Introduction to *In-Silico* learning
Machine Learning Algorithms

Abstract

After analysing some data properties via static reporting and making sure that there exist a correlation between the predictors and the class to be learned of choice, we exploit three supervised learning approaches (Multilayer Neural Networks, Decision Trees and $\nu-$Support Vector Machines with Kernel Tricks) for fitting a model based on the data of choice. Finally, we're going to exploit the former mathematical formulations for generalize and then compare the different data model of choices and thus remarking their pros and cons.

1 Static Report

Let's go back to the previous tutorial, and let us focus on a real game use case example: StarCraft II is a science fiction real-time strategy video game, where the player needs to make his civilization's economic progress so to dominate the other players in the battles. Giving the odds that any player and his opponents have to win a battle, they might increase or decrease their score, so they can dynamically change the expertise level, called league index[1]. The way how such score can be increased or decreased is poorly understood [2]: the reason why is the game is closed source, and so we don't know the punish and reward tactics that are effectively implemented in the game. In this case, we're wondering whether the level increase of each player is dictated by some specific game logic or not. To answer these questions, we can sample some data that the game API can give us (Table 1): in this dataset, we have no information that can us possibly correlate the user's actions in the whole gameplay (both economic and battle strategy) to an increase of the XP level via the outcome of every battle. Still, we can establish correlations between all the past actions performed by a player to his current XP level, which is pretty close to our initial aim.

A better correlation between gameplay strategies and the battles' outcome over a different dataset will be covered in the next and last *In-Silico* learning tutorial, where a completely different data schema is going to lean us to a completely different learning approach.

Before we start our analysis, we wonder if there is any correlation between the predictors and the XP level: in order to do so, we need to create a static and statistic report of our dataset[3]. **Static reporting** is a process creating a final report including non-real time information about a specific resource such as inventory, or a collection of periodically created resources. In particular, a specific tool retrieves the pieces of information that are relevant to the final analysis and outputs a table as the one provided in Table 1: as you might see from the provided average information, such information could also summarize several pieces of information into just one single value, thus providing a more compact and human readable representation (*data aggregation* or *summarization*). This is the preliminary and more crucial step for decision support, that is - alas - often overlooked by middling researchers [6]. In fact, violations of the statistical assumptions might produce biased or irrelevant trained models, that can be easily avoided if we examine each single variable and also the relationship between those. Moreover, only if we are able to understand the basic characteristics of the underlaying data and relationships, we will be able to pick the best data model or the task that we need to solve (Figure 1). As a naïf example, we cannot possibly try to use one single neuron or a SVM without kernels if we want to describe data that is not clearly linearly separable.

[1] `https://liquipedia.net/starcraft2/Battle.net_Leagues`

[2] `https://tl.net/forum/starcraft-2/115901-an-incomplete-guide-to-bnet-20-ranking-system`

[3] A more all-embracing tutorial on several different techniques of static and statistical reporting is provided in my other set of slides: `https://drive.google.com/file/d/0B3tBL-tX2EdQV2Q4Znh5UnBrTXM/view`.

Age	HoursPerWeek	TotalHours	APM	SelectByHotkeys	AssignToHotkeys	UniqueHotkeys	MinimapAttacks	MinimapRightClicks	NumberOfPACs	GapBetweenPACs	ActionLatency	ActionsInPAC	TotalMapExplored	WorkersMade	UniqueUnitsMade	ComplexUnitsMade	ComplexAbilityUsed	MaxTimeStamp	LeagueIndex
27	10	3000	143718	0.003515159	0.000219697	5.49e-05	0.000109849	0.000392317	0.004849036	32.6677	40.8673	4.7508	0.000219697	0.0013966	4.71e-05	0	0	127448	5
23	10	5000	129.2322	0.003303812	0.000259462	6.92e-05	0.000294057	0.000432436	0.004307064	32.9194	42.3454	4.8434	0.000380544	0.0011935	8.65e-05	0	0.00020757	57812	5
30	10	200	69.9612	0.001101091	0.00033557	4.19e-05	0.000293624	0.000461409	0.002925755	44.6475	75.3548	4043	0.000230705	0.00074455	6.29e-05	0	0.00018876	95360	4
19	20	400	107.6016	0.001033542	0.000213101	1.07e-05	5.33e-05	0.000543409	0.003782551	29.2203	53.7352	4.9155	0.000202446	0.0004262	7.46e-05	0	0.00038358	93852	3
32	10	500	122.8908	0.001136014	0.000327326	3.85e-05	0	0.001328558	0.002368299	22.6885	62.0813	9374	0.000288817	0.0011745	7.7e-05	0	1.93e-05	51936	3
27	6	70	44457	0.00097839	0.000255232	2.13e-05	0	0	0.002424706	76.4405	98.7719	3.0965	0.000170155	0.00037221	6.38e-05	0	0	94032	2
21	8	240	46.9962	0.000820114	0.000168517	6.74e-05	0	4.49e-05	0.001988496	94.0227	90.5311	4.1017	0.000168517	0.00057296	5.62e-05	0	0	89012	1
17	42	10000	212.6022	0.009039739	0.00067624	5.97e-05	0.001163531	0.001253033	0.004952464	24.6117	41.7671	6.6104	0.000447512	0.0022773	8.95e-05	0.000129281	0.00024862	100556	7
20	14	2708	117.4884	0.002944275	0.000526771	1.88e-05	1.88e-05	0.000413892	0.005399406	52014	46.4321	3.3746	0.000272792	0.0010347	6.58e-05	0.000272792	0.00047033	106308	4
18	24	800	155.9856	0.005053908	0.000524109	9.98e-05	2.5e-05	0.000399321	0.003568933	24.4632	52.1538	6.5664	0.000336927	0.0013103	7.49e-05	0	0	80136	4
16	16	6000	153801	0.001676615	0.000318557	6.71e-05	0	0.000821541	0.003772383	23.4107	48.0711	7.0044	0.000402387	0.0015928	0.000117363	0	1.68e-05	59644	3
26	4	190	79.2948	0.000378539	0.000255102	2.47e-05	1.65e-05	0.000164582	0.00355497	39.6381	65.5	4.2269	0.000156353	0.00075708	5.76e-05	0.000106978	0.00026333	121520	4
18	12	350	67.4754	0.000422522	0.000169009	1.21e-05	2.41e-05	0.000144865	0.002885219	42437	68.0502	4.3222	0.000193153	0.00074847	8.45e-05	0	0.00043459	82836	3
38	6	1000	119.4366	0.004952043	5.21e-05	3.48e-05	8.69e-05	3.48e-05	0.002727968	54.8718	79.2102	6.2293	0.000364887	0.0014943	8.69e-05	0	0	57552	3
16	30	5000	160.4754	0.00425386	0.000431735	2.54e-05	0.000774584	0.000406339	0.004571312	36.2897	46.8889	5.4361	0.000355547	0.0019809	8.89e-05	0	0	78752	5
17	16	1500	81.7722	0.002333484	0.000430449	9.06e-05	0	0.000249207	0.002899864	45.1654	64.75	4.5312	0.000339828	0.001246	0.000135931	0	0.00022655	44140	5
28	8	2000	50.8374	0.000664109	0.00022137	1.84e-05	1.84e-05	0.00014758	0.002656434	45.7902	76.8889	3.5	0.000239817	0.00066411	0.000110685	0	0.00029516	54208	4
20	10	120	160.6464	0.003430344	0.00063363	5.1e-05	0	9.47e-05	0.005535163	28.3636	37.7947	4.7671	0.00021121	0.0027676	7.28e-05	8.74e-05	0.00010196	137304	5
16	14	350	107.9118	0.006701306	0.000706102	6.66e-05	1.33e-05	0.000466294	0.00323741	67.0744	71.3251	4.3786	0.000359712	0.0013323	0.000119904	0.000239808	0	75060	5
26	28	1100	114.7806	0.002629613	0.000287389	7.18e-05	0.000244281	0.000344867	0.003649845	40.4427	59937	4.9961	0.000229911	0.0015806	7.18e-05	0	0	69592	4
21	10	800	115.1274	0.002651148	0.000660037	8.8e-05	9.9e-05	0.000693039	0.004147232	30766	49.4854	4679	0.000330018	0.00082505	8.8e-05	0	0.00051703	90904	5
21	6	500	133.7016	0.004499795	0.000420105	2.8e-05	1.87e-05	7.47e-05	0.003874304	21.4686	50.5253	5.4892	0.000270735	0.0010083	9.34e-05	0	0.00056014	107116	6
18	20	800	99.5088	0.00073402	0.000108298	1.2e-05	2.41e-05	0.00057759	0.002695418	22.7265	68.7321	6.7054	0.000276762	0.0019614	8.42e-05	0	0	83104	5
26	10	500	83.9172	0.002853513	0.000492966	2.84e-05	0	0	0.004607334	51.2082	52.0494	2.8663	0.000293883	0.00065413	8.53e-05	0	0	105484	5
17	14	500	216.6936	0.012494649	0.000481592	0.000107021	5.35e-05	0.000107021	0.004334332	27.9255	45.1605	6.3395	0.000481592	0.001204	0.000133776	0	0	37376	4
23	20	800	129.8598	0.002315078	0.000333769	3.55e-05	7.81e-05	0.001008408	0.003472617	36.3934	57.5215	6.7587	0.000255653	0.00075276	7.81e-05	0.000284059	0.0012286	140816	4
18	70	2520	267.5586	0.027814934	0.00070816	9.84e-05	0	0.000127862	0.005616099	34.6035	40.6025	4.1629	0.00035408	0.00085569	0.000118027	8.85e-05	0.00019671	101672	6
25	6	800	74.1174	0.000875	0.000205357	5.36e-05	1.79e-05	4.46e-05	0.003517857	46.4122	71.8173	3.7944	0.000303571	0.00092857	4.46e-05	0	0	112000	5
25	20	700	101.6796	0.001406194	0.000427198	8.9e-05	0	0.0001246	0.003453186	26.7979	54.5979	5.3144	0.000391598	0.0010324	7.12e-05	0	0.0005696	56180	5
18	10	160	150.5004	0.005666689	0.000631529	0.00010241	0.000375504	0.00040964	0.004625521	30.2222	45.5941	4.9077	0.000238957	0.001007	0.00010241	0	0	58588	6
18	8	250	41.9094	0.000275751	0.000190904	0	0	0.000169693	0.002142372	54.4677	91.3069	3.6881	0.000180299	0.00066817	6.36e-05	0	2.12e-05	94288	3
16	16	1000	128.0784	0.002753792	9.5e-05	2.53e-05	0.000240561	0.000164594	0.003969259	35393	56.4019	5.2711	0.000170925	0.00043048	5.06e-05	0	0	157964	5
16	28	500	161.3466	0.006109907	0.000577482	6.42e-05	0.000327953	0.0007272	0.00413506	34.1209	47.1034	5.8466	0.000342212	0.0013475	6.42e-05	0.000377859	0.00017111	140264	6
22	4	400	90.7686	0.001610454	0.000392794	5.24e-05	0	0.000183304	0.003469676	44.7424	59.5321	4.5698	0.000196397	0.00070703	7.86e-05	0	0.00032733	76376	4
20	14	730	162.0876	0.007596727	0.000571635	0.000105301	0.000255731	0.000752151	0.003986401	23.5303	51034	5.7698	0.000421205	0.0026024	9.03e-05	0	1.5e-05	66476	6
18	6	230	69.5076	0.00017507	0.000204248	0.000116713	5.84e-05	0.000204248	0.001458917	45.0612	112.96	9.14	0.000320962	0.00058357	0.000116713	0	0	34272	1
16	24	300	81.0702	0.001746793	0.000104286	0	2.61e-05	0.000143393	0.00262019	47.48	75.0846	5.0796	0.000182501	0.00080822	7.82e-05	2.61e-05	3.91e-05	76712	3
22	8	300	157332	0.007742665	0.000223472	1.31e-05	0.000131454	0.000985908	0.003799032	33.0417	57.7716	5.8581	0.000381218	0.0018929	7.89e-05	0	0	76072	5
23	4	100	67.6194	0.001565762	0.000398308	5.49e-05	4.12e-05	0.00112625	0.001661905	49.4333	89.9174	7.1322	0.000233491	0.00083782	8.24e-05	9.61e-05	0.0002747	72808	4
25	10	200	75.9642	0.000609665	0.000136864	4.98e-05	0.000124421	0.000323496	0.00209028	45.1018	78.3095	6.5417	0.000186632	0.00093316	7.47e-05	0	0	80372	2
25	12	200	164.1114	0.003372032	0.000403892	6.57e-05	0.000112714	0.001568605	0.004987601	40.4604	43.7815	5.6365	0.000300571	0.0012774	7.51e-05	0.000216035	0.0010896	106464	5
20	14	270	101.7864	0.001453175	0.000538213	3.59e-05	5.38e-05	0.000304987	0.004180122	22569	54.6094	4.2961	0.000269107	0.00096878	7.18e-05	0	0.00021529	55740	4
25	8	100	79128	0.00050077	8.99e-05	2.57e-05	1.93e-05	0.001720596	0.002754237	50.5327	67.1422	5345	0.000179764	0.00043657	3.85e-05	0	6.42e-06	155760	2
22	20	1200	120.7014	0.005495624	0.000239888	3.63e-05	0	2.18e-05	0.003612864	44.8387	60.9175	4.8089	0.00022535	0.00051612	3.63e-05	0	0	137564	4
18	20	800	108.5424	0.002100536	0.000398378	7.24e-05	0	0.000434594	0.003078372	28.4762	61.9294	5.8353	0.000398378	0.0025351	0.000144865	0	0	27612	6
23	2	30	66.7818	0.000685853	0.000168228	2.59e-05	5.18e-05	0.001112894	0.002406957	56.7784	97.9785	4.9355	0.000245872	0.00042704	7.76e-05	0	0	77276	3
18	28	2000	129.1464	0.00263107	0.000599729	3.87e-05	7.74e-05	0.000338557	0.005146063	25.0245	43.0827	4.1917	0.000338557	0.00083188	8.71e-05	0.000174115	0.00048365	103380	6
18	8	600	62.2194	0.001099196	0.000321716	0	4.02e-05	0.000201072	0.002386059	44.3164	90.8989	4.5393	0.000187668	0.00071046	9.38e-05	0	0.00010724	74600	3
20	6	150	72141	0.001237902	0.000393878	4.69e-05	1.88e-05	0.000271963	0.003376097	45.9164	68.8778	3.7528	0.000225073	0.00079713	7.5e-05	0.000121915	0.00035637	106632	3
23	20	100	83.9658	0.002672697	0.000376919	0.000102796	0	0.000171327	0.002912555	44.0476	78.0706	4.6118	0.000342654	0.00071957	0.000102796	0	0	29184	4
17	56	540	199.6854	0.005146297	0.000453133	1.62e-05	3.24e-05	0.0002913	0.004790264	25.3831	33.9054	6.7095	0.000460316	0.0020876	0.000113283	0	1.62e-05	61792	5
29	12	800	66.6738	0.000320895	4.58e-05	2.29e-05	0	0.000194829	0.002292106	38.5126	72.16	5.46	0.000229211	0.0010314	0.000103145	0.000252132	0.00014899	87256	4
27	24	250	73.9722	0.002647278	0.000454884	5.22e-05	7.46e-06	0.000186428	0.002684564	58.8858	88.6889	4.4111	0.000178971	0.00058166	5.97e-05	0.0001044	0.00023863	134100	2
21	8	730	130941	0.004758302	0.00075655	0.000199092	7.96e-05	0.000139364	0.003384566	36.8047	57.2471	5.7765	0.000219001	0.00059728	3.98e-05	0	0	50228	6
27	8	280	55047	0.000642809	0.000126454	0	7.38e-05	9.48e-05	0.002529084	62.8452	83.8667	3.9333	0.000231833	0.00052689	6.32e-05	0	0.00017914	94896	2
16	10	1600	181104	0.00563541	0.000581793	0.000114077	0	0.000456308	0.005007985	20.8037	39.2255	5.6993	0.000342231	0.0017226	9.13e-05	0.000365047	0.001175	87660	6
22	36	1000	98.6694	0.004474848	0.000462915	2.57e-05	0.000257175	0.000360045	0.002391729	52.6957	62.5806	5828	0.000437198	0.0005915	7.72e-05	0	0	38884	4
18	6	50	107.4186	0.003028577	0.000310623	5.18e-05	0.000220025	0.000336509	0.003559226	37.3723	56.0436	5.0873	0.000310623	0.00095776	9.06e-05	0	0	77264	4
0	0	0	189.7404	0.004582151	0.000654593	7.27e-05	7.27e-05	0.000618227	0.006291367	23513	32.5665	4.4451	0.000454579	0.0022183	0.000109099	0	0	54996	8
0	0	0	287.8128	0.029039722	0.001041265	0.000173544	0.000231392	0.000655611	0.005399152	31.6416	36.1143	4.5893	0.000655611	0.0011377	0.000115696	5.78e-05	0	51860	8
0	0	0	294.0996	0.029639771	0.001075709	5.33e-05	0.000302265	0.002373671	0.006294229	16.6393	36.8192	4185	0.000231144	0.00098681	5.33e-05	0	0	112484	8
0	0	0	274.2552	0.018120523	0.001264223	0.000105352	5.27e-05	0.000974505	0.007111252	10.6419	24.3556	4387	0.000368732	0.0011062	7.9e-05	0	0	75936	8
0	0	0	274.3404	0.023131288	0.000739192	7.21e-05	0.000622003	0.003551725	0.005354632	19.1568	36.3098	5.2811	0.000252407	0.00073919	5.41e-05	0	0	110932	8
0	0	0	245.8188	0.010471414	0.000841238	8.01e-05	0.000656967	0.001313934	0.005031406	14.5518	36.7134	7.1943	0.000264389	0.0014742	8.81e-05	4.01e-05	4.81e-05	124816	8
0	0	0	211.0722	0.01304902	0.000940282	0.000104476	0.000365665	0.000908939	0.003719336	19.6169	38.9326	7132	0.000240294	0.00089849	9.4e-05	0	0	95716	8
0	0	0	189.5778	0.007558997	0.000487252	0.00013169	0.000605773	0.000566266	0.005820691	22.0317	36733	4905	0.000368732	0.00053993	6.58e-05	0	0	75936	8
0	0	0	210.5088	0.007973528	0.000867121	6.98e-05	0.00054818	0.000637882	0.006518359	15.7856	30.7156	4.8058	0.000338875	0.00081729	5.98e-05	0	0	100332	8
0	0	0	248.0118	0.014721527	0.00175219	0.000109512	0.000375469	0.000109512	0.004114518	17.4656	34.2357	7.8973	0.000312891	0.0011108	0.000125156	0	0	63920	8
0	0	0	299229	0.026428161	0.00095137	0.000110624	0.000154874	0.000929245	0.005442719	17.0835	33.7398	5.2703	0.000176999	0.00069693	6.64e-05	3.32e-05	1.11e-05	90396	8

Table 1: A subset of the StarCraft II dataset available online at `http://summit.sfu.ca/item/13328`. This table subset provides the following fields: *(i)* the age of each player $(X_1, \ldots)$, *(ii)* the reported hours spent playing per week, *(iii)* the reported total hours spent playing, *(iv)* the number of continuous actions per minute, *(v)* the number of item selections via hotkeys per time frame, *(vi)* the number of assigned items via hotkeys per time frame, *(vii)* the number of unique and continuous hotkeys used per time frame, *(viii)* the number of continuous attack on a minimap per time frame, *(ix)* the number of continuous right clicks on a minimap per time frame, *(x)* the number of PACs per time frame, *(xi)* the mean duration between PACs, *(xii)* the mean latency from the onset of PACs to their first action, *(xiii)* the mean number of actions within each PAC, *(xiv)* the total number of 24×24 game coordinate grids viewed by player per time frame, *(xv)* the number of battle items trained per time frame, *(xvi)* the number of unique units made per time frame, *(xvii)* the number of complex battle items trained per time frame, *(xviii)* the number of abilities requiring specific targeting instructions used per time frame, *(xix)* Time stamp of game's last recorded event $(\ldots, X_{19})$, and *(xx)* the target XP level class (Y).

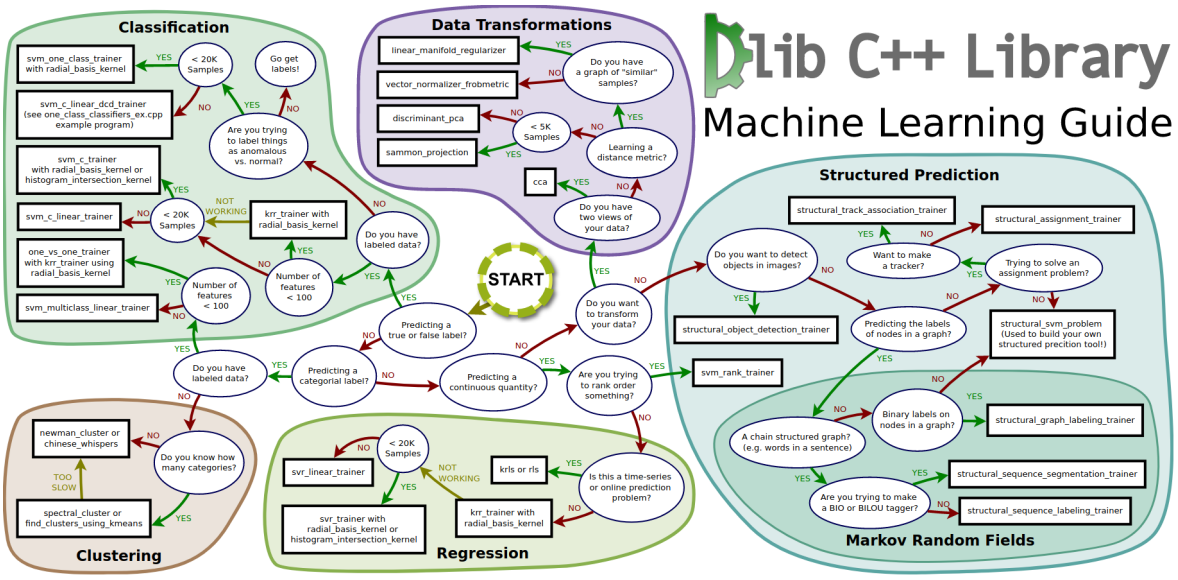
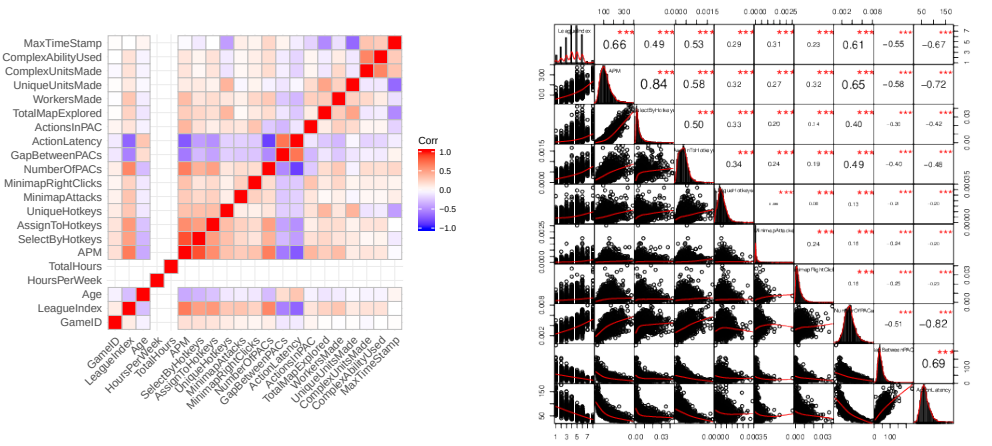


Figure 1: *The knowledge of the basic features expressed by the data is the guiding force for choosing the best data model.*



(a) Correlation Matrix for the StarCraft II Replay Dataset. White or transparent tiles represent variables with no evident correlation.

(b) Scatter-plot matrix for the best `LeagueIndex` predictor for our dataset. The numbers on the upper part of the matrix represent ρ.

Figure 2: Some static report figures for multi-variate data analysis performed in R.

1.1 Correlation Matrix

One of the simples techniques that we can possibly do is to find if there is any possible correlation among the predictors and the class value is to use a **correlation matrix** to graphical represent the positive (negative, or null) correlation of the values associated to each attribute. Figure 2a provides a colorized representation of such matrix: darker blue (red) tones describe data being negatively (positively) correlated, while white or transparent tiles present data elements with no correlations. From this plot, we can observe that the best predictors for the `LeagueIndex` are all the attributes from `APM` to `ActionLatency`. That consideration is relevant if we want to reduce the dimension of the data that we want to train: as we will observe, the trained models on this subset of data will provide similar precision results as the ones trained over the full set of attributes. So, such dimensionality reduction will be extremely relevant in the context of big data scenarios.

We now provide some theoretic formalization required to understand the insights of the depicted matrix, and to better understand the meaning of *correlation*.

Formalization (*) In some sample data from a table t with schema $R(X_1, \ldots, X_n, Y)$, a **random variable** X_i can be considered as each one of the attributes in such schema, where the set $I(X_i) \subseteq dom(X_i)$ representing all the possible values associated to X_i in each record r of t (more formally, $I(X_i) = \{\, r.X_i \mid r \in t \,\}$). For each value $x \in I(X_i)$ we can define an independent **event** $E_x := (X_i = x)$ which **probability** $\mathbb{P}(E_x)$ is defined as the frequency of x in t as a value for an attribute X_i (more formally, $\mathbb{P}(X_i = x) = \frac{|\{\, r \mid r.X_i = x \,\}|}{|t|}$). The **prediction** $\mathbb{P}(X_i)$[4] of a random variable is defined (in these use case scenario) as the *sample mean* of all the values occurring in t, that is:

$$\mathbb{P}(X_i) = \frac{1}{|t|} \sum_{r \in t} r.X_i = \sum_{x \in I(X_i)} \mathbb{P}(X_i = x)x$$

Covariance $\mathbf{cov}(X_i, X_j)$ is the measure of the joint variablity of two random variables X_i and X_j. When one variable is higher values correspond mostly to the higher values of the other variable, and the same applies for the lower values, then the covariance is positive ($\mathbf{cov}(X_i, X_j) > 0$). On the other hand, the two variables are not correlated ($\mathbf{cov}(X_i, X_j) = 0$) if any value that X_i can assume is completely unrelated to the value assumed by X_j, and therefore the probability $\mathbb{P}(X_i = x, X_j = x')$ that two random variables X_i and X_j will respectively assume the value x and x' is the same as the probability of the same two events happening separately, i.e. $\mathbb{P}(X_i = x)\mathbb{P}(X_j = x')$. Otherwise (when the greater values of one variable mainly correspond to the lesser values of the other), the covariance will be negative ($\mathbf{cov}(X_i, X_j) < 0$). A formal definition of such covariance is provided in the following equation:

$$\mathbf{cov}(X_i, X_j) = \mathbb{P}((X_i - \mathbb{P}(X_i)) \cdot (X_j - \mathbb{P}(X_j))) = \mathbb{P}(X_i X_j) - \mathbb{P}(X_i)P(X_j)$$

We can then prove [4] that this equation reduces for sample data to the following fomulation:

$$\mathbf{cov}(X_i, X_j) = \sum_{x \in I(X_i)} \sum_{x' \in I(X_j)} x \cdot x' \mathbb{P}(X_i = x,\ X_j = x')$$

In particular, **variance** measures the distance of a set of random numbers from their average value, and it can be defined as $\sigma^2(X_i) = \mathbf{cov}(X_i, X_i)$: when $\sigma^2(X_i) = 0$, then we can say that the most probable outcome of X_i is the sample mean $\mathbb{P}(X_i)$. We call **standard deviation** the square root of the variance: $\sigma(X_i) = \sqrt{\sigma^2(X_i)}$.

All these definitions lead to the definition of the **correlation coefficient** $\rho(X_i, X_j)$ expressing the degree of linear correlation of two random variables. We can also interpret this measure as a normalized representation of the covariance, so that $Im(\rho) = [-1, 1] \subseteq \mathbb{R}$. We can formally define such coefficient as $\rho(X_i, X_j) = \frac{\mathbf{cov}(X_i, X_j)}{\sigma(X_i)\sigma(X_j)}$. We can prove that, given the properties of this function, we can also normalize any random variable X_i as $X_i^* = \frac{X_i - \mathbb{P}(X_i)}{\sigma(X_i)}$, so that the numbers will have a mean of zero and a variance of zero. This concept will be also used in ν-Support Vector Machines to normalize our training set.

Therefore, a (square) correlation matrix M as the one depicted in Figure 2a represents the value $\rho(X_i, X_j)$ for each possible variable X_i, X_j in the dataset. We can obtain the definition of the sample correlation coefficient from the formal definition as follows:

$$\rho(X_i, X_j) = \frac{\sum_{r \in t}(r.X_i - \mathbb{P}(X_i))(r.X_j - \mathbb{P}(X_j))}{\sqrt{\sum_{r \in t}(r.X_i - \mathbb{P}(X_i))^2 \sum_{r \in t}(r.X_j - \mathbb{P}(X_j))^2}}$$

1.2 Scatterplot Matrix

A scatterplot matrix allows a more accurate profiling of N metric variables. Figure 2b represents a scatterplot matrix for the subset of the variables that are the best candidate predictors for the `LeagueIndex`: while the values above such matrix actually express the actual correlation values, the diagonal represents **histograms** approximating the data distribution for each attribute. In order to construct a histogram for an attribute X_i, we must divide the $I(X_i)$ values into a series of intervals

[4] Also written as μ_X

$[x, x']$, and then represent each bin as a vertical bar, which height is proportional to $\mathbb{P}(x \leq X_i \leq x')$. Last, each scatter plot below the diagonal represent the scatterplots for each pair of variables: as we might see from the data, there is no clear linear correlation between the predictor and one of the variables, thus implying that there is no clear planar correlation of the data. We might also observe with some further plots that such dataset is not even linearly separable.

2 Characterizing Learning

After analysing our data, we can observe that one user cannot contemporary belong to multiple XP levels, and therefore we can model the function that we want to learn as a function h. Moreover, given that the different classes level can be represented as numbers and that there is an ordering such that the higher XP level intuitively corresponds to the most senior experience, $\tilde{h}$ can be learned via a regression function. As described in [13], a set of possible explainations provided by an hypothesis h could be the following:

- Direct mapping between input and output (i.e., the h function itself described as a map).
- *Relevant properties* from the data that is perceived from the **training dataset** $D \subseteq E$: if the map learned from such training dataset can be expressed by a mathematical function or as a set of logical rules motivating the decision.
- Information about the way the *world evolves in time*: this implies that the learning model is not forgetful and it can store all the learned h_t at a given timestamp in a history $\mathcal{H}$. At this stage, an *agent* can possibly learn from his past experience $\mathcal{H}$ and try to generalize from the former examples as a forecasting function $\overline{h}$: $\mathcal{L}_h \to \mathcal{L}_h$. Paper [5] or Time Series Analysis are possible examples for this approach.
- Learning the *desirability* of world states or actions: this other kind of learning tasks require advanced knowledge of statistics and are not going to be covered in this module. These approaches can be assimilated to Data Mining algorithms as in [11] where probability is also used. Markov Logic Networks[5] are a possible example of learning objectives with associated probability values.

While the first three steps are one the generalization of the other, the last requirement can be implemented independently from the others. In these tutorials, we will approximate the desirability by trying to learn $\tilde{h}$ instead of h, thus estimating the trustworthiness of the classification outcome, and we will not consider learning the evolution of the world during time.

Let us now deep dive into the different types of learning that we can define for learning, so that we can provide a more concrete characterization and less abstract formulation of learning:

- In **supervised learning**, the agent observes some input-output pairs $(\mathbf{x}, f(\mathbf{x})) \in \mathcal{L}_e \times \mathcal{Y}$ where $f : \mathcal{L}_e \to \mathcal{Y}$ is unknown, and tries to approximate such function as a hypothesis or decision function h approximating f. By comparing this definition with the one provided in the former tutorial, that is the proper "learning" as intended in [11].
- In **unsupervised learning**, the agent observes some regularities in $\mathbf{x}$ from which he tries to formulate hypotheses h without using the expected classification $f(\mathbf{x})$ provided by the dataset. Topological Data Analysis[6] provides a good learning example that are insensitive to the particular distance or similarity metric of choice, and so no specific input parameters are explicitly provided by the user.
- In **reinforcement learning**, the agent learns from a set of rewards or punishments, without necessarily knowing which is the right decision outcome. Evolutionary algorithms such as genetic algorithms are an example of reinforcement learning algorithms [10], where the generation that is rewarded to evolve as following generations are the ones minimizing the loss function.

In this tutorial we're going to use supervised learning approaches because they contain the maximum amount of information that you need to know for understand the remaining approaches.

[5] `https://en.wikipedia.org/wiki/Markov_logic_network`.
[6] `https://en.wikipedia.org/wiki/Topological_data_analysis`

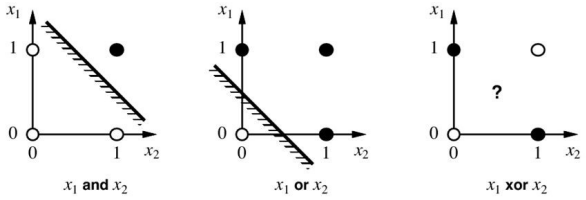


Figure 3: Black (white) dots represents the positive (negative) examples for learning specific binary functions that can still represented as values in $\mathbb{R}$. While for conjunction and disjunction there exists a line separating all the positive examples from all the negative ones, for the xor function we cannot learn a function without evident misclassification errors [13].

2.1 The eXclusive OR (XOR) problem

From the scatterplot matrix, we have also learned that our use case problem is not linearly separable. So, we want to *a priori* exclude all the possible techniques that might involve only a linear data correlation such as linear regression[7] and, if possible, to investigate whether current literature has already provided workarounds for solving complex problems by generalizing linearly binary classifiers. Albeit several real world problems are non linear separable, we're going to focus on one simple and self evident function, which is the binary *exclusive disjunction*, or XOR. Figure 3 provides an example of such a function: a good machine learning algorithm should be able to learn a XOR function by increasing the degree of complexity of the model, thus generalizing a simplistic initial configuration. We can formalize this problem as follows:

Definition 1 (Linearly Separable Problem)**.** *Two set of points $E_0, E_1 \subseteq \mathbb{R}^n$ in $(\mathbb{R}^n, \leq)$ are **linearly separable** if and only if there exist $n+1$ real numbers in $\vec{w} \in \mathbb{R}^n$ and $\theta \in \mathbb{R}$ such that the following condition is met: $\left(\forall \vec{x} \in E_0, \sum_{0 \leq i \leq n} W_i \vec{x}_i > \theta\right) \wedge \left(\forall x \in E_1, \sum_{0 \leq i \leq n} W_i \vec{x}_i < \theta\right)$.*

2.2 Overfitting

The simplistic approach introduced in the last tutorial can be applied when we want to train our model over well-known functions f over finite set of examples[8]: in this situation, we can generate all the possible positive and negative examples as $E = \{ (\mathbf{x}, f(\mathbf{x})) \mid \mathbf{x} \in \mathcal{L}_e \}$ and then try to generate an hypothesis $h = ML(loss, E, \mathcal{L}_h)$. By definition of loss function, this approach will always provide the best approximation of a function f.

On the other hand, when either f is unknown or $\mathcal{L}_e$ is not necessarily finite, this approach won't necessarily provide an hypotheses meeting the expectations, as it might only describe a subset E of all the possible situations that might occur in the real world. In order to mimic this problem and therefore test the predictive power of our hypotheses h, we need to partition E into two randomly sampled sets, the **train** (T_r) and **test** (T_e) sets, such that $T_r \cup T_e = E$ and $T_r \cap T_e = \emptyset$. Then, we train our model over T_r (i.e., $h = ML(loss, T_r, \mathcal{L}_e)$) and test it via T_e (i.e., $loss(h, T_e)$): we say that h doesn't **overfit**[9] over T_r if h returns a "small" value of $loss(h, T_e)$. We can now observe that the function minimizing the loss over T_r is not necessarily the expected function providing a best approximation of f as $loss(h, T_r) \leq loss(h, E)$ for $T_e \subseteq E$. In fact, for the mean squares loss function we might observe that the following condition follows by construction of our training and test sets:

$$loss^{\delta_{\mathcal{Y}}}_{\text{lms}}(h, E) = loss^{\delta_{\mathcal{Y}}}_{\text{lms}}(h, T_r) + loss^{\delta_{\mathcal{Y}}}_{\text{lms}}(h, T_e)$$

One possible approach to avoid overfitting problem is to further subdivide T_r in further training and testing subsets (see Exercise 2). We won't investigate such approaches in these tutorials, as we're going to limit our analysis to either train a known function f or to simply split our dataset into train and test set.

[7] `https://en.wikipedia.org/wiki/Linear_regression`

[8] I.e., $|\mathcal{L}_e| \in \mathbb{N}$, where $\mathbb{N}$ is the set of the natural numbers.

[9] Please observe that the notion of generalization provided in [13] is actually the dual definition of overfitting. Given that in the previous tutorial we adopted the notion of generality [11] referring to the notion of logical implication over a fixed dataset D, in these tutorials we're always going to refer to generalization as in [13] as *non-overfitting*.

We can now try to encode the split between training set and testing set with the `DLib_split` structs: we want to remember the schema of our dataset (`colnames`) implying a given number of dimensions `dimension`. Then, a function `generateSplit` (at page 11) initializes an object belonging to such class by randomly splitting the set E into a train and a test set. The labels $f(\mathbf{x}) \in \{1, \ldots, n\} = \mathcal{Y}$ for the train and test set data $(\mathbf{x}, f(\mathbf{x})) \in T_r$ will be represented both as single numbers (`training_labels` and `testing_labels`) and as 1-dimension vectors (`training_label_vector` and `testing_label_vector`) for different kind of algorithms accepting different data representations. Similarly, the input data for the train (test) set will be stored as "`dimension`"-dimensional vectors in `training_input` (`testing_input`).

```
#include <vector>
#include <dlib/matrix.h>

struct DLib_Splits {
    size_t dimension;                   ///< How many dimensions are provided in this numeric data
    std::vector<std::string> colnames;  ///< The properties' names associated to each dimension

    /*
     * The classes/labels are here represented as doubles, but the actual class value is discrete.
     */
    std::vector<double> training_labels, testing_labels;

    /*
     * Representing the input as columns of ‘dimension‘ dimensions
     */
    std::vector<dlib::matrix<double>> training_input, testing_input,

    /*
     * Each of the n labels is now represented as an (n+1)-dimensional vector, where the first
         ↪ dimension is never used. In particular, the i-th label will be represented by an unit
         ↪ vector having the (i+1)-th component set to 1. By doing so, we can interpret the vector
         ↪ returned by a classifier as follows: the element will belong to the i-th class if the (i
         ↪ +1)-th component of the vector will be a
     */
    training_label_vector, testing_label_vector;
};
```

Each dataset record will be described by a `StarcraftReplayDataset` object, where all the records X_i and Y in our schema are going to be represented as distinct record fields. We also delegate C++ to infer the default constructor, the copy constructor and the assignment operator from the configuration of `StarcraftReplayDataset`. After providing a simple debugging output stream operator (Line 13), we define a `wasReadingSuccessful` method for initializing a record from a CSV file, and a `read` static method for returning a new `StacraftReplayDataset` record if and only if we haven't reached the end of the file. For parsing a CSV file and read only its relevant columns, we're going to use a fork of the Fast C++ CSV Parser library[10] that is also compatible with MSVC.

```
struct StarcraftReplayDataset{
    unsigned long GameId, MaxTimeStamp;
    unsigned short league_index, age;
    unsigned int hours_per_week, total_hours;
    double APM, SelectByHotkeys, AssignToHotkeys, UniqueHotkeys, MinimapAttacks, MinimapRightClicks,
        ↪ NumberOfPACs, GapBetweenPACs, ActionLatency, ActionsInPAC, TotalMapExplored, WorkersMade,
        ↪ UniqueUnitsMade, ComplexUnitsMade, ComplexAbilityUsed;

    StarcraftReplayDataset() = default;
    StarcraftReplayDataset(const StarcraftReplayDataset& copy ) = default;
    StarcraftReplayDataset& operator=(const StarcraftReplayDataset& rhs) = default;

    static std::optional<StarcraftReplayDataset> read(io::CSVReader<21>& lineReader);
    bool wasReadingSuccessful(io::CSVReader<21>& lineReader);
    friend std::ostream &operator<<(std::ostream &os, const StarcraftReplayDataset &dataset);
};
```

The first method simply associates each attribute value from the file to a specific record field, and then tests whether the reading was successful. If it was successful, some sanity checks are performed over the data for ensuring that the values are within the expected interval range. The method provides the outcome of such test.

[10] `https://github.com/ben-strasser/fast-cpp-csv-parser`

```
bool StarcraftReplayDataset::wasReadingSuccessful(io::CSVReader<21> &lineReader) {
    bool test = lineReader.read_row(GameId, league_index, age, hours_per_week, total_hours, APM,
        ↪ SelectByHotkeys, AssignToHotkeys, UniqueHotkeys, MinimapAttacks, MinimapRightClicks,
        ↪ NumberOfPACs, GapBetweenPACs, ActionLatency, ActionsInPAC, TotalMapExplored, WorkersMade,
        ↪ UniqueUnitsMade, ComplexUnitsMade, ComplexAbilityUsed, MaxTimeStamp);
    if (!test) return test;
    assert(league_index <= 8);
    assert(APM <= 390);
    assert(SelectByHotkeys <= 1.0);
    assert(AssignToHotkeys <= 1.0);
    assert(UniqueHotkeys <= 1.0);
    assert(MinimapAttacks <= 1.0);
    assert(MinimapRightClicks <= 1.0);
    assert(NumberOfPACs <= 1.0);
    assert(GapBetweenPACs <= 238.0);
    assert(ActionLatency <= 177.0);
    assert (age <= 44.0);
    assert(hours_per_week <= 169.0);
    assert(total_hours <= 1.0e+06);
    assert(ActionsInPAC <= 186.0);
    assert(TotalMapExplored <= 1.0);
    assert(WorkersMade <= 1.0);
    assert(UniqueUnitsMade <= 1.0);
    assert(ComplexUnitsMade <= 1.0);
    assert(ComplexAbilityUsed <= 1.0);
    assert(MaxTimeStamp <= 389000.0);
    return test;
}
```

The second method uses C++17 optionals for differentiate between failed (e.g., EOF reached or parsing errors) and successful read operations: the former won't contain a record, while the latter will contain the record that was parsed from the data.

```
std::optional <StarcraftReplayDataset> StarcraftReplayDataset::read(io::CSVReader<21> &lineReader) {
    struct StarcraftReplayDataset toReturn;
    if (toReturn.wasReadingSuccessful(lineReader)) {
        return {toReturn};
    } else {
        return {};
    }
}
```

Now, we would like to store `StarcraftReplayDataset` records in an `unordered_set` so to remove potential duplications. This is an appropriate precaution as diplicates might produced a model more biased towards the most represented class of candidates. A straightforward way to remove duplicates is to use sets to collect data instead of vectors. Given that such struct represents a custom data structure, we need to define explicit equality and hashing functions: in fact, C++ `unordered_sets` are represented as *chained hash tables*[8], and therefore we need a hashing function ξ[11] mapping each record to a given key in $\mathbb{N}$, and an equality predicate for distinguishing different records using the same hashing function. Given that our record contains fields that are all native types and given that C++ already encodes hashing functions for the values associated to such types via `std::hash<T>`, we can use variadic function templates[12] to express the following hashing function over all the relevant fields of the record.

$$\xi(\texttt{hd::tl}) = \begin{cases} \texttt{std::hash(hd)} & \texttt{tl} = \emptyset \\ \texttt{std::hash(hd)} + 31 \cdot \xi(\texttt{tl}) & \text{oth.} \end{cases}$$

Such recursive function can now implemented in C++11 using the following syntax:

```
size_t multihash() { return 0; }
template <typename T> size_t multihash(T var1) { return std::hash<T>()(var1); }
template <typename T, typename... Types> size_t multihash(T var1, Types... var2) {
    return 31 * multihash(var2...) + std::hash<T>()(var1);
}
```

[11] Usually, the hashing function is defined by h. Due to a symbol clash with the hypothesis h from the former tutorial, we prefer to use ξ for disambiguation purposes.

[12] Variadic function templates are functions using variadic templates for ensuring type safety and simulating iteration via function recursive calls that, at compile time, are going to be rewritten as one single expression. See `https://en.wikipedia.org/wiki/Variadic_template` for more examples.

In order to implement the equality predicate, we can generate a tuple from the relevant attributes of choice, and then use the equality `operator()==` that is already defined for tuples. In particular, tuples are a fixed size collection of heterogeneous values generalizing `std::pair` for multiple possible dimensions. Tuples can be constructed via the `tie` function calling the `std::tuple` constructor defined via variadic templates.

Let us suppose that we now want to store the whole StarCraft II Replay dataset and represent each record with only the best 9 predictors. In order to do so, we can both define a hashing and equality predicate only focusing on 9 record field, and represent each input record as a 9-dimensional vector. In order to do so, we need to define the following struct:

```
#include <ai/datasets/StarcraftReplayDataset.h>
#include <utils/numeric/hashing.h>
#include <dlib/matrix.h>

/**
 * Delegating the different equality and hashing functions for the StarcraftReplayDataset struct to
     ↪ different structs. This struct will provide the subdivision of the dataset into 9 dimensions,
     ↪  and will make it fit inside matrices.
 */
struct dataset_9_dimensions {
    static constexpr size_t dimensions = 9;

    /**
     * Implements the equality predicate
     * @param str1 Left hand side
     * @param str2 Right hand side
     * @return     Comparison output
     */
    bool operator()(const StarcraftReplayDataset & str1, const StarcraftReplayDataset & str2) const;

    /**
     * Hashing function for the projected dataset
     * @param str  The data point to be hashed
     * @return     The associated hashing value
     */
    size_t operator()(const StarcraftReplayDataset & str) const;

    /**
     * Initializes some components that will be part of the DLib_Split class (1)
     * @param dataset     Input vectorial representation
     * @param classes     Single value class representation (expected outcome)
     * @param x           Data point from which extract the representation
     * @param normalize   Whether the values should be normalized (e.g., for Multilayer Perceptron
         ↪ Networks, and not necessairly for SVMs)
     */
    void fit_sample(std::vector<dlib::matrix<double>> &dataset, std::vector<double> &classes, const
        ↪ StarcraftReplayDataset &x, bool normalize = true);

    /**
     * Initializes some components that will be part of the DLib_Split class (2)
     * @param dataset     Vectorial representation of the expected output of the classification outcome
         ↪  as a vector
     * @param x           Data point from which extract the representation
     * @param max_classes How many classes are we expect to provide as an output
     */
    void fit_output(std::vector<dlib::matrix<double>>& dataset, const StarcraftReplayDataset& x, const
        ↪  size_t max_classes = 8);

    /**
     * Returns the dimensions that this projection will represent
     * @param schema_name
     */
    void set_label_names(std::vector<std::string>& schema_name);
};
```

The complete implementation of such structure is provided at `https://github.com/jackbergus/NotesOnProgramming2020/blob/master/src/ai/lib/datasets/Starcraft9Dimensions.cpp`. In the same folder, we also provide a dummy implementation for preserving all the attributes originally provided by the StarCraft II Replay dataset. Now, we can implement the first and second operator

respectively representing the equality predicate and the hashing function:

```
#include "ai/datasets/Starcraft9Dimensions.h"

bool dataset_9_dimensions::operator()(const StarcraftReplayDataset &str1, const StarcraftReplayDataset
    ↪ &str2) const {
    return std::tie(str1.APM, str1.SelectByHotkeys, str1.AssignToHotkeys, str1.UniqueHotkeys, str1.
        ↪ MinimapAttacks, str1.MinimapRightClicks, str1.NumberOfPACs, str1.GapBetweenPACs, str1.
        ↪ ActionLatency) ==
           std::tie(str2.APM, str2.SelectByHotkeys, str2.AssignToHotkeys, str2.UniqueHotkeys, str2.
               ↪ MinimapAttacks, str2.MinimapRightClicks, str2.NumberOfPACs, str2.GapBetweenPACs,
               ↪ str2.ActionLatency);
}

size_t dataset_9_dimensions::operator()(const StarcraftReplayDataset &str) const {
    return multihash(str.APM, str.SelectByHotkeys, str.AssignToHotkeys, str.UniqueHotkeys,
                     str.MinimapAttacks, str.MinimapRightClicks, str.NumberOfPACs, str.GapBetweenPACs,
                     str.ActionLatency);
}
```

Now, we can provide a preliminar split of our dataset by defining the following record splitting an `originalSet` E into a training and a testing set. `percentage=0.7` is used to determine the size of the training set with respect of the whole record set size. The remaining elements will be part of the test set.

```
/**
 * Utility struct providing a splitting between training and test data set from a set of unique
     ↪ elements.
 *
 * @tparam T          Type of the record that needs to be sorted
 * @tparam Hash       Hashing function and equality operator associated to the record
 */
template <typename T, typename HashEq>
struct training_testing_sets {
    std::unordered_set<T, HashEq, HashEq> training, testing;

    training_testing_sets(std::unordered_set<T, HashEq, HashEq>& originalSet, const double percentage
        ↪ = 0.7) {
        double i = 0, N = originalSet.size();
        for (auto it = originalSet.begin(); it != originalSet.cend(); it++) {
            (((i)/N < percentage) ? training : testing).emplace(*it);
            i += 1;
        }
    }

    friend std::ostream &operator<<(std::ostream &os, const training_testing_sets &sets) {
        double Tr = sets.training.size();
        double Ts = sets.testing.size();
        os << "training:␣" << (Tr)/(Tr+Ts) << "␣testing:␣" << (Ts)/(Tr+Ts);
        return os;
    }
};
```

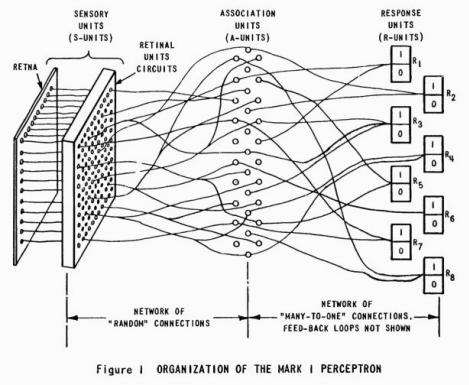
The `generateSplit` template function following source code splits the dataset in train and test dataset for each class, and then permutes the order of appearance of each data sample for each class (Line 14). After specifying that our dataset E contained in the file `datasetName` will contain 21 attributes (Line 5) which names will be later on specified in Line 7, we create an instance of our specific hashing and equality operator, that is also going to be used to fit our data into a `DLib_Splits` record (Line 6). Then, we're going to read data from the CSV file (Line 21) until we reach the end of the file (Line 25). The set of record associated to one single class is then splitted via training test and test set via `training_testing_sets` record (Line 33). Then, the training and testing set of each class is converted to the vectorial representation that is compatible with the DLib[13] library that we are going ot use for our Machine Learning algorithms. Values are possibly normalized within the interval $[0, 1] \subseteq \mathbb{R}$ if required by a specific Machine Learning algorithm (`normalize`). Last, the training and testing set data are randomly shuffled using a random permutation generated[14] at Line 57.

[13] Official Website: `dlib.net`, Online Repository: `https://github.com/davisking/dlib/`.

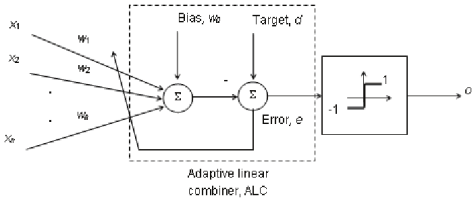
[14] The code for generating a permutation of a sequence of n values is provided in `https://github.com/jackbergus/NotesOnProgramming2020/blob/master/src/utils/permutation.cpp`, while the function that is taking the permuta-

```cpp
template <typename HashEq>
std::pair<const size_t, const size_t>
generateSplit(const std::string &datasetName, DLib_Splits &out, bool normalize = true, const double trainingSize = 0.7) {
    // Reading the StarCraft dataset
    io::CSVReader<21> starcraft{datasetName};
    HashEq dlibMover;
    starcraft.read_header(io::ignore_extra_column, "GameID", "LeagueIndex", "Age", "HoursPerWeek", "TotalHours", "APM", "SelectByHotkeys", "AssignToHotkeys", "UniqueHotkeys", "MinimapAttacks", "MinimapRightClicks", "NumberOfPACs", "GapBetweenPACs", "ActionLatency", "ActionsInPAC", "TotalMapExplored", "WorkersMade", "UniqueUnitsMade", "ComplexUnitsMade", "ComplexAbilityUsed", "MaxTimeStamp");

    // Used for the table name information
    dlibMover.set_label_names(out.colnames);

    // This final map will contain both the training and the testing dataset
    // We will split the dataset into training and testing for each of the classes
    std::unordered_map<unsigned short, training_testing_sets<StarcraftReplayDataset, HashEq>> splitting_map;
    {
        // Reading all the datasets: using the custom hashing and equality functions, so that we automatically project the data as we wish
        std::cout << "Reading the CSV file" << std::endl;
        std::unordered_map<unsigned short, std::unordered_set<StarcraftReplayDataset, HashEq, HashEq>> map;
        bool continueReading = true;
        do {
            auto x = StarcraftReplayDataset::read(starcraft);
            if (x) {                                            // If it was possible to read the information from the file
                map[x->league_index].emplace(x.value());  // Insert it inside the map
            } else {
                continueReading = false;                        // Otherwise, fail and stop reading
            }
        } while (continueReading);

        // Defining the split size for the training/testing dataset. Inserting those after removing the duplicates.
        std::cout << "Splitting everything in human-readable training and testing sets" << std::endl;
        for (auto it = map.begin(); it != map.cend(); it++) {
            // Splitting the previously loaded set of data into the training and the testing dataset
            splitting_map.insert(std::make_pair(it->first, training_testing_sets<StarcraftReplayDataset, HashEq>{it->second, trainingSize}));
        }
    }

    // Preparing the loaded data in the format that DLib expects.
    // Please note that this multiple-step implementation is just to be debuggable and to be sure what we're doing, but everything can be done more efficiently in one single step.
    std::cout << "Copying the information for a DLib-compatible representation" << std::endl;
    for (auto it = splitting_map.begin(); it != splitting_map.cend(); it++) {
        const training_testing_sets<StarcraftReplayDataset, HashEq>& splits = it->second;

        for (const StarcraftReplayDataset& trainingDatum : splits.training) {
            dlibMover.fit_sample(out.training_input, out.training_labels, trainingDatum, normalize);
            dlibMover.fit_output(out.training_label_vector, trainingDatum, splitting_map.size());
        }

        for (const StarcraftReplayDataset& testingDatum : splits.testing) {
            dlibMover.fit_sample(out.testing_input, out.testing_labels, testingDatum, normalize);
            dlibMover.fit_output(out.testing_label_vector, testingDatum, splitting_map.size());
        }
        std::cout << it->first << "~" << it->second << std::endl;
    }

    // Shuffling the data
    std::cout << "Performing a random data permutation" << std::endl;
    std::vector<int> permutation = generateRandomPermutation(out.training_input.size());
    out.training_input = permute(out.training_input, permutation);
```

tion vector to permute another vector is provided in https://github.com/jackbergus/NotesOnProgramming2020/blob/master/include/utils/numeric/permutation.h.



(a) The Perceptron.



(b) ADALINE Neuron.

Figure 4: First attempts defining neruons in Artificial Intelligence.

```
59      out.training_labels = permute(out.training_labels, permutation);
60      out.training_label_vector = permute(out.training_label_vector, permutation);
61
62      // <number of classes, number of dimensions == mlp input>
63      std::cout << "Dataset ordering done" << std::endl;
64      out.dimension = dlibMover.dimensions;
65      return {splitting_map.size(), dlibMover.dimensions};
66  }
```

3 Multilayer Neural Networks

In this section we're going to discuss traditional Multilayer Neural Networks instead of modern deep learning models. The motivation for doing so is twofold: first, major artificial intelligence books such as [13] always provide this model as an introduction to neural networks; second, modern "deep" neural networks overcome to some of the Multilayer Neural Network limitations and, to understand those modern implementations, we need to first learn a simpler model. Second, some "deep" neural network models also require additional knowledge to be understood theoretically, such as the definition of kernel functions, how those could help in learning non-linear problems and simplify the backpropagation algorithm here presented.

3.1 ADaptive LInear NEuron

The main element of a neural network is a neuron, conceived initially as a **perceptron** by Rosenblatt in 1959. The perceptron (Figure 4a) was, however, much more complicated than a neuron as currently conceived: it consisted of a complex retina comprised of a network of n input elements, which in turn were linked to an association layer of m elements, which then were connected via a Φ_i function to the neuron's main sum function Ω. The following year, Widrow and Hoff defined the ADaptive LInear NEuron, which is a specific case of a neuron (Figure 4b). In particular, each neuron performs the weighted sum of all the input signals $\vec{x}$ with a set of weights $\vec{w}$. Such value must reach a threshold value θ' over which the neuron will be activated and will send a signal $\simeq 1$. Therefore, the resulting signal procedding can be modelled as follows:

$$\text{net} = \vec{w}^t\vec{x} - \theta'$$

Such signal is then sent to the activation function φ, which then rectifies the output signal.

Definition 2 (Neuron)**.** *A neuron is a function defined as follows:*

$$f_{\vec{w},\theta}(\vec{x}) = \varphi(\vec{w}^t\vec{x} + \theta)$$

where $\vec{x}$ represents the input, $\vec{w}$ represents the weight associated to each input, $\theta \in \mathbb{R}$ is a bias factor and φ is the activation function distinguishing two classes $\{-1, 1\} = \mathcal{Y}$.

As we can see, ADALINEs are specific neurons with $\varphi = \text{sign}$ and $\theta = -\theta'$. As per Definition 1 on page 6, we can observe that these kind of neurons can only solve linearly separable problems. Moreover, changing φ to the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ doesn't solve the problem. As we might observe more empirically after the definition of the *backpropagation* algorithm, we can potentially solve non-linear problem only by combining different distinct neurons together.

A network of multiple ADALINEs connected together was called MADALINE for MULTIPLE ADALINE: such networks were **feed-forward networks**, as the information (only) moved from the input towards the output layer, and there was no way to "backpropagate" the error signal from the loss function evaluated from the output back to the input layer. As a consequence, MADALINES couldn't be easily trained.

3.2 Multilayer Neural Networks with Backpropagation

In 1986, a paper by Rumelhart, Hinton and Williams finally provided a solution for training multiple neuron networks for non-linear problems by providing the following novelties:

1. The paper introduced **hidden layers**: they introduced the concept of a layer, containing several distinct neurons. Multiple interconnected layers can now connect the input to the output layer: if we define $W^L = (w^L_{jk})$ the matrix of all the weights of the inputs coming from layer L from layer $L-1$ where w^L_{jk} is the weights between the k-th node from layer $L-1$ sending his signal to the j-th neuron of layer L, then we can express the overall network via function composition as follows:
$$\tilde{h}_{W^1,\dots,W^L}(x) = f^L(W^L f^{L-1}(W^{L-1} \dots f^1(W^1 x) \dots))$$
When $W^1, \dots, W^L$ are clear from the context, we might represent such function simply as $\tilde{h}$. As the adjective *Deep* usually refers to a huge number of hidden layers, we can say that Deep Neural Networks were already known in 1986!

2. Also, the paper provided the **backpropagation** algorithm for propagating the loss value backwards from the output layer towards the input, thus allowing the weight update in each neuron. As an intuition, the backpropagation algorithm aims to minimize the loss function. In order to make convenient mathematical operations that will follow, the following loss function was defined:
$$\mathcal{L} := loss(\tilde{h}, E) = \frac{1}{2} \sum_{(\mathbf{x},y)\in E} (\tilde{h}(\mathbf{x}) - y)^2$$
As you might remember from mathematical analysis, a good way to find the minimum points of a function is to evaluate the first derivative of $\mathcal{L}$: given that we are interested to minimize the loss while updating the weights of a neuron, we need also to do the partial derivative of the loss function over each possible weight w_{ij} of the j-th neuron connected to the i-th neuron (or input) from the former layer. To update the weight of w_{ij} using the gradient descent, we will have that if $\frac{\partial \mathcal{L}}{\partial w_{ij}} < 0$ an increase of w_{ij} will make the loss function decrease. To guarantee that, we multiply the gradient by -1; moreover, we can define a non-negative **learning rate** $\eta > 0$ that defines the convergence rate, that is the speed at which we're going to reach the local minimum. Please note that a too big value η could make the minimization function skip the minimum completely.

As you might experience from training single neurons from the source code released at `https://github.com/jackbergus/NotesOnProgramming2020/blob/master/src/ai/lib/mlp/SinglePerceptron.cpp`, single neurons can learn the disjunction or the conjunction, but cannot learn the exclusive or. In order to effectively learn the function, we are forced to combine three linear classifiers together with the network described in Figure 5. Before analysing how to train Multilayer Neural Networks in DLib, let us analyse a custom implementation of such neural networks. Each neuron can be defined by the following source code:

```
/**
 * The perceptron is the atomic component of a neural network
 */
struct Perceptron {
    /**
```

```
 * Weights associated to the input signal. These weights are the one used also for the momentum
     ↪ equation, and therefore
 * They remember also the weight in the previous computation step
 */
std::vector<weight> weights;

/**
 * Weight associated to the theta threshold: this acts as a dummy input having always a signal of
     ↪ 1, but which weight can be changed.
 */
weight theta;

/**
 * Gradient value associated to the dummy/theta imput
 */
double theta_gradient;

/**
 * Copy of the input values: these will be useful once we have to compute the gradient.
 */
std::vector<double> input_values;

/**
 * Gradients associated to the current neuron
 */
std::vector<double> input_gradients;
size_t neuronIdInLayer;

/**
 * Value returned by the sigmoid function.
 */
double out;
/**
 * Weighted sum provided to the sigmoid function
 */
double net;
/**
 * If this neuron belongs to the outer layer, this will store the difference between the output
     ↪ value and the expected one. This will be then used for the derivative calculation
 */
double error;

double derivative;

/**
 * Utility printing function
 * @param os
 * @param perceptron
 * @return
 */
friend std::ostream &operator<<(std::ostream &os, const Perceptron &perceptron);

/**
 * Using the default random initialization of the weights;
 */
Perceptron();

/**
 * Randomly setting up a neuron using the default random initialization of the weights, but using
     ↪ a specific neuron id, that will be the neuron id within the layer
 *
 * @param neuronIdInLayer
 */
Perceptron(size_t neuronIdInLayer, size_t N);

/**
 * Importing the weights and the theta from a previously-saved configuration
 *
 * @param weights
 * @param theta
 */
Perceptron(size_t neuronIdInLayer, std::vector<weight> weights, const weight &theta);
```

```
    Perceptron(size_t neuronIdInLayer, std::vector<double> weights, const double theta);

    /**
     *
     * @param id      Number associated to the current neuron
     * @param N       Number of the input size, that is the number of the neurons in the previous
          ↪ layer or the length of the input
     * @param re      Random number generator
     */
    Perceptron(size_t id, size_t N, std::default_random_engine& re);

    /**
     * Computing the function of the neuron
     *
     * @param input Input coming from the outside, that will be also saved in the inside.
     *
     * @return computed value
     */
    double compute(std::vector<double>& input);

    /**
     * Calculates the discrepancy between the computed value and the expected value. This function
          ↪ shall be called only
     * over output layer networks
     *
     * @param expectedValue
     * @return Difference
     */
    double calculateError(double expectedValue);

    /**
     * Gradient calculation for the other layer networks
     *
     * @param deltasFromForward
     * @return
     */
    std::vector<double> calculateDerivative(double deltasFromForward);

    /**
     * Gradient calculation for the output layer neurons
     *
     * @param expectedValue
     * @return
     */
    std::vector<double> calculateDerivativeFromExpected(double expectedValue);

    void updateGradient();

    /**
     * Updates the weights of the current perceptron using the previous computation of the gradient.
     *
     * @param learningRate
     * @param momentum
     */
    void updateWeight(double learningRate = 1.414213562, double momentum = 0.25);

    /**
     * Resets the neuron's weights using the standard C way to generate random numbers
     */
    void reset();

    /**
     * Resetting the neuron's weights using a default random number generator
     * @param re   Random number generator
     */
    void reset(std::default_random_engine &re);
};
```

For some reasons that will be clearer after we deal with the backpropagation algorithm, each weight is defined as a pair containing both the current weight and the one obtained in the previous step of the backpropagation algorithm:

```
/**
```

```
 * A weight structure contains both the current weight value and the one obtained in the previous
     ↪ training step
 */
struct weight {
    double previous;
    double current;

    /**
     * Using a random engine to initialize the value
     * @param re   random engine
     */
    weight(std::default_random_engine& re);

    /**
     * Using a default value to initialize the weight
     * @param val
     */
    weight(double val);

    /**
     * Using the system rand to initialize the weight
     */
    weight();

    /**
     * Resetting the weight using the C default random number generator
     */
    void reset();

    /**
     * Resetting the weight using a C++11 random number generator
     * @param re
     */
    void reset(std::default_random_engine &re);

    friend std::ostream &operator<<(std::ostream &os, const weight &weight);
};
```

The simple implementation of this `struct` is defined as follows:

```
weight::weight(std::default_random_engine &re) {
    reset(re);
}

weight::weight(double val) {
    current = val;
    previous = 0.0; // Previously, there was no increment.
}

weight::weight() {
    reset();
}

void weight::reset() {
    current = (((rand() % 2) == 1) ?
                        (-1 * (double(rand()) / (double(RAND_MAX) + 1.0))) :
                        ( double(rand()) / (double(RAND_MAX) + 1.0)));
    previous = 0.0; // Previously, there was no increment.
}

void weight::reset(std::default_random_engine &re) {
    std::uniform_real_distribution<double> unif(-1, 1);
    current = unif(re);
    previous = 0.0; // Previously, there was no increment.
}

std::ostream &operator<<(std::ostream &os, const weight &weight) {
    os << "[" << weight.previous << "," << weight.current << "]";
    return os;
}
```

At this point, the function computed by a single neuron matching Definition 2 with $\varphi := \sigma$ can be computed as follows:

```
double Perceptron::compute(std::vector<double> &input) {
    size_t N = std::min(input.size(), weights.size());
    input_values = input; // For backpropagation purposes, save the values provided as input

    // Perform the weighted sum of all the inputs with their associated weight values
    net = 0.0;
    for (size_t i = 0; i<N; i++) {
        net += input[i] * weights[i].current;
    }
    // Add the bias as a different
    net += theta.current;

    // Apply the activation sigmoid function, store a copy of the result, and then return it
    return (out = SIGMOID(net));
}
```

We will go back to the other methods while discussing the gradient descent: now, each layer will be jut composed of a set of different neurons. For gradient calculation and for debugging purposes, each layer will store the value that has forward propagated to the next layer, or possibly provided as an output.

```
#include <ai/mlp/Perceptron.h>

/**
 * A layer is just an array of neurons
 */
struct Layer {
    std::vector<Perceptron> perceptrons; // neurons in the layer
    std::vector<double> currentOutput; // output provided by each neuron in the current layer for a
        ↪ given input

    friend std::ostream &operator<<(std::ostream &os, const Layer &layer);

    /**
     * Creating N neurons in the layer with some C++11-like randomly generated weights (via a random
         ↪ number generator)
     * @param previousLayerSize      Number of perceptrons in the previous layer/input
     * @param N                      Number of perceptrons in the current layer
     * @param re                     Random number generator
     */
    Layer(size_t previousLayerSize, size_t N, std::default_random_engine& re);

    /**
     * Creating N neurons in the layer with some C-like randomly generated weights
     * @param previousLayerSize
     * @param N
     */
    Layer(size_t previousLayerSize, size_t N) {
        for (size_t j = 0; j < N; j++) {
            perceptrons.emplace_back(j, previousLayerSize);
        }
    }

    /**
     * Creating an empty layer
     */
    Layer();

    /**
     * Adding a perceptron to the layer
     * @param weights
     * @param theta
     * @return
     */
    Layer& addPerceptron(std::vector<double> weights, double theta);

    std::vector<double> compute(std::vector<double>& input);

    void reset();

    void reset(std::default_random_engine& re);
```

```
    double calculateQuadraticError(std::vector<double>& expectedOuput);

    /**
     * Gradient calculation for the neurons (either in the output layer or in the input layer)
     *
     * @param expectedValue
     * @param FromExpectedValue  If the value is set to true, then @expectedValue is the expected
         ↪ value from the output.
     *                           Otherwise, that vector will contain the delta values to be
         ↪ backpropagated within the network
     * @return                  The value to be backpropagated from the network
     */
    std::vector<double> calculateDerivative(std::vector<double>& expectedValue, bool FromExpectedValue
        ↪ );

    /**
     * Recursively updating the gradient for each neuron in the layer
     */
    void updateGradient();

    /**
     * Recursively updating the gradient for each neuron in the layer
     * @param learningRate
     * @param momentum
     */
    void updateWeight(double learningRate = 1.414213562, double momentum = 0.25);
};
```

We can let the layer have the responsibility of orchestrating all the actions that need to be to be performed over all its neurons (e.g., `compute`, `reset`, `updateGradient`). Let us just focus on how the computation is propagated from one layer to the other: we store the real value returned by each neuron in the current layer in a result vector, that is then going to be copied in `currentOutput`. This output is also the one going to be forward propagated to the following layer.

```
std::vector<double> Layer::compute(std::vector<double> &input) {
    std::vector<double> result;
    for (Perceptron& p : perceptrons)
        result.emplace_back(p.compute(input));
    currentOutput = result;
    return currentOutput;
}
```

In fact, a network is just a function $\tilde{h}\colon [0,1]^{\texttt{inputSize}} \to [0,1]^{\texttt{layersSize.rbegin()}}$, composed of multiple `layers`, where each of those `i` has `layerSize[i]` neurons. A whole network can be then defined as follows:

```
struct BackwardPropagationNetwork {
    size_t inputSize;
    std::vector<size_t> layersSize;
    std::vector<Layer> layers;
    std::default_random_engine re;
    bool useRE;

    friend std::ostream &operator<<(std::ostream &os, const BackwardPropagationNetwork &network);

    /**
     * Default constructor of an empty network
     */
    BackwardPropagationNetwork();

    /**
     * Generates a new layer, that is returned by reference
     * @return
     */
    Layer& createLayer();

    /**
     * Generates a network given a network configuration
     * @param inputSize          Size of the input
     * @param layersSize         Number of neurons per hidden layer and output layer
```

```
     * @param useRE             If set to true and if the neural network is not converging after 4000
         ↪  epochs,
     *                          it will reset the neuron weights and start over again until it
         ↪ converges
     */
    BackwardPropagationNetwork(size_t inputSize, const std::vector<size_t> &layersSize, bool useRE =
        ↪ false);

    /**
     *
     * @param input             Input value
     * @return
     */
    std::vector<double> compute(std::vector<double> &input);

    void reset();

    void reset(std::default_random_engine& re);

    /**
     * Calculates the loss function
     * @param output
     * @return
     */
    double calculateQuadraticError(std::vector<double>& output);

    /**
     * Make all the neurons in the network calculate their derivatives
     * @param expectedOutput    Starts the derivatives computation from the quadratic error computed
         ↪ over the expected output
     */
    void calculateDerivatives(std::vector<double> expectedOutput);

    /**
     * Updates the gradient of all the neurons in the netwokr
     */
    void gradientUpdate();

    /**
     * Updates the weight using the backpropagation of all the neurons in the network
     * @param learningRate
     * @param momentum
     */
    void updateWeight(double learningRate = 1.414213562, double momentum = 0.25);

    /**
     * Trains the network to learn a hidden function which map is known and provided via f
     *
     * @param f                 Map of the function
     * @param iterationNumber   Maximum number of iterations that we can wait to reach convergence
     * @param learningRate
     * @param momentum
     * @return                  Loss function associated to the final configuration of the network (
         ↪ at the end of the training phase)
     */
    double train(struct finite_function& f, const size_t iterationNumber = 20000, const double
        ↪ learningRate = 1.414213562, const double momentum = 0.25);

    /**
     * Finalizes the network creation if I've been using layerCreate to create one single layer.
     */
    void finalize();
};
```

The computation of the final function $\tilde{h}$ will be then carried out by the following method. This method will chain together all the outputs `computed` by each single layer and providing such result to the next layer. When no more layers are available, we return the vector computed by the last layer as the neural network output.

```
std::vector<double> BackwardPropagationNetwork::compute(std::vector<double> &input) {

    // Input for the first layer
    std::vector<double> currentInput = input;
```

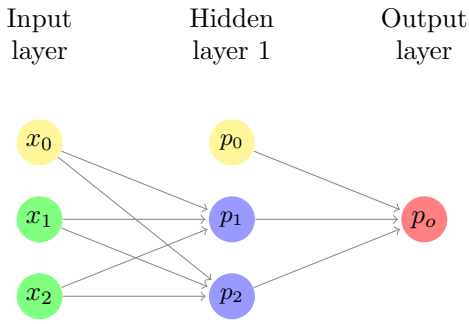


Figure 5: Representing the minimal configuration required to learn the XOR problem using neural networks. Yellow nodes (x_0, p_0) represents constant inputs $x_0 = p_0 = 1$, so that each bias θ can be modelled as a specific weight for a constant input.

```
    // Compute the intermediate output of each layer, and forward propagate that to the next layer
    for (Layer& layer : layers) {
        std::vector<double> intermediateOutput = layer.compute(currentInput);
        currentInput = intermediateOutput;
    }
    return currentInput; // Return the output provided by the last layer
}
```

After deep-diving how the network does feed-forward the input signal in each layer, we can go back and analyse how to solve the XOR problem with combinations of our nero linear classifiers (neurons with $\varphi := \varphi$). Intuitively, the minimum number of neurons to solve this problem should be three: two neurons should bound the two positive cases by selecting two different semi-planes, and one output neuron merging the two planes' decision problems into the final decision (Figure 5). The resulting classifier can be described by the following regression function:

$$\tilde{h}(\mathbf{x}) = p_0(p_1(\mathbf{x}), p_2(\mathbf{x}))$$

After some training, each neuron might assume some specific weights which will then represent the following functions:

$$p_1(x, y) = \frac{1}{1 + e^{-(7.10607 \cdot x - 6.50904 \cdot y + 3.41603)}}$$

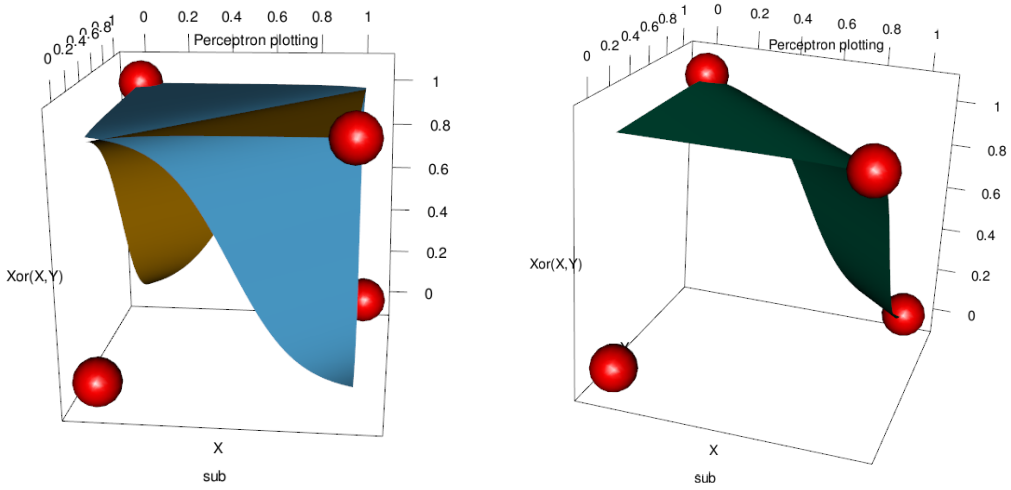
$$p_2(x, y) = \frac{1}{1 + e^{-(-8.05875 \cdot x + 8.086 \cdot y + 4.31492)}}$$

$$p_o(x, y) = \frac{1}{1 + e^{-(-8.65042 \cdot x - 8.48615 \cdot y + 12.8072)}}$$

Figure 6 plots the former functions generating the network $\tilde{h}$. With respect to the first hidden layer, the first neuron p_1 is mapping $p_1(1, 0) \approx 1$ and $p_1(0, 1) \approx 0$, while the second is doing the opposite, i.e., $p_2(1, 0) \approx 0$ and $p_2(0, 1) \approx 1$. As a result, both neurons ar acting as an identity function for positive examples, i.e. $(p_1(\mathbf{x}), p_2(\mathbf{x})) = \mathbf{x}$ if $f(\mathbf{x}) = 1$. On the other hand, both neurons map negative examples to 1, so $p_i(1, 1) = 1$ and $p_i(0, 0) = 1$ for each neuron $i \in \{1, 2\}$. The outcome of such operations is provided in Figure 6a. With respect to the output layer consisting of just neuron p_o, such neuron maps all the inputs generated from positive examples to 1 and $(1, 1)$ as geenrated by negative examples to zero. As a result, all the inputs are correctly classified.

We can summarize the interpretation of $\tilde{h}$ as follows: *all the negative examples are mapped to values near to* $(1, 1)$*, while the positive examples preserve their original information; then,* $(1, 1)$ *is predicted to be a negative example, while the inputs preserving their original representation (as inputs of the output neuron) are predicted to be positive examples.* Therefore, we can see that neural networks provide a convoluted mathematical motivation of relevant properties of the data via sigmoid transformations. Such interpretation might be very complicated to extract for more complicated networks.

In the next paragraph, we're going to provide additional details and explanations with respect to the backpropagation algorithm required to achieve the former result.



(a) Plotting the two neurons p_1 and p_2.

(b) Plotting the output neuron p_0.

Figure 6: Graphical representation of equations p_1, p_2, p_o determining $\tilde{h}$.

3.2.1 Multiclass classification

As we've just saw, multilayer neural networks containing one single output neuron provide a binary classifier. At this point, we would tend to define a multiclass classification problem by just partition the $[0, 1] \subseteq \mathbb{R}$ interval into $|\mathcal{Y}|$ distinct and continuous intervals: nevertheless, the best solution is to create as many output neurons as classes, so that the final regression function will be $\tilde{h} \colon [0, 1]^n \to [0, 1]^{|\mathcal{Y}|}$, where n is the number of the dimensions of the data. Then, for each $(\mathbf{x}, y) \in T_r$, train the network so that only the y-th neuron should be set to 1, and all the others are expected to return zero.

Finally, the decision function for our $\tilde{h}$ can be defined as follows:

$$h_{\tilde{h}}(\mathbf{x}) = \arg\max_{1 \leq i \leq |\mathcal{Y}|} \tilde{h}(\mathbf{x})_i$$

where $\tilde{h}(\mathbf{x})_i$ is the i-th component of the vector returned by the output layer. In the general setting, there could be multiple elements providing the maximum value, and therefore these values can be stored in a set (Line 3). As a consequence, we can determine the predicted desirability of our classification outcome as $\tilde{h}(\mathbf{x})_{h_{\tilde{h}}(\mathbf{x})}$ or, as suggested by the following code, compute a distance between the predicted candidate and the set of all the possible candidates (Line 13).

```
1  double class_prediction_distance(const dlib::matrix<double> &m, const unsigned long expected) {
2      double maxValue = -std::numeric_limits<double>::max();
3      std::set<unsigned long> candidates; // Returning the caniddates maximizing the \tilde{h} value
4      for (size_t i = 0, N = m.size(); i<N; i++) {
5          if (maxValue < m(i)) {       // Getting a new maximum value
6              maxValue = m(i);
7              candidates.clear();
8              candidates.emplace(i);   // Adding the new candidate to the cleared set
9          } else if (maxValue == m(i)) {
10             candidates.emplace(i);   // Adding a candidate with the same size
11         }
12     }
13     return class_prediction_distance(expected, candidates);
14 }
```

The code computing the last line of the former code is computed by the following template function:

```
template <typename T>
double class_prediction_distance(const T expected,
                                 std::set<T> &candidates) {//assert(candidates.size() <= 1);

    // Getting the average distance between all the candidates and the expected label
    double avgPredictionSimilarity = 0.0;
```

```
    // a) first, evaluate the average distance between the components
    for (auto it = candidates.begin(); it != candidates.cend(); it++) {
        // evaluate the distance between expected and predicted label
        double tmpDistance = std::abs(((double)*it) - (double)expected);
        // sum the normalized distance
        avgPredictionSimilarity += (tmpDistance / (tmpDistance + 1));
    }
    if (!candidates.empty()) avgPredictionSimilarity /= ((double)candidates.size());
    // b) then, invert it so it becomes a similarity
    avgPredictionSimilarity = 1.0 - avgPredictionSimilarity;

    if (candidates.empty())
        return 1.0;          // If no candidates were returned, then I have the maximum error
    if (candidates.find(expected) != candidates.cend()) {
        if (candidates.size() == 1)
            return 0.0;      // If the precision was maximum, then the error was minimum
        else {
            // The distance shall be weighted by the number of wrong candidates that we get (all the
                ↪ candidates minus the correclty predicted one)
            return 1.0 - ((avgPredictionSimilarity * (1.0 - ((double)candidates.size() - 1.0) / ((
                ↪ double)candidates.size()))));
        }
    } else {
        // The distance shall be weighted by the number of wrong candidates that we get (all the
            ↪ predicted candidates)
        return 1.0 - ((avgPredictionSimilarity * (1.0 - ((double)candidates.size()) / ((double)
            ↪ candidates.size() + 1.0))));
    }
}
```

Before deep-diving into the labyrinth of the backpropagation algorithm, let's see how to use DLib and the former distance function for training a network over the StarCraft II Replay dataset:

```
void mlp_train(const DLib_Splits& splits, const unsigned numberClasses, const size_t input_size) {
    // Even better, we could change the former source code to ensure that all the training and testing
        ↪  datasets for the
    // 8 classes will have the same size.
    // The mlp takes column vectors as input and gives column vectors as output. The dlib::matrix
    // object is used to represent the column vectors. So the first thing we do here is declare
    // a convenient typedef for the matrix object we will be using.
    const size_t iterationNumber = 100;

    double candidateScore = std::numeric_limits<double>::max();
    dlib::mlp::kernel_1a_c candidateNet(1, 1); // dummy net: at the end, will store the best trained
        ↪ one

    std::cout << "Training␣is␣starting..." << std::endl;
    size_t maxFirstLayer = std::min(((unsigned long)(numberClasses+1L) * 5L), (unsigned long)30L);
    std::cout << "maxFirstLayer:␣" << maxFirstLayer << std::endl;
    for (int firstLayer = maxFirstLayer; firstLayer >= 0; firstLayer--) {
        size_t maxSecondLayer = std::max(std::floor(((numberClasses+1.0) + firstLayer)/2.0), (
            ↪ numberClasses+1.0)*2.0);
        std::cout << "--␣maxSecondLayer:␣" << maxSecondLayer << std::endl;
        for (size_t secondLayer = 0; secondLayer < maxSecondLayer; secondLayer++) {

            // Create a multi-layer perceptron network, each time changing the configuration of the
                ↪ network, id est the number of neurons in the first two hidden layers: we need to
                ↪ test several different possible configurations, as we cannot directly determine
                ↪ from the data which is the best configuration for solving the problem (open
                ↪ research question!). Also, we set the number of the output neurons to |𝒴| + 1: this
                ↪ will guarantee us that one output neuron can possibly determine that a given input
                ↪ does belong to none of the classes in 𝒴 as trained from the input dataset.
            dlib::mlp::kernel_1a_c net(input_size, firstLayer, secondLayer, numberClasses+1);
            size_t epoch = 0;

            double distance;
            while (epoch < iterationNumber) {
                // Now let's put some data into our sample and train on it. We do this
                // by looping over all the training records

                distance = 0.0;
                size_t N = std::min(splits.training_input.size(), splits.training_label_vector.size());
```

```
            // Feeding the network all the inputs and outputs that were previously obtained
            for (size_t i = 0; i<N; i++) {
                net.train(splits.training_input[i], splits.training_label_vector[i]);
            }

            // Computing the average of the distance between the expectation vs. the retrieved
                ↪ candidates
            for (size_t i = 0; i<N; i++) {
                double ithDistance = 0.0;
                auto prediction = net(splits.training_input[i]);
                const auto& expected = splits.training_label_vector[i];
                ithDistance = class_prediction_distance(prediction, splits.training_labels[i]);
                distance += ithDistance;
            }
            distance /= ((double)N);

            // Increment the current epoch
            epoch = epoch + 1;
        }

        std::cout << "firstLayer:␣" << firstLayer << "␣secondLayer:␣" << secondLayer << "␣score:␣"
            ↪ << distance << std::endl;
        if (distance >= 0.67803) {
            /// Heuristic: from the previous training, it seems that this is the maximum error that
                ↪  can be achieved, and that if I reach this value, then no further configuration
                ↪ will work
            std::cout<< "break!" << std::endl;
            break;
        }
        if (distance < candidateScore) {
            std::cout << "==>␣New␣candidate!␣" << std::endl;
            candidateScore = distance;
            candidateNet.swap(net); // Putting in candidateNet the best candidate: net is going to
                ↪ be discarded after quitting this loop anyway
        }
    }
}


// Now we have trained our mlp. Let's see how well it did.
// Note that if you run this program multiple times you will get different results. This
// is because the mlp network is randomly initialized.
size_t N = std::min(splits.testing_label_vector.size(), splits.testing_input.size());
double distance = 0.0;
for (size_t i = 0; i<N; i++) {
    double ithDistance = 0.0;
    auto prediction = candidateNet(splits.testing_input[i]);
    const auto& expected = splits.testing_label_vector[i];
    ithDistance = class_prediction_distance(prediction, splits.testing_labels[i]);
    // normalize the distance
    ithDistance = ithDistance / (ithDistance+1.0);
    distance += ithDistance;
}
// distance normalization
distance = (distance) / ((double)N);
// invert the distance, so to obtain the precision
double precision = 1.0 - distance;

std::cout << "Model␣precision␣over␣the␣testing␣data:␣" << precision << std::endl;
}
```

Please observe that, unlike the XOR problem, we cannot determine in advance which the network configuration minimizing the loss function might be. Therefore, we are forced to train multiple multilayer neural networks and to keep the one minimizing the loss function.

Backpropagation Algorithm (*) Let us now continue our journey for deriving the backpropagation equation required by the neural network. We assume that each bias θ is modelled as a weight coming from a specific neuron providing a constant output 1 (See nodes in yellow in Figure 5). By assembling all the previous considerations at page 13, the weight update that we want to evaluate

while adjusting the network's weights to minimize the loss function is the following:

$$\Delta w_{ij} = -\eta \frac{\partial \mathcal{L}}{\partial w_{ij}} \tag{1}$$

Let us now focus on the partial derivative: given that the *loss* function is not explicitly defined over w_{ij}, we can decompose it via the chain rule as follows:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \underbrace{\frac{\partial \mathcal{L}}{\partial \varphi(\text{net}_j)} \frac{\partial \varphi(\text{net}_j)}{\partial \text{net}_j}}_{\delta_j} \frac{\partial \text{net}_j}{\partial w_{ij}} \tag{2}$$

Given that the loss function is explicitely defined over the output of a neuron, we obtain the $\frac{\partial \mathcal{L}}{\partial \varphi(\text{net})}$ contribution; on the other hand, we derive the sigmoid function over "net" instead than deriving it over the input because the former allows us to express the derivative over the output as follows: $\frac{\partial \varphi(\text{net})}{\partial \text{net}} = \varphi'(\text{net}) = \text{net}(1 - \text{net})$. On the other hand, the derivative of "net" over a given weight will just provide the value associated to the output.

Now, we should notice that the way to compute $\frac{\partial \mathcal{L}}{\partial \varphi(\text{net})}$ will substantially differ whether we're considering an non-output neuron or if we're referring to any other neuron.

- **Output neuron** (Figure 7a) : without any loss of generality, let us determine the loss function over just one sample pair $(\mathbf{x}, y)$. We will obtain that $\tilde{h} \approx \varphi(\text{net})$ for this single neuron, and the loss function will become $= \frac{\partial}{\partial \varphi(\text{net})} \frac{1}{2}(y - \varphi(net))^2 = \varphi(\text{net}) - y$. This final value is computed by `calculateError`:

```
double Perceptron::calculateError(double expectedValue) {
    error = (out - expectedValue);
    return (error);
}
```

  The δ_j in Equation 2 for the output neuron can be computed as in the following code:

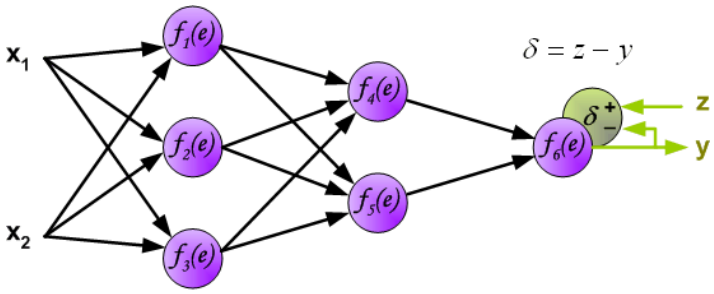
```
1  std::vector<double> Perceptron::calculateDerivative(double deltasFromForward) {
2      // Yet another approach to represent the second part from \delta_j: instead of computing net
           ↪ *(1-net) as in the
3      // tutorial, we can use e^net/(1+e^net)^2.
4      // deltasFromForward can be also the delta weight backprogagated from the network, as we
           ↪ will see in the next bullet point
5      derivative = deltasFromForward * (exp(net) / pow((1 + exp(net)), 2)) ;
6      return weights * derivative;
7  }
8  std::vector<double> Perceptron::calculateDerivativeFromExpected(double expectedValue) {
9      return calculateDerivative(-calculateError(expectedValue));
10 }
```

- **Internal neuron**: for any other neuron, we know that the output of a neuron j might influence a set of other neurons in $\mathcal{N}$ in the following layer and more specifically their net_ℓ values for $\ell \in \mathcal{N}$. Now, we need to backpropagate the error signal from the $\mathcal{N}$ neurons towards j; we then obtain (Figure 7b):

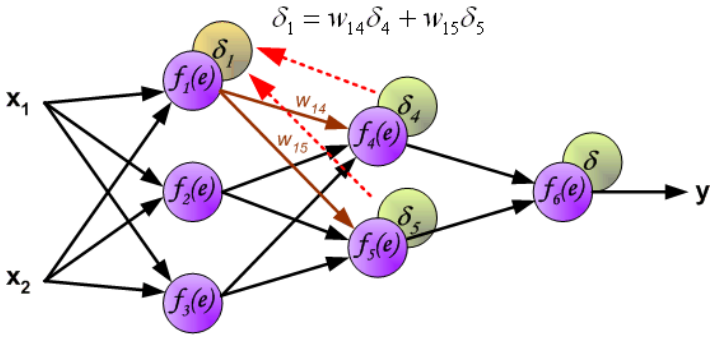
$$\frac{\partial \mathcal{L}}{\partial \varphi(\text{net}_j)} = \sum_{\ell \in \mathcal{N}} \frac{\partial \mathcal{L}}{\partial \text{net}_\ell} \frac{\partial \text{net}_\ell}{\partial \varphi(\text{net}_j)} \tag{3}$$

  Given that each neuron ℓ follows j, then the $\varphi(\text{net}_j)$ will be necessarily a contribution x_j of net_ℓ, for which $\frac{\partial \text{net}_\ell}{\partial \varphi(\text{net}_j)} = w_{j\ell}$. Similarly to what we have done in Equation 2, we decompose $\frac{\partial \mathcal{L}}{\partial \text{net}_\ell}$ as $\frac{\partial \mathcal{L}}{\partial \varphi(\text{net}_\ell)} \frac{\partial \text{net}_\ell}{\text{net}_\ell}$: given that this contribution now only contains the information pertaining to the neuron $\ell \in \mathcal{N}$, then this will be the δ_ℓ contribution that we want to backpropagate in the network. So, the contribution that any non-output neuron j backpropagates is the following:

$$\delta_j = \frac{\partial \mathcal{L}}{\partial \varphi(\text{net}_j)} \cdot \frac{\partial \text{net}_j}{\text{net}_j} = \sum_{\ell \in \mathcal{N}} w_{j\ell} \delta_\ell \cdot \varphi'(\text{net}_j)$$



(a) Determining the backpropagation quantity for the output neuron(s).

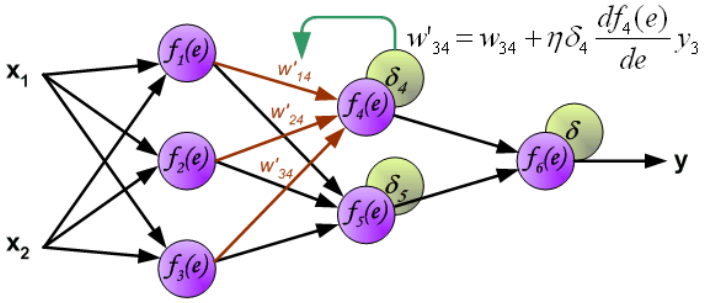


(b) By exploiting the chaining rule, we see that, for each non-output neuron, we replace the contribution $\delta = \tilde{y} - y$ with the weighted sum of the δ_j-s coming from the perceptrons in the subsequent layer.

$$w'_{14} = w_{14} + \eta\delta_4 \frac{df_4(e)}{de} y_1$$

$$w'_{24} = w_{24} + \eta\delta_4 \frac{df_4(e)}{de} y_2$$

$$w'_{34} = w_{34} + \eta\delta_4 \frac{df_4(e)}{de} y_3$$



(c) Via the derivative we calculate the *gradient* contribution, through which we update every single weight, θ included (not in the picture.

Figure 7: Graphical representation of the Backpropagation algorithm (`http://galaxy.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html`)

For any internal neuron, the summation will be the actual value of `deltasFromForward`, and the weight that is required by neuron j can be accessed directly from neuron ℓ: therefore, the ℓ can backpropagate the weight as well as the delta as already shown in Line 6.

Going back to Equation 2, we can also determine which is the δ_j contribution for any output neuron. Now, we can perform the summation at the layer level, so that we can return it to all the previous layers in the network. The derivative calculation is computed by the chaining of the two methods from both `Layer` and `BackwardPropagationNetwork` itself.

```
void BackwardPropagationNetwork::calculateDerivatives(std::vector<double> expectedOutput) {
    static size_t N = layersSize.size();
    for (int i = N-1; i >= 0; i--) {
        expectedOutput = layers[i].calculateDerivative(expectedOutput, (i == N - 1));
    }
}

std::vector<double> Layer::calculateDerivative(std::vector<double> &expectedValue, bool
    ↪ FromExpectedValue) {
    assert(expectedValue.size() == this->perceptrons.size());
    size_t N = expectedValue.size();

    std::vector<double> result; // Computing the
    for (size_t i = 0; i<N; i++) {
        result += FromExpectedValue ?
                  perceptrons[i].calculateDerivativeFromExpected(expectedValue[i]) : // output
                      ↪ neuron
                  perceptrons[i].calculateDerivative(expectedValue[i]);       // any other neuron
    }

    return result;
}
```

After propagating all the δ_j values, each neuron can update himself independently from the other neurons. In order to implement this goal, we can expand the delta contribution in Equation 1 as $\Delta w_{ij} = w_{ij}^{(\tau+1)} - w_{ij}^{(\tau)}$ where $w_{ij}^{(\tau)}$ provides current network weight prior to the present backpropagation run, that is at time τ. We can now express that equation as follows (Figure 7c):

$$w_{ij}^{(\tau+1)} = w_{ij}^{(\tau)} - \eta\delta_j\varphi'(\text{net}_j)\varphi(\text{net}_i)$$

After remembering that $\varphi(\text{net}_i)$ corresponds to the actual i-th input received by the j-th neuron in the following layer, we can also encode this part of the function as follows:

```
static inline double gradient(double input, double derivative) { return (input * derivative); }
static inline double gradient(double derivative)               { return (derivative); }

void Perceptron::updateGradient() {
    for (size_t i = 0; i<input_values.size(); i++) {
        input_gradients[i] = gradient(input_values[i], derivative);
    }
    theta_gradient = gradient(derivative);
}

void Layer::updateGradient() { for (Perceptron& p : perceptrons) p.updateGradient(); }

void BackwardPropagationNetwork::gradientUpdate() { for (Layer& p : layers) p.updateGradient(); }
```

In order to attenuate the oscillations during the gradient descent updating the weight, we can introduce a momentum parameter α so to smothen the transition from the current updated weight from the previous value:

$$\Delta w_{ij}^{(\tau+1)} = -\eta\delta_j\varphi'(\text{net}_j)\varphi(\text{net}_i) + \alpha\Delta w_{ij}^{(\tau)}$$

More evidence for the momentum are provided in [9]. The final following source code implements such equation for each neuron:

```
void Perceptron::updateWeight(double learningRate, double momentum) {
    double deltaWeight;
    for (size_t i = 0; i<weights.size(); i++) {
        weight& x = weights[i]; // ref
```

```
        // The last equation needs to be computed for each weight associated to the network
        deltaWeight = (learningRate * input_gradients[i]) + (momentum * x.previous);
        // Preserving the current increment for the momentum computation
        x.previous = deltaWeight;
        // Updating the current weight
        x.current = x.current + deltaWeight;
    }
    // Remember! The bias is also considered as a specific case of a weight associated to an input
        ↪ always returning 1. For this code we repeat the same logic as we did for all the other
        ↪ weights.
    deltaWeight = (learningRate * theta_gradient) + (momentum * theta.previous);
    theta.previous = deltaWeight;
    theta.current = theta.current + deltaWeight;
}

void Layer::updateWeight(double learningRate, double momentum) {
    for (Perceptron& p : perceptrons) p.updateWeight(learningRate, momentum);
}

void BackwardPropagationNetwork::updateWeight(double learningRate, double momentum) {
    for (Layer& p : layers) p.updateWeight(learningRate, momentum);
}
```

The following train method is the one used for training the network over the XOR function:

```
double BackwardPropagationNetwork::train(struct finite_function &f, size_t iterationNumber, const
    ↪ double learningRate,
                                         const double momentum) {
    size_t epoch = 0;
       double error;
    while (epoch < iterationNumber)
    {

        error = 0.0;
        size_t N = f.finite_function.size();

        // For all the elements in the finite function f:
        for (int i = 0; i < N; i++) {
            // Train the network over the current input
            std::vector<double> result = compute(f.finite_function[i].input);
            // Calculate the error just for the heuristic's sake
            error += std::pow(calculateQuadraticError(f.finite_function[i].output), 2);

            // Perform the backpropagation algorithm over this input
            calculateDerivatives(f.finite_function[i].output);

            // Update gradient and weight of each neuron in the network independently
            gradientUpdate();
            updateWeight(learningRate, momentum);
        }
        error = std::sqrt(error);
        std::cout << error << "--" << epoch << std::endl;
        epoch = epoch + 1;

        //If the neural network is not converging after 4000 epochs, it might be possible that this
            ↪ network configuration does not provide a good local minimum. Therefore, it might be
            ↪ useful to reset the weights of the network and restart the training
        if (epoch > 4000 && error > 0.5) {
            if (useRE) reset(re); else reset(re);
            epoch = 0;
        }
    }

    // Error associated to the final configuration
        return error;
}
```

3.3 Final Considerations

As showed in the `mpl_train` function at page 22, given an arbitrary dataset E and its training data T_r, we cannot determine for sure which will be the best network configuration to minimize the

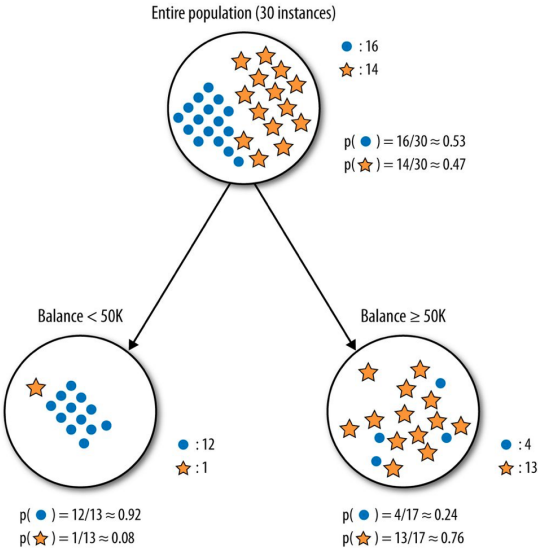


Figure 8: This figure provides an intuitive representation of the decrease of impurity that we want to assess for splitting the original dataset (root node) into two different subsets represented by two distinct childs: given an attribute named *Balance*, the predicate $r.Balance < 50K$ for a given record r seems to be a good predictor as it is a good predictor for separating the class *circle* from the *stars*.

loss function. Furthermore, any initial random initialization of the network's weight might produce different local minima, thus potentially requiring to re-initialize the weights of the network and re-start the training from the first epoch (see the last code snippet). As a consequence, for an arbitrary classification problem, the problem of training a network is:

$$\min_{L\in\mathbb{N},\ell\in\mathbb{N}^L,W^1,\dots,W^L} loss(\tilde{h}_{W^1,\dots,W^L}, T_r)$$

As we saw for the XOR example, the task of determining L and ℓ could be simply determined from the function that we want to train but, generally speaking, that is not possible when f is implicit. On the other hand, neural networks can be easily tricked to misclassify some images by inserting unforseed object [1, 7] or by just adding some random noise[15]. The usual approach to overcome such shortcomings is to use Adversarial Neural Networks which require the usage of `minimax` optimization problems. Nevertheless, this other strategy has been recently proved to be uneffective for videogame AI[16]. In the following sections, we're going to show other machine algorithms that can already overcome such shortcomings: with decision trees, we will see how to approximate the space partitioning problem via a greedy algorithm. On the other hand, such technique is more prone to overfitting.

Last, we're going to show that, if you don't have any specialized prior knowledge about a domain, then Support Vector Machines are a better choice to do off-the-shelf supervised learning. The usage of RBF kernels is also going to make such model resistant to overfitting.

4 Decision Trees

Given that Decision Trees are not implemented in DLib, we're going to provide a from scratch implementation of those in our code.

An k-ary decision tree is a classifier constructed by repeatedly splitting T_r into k distinct subsets. In order to detect which is the best way to do the split, we need to chose them in such a way that the descendent subsets are always purer than their parents. Figure 8 provides an intuitive representation of what a good impurity function should do, that is providing a good separation for the data. Albeit the concept might seem quite easy, providing an optimal binary decision tree is still a NP-Hard problem:

[15] `https://codewords.recurse.com/issues/five/why-do-neural-networks-think-a-panda-is-a-vulture`
[16] See `https://www.technologyreview.com/s/615299/reinforcement-learning-adversarial-attack-gaming-ai-deepmind-alphazero-`

in fact, we don't know how many regions are required to optimally separate our data but still, even if M is known in advance, finding the optimal subdivision is a combinatorial problem. In fact, an optimal split should minimize the number of elements belonging to different classes per region:

$$\min_{M} \min_{\{R_m\}_{1\leq m\leq |M|}} \frac{1}{|R_m|} \sum_{(\mathbf{x},y)\in R_m} (y - \mathbb{P}(R_m))^2$$

In fact, we can see that all the possibile combinations of grouping T_r in M groups are $\binom{|E|}{M} = \frac{|N|!}{M!(|N|-M)!}$ [13], thus reducing to analyse a nearly factorial number of combinations. Therefore, most algorithms use a greedy approach to separate the data, so that the impurity function is used as an heuristic to determine the split conditions.

In our tutorials, we're going to narrow our discussion to binary decision trees, so using $k = 2$. We can formally define such data structures as follows:

Definition 3 (Binary Decision Tree)**.** *A binary node contains a (sub)set of the training set $\kappa(t) \subseteq T_r$. Any non-leaf node contains two node children, $t.\texttt{left}$ and $t.\texttt{right}$ and induces a decision predicate P, where all the samples in the left child satisfy the predicate ($\forall x \in \kappa(t.\texttt{left}).\ P(x)$) and all the samples in the right child do not satisfy it ($\forall x \in \kappa(t.\texttt{left}).\ \neg P(x)$); as a consequence, we will have that $\kappa(t.\texttt{left}) \cap \kappa(t.\texttt{right}) = \emptyset$ and $\kappa(t.\texttt{left}) \cup \kappa(t.\texttt{right}) = \kappa(t)$. Each leaf node ℓ has an associated $\texttt{precision}$ value, determining how elements in $\kappa(\ell)$ are positive examples.*

A binary research tree is a non-leaf binary node inducing the following regression function:

$$\tilde{h}_t(\mathbf{x}) = \begin{cases} t.\texttt{precision} & t \text{ is leaf} \\ \tilde{h}_{t.\texttt{left}}(\mathbf{x}) & t.P(\mathbf{x}) \\ \tilde{h}_{t.\texttt{right}}(\mathbf{x}) & \text{oth.} \end{cases} \tag{4}$$

With reference to the binary decision tree τ in Figure 8, let us suppose that circles represent the positive examples for our binary classification; we can see that any record r having a *Balance* of 20K will return $\tilde{h}_\tau(r) \approx 0.92$ while for a balance of 99K you will obtain $\tilde{h}_\tau(r) \approx 0.24$. This implies that the first candidate will be more likely to belong to the circle class than the second one. The following structure characterizes the decision P associated to each single node, as well as the `precision` value and the *impurity decrease* value associated to the node.

```
#include <unordered_map>
#include <ostream>

/**
 * Defining the row assoicated to the dataset: a row is just a map assoicating an attribute to a given
     ↪ value
 */
using metric_row = std::unordered_map<std::string, double>; // same as typedef

struct split_field {
    std::string field_name;             ///<@ Attribute name performing the separation
    bool        is_equal   = false;     ///<@ If set to true, then it is the equality predicate,
        ↪ otherwise it is <
    double      splitting_point = -std::numeric_limits<double>::max(); ///<@ Value associated to the
        ↪ predicate
    double      impurity_decrease = 0.0 ///<@ Entropy gain associated to the current field
    double      precision = 0.0;       ///<@ Ratio of the samples belonging to the class over the
        ↪ elements that do not belong to it
    bool        is_pos_leaf = false;    ///<@ Determining whether the current node contains all elements
        ↪ that are positive examples
    bool        is_neg_leaf = false;    ///<@ Determining whether the current node contains all elements
        ↪ that are negative examples
    bool        is_not_discriminative = false;                    ///<@ The decision determined here is
        ↪ said to be not discriminative if it will end up having one empty child and the other
        ↪ containing the same dataset of the root

    split_field(const struct split_field& x ) = default;
    split_field& operator=(const struct split_field& x) = default;
    split_field() = default;
    friend std::ostream &operator<<(std::ostream &os, const split_field &field);
```

Algorithm 1 C4.5 Algorithm (Quinlan, 1993).

1: **function** SPLIT($n, d, \Delta\iota$)
2: **if** $\forall(\mathbf{x}, y) \in \kappa(n).\ y = 1$ **then** ▷ Contains only positive examples
3: $n.\texttt{precision} = 1$
4: **else if** $\forall(\mathbf{x}, y) \in \kappa(n).\ y = 0$ **then** ▷ Contains only negative examples
5: $n.\texttt{precision} = 0$
6: **else if** $d \leq 0$ **then** ▷ Reached maximum depth
7: $n.\texttt{precision} = \frac{|\{\ (\mathbf{x},y)\in\kappa(n)\ |\ y=1\ \}|}{|\kappa(n)|}$
8: **else**
9: Let $R(X_1, \ldots, X_n, Y)$ be the schema associated to the data
10: **map** $= \{\}$
11: **for each** X_i with $1 \leq i \leq n$ **do** ▷ Find the best cutoff s_i for an attribute X_i
12: $s_i = \left(\max_{r\in\kappa(n)} \arg\ \Delta\iota(n, X_i, r.X_i) \right).X_i$
13: **map**$[X_i]$:=s_i
14: **end for**
15: $X_k := \max_{X_i, 1\leq i\leq n} \arg$ **map**$[X_i]$ ▷ Variable maximising the decrease of impurity via its cutoff
16: $P := r \mapsto r.X_k <$ **map**$[X_k]$ ▷ Defining the predicate
17: $\kappa(n.\texttt{left}) := \{\ x \in \kappa(n)\ |\ P(x)\ \}$ ▷ Fit the left child with all the samples satisfying P
18: $\kappa(n.\texttt{right}) := \{\ x \in \kappa(n)\ |\ \neg P(x)\ \}$ ▷ Fit the right child with the remaining nodes
19: SPLIT$(n.\texttt{left}, d-1, \delta)$; SPLIT$(n.\texttt{right}, d-1, \delta)$; ▷ Run the algorithm recursively
20: **end if**
21: **end function**

```
    bool operator==(const split_field &rhs) const;
    bool operator!=(const split_field &rhs) const;
};
```

Given that finding the optimal number of splits is an NP-Hard problem, we often prefer to use a *greedy* approach to determine how to generate the new splits: these strategies follow the "*a bird in the hand is worth two in the bush*" approach, so preferring to achieve optimal splitting while dividing each node rather than obtaining the global optimum. Albeit this approach might potentially be detrimental for the model's precision, it provides a descriptive motivation of the reason why one element might fall in a given class.

Such greedy algorithms require that the impurity function ϕ acts as a heuristic function for our algorithm. At the moment, let us put aside the formal definition of such metric, and let us just focus of the pseudocode of the algorithm, and try to implement the previously said requirements in a code. In fact, let us suppose that from ϕ it is possible to determine a function $\Delta\iota(n, X_i, s_i)$ for decreasing impurity when $P(r) := r.X_i < s_i$ is chosen as a decision predicate. Algorithm 1 provides the pseudocode of the decision process that is associated to the greedy algorithm, where each node n is associated to a region $\kappa(n)$ that might be potentially splitted. We need to stop the recursion splitting the current node in other two child nodes as soon as you find nodes containing all samples belonging to one of the two classes (Line 3 and 5) or when you reach the maximum tree depth (Line 7). Otherwise, we need to determine which attribute X_i provides the best cutoff point $s_i \in dom(X_i)$ maximizing the decrease of impurity (Line 15) but, before doing that, we need to find the best cutoff point s_i for each fixed attribute X_i (Line 12). Then, we split the data elements associated to the node via the decision predicate induced by the cutoff point, and then recursively split the left and right nodes if possible.

Given that the actual implementation of the C4.5 algorithm requires also some additional knowledge related to the definition of the impurity function, we're going to provide a more concrete C++ implementation after the following paragraph.

Impurity Function (*) Before introducing the impurity function, we need some additional terminology and notation. With reference to the last tutorial, we can model a generalization of the cover function $\mathbf{c}_2$, where $\mathbf{c}_2(y, S)$ returns all the objects in S that belong to a class $y \in \mathcal{Y}$. If we approximate

probabilities with a *frequentist*[17] approach, then we can denote the probability than an object in S belongs to a class y as $p(y, S) = \frac{|\mathbf{c}_2(y,S)|}{|S|}$. When a node n is clear from the context, then the **prior probability** that an object belongs to a given class y is $\mathbb{P}(y) = p(y, \kappa(n))$.

Now, given two events E and F, the **condidtional probability** of E happening given that we know that B happened too is defined as the ratio of the probability when both A and B occur over the probability of event B happening. When a node n is clear from the context, we can determine the conditional probability $\mathbb{P}(y|P)$ of an element of n belongs to class y given that the test P of the left child node passed as follows:

$$\mathbb{P}(y|P) = \frac{p(y, \kappa(n.\texttt{left}))}{|\kappa(n.\texttt{left})|/|T_r|}$$

where he probability that P is satisfied is $\mathbb{P}(P) = \frac{|\kappa(n.\texttt{left})|}{|T_r|}$. Similarly, the conditional probability of an element of n belongs to a class y given that the test P doesn't hold can be modelled as $\mathbb{P}(y|\neg P) = \frac{p(y,\kappa(n.\texttt{right}))}{|\kappa(n.\texttt{right})|/|T_r|}$, while the probability of the test is not passed is $\mathbb{P}(\neg P) = \frac{|\kappa(n.\texttt{right})|}{|T_r|}$.

Given all these definitions, we can now define the impurity function for each node n of a decision tree that needs to be splitted.

Definition 4 (Impurity Function)**.** *Given a node n that is clear from the context and a decision problem over an arbitrary $\mathcal{Y}$, an **impurity function** ϕ [12] is defined over a set of tuples $k \in \mathcal{K}$ determining the probability of each class $y \in \mathcal{Y}$ for a specific node, i.e. $\langle \mathbb{P}(y) \rangle_{y \in \mathcal{Y}}$. Any impurity function shall meet the following criterion:*

- *ϕ reaches its maximum when the given node is impure, i.e. we have an uniform class distribution:* $\arg\max_{k \in \mathcal{K}} \phi(k) = \langle 1/|\mathcal{Y}| \rangle_{y \in \mathcal{Y}}$
- *ϕ reaches its maximum when the given node is pure, i.e. only one class y is certain ($\mathbb{P}(y) = 1$) and all the others have zero probability*[18]*.*

For a binary decision problem $|\mathcal{Y}| = 2$ represented via a binary decision tree, the impurity measure $\iota(n) = \phi(\langle \mathbb{P}(y|n) \rangle_{y \in \mathcal{Y}})$ over a node n having a decision predicate $P(r) := r.X_i < s_i$ which is clear from n defines a decrease of impurity (or information gain) for a binary decision tree rooted in n as:

$$\Delta\iota(n) \equiv \Delta\iota(n, X_i, s_i) := \iota(n) - \underbrace{\mathbb{P}(P)\iota(n.\texttt{left}) - \mathbb{P}(\neg P)\iota(n.\texttt{right})}_{\iota(n|P)}$$

where the quantity $\iota(n|P)$ is often referred as the "posterior" impurity, i.e. the impurity achieved after splitting the tree with the decision predicate P.

Two very well known impurity functions are the **Gini Index** and the **(Entropy) Information Gain**. The aim of the *Gini Index* is to describe the distribution of inequality within a population ($\kappa(n)$) of some give values that, in our cases, are the distribution of the target classes. If we express that in terms of variance, we can adopt the one-against-all approach and express the variance of each class as the representativeness of class y against all the other $\mathcal{Y} \backslash \{y\}$, so the sample variance of these values is $\mathbb{P}(t)(1 - \mathbb{P}(t))$ for the positive examples. For the binary classification problem and the Gini Index, we will obtain the following impurity function:

$$\iota_{\text{Gini}}(n|P) := 1 - \mathbb{P}^2(P) - \mathbb{P}^2(\neg P)$$

The Gini metric can be now coded as follows:

```
struct gini_metric {
    double node_iota(double freq_posLeft);
    double posterior(double sizeLeft, double sizeTotal, double posLeft, double posRight);
};

double gini_metric::node_iota(double freq_posLeft) { return 1.0 - std::pow(freq_posLeft, 2) - std::pow
    ↪ (1.0 - freq_posLeft, 2); }
```

[17] See `https://en.wikipedia.org/wiki/Frequentist_inference` for more details.

[18] More formally, $\arg\min_{k \in \mathcal{K}} \phi(k) = t \; \exists! y \in \mathcal{Y}.\; t_y = 1 \wedge \forall y' \in \mathcal{Y} \backslash \{y\}.\; t_{y'} = 0$

```
double gini_metric::posterior(double sizeLeft, double sizeTotal, double posLeft, double posRight) {
    double sizeRight = sizeTotal - sizeLeft;
    return (sizeLeft / sizeTotal) * node_iota(posLeft) + (sizeRight / sizeTotal) * node_iota(posRight)
        ↪ ;
}
```

For the Entropy Information Gain, we want to determine the average amount of information needed to identify the class of an example randomly picked from $\kappa(n)$. Given that from probability theory we will have that the classes form a partition and therefore from probability theory we have $\sum_{y \in \mathcal{Y}} \mathbb{P}(y) = 1$. Therefore, we can use those quantities to determine the average quantity of information conveyed by each class: given that the quantity of information conveyed in a binary representation can be modelled as $\log_2(\mathbb{P}(y))$, then we will obtain the following impurity function:

$$\iota_{\text{Entropy}}(n|P) := -\mathbb{P}(P)\log_2(\mathbb{P}(P)) - \mathbb{P}(\neg P)\log_2(\mathbb{P}(\neg P))$$

where we must take specific care of the logarithmic function when $\mathbb{P}(P) = 0$. The entropy metric can now be coded as follows:

```
struct entropy_metric {
    double node_iota(double freq_posLeft);
    double posterior(double sizeLeft, double sizeTotal, double posLeft, double posRight);
};

#include <cmath>
#ifdef _WIN64
#include <limits>           // MSVC requires limits to have the numeric setup
#endif

static inline double xlogeps(double x) {
    static double eps = std::numeric_limits<double>::epsilon();
    return (x <= eps) ? 0.0 : x * std::log2(x);
}

double entropy_metric::node_iota(double freq_posLeft) { return - freq_posLeft * xlogeps(freq_posLeft)
    ↪ - (1.0 - freq_posLeft) * xlogeps(1.0 - freq_posLeft); }

double entropy_metric::posterior(double sizeLeft, double sizeTotal, double posLeft, double posRight) {
    double sizeRight = sizeTotal - sizeLeft;
    return (sizeLeft / sizeTotal) * node_iota(posLeft) + (sizeRight / sizeTotal) * node_iota(posRight)
        ↪ ;
}
```

As you might observe, we haven't used any inheritance here: in fact, the `decision_tree` accepting as a template argument `template <typename impurity_function>` and containing a field `impurity_function functions` will be instantiated at compile time once either `gini_metric` or `entropy_metric` will be passed as type parameters: only at that time the compiler will check whether the object of type `impurity_function` will have both `posterior` and `node_iota` methods declared.

As formally proved in [12], these two metrics disagree only on the 2% of the cases (some of which are also provided in the current classification task), so that's why actual empirical results cannot decide which metric actually performs better. Therefore, we might prefer using the Gini Index after all since it does not involve computations of logarithmic quantities.

4.1 Implementation

In this section, we're going to focus on how to implement a binary decision tree, and how to train it using a one-against-all approach. If we pick a class `positive_class` in $\mathcal{Y}$ to be picked as a positive one while training a tree, then we can define the decision tree with the following structure:

```
#include "split_field.h"
#include <vector>
#include <ai/datasets/DLibSplits.h>

/**
 * Implementation of the decision tree, where the impurity function is passed as a type parameter
 */
template <typename impurity_function>
```

```
struct decision_tree {
    std::vector<metric_row> rows;          ///<@ all the rows that are currently associated to the
        ↪ current elemtn
    std::string           class_field;     ///<@ the field that contains the class names
    double                positive_class; ///<@ The class among the @|\mathcal{Y}|@ that is going to
        ↪ provide the positive example
    std::vector<std::string> table_schema;  ///<@ the columns' names except from the class field
    struct split_field    root;            ///<@ decision associated to the current root
    std::vector<decision_tree> children;    ///<@ number of children in the decision tree (in this
        ↪ implementation, only two children)
    impurity_function     functions;       ///<@ impurity function function that is used to do compute
        ↪  the gain

    decision_tree() = default;
    ~decision_tree() = default;
    decision_tree(const decision_tree<impurity_function>& x) = default;
    decision_tree& operator=(const decision_tree<impurity_function>& x) = default;

    /**
     * Populating the root of the tree with all the samples from the dataset
     * @param populator      Element containing all the initial examples
     * @param positive_class      The target class that is now considered to be the positive example
     */
    decision_tree(DLib_Splits& populator, double positive_class) {
        table_schema = populator.colnames;
        class_field = table_schema.front(); // The class is defined as the first attribute in the
            ↪ schema
        table_schema.erase(table_schema.begin()); //Removing the classification attribute from the
            ↪ predictors
        this->positive_class = positive_class;

        // Iterating over all the possible rows
        size_t N = std::min(populator.training_input.size(), populator.training_labels.size());
        for (size_t i = 0; i<N; i++) {
            metric_row row;

            // Converting each vector into a attribute-value row
            auto& matrixRow = populator.training_input[i];
            size_t M = matrixRow.size();
            for (size_t j = 0; j<M; j++) {
                row[table_schema[j]] = matrixRow(j);
            }
            // Also, adding the class information
            row[class_field] = populator.training_labels[i];
            // Adding the current record to the set of all the records
            rows.emplace_back(row);
        }
    }

    // <bestField>
    // <continues>
    // <classification>
private:
    // <classification2>
    // <varSplit>
};
```

For each resulting decision tree, we can implement the regression function provided in Equation 4 as the following recursive function:

```
    double class_probability(const dlib::matrix<double>& matrixRow) {
        // Converting the vector into a row representation
        metric_row row;
        size_t M = matrixRow.size();
        for (size_t j = 0; j<M; j++) {
            row[table_schema[j]] = matrixRow(j);
        }

        // Running the actual probability evaluation that the element belongs to the current class
            ↪ given the decision
        return class_probability(row);
    }
```

```
// <varSplit>:=
double class_probability(metric_row& row) {
    // I need to stop the recursion if the current node is a leaf, and if therefore it has no
        ↪ children.
    if ((root.is_pos_leaf || root.is_neg_leaf) || (root.is_not_discriminative || children.empty()))
        ↪  {
        if (root.is_pos_leaf)
            return 1.0; // If I reach this point, I will have a positive element for sure
        else if (root.is_neg_leaf)
            return 0.0; // If I reach this point, I will have a negative element for sure
        else
            return root.precision;  // If I reach a non-discriminative node or just a leaf, then the
                ↪  probability is the ratio between the positive and negative examples in that
                ↪ leaf
    } else {
        // Running the decision function from the root information, and then deciding which is the
            ↪ element to visit next
        if ((root.is_equal) ? (row[root.field_name] == root.splitting_point) : (row[root.field_name
            ↪ ] < root.splitting_point)) {
            return children[0].class_probability(row);
        } else {
            return children[1].class_probability(row);
        }
    }
}
```

Please note that, in our implementation, we also provide an equality predicate for the cases when the class only determines one single element from the dataset. Given this definition, we can define the classification function of the one-over-all approach similarly to what we have previously seen for the Multilayered Neural Networks:

```
//<classification>:=

/**
 * Estimating the hypothesis from the regression function
 * @tparam my_impurity_function  Custom impurity function associated to the decision tree
 * @param classes                Vector containing all the decision trees for each one of the
     ↪ classes
 * @param row                    Data record represented as a vector over which the N classifiers
     ↪ should do the decision
 * @return
 */
template <typename my_impurity_function> static std::pair<double, std::set<double>> classify(std::
    ↪ vector<struct decision_tree<my_impurity_function>>& classes, const dlib::matrix<double>
    ↪ row) {
    double score = -std::numeric_limits<double>::max();
    std::set<double> candidates;
    // Running similarly as the one-to-one classification problem for neuroal networks
    for (struct decision_tree<my_impurity_function>& classifier : classes) {
        double currentClass = classifier.positive_class;
        double prob = classifier.class_probability(row); // Running the regression function h̃
        if (prob > score) {
            score = prob;
            candidates.clear();
            candidates.emplace(currentClass);
        } else if (prob == score) {
            candidates.emplace(currentClass);
        }
    }

    return {score, candidates};
}
```

As we can see in the next coding step, the implementation of Line 12 from Algorithm 1 requires nearly 100 lines of code, and so we're going to implement it as one single method. We assume that `filed` stores the candidate X_i that we're considering at the current iteration step in `field.field_name`. Then, we scan the rows a first time to obtain a partial pre-computation of the entropy (Line 19): in particular, we compute the number of positive and negative cases associated to each value `r_field` (Lines 23-33), as well as counting the total number of examples (Line 35). This value is then use to determine the frequency of the positive (Line 38) and negative (Line 39) examples

and, as in the following lines, if we reached a node that might be either a positive or a negative leaf. Then, we determine whether the current node is empty, and therefore we won't be able to do any further splitting, or if the node contains only one root, and so we can use an equality predicate instead of the less-equal. Then, we can perform an ordered iteration over each potential splitting point in the map (Line 64) and get the induced frequencies of the positive examples for the rows that will belong eihter to the left or to the right subtree. We can now evaluate the impurity function for both the left (Line 80) and right (Line 84) subtree and compute the impurity decrease via the previously computed root impurity (Line 89). Next, at each iteration we select the best candidate, thus updating the information of `field` passed by reference (Line 92).

```
1   // <varSplit>:=
2   /**
3    * For a given attribute, determines which is the best value providing the greatest impurity
         ↪ decrease
4    * @param field   Field containing the attribute's name, and containing (at the end) the value
         ↪ over which the split is performed
5    * @return        <is_positive,is_negative> pair determining if the decision is able to target one
         ↪  of the two classes of interest.
6    */
7   std::pair<bool,bool> bestFieldSplit(struct split_field& field) {
8       assert (field.field_name != class_field);
9       std::set<double>        sorted_values;
10      std::map<double, double> entropy_precompute;
11      std::map<double, double> frequency;
12
13      double maxGain = -std::numeric_limits<double>::max();
14      double posRoot = 0.0;
15      double negRoot;
16      double rootImpurity;
17
18      // Getting all the possible values for the given field name. In order to do so, we need to
            ↪ iterate over the rows
19      for (metric_row& r : rows) {
20          double r_field = r[field.field_name];
21          double r_class = r[class_field];
22
23          // check if we have already encountered the value r_field
24          auto  it = entropy_precompute.find(r_field);
25          // key already present in the map
26          if (it != entropy_precompute.cend()) {
27              if (r_class == positive_class) it->second+=1.0; // count the number of positive
                    ↪ examples for the class
28              frequency.find(r_field)->second+=1.0;         // count the number of total examples for
                    ↪ the value
29          } else { // key not found
30              if (r_class == positive_class)
31                  entropy_precompute[r_field] = 1.0; // count the number of positive examples for the
                        ↪ class
32              frequency[r_field] = 1.0;           // count the number of total examples for the value
33          }
34
35          if (r_class == positive_class) posRoot++;
36      }
37
38      posRoot /= ((double)rows.size()); // Evaluating the positive coverage
39      negRoot = 1.0 - posRoot;        // Evaluating the negative coverage
40
41      // if the leaf has posFreq == 1, then this node will contain only the nodes that are positive
            ↪ examples: stop the iteration.
42      if (posRoot == 1.0) {
43          return {true, false};
44      }
45      // similarly, if negFreq == 1, then this node will contain only the nodes that are negative
            ↪ examples: stop the iteration.
46      if (negRoot == 1.0) {
47          return {false, true};
48      }
49
50      // If there is only one value over which we can discriminate, then we cannot use the current
            ↪ variable to further determine the data
51      if (entropy_precompute.empty()) {
52          field.is_not_discriminative = true;
```

```
            return {false, false};
        } else if (entropy_precompute.size() == 1) {
            // We use the equality predicate if the class is only going to tear apart one single
                ↪ element from the whole set.
            field.is_not_discriminative = false;
            field.is_equal = true;
        }

        // Using the metric to compute the impurity of the root prior to any split
        rootImpurity = functions.node_iota(posRoot);

        // Computing the entropy for each possible value (in a discrete way)
        auto it = entropy_precompute.begin();
        if (!field.is_equal) it++; // Skipping the first value, only if I have more than one value
        while (it != entropy_precompute.end()) {
            double positiveLeft = 0.0;
            double sizeLeft  = 0.0;
            double positiveRight = 0.0;
            double sizeRight = 0.0;

            // Getting all the elements that are less or equal than the current element in it->first
            for (auto it2 = entropy_precompute.begin(); it2 != entropy_precompute.end(); it2++) {
                ((it2->first < it->first) ? positiveLeft : positiveRight) += it2->second;
                ((it2->first < it->first) ? sizeLeft : sizeRight) += frequency[it2->first];
            }

            // Getting the frequencies for the left branch of the decision
            double freq_posLeft = sizeLeft == 0 ? 0.0 : positiveLeft / sizeLeft;
            double eLeft_eps = functions.node_iota(freq_posLeft);

            // Getting the frequencies for the right branch of the decision
            double freq_posRight = sizeRight == 0 ? 0.0 : positiveRight / sizeRight;
            double eRight_eps = functions.node_iota(freq_posRight);

            // Computing the impurity once we know which is the
            double entropyPosterior =
                   functions.posterior(sizeLeft, rows.size(), freq_posLeft, freq_posRight);
            double impurity_decrease = rootImpurity - entropyPosterior;

            // Setting up the new decision point once we obtain a better impurity decrease
            if ((impurity_decrease > maxGain) && (field.splitting_point < it->first)) {
                field.splitting_point = it->first;
                field.impurity_decrease = impurity_decrease;
                field.precision = posRoot;
                maxGain = rootImpurity;
            }

            it++;
        }

        // field now contains the best splitting point for the given field name
        return {false, false};
    }
```

As a next step, we determine X_k and P by iterating over all the possible attributes in the schema that are not classes, and by updating the `root` field of the decision tree with the best candidate `f`:

```
    // <bestField>:=
    /**
     * Select the best attribute with the best splitting point
     * @param parent_field:  We pass the parent's decision predicate information, so to skip a
         ↪ decision that is similar to the one of the parent
     */
    void bestField(struct split_field* parent_field = nullptr) {
        struct split_field f;                                  // Temporary decision predicate
        double maxEntropy = -std::numeric_limits<double>::max(); // Setting a low value of entropy, so
             ↪ that we can detect the maximum
        root.is_not_discriminative = true;

        // Get which is the best column providing the best separation
        for (const std::string& columnName : table_schema) {  // Iterating over all the possible
             ↪ attribute names
            split_field currentField;                          // Current field
```

```
            currentField.field_name = columnName;
            auto cp = bestFieldSplit(currentField);        // Using the outcome if we had a positive or
                ↪  a negative leaf via cp
            if (cp.first) {
                root.is_pos_leaf = true;
                root.is_not_discriminative = false;
                return;
            } else if (cp.second) {
                root.is_neg_leaf = true;
                root.is_not_discriminative = false;
                return;
            } else {                                          // If I'm still considering the general
                ↪ case where there is no perfect division, then
                if ((currentField.is_not_discriminative) || (root == currentField) || (parent_field !=
                    ↪ nullptr && (currentField == *parent_field)))
                    continue;                                 // Skip the current decision outcome if
                        ↪ either makes one child empty, or if I make the same decision as the root (
                        ↪ break the loop).
                else if (currentField.impurity_decrease > maxEntropy) { // Otherwise, consider this
                    ↪ field as a candidate only if it maximizes the entropy from the former candidate.
                    root.is_not_discriminative = false;
                    maxEntropy = currentField.impurity_decrease;
                    f = currentField;
                }
            }
        }

        // f now contains the best table option for the field
        if (!root.is_not_discriminative) root = f;
    }
```

Finally, the expand function provides the whole implementation of Algorithm 1:

```
    /**
     * Expanding the current node into multiple subtrees up until a given distance is reached
     *
     * @param maxDepth       Maximum depth that needs to be reached
     * @param depth          Current iteration depth
     * @param type           Trailing string used for displaying the tree while generating it
     */
    void expand(int maxDepth, const int depth = 0, const std::string& type = "", struct split_field*
        ↪ parent_field = nullptr) {
        bestField(parent_field); // Computing the best field providing the split
        std::cout << std::string(depth*2, '.') << type;
        if ((root.is_pos_leaf || root.is_neg_leaf) || (depth >= maxDepth) || (root.
            ↪ is_not_discriminative)) {
            // Stopping condition: if I reached the maximum tree size, or if I reach a non-
                ↪ discriminative node, or if I reach a leaf.
            std::cout << "␣=␣" << ((root.is_pos_leaf) ? 100.0 : (root.is_neg_leaf ? 0.0 : root.
                ↪ precision * 100)) << "%" << std::endl;
        } else {
            // Generating two new childs
            children.emplace_back();
            children.emplace_back();
            decision_tree& left = children[0];
            decision_tree& right = children[1];
            // Setting to each child the same information as the root
            left.table_schema = right.table_schema = table_schema;
            left.class_field = right.class_field = class_field;
            left.positive_class = right.positive_class = positive_class;

            // Splitting the nodes into left and right child
            for (metric_row& row : rows) {
                if ((root.is_equal) ?
                (row[root.field_name] == root.splitting_point) : // Using the equality predicate
                (row[root.field_name] < root.splitting_point)) // Using the less-than predicate
                {
                    left.rows.emplace_back(row);
                } else {
                    right.rows.emplace_back(row);
                }
            }
```

```
            // Removing duplicated data
            rows.clear();
            if (depth > 0) table_schema.clear();

            // recursive calls
            std::cout << "[" << root.field_name << (root.is_equal ? '=' : '<') << root.splitting_point
                ↪ << "] ?" << std::endl;
            left.expand(maxDepth, depth+1, "lhs ", &root);
            right.expand(maxDepth, depth+1, "rhs ", &root);
        }
    }
```

Then, given the number of the classes k from `generateSplit`, we can generate and train k distinct classifiers. Then, we approximate the model's precision with the inverse of the normalized average distance between the predicted and expected classification for each of the rows in the testing set.

```
template <typename Metric> void multi_rtree_train() {
    DLib_Splits splits;
    const std::pair<const size_t, const size_t> &info = generateSplit<dataset_full_dimensions>("data/
        ↪ starcraft.csv", splits,
                                                                                    false, 0.7);

    // Training the model one class against the other
    using CLS = std::vector<decision_tree<Metric>>;
    CLS classifiers;
    for (double classe = 1.0; classe <= 8.0; classe++) {
        std::cout << classe << std::endl;
        struct decision_tree<Metric> predict_class(splits, classe);
        predict_class.expand(5); // Stopping the recursion at depth of 5
        classifiers.emplace_back(predict_class);
        std::cout << std::endl << std::endl;
    }

    size_t N = std::min(splits.testing_label_vector.size(), splits.testing_input.size());
    double distance = 0.0;
    for (size_t i = 0; i<N; i++) {
        double ithDistance = 0.0;
        std::pair<double, std::set<double>> result = decision_tree<Metric>::template classify<Metric>(
            ↪ classifiers, splits.testing_input[i]);
        ithDistance = class_prediction_distance(splits.testing_labels[i], result.second);
        // normalize the distance
        ithDistance = ithDistance / (ithDistance+1.0);
        distance += ithDistance;
    }
    // distance normalization
    distance = (distance) / ((double)N);
    // invert the distance, so to obtain the precision
    double precision = 1.0 - distance;

    std::cout << "Model precision over the testing data: " << precision << std::endl;

}
```

4.2 Final Considerations

Given that region separation can promptly lead to overfitting, in these tutorials we applied decision tree pruning by detecting the irrelevance of each node: if the tree node has a decision $P(r) = r.X_i \leq s_i$ and the right leaf has the decision $P'(r) = r.X_i \leq s'_i$ with $s_i \leq s'_i$, when we can merge the child node with the parent one. Moreover, we implemented **early stopping** by stopping the region splitting when we reach a given depth. Nevertheless, a better approach to determine to stop splitting the node in different regions should include the relevance of the best splitting. Albeit the usual technique requires the usage of a χ^2 distribution[19], we can also approximate such value with a threshold value to the impurity decrease, i.e., we are not splitting the nodes that are not going to provide a significant improvement in data classification.

[19] http://www.cplusplus.com/reference/random/chi_squared_distribution/

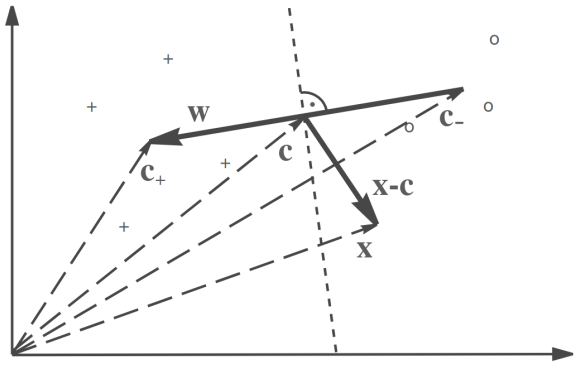


Figure 9: The dotted line represents the 2D plane π separating the postive examples (crosses) with centroid $\mathbf{c}_+$ from the negative examples with centroid $\mathbf{c}_-$ [2].

5 ν-Support Vector Machines with Kernel Trick

Summing up the considerations made for previous learning models, while there is no heuristic for neural networks to date allowing to navigate the configuration parameter space by providing a good network configuration for our dataset, decision trees having the configuratin parameter space being identical to the $\mathcal{L}_e$ search space possess a greedy heuristic that partition both of them. Also, the minimization of a loss function for decision trees of is not determined by randomly initialized weights as in the case of neural networks but is all governed by a heuristic, the impurity function. However, both approaches have structural problems: while the former requires to trade off between network parameters optimizing the loss over T_r and good predictions over T_e , the latter involves the use of similar metrics to those already used to decide on splitting. Consequently, we wonder if there are available models that both do not necessarily require a-priori knowledge of the data distribution and, at the same time, are resistant to overfitting. These questions boil down to ask ourselves whether an off-the-shelf learning model provides a proper decision function quickly. ν-Support Vector Machines are a possible answer to this quest [13]: in fact, they can learn problems that cannot be separated linearly through the use of specific similarity functions called kernels and are also roughly resistant overfitting. While the first solution allows us to transform our data into another "feature space" where the problem can become linearly separable, the latter is due to the reduction in the number of model parameters (in the training phase). E.g., ν-Support Vector Machines with a Radial Basis Kernel Function narrow the training parameters down to ν itself and a γ parameter for the Gaussian distribution. To better understand these details, however, we need to deepen our knowledge of the learning model in question.

As per previous sections, let us focus on designing a binary classifier over an n-dimensional dataset and, for the moment, let us assume that we want only to deal with linearly separable problems, which definition was provided at page 6. We can boil down the latter definition to ask ourself which is the equation of an (hyper-)plane π in multiple dimensions, that is:

Definition 5 (Hyperplane)**.** *Given a (hyper-)plane π, a point $\mathbf{c} \in \pi$, and a non-null vector $\mathbf{w} \perp \pi$, we have that $\mathbf{x} \in \pi$ if and only if $\mathbf{x} - \mathbf{c} \perp \mathbf{w}$.*

The former definition also allows us to derive the Cartesian coordinates of π: given that two perpendicular vectors have a zero dot product, then we can express the π containing $\mathbf{c}$ and having a norm $\mathbf{w}$ as $\mathbf{w}(\mathbf{x}-\mathbf{c}) = 0$. As we might observe, this definition of a plane is similar to the characterization of the linearly separable problem provided at page 6 (where $\mathbf{c} \approx \theta$). We can now describe the simplest pattern recognition algorithm [2] (Figure 9): we have a set of positive (+ or $1 \in \mathcal{Y}$) and of negative ($-$ or $-1 \in \mathcal{Y}$) examples that are linearly separable. As a consequence, the perpendicular $\mathbf{w}$ to the plane π should be the line connecting the two centroids of the two classes ($\mathbf{w} = \mathbf{c}_+ - \mathbf{c}_-$). Given that the *centroid* of a class is the arithmetic mean position of all the data points, then we will obtain the required centroids as follows:

$$\mathbf{c}_+ := \frac{1}{n_{\mathbf{1}}} \sum_{(\mathbf{x},1) \in T_r} \mathbf{x} \qquad \mathbf{c}_- := \frac{1}{n_{-\mathbf{1}}} \sum_{(\mathbf{x},-1) \in T_r} \mathbf{x}$$

where $n_{\mathbf{1}} := |\mathbf{c}_2(1, T_r)|$ ($n_{-\mathbf{1}} := |\mathbf{c}_2(-1, T_r)|$) represent the number of positive (negative) examples. Let $\mathbf{c} := {}^{\mathbf{c}_+ + \mathbf{c}_-}/_2$ be the midpoints of the two centroids: the decision plane π should pass through $\mathbf{c}$ and should be perpendicular to the vector $\mathbf{w}$. Given the regression function $\tilde{h}(\mathbf{x}) = \mathbf{w}(\mathbf{x} - \mathbf{c})$, a binary approximation of the decision function for $\mathbf{x}$ becomes $h(\mathbf{x}) = \text{sign}(\tilde{h}(\mathbf{x}))$.

5.1 Kernel Trick

If we assume that K is a normalized similarity measure (**kernel**) such that $K \colon \mathbb{R}^n \times \mathbb{R}^n \to [0, 1]$, then we can express a distance d as the inverse of a similarity, as distance should measure dissimilarity [11]. Therefore, we could write that $d(X, Y) \propto 1 - K(X, Y)$. If we then assume that the distance is the squared Euclidean Distance, we can show that K can be defined as a dot product between normalized vectors.

Lemma 1. *The dot product represents vector similarity, i.e.* $\|X - Y\|_2 \propto 1 - (X \cdot Y)$ *with* $d(X, Y) = \|X - Y\|_2$ *and* $K(X, Y) = (X \cdot Y)$ *over two normalized vectors* X *and* Y*.*

Proof. Trivial by expanding the definition of Euclidean Distance:

$$\|X - Y\|_2^2 = ((X - Y) \cdot (X - Y)) = (X \cdot X) + (Y \cdot Y) - 2(X \cdot Y) = 2(1 - (X \cdot Y)) \tag{5}$$

□

As per previous considerations, the dot product will be a good similarity function only for linearly separable problems but, as we might see in a while, by changing the kernel function we might be able to deal also with non linearly separable problems. At this stage, we want to rewrite the definition of $\tilde{h}$ so that it so the dot product between $\mathbf{x}$ and all the positive and negative examples is explicit [2]:

$$\begin{aligned}
\tilde{h}(\mathbf{x}) &= (\mathbf{x} - \mathbf{c})\mathbf{w} \\
&= (\mathbf{x} - {}^{\mathbf{c}_+ + \mathbf{c}_-}/_2) \cdot (\mathbf{c}_+ - \mathbf{c}_-)) \\
&= ((\mathbf{x} \cdot \mathbf{c}_+) - (\mathbf{x} \cdot \mathbf{c}_-) + b) \\
&= \frac{1}{n_{\mathbf{1}}} \sum_{(\mathbf{x}', 1) \in T_r} (\mathbf{x}' \cdot \mathbf{x}) - \frac{1}{n_{-\mathbf{1}}} \sum_{(\mathbf{x}, -1) \in T_r} (\mathbf{x}' \cdot \mathbf{x}) + b \\
&= \frac{1}{n_{\mathbf{1}}} \sum_{(\mathbf{x}', 1) \in T_r} K(\mathbf{x}', \mathbf{x}) - \frac{1}{n_{-\mathbf{1}}} \sum_{(\mathbf{x}, -1) \in T_r} K(\mathbf{x}', \mathbf{x}) + b
\end{aligned}$$

where b is an offset. Given that K abstracts from the actual data representation format, it follows that we can now choose $\mathcal{L}_e$ to be any possible data representation beyond simple numeric data point. In particular, we can also define kernel functions for determining tree data structures similarities as the one proposed in [11]:

$$\mathrm{K}_{tree}^{K_\ell, K_\iota}(t, s) = \begin{cases} K_\ell(t, s) & t, s \text{ leaves} \\ K_\iota(t.\texttt{node}, s.\texttt{node}) + \sum_j K_{tree}^{K_\ell, K_\iota}(t.\texttt{child}(j), s.\texttt{child}(j)) & t.\texttt{node} = s.\texttt{node} \wedge |t.\texttt{child}| = |s.\texttt{child}| \\ K_\iota(t.\texttt{node}, s.\texttt{node}) & \text{oth.} \end{cases}$$

One of the first algorithms for inferring $\tilde{h}$ from empirical data was based on the observation that, among all hyperplanes providing a linear separation, there exists an unique π providing the maximum margin of separation between the classes. Using Figure 9 as a reference and given $\mathbf{x}_+$ ($\mathbf{x}_-$) the nearest point to π of the positive (negative) class where $\mathbf{w}$ and b are rescaled such that points satisfy $\tilde{h}(\mathbf{x}_+) = 1$ and $\tilde{h}(\mathbf{x}_-) = -1$, we can see that the distance between the two points is $\tilde{h}(\mathbf{x}_+) - \tilde{h}(\mathbf{x}_-)$, thus obtaining the following equation:

$$\begin{aligned}
(\mathbf{w} \cdot \mathbf{x}_+) + b - ((\mathbf{w} \cdot \mathbf{x}_-) + b) &= 2 \\
\Updownarrow & \\
\mathbf{w} \cdot (\mathbf{x}_+ - \mathbf{x}_-) &= 2
\end{aligned}$$

After projecting such distance value over the normal vector $\hat{\mathbf{w}} = \frac{\mathbf{w}}{\|\mathbf{w}\|_2}$, we will obtain that the actual distance between the two points is $\frac{2}{\|\mathbf{w}\|_2}$, that is the quantity that we want to maximize. Given that

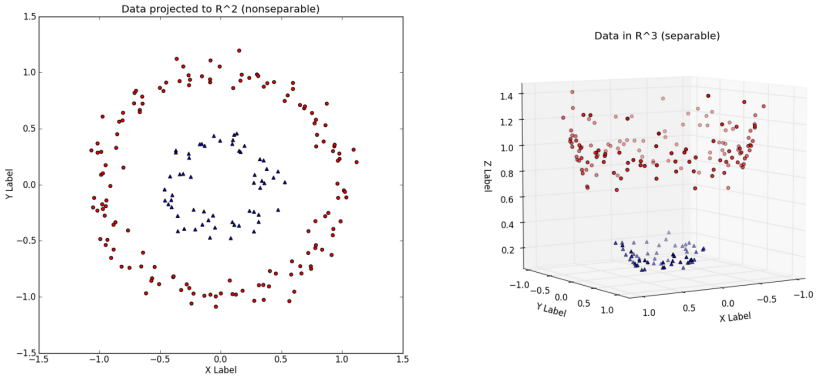


Figure 10: A circular concept: the blue (red) data points represent the positive (negative) examples. Whereas there is no hyperplane in the original $\mathbb{R}^2$ space separating the positive from the negative samples (left), there is one in the feature space obtained via Φ transformation (right) `https://towardsdatascience.com/understanding-the-kernel-trick-e0bc6112ef78`.

the maximization of such quantity corresponds to the minimization of $\frac{1}{2}\|\mathbf{w}\|_2^2$, we can also rewrite our maximization problem as the following minimization problem:

$$\begin{aligned} \min_{\mathbf{w},b} \quad & \frac{1}{2}\|\mathbf{w}\|_2^2 \\ \text{s.t.} \quad & \forall(\mathbf{x}, y) \in T_r.\ y(\mathbf{w}\mathbf{x} + b) \geq 1 \end{aligned} \tag{6}$$

Given that the solution of such problem requires knowing both what a Lagrangian Dual problem is and how to implement robust optimization problems, we refer to [2] for additional information regarding to Lagrangian Duals. After additional mathematical steps and replacing the dot product with the kernel function, we can rewrite such problem in the following form:

$$\begin{aligned} \max_{\alpha \in \mathbb{R}^m} \quad & \sum_{i=1}^{|T_r|} \alpha_i - \frac{1}{2} \sum_{(\mathbf{x}_i, y_i), (\mathbf{x}_j, y_j) \in T_R}^{m} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) \\ \text{s.t.} \quad & \forall \alpha_i \in \alpha . \alpha \geq 0 \wedge \sum_{(\mathbf{x}_i, y_i) \in T_r} \alpha_i y_i = 0 \end{aligned} \tag{7}$$

Feature Maps (*)

At this point, we observe that while all the previous data representations dealt with data points represented in some n-dimensional space, kernel functions allow us to compare disparate data representations. If we now could possibly represent any kernel as the dot product between two vector representations, it would imply that we might be able to represent each possible data structure as a vector embedding. Given the **feature map** function Φ performing such transformation into vectors, we can now represent our kernel function K as follows:

$$K(X, Y) \approx \Phi(X) \cdot \Phi(Y)$$

As an intuition on why such consideration is possible, consider the classification problem provided in Figure 10: we can find a hyperplane separating the elements that are within the unit circle in $\mathbb{R}^2$ from the ones that aren't if we use the following feature map $\Phi \colon \mathbb{R}^2 \to \mathbb{R}^3$[11]:

$$\Phi(\mathbf{x}) = (\mathbf{x}_1^2, \sqrt{2}\mathbf{x}_1\mathbf{x}_2, \mathbf{x}_2^2)$$

Now, we might ask ourselves how to obtain Φ when K is a similarity function over an arbitrary data representation: given that from a kernel we can always determine the distance function (see also Exercise 6), we can generate from it a dissimilarity matrix D. Now, we can determine an embedding in $\mathbb{R}^k$ with $k \ll |T_r|$ by learning a Multi-Dimensional Scaling matrix M providing the feature mapping $\Phi(i) = M_i$ [14].

Another interesting feature map is the one allowing to normalize the dataset so to get better classification results. In order to do so, we might normalize the samples in T_r by calculating the component-wise average and standard deviation, thus obtaining the average (standard deviation) vector $\vec{\mu}_{T_r}$ ($\vec{\sigma}_{T_r}$). If we now remember the normalization formula provided at page 4 and we remember that each vector dimension can be indeed represent one random variable, then given the component-wise vector product $\odot$ we can write the normalization feature map as follows:

$$\Phi_{\vec{\mu}_{T_r},\vec{\sigma}_{T_r}}(\mathbf{x}) := (\mathbf{x} - \vec{\mu}_{T_r}) \odot \frac{1}{\vec{\sigma}_{T_r}} \tag{8}$$

5.1.1 Radial Basis Function (or Gaussian) Kernel

In the previous optional subsubsection, we saw that K via Φ can map any data sample into a high-dimensional feature space via a non-necessarily-linear transformation, resulting into a hyperplane classification. If we would train the SVM machine over an arbitrary K without the Multi-Dimensional Scaling matrix trick, then state of the art kernel SVM training methods may take days or even weeks over big data training sets with just lower than 100 dimensions. One of the possible approaches to boost up the performances is to choose a convenient K allowing high dimensional scaling for an arbitrary dimension k. One popular kernel is the Radial Basis Function (or Gauissian) Kernel, which is defined as follows:

$$K_G(\mathbf{x}, \mathbf{x}') = e^{-\gamma\|\mathbf{x}-\mathbf{x}'\|^2}$$

Given that this model is easy to calibrate (we have just one weight γ) and it is robust to incremental data perturbations ($K_G(\mathbf{x}+\delta, \mathbf{x}'+\delta) = K_G(\mathbf{x}, \mathbf{x}')$), it becomes an ideal kernel function for off-the-shelf machine learning. Plus, it is also possible to boost up the performances by decomposing K via Taylor Expansion [3]:

$$\Phi_{G,k}(\mathbf{x}) = e^{-\gamma\|\mathbf{x}\|^2} \frac{1}{(2\gamma)^{k/2}\sqrt{k!}} \prod_{0\leq i\leq k} \mathbf{x}_j$$

we might also observe that such kernel function might help us from solving the XOR problem effectively, as we will see in the next section.

5.2 ν-Support Vector Machines

We might observe that, due to the potentially noisy nature of data, we might never get an hyperplane even after applying the kernel trick. In fact, even after mapping the data points into a feature space where the general problem might be linearly separable, noisy data might make some classes overlap, and so we cannot guarantee that any $\frac{2}{\|\mathbf{w}\|_2^2}$ boundary separates them. In these scenarios, we might introduce some non-negative slack variables $\Xi = \{\, \xi_i \,\}_{1\leq i\leq |T_r|}$, thus relaxing Equation 6 on page 40 as follows:

$$\begin{aligned} \min_{\mathbf{w},b} \quad & \frac{1}{2}\|\mathbf{w}\|_2^2 \\ \text{s.t.} \quad & \forall(\mathbf{x}_i, y_i) \in T_r.\ y(\mathbf{w}\mathbf{x} + b) \geq 1 - \xi_i \end{aligned}$$

Now, we can also relax the dual maximization problem presented in Equation 7: instead of forcing the algorithm to learn a feature map Φ function for all the possible elements in T_r, we can provide a lower bound ν on the number of examples in T_r that will be associated to a support vector and that lie on the wrong side of the plane. The previously mentioned equation becomes:

$$\begin{aligned} \max_{\alpha\in\mathbb{R}^m} \quad & -\frac{1}{2} \sum_{(\mathbf{x}_i,y_i),(\mathbf{x}_j,y_j)\in T_R}^{m} \alpha_i\alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) \\ \text{s.t.} \quad & \forall\alpha_i \in \alpha.0 \leq \alpha \geq \frac{1}{|T_r|} \wedge \sum_{(\mathbf{x}_i,y_i)\in T_r} \alpha_i y_i = 0 \wedge \sum_{i=1}^{|T_r|} \alpha_i \geq \nu \end{aligned}$$

At this point, we can now represent the regression and hypothesis as follows:

$$\tilde{h}(\mathbf{x}) = \sum_{(\mathbf{x}',y)\in T_r} \alpha_i y_i K(\mathbf{x}, \mathbf{x}') + b \qquad h(\mathbf{x}) = \text{sign}(\tilde{h}(\mathbf{x}))$$

The XOR Problem We nowhave all the right strategies for effectively learning the XOR function over ν-SVM. We can also try to code this task using DLib: given our that in this simplistic situation the training set is E because our function is known and not implicit within the data (Line 3 on the following source code), then we can map the boolean false values to -1 and the boolean true values to 1 (Line 27). At this stage, the `samples` dataset will be the following:

$$E = \{ ((0,0),-1),\ ((1,0),1),\ ((1,1),1),\ ((0,1),-1) \}$$

We now need to normalize our dataset. After computing the average and the standard deviation for both the x and the y components, we will obtain the following normalization feature map: $\Phi_{\vec{\mu}_E,\vec{\sigma}_E}(\mathbf{x}) = (\mathbf{x} - (0.5, 0.5)) \odot (\sqrt{3}, \sqrt{3})$. This feature map is implicitly computed in DLib via a vector normalizer (Line 47) which, after some training (Line 49), can be now applied to the original sampled data points (Line 52). After shuffling the training dataset and determining the maximum admissible value for ν (Line 70), we can naïvely search the parameters' space for γ and ν (Line 81) to to minimize the loss function over the data (Line 109).

```
1  bool bool_and(int l, int r) { return (l && r); }
2  bool bool_or(int l, int r) { return (l || r); }
3  bool bool_xor(int l, int r) { return ((!l != !r)); }
4
5  void train_binary_svm(bool (*binary_function)(int, int)) {
6      // The svm functions use column vectors to contain a lot of the data on which they
7      // operate. So the first thing we do here is declare a convenient typedef.
8
9      // This typedef declares a matrix with 2 rows and 1 column. It will be the object that
10     // contains each of our 2 dimensional samples. (Note that if you wanted more than 2
11     // features in this vector you can simply change the 2 to something else. Or if you
12     // don't know how many features you want until runtime then you can put a 0 here and
13     // use the matrix.set_size() member function)
14     typedef dlib::matrix<double, 2, 1> sample_type;
15
16     // This is a typedef for the type of kernel we are going to use in this example. In
17     // this case I have selected the radial basis kernel that can operate on our 2D
18     // sample_type objects
19     // https://en.wikipedia.org/wiki/Radial_basis_function_kernel
20     typedef dlib::radial_basis_kernel<sample_type> kernel_type;
21
22     // Now we make objects to contain our samples and their respective labels.
23     std::vector<sample_type> samples;
24     std::vector<double> labels;
25
26     // Now let's put some data into our samples and labels objects.
27     for (int r = 0; r <= 1; ++r)
28     {
29         for (int c = 0; c <= 1; ++c)
30         {
31             sample_type samp;
32             samp(0) = r;
33             samp(1) = c;
34             samples.push_back(samp);
35             labels.push_back(binary_function(r, c) ? 1 : -1);
36         }
37     }
38
39
40
41
42     // Here we normalize all the samples by subtracting their mean and dividing by their
43     // standard deviation. This is generally a good idea since it often heads off
44     // numerical stability problems and also prevents one large feature from smothering
45     // others. Doing this doesn't matter much in this example so I'm just doing this here
46     // so you can see an easy way to accomplish this with the library.
47     dlib::vector_normalizer<sample_type> normalizer;
48     // let the normalizer learn the mean and standard deviation of the samples
49     normalizer.train(samples);
50     // now normalize each sample
51     for (unsigned long i = 0; i < samples.size(); ++i)
52         samples[i] = normalizer(samples[i]);
53
54
55     // Now that we have some data we want to train on it. However, there are two
```

```
    // parameters to the training. These are the nu and gamma parameters. Our choice for
    // these parameters will influence how good the resulting decision function is. To
    // test how good a particular choice of these parameters is we can use the
    // cross_validate_trainer() function to perform n-fold cross validation on our training
    // data. However, there is a problem with the way we have sampled our distribution
    // above. The problem is that there is a definite ordering to the samples. That is,
    // the first half of the samples look like they are from a different distribution than
    // the second half. This would screw up the cross validation process but we can fix it
    // by randomizing the order of the samples with the following function call.
    randomize_samples(samples, labels);


    // The nu parameter has a maximum value that is dependent on the ratio of the +1 to -1
    // labels in the training data. This function finds that value.
    const double max_nu = dlib::maximum_nu(labels);

    // here we make an instance of the svm_nu_trainer object that uses our kernel type.
    dlib::svm_nu_trainer<kernel_type> trainer;

    // Now we loop over some different nu and gamma values to see how good they are.
    // model_selection_ex.cpp from the dlib library provides a different approach to pick the best
        ↪ gamma and nu values.
    std::cout << "doing␣cross␣validation" << std::endl;
    double candidateGamma = 0.0, candidateNu = 0.0;
    double scoresforCandidates = std::numeric_limits<double>::max();

    for (double gamma = 0.00001; gamma <= 1; gamma *= 5)
    {
        for (double nu = 0.00001; nu < max_nu; nu *= 5)
        {
            // Here we are making an instance of the normalized_function object. This object
            // provides a convenient way to store the vector normalization information along with
            // the decision function we are going to learn.

            // Set the kernel gamma and nu values from the iteration loop
            typedef dlib::decision_function<kernel_type> dec_funct_type;
            typedef dlib::normalized_function<dec_funct_type> funct_type;
            funct_type learned_function;
            trainer.set_kernel(kernel_type(gamma));
            trainer.set_nu(nu);

            // Extract the learned function
            learned_function.normalizer = normalizer; // save normalization information
            learned_function.function = trainer.train(samples, labels); // perform the actual SVM
                ↪ training and save the results

            // Evaluating the mean square error between the predictions and the actual value
            double distance = 0.0;
            for (size_t i = 0, N = std::min(samples.size(), labels.size()); i<N; i++) {
                distance += std::pow(learned_function(samples[i]) - labels[i], 2);
            }
            distance = std::sqrt(distance);
            std::cout << "␣gamma:␣" << gamma << "␣nu:␣" << nu << "␣score:␣" << distance << std::endl;

            // Picking for gamma and nu the best value, that is the one minimizing the distance
            if (distance < scoresforCandidates) {
                candidateGamma = gamma;
                candidateNu = nu;
                scoresforCandidates = distance;
                std::cout << "␣~~~␣New␣candidate␣!␣~~~~␣" << std::endl;
            }
        }
    }


    // From looking at the output of the above loop it turns out that a good value for nu
    // and gamma for this problem is 0.15625 for both. So that is what we will use.

    // Now we train on the full set of data and obtain the resulting decision function. We
    // use the value of 0.15625 for nu and gamma. The decision function will return values
    // >= 0 for samples it predicts are in the +1 class and numbers < 0 for samples it
    // predicts to be in the -1 class.
    trainer.set_kernel(kernel_type(candidateGamma));//candidateGamma
```

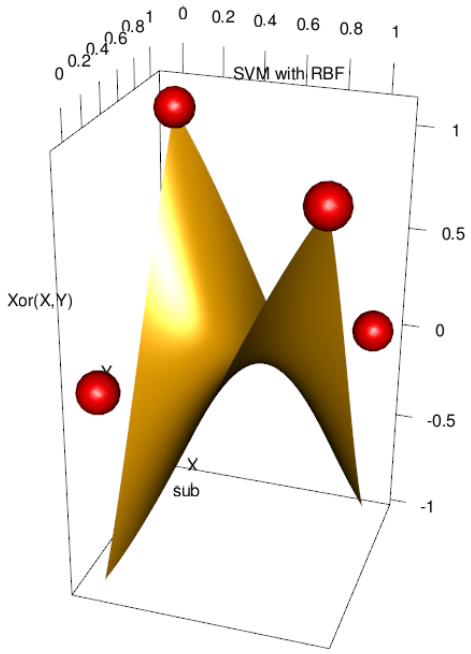


Figure 11: ν-SVM separation of the XOR function using a RBF Kernel

```
127     trainer.set_nu(candidateNu);
128     typedef dlib::decision_function<kernel_type> dec_funct_type;
129     typedef dlib::normalized_function<dec_funct_type> funct_type;
130
131     // Here we are making an instance of the normalized_function object. This object
132     // provides a convenient way to store the vector normalization information along with
133     // the decision function we are going to learn.
134     funct_type learned_function;
135     learned_function.normalizer = normalizer; // save normalization information
136     learned_function.function = trainer.train(samples, labels); // perform the actual SVM training and
            ↪ save the results
137
138     // print out the number of support vectors in the resulting decision function
139     std::cout << "\nnumber␣of␣support␣vectors␣in␣our␣learned_function␣is␣"
140               << learned_function.function.basis_vectors.size() << std::endl;
141
142     // Now let's try this decision_function on some samples we haven't seen before.
143     sample_type sample;
144
145     sample(0) = 0;
146     sample(1) = 0;
147     std::cout << "(0,0):␣the␣classifier␣output␣is␣" << learned_function(sample) << std::endl;
148
149     sample(0) = 0;
150     sample(1) = 1;
151     std::cout << "(0,1):␣the␣classifier␣output␣is␣" << learned_function(sample) << std::endl;
152
153     sample(0) = 1;
154     sample(1) = 0;
155     std::cout << "(1,0):␣the␣classifier␣output␣is␣" << learned_function(sample) << std::endl;
156
157     sample(0) = 1;
158     sample(1) = 1;
159     std::cout << "(1,1):␣the␣classifier␣output␣is␣" << learned_function(sample) << std::endl;
160 }
```

Figure 11 provides the outcome of the previous source code where the best model parameters were $\gamma = 10 \cdot 0.5^6$ and $\nu = 0.5^4 \cdot 10^{-1}$. After observing that DLib represents $\tilde{h}$ as:

$$\tilde{h}(\mathbf{x}) = \sum_{(\mathbf{x}',y) \in \Phi_{\vec{\mu}_E, \vec{\sigma}_E}(T_r)} \tilde{\alpha}_i K(\Phi_{\vec{\mu}_E, \vec{\sigma}_E}(\mathbf{x}), \mathbf{x}')$$

where $\tilde{\alpha}_i := \alpha_i y_i$, then the inferred $\tilde{\alpha}$ values by the trainer are:

$$\tilde{\alpha} = \{ -7.14094, 7.14094, -7.14094, 7.14094 \}$$

As we might observe, the model interpretation is simpler than the one from Multilayer Neural Networks (we need to plot one single function instead of three) but more obscure than the outcome of Binary Decision Trees (the decision is not described as a nested chain of `if...then ...else`). Given that the XOR function was successfully trained, we can now try to learn the classification from the StarCraft II dataset using a one-versus-one approach. As we might observe from the following code, such classifier is pretty simple, and the parameter tuning is all hidden by the `cross_validate_multiclass_trainer`, which plots a confusion matrix to show the trained model's precision. Last, we obtain the ensemble hypothesis from `df`, and finally print the model's precision over the testing data.

```
void multi_svm_train(const DLib_Splits& splits, const unsigned numberClasses, const size_t input_size)
    ↪ {

    // The main object in this example program is the one_vs_one_trainer. It is essentially
    // a container class for regular binary classifier trainer objects. In particular, it
    // uses the any_trainer object to store any kind of trainer object that implements a
    // .train(samples,labels) function which returns some kind of learned decision function.
    // It uses these binary classifiers to construct a voting multiclass classifier. If
    // there are N classes then it trains N*(N-1)/2 binary classifiers, one for each pair of
    // labels, which then vote on the label of a sample.
    //
    // In this example program we will work with a one_vs_one_trainer object which stores any
    // kind of trainer that uses our sample_type samples.
    typedef dlib::one_vs_one_trainer<dlib::any_trainer<dlib::matrix<double>>> ovo_trainer;

    // Finally, make the one_vs_one_trainer.
    ovo_trainer trainer;

    // Create the binary trainer
    typedef dlib::radial_basis_kernel<dlib::matrix<double>> rbf_kernel;
    dlib::krr_trainer<rbf_kernel> rbf_trainer;

    // Now tell the one_vs_one_trainer that, by default, it should use the rbf_trainer
    // to solve the individual binary classification subproblems.
    trainer.set_trainer(rbf_trainer);


    std::cout << "cross␣validation:␣\n" << cross_validate_multiclass_trainer(trainer, splits.
        ↪ training_input, splits.training_labels, 5) << std::endl;

    // Next, if you wanted to obtain the decision rule learned by a one_vs_one_trainer you
    // would store it into a one_vs_one_decision_function.
    dlib::one_vs_one_decision_function<ovo_trainer> df = trainer.train(splits.training_input, splits.
        ↪ training_labels);

    size_t N = std::min(splits.testing_label_vector.size(), splits.testing_input.size());
    double distance = 0.0;
    for (size_t i = 0; i<N; i++) {
        double ithDistance = 0.0;
        double prediction = df(splits.testing_input[i]);
        const auto& expected = splits.testing_label_vector[i];
        ithDistance = std::abs(prediction - (double)splits.testing_labels[i]);
        // normalize the distance
        ithDistance = ithDistance / (ithDistance+1.0);
        distance += ithDistance;
    }
    // distance normalization
    distance = (distance) / ((double)N);
    // invert the distance, so to obtain the precision
    double precision = 1.0 - distance;

    std::cout << "Model␣precision␣over␣the␣testing␣data:␣" << precision << std::endl;
}
```

6 Further work

1. Show why a binary classification is not a necessary and sufficient conditions for reducing a general reinforcement learning algorithm into a supervised learning problem.

2. Investigate k-fold cross-validation techniques[20] in order to cope with overfitting for large training test sets.

3. The binary decision trees in these tutorials reduced the multiclass classification problem into a binary classification problem by using a **one-versus-all** approach. Change the implementation so that multiclass classification is done via a **one-versus-one** approach.

4. Binary classification trees can be also generalized to discriminate multiple different classes: generalize the implementation of the present code by implementing the generalized version of the impurity function available at [12].

5. Investigate the usage of convolutional neural networks (CNN) in DLib, and detect if there is a spot where multilayer neural networks are used for the classification outcome. Are there any differences between SVM's kernels and CNN's ones? how do max-pooling and convolution layers simplify the backpropagation algorithm's definition?

6. Walking on the footsteps provided at Equation 5, it is trivial to prove that we can define an arbitrary distance function from a similarity measure as follows:

$$d_K(X,Y) := \sqrt{K(X,X) + K(Y,Y) - 2K(X,Y)}$$

References

[1] Tom B. Brown, Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. Adversarial patch. *CoRR*, abs/1712.09665, 2017. URL: `http://arxiv.org/abs/1712.09665`, `arXiv:1712.09665`.

[2] Pai-Hsuen Chen, Chih-Jen Lin, and Bernhard Schölkopf. A tutorial on ν-support vector machines. *Applied Stochastic Modes in Business and Industry*, 21(2):111–136, 2005.

[3] Andrew Cotter, Joseph Keshet, and Nathan Srebro. Explicit approximations of the gaussian kernel. *CoRR*, abs/1109.4603, 2011. URL: `http://arxiv.org/abs/1109.4603`, `arXiv:1109.4603`.

[4] Bruno De Finetti. *Theory of Probability: A Critical Introductory Treatment*. Wiley, USA, 2017.

[5] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 1126–1135. JMLR.org, 2017.

[6] Joseph F. Hair, Rolph E. Anderson, Ronald L. Tatham, and William C. Black. *Multivariate Data Analysis (4th Ed.): With Readings*. Prentice-Hall, Inc., USA, 1995.

[7] Sandy Huang, Nicolas Papernot, Ian Goodfellow, Yan Duan, and Pieter Abbeel. Adversarial attacks on neural network policies, 2017. `arXiv:1702.02284`.

[8] Open Data Structures (in C++). Pat morin, 2011. URL: `https://opendatastructures.org/`.

[9] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., USA, 1 edition, 1997.

[10] David E. Moriarty, Alan C. Schultz, and John J. Grefenstette. Evolutionary algorithms for reinforcement learning. *J. Artif. Int. Res.*, 11(1):241–276, July 1999.

[11] Luc De Raedt. *Logical and Relational Learning: From ILP to MRDM (Cognitive Technologies)*. Springer-Verlag, Berlin, Heidelberg, 2008.

[20] `https://en.wikipedia.org/wiki/Cross-validation_(statistics)`

[12] Laura Elena Raileanu and Kilian Stoffel. Theoretical comparison between the gini index and information gain criteria. *Annals of Mathematics and Artificial Intelligence*, 41(1):77–93, May 2004. doi:10.1023/B:AMAI.0000018580.96245.c6.

[13] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach*. Pearson, 3 edition, 2009.

[14] Josh Wills, Sameer Agarwal, David Kriegman, and Serge Belongie. Toward a perceptual space for gloss. *ACM Trans. Graph.*, 28(4), September 2009. doi:10.1145/1559755.1559760.