki}}$$

where:

$$\frac{\partial u_i}{\partial w_{ki}} = x_k$$

and

$$\frac{\partial E}{\partial h_i} = \sum_j^M \left(\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial v_j}\frac{\partial v_j}{\partial h_i}\right) = \sum_j^M \left(\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial v_j}w'_{ij}\right)$$

where $\frac{\partial E}{\partial y_j}$ and $\frac{\partial y_j}{\partial v_j}$ have already been computed for the weights w'_{ij}. Thus the gradient can be written as:

$$\frac{\partial E}{\partial w_{ki}} = \sum_j^M \left(\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial v_j}w'_{ij}\right)\frac{\partial h_i}{\partial u_i}x_k \forall w_{ki} \in W$$

Using the gradients defined for $\frac{\partial E}{\partial w'_{ij}}$ and $\frac{\partial E}{\partial w_{ki}}$ the update algorithm of each set of weights respectively is given by the SGD-algorithm (this is the simplest approach with a momentum parameter of 0):

$$w'_{ij} \leftarrow w'_{ij} - \eta \frac{\partial E}{\partial w'_{ij}} = w'_{ij} - \eta \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial v_j} h_i \forall w'_{ij} \in W'$$

$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{j}^{M} \left(\frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial v_j} w'_{ij} \right) \frac{\partial h_i}{\partial u_i} x_k \qquad \forall w_{ki} \in W$$

2.5.11 Regularization

Regularization is a method commonly used to prevent overfitting on the training data, which occurs when the model describes noise instead of the underlying relationship of the data. The desired effect of a model is a good generalization for all observed and unobserved data in the domain it predicts in. There exists many regularization approaches such as adding regularization terms to the loss function, ensemble methods and early stopping. Regularization terms such as L1 (Lasso) and L2 (Ridge Regression) adds a constraint function to the weights, this adds extra terms to optimize and gets quite costly as a neural network usually has many weights, thus this isn't always feasible or preferred. Ensemble methods trains a lot of models and averages the output and this is not efficient for deep neural networks as it is quite costly to train a neural network and it would be extremely costly to train enough models for a good ensemble. Early stopping depends on a validation set (which is not part of the training set) and stops training when accuracy (or another evaluation metric) on the validation set stops improving. Early stopping is more of an intuitive method and isn't very theoretically backed. Another regularization method is needed as common ensemble methods and regularization terms are to costly and inefficient when used for deep neural

networks (further early stopping is poorly theoretically motivated and more than a little luck based).

Droput Dropout is a regularization method introduced by [54] tailored specifically for deep neural networks. It effectively approximate model combination, prevents overfitting and approximates exponentially many neural nets efficiently. The idea is to prevent the model from being too specialized on the training data (i.e. overfit) by at random removing (hidden and input) units from the model temporarily. The units are present with probability p as illustrated in Figure 2.21.$p = 0.5$ seems to be close to the optimal value for a wide range of networks and tasks. The input nodes seems to have an optimal p closer to 1 (a typical value is 0.8) Since it is not feasible to average the prediction of many thinned models at test time an approximate averaging method is used. The final prediction model is one neural net where the weights are scaled by the dropout probability p, as shown in Figure 2.20. This ensurers that the output at test time is the same as the expected output at training time. Dropout leads to significantly lower generalization error compared to other regularization methods on a wide variety of task including object classification, digit recognition, speech recognition, document classification and analysis of computational biology data. A drawback of droput is that it increases training time ($p = 0.5$ results in a approximately 2-3 times longer training time than the basis model). This is mainly because the parameter updates becomes very noisy. However this stochasticity is likely the factor that prevents overfitting. This results in a trade-off between overfitting and training time (i.e. with more training time, one can use higher dropout and suffer less overfitting).

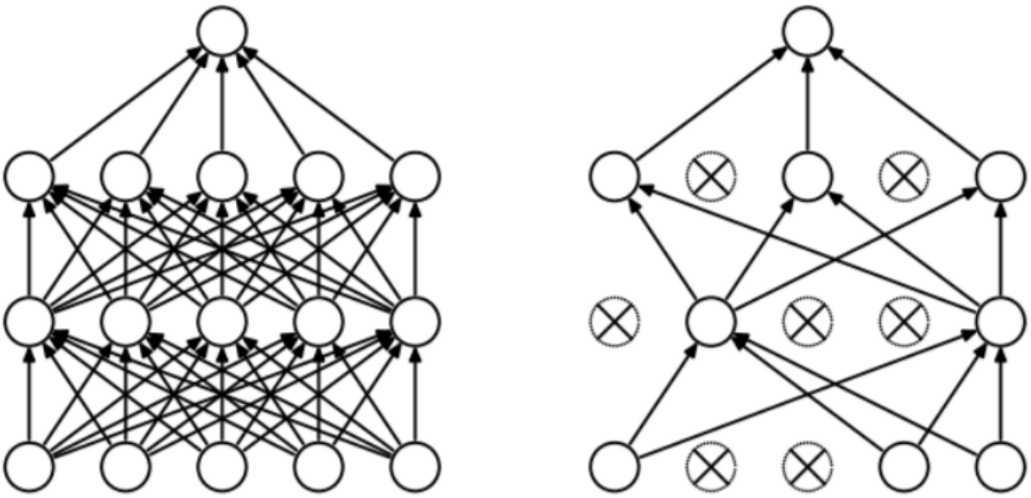


Figure 2.20: The Dropout Neural Network model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned network produced by applying dropout to the network on the left. Crossed units have been dropped.

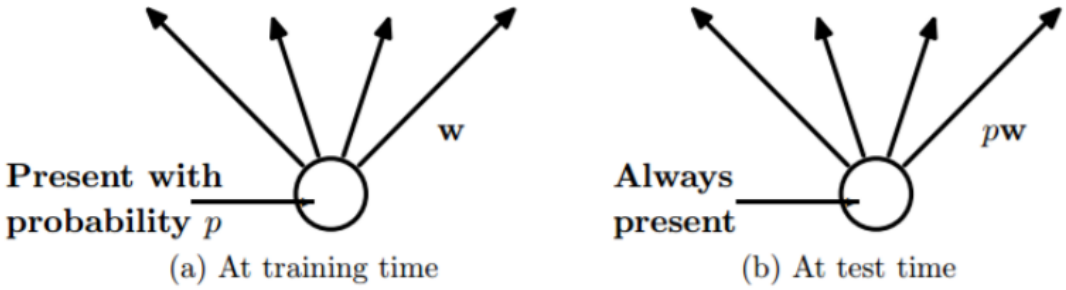


Figure 2.21: **Left**: At training time, the unit is present with probability p and is connected to units in the next layer with weights w. **Right**: At test time, the unit is always present and the weights are scaled by a factor of p. The output at test time is then the same as the expected output at training time.

2.5.12 World vectors model

This section explains how the word vectors are trained. Creating vector representations of words and phrases that retain semantic meanings requires appropriate models. The most popular models for training word vectors are context-counting and context-predicting model types.

Count-based models - GloVe Count-based models are trained by doing dimensionality reduction on a co-occurrence counts matrix. The co-occurrence count matrix C (words $\times$ context) counts the cooccurrence of words and context. Context could for example be retrieved from document tags, for example a sports article or a deep learning paper. Since this matrix is extremely large one factorizes this matrix into (word $\times$ features) matrix U and (context $\times$ features) matrix V . The relation can be represented as

$$C = UV^T$$

These matrices U and V are trained by minimizing the "reconstruction loss", while trying to use low dimensional representations. The aim is to be able to explain most of the variance of the data given these matrices. The (word $\times$ features) matrix U is used to represent the words, each row represents a word (or a class of words if one combines certain similar words like "is" and "are"). GloVe (Global Vectors for Word Representation) is a new and popular Count-based model out of Stanford. The model was introduced by [49] and is an unsupervised algorithm for training word vectors. Pretrained GloVe vectors are available at the GloVe project site [4].

Predictive models - word2vec Predictive models are trained by minimizing a loss function. The Skip-Gram model and the Continuous Bag-of-Words (CBOW) model are two popular predictive models from Google. These models are introduced and evaluated by [46], [45] and [47]. These models are usually referred to as word2vec and pre-trained word vectors are available at the word2vec project site [5]. Both of these models are modelled by a simple neural network with one hidden layer and they are trained using back-propagation. The Continuous Bag of Words (CBoW) model aims to predict a word given the context it

is surrounded by. While the Skip-gram model tries to predict the context given the target word, it also creates more training cases by creating skip-grams of the word context (as already shown). The skip-grams may create context examples of words that are far away from each other which together can give significant context information. A simple example of both models is illustrated in Figure 4.12, where $w(t)$ represents the target word and $w(t-2), w(t-1), w(t+1), w(t+2)$ represents the context of the word. This could represent the sentence:

"Nobles dislike potato eating peasants."

Which could be split up into the word set

$$\{"Nobles", "dislike", "potato", "eating", "peasants"\}$$

. Thus if the window size is 4 and the context is given as

$$\{"Nobles", "dislike", "eating", "peasants"\}$$

the target word is in this instance "potato".

According to Mikolov the Skip-gram model works well with small amounts of training data and represents even rare words and phrases well, while the CBoW model is much faster to train and has a slightly better accuracy for frequent words.

How the models are trained? This section will give a short description on how the Skip-gram and CBoW models are trained. These models are as mentioned trained with back-propagation. Initial definitions:

- C is the context size.
- V is the size of the vocabulary used.
- N is the number of features in the word vector representations.
- $J = \{1, 2, ...V\}$

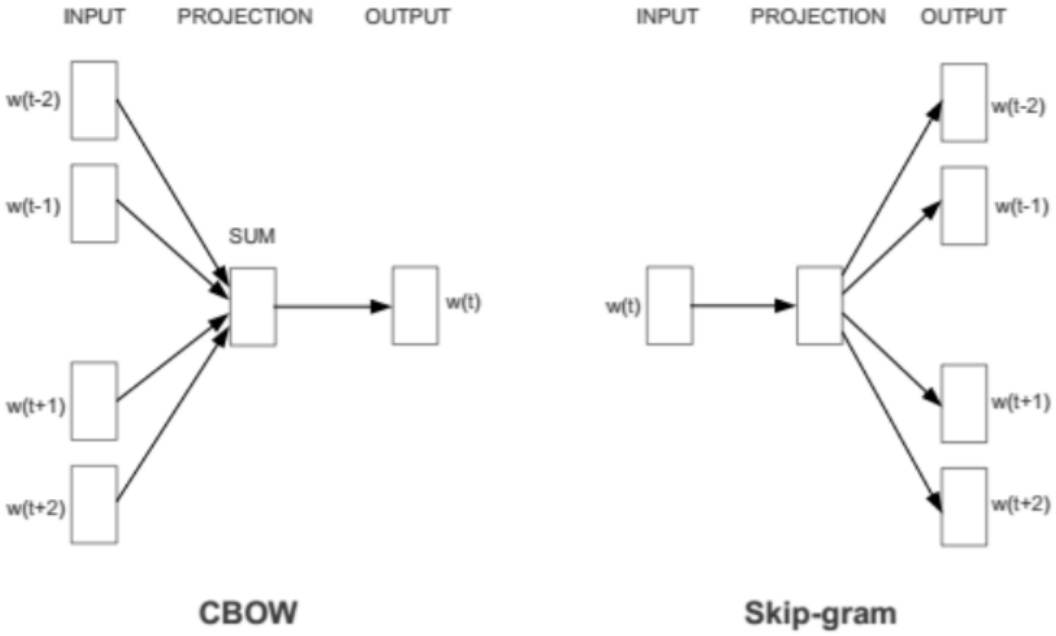


Figure 2.22: A simple example for comparison of the Continuous bag of words model and Skip-gram model. $w(t)$ represents the target word and $w(t-2), w(t-1), w(t+1), w(t+2)$ represents the word context.

- $I = \{1, 2, ..., C\}$
- Hot-encoded vectors: zero-vectors with the value 1 in the cell $j \in J$ which corresponds to the word in the vocabulary it represents.
- Weight matrix $W(V \times N)$
- Weight matrix $W'(N \times V)$

How the Continuous Bag of Word (CBoW) model is trained is illustrated in Figure 2.23. As mentioned, given the word context $x_1, x_2, ...x_C$ the model tries to predict which word the context surrounds. These vectors $x_i i \in I$ are hot-encoded. The model can be simplified by summing the context vectors into a context vector $X = \sum_i^C x_i$. This context vector is connected to a weight matrix W and through it the hidden layer h. The hidden layer is further connected to the weight matrix W' and through it the output layer. The output layer computes

a vector $y(1 \times V)$ and the highest value $y_j j \in J$ corresponds to the word in the vocabulary the model predict that the context X surrounds. How the Skip-gram model is trained is illustrated in Figure 2.24. As mentioned, the only difference from the CBoW model is that the Skip-gram model tries to predict the context $y_1, y_2, ...y_C$ surrounding the word x. The model can be simplified by summing the context vectors into a context vector $Y = \sum_i^C y_i$. The hot-encoded vector x is connected to a weight matrix W and hidden layer h. The hidden layer is further connected to the weight matrix W' and the output layer. The output layer computes a vector y, where the cells with the C strongest activations refers to which words the model predicts that surrounds the word represented by x. For both the models (after they have been trained) the rows in either W or W'^T can be used to represent the words in the vocabulary as word vectors, it is usually W that is used. Meaning that row $j \in J$ in W represents the word vector for word j in the vocabulary.

Count-based models vs. Predictive models [11]report that training word vectors using predictive (context-predicting) models outperform or perform as well as when using count-based (context-counting) models. While [49] and [41] motivates that count-based models are more efficient to train, more easily parallelized and able to infer unseen words and phrases, which is an advantage over predictive models (which has to train representations of any new words if one wants to gain any relevant information from the word).

2.5.13 Convolutional Neural Network

The Convolutional Neural Network (CNN) is a neural network model inspired by how living creatures process natural image data. It is based on the work on the cat's visual cortex by [32] which found that there exists cells that acts as local filters which search the natural images for patterns. These cells are ideal for

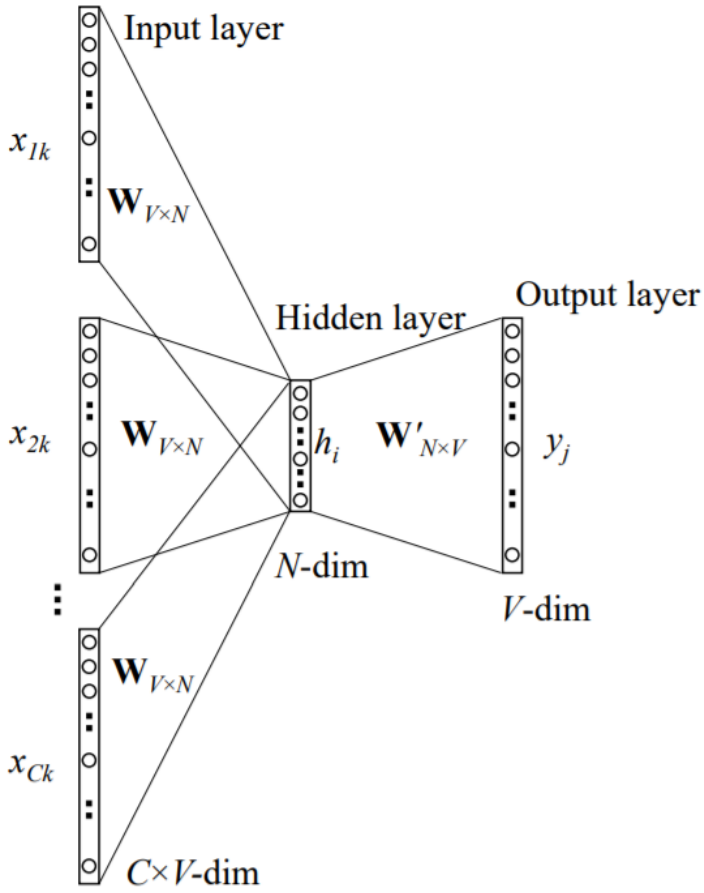


Figure 2.23: The Continuous bag-of-word model

exploiting the strong local correlation which is present in natural images. There exist many CNN models for image processing inspired by the visual cortex, a few of those are [25], [52] e [38], which) achieved "state-of-the-art" performance on the ImageNet dataset using a CNN model, which further supports that the CNN model is well-suited for processing images that have a 2D structure with strong spatial correlation. It has in recent years been motivated that the CNN is a viable model for NLP tasks, as there exists a 1D structure with strong local spatial correlation in natural languages. The effectiveness of CNN model on NLP tasks in comparison to "state-ofthe-art" methods has been demonstrated by [34], [36], [35], [53] and [27], implying that it is able to exploit the correlated 1D structure of text quite well. [34] used a CNN model with hot-encoded input to train domain

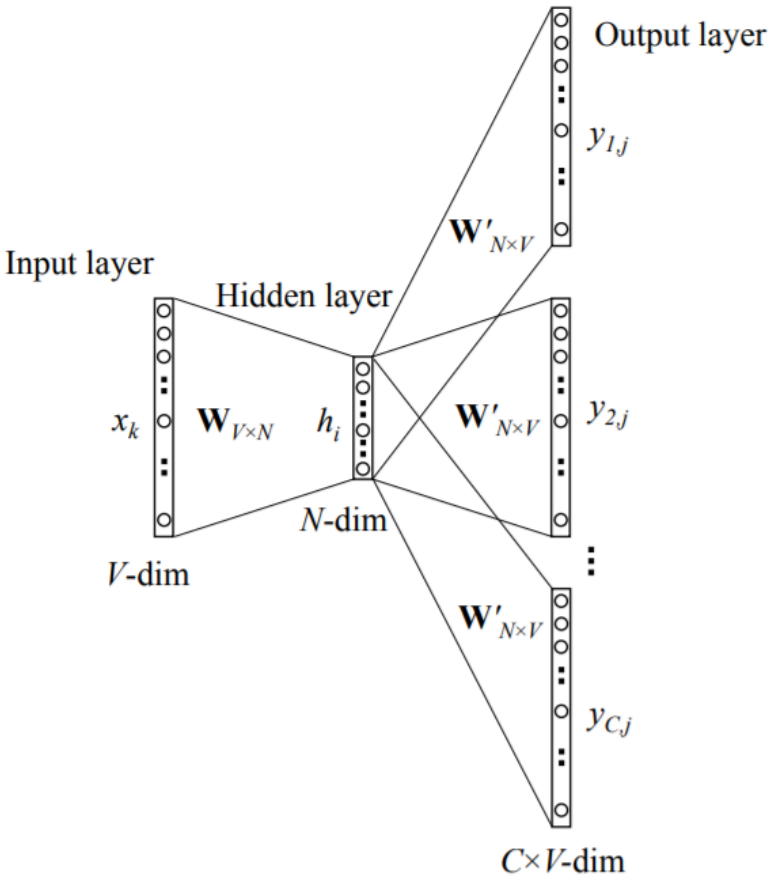


Figure 2.24: The skip-gram model

specific word vectors in the same model which did the main task of classifying documents. [36] used the publicly available word2vec word vectors to encode the input of their CNN variants. [35] introduced the Dynamic CNN which use dynamic k-max pooling. Further [53] presents the Convolutional Latent Semantic Model (CLSM) and [27] (2015) presents the Deep Semantic Similarity Model (DSSM), both models learn semantic representations of sentences for Information Retrieval (these models were trained to recommend documents to users based on what they are currently reading or have been reading).

The model The CNN model consists of an input layer, a convolution layer, a pooling layer, fully connected nodes and a final prediction layer. It can consist

of multiple convolution and pooling layers, but these are usually used in order (i.e. a convolution layer is always preceded by an input layer or pooling layer, while a pooling layer is always preceded by a convolution layer). Deeper CNN models with multiple convolution and pooling layers are usually reserved for extremely large datasets. In the text case the input is given as word sequences (as already shown for sequential inputs) where each word is encoded as a word vector. The word vectors can be static or be trained with the rest of the model. An example of a CNN model with text input is illustrated in Figure 2.25. The CNN model is like the other neural networks trained using back-propagation.

Convolution Each convolution node got one unique filter, which searches the input for a unique pattern. The convolution nodes computes a vector (a matrix in the image case) of the length equal to possible placements of the filter, each cell refers to how strongly the pattern was observed in a the unique location connected to that cell. A simple convolution on an image matrix is illustrated in figure 2.26, in the example the matrix represents a black and white image, where black is represented by 0 and white by 1. The 3×3 yellow window is a filter which slides over the image. In this example the filter multiplies its values element-wise with the part of the image it covers, and sends that value to the "Convolved Feature", which is the matrix it will send to the convolution node it is connected to. The step-wise convolution update given the filters iteration over the image is shown in figure 2.27. In the common text case the height of the filters is set to be the same length as the number of features in the word vectors. While the filters width, called the window size, which decides how many words fits in the filter is a hyper-parameter chosen through tuning (eg. line-search).

Pooling Each convolution node is connected to a unique pooling node. These pooling nodes looks for the most significant information retrieved by the convo-

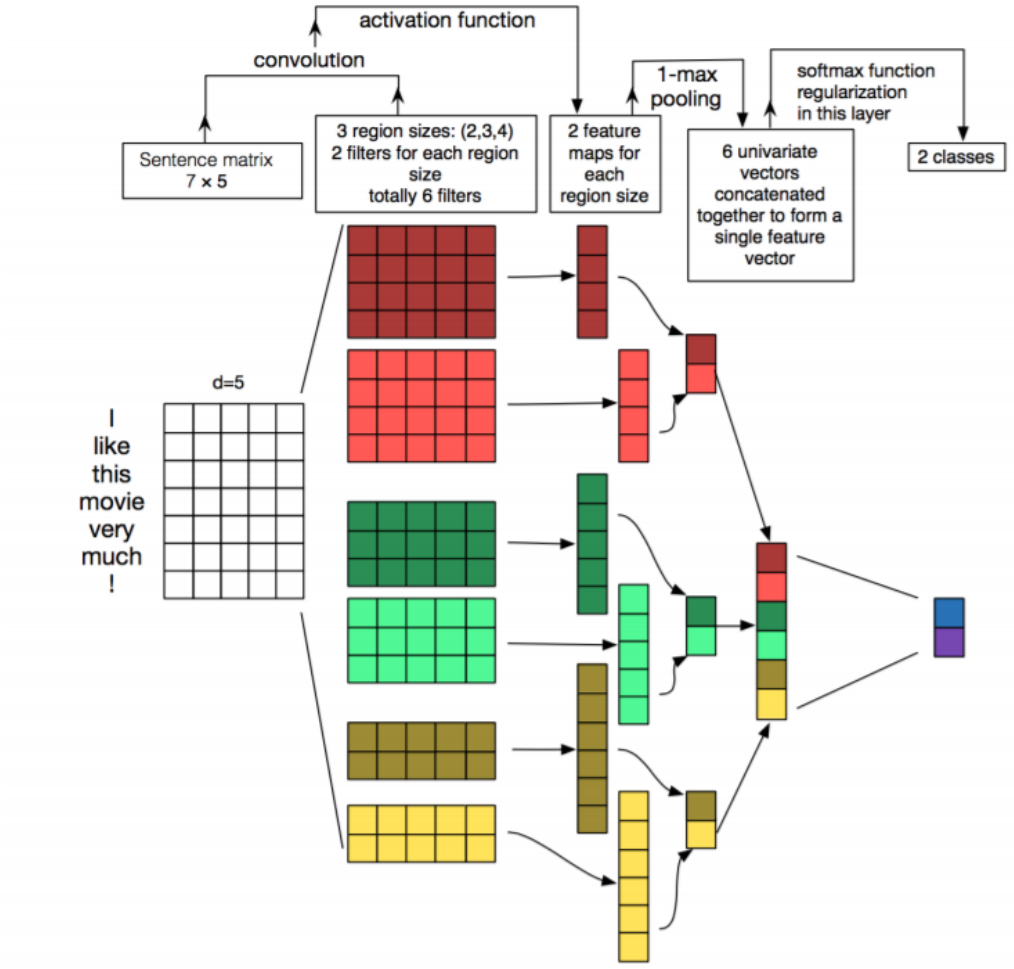


Figure 2.25: A graphical depiction of a CNN model using 1-max-pooling with text input

lution nodes by following different pooling schemes. The simplest pooling scheme is 1-max-pooling, which simply means that the pooling node retrieves the highest value from the filter. Pooling reduces dimensionality while retaining the most important information, this makes the model more computationally effective compared to just using convolution layers and fully connected layers. A max-pooling example is shown in Figure 2.28, where each colored area refers to a separate convolution node. The input matrix consists of 4 convolution nodes which are processed by 4 pooling nodes using max-pooling.

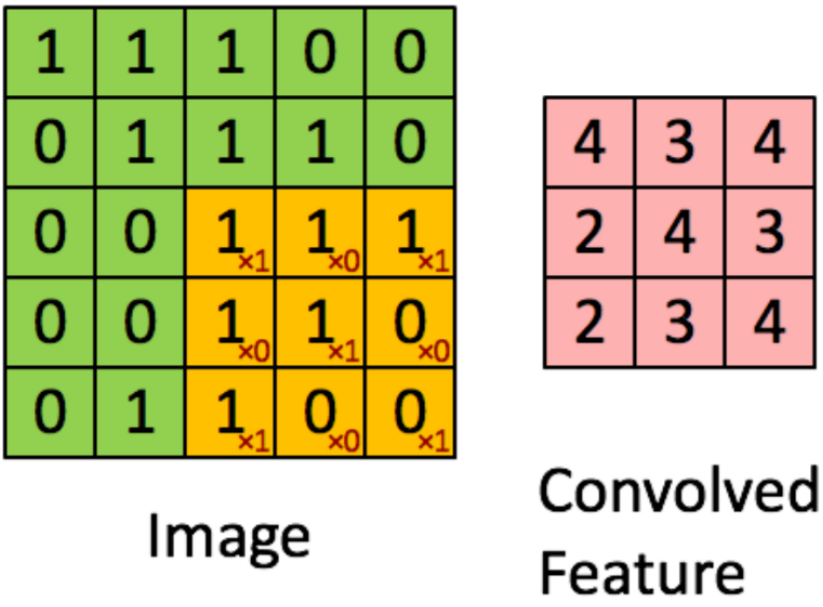
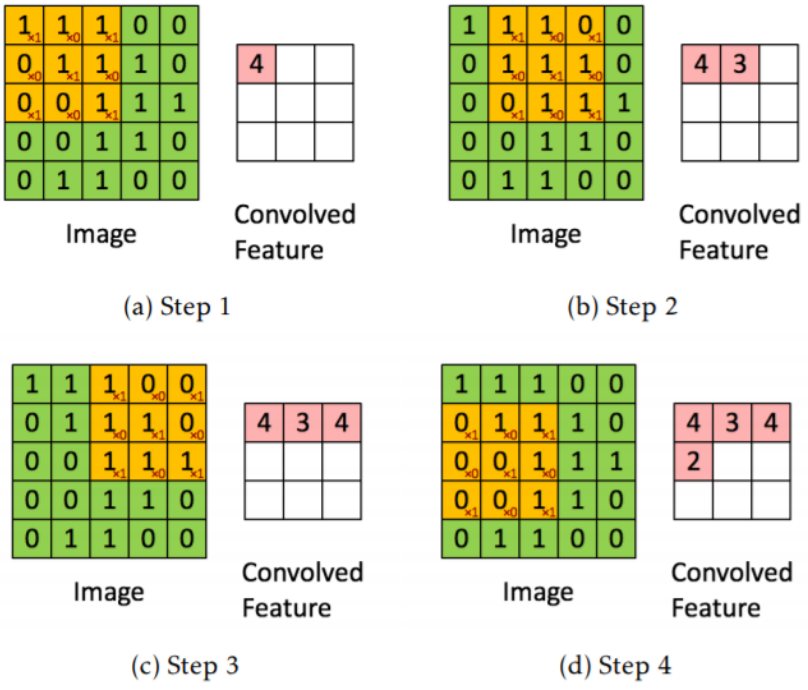


Figure 2.26: **Simple 2D convolutional layer example**: The matrix represents a black and white image, where black is represented by 0 and white by 1. The 3×3 yellow window is a filter which slides over the image. In this example the filter multiplies its values element-wise with the part of the image it covers, and sends that sum to the "Convolved Feature", which is the matrix it will send to the convolution node it is connected to. The step-wise convolution update given the filters iteration over the image is shown in Figure 2.27



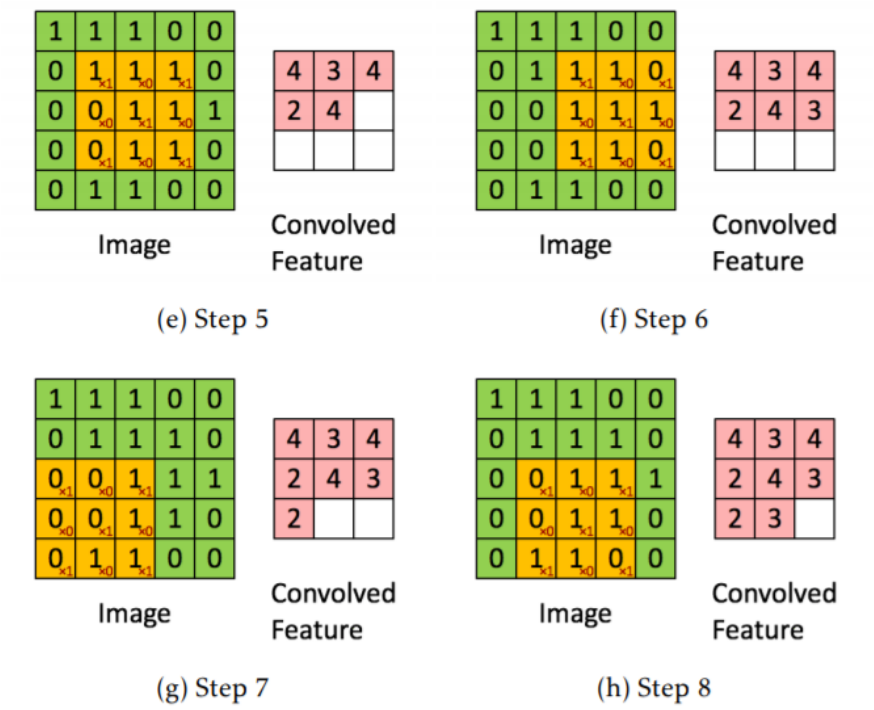


Figure 2.27: The stepwise convolution update given the filters iteration over the image is shown in this figure, see image 2.26

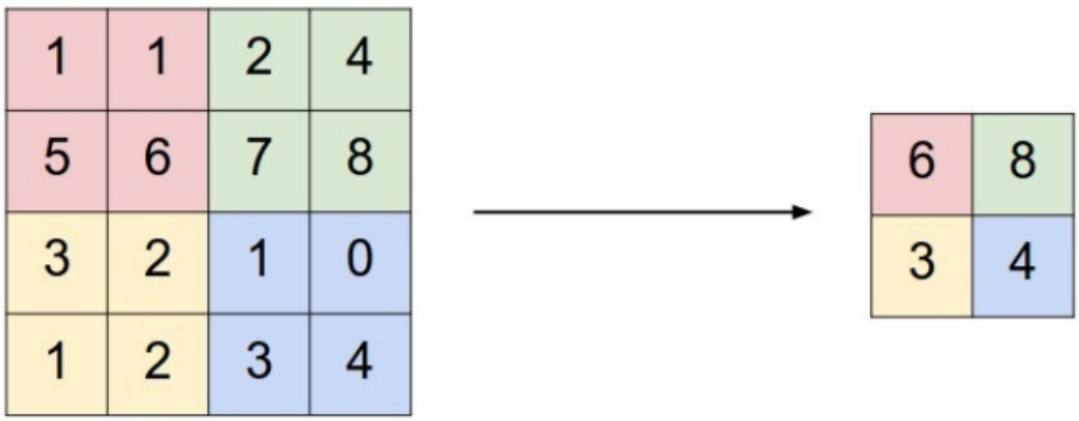


Figure 2.28: A pooling example where each colored area refers to a separate convolution node. The input matrix consists of 4 convolution nodes which are processed by 4 pooling nodes using max-pooling

Filters learning During the training phase, a CNN automatically learns the values of its filters based on the task you want to perform. Each layer applies different filters. For example, in Image Classification a CNN may learn to detect

edges from raw pixels in the first layer, then use the edges to detect simple shapes in the second layer, and then use these shapes to deter higher-level features, such as facial shapes in higher layers. The last layer is then a classifier that uses these high-level features. When training a CNN from scratch, the filters elements of the layers are usually initialized from a gaussian distribution. This is random. Training is the procedure of adjusting the values of these elements. Given an input, all the layers elements effectively constitute a transformation of this input to a predicted output. The measure of variation between this predicted output and the actual output is defined as loss. The value of this loss is used to adjust the values in the filters to effectively minimize the difference between predicted and actual output. This way the value of the filters are adjusted during training and system is said to have converged when the loss is minimized.

2.6 OpenAPI Specifications

The OpenAPI Specification [3] (formerly known as the Swagger Specification) defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. When properly defined, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. An OpenAPI definition can then be used by documentation generation tools to display the API, code generation tools to generate servers and clients in various programming languages, testing tools, and many other use cases.

2.6.1 Definitions

OpenAPI Document A document (or set of documents) that defines or describes an API. An OpenAPI definition uses and conforms to the OpenAPI

Specification.

Path templating Path templating refers to the usage of curly braces {} to mark a section of a URL path as replaceable using path parameters.Example: if you specify the following path */pets/{petId}*, with a value of $petId = 1$, when you are making the request the corrispondent path becomes */pets*/1

Media types Media type definitions are spread across several resources. The media type definitions sgould be in compliance with RFC6838 [24]. Some examples of possible media type definitions: text/plain, charset=utf-8, application/json, application/vnd.github+json, application/vnd.github.v3+json, application/vnd.github.v3.raw+json, application/vnd.github.v3.text+json, ...

HTTP Status Code The HTTP Status Codes are used to indicate the status of the executed operation. The available status codes are defined by [20].

2.6.2 Specification

There are 8 base objects that correspond to the 8 root-level objects of a complete OpenAPI documentation (that corresponds to the OpenAPI Object):

1) **openapi** (required): contains the OpenAPI version used for the API (2.0.0 or 3.0.0);

2) **info object** (required): contains base informations about API, like name, description, license, contacts, ...

3) **servers object** (optional): contains URL basepaths used to make API requests;

4) **components object** (optional): contains objects that usually are going to be used more than once in OpenAPI documentation, like: parameter object, response object, example object, schema object,... Defining an object in the com-

ponent object allows to you to refer to it in other objects, so you don't need to specify this same object again, but if you need you can still override this definition with a new one;

5) **paths object** (required): specifies API end-points and their operations (these end-points are going to be appended to basepaths, if existing, in order to obtain full paths). The end-point needs to start with a / and it is allowed to use Path templating;

6) **security requirement object** (optional): contains the names of security schemes objects, that allow to define API authentication mechanisms which are HTTP authentication, OAuth2 or an APIkey;

7) **tags object** (optional): allows to define tags;

8) **externalDocs object** (optional): allows to define links to external documents that can help consumers to understand better the API.

All these objects are composed by other objects or simple components that you need to specify in order to obtain OpenAPI documentation.

2.6.3 Info Object

Provides metadata about the API. The metadata may be used by tooling as required. It is composed by:

1) **title** (required): is a *string* that represents the application name;

2) **description** (required for this work, but not for OpenAPI specification): obviously a natural language description in CommonMark sintax, see [9] for more informations;

3) **termsOfService** (optional) (string): a URL to the Terms of Service for the API;

4) **contact** (optional): the contact information for the exposed API. For every contact developer can specify: *name*, *url* and *email* (all optionals and all *string*);

5) **license** (optional): the license information for the exposed API. If developer wants to specify a license needs to indicate a *name* and optionally an *URL* (both *string*);

6) **version** (required): the version of the OpenAPI document (which is distinct from the OpenAPI Specification version or the API implementation version). It is used *major.minorportion*. Typically, *.patch* versions address errors in this document, not the feature set. Tooling which supports OAS 3.0 SHOULD be compatible with all OAS 3.0.* versions. The patch version SHOULD NOT be considered by tooling, making no distinction between 3.0.0 and 3.0.1 for example.

YAML

```yaml
title: Sample Pet Store App
description: This is a sample server for a pet store.
termsOfService: http://example.com/terms/
contact:
  name: API Support
  url: http://www.example.com/support
  email: support@example.com
license:
  name: Apache 2.0
  url: http://www.apache.org/licenses/LICENSE-2.0.html
version: 1.0.1
```

Figure 2.29: An Info Object example

2.6.4 Servers Object

Servers object is optional and it is recommended to be specified only if there are some basepaths used more than once in API calls. If not, it' s more simple to specify this information directly in path object, as will be described soon. Servers object is obviously a collection of server objects, which are composed by:

1) **URL** (required) (string): a URL to the target host. This URL supports Server Variables and MAY be relative, to indicate that the host location is relative to the location where the OpenAPI document is being served. Variable substitutions will be made when a variable is named in {}.;

2) **description** (optional): an optional string describing the host designated by the URL;

3) **variables** (optional): a map between a variable name and its value. The value is used for substitution in the server's URL template. Some variable examples are: port, username, ... For every variable the developer needs to specify a *name* (string), the *default value* (string) and optionally *enum*: all the possible values for this variable (strings) and a *description*;

```yaml
servers:
- url: https://{username}.gigantic-server.com:{port}/{basePath}
  description: The production API server
  variables:
    username:
      # note! no enum here means it is an open value
      default: demo
      description: this value is assigned by the service provider, in this example `gigantic-server.co
    port:
      enum:
        - '8443'
        - '443'
      default: '8443'
```

Figure 2.30: A Server Object example

2.6.5 Components Object

Components object is another optional object: it is recommended to be specified if you have some common objects, like parameters, responses, schemes,... that are going to be used more than once in the documentation. It can contain: schema objects, response objects, parameter objects, example objects, requestBody objects, header objects, securityScheme objects, link objects and callback objects (some of these will be described in this section). All these can also just be a reference to the real object: to make this link you can use a reference object.

2.6.6 Reference Object

A simple object to allow referencing other components in the specification, internally and externally. It has only one component:

1) **$ref** (required): is a *string* indicating the referred object path in documentation.

```yaml
$ref: '#/components/schemas/Pet'
```

Figure 2.31: A Reference Object example

2.6.7 Schema Object

The Schema Object allows the definition of input and output data types. These types can be objects, but also primitives and arrays. It can contain the following properties (all optional) taken from the JSON Schema definition but their definitions were adjusted to the OpenAPI Specification:

1) **type** - Value MUST be a string. Multiple types via an array are not supported. The allowed types are: integer, long, float, double, string, byte, binary, boolean, date, dateTime, password.

2) **items** - Value MUST be an object and not an array. items MUST be present if the type is array.

3) **properties** - Property definitions MUST be a Schema Object and not a standard JSON Schema (inline or referenced).

4) **additionalProperties** - Value can be boolean or object.

5) **description** - CommonMark syntax MAY be used.

6) **default** - The default value represents what would be assumed by the consumer of the input as the value of the schema if one is not provided.

7) **format** - the allowed formats are: int32, int64, float, double, byte, binary,

date, date-time, password.

```yaml
type: string
format: email
```

Figure 2.32: A Schema Object example for a simple type

```yaml
type: object
required:
- name
properties:
  name:
    type: string
  address:
    $ref: '#/components/schemas/Address'
  age:
    type: integer
    format: int32
```

Figure 2.33: A Schema Object example for an object type

```yaml
animals:
  type: array
  items:
    type: string
```

Figure 2.34: A Schema Object example for an array type

2.6.8 Parameter Object

Describes a single operation parameter. A unique parameter is defined by a combination of a name and location. A parameter object contains the following fields:

1) **name** (required): a *string* indicating the name of the parameter;

2) **in** (required): a *string* indicating location of the parameter. Developer can choose between four choices: *path* if this parameter is gonna be used with path templating in order to complete paths; *query* if this parameter is appended to URL (example: /pets with petId=1 becomes /pets?petId=1); *header* if this parameter

is expected in the request headers and *cookie* if this parameter value is going to be taken from a cookie;

3) **description** (required for this work, but not for OpenAPI specification): a *string* indicating a brief description of the parameter;

4) **required** (required): a *boolean* indicating if a parameter is required (true) or optional (false) in order to make the request;

5) **deprecated** (optional): a *boolean* indicating if a parameter is deprecated (true) or not (false);

6) **allowEmptyvalue** (optional): a *boolean* indicating if a parameter can be an empty value (true) or not (false);

7) **style** (optional): a *string* that describes how the parameter value will be serialized depending on the type of the parameter value. Default values (based on value of in): for query - form; for path - simple; for header - simple; for cookie - form. The possible values are: matrix, label, form, simple, spaceDelimited, pipeDelimited, deepObject.

8) **explode** (optional): a *boolean* value to indicate if the parameter, that is an array or an object, needs to be separated when transmitted;

9) **allowReserved** (optional): a *boolean* value to indicate if the parameter value can contain special values like ?, /, :, [], ...;

10) **schema**: is a schema object that allows to define a parameter type, it can be also a reference to a schema object defined in the component object. If it is a simple type, like string, integer, boolean, ... then you don't need to specify anything else, but if it's an *array* you need to specify also which is the type of array components or if it is an *object*, you need to specify all the parameter names and their relative types that make this object (as illustrated in response object section);

11) **example** (optional): an example inline or a reference to an example specified in the component object (as shown in schema object section).

```yaml
name: id
in: query
description: ID of the object to fetch
required: false
schema:
  type: array
  items:
    type: string
style: form
explode: true
```

Figure 2.35: A Parameter Object example

2.6.9 Response Object

Describes a single response from an API Operation. It contains:

1) **description** (required): a short description of the response.

2) **headers** (optional): maps a header name to its definition. The header object follows the structure of a Parameter Object with the following changes: *name* MUST NOT be specified, it is given in the corresponding headers map; *in* MUST NOT be specified, it is implicitly in header; all traits that are affected by the location MUST be applicable to a location of header (for example, style).

3) **content** (optional): a map to one or more media type object containing descriptions of potential response payloads. A media type object is a schema object with some examples (optional) and an encoding object(optional). The example can be inline or a reference to an example object, which contains the following components (all optional): *summary* (string), a short description for the example; *description* (string), a long description for the example; *value* (any): contains the possible example values for that input/output; *externalValue* (string), a URL that points to an example value definition that can't be represented in value field. The encoding object contains the following field: *contentType*, default value depends on the property type: for string with format being binary – application/octet-stream; for other primitive types – text/plain; for object - application/json; for

array – the default is defined based on the inner type. The value can be a specific media type (e.g. application/json), a wildcard media type (e.g. image/*), or a comma-separated list of the two types.

```yaml
# in a model
schemas:
  properties:
    name:
      type: string
      examples:
        name:
          $ref: http://example.org/petapi-examples/openapi.json#/components/examples/name-example

# in a request body:
  requestBody:
    content:
      'application/json':
        schema:
          $ref: '#/components/schemas/Address'
        examples:
          foo:
            summary: A foo example
            value: {"foo": "bar"}
          bar:
            summary: A bar example
            value: {"bar": "baz"}
```

Figure 2.36: An Example Object example

```yaml
description: A simple string response
content:
  text/plain:
    schema:
      type: string
    example: 'whoa!'
headers:
  X-Rate-Limit-Limit:
    description: The number of allowed requests in the current period
    schema:
      type: integer
```

Figure 2.37: A Response Object example

2.6.10 RequestBody Object

Describes a single request body. It contains the following fields:

1) **description** (optional): a brief description of the request body;

2) **content** (required): the content of the request body. It is a map between a string and a media type object (shown in response section).

3) **required** (otional): a *boolean* value that determines if the request body is required in the request. Default value is false.

2.6.11 Security Requirement Object

Lists the required security schemes to execute this operation. The name used for each property MUST correspond to a security scheme declared in the Security Schemes under the Components Object.

A security scheme object defines a security scheme that can be used by the operations. Supported schemes are HTTP authentication, an API key (either as a header or as a query parameter), OAuth2's common flows (implicit, password, application and access code) and OpenID Connect Discovery.. It contains the following fields:

1) **type** (required): the type of the security scheme. Valid values are "apiKey", "http", "oauth2", "openIdConnect";

2) **description** (optional): a short description of security scheme;

3) **name** (required only for apiKey): the name of the header, query or cookie parameter to be used;

4) **in** (required only for apiKey): the location of the API key. Valid values are "query", "header" or "cookie";

5) **scheme** (required only for HTTP): The name of the HTTP Authorization scheme to be used in the Authorization header.

6) **flows** (required only for oauth2): an object containing configuration information for the flow types supported. This object contains: *authorizationUrl* (required for implicit and authorizationCode), *tokenUrl* (required for password,

clientCredentials and authorizationCode), *scopes*, the available scopes for the OAuth2 security scheme. A map between the scope name and a short description for it.

7) **flows** (required only for openIdConnect): OpenId Connect URL to discover OAuth2 configuration values. This MUST be in the form of a URL.

```yaml
type: oauth2
flows:
  implicit:
    authorizationUrl: https://example.com/api/oauth/dialog
    scopes:
      write:pets: modify pets in your account
      read:pets: read your pets
  authorizationCode:
    authorizationUrl: https://example.com/api/oauth/dialog
    tokenUrl: https://example.com/api/oauth/token
    scopes:
      write:pets: modify pets in your account
      read:pets: read your pets
```

Figure 2.38: An Oauth Flow Object example

```yaml
type: apiKey
name: api_key
in: header
```

Figure 2.39: A Security Scheme Object example

2.6.12 ExternalDocs Object

Allows referencing an external resource for extended documentation. It contains only two fields:

1) **description** (optional): a description of the external document;

2) **URL** (required): the URL for the target documentation.

```yaml
description: Find more info here
url: https://example.com
```

Figure 2.40: An ExternalDocs Object example

2.6.13 Tags Object

Adds metadata to a single tag that is used by the operation object (in next section). These objects are fundamental for this work, as it will be explained later. It contains:

1) **name** (required): the name of the tag;

2) **description** (optional, but required for this work): a short description of the tag.

```yaml
name: pet
description: Pets operations
```

Figure 2.41: An Tags Object example

2.6.14 Paths Object

Holds the relative paths to the individual endpoints and their operations. The path is appended to the URL from the Server Object in order to construct the full URL. This is a collection of Path Item Object, which contains:

1) **path** (required): relative path to an individual endpoint. The field name MUST begin with a slash. The path is appended (no relative URL resolution) to the expanded URL from the Server Object's url field in order to construct the full URL;

2) **summary** (optional);

3) **description** (optional);

4) one or more **Operation Objects**;

5) one or more Parameter Object (common to all the operations under this path).

An Operation Object is formed by:

1) **method name** (required only for this work, optional for OpenAPI specifications under the name of operationId);

2) **operation** (required): the HTTP operation related to this path, like get, post, put, delete, ...;

3) **summary** (optional): a method summary description;

4) *description* (required for the other this work, but optional for OpenAPI specifications): a method description;

5) **request body** (optional): it can be a reference to a request body object in component object or developer needs to specify a new one;

6) **responses** (required): it can be a reference to a response object in component object or developer needs to specify a new one;

7) **parameters** (optional): actually this method could not have any parameter, but if it has also only one parameter, developer needs to fill this object with a reference to a parameter object in component object or by adding a new parameter object, as described before;

8) **servers** (optional): if developer has already specified in servers object the basepath for this path, then he can just avoid to provide again this information, but if not he needs to define a new server object, as described before;

9) **deprecated** (optional);

10) **externalDocs** (optional): it can be a reference to an externalDocs object in component object or developer needs to define a new one.

11) **tags** (optional, but required for this work): it can be a reference to a tag object or can be specified a new one.

```yaml
/pets:
  get:
    description: Returns all pets from the system that the user has access to
    responses:
      '200':
        description: A list of pets.
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/pet'
```

Figure 2.42: A Paths Object example

```yaml
get:
  description: Returns pets based on ID
  summary: Find pets by ID
  operationId: getPetsById
  responses:
    '200':
      description: pet response
      content:
        '*/*' :
          schema:
            type: array
            items:
              $ref: '#/components/schemas/Pet'
    default:
      description: error payload
      content:
        'text/html':
          schema:
            $ref: '#/components/schemas/ErrorModel'
parameters:
- name: id
  in: path
  description: ID of pet to use
  required: true
  schema:
    type: array
    style: simple
    items:
      type: string
```

Figure 2.43: A Path Item Object example

Chapter 3

Defined Procedure

This work basically tries to achieve two goals:

- **Help consumer to choose the Cloud Service that suits his needs**
- **Help consumer to use the Identified Cloud Service's API**

In order to reach these two goals this work proposes the following procedure, trying to be user-friendly as much as possible in this approach. Let's begin from start: when a user needs to identify a cloud service it' s most likely because he needs a certain service or because he needs to perform a particular task. So we can actually think that there exist some services and methods categories that can be defined as *Agnostic*, because are independent from vendor definitions, that can be used in order to identify which services and which between their methods to use to fulfill consumer needs. In fact, service providers describe similar services with different manners (different names for services belonging categories, different names for services functionalities, etc.); such heterogeneous descriptions complicate service comparison and selection, and may create interoperability problems among multiple providers (especially when migrating from a provider to another).

To overcome the problems stemming from heterogeneous service descriptions this work proposes to define a common ontology for Cloud Services description. It

is worth nothing that, there exists some industrial standards and research projects that treat the aforementioned problems; unfortunately, current propositions focus only on the IaaS layer. That is, they do not offer solutions to the PaaS and the SaaS layers. The only work that actually tries to do something similar to this work is **Mosaic** project, that has defined an ontology ,described in [13], where there exist these Agnostic Categories mentioned before. This allows to represent all cloud services, regardless of their belonging provider and this can be used to easily and automatically find which cloud services are needed by consumer, simply finding service that are classified as the specific agnostic category chosen by the user. In order to make this classification are used **Multi Layer Neural Networks**, **Convolutional Neural Networks** and **Multinomial Naive Bayes Classifiers**, opportunely trained and using as input services' natural language descriptions.

Why use all these different classifiers? Because a training dataset enough big to train neural networks exists (or better it was possible to build) only for services categories thanks to OpenCrowd Cloud Taxonomy [48]. But these categories can be considered as macro-categories and actually are not enough to achieve the first goal of this work, because each of them contains other several micro-categories that can be associated to agnostic categories previously defined, the ones that are needed for first goal. Instead for these agnostic categories it was possible to build only small training datasets, not enough for a neural network but for a multinomial Naive Bayes classifier. This work presents both approach in order to make a comparison between them and as a robustness technique: the component "classifier" is replicated in order to reduce errors. Indeed, offering to developer multiple choices and the possibility of choose another agnostic category, different from the resulting ones, makes more robust the system.

However, in this ontology the defined Agnostic categories are old or not enough to represent the actual Cloud Services' world. So, at first, it was thought to define

new Agnostic categories that could represent the nowadays situation, but this huge work would have been obsolete too in few years. Hence the idea of updating periodically and automatically this ontology. Furthermore, the idea of representing all the knowledge related to Cloud Services only thanks to these agnostic concepts it's insane, because this world changes too fast and it's too big and variegated. Therefore this work defines a strategy in order to overcome these problems: associate to every service, in another ontology, *provider specific* and related to the agnostic one (all the ontologies and the links between them will be described in details in the next chapter), specific concepts that can represent functionalities not already present in the ontology. These specific functionalities are extracted from natural language descriptions using **RAKE** or can be inserted by developer. So, when a user needs to make a particular task, if the research between agnostic concepts fails, this work proposes to search for this task between specific functionalities present in the ontology. Obviously, in order to don't put in these provider specific ontologies equivalent concepts with different names, this work proposes to measure a **score similarity** between the new specific concepts that should be inserted and the already existing ones. If this score is over a certain limit (0.7), then it is suggested to developer that already exists, in that specific provider ontology, a concept very similar to this new one that he wants to insert and so he can actually choose to don't insert this new concept, but choose the already existing one. Actually this new concept is inserted anyway, but with a link to the similar concept present in the other provider specific ontology. When periodically these ontologies are updated, if there is a specific functionality that is linked to at least another one, then this can be considered, for the definition of Agnostic (provider independent), an agnostic concept. So this specific functionality it's removed from provider specific ontologies and it's put in the agnostic ontology. Obviously, now it's needed also to update training dataset of classifiers by adding this new agnostic concept: being new, it can not be expected to have a sufficiently large training

dataset to well train a neural network, so only Multinomial Naive Bayes classifier can be used in a first step. When a more large dataset will be ready, then you can think of updating also Neural Network Classifiers. This "provider specific" ontology is needed in order to help consumer to find:

- services that offer a method that is classified as particular agnostic concept;
- services that are associated to a particular specific concept;
- services that offer a method that is associated to a particular specific concept;

In the Mosaic project there is also another ontology for every service that represents it semantically, based completely on **OWLS**, a semantic markup language used to describe Web Services based on **SOAP** (Simple Object Access Protocol). SOAP is a protocol specification for exchanging structured information in the implementation of web services in computer networks. It uses XML Information Set for its message format, and relies on application layer protocols, most often Hypertext Transfer Protocol (HTTP) or Simple Mail Transfer Protocol (SMTP), for message negotiation and transmission. Since Web protocols like HTTP are installed and running on all operating systems, SOAP allows clients to invoke web services and receive responses independent of language and platforms. The problem is that nowadays SOAP is no longer used, because it has been replaced from **REST** (described in previous chapter, section 2.2). The first clear difference between the two types of Web Service is the vision of the Web proposed as a processing platform. REST offers a web vision focused on the concept of *resource* while SOAP Web Services highlight the concept of *service*. A RESTful Web Service is the custodian of a set of resources on which a client can request the canonical operations of the HTTP protocol. A SOAP-based Web Service exposes a set of methods that can be recalled remotely by a client.

The approach of the SOAP Web services has borrowed an application architecture called **SOA**, Service Oriented Architecture, which has recently opposed

the **ROA** architecture, Resource Oriented Architecture, inspired by the REST principles.

The Simple Object Access Protocol (SOAP) defines a data structure for the exchange of messages between applications, in a certain sense re-proposing part of what the HTTP protocol already did. SOAP uses HTTP as the transport protocol, but it is not limited or bound to it, since it can very well use other transport protocols. Unlike HTTP, however, the SOAP specifications do not address topics such as **security** or **addressing**, for which separate standards have been defined, specifically WS-Security and WS-Addressing. So SOAP does not fully exploit the HTTP protocol, using it as a simple transport protocol. REST, on the other hand, uses HTTP for what it is, an application layer protocol, and fully utilizes its potential.

It is evident that the approach adopted by SOAP-based Web Services is derived from interoperability technologies existing outside the Web and based essentially on remote procedure calls, such as DCOM, CORBA and RMI. Basically this approach can be seen as a sort of adaptation of these technologies to the Web. REST approach, instead, tends to preserve and enhance the intrinsic characteristics of the Web, highlighting its predisposition to be a platform for distributed processing. Thus, there is no need to add anything to what is already on the Web to allow remote applications to interact.

In addition, SOAP-based Web Services provide the **WSDL** standard, Web Service Description Language, to define the interface of a service. This is further evidence of the attempt to adapt the interoperability approach based on remote calls to the Web. In fact, the WSDL is nothing more than an IDL (Interface Description Language) for a software component. On the one hand, the existence of WSDL favors the use of tools to automatically create clients in a given programming language, but at the same time induces to create a strong dependence between client and server. REST does not explicitly provide any way to

describe how to interact with a resource. Operations are implicit in the HTTP protocol. Something similar to WSDL is **WADL**, the Web Application Definition Language, an XML application for defining resources, operations and exceptions provided by a REST-like Web Service. WADL has been submitted to the W3C for standardization in 2009, but at present there are no plans for its discussion and possible approval. In reality it has not had a very favorable reception by the REST community, as it offers a static view of a Web Service, contradicting the principle HATEOAS which places in the presence of links within the representation of a resource the definition of a contract with the client, with a view therefore much more dynamic and a weak coupling between client and server.

In conclusion, SOAP-based Web services build a long and intricate web infrastructure above the Web to do things the Web is already able to do. The advantage of this type of service is that it actually defines a Web-independent standard and the infrastructure can also be based on different protocols. REST, on the other hand, intends to restore the Web to architecture for distributed programming, without adding unnecessary superstructures. That's why SOAP is no longer used.

Now, returning to this work's defined procedure, we actually have this service ontology, to represent it semantically, based on OWLS, but SOAP it is no longer used. This is actually not the biggest problem, because OWLS is based on WSDL concepts and in WSDL 2.0 actually it is possible to represent in part a REST web service, as described in [1]. But REST requires a different approach because services are based on the principles of resources identified with unique and opaque URIs, that are resolved to clients in various "representations" with a media type, and are handled through a uniform interface. REST resources are interlinked through hyperlinks that are found in these representations and guide clients in their interactions with a service. So actually the need to change or replace OWLS with something more suitable for REST is apparent. In this paper [10] is presented a new metamodel for describing RESTful services and a new language for

creating descriptions of these services that is based on REST's central principle, the hyperlinking of resources. Instead In this work, it's has been tried to adapt already existing OWLS for REST services (as will be described in next chapter).

Indeed, after search for cloud services is completed, there is another problem for users: understand how to use these services through their APIs. It is not an easy task for users to correctly use these services: they have to read a lot of documentation pages to understand how to properly make calls to Restful APIs in order to receive expected outputs. Luckily, in this field there are already some standardization projects like OpenAPI Specifications. The specification creates a Restful interface for easily developing and consuming an API by effectively mapping all the resources and operations associated with it. So this work proposes an automatic way to obtain Swagger documentation of API by answering to specific questions about this API. In fact, despite the now undisputed Swagger diffusion, which can be considered a de facto standard at present, APIs producers hardly provide a description using such formalism, because developers should have a great knowledge of Swagger Specifications and must manually generate the OpenAPI file from their existing API codebase, which makes for a long, cumbersome process. Yet swagger, with a set of tools connected to it, undoubtedly provides benefits to both developers and consumers of these APIs. Among these tools surely we must count:

Swagger-Codegen: can convert OpenAPI definitions into code, saving time and effort. Examples of auto-generated code include stubs for the implementation of the server-side logic, or a complete mock server that allows the client-side developers to start working with the API even before it is done. Services like APIMatic can even use use OpenAPI definitions to create complete SDKs for developers in programming languages that nobody on developer team is fluent in.

Swagger-UI: was the first tool to offer API documentation with integrated test client, so developers could see a list of available API operations and quickly

send a request in their browser to test how the API responds before writing any code of their own. Basing documentation on a definition ensures that the documentation covers the entirety of an API and correctly lists all methods, parameters and responses.

Some attempts have already been made in this direction, but none of them has yet provided a solution that is totally human independent.: Among these should be included:

Swagger-Inspector [7]: using Inspector, developers can easily call any API end-point, and see if the response is what was expected. Inspector's simple UI is built to allow developers to test as quickly as possible without any learning curve or process change. Also Swagger Inspector allows to automatically generate the OpenAPI file for any called end-point. With less than 3 clicks, Swagger Inspector can generate OpenAPI documentation, saving valuable development time.

SmartAPI Editor [6]: this is a little fragment of a more complex project that aims to maximize the FAIRness (Findability, Accessibility, Interoperability, and Reusability) of web-based Application Programming Interfaces (APIs) using rich metadata. This is essential how to properly describe API so that it becomes discoverable, connected, and reusable. They have developed a openAPI-based specification for defining the key API metadata elements and value sets and an editor that helps developers to produce OpenAPI documentation, using predefined 'snippets' that needs to be filled from developer.

The Inspector biggest problem is that developers need to make all the possible calls that their APIs allow to do with all the possible combinations of parameters in order to obtain an half-complete OpenAPI documentation. Indeed, at least for what I could observe using it, it doesn't seem to offer a solution to describe all the possible cases that an API call could represent, like: representing object or array type parameters, specifying the 'in' parameter field, asserting if a parameter is required or not, defining a component object, a schema object, an example object

and some other stuffs. So this produced documentation could actually be used as a starting point to obtain a complete documentation, but this involves more developers hand maded work. On the other hand, SmartAPI Editor allows to produce a complete OpenAPI documentation, but there are other problems: 1) these snippets contain only the required fields for every section of the documentation, so if a developer would like to specify optional fields, he need to add them manually; 2) a developer in order to fill these snippets, should have a great knowledge of OpenAPI specification, that is required also to add the optional fields.

That's why this work proposes a new solution that tries to overcome to the problems of both presented tools, in order to easily and automatically produce a complete OpenAPI documentation with **meta-informations**, without requesting a great knowledge of OpenAPI specification, but only requiring to developers to insert, in a guided way and in specific HTML forms, information about the API.

The meta-informations are provided in documentation using the tags object and they are needed from consumer in order to understand which methods use. Let's make an example to make this more clear: let's suppose that our consumer needs a compute capacity service (that is a service offering cloud compute capacity, like Amazon EC2). So he actually can select the agnostic concept *compute-capacity* and he gets all the services that are classified under this category. Obviously these services as well as methods and parameters need to be present in the mentioned above ontologies. Therefore this work defines also a way to automatic populate these ontology, using the same information requested for produce swagger documentation. After consumer got the cloud services list, he can download, from a Github repository, the auto-generated Swagger documentation for these services and load these in Swagger-UI. Now he needs to understand which methods use of these services. Usually, first of all, when a consumer uses a compute capacity service, needs to create a virtual machine: in order to achieve this task he can select the agnostic method *create-instance* and he gets, for each service identified

in the previous step, the specific methods related to this agnostic concept or he can just search for this agnostic concept between paths tags that can be visualized very clearly thanks to Swagger-UI. Then, simply clicking on the identified method, he can insert all needed parameters in order to make the request and obviously he can make the request, by simply pressing a button.

The entire procedure, grouped for main goals, is showed in Figs. 3.1, 3.2 and 3.3, but reordered following the flow of the tool realized for this work, which will be widely described in the Implementation Chapter. This tool offers intuitive HTML forms to fill in order to achieve all the goals described in this section.

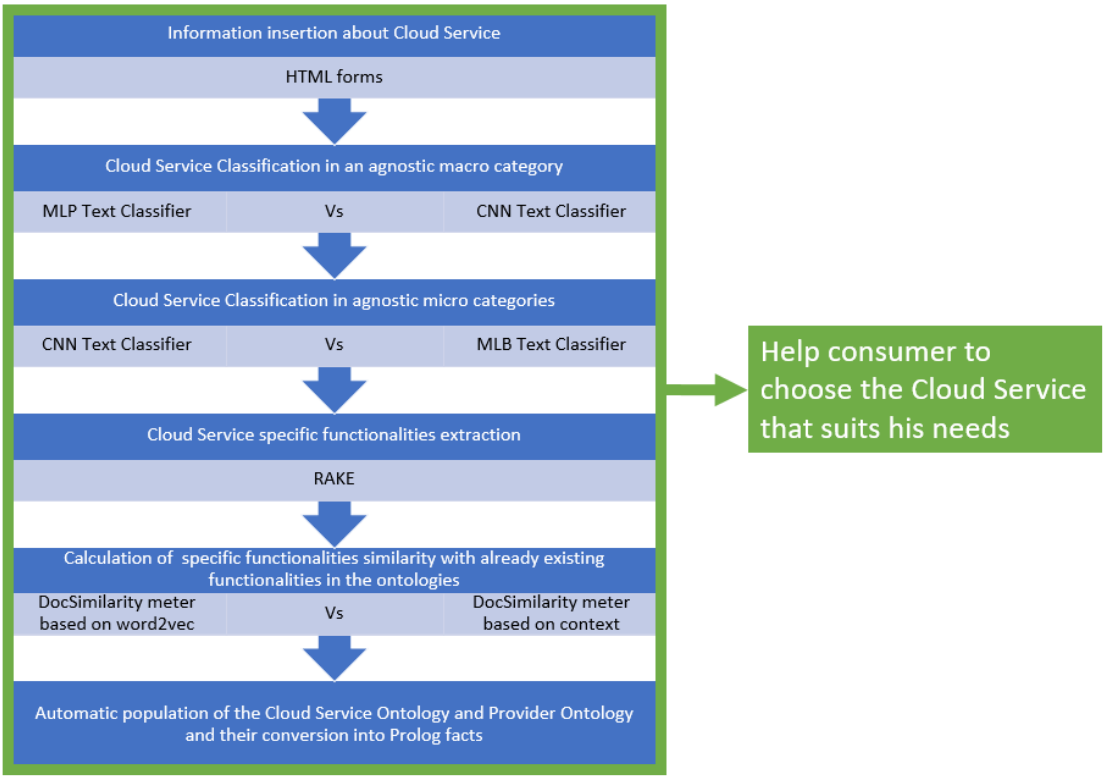


Figure 3.1: Tool work flow to automatically populate Cloud Service Ontology

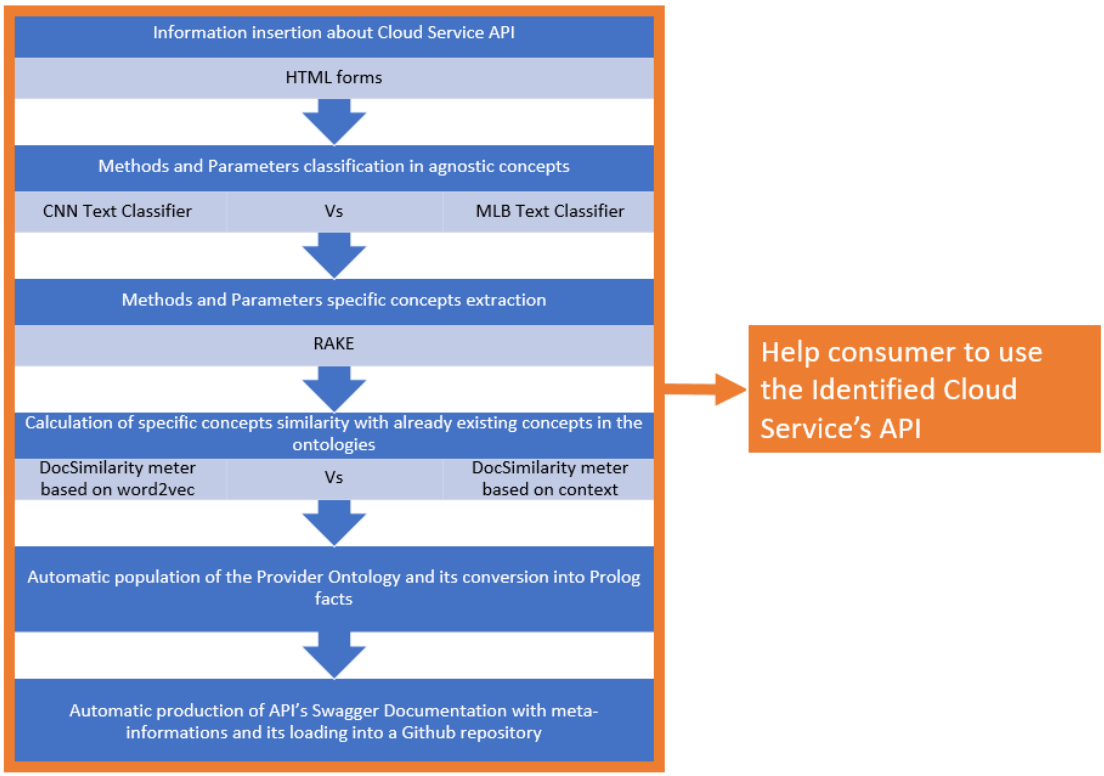


Figure 3.2: Tool work flow to automatically populate REST API Ontology and to automatically produce Swagger documentation

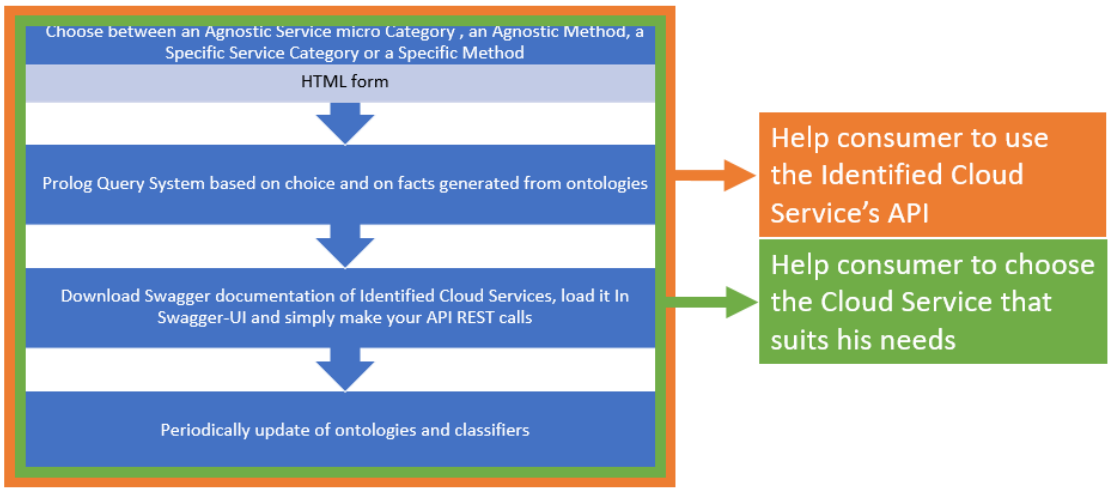


Figure 3.3: Tool work flow to automatically find cloud services needed and to use their APIs

Chapter 4

Implementation and results

In this chapter it is illustrated in details all the work steps taken to build the tool, object of this thesis, that helps cloud consumer to identify cloud services and use their APIs, and that helps developer to automatically produce Swagger documentation of an API, without requesting any knowledge about OpenAPI specifications. These work steps are showed in Figs. 4.1, 4.2: each of this will be widely discussed in following sections, showing not only how these steps have been made but also which results they have produced. It will be given greater importance to explain how the work step has been realized instead to how it is actually implemented in code.

4.1 Ontologies Build

In this section it is described the ontologies system, already mentioned in previous chapter, that allows to semantically represent cloud services and their APIs, in terms of methods and parameters, with the related agnostic and specific concepts. This system extends and modifies Mosaic project in order to make it more actual, respecting OpenCrowd Taxonomy [48] (a taxonomy of cloud services reported by NIST in [22] and described in chapter 2) and easily and automatically updated,

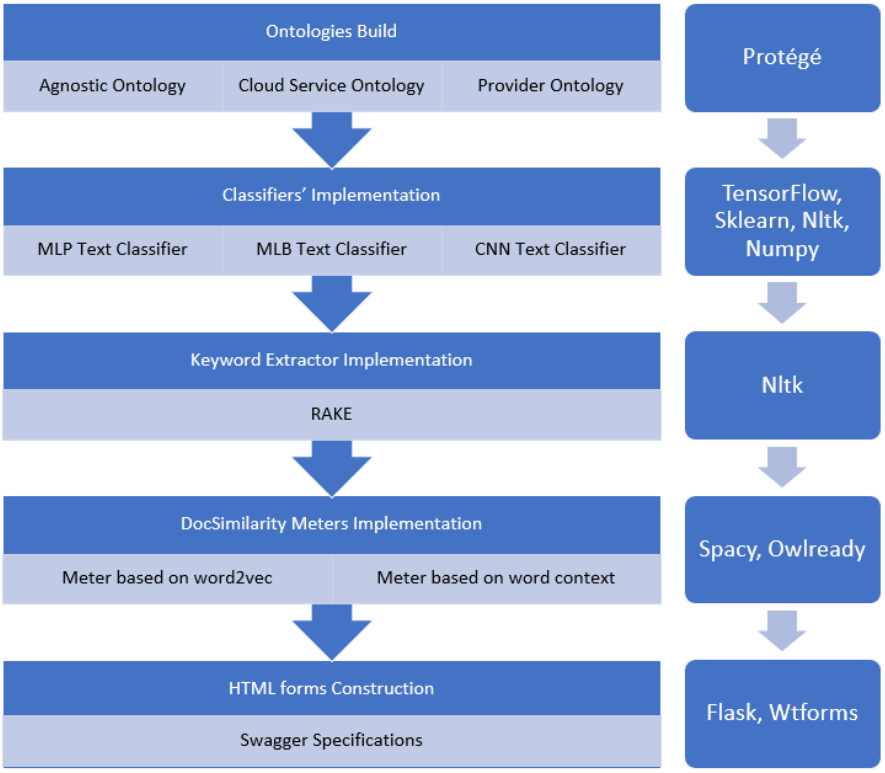


Figure 4.1: Work steps with relative used tools and/or libraries

thanks to the procedure defined in chapter 3. This system is formed by three ontologies:

- **Agnostic Ontology** (AgnosticOntology.owl): contains all the agnostic concepts that are used as research key from consumer in order to find services classified as a certain agnostic service or services that offer a method classified as certain agnostic method;
- **Cloud Services Categorization Ontology** (CSOntology.owl): contains the OpenCrowd Taxonomy and some other informations about Cloud Service that are provider independent;
- **Provider Specific Ontology** (Provider.owl): contains all services offered by that specific provider and for every service all its methods and parameters. It contains also the "specific functionalities", mentioned in chapter 3.

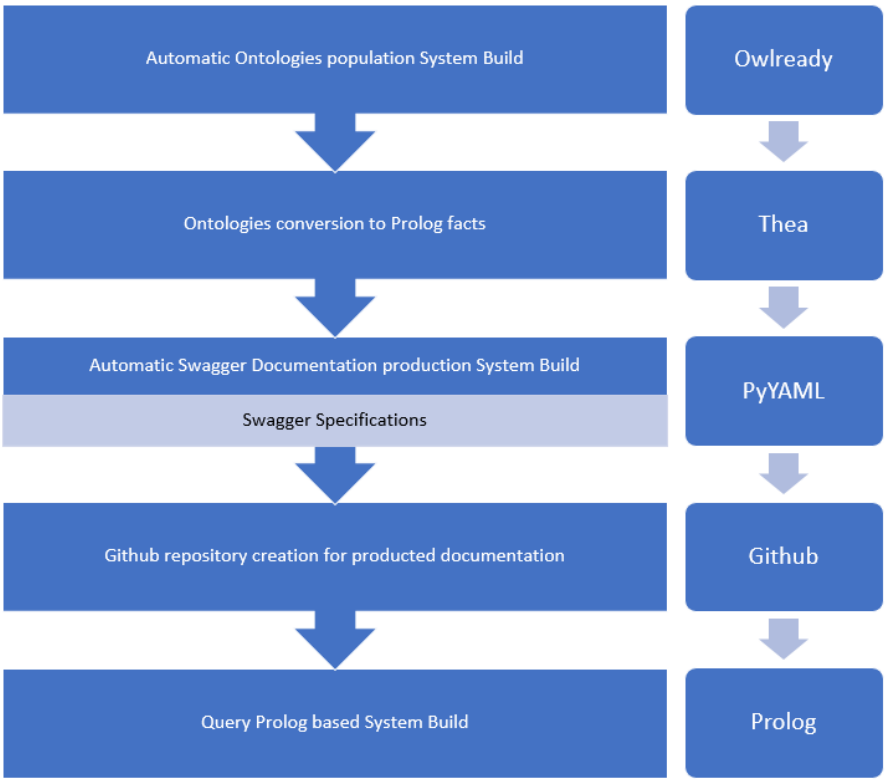


Figure 4.2: Work steps with relative used tools and/or libraries

4.1.1 Protégé

Protégé is a free, open-source platform that provides a growing user community with a suite of tools to construct domain models and knowledge-based applications with ontologies.

Protégé Desktop is a feature rich ontology editing environment with full support for the OWL 2 Web Ontology Language, and direct in-memory connections to description logic reasoners like HermiT and Pellet.

Protégé Desktop supports creation and editing of one or more ontologies in a single workspace via a completely customizable user interface. Visualization tools allow for interactive navigation of ontology relationships. Advanced explanation support aids in tracking down inconsistencies. Refactor operations available including ontology merging, moving axioms between ontologies, rename of multiple entities, and more.

4.1.2 Agnostic Ontology

It is illustrated in Fig. 4.3,4.4 and it is formed by three classes:

- *AgnosticService*: contains all the agnostic services defined or found with periodical updates, such as "compute capacity", "container orchestration", ...
- *AgnosticMethod*: contains all the agnostic methods defined or found with periodical updates, such as "create instance", "run instance", "stop instance", ...
- *AgnosticParameter*: contains all the agnostic parameters defined or found with periodical updates, such as "template name", "template info", "cluster name", ...

and two object properties:

- *hasMethod*: links an agnostic service to its agnostic methods. This relation is useful to understand which are the common methods that a service classified as a certain agnostic category should have;
- *hasParameter*: links an agnostic method to its agnostic parameters. Also this relation is useful to understand which are the common parameters that a method classified as a certain agnostic category should have.

These two relations together help to improve interoperability between different providers. Indeed let's suppose to have the following situation: a cloud consumer needs to deploy an application on a container and then store its image in a registry. So he needs these two agnostic services: "container orchestration" and "conatiner registry"; and these three agnostic methods: "create cluster", "create registry" and "put image". These agnostic services correspond to Amazon ECS or Azure Container Service and Amazon ECR or Azure Container Registry. Instead, the

agnostic methods correspond to *CreateCluster* of Amazon ECS, *CreateRepository* and *PutImage* of Amazon ECR and to *Create* of Azure Container Service on resource clusters, *Create* and *Put* of Azure Container Registry on resource registry. Our consumer can actually choose one provider for both services or choose one service from a provider and one from another. Indeed, he can choose, for example, Amazon ECS and Azure Container Registry: then the association between agnostic methods and specific one it' s not a problem of our consumer, because he just needs to know that for his tasks he needs to use the agnostic methods "create cluster", "create registry" and "put image", whatever it is the specific provider names given to these methods. After in order to store the image he needs to give as input the agnostic concept "image name". This corresponds for Amazon ECR to *registryID* and for Azure Container Registry to *imageName*, but our consumer no needs to know this details thanks to agnostic concepts. He just needs to know that he needs to specify the "image name", whatever it is the name used by provider. Obviously there are some exceptions, because not all services presents methods or parameters that can be classified as agnostic concepts. In that case the only walkable way is that consumer specifies the specific parameters requested, using as help the specific functionalities associated to them, that are not contained in this ontology, but in the Provider Specific Ontology.

4.1.3 Cloud Services Categorization Ontology

It is illustrated in Fig. 4.5, 4.6 and contains the following classes:

- *CloudProvider*: contains a list of all the possible cloud services providers, such as Amazon, IBM, Azure, ...
- *CloudService*: contains all the cloud services, independently from their providers, such as Amazon EC2, Azure Container Service, ...

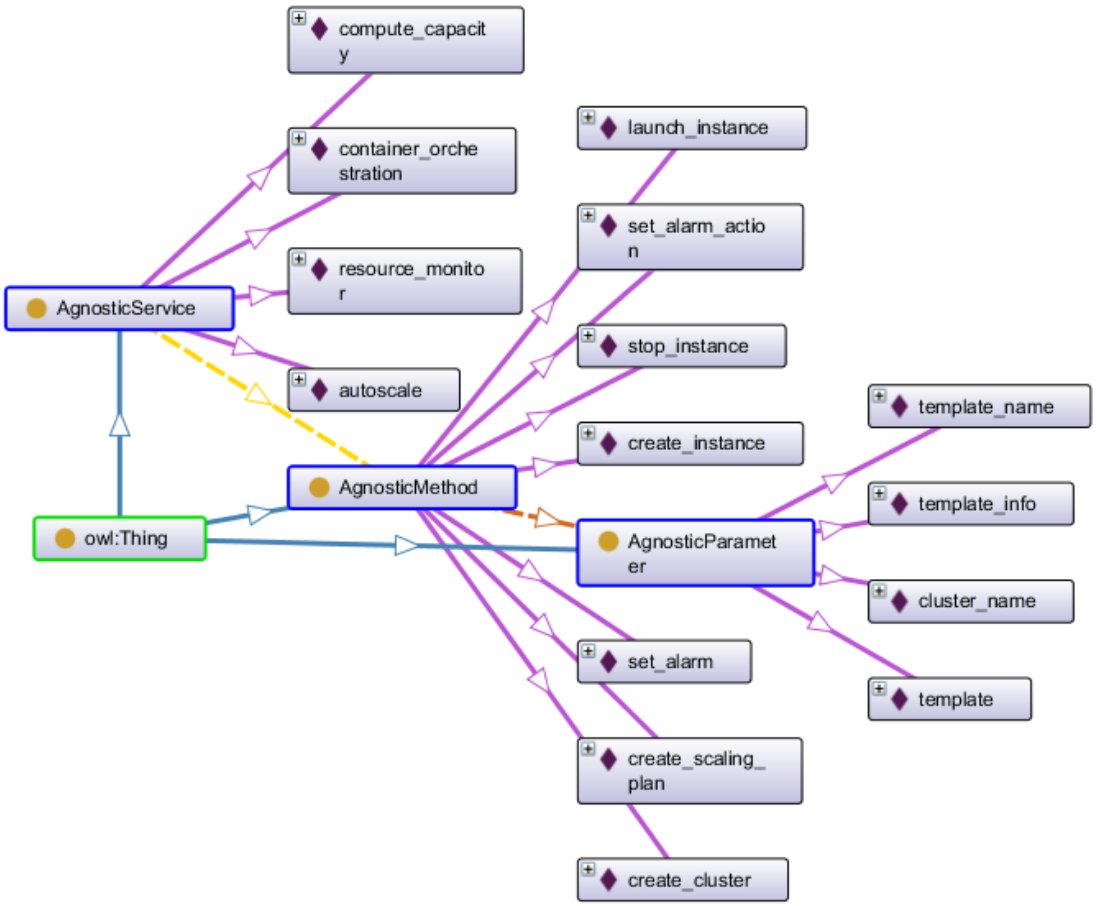


Figure 4.3: Agnostic Ontology with some agnostic concepts (represented by individuals)

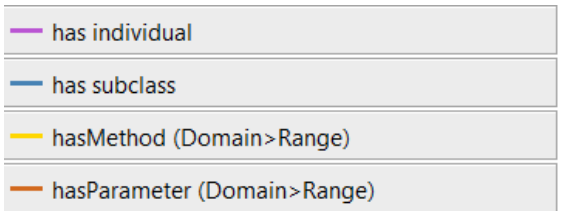


Figure 4.4: Agnostic Ontology: Object properties legend

- *ModelCloudService*: contains the three possible models of a cloud service: IaaS, PaaS and SaaS;

- *ServiceCategory*: it is a superclass that contains, as subclasses, all the service categories defined by OpenCrowd Taxonomy (already described in chapter 2). Each of these macro-category contains, as individuals, several agnostic

service (the same defined in the Agnostic Ontology), in order to make more simple the research. Indeed, if a consumer needs an "autoscale" service, if we provide to him all the services that can be classified as "Compute" helps him in the research, but he still need to read all the descriptions of these services in order to find the one that is really an autoscale service and after compare them. Instead, if we provide to consumer only the services classified as autoscale, restricts the research field and this helps consumer a lot, because now he just need to compare them.

the following object properties:

- *aKindOf*: links every real cloud service to an agnostic service;
- *isModel*: links every cloud service to a model;
- *offeredBy*: links every cloud service to its provider.

and the following data property:

- *sameIndividual*: this property it is used to create a relation between the different ontologies. Indeed the only way to relate individuals of two different ontologies, without importing an ontology in another, is to create a datatype property, put its range as anyURI and then specify the complete URI of the individual at which the individual in question is related to. In particular this property is needed in order to assert that two individuals, belonging to different ontologies, are the same individual. It is applied to agnostic service in this case: indeed the agnostic services that we can find under ServiceCategory class of this ontology are the same that we can find under AgnostiService class of Agnostic Ontology. Also this property is used to relate individuals under CloudService class of this ontology to individuals under ProviderService class of Provider Ontology, because they are effectively same individuals.

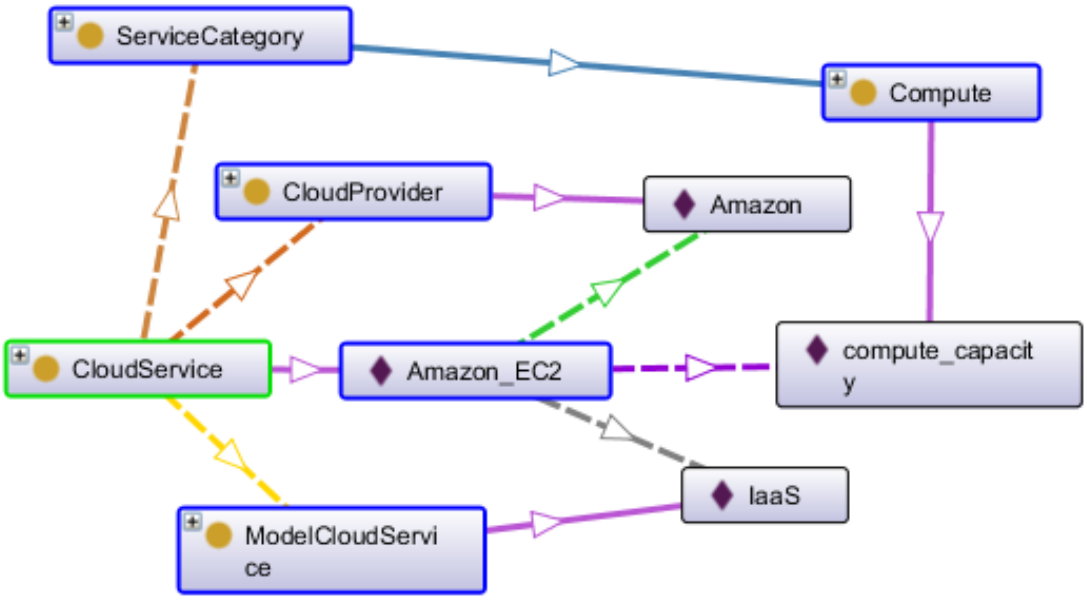


Figure 4.5: Cloud Service Categorization Ontology, with an example of a service

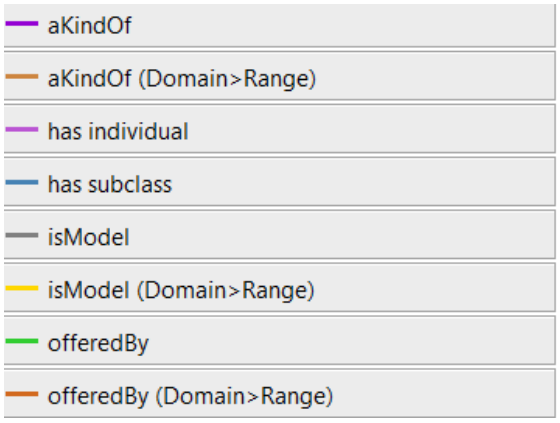


Figure 4.6: Cloud Service Categorization Ontology: Object properties legend

4.1.4 Provider Specific Ontology

Every provide has its specific ontology, named as provider. So for example for Amazon provider the ontology it's named *Amazon.owl*. This ontology is illustrated in Fig. 4.7, 4.8 and contains the following classes:

- *AmazonService*: contains all the services offered by Amazon, such as Amazon EC2, Amazon ECS, ...

- *Functionality*: this is a superclass that contains all the specific functionalities extracted for services, methods and parameters with RAKE from natural language descriptions (as already said in previous chapter). So under this class there are three subclasses: *MethodFunctionality*, *ParameterFunctionality* and *ServiceFunctionality*;

- *Method*: this is a superclass that contains, for every service in AmazonService class, a class named *"NameService"+Method* where are contained all the methods offered by that service.

- *MethodCategory*: in this class we can find the same agnostic categories that exist as individuals in AgnosticMethod class of Agnostic Ontology;

- *Parameter*: this is a superclass that contains, for every service in AmazonService class, a class named *"NameService"+Parameter* where are contained all the parameters used for the service methods;

- *ParameterCategory*: in this class we can find the same agnostic categories that exist as individuals in AgnosticParameter class of Agnostic Ontology.

the following object properties:

- *allowsTo*: links a method to its specific functionality;

- *isEquivalentTo*: links a method to its agnostic category;

- *hasFunctionality*: links a service to its specific functionality;

- *isFor*: links a parameter to its agnostic category;

- *relatedTo*: links a parameter to its specific functionality;

- *hasInputParameter*: links a method to its input parameter;

- *hasOutputParameter*: links a method to its output parameter;

- *hasMethod*: links a service to its method.

and the following datatype properties:

- *sameIndividual*: this is the same dataproperty already defined also for the Cloud Service Categorization Ontology. Indeed this property is needed to create relations between the three ontologies, linking again same individuals present in different ontologies. In particular it is needed for link: individuals under MethodCategory and ParameterCategory classes of this ontology respectively to the individuals under AgnosticMethod and AgnosticParameter classes of Agnostic Ontology; individuals of AmazonService class of this ontology to individuals of CloudService class of Cloud Service Categorization Ontology;

- *similarTo*: this is the dataproperty that helps on which is based the updating system. This property links the specific functionalities of different Provider Ontologies that are similar to each other, thanks to the similarity meter implemented (that will be described in next sections). So when the updating system finds a specific functionality referred at least from another specific functionality, thanks to this property, then this specific functionality becomes an agnostic concept (for the definition of Agnostic = vendor independent) in the Agnostic Ontology and its removed from Provider Ontologies. Also the updating system will take care of update the classifiers, in particular the Multinomial Naive Bayes Classifier.

4.1.5 Why three different ontologies?

It was thought to use three different ontologies because the domains they represent are quite separate: indeed a distinction between agnostic and provider specific concepts seems to be legit. But it was needed also an ontology that could represent information about cloud services, independently from their provider, in order

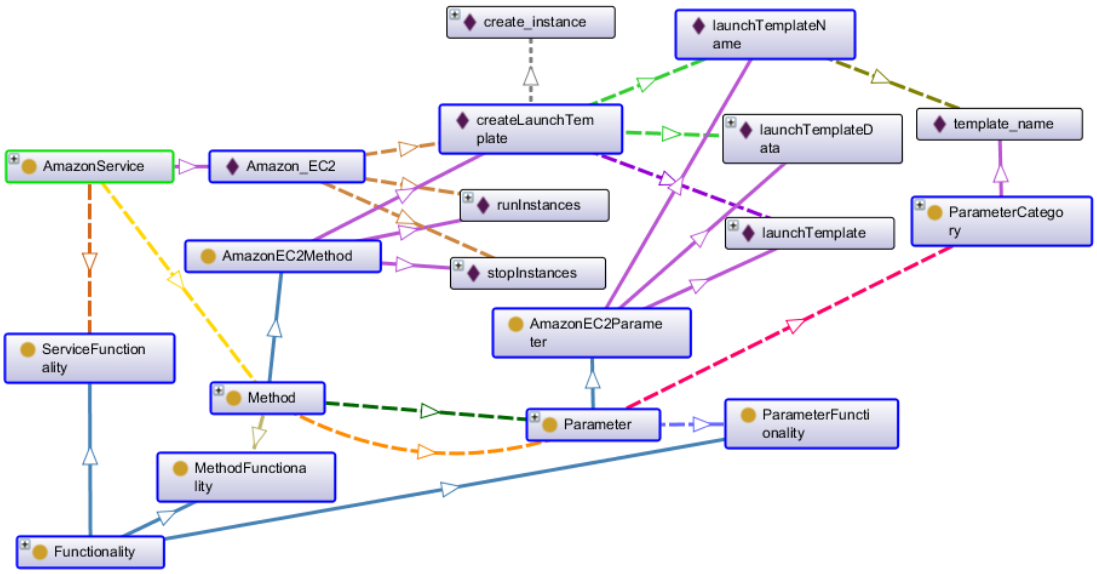


Figure 4.7: Amazon Ontology, with an example of a service, with its methods and parameters

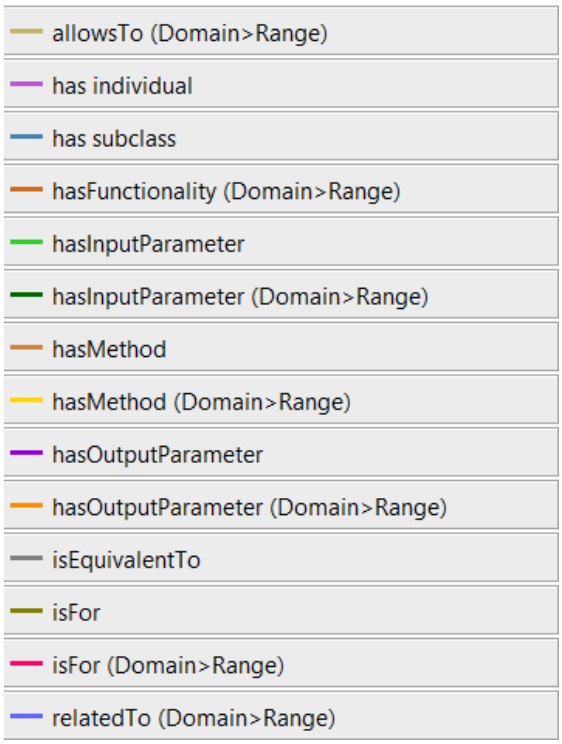


Figure 4.8: Amazon Ontology: object properties legend

to identify cloud services for consumer needs: that's why it was thought to add Cloud Service Categorization Ontology. Furthermore not all the information they

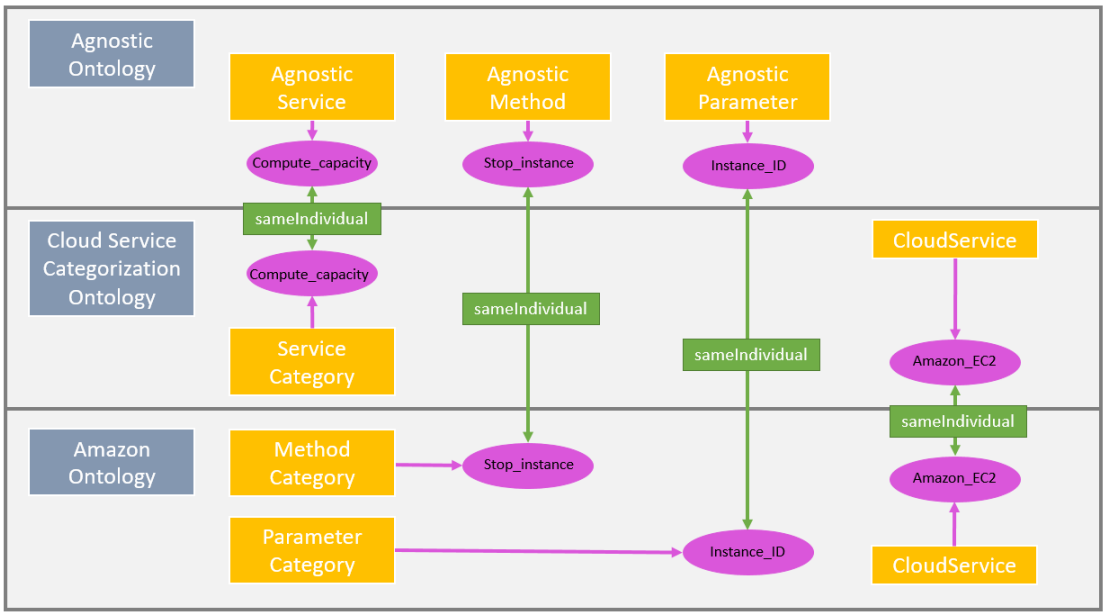


Figure 4.9: Relations between ontologies

contain is always useful. Let's suppose our user only wants to know the methods and parameters of a service offered by Amazon: he does not need all the information contained in the other ontologies, but only those contained in the Amazon Ontology. Obviously in these ontologies some information are replied in order not to require imports, but at the same time to link the ontologies. In Fig. 4.9 you can see these replications and links.

4.2 Classifiers' implementation

In this section it is described how it was possible to implement the text classifiers needed in order to analyze natural language descriptions of services, methods and parameters and categorize them as agnostic concepts. As already mentioned three different approaches have been used in order also to make a comparison between them, were it was possible to do it. In particular for this work three different text classifiers have been implemented and used:

- **Multi Layer Perceptron text classifier**: this is a text classifier that uses

multiple layers neural networks (described in section 2.5.9);

- **Convolution Neural Network text classifier**: this is a text classifier that uses convolutional neural networks (described in section 2.5.13);
- **Multinomial Naive Bayes text classifier**: this is a text classifier that uses the Multinomial variant of Naive Bayes classifier (described in sections 2.5.6 and 2.5.7).

The main difference between an approach with neural networks and one with Naive Bayes classifier is in the training dataset dimension: indeed in order to well train a neural network you need a huge training dataset if you want to have good results. Instead, a Naive Bayes classifier produce appreciable results also with reduced training dataset.

So, for this work it have been used MLP and CNN text classifier to analyze service natural language descriptions in order to classify cloud services in OpenCrowd categories. It was possible to use neural networks because to build the training dataset have been used the information already present on official site [48]. Indeed for every category there is a brief description of this category and some cloud services, with their relative natural language descriptions, as you can see in Fig. 4.10. Obviously cloud service's names have been eliminated, as every connection to the specific service or provider: indeed from these descriptions have been taken only some keywords that for us turned out to be relevant for that category. For this extraction has been used RAKE algorithm, in order to make more fast the training dataset built, but, as already said, most of the keywords extracted with RAKE are actually not relevant and also if a score is associated to every keyword, it wasn't possible to identify a clear limit under which discard keywords, because sometimes keywords with high scores turned out to be not relevant. So this building process is not automatable, but actually RAKE helps a lot to at least identify all keywords, only the choices need to be done manually. A

convolutional neural network is a type of multi-layer perceptron. You can think of a convolutional neural network as a multi-layer perceptron with: 1) many of the weights forced to be the same (think of a convolution running over the entire image: the weights in the convolution are forced to be the same); 2) many of the weights are forced to be zero. Every so often, a sub-sampling happens so that the layers get progressively smaller and smaller. Because of this design, convolutional neural networks have many fewer parameters than a multi-layer perceptron design. The smaller number of parameters that mirrors properties that we see in the real world (e.g. those found in images) mean that it can learn structures that would not be possible with a fully connected multi-layer perceptron. Indeed also for text classification you can see better results of CNN over MLP classifier.

Instead, in order to classify services, methods and parameters as agnostic concepts in this work have been used MLB and CNN text classifier. Indeed, despite of training dataset dimension (really small: it is formed by only few keywords for every agnostic concept made by hand), it was tried to use anyway CNN, but with poor results. Naive Bayes and neural networks have different performance characteristics with respect to the amount of training data they receive. The Naive Bayes classifier has been shown to perform surprisingly well with very small amounts of training data that most other classifiers, and especially neural networks, would find significantly insufficient [2]. As a result, if you find yourself with a small amount of training data Naive Bayes would be a good bet. However, the two classifiers also behave differently on the other end of the spectrum, when provided with large amounts of training data. As it is fed increasing quantities of training data, the performance of the Naive Bayes classifier plateaus above a certain threshold. Its simplicity prevents it from benefiting incrementally from training data past a certain point. In contrast, neural networks' complexity makes them capable of benefiting from huge amounts of data. While they may train annoyingly slowly on large data sets, their performance nonetheless has been shown to improve as

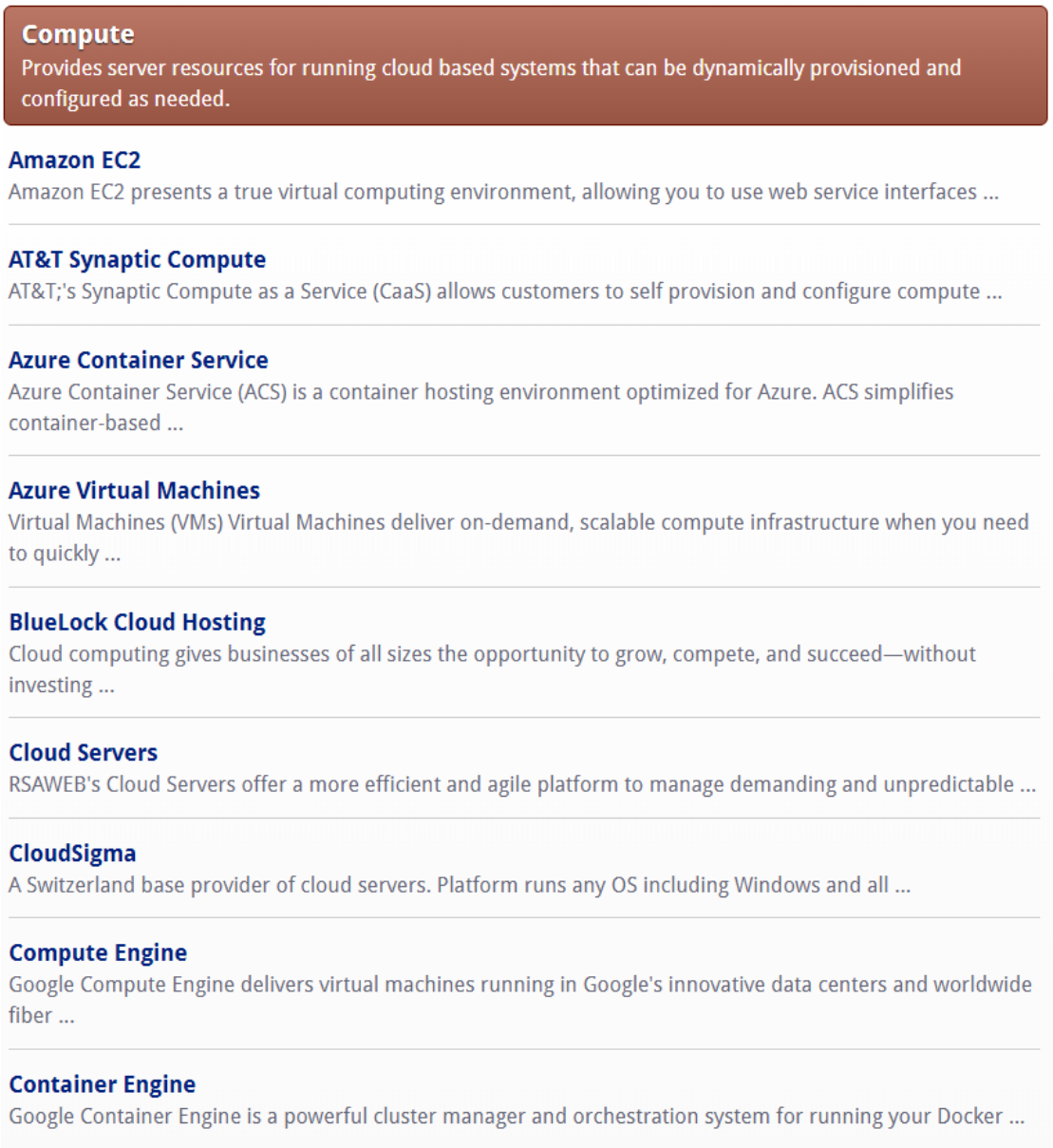


Figure 4.10: Cloud services classified as Compute from OpenCrowd Site

large amounts of training data are accumulated. Furthermore, the two models' complexities impact their tendencies to overfit. Naive Bayes' simplicity prevents it from fitting its training data too closely. In contrast, due to their complexity Neural Networks can very easily over fit training data, especially when provided with large data sets. This has motivated the investment of a good deal of research effort into finding ways to prevent Neural Networks from overfitting. Techniques

such as Dropout resulted from such efforts and are arguably necessary for making neural networks generalize well.

4.2.1 Training dataset construction

In order to build training datasets to use for python classifier implementations it has been used a list of dictionaries. You can see an example in Fig. 4.11. For every category there is a directory, named as the specific category and in each of these directories there is a file *servizi.txt*, that contains the sentences or keywords used to train classifiers: you can see an exampple for compute category in Fig. 4.12

```
with open('C:/Users/tony_/OneDrive/Desktop/train_dataset/compute/servizi.txt', 'r') as myfile1:
    data1=myfile1.readlines()
    data1=[x.strip() for x in data1]
    for y in data1:
        training_data.append({"class":"Compute","sentence":y})
with open('C:/Users/tony_/OneDrive/Desktop/train_dataset/backup&recovery/servizi.txt', 'r') as myfile2:
    data2 = myfile2.readlines()
    data2 = [x.strip() for x in data2]
    for y in data2:
        training_data.append({"class": "Backup & Recovery", "sentence": y})
with open('C:/Users/tony_/OneDrive/Desktop/train_dataset/cloud broker/servizi.txt', 'r') as myfile3:
    data3 = myfile3.readlines()
    data3 = [x.strip() for x in data3]
    for y in data3:
        training_data.append({"class":"Cloud Broker", "sentence":y})
```

Figure 4.11: Training dataset in python: it reads sentences from a file *servizi.txt* under a directory named as the category at which are associated these sentences

```
Provides server resources for running cloud based systems that can be
dynamically provisioned and configured as needed.
Presents a true virtual computing environment, allowing you to use web
service interfaces to launch instances with a variety of operating systems,
load them with your custom application environment, manage your network's
access permissions, and run your image using as many or few systems as you
desire. Preconfigured or custom machine images (AMI) can be used on servers.
allows customers to self provision and configure compute resources.
```

Figure 4.12: Content example of a *servizi.txt* file: this is for compute category

4.2.2 MLB Classifier

In order to build a Muiltinomial Naive Bayes classifier it has been used a python library **Sklearn**, that allows to resolve machine learning problems. In general, a learning problem considers a set of n samples of data and then tries to predict properties of unknown data. If each sample is more than a single number and, for instance, a multi-dimensional entry (aka multivariate data), it is said to have several attributes or features. We can separate learning problems in a few large categories:

- supervised learning, in which the data comes with additional attributes that we want to predict. This problem can be either:

  classification: samples belong to two or more classes and we want to learn from already labeled data how to predict the class of unlabeled data. An example of classification problem would be the handwritten digit recognition example, in which the aim is to assign each input vector to one of a finite number of discrete categories. Another way to think of classification is as a discrete (as opposed to continuous) form of supervised learning where one has a limited number of categories and for each of the n samples provided, one is to try to label them with the correct category or class.

  regression: if the desired output consists of one or more continuous variables, then the task is called regression. An example of a regression problem would be the prediction of the length of a salmon as a function of its age and weight.

- unsupervised learning, in which the training data consists of a set of input vectors x without any corresponding target values. The goal in such problems may be to discover groups of similar examples within the data, where it is called clustering, or to determine the distribution of data within the input space, known as density estimation, or to project the data from a

high-dimensional space down to two or three dimensions for the purpose of visualization

Let's go back to MLB classifier with sklearn: first of all the training dataset, that is formed by text associated to a class, needs to be transformed in vectors that can be used to feed the classifier. In sklearn text preprocessing, tokenizing and filtering of stopwords are included in a high level component that is able to build a dictionary of features and transform documents to feature vectors: this is **CountVectorizer** and it supports counts of N-grams of words or consecutive characters. Once fitted, the vectorizer has built a dictionary of feature indices: the index value of a word in the vocabulary is linked to its frequency in the whole training corpus. Occurrence count is a good start but there is an issue: longer documents will have higher average count values than shorter documents, even though they might talk about the same topics. To avoid these potential discrepancies it suffices to divide the number of occurrences of each word in a document by the total number of words in the document: these new features are called *tf* for Term Frequencies. Another refinement on top of tf is to downscale weights for words that occur in many documents in the corpus and are therefore less informative than those that occur only in a smaller portion of the corpus. This downscaling is called *tf-idf* (described in section 2.5.1) for "Term Frequency times Inverse Document Frequency". Both tf and tf–idf can be computed with **TfidfTransformer**. Now that we have our features, we can train a classifier to try to predict the category. Sklearn offers a function that implements a Multinomial Naive Bayes classifier: **MultinomialNB**. Obviously, to try to predict the outcome on a new document we need to extract the features using almost the same feature extracting chain as before. You can see the full code in Fig. 4.13.

```
sentences=[]
categories=[]
for p in training_data:
    sentences.append(p['sentence'])
    categories.append(p['class'])
count_vect = CountVectorizer()
X_train_counts = count_vect.fit_transform(sentences)
print(X_train_counts.shape)
tf_transformer = TfidfTransformer(use_idf=False).fit(X_train_counts)
X_train_tf = tf_transformer.transform(X_train_counts)
print(X_train_tf.shape)
tfidf_transformer = TfidfTransformer()
X_train_tfidf = tfidf_transformer.fit_transform(X_train_counts)
print(X_train_tfidf.shape)
clf = MultinomialNB().fit(X_train_tfidf, categories)
docs_new = [description]
X_new_counts = count_vect.transform(docs_new)
X_new_tfidf = tfidf_transformer.transform(X_new_counts)
predicted = clf.predict(X_new_tfidf)
print(predicted)
return predicted
```

Figure 4.13: MLB classifier in python

4.2.3 MLP Classifier

To implement an MLP Classifier it have been used two python libraries: **Numpy** and **Nltk**.

NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of highlevel mathematical functions to operate on these arrays. It contains among other things:

- a powerful N-dimensional array object;
- sophisticated (broadcasting) functions;
- tools for integrating C/C++ and Fortran code;
- useful linear algebra, Fourier transform, and random number capabilities.

Besides its obvious scientific uses, NumPy can also be used as an efficient multidimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

NLTK is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial-strength NLP libraries, and an active discussion forum. NLTK has been called "a wonderful tool for teaching, and working in, computational linguistics using Python," and "an amazing library to play with natural language." *Natural Language Processing with Python* provides a practical introduction to programming for language processing. Written by the creators of NLTK, it guides the reader through the fundamentals of writing Python programs, working with corpora, categorizing text, analyzing linguistic structure, and more.

Going back to MLP classifier it uses 2 layers of neurons (1 hidden layer) and a "bag of words" approach to organizing our training data. Text classification comes in 3 flavors: pattern matching, algorithms, neural nets. While the algorithmic approach using Multinomial Naive Bayes is surprisingly effective, it suffers from 3 fundamental flaws:

- the algorithm produces a score rather than a probability. We want a probability to ignore predictions below some threshold. This is akin to a 'squelch' dial on a VHF radio.
- the algorithm 'learns' from examples of what is in a class, but not what isn't. This learning of patterns of what does not belong to a class is often very important.
- classes with disproportionately large training sets can create distorted clas-

sification scores, forcing the algorithm to adjust scores relative to class size. This is not ideal.

As with its 'Naive' counterpart, this classifier isn't attempting to understand the meaning of a sentence, it's trying to classify it. In fact so called "AI chat-bots" do not understand language, but that's another story.

First of all we need to import natural language toolkit. We need a way to reliably tokenize sentences into words and a way to stem words, Fig. 4.14.

```
1  # use natural language toolkit
2  import nltk
3  from nltk.stem.lancaster import LancasterStemmer
4  import os
5  import json
6  import datetime
7  stemmer = LancasterStemmer()
```

Figure 4.14: Imports

Then we need to provide a training dataset, as shown in section 4.2.1 and organize data structures for documents, classes and words, Fig. 4.15

Our training data is transformed into "bag of words" for each sentence, Fig. 4.16. Each training sentence is reduced to an array of 0's and 1's against the array of unique words in the corpus.

Next we have our core functions for our 2-layer neural network, Fig. 4.23. It is used numpy because to make matrix multiplication fast. It is used a sigmoid function to normalize values and its derivative to measure the error rate. Iterating and adjusting until our error rate is acceptably low. Also below, in Fig. 4.17 it is implemented bag-of-words function, transforming an input sentence into an array of 0's and 1's. This matches precisely with our transform for training data, always crucial to get this right.

And now let's build code our neural network training function to create synaptic weights, Fig 4.18, 4.19.

```
words = []
classes = []
documents = []
ignore_words = ['?']
# loop through each sentence in our training data
for pattern in training_data:
    # tokenize each word in the sentence
    w = nltk.word_tokenize(pattern['sentence'])
    # add to our words list
    words.extend(w)
    # add to documents in our corpus
    documents.append((w, pattern['class']))
    # add to our classes list
    if pattern['class'] not in classes:
        classes.append(pattern['class'])

# stem and lower each word and remove duplicates
words = [stemmer.stem(w.lower()) for w in words if w not in ignore_words]
words = list(set(words))

# remove duplicates
classes = list(set(classes))

print (len(documents), "documents")
print (len(classes), "classes", classes)
print (len(words), "unique stemmed words", words)
```

Figure 4.15: Training data structure

We are now ready to build neural network model, we will save this as a json structure to represent our synaptic weights. Should experiment with different 'alpha' (gradient descent parameter) and see how it affects the error rate. This parameter helps our error adjustment find the lowest error rate, Fig. 4.20.

This work uses 20 neurons in hidden layer. These parameters will vary depending on the dimensions and shape of training data, tune them down to 10^{-3} as a reasonable error rate, Fig 4.21.

```
training = []
output = []
# create an empty array for our output
output_empty = [0] * len(classes)

# training set, bag of words for each sentence
for doc in documents:
    # initialize our bag of words
    bag = []
    # list of tokenized words for the pattern
    pattern_words = doc[0]
    # stem each word
    pattern_words = [stemmer.stem(word.lower()) for word in pattern_words]
    # create our bag of words array
    for w in words:
        bag.append(1) if w in pattern_words else bag.append(0)

    training.append(bag)
    # output is a '0' for each tag and '1' for current tag
    output_row = list(output_empty)
    output_row[classes.index(doc[1])] = 1
    output.append(output_row)

# sample training/output
i = 0
w = documents[i][0]
```

Figure 4.16: Bag of words approach

The synapse.json file contains all of our synaptic weights, this is the model. This classify() function, in Fig. 4.22 is all that's needed for the classification once synapse weights have been calculated. The catch: if there's a change to the training data our model will need to be re-calculated. For a very large dataset this could take a non-insignificant amount of time. We can now generate the probability of a sentence belonging to one (or more) of classes. This is super fast because it's dot-product calculation.

```
import numpy as np
import time

# compute sigmoid nonlinearity
def sigmoid(x):
    output = 1/(1+np.exp(-x))
    return output

# convert output of sigmoid function to its derivative
def sigmoid_output_to_derivative(output):
    return output*(1-output)

def clean_up_sentence(sentence):
    # tokenize the pattern
    sentence_words = nltk.word_tokenize(sentence)
    # stem each word
    sentence_words = [stemmer.stem(word.lower()) for word in sentence_words]
    return sentence_words

# return bag of words array: 0 or 1 for each word in the bag that exists in the sentence
def bow(sentence, words, show_details=False):
    # tokenize the pattern
    sentence_words = clean_up_sentence(sentence)
    # bag of words
    bag = [0]*len(words)
    for s in sentence_words:
        for i,w in enumerate(words):
            if w == s:
                bag[i] = 1
                if show_details:
                    print ("found in bag: %s" % w)

    return(np.array(bag))

def think(sentence, show_details=False):
    x = bow(sentence.lower(), words, show_details)
    if show_details:
        print ("sentence:", sentence, "\n bow:", x)
    # input layer is our bag of words
    l0 = x
    # matrix multiplication of input and hidden layer
    l1 = sigmoid(np.dot(l0, synapse_0))
    # output layer
    l2 = sigmoid(np.dot(l1, synapse_1))
    return l2
```

Figure 4.17: Bag of words function

```
def train(X, y, hidden_neurons=10, alpha=1, epochs=50000, dropout=False, dropout_percent=0.5):

    print ("Training with %s neurons, alpha:%s, dropout:%s %s" % (hidden_neurons, str(alpha), dropout, dropout_percent if dropout else '') )
    print ("Input matrix: %sx%s    Output matrix: %sx%s" % (len(X),len(X[0]),1, len(classes)) )
    np.random.seed(1)

    last_mean_error = 1
    # randomly initialize our weights with mean 0
    synapse_0 = 2*np.random.random((len(X[0]), hidden_neurons)) - 1
    synapse_1 = 2*np.random.random((hidden_neurons, len(classes))) - 1

    prev_synapse_0_weight_update = np.zeros_like(synapse_0)
    prev_synapse_1_weight_update = np.zeros_like(synapse_1)

    synapse_0_direction_count = np.zeros_like(synapse_0)
    synapse_1_direction_count = np.zeros_like(synapse_1)

    for j in iter(range(epochs+1)):

        # Feed forward through layers 0, 1, and 2
        layer_0 = X
        layer_1 = sigmoid(np.dot(layer_0, synapse_0))

        if(dropout):
            layer_1 *= np.random.binomial([np.ones((len(X),hidden_neurons))],1-dropout_percent)[0] * (1.0/(1-dropout_percent))

        layer_2 = sigmoid(np.dot(layer_1, synapse_1))

        # how much did we miss the target value?
        layer_2_error = y - layer_2

        if (j% 10000) == 0 and j > 5000:
            # if this 10k iteration's error is greater than the last iteration, break out
            if np.mean(np.abs(layer_2_error)) < last_mean_error:
                print ("delta after "+str(j)+" iterations:" + str(np.mean(np.abs(layer_2_error))) )
                last_mean_error = np.mean(np.abs(layer_2_error))
            else:
                print ("break:", np.mean(np.abs(layer_2_error)), ">", last_mean_error )
                break

        # in what direction is the target value?
        # were we really sure? if so, don't change too much.
        layer_2_delta = layer_2_error * sigmoid_output_to_derivative(layer_2)

        # how much did each l1 value contribute to the l2 error (according to the weights)?
        layer_1_error = layer_2_delta.dot(synapse_1.T)
```

Figure 4.18: Training function: first part

```
        # in what direction is the target l1?
        # were we really sure? if so, don't change too much.
        layer_1_delta = layer_1_error * sigmoid_output_to_derivative(layer_1)

        synapse_1_weight_update = (layer_1.T.dot(layer_2_delta))
        synapse_0_weight_update = (layer_0.T.dot(layer_1_delta))

        if(j > 0):
            synapse_0_direction_count += np.abs(((synapse_0_weight_update > 0)+0) - ((prev_synapse_0_weight_update > 0) + 0))
            synapse_1_direction_count += np.abs(((synapse_1_weight_update > 0)+0) - ((prev_synapse_1_weight_update > 0) + 0))

        synapse_1 += alpha * synapse_1_weight_update
        synapse_0 += alpha * synapse_0_weight_update

        prev_synapse_0_weight_update = synapse_0_weight_update
        prev_synapse_1_weight_update = synapse_1_weight_update

    now = datetime.datetime.now()

    # persist synapses
    synapse = {'synapse0': synapse_0.tolist(), 'synapse1': synapse_1.tolist(),
               'datetime': now.strftime("%Y-%m-%d %H:%M"),
               'words': words,
               'classes': classes
              }
    synapse_file = "synapses.json"

    with open(synapse_file, 'w') as outfile:
        json.dump(synapse, outfile, indent=4, sort_keys=True)
    print ("saved synapses to:", synapse_file)
```

Figure 4.19: Training function: second part

synapse_0 += **alpha** * synapse_0_weight_update

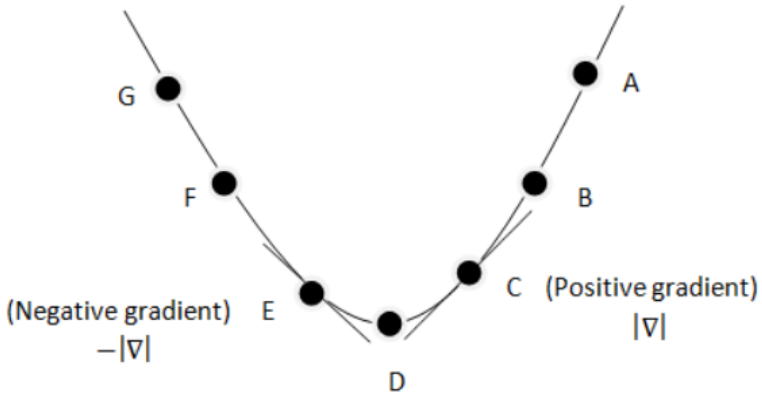


Figure 4.20: Synaptic weights

```
X = np.array(training)
y = np.array(output)

start_time = time.time()

train(X, y, hidden_neurons=20, alpha=0.1, epochs=100000, dropout=False, dropout_percent=0.2)

elapsed_time = time.time() - start_time
print ("processing time:", elapsed_time, "seconds")
```

Figure 4.21: Error rate estimation

```
# probability threshold
ERROR_THRESHOLD = 0.2
# load our calculated synapse values
synapse_file = 'synapses.json'
with open(synapse_file) as data_file:
    synapse = json.load(data_file)
    synapse_0 = np.asarray(synapse['synapse0'])
    synapse_1 = np.asarray(synapse['synapse1'])

def classify(sentence, show_details=False):
    results = think(sentence, show_details)

    results = [[i,r] for i,r in enumerate(results) if r>ERROR_THRESHOLD ]
    results.sort(key=lambda x: x[1], reverse=True)
    return_results =[[classes[r[0]],r[1]] for r in results]
    print ("%s \n classification: %s" % (sentence, return_results))
    return return_results
```

Figure 4.22: Classify function

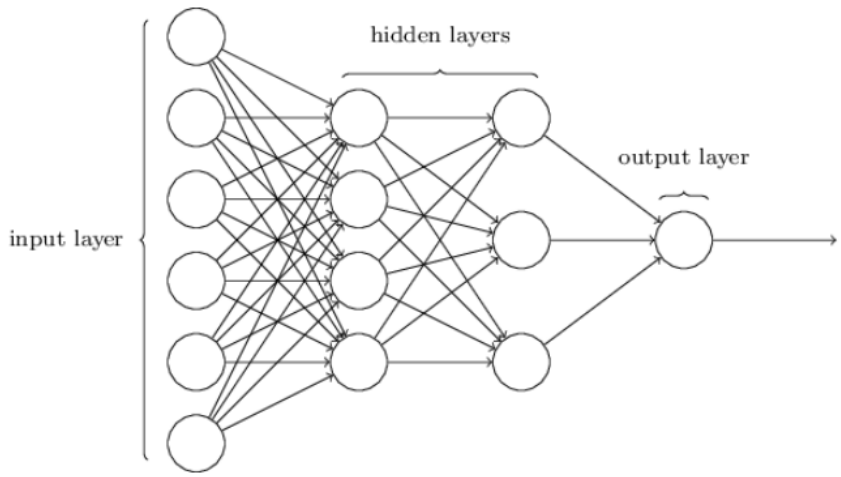


Figure 4.23: Neural network scheme

4.2.4 CNN Classifier

To create a CNN classifier Numpy and TensorFlow have been used.

TensorFlow

TensorFlow is an open source software library for numerical computation using dataflow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. TensorFlow was originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well. TensorFlow uses a dataflow graph to represent your computation in terms of the dependencies between individual operations. This leads to a low-level programming model in which you first define the dataflow graph, then create a TensorFlow session to run parts of the graph across a set of local and remote devices. Building a computational graph can be considered as the main ingredient of TensorFlow. Dataflow is a common programming model for parallel computing. In a dataflow graph (Figure 4.24), the nodes represent units of computation, and the edges represent the data consumed or produced by a computation. For example, in a TensorFlow graph, the tf.matmul operation would correspond to a single node with two incoming edges (the matrices to be multiplied) and one outgoing edge (the result of the multiplication).

A **tensor** is a generalization of vectors and matrices to potentially higher dimensions. Internally, TensorFlow represents tensors as n-dimensional arrays of base datatypes. When writing a TensorFlow program, the main object you

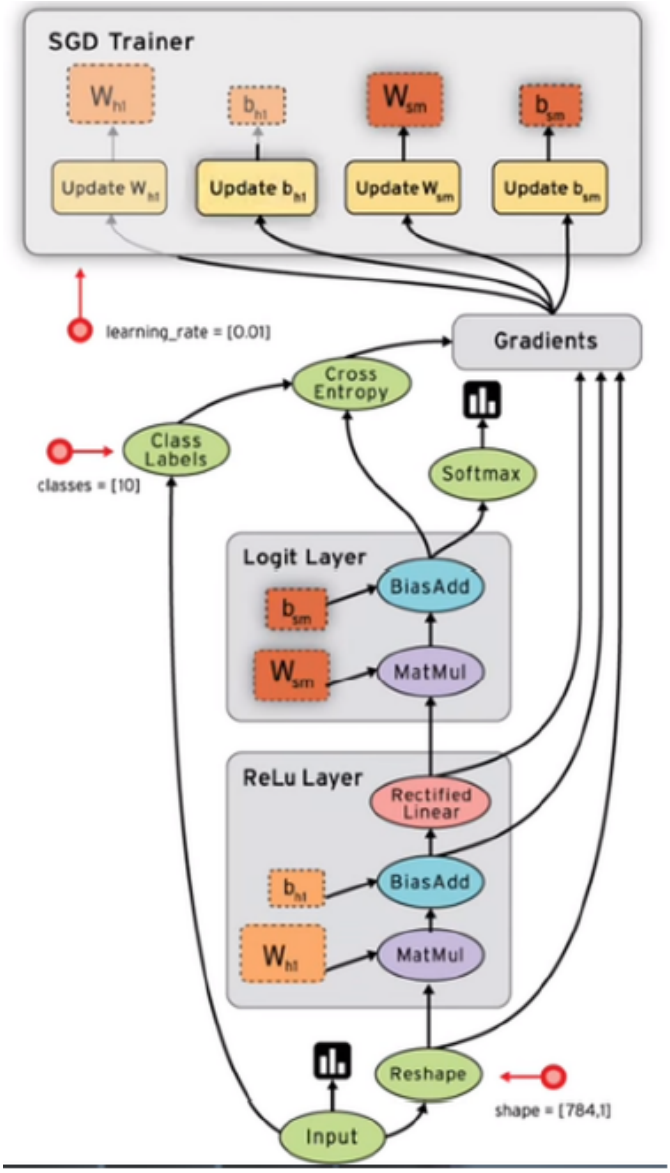


Figure 4.24: Dataflow example of a neural network

manipulate and pass around is the tf.Tensor. A tf.Tensor object represents a partially defined computation that will eventually produce a value. TensorFlow programs work by first building a graph of tf.Tensor objects, detailing how each tensor is computed based on the other available tensors and then by running parts of this graph to achieve the desired results. A tf.Tensor has the following properties: a data type and a shape. Each element in the Tensor has the same data type, and the data type is always known. The shape (that is, the number of dimensions it has and the size of each dimension) might be only partially known. Most operations produce tensors of fully-known shapes if the shapes of their inputs are also fully known, but in some cases it's only possible to find the shape of a tensor at graph execution time. Some types of tensors are special:

- tf.Variable
- tf.Constant
- tf.Placeholder
- tf.SparseTensor

With the exception of tf.Variable, the value of a tensor is immutable, which means that in the context of a single execution tensors only have a single value. However, evaluating the same tensor twice can return different values; for example that tensor can be the result of reading data from disk, or generating a random number.

A TensorFlow variable is the best way to represent shared, persistent state manipulated by your program. Variables are manipulated via the tf.Variable class. A tf.Variable represents a tensor whose value can be changed by running ops on it. Unlike tf.Tensor objects, a tf.Variable exists outside the context of a single session.run call. Internally, a tf.Variable stores a persistent tensor. Specific ops allow you to read and modify the values of this tensor. These modifications are visible across multiple tf.Sessions, so multiple workers can see the same values

for a tf.Variable. Using tf.Variable we have to provide an initial value when you declare it.

Using tf.placeholder we don't have to provide an initial value and you can specify it at run time with the feed dict argument (a feed temporarily replaces the output of an operation with a tensor value) inside a Session.run (launch the graph with a Session and use the Session.run() method to execute operations).

CNN for NLP

Let's go back to the CNN text classifier. In this work will be implemented a model similar to [37]. The model presented in the paper achieves good classification performance across a range of text classification tasks (like Sentiment Analysis) and has since become a standard baseline for new text classification architectures. But fist we need to understand how to use Convolutional Neural Networks for NLP. CNNs are basically just several layers of convolutions with nonlinear activation functions like ReLU or tanh applied to the results. In a traditional feedforward neural network we connect each input neuron to each output neuron in the next layer. That's also called a fully connected layer, or affine layer. In CNNs we don't do that. Instead, we use convolutions over the input layer to compute the output. This results in local connections, where each region of the input is connected to a neuron in the output. Each layer applies different filters, typically hundreds or thousands like the ones showed above, and combines their results. There's also something something called pooling (subsampling) layers, but I'll get into that later. During the training phase, a CNN automatically learns the values of its filters based on the task you want to perform. For example, in Image Classification a CNN may learn to detect edges from raw pixels in the first layer, then use the edges to detect simple shapes in the second layer, and then use these shapes to deter higher-level features, such as facial shapes in higher layers. The last layer is then a classifier that uses these high-level features. There are two

aspects of this computation worth paying attention to: **Location Invariance** and **Compositionality**. Let's say you want to classify whether or not there's an elephant in an image. Because you are sliding your filters over the whole image you don't really care where the elephant occurs. In practice, pooling also gives you invariance to translation, rotation and scaling, but more on that later. The second key aspect is (local) compositionality. Each filter composes a local patch of lower-level features into higher-level representation. That's why CNNs are so powerful in Computer Vision. It makes intuitive sense that you build edges from pixels, shapes from edges, and more complex objects from shapes.

Instead of image pixels, the input to most NLP tasks are sentences or documents represented as a matrix. Each row of the matrix corresponds to one token, typically a word, but it could be a character. That is, each row is vector that represents a word. Typically, these vectors are word embeddings (low-dimensional representations) like word2vec or GloVe, but they could also be one-hot vectors that index the word into a vocabulary. For a 10 word sentence using a 100-dimensional embedding we would have a 10×100 matrix as our input. That's our "image". In vision, our filters slide over local patches of an image, but in NLP we typically use filters that slide over full rows of the matrix (words). Thus, the "width" of our filters is usually the same as the width of the input matrix. The height, or region size, may vary, but sliding windows over 2-5 words at a time is typical. Putting all the above together, a Convolutional Neural Network for NLP may look like in Fig. 4.25.

What about the nice intuitions we had for Computer Vision? Location Invariance and local Compositionality made intuitive sense for images, but not so much for NLP. You probably do care a lot where in the sentence a word appears. Pixels close to each other are likely to be semantically related (part of the same object), but the same isn't always true for words. In many languages, parts of phrases could be separated by several other words. The compositional aspect isn't

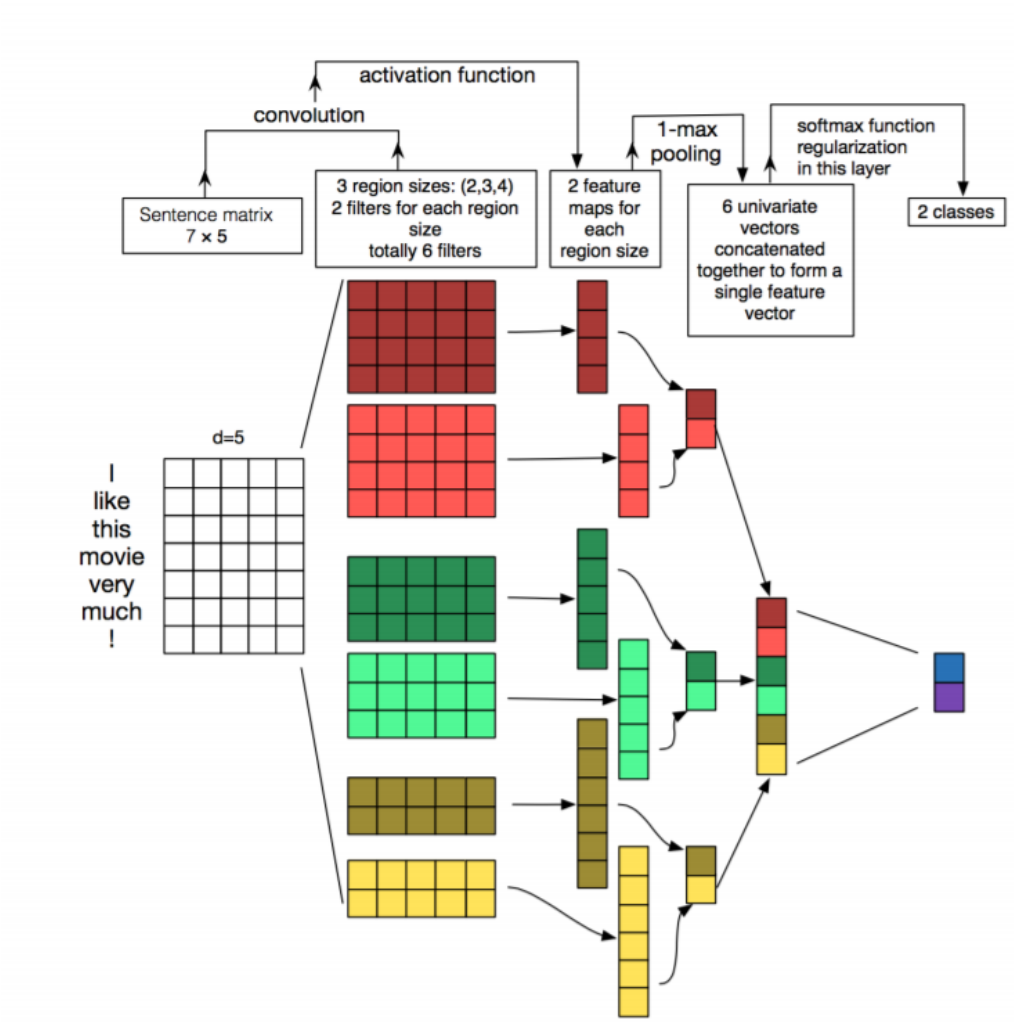


Figure 4.25: CNN for NLP. Here we depict three filter region sizes: 2, 3 and 4, each of which has 2 filters. Every filter performs convolution on the sentence matrix and generates (variable-length) feature maps. Then 1-max pooling is performed over each map, i.e., the largest number from each feature map is recorded. Thus a univariate feature vector is generated from all six maps, and these 6 features are concatenated to form a feature vector for the penultimate layer. The final softmax layer then receives this feature vector as input and uses it to classify the sentence; here we assume binary classification and hence depict two possible output states.

obvious either. Clearly, words compose in some ways, like an adjective modifying a noun, but how exactly this works what higher level representations actually "mean" isn't as obvious as in the Computer Vision case.

Given all this, it seems like CNNs wouldn't be a good fit for NLP tasks. Fortunately, this doesn't mean that CNNs don't work. All models are wrong, but some are useful. It turns out that CNNs applied to NLP problems perform quite well. The simple Bag of Words model is an obvious oversimplification with incorrect assumptions, but has nonetheless been the standard approach for years and lead to pretty good results. A big argument for CNNs is that they are fast. Very fast. Convolutions are a central part of computer graphics and implemented on a hardware level on GPUs. Compared to something like n-grams, CNNs are also efficient in terms of representation. With a large vocabulary, computing anything more than 3-grams can quickly become expensive. Even Google doesn't provide anything beyond 5-grams. Convolutional Filters learn good representations automatically, without needing to represent the whole vocabulary. It's completely reasonable to have filters of size larger than 5. I like to think that many of the learned filters in the first layer are capturing features quite similar (but not limited) to n-grams, but represent them in a more compact way.

CNN Hyperparameters When I explained convolutions above I neglected a little detail of how we apply the filter. Applying a 3×3 filter at the center of the matrix works fine, but what about the edges? How would you apply the filter to the first element of a matrix that doesn't have any neighboring elements to the top and left? You can use zero-padding. All elements that would fall outside of the matrix are taken to be zero. By doing this you can apply the filter to every element of your input matrix, and get a larger or equally sized output. Adding zero-padding is also called **wide convolution**, and not using zero-padding would be a **narrow convolution**. An example in 1D looks like in Fig. 4.26.

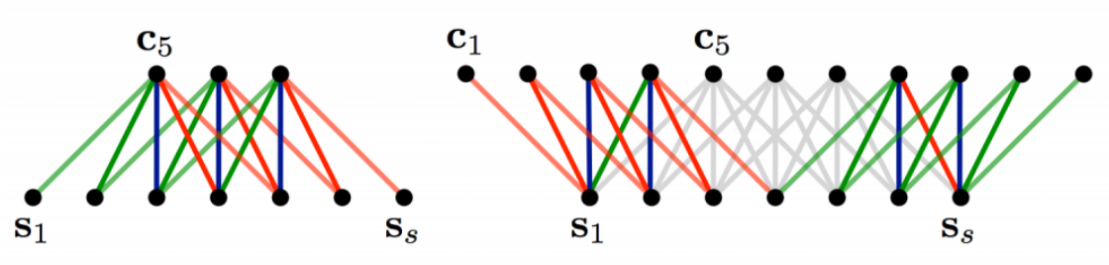


Figure 4.26: Narrow vs. Wide Convolution. Filter size 5, input size 7

Another hyperparameter for your convolutions is the **stride size**, defining by how much you want to shift your filter at each step. In all the examples above the stride size was 1, and consecutive applications of the filter overlapped. A larger stride size leads to fewer applications of the filter and a smaller output size.

A key aspect of Convolutional Neural Networks are **pooling layers**, typically applied after the convolutional layers. Pooling layers subsample their input. The most common way to do pooling it to apply a max operation to the result of each filter. You don't necessarily need to pool over the complete matrix, you could also pool over a window. One property of pooling is that it provides a fixed size output matrix, which typically is required for classification. For example, if you have 1,000 filters and you apply max pooling to each, you will get a 1000-dimensional output, regardless of the size of your filters, or the size of your input. This allows you to use variable size sentences, and variable size filters, but always get the same output dimensions to feed into a classifier. Pooling also reduces the output dimensionality but (hopefully) keeps the most salient information. You can think of each filter as detecting a specific feature, such as detecting if the sentence contains a negation like "not amazing" for example. If this phrase occurs somewhere in the sentence, the result of applying the filter to that region will yield a large value, but a small value in other regions. By performing the max operation you are keeping information about whether or not the feature appeared in the sentence, but you are losing information about where exactly it appeared. But isn't this information

about locality really useful? Yes, it is and it's a bit similar to what a bag of n-grams model is doing. You are losing global information about locality (where in a sentence something happens), but you are keeping local information captured by your filters, like "not amazing" being very different from "amazing not".

The last concept we need to understand are **channels**. Channels are different "views" of your input data. For example, in image recognition you typically have RGB (red, green, blue) channels. You can apply convolutions across channels, either with different or equal weights. In NLP you could imagine having various channels as well: You could have a separate channels for different word embeddings (word2vec and GloVe for example), or you could have a channel for the same sentence represented in different languages, or phrased in different ways.

Evaluates a CNN architecture on various classification datasets [37], mostly comprised of Sentiment Analysis and Topic Categorization tasks. The CNN architecture achieves very good performance across datasets, and new state-of-the-art on a few. Surprisingly, the network used in this paper is quite simple 4.27, and that's what makes it powerful.The input layer is a sentence comprised of concatenated word2vec word embeddings. That's followed by a convolutional layer with multiple filters, then a max-pooling layer, and finally a softmax classifier. The paper also experiments with two different channels in the form of static and dynamic word embeddings, where one channel is adjusted during training and the other isn't.

CNN text classifier implementation

In this work will be implemented a model similar to the one described in previous section [37]. First of all we need to preprocess the training dataset, it can be done with the following steps:

- Load sentences from the raw data files.

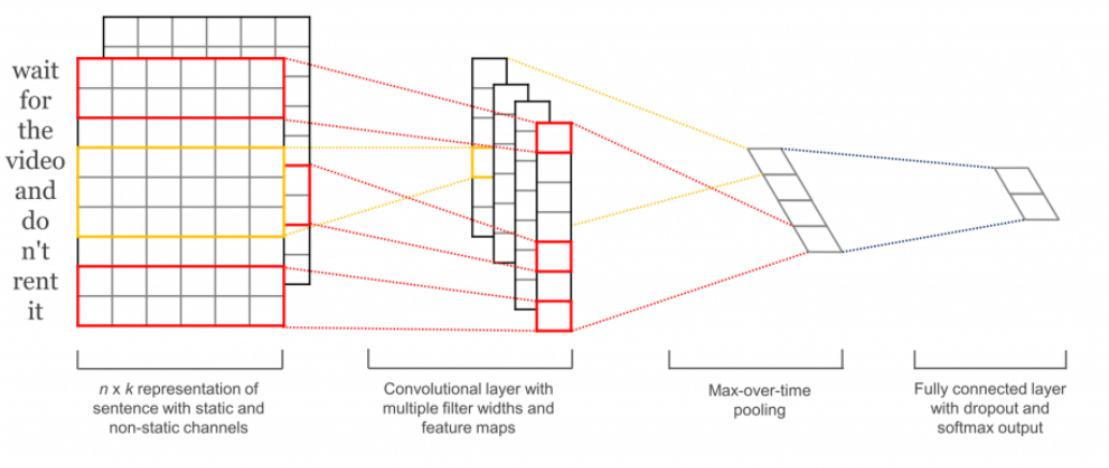


Figure 4.27: Convolutional Neural Networks for Sentence Classification

- Clean the text data using the same code as the original paper.
- Pad each sentence to the maximum sentence length. Padding sentences to the same length is useful because it allows us to efficiently batch our data since each example in a batch must be of the same length.
- Build a vocabulary index and map each word to an integer between 0 and the vocabulary size. Each sentence becomes a vector of integers.

This implementation that I'm going to describe simplify the model from the original paper a little:

- We will not used pre-trained word2vec vectors for our word embeddings. Instead, we learn embeddings from scratch.
- We will not enforce L2 norm constraints on the weight vectors because they have little effect on the end result.
- The original paper experiments with two input data channels – static and non-static word vectors. We use only one channel.

To allow various hyperparameter configurations we put our code into a TextCNN class, generating the model graph in the init function (Fig. 4.28).

```python
import tensorflow as tf
import numpy as np

class TextCNN(object):
    """
    A CNN for text classification.
    Uses an embedding layer, followed by a convolutional, max-pooling and softmax layer.
    """
    def __init__(
      self, sequence_length, num_classes, vocab_size,
      embedding_size, filter_sizes, num_filters):
        # Implementation...
```

Figure 4.28: TextCNN class

To instantiate the class we then pass the following arguments:

- sequence length – The length of our sentences. Remember that we padded all our sentences to have the same length.
- num classes – Number of classes in the output layer.
- vocab size – The size of our vocabulary. This is needed to define the size of our embedding layer, which will have shape (vocabulary size, embedding size).
- embedding size – The dimensionality of our embeddings.
- filter sizes – The number of words we want our convolutional filters to cover. We will have num filters for each size specified here. For example, (3, 4, 5) means that we will have filters that slide over 3, 4 and 5 words respectively, for a total of 3 * num filters filters.
- num filters – The number of filters per filter size (see above).

We start by defining the input data that we pass to our network, Fig. 4.29.

tf.placeholder creates a placeholder variable that we feed to the network when we execute it at train or test time. The second argument is the shape of the input tensor. None means that the length of that dimension could be anything. In our case, the first dimension is the batch size, and using None allows the network

```
# Placeholders for input, output and dropout
self.input_x = tf.placeholder(tf.int32, [None, sequence_length], name="input_x")
self.input_y = tf.placeholder(tf.float32, [None, num_classes], name="input_y")
self.dropout_keep_prob = tf.placeholder(tf.float32, name="dropout_keep_prob")
```

Figure 4.29: Input placeholders

to handle arbitrarily sized batches. The probability of keeping a neuron in the dropout layer is also an input to the network because we enable dropout only during training. We disable it when evaluating the model (more on that later).

The first layer we define is the embedding layer, which maps vocabulary word indices into low-dimensional vector representations. It's essentially a lookup table that we learn from data, Fig. 4.30.

```
with tf.device('/cpu:0'), tf.name_scope("embedding"):
    W = tf.Variable(
        tf.random_uniform([vocab_size, embedding_size], -1.0, 1.0),
        name="W")
    self.embedded_chars = tf.nn.embedding_lookup(W, self.input_x)
    self.embedded_chars_expanded = tf.expand_dims(self.embedded_chars, -1)
```

Figure 4.30: Embedding Layer

We're using a couple of new features here so let's go over them:

- tf.device("/cpu:0") forces an operation to be executed on the CPU. By default TensorFlow will try to put the operation on the GPU if one is available, but the embedding implementation doesn't currently have GPU support and throws an error if placed on the GPU.

- tf.name scope creates a new Name Scope with the name "embedding". The scope adds all operations into a top-level node called "embedding" so that you get a nice hierarchy when visualizing your network in TensorBoard.

W is our embedding matrix that we learn during training. We initialize it using a random uniform distribution. tf.nn.embedding lookup creates the actual embed-

ding operation. The result of the embedding operation is a 3-dimensional tensor of shape (None, sequence length, embedding size). TensorFlow's convolutional conv2d operation expects a 4-dimensional tensor with dimensions corresponding to batch, width, height and channel. The result of our embedding doesn't contain the channel dimension, so we add it manually, leaving us with a layer of shape (None, sequence length, embedding size, 1).

Now we're ready to build our convolutional layers followed by max-pooling. Remember that we use filters of different sizes. Because each convolution produces tensors of different shapes we need to iterate through them, create a layer for each of them, and then merge the results into one big feature vector, Fig. 4.31.

```
pooled_outputs = []
for i, filter_size in enumerate(filter_sizes):
    with tf.name_scope("conv-maxpool-%s" % filter_size):
        # Convolution Layer
        filter_shape = [filter_size, embedding_size, 1, num_filters]
        W = tf.Variable(tf.truncated_normal(filter_shape, stddev=0.1), name="W")
        b = tf.Variable(tf.constant(0.1, shape=[num_filters]), name="b")
        conv = tf.nn.conv2d(
            self.embedded_chars_expanded,
            W,
            strides=[1, 1, 1, 1],
            padding="VALID",
            name="conv")
        # Apply nonlinearity
        h = tf.nn.relu(tf.nn.bias_add(conv, b), name="relu")
        # Max-pooling over the outputs
        pooled = tf.nn.max_pool(
            h,
            ksize=[1, sequence_length - filter_size + 1, 1, 1],
            strides=[1, 1, 1, 1],
            padding='VALID',
            name="pool")
        pooled_outputs.append(pooled)

# Combine all the pooled features
num_filters_total = num_filters * len(filter_sizes)
self.h_pool = tf.concat(3, pooled_outputs)
self.h_pool_flat = tf.reshape(self.h_pool, [-1, num_filters_total])
```

Figure 4.31: Convolutional layers

Here, W is our filter matrix and h is the result of applying the nonlinearity to the convolution output. Each filter slides over the whole embedding, but varies in how many words it covers. "VALID" padding means that we slide the filter over

our sentence without padding the edges, performing a narrow convolution that gives us an output of shape (1, sequence length - filter size + 1, 1, 1). Performing max-pooling over the output of a specific filter size leaves us with a tensor of shape (batch size, 1, 1, num filters). This is essentially a feature vector, where the last dimension corresponds to our features. Once we have all the pooled output tensors from each filter size we combine them into one long feature vector of shape (batch size, num filters total). Using -1 in tf.reshape tells TensorFlow to flatten the dimension when possible. Visualizing the operations in TensorBoard may help to understand (for specific filter sizes 3, 4 and 5 here), in Fig. 4.32, 4.33.

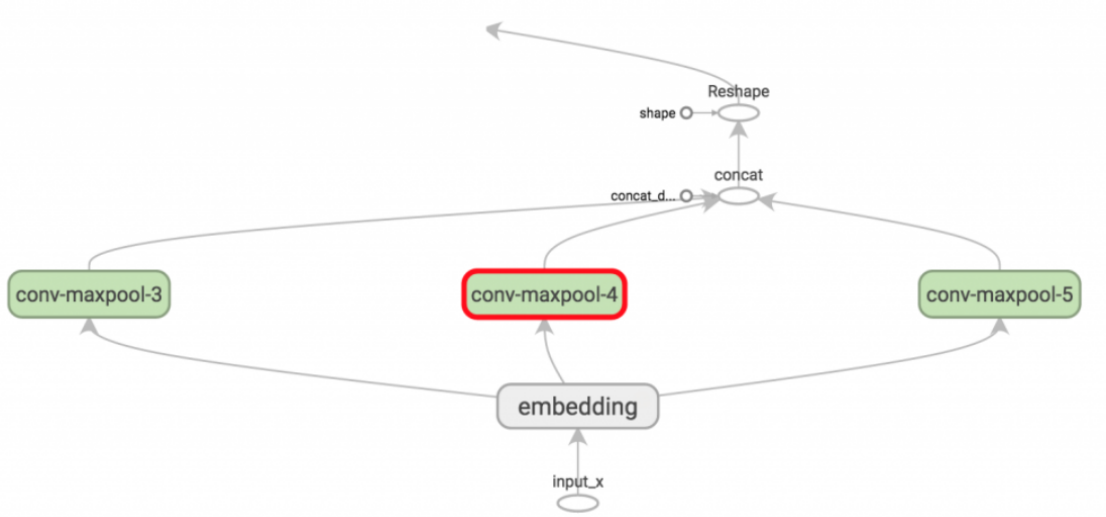


Figure 4.32: Operations on TensorBoard

Dropout is the perhaps most popular method to regularize convolutional neural networks. The idea behind dropout is simple. A dropout layer stochastically "disables" a fraction of its neurons. This prevent neurons from co-adapting and forces them to learn individually useful features. The fraction of neurons we keep enabled is defined by the dropout keep prob input to our network. We set this to something like 0.5 during training, and to 1 (disable dropout) during evaluation (Fig. 4.34).

Using the feature vector from max-pooling (with dropout applied) we can gen-

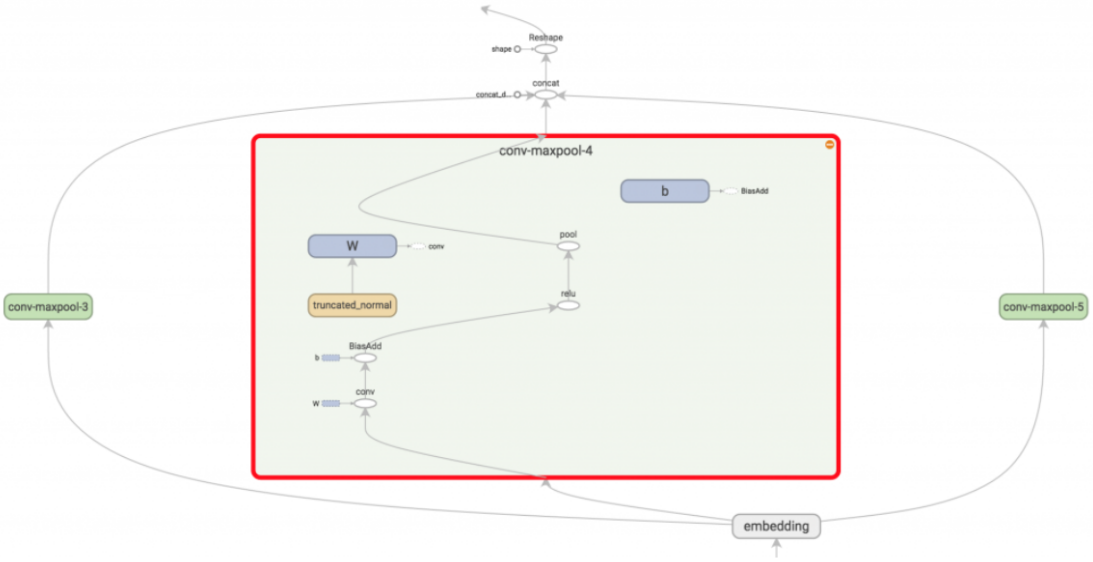


Figure 4.33: Operations on TensorBoard

```
# Add dropout
with tf.name_scope("dropout"):
    self.h_drop = tf.nn.dropout(self.h_pool_flat, self.dropout_keep_prob)
```

Figure 4.34: Dropout Layer

erate predictions by doing a matrix multiplication and picking the class with the highest score (Fig. 4.35). We could also apply a softmax function to convert raw scores into normalized probabilities, but that wouldn't change our final predictions.

```
with tf.name_scope("output"):
    W = tf.Variable(tf.truncated_normal([num_filters_total, num_classes], stddev=0.1), name="W")
    b = tf.Variable(tf.constant(0.1, shape=[num_classes]), name="b")
    self.scores = tf.nn.xw_plus_b(self.h_drop, W, b, name="scores")
    self.predictions = tf.argmax(self.scores, 1, name="predictions")
```

Figure 4.35: Scores and predictions

Using our scores we can define the loss function. The loss is a measurement of the error our network makes, and our goal is to minimize it. The standard loss function for categorization problems it the cross-entropy loss (Fig. 4.36, 4.37).

```
# Calculate mean cross-entropy loss
with tf.name_scope("loss"):
    losses = tf.nn.softmax_cross_entropy_with_logits(self.scores, self.input_y)
    self.loss = tf.reduce_mean(losses)
```

Figure 4.36: Loss

```
# Calculate Accuracy
with tf.name_scope("accuracy"):
    correct_predictions = tf.equal(self.predictions, tf.argmax(self.input_y, 1))
    self.accuracy = tf.reduce_mean(tf.cast(correct_predictions, "float"), name="accuracy")
```

Figure 4.37: Accuracy

We can visualize our CNN in TensorBoard: it looks like in Fig. 4.38.

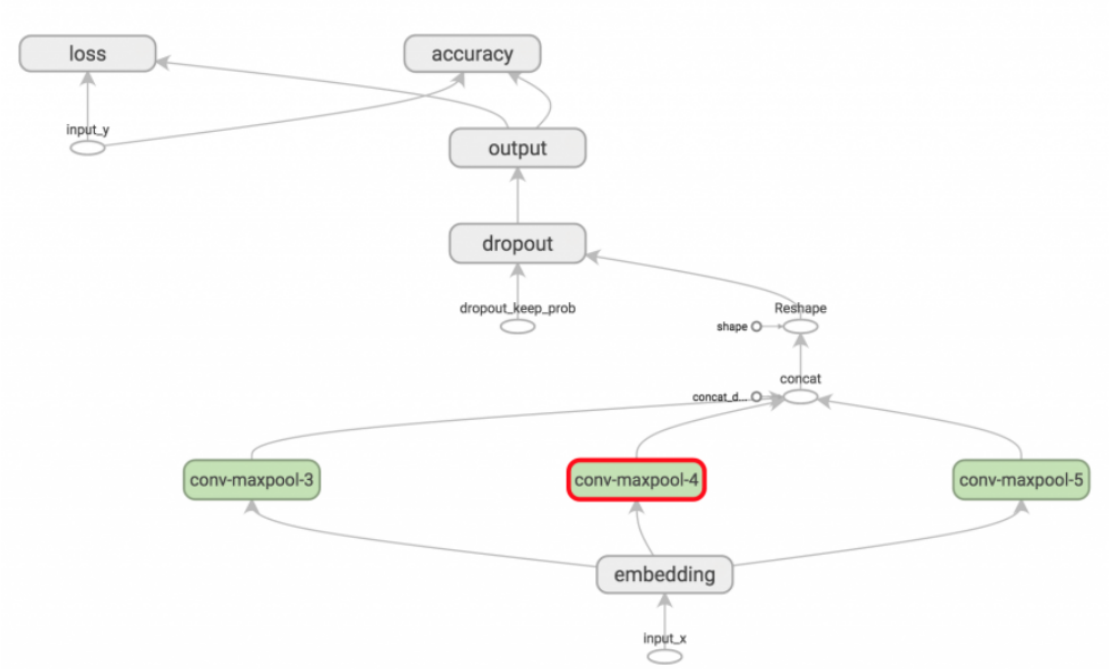


Figure 4.38: CNN in TensorBoard

Now we need to create a dataflow graph as in Fig. 4.39

When we instantiate our TextCNN models, as in Fig. 4.40, all the variables and operations defined will be placed into the default graph and session we've created above.

Next, we define how to optimize our network's loss function. TensorFlow has

```
with tf.Graph().as_default():
    session_conf = tf.ConfigProto(
      allow_soft_placement=FLAGS.allow_soft_placement,
      log_device_placement=FLAGS.log_device_placement)
    sess = tf.Session(config=session_conf)
    with sess.as_default():
        # Code that operates on the default graph and session comes here...
```

Figure 4.39: Dataflow Graph

```
cnn = TextCNN(
    sequence_length=x_train.shape[1],
    num_classes=2,
    vocab_size=len(vocabulary),
    embedding_size=FLAGS.embedding_dim,
    filter_sizes=map(int, FLAGS.filter_sizes.split(",")),
    num_filters=FLAGS.num_filters)
```

Figure 4.40: TextCNN example instantiation

several built-in optimizers. We're using the Adam optimizer (Fig. 4.41).

```
global_step = tf.Variable(0, name="global_step", trainable=False)
optimizer = tf.train.AdamOptimizer(1e-4)
grads_and_vars = optimizer.compute_gradients(cnn.loss)
train_op = optimizer.apply_gradients(grads_and_vars, global_step=global_step)
```

Figure 4.41: Optimizer

Here, train op is a newly created operation that we can run to perform a gradient update on our parameters. Each execution of train op is a training step. TensorFlow automatically figures out which variables are "trainable" and calculates their gradients. By defining a global step variable and passing it to the optimizer we allow TensorFlow handle the counting of training steps for us. The global step will be automatically incremented by one every time you execute train op.

TensorFlow has a concept of a summaries, which allow you to keep track of and visualize various quantities during training and evaluation. For example, you probably want to keep track of how your loss and accuracy evolve over time. You can also keep track of more complex quantities, such as histograms of layer activations. Summaries are serialized objects, and they are written to disk using

a SummaryWriter (Fig. 4.42).

```
# Output directory for models and summaries
timestamp = str(int(time.time()))
out_dir = os.path.abspath(os.path.join(os.path.curdir, "runs", timestamp))
print("Writing to {}\n".format(out_dir))

# Summaries for loss and accuracy
loss_summary = tf.scalar_summary("loss", cnn.loss)
acc_summary = tf.scalar_summary("accuracy", cnn.accuracy)

# Train Summaries
train_summary_op = tf.merge_summary([loss_summary, acc_summary])
train_summary_dir = os.path.join(out_dir, "summaries", "train")
train_summary_writer = tf.train.SummaryWriter(train_summary_dir, sess.graph_def)

# Dev summaries
dev_summary_op = tf.merge_summary([loss_summary, acc_summary])
dev_summary_dir = os.path.join(out_dir, "summaries", "dev")
dev_summary_writer = tf.train.SummaryWriter(dev_summary_dir, sess.graph_def)
```

Figure 4.42: Summary

Another TensorFlow feature you typically want to use is checkpointing – saving the parameters of your model to restore them later on. Checkpoints can be used to continue training at a later point, or to pick the best parameters setting using early stopping. Checkpoints are created using a Saver object (Fig. 4.43).

```
# Checkpointing
checkpoint_dir = os.path.abspath(os.path.join(out_dir, "checkpoints"))
checkpoint_prefix = os.path.join(checkpoint_dir, "model")
# Tensorflow assumes this directory already exists so we need to create it
if not os.path.exists(checkpoint_dir):
    os.makedirs(checkpoint_dir)
saver = tf.train.Saver(tf.all_variables())
```

Figure 4.43: Checkpointing

Let's now define a function for a single training step, evaluating the model on a batch of data and updating the model parameters (Fig. 4.44).

Next, we execute our train op using session.run, which returns values for all the operations we ask it to evaluate. Note that train op returns nothing, it just updates the parameters of our network. Finally, we print the loss and accuracy of the current training batch and save the summaries to disk. Note that the loss

```
def train_step(x_batch, y_batch):
    """
    A single training step
    """
    feed_dict = {
      cnn.input_x: x_batch,
      cnn.input_y: y_batch,
      cnn.dropout_keep_prob: FLAGS.dropout_keep_prob
    }
    _, step, summaries, loss, accuracy = sess.run(
        [train_op, global_step, train_summary_op, cnn.loss, cnn.accuracy],
        feed_dict)
    time_str = datetime.datetime.now().isoformat()
    print("{}: step {}, loss {:g}, acc {:g}".format(time_str, step, loss, accuracy))
    train_summary_writer.add_summary(summaries, step)
```

Figure 4.44: Single training step: first part

and accuracy for a training batch may vary significantly across batches if your batch size is small. And because we're using dropout your training metrics may start out being worse than your evaluation metrics. We write a similar function to evaluate the loss and accuracy on an arbitrary data set, such as a validation set or the whole training set. Essentially this function does the same as the above, but without the training operation. It also disables dropout (Fig. 4.45).

```
def dev_step(x_batch, y_batch, writer=None):
    """
    Evaluates model on a dev set
    """
    feed_dict = {
      cnn.input_x: x_batch,
      cnn.input_y: y_batch,
      cnn.dropout_keep_prob: 1.0
    }
    step, summaries, loss, accuracy = sess.run(
        [global_step, dev_summary_op, cnn.loss, cnn.accuracy],
        feed_dict)
    time_str = datetime.datetime.now().isoformat()
    print("{}: step {}, loss {:g}, acc {:g}".format(time_str, step, loss, accuracy))
    if writer:
        writer.add_summary(summaries, step)
```

Figure 4.45: Single training step: second part

Finally, we're ready to write our training loop. We iterate over batches of our data, call the train step function for each batch, and occasionally evaluate and checkpoint our model (Fig. 4.46).

```
# Generate batches
batches = data_helpers.batch_iter(
    zip(x_train, y_train), FLAGS.batch_size, FLAGS.num_epochs)
# Training loop. For each batch...
for batch in batches:
    x_batch, y_batch = zip(*batch)
    train_step(x_batch, y_batch)
    current_step = tf.train.global_step(sess, global_step)
    if current_step % FLAGS.evaluate_every == 0:
        print("\nEvaluation:")
        dev_step(x_dev, y_dev, writer=dev_summary_writer)
        print("")
    if current_step % FLAGS.checkpoint_every == 0:
        path = saver.save(sess, checkpoint_prefix, global_step=current_step)
        print("Saved model checkpoint to {}\n".format(path))
```

Figure 4.46: Training Loop

Actually running this with the training set we have defined in previous section, based on OpenCrowd categorization of natural language descriptions, and providing as test set some natural language descriptions, randomly taken from different cloud providers of different belonging categories, it has been achieved an accuracy of, see Fig. 4.47.

```
Evaluating...

2018-03-03 22:17:28.385092: I C:\tf_jenkins\home\workspace\rel-win\M\windows-gpu\PY\36\tensorflow\core\platform\cpu_feat
ure_guard.cc:137] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX AVX2
2018-03-03 22:17:29.153593: I C:\tf_jenkins\home\workspace\rel-win\M\windows-gpu\PY\36\tensorflow\core\common_runtime\gp
u\gpu_device.cc:1030] Found device 0 with properties:
name: GeForce 920M major: 3 minor: 5 memoryClockRate(GHz): 0.954
pciBusID: 0000:03:00.0
totalMemory: 2.00GiB freeMemory: 1.67GiB
2018-03-03 22:17:29.161338: I C:\tf_jenkins\home\workspace\rel-win\M\windows-gpu\PY\36\tensorflow\core\common_runtime\gp
u\gpu_device.cc:1120] Creating TensorFlow device (/device:GPU:0) -> (device: 0, name: GeForce 920M, pci bus id: 0000:03:
00.0, compute capability: 3.5)
Total number of test examples: 21
Accuracy: 0.809524
Saving evaluation to C:\Users\tony_\OneDrive\Desktop\nist-cnn-text-classification\runs\1518178313\checkpoints\..\predict
ion.csv
```

Figure 4.47: CNN text classifier results

This is actually a great result!

4.3 Keyword Extractor Implementation

In order to extract keyword from natural language descriptions provided as inputs and to associate to every service, method and parameter specific functionalities, it

has been used RAKE, that is a domain independent keyword extraction algorithm which tries to determine key phrases in a body of text by analyzing the frequency of word appearance and its co-occurance with other words in the text (widely described in section 2.4.4). Fortunately it is already implemented in Nltk: so I just used nltk prebuilt functions in order to built my keyword extractor using RAKE. You can see code in Fig. 4.48: this function "estrai" takes as input a description (a string formed by one or more sentences) and gives back the extracted keywords with their relative scores.

```
def estrai (description):
    from rake_nltk import Rake

    r = Rake() # Uses stopwords for english from NLTK, and all puntuation characters.
    # If you want to provide your own set of stop words and punctuations to
    # r = Rake(<list of stopwords>, <string of puntuations to ignore>)
    r.extract_keywords_from_text(description)
    print (r.get_ranked_phrases()) # To get keyword phrases ranked highest to lowest.
    print (r.get_ranked_phrases_with_scores())
    print()
    return r.get_ranked_phrases()
```

Figure 4.48: RAKE implementation using Nltk

DocSimilarity Meters Implementation

In this section it is described the procedure used to score the similarity between two sentences. In order to make this it has been used another python library, **Spacy**. spaCy excels at large-scale information extraction tasks. It's written from the ground up in carefully memory-managed Cython. Independent research has confirmed that spaCy is the fastest in the world. spaCy is the best way to prepare text for deep learning. It interoperates seamlessly with TensorFlow, PyTorch, scikit-learn, Gensim and the rest of Python's awesome AI ecosystem. With spaCy, you can easily construct linguistically sophisticated statistical models for a variety of NLP problems. spaCy is able to compare two objects, and make a prediction of how similar they are. Predicting similarity is useful for building recommendation systems or flagging duplicates. For example, you can suggest a user content that's

similar to what they're currently looking at, or label a support ticket as a duplicate if it's very similar to an already existing one. This is actually the same purpose for which it is used in this work: to warn developer that a specific functionality very similar to the chosen one already exist in ontologies in order to avoid duplicates and automatically update the ontologies whit the mechanism already discussed in chapter 3. Each Doc, Span and Token comes with a .similarity() method that lets you compare it with another object, and determine the similarity. Of course similarity is always subjective – whether "dog" and "cat" are similar really depends on how you're looking at it. spaCy's similarity model usually assumes a pretty general-purpose definition of similarity. Similarity is determined by comparing word vectors or "word embeddings", multi-dimensional meaning representations of a word. Word vectors can be generated using an algorithm like word2vec. The words "dog", "cat" and "banana" are all pretty common in English, so they're part of the model's vocabulary, and come with a vector. The word "sasquatch" on the other hand is a lot less common and out-of-vocabulary – so its vector representation consists of 300 dimensions of 0, which means it's practically nonexistent. If your application will benefit from a large vocabulary with more vectors, you should consider using one of the larger models or loading yourself vectors in order to enrich the model.

Aside from spaCy's built-in word vectors, which were trained on a lot of text with a wide vocabulary, the parsing, tagging and NER models also rely on vector representations of the meanings of words in context. As the processing pipeline is applied spaCy encodes a document's internal meaning representations as an array of floats, also called a tensor. This allows spaCy to make a reasonable guess at a word's meaning, based on its surrounding words. Even if a word hasn't been seen before, spaCy will know something about it. Because spaCy uses a 4-layer convolutional network, the tensors are sensitive to up to four words on either side of a word.

Word vectors let you import knowledge from raw text into your model. The knowledge is represented as a table of numbers, with one row per term in your vocabulary. If two terms are used in similar contexts, the algorithm that learns the vectors should assign them rows that are quite similar, while words that are used in different contexts will have quite different values. This lets you use the row-values assigned to the words as a kind of dictionary, to tell you some things about what the words in your text mean.

Word vectors are particularly useful for terms which aren't well represented in your labelled training data. For instance, if you're doing named entity recognition, there will always be lots of names that you don't have examples of. For instance, imagine your training data happens to contain some examples of the term "Microsoft", but it doesn't contain any examples of the term "Symantec". In your raw text sample, there are plenty of examples of both terms, and they're used in similar contexts. The word vectors make that fact available to the entity recognition model. It still won't see examples of "Symantec" labelled as a company. However, it'll see that "Symantec" has a word vector that usually corresponds to company terms, so it can make the inference.

In order to make best use of the word vectors, you want the word vectors table to cover a very large vocabulary. However, most words are rare, so most of the rows in a large word vectors table will be accessed very rarely, or never at all. You can usually cover more than 95% of the tokens in your corpus with just a few thousand rows in the vector table. However, it's those 5% of rare terms where the word vectors are most useful. The problem is that increasing the size of the vector table produces rapidly diminishing returns in coverage over these rare terms.

To help you strike a good balance between coverage and memory usage, spaCy's Vectors class lets you map multiple keys to the same row of the table. If you're using the spacy vocab command to create a vocabulary, pruning the vectors will be taken care of automatically. You can also do it manually in the following steps:

- Start with a word vectors model that covers a huge vocabulary. For instance, the en-vectors-web-lg model provides 300-dimensional GloVe vectors for over 1 million terms of English.
- If your vocabulary has values set for the Lexeme.prob attribute, the lexemes will be sorted by descending probability to determine which vectors to prune. Otherwise, lexemes will be sorted by their order in the Vocab.
- Call Vocab.prune-vectors with the number of vectors you want to keep.

Vocab.prune-vectors reduces the current vector table to a given number of unique entries, and returns a dictionary containing the removed words, mapped to (string, score) tuples, where string is the entry the removed word was mapped to, and score the similarity score between the two words. Spacy allows also to you to add new vectors to the vocabulary thanks to the Vocab.set-vector method.

In our work have been used both approaches: the one based on simply comparing word vectors and the one based on context, in order to make also a comparison between them, but they actually presented almost same results, at least on the poor test set that has been used, and they are: for the first one an accuracy of 0.75 and for the other one an accuracy of 0.80. This calculation has been done with a simple division: on a test set of 20 sentences to compare to other 20 sentences, 10 similar and 10 not similar to the first ones, putting a score limit, fixed after some tests to 0.7, above which two sentences can be considered equivalent, we found for the first approach 15/20 right answers and for the second approach 16/20 right answers. This obviously not pretends to be an exhaustive test of this tool offered by spacy, but actually these same techniques have already been tested very well in other works and research papers, as mentioned in chapter 2, so it was actually not a necessary test. You can see the context based approach implementation in Fig. 4.49

```
for doc_list in list:
    doc_found = nlp1(u"{}".format(doc_list))
    for doc in [doc_found, docs]:
        if (doc == doc_found):
            # print (doc)
            for other_doc in docs:
                if (doc.similarity(other_doc) > 0.7):
                    entry = {'extractedFunctionality': doc, 'similarTo': other_doc,
                             'withScore': doc.similarity(other_doc)}
                    similar_functionalities_semantic.append(entry)
```

Figure 4.49: DocSimilarity meter implementation context based using spacy

Chapter 5

Conclusion and Future works

Bibliography

[1] Describe rest web services with wsdl 2.0.

[2] Learning from little: Comparison of classifiers given little training.

[3] OpenAPi Specifications. https://swagger.io/specification/.

[4] Pre-trained GloVe word vectors. http://nlp.stanford.edu/projects/glove/.

[5] Pre-trained word2vec word vectors. https://code.google.com/p/word2vec/.

[6] Smart-api editor.

[7] Swagger-inspector.

[8] Restful webservices. https://www.slideshare.net/gouthamrv/restful-services-2477903, 2008.

[9] CommonMark syntax. http://spec.commonmark.org/, 2017.

[10] Rosa Alarcon and Erik Wilde. From restful services to rdf: Connecting the web and the semantic web.

[11] G. Dinu Baroni, M. and G. Kruszewski. Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors. *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics. Vol. 1.*

[12] D. Horn H. T. Siegelmann Ben-Hur, A. and V. Vapnik. Support vector clustering. *The Journal of Machine Learning Research.*

[13] Antonio Esposito Raaele Giulio Sperandeo Graziella Carta Beniamino Di Martino, Giuseppina Cretella. Ontologies manual.

[14] O. Zimmermann C. Pautasso and F. Leymann. Restful web services vs. "big" web services: Making the right architectural decision. *IW3C2*, 2008.

[15] T. Chen and H. Chen. Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems. *Neural Networks.*

[16] D. Booth et al. Web services architecture. *W3C*, 2004.

[17] E. Christensen et al. Web services description language (wsdl) 1.1. *W3C Note.*, 2001.

[18] M. Gudgin et al. Soap version 1.2 part 1: Messaging framework (second edition). *W3C Recommendation*, 2007.

[19] R. Fielding et al. Hypertext transfer protocol – http/1.1. *IETF RFC 2068*, 1997.

[20] R. Fielding et al. Hypertext transfer protocol – http/1.1. *IETF RFC 2616*, 1999.

[21] T. Berners-Lee et al. Hypertext transfer protocol - http/1.0. *IETF RFC 1945*, 1996.

[22] Jian Mao Robert Bohn John Messina Lee Badger Fang Liu, Jin Tong and Dawn Leaf. NIST 500-292. `http://ws680.nist.gov/publication/get_pdf.cfm?pub_id=909505`, 2011.

[23] R. Fielding. Architectural styles and the design of network-based software architectures. phd thesis. *University of California*, 2000.

[24] Oracle Freed N. and Klensin J. Media Type Specifications and Registration Procedures. `https://tools.ietf.org/html/rfc6838`, 2013.

[25] K Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics 36.4.*

[26] H. Kuno G. Alonso, F. Casati and V. Machiraju. Web services: Concepts,architecture and applications. *Springer Verlag*, 2004.

[27] L. Deng M. Gamon X. He Gao, J. and P. Pantel. Modeling interestingness with deep neural networks. *US Patent 20,150,363,688.*

[28] A. Bordes Glorot, X. and Y. Bengio. Deep sparse rectifier neural networks. *International Conference on Artificial Intelligence and Statistics.*

[29] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *International conference on artificial intelligence and statistics.*

[30] B. Allison W. Liu L. Guthrie Guthrie, D. and Y. Wilks. A closer look at skip-gram modelling. *Conference on Language Resources and Evaluation (LREC-2006).*

[31] X. Zhang S. Ren He, K. and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *arXiv preprint arXiv:1502.01852.*

[32] D. H. Hubel and T. N. Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology 195.1.*

[33] V. Manjunatha J. Boyd-Graber Iyyer, M. and H. D. Deep unordered composition rivals syntactic methods for text classification. 2015.

[34] R. Johnson and T. Zhang. Effective use of word order for text categorization with convolutional neural networks. *arXiv preprint arXiv:1412.1058*.

[35] E. Grefenstette Kalchbrenner, N. and P. Blunsom (. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188*.

[36] Y Kim. Convolutional neural networks for sentence classification. *CoRR abs/1408.5882*.

[37] Yoon Kin. Convolutional neural networks for sentence classification.

[38] I. Sutskever Krizhevsky, A. and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*.

[39] et al. L. Richardson, S. Ruby. Restful web services. *O'Reilly*, 2007.

[40] Q. V. Le and T. Mikolov. Distributed representations of sentences and documents. *arXiv preprint arXiv:1405.4053*, 2014.

[41] R. Lebret and R. Collobert. Rehabilitation of count-based models for word vector representations. *Computational Linguistics and Intelligent Text Processing*.

[42] L. Bottou G. B. Orr LeCun, Y. A. and K.-R. Müller. Efficient backprop. *Neural networks: Tricks of the trade*.

[43] Y. Bengio LeCun, Y. and G. Hinton. Deep learning. *Nature 521.7553*.

[44] van Harmelen F. McGuinness, D. L. OWL Web Ontology LanguageOverview. `http://www.w3.org/TR/2004/REC-owl-features-20040210/`, 2004.

[45] I. Sutskever K. Chen G. S. Corrado Mikolov, T. and J. Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 2013.

[46] K. Chen G. Corrado Mikolov, T. and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[47] W.-t. Yih Mikolov, T. and G. Zweig. Linguistic regularities in continuous space word representations. *HLT-NAACL*, 2013.

[48] OpenCrowd. Cloud Taxonomy. `http://cloudtaxonomy.opencrowd.com/taxonomy/`, 2011.

[49] R. Socher Pennington, J. and C. D. Manning. Glove: Global vectors for word representation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, 2014.

[50] editor. R. Fielding. Rfc for rest. *REST Discussion Mailing List*, 2006.

[51] Jim Hendler Ian Horrocks Deborah L. Mcguinness Peter F. Patel-Schneider Sean Bechhofer, Frank van Harmelen and Lynn A. Stein. Owl web ontology language reference. *W3C*, 2004.

[52] L. Wolf S. Bileschi M. Riesenhuber Serre, T. and T. Poggio. Robust object recognition with cortex-like mechanisms. *Pattern Analysis and Machine Intelligence.*

[53] X. He J. Gao L. Deng Shen, Y. and G. Mesnil. A latent semantic model with convolutional-pooling structure for information retrieva. *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management.*

[54] G. Hinton A. Krizhevsky I. Sutskever Srivastava, N. and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research 15.1.*

[55] Nick Cramer Stuart Rose, Dave Engel and Wendy Cowley. Automatic keyword extraction from individual documents.

[56] D. Sussillo. Random walks: Training very deep nonlin-ear feed-forward networks with smart ini. *arXiv preprint arXiv:1412.6558.*

[57] Y Tang. Deep learning using linear support vector machines. *arXiv preprint arXiv:1306.0239.*

[58] H. Wu. Global stability analysis of a general class of discontinuous neural networks with linear growth activation functions. *Information Sciences 179.19.*