

# Python Notes

## Contents

<b>1</b>	<b>Syntax &amp; Fundamentals</b>	<b>12</b>
1.1	Methods . . . . .	12
1.1.1	__getitem__ . . . . .	12
1.1.2	__repr__ . . . . .	12
1.1.3	__abs__ . . . . .	12
1.1.4	__add__ . . . . .	12
1.1.5	__mul__ . . . . .	12
1.1.6	__and__ . . . . .	12
1.1.7	__len__ . . . . .	12
1.1.8	Emulating numeric types . . . . .	12
1.1.9	String representation of objects . . . . .	13
1.1.10	Boolean value of an object . . . . .	13
1.1.11	Implementing collections . . . . .	13
1.2	Formatting . . . . .	14
1.2.1	f" . . . . .	14
1.2.2	.format() . . . . .	14
1.3	Functions . . . . .	14
1.3.1	Declaration structure . . . . .	14
1.3.2	lambda method . . . . .	14
1.3.3	Decorators . . . . .	15
1.4	Classes . . . . .	15
1.4.1	__init__ . . . . .	15
1.4.2	__call__ . . . . .	15
<b>2</b>	<b>Major Modules</b>	<b>16</b>
2.1	Itertools . . . . .	16
2.1.1	Import . . . . .	16
2.1.2	combinations . . . . .	16
2.1.3	permutations . . . . .	16
2.2	OS . . . . .	16
2.2.1	Import . . . . .	16
2.2.2	Execute command in prompt . . . . .	16
2.2.3	Get current path . . . . .	17
2.2.4	Join paths . . . . .	17
2.3	Requests . . . . .	17
2.3.1	Import . . . . .	17
2.3.2	Get request to url . . . . .	17
2.4	JSON . . . . .	17
2.4.1	Save json file . . . . .	17
2.4.2	Load json file . . . . .	17
2.5	Pickle . . . . .	17
2.5.1	Dump . . . . .	17
2.5.2	Load . . . . .	18
2.6	Matplotlib . . . . .	18
2.6.1	Import . . . . .	18
2.6.2	Save figure . . . . .	18
2.6.3	Multiple Graphs . . . . .	18
2.6.4	Plot dataframe column in graph . . . . .	19
2.6.5	Buy and sell markers . . . . .	19

2.6.6	Modern graph	20
2.6.7	3D Scatter Plot	21
2.6.8	Graph settings	23
2.6.9	Line plot	23
2.6.10	Histogram	23
2.6.11	Scatter plot	23
2.6.12	Heatmap	24
2.7	Seaborn	24
2.7.1	Pair plot	24
2.8	SciPy	25
2.9	Numpy	25
2.9.1	Import	25
2.9.2	Operators	25
2.9.3	Vertical slicing	25
2.9.4	np.astype(type)	26
2.9.5	np.trunc()	26
2.9.6	np.choice(sample, size, replace)	26
2.9.7	np.mean()	26
2.9.8	np.unique()	26
2.9.9	np.where()	27
2.9.10	np.eye()	27
2.9.11	np.reshape(x, y)	27
2.9.12	np.reshape(-1, 1)	27
2.9.13	np.reshape(1, -1)	27
2.9.14	..c	28
2.9.15	np.linspace(start, end, steps)	28
2.9.16	np.cov(x, y)	28
2.9.17	np.var(x, y, ddof=k)	29
2.9.18	np.cumsum()	29
2.9.19	np.random.randn(n)	29
2.9.20	np.random.standard_normal((rows, columns))	29
2.9.21	np.eye()	29
2.9.22	np.repeat(array, iterations)	30
2.9.23	np.convolve()	30
2.9.24	np.ones()	30
2.9.25	np.linalg.inv(A)	30
2.9.26	np.linalg.pinv(A)	31
2.10	Pandas	31
2.10.1	Import	31
2.10.2	Axis representation	31
2.10.3	Load dataframe	31
2.10.4	Create dataframe	31
2.10.5	Create dataframe with dict	31
2.10.6	Get dataframe shape	31
2.10.7	Iterate through rows	31
2.10.8	Change column datatype	32
2.10.9	Get rows specific index	32
2.10.10	Shuffle dataframe	32
2.10.11	Confusion matrix	33
2.10.12	Load from the web	33
2.10.13	.loc(row, column)	33
2.10.14	.iloc()	33
2.10.15	.values	33
2.10.16	.merge(df1, df2, left_on, right_on, how)	34
2.10.17	.drop([column1, column2, ...])	34
2.10.18	.info()	34
2.10.19	.describe()	35
2.10.20	.diff()	35
2.10.21	.sort_values	35
2.10.22	.shuffle(df, random_state)	35
2.10.23	.concat([df1, df2])	35

2.10.24.sample(frac)	35
2.10.25.shift()	36
2.10.26.count()	36
2.10.27.head()	37
2.10.28.ffill()	37
2.10.29.get_dummies()	37
2.10.30.set_index()	38
2.10.31.reset_index()	38
2.10.32.pct_change()	38
2.10.33.corr()	39
2.10.34.rolling()	40
2.10.35.isin()	40
2.10.36.at()	41
2.10.37.merge()	41
2.10.38.sort_values()	42
2.10.39.iloc()	42
2.10.40.ewm() (Exponentially Weighted Moving Average)	43
2.10.41.MACD	44
2.11.YFinance	44
<b>3 Major APIs</b>	<b>45</b>
3.1.Kraken	45
3.1.1.Required modules	45
3.1.2.Pair info	45
3.1.3.OHLC data	45
3.1.4.Historical dataframe	45
3.1.5.Pair price dataframe	45
3.1.6.Status code	46
<b>4 Algorithms</b>	<b>46</b>
4.1.Hashing	46
4.1.1.Hash function	46
4.1.2.Hash function characteristics	46
4.1.3.Hash buckets	46
<b>5 Finance</b>	<b>46</b>
5.1.Drawdown	46
5.2.Technical Indicators	47
5.2.1.SMA	47
5.2.2.EMA	47
5.2.3.ROC (Rate of Change)	47
5.2.4.BBands (Bollinger Bands)	47
5.2.5.RSI	48
5.2.6.MOM (MOMentum)	48
5.2.7.STOK & STOD	48
<b>6 Data Science</b>	<b>49</b>
6.1.Lifecycle	49
6.1.1.Stages	49
6.1.2.Data scope	49
6.1.3.Target population, access frame, sample	49
6.1.4.Protocols	49
6.1.5.Accuracy	49
6.2.Bias	49
6.2.1.Coverage bias	49
6.2.2.Selection bias	49
6.2.3.Nonresponse bias	49
6.2.4.Measurement bias	49
6.3.Variations	50
6.3.1.Sampling variation	50
6.3.2.Assignment variation	50

6.3.3	Measurement variation . . . . .	50
6.4	Sampling . . . . .	50
6.4.1	Stratified sampling . . . . .	50
6.4.2	Cluster sampling . . . . .	50
6.5	Distributions . . . . .	50
6.5.1	Hypergeometric distribution . . . . .	50
6.5.2	Multivariate Hypergeometric distribution . . . . .	50
6.5.3	Binomial distribution . . . . .	50
<b>7</b>	<b>Machine Learning</b>	<b>50</b>
7.1	Lingo . . . . .	50
7.2	Bias . . . . .	51
7.2.1	Data snooping bias . . . . .	51
7.2.2	Selection bias . . . . .	51
7.2.3	Sampling bias . . . . .	51
7.3	Classification vs. Regression . . . . .	51
7.3.1	Classification . . . . .	51
7.3.2	Multiclass classification . . . . .	51
7.3.3	Singlelabel vs. Multilabel . . . . .	51
7.3.4	Regression . . . . .	51
7.4	Supervised vs. Unsupervised . . . . .	51
7.4.1	Supervised . . . . .	51
7.4.2	Unsupervised . . . . .	51
7.5	Supervised Models . . . . .	51
7.5.1	Logistic regression . . . . .	52
7.5.2	LDA (Linear Discriminant Analysis) . . . . .	52
7.5.3	Polynomial regression . . . . .	52
7.5.4	DecisionTreeRegressor . . . . .	52
7.6	Semisupervised Models . . . . .	52
7.6.1	deep belief networks (DBNs) . . . . .	52
7.7	Unsupervised Models . . . . .	53
7.7.1	K-means . . . . .	53
7.7.2	Hierarchical Clustering . . . . .	53
7.7.3	K-Means mini-batches . . . . .	54
7.7.4	Agglomerative Clustering . . . . .	54
7.7.5	AffinityPropagation . . . . .	55
7.7.6	DBSCAN (Density-Based Spatial Clustering of Applications with Noise) . . . . .	56
7.7.7	HDBSCAN (Hierarchical DBSCAN) . . . . .	56
7.7.8	Agglomeration Clustering . . . . .	56
7.7.9	BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) . . . . .	56
7.7.10	Mean-Shift . . . . .	56
7.7.11	Affinity Propagation . . . . .	56
7.7.12	Spectral Clustering . . . . .	56
7.7.13	GMM (Gaussian Mixtures Model) . . . . .	57
7.7.14	Fast-MCD (Fast-Minimum Covariance Determinant) . . . . .	57
7.7.15	Isolation forest . . . . .	57
7.7.16	LOF (Local Outlier Factor) . . . . .	57
7.7.17	One-Class SVM . . . . .	57
7.7.18	Hierarchical Cluster Analysis (HCA) . . . . .	57
7.7.19	One-class SVM . . . . .	57
7.7.20	Isolation Forest . . . . .	57
7.7.21	Apriori . . . . .	57
7.7.22	Eclat . . . . .	57
7.8	Ensemble Learning . . . . .	58
7.8.1	Voting classifiers . . . . .	58
7.8.2	Bagging (Bootstrap Aggregating) . . . . .	58
7.8.3	Boosting . . . . .	58
7.8.4	AdaBoost (Adaptive Boosting) . . . . .	58
7.8.5	Gradient Boosting . . . . .	58
7.8.6	XGBoost . . . . .	58
7.8.7	Pasting . . . . .	58

7.8.8	Stacking (Stacked Generalization)	59
7.8.9	Out-of-Bag (oob)	59
7.8.10	Random patches	59
7.8.11	Random subspaces	59
7.9	Ensemble Models	59
7.9.1	GBRT (Gradient Boosted Regression Trees)	59
7.9.2	Voting Classifier	59
7.9.3	GDM (Gradient Boosting Machine)	59
7.9.4	AdaBoost (Adaptive Boosting)	59
7.10	Batch Learning vs. Online Learning	60
7.11	Instance-Based vs. Model-Based Learning	60
7.12	Performance Evaluation	60
7.12.1	No Free Lunch (NFL) theorem	60
7.12.2	Bias	60
7.12.3	Variance	60
7.12.4	Bias-variance trade-off	60
7.12.5	Testing and validating	60
7.12.6	Cross-validation	60
7.12.7	Irreducible error	61
7.12.8	Data drift	61
7.12.9	Cost/Loss Function	61
7.12.10	Evaluation Metrics	61
7.13	Cost/Loss Function Algorithms	61
7.13.1	MSE (Mean Squared Error)	61
7.13.2	RMSE (Root Mean Squared Error)	61
7.13.3	MAE (Mean Absolute Error)	62
7.13.4	Huber loss	62
7.14	Distance Measurement	62
7.14.1	Euclidean	62
7.14.2	Manhattan	62
7.14.3	Minkowski	62
7.14.4	Cosine similarity	62
7.15	Optimization Algorithms	63
7.15.1	GD (Gradient Descent)	63
7.16	Evaluation Metrics	63
7.16.1	Confusion matrix	63
7.16.2	Accuracy	64
7.16.3	Precision (sensitivity)	64
7.16.4	Recall (specificity)	65
7.16.5	F1	65
7.16.6	Classification report	65
7.16.7	R-squared	66
7.16.8	adjusted R-squared	66
7.16.9	ROC curve (Receiver Operating Characteristic curve)	66
7.16.10	AUC (Area Under Curve)	66
7.16.11	OOB (Out-of-Bag)	67
7.16.12	mAP (mean Average Precision)	67
7.16.13	K-Means scoring	67
7.17	Overfitting	67
7.17.1	Regularization	67
7.18	Regularization Algorithms	67
7.18.1	Lasso (L1)	68
7.18.2	Ridge (L2)	68
7.18.3	ElasticNet	69
7.19	Hyperparameters Tuning	69
7.19.1	Grid search	69
7.19.2	Random search	69
7.19.3	Bayesian optimization	69
7.19.4	Genetic algorithms	69
7.20	Distributed Training	69
7.20.1	Map reduce	69

7.21	Feature Engineering . . . . .	69
7.21.1	Feature selection . . . . .	69
7.21.2	Feature extraction . . . . .	70
7.21.3	Feature cross . . . . .	70
7.21.4	LabelEncoder . . . . .	70
7.21.5	One-Hot encoding . . . . .	70
7.21.6	Data augmentation . . . . .	70
7.21.7	Dimensionality reduction . . . . .	70
7.21.8	Data scaling . . . . .	71
7.22	Dimensionality Reduction Algorithms . . . . .	71
7.22.1	Principal Component Analysis (PCA) . . . . .	71
7.22.2	IPCA (Incremental PCA) . . . . .	72
7.22.3	t-distributed Stochastic Neighbor Embedding (t-SNE) . . . . .	72
7.22.4	kernel PCA . . . . .	72
7.22.5	sparse PCA . . . . .	72
7.22.6	ED (Eigen Decomposition) . . . . .	73
7.22.7	SVD (Singular Value Decomposition) . . . . .	73
7.22.8	TSVD (Truncated SVD) . . . . .	73
7.22.9	LLE (Locally Linear Embedding) . . . . .	73
7.22.10	K-Means . . . . .	73
7.22.11	t-SNE (t-Distributed Stochastic Neighbor Embedding) . . . . .	73
7.22.12	LLE (Locally Linear Embedding) . . . . .	74
7.22.13	Random projection . . . . .	74
7.22.14	Manifold learning . . . . .	74
7.22.15	Isomap . . . . .	74
7.22.16	Other unsupervised algorithms . . . . .	74
7.23	Data Scaling Algorithms . . . . .	74
7.23.1	Standard scaling . . . . .	74
7.23.2	MinMaxScaler . . . . .	74
7.23.3	Clipping . . . . .	74
7.23.4	Z-score normalization . . . . .	75
7.23.5	LRN (Local Response Normalization) . . . . .	75
7.24	Feature Selection Algorithms . . . . .	75
7.24.1	Chi-squared $\chi^2$ . . . . .	75
7.25	Embedding . . . . .	75
7.25.1	Word embedding . . . . .	75
7.25.2	Word2Vec . . . . .	75
7.26	Cascade . . . . .	75
7.27	Rebalancing . . . . .	75
7.27.1	Downsampling . . . . .	75
7.27.2	Upsampling . . . . .	75
7.28	Model Explainability . . . . .	76
7.28.1	Attribution values . . . . .	76
7.29	Shap . . . . .	76
7.29.1	Feature importance . . . . .	76
7.30	TPOT . . . . .	76
7.31	Resilient Serving . . . . .	77
7.32	Distributed Learning . . . . .	77
7.32.1	Synchronous training . . . . .	77
7.32.2	Asynchronous training . . . . .	77
7.33	Natural Language Processing . . . . .	77
7.33.1	Vectorization . . . . .	77
7.33.2	CountVectorizer . . . . .	77
7.33.3	HashingVectorizer . . . . .	77
7.33.4	TfidfVectorizer . . . . .	77
7.33.5	Stemming . . . . .	77
7.33.6	Lemmatizing . . . . .	78
7.33.7	Bag of words . . . . .	78
7.34	Recommender Systems . . . . .	78
7.34.1	Popularity based systems . . . . .	78
7.34.2	Collaborative systems . . . . .	78

7.34.3	Content-based systems . . . . .	78
7.35	Scikit-learn . . . . .	78
7.35.1	.train_test_split() . . . . .	78
7.35.2	GridSearchCV (GridSearch Cross Validation) . . . . .	78
7.35.3	KFold . . . . .	79
7.35.4	.score() . . . . .	79
7.35.5	cross_val_score() . . . . .	79
7.35.6	.summary() . . . . .	79
7.35.7	accuracy_score(actual, predicted) . . . . .	80
7.35.8	make_classification() . . . . .	80
7.35.9	Feature importance bar chart . . . . .	81
7.36	TensorFlow . . . . .	81
7.37	Keras . . . . .	81
7.37.1	Convolutional layer . . . . .	81
7.38	Generate Random Data . . . . .	81
7.38.1	Blobs . . . . .	81
7.38.2	Moons . . . . .	82
7.38.3	Regression . . . . .	82
7.39	Linear regression . . . . .	83
7.39.1	OLS (Ordinary Least Squares) . . . . .	83
7.40	KNN (K-Nearest Neighbors) . . . . .	84
7.40.1	Regressor . . . . .	84
7.40.2	Classifier . . . . .	85
7.41	K-means . . . . .	85
7.41.1	Description . . . . .	85
7.41.2	Fit clusters . . . . .	85
7.41.3	Cluster centers . . . . .	85
7.41.4	Elbow method for number of clusters . . . . .	85
7.41.5	Inertia calculation . . . . .	86
7.42	RNC (Radius Neighbors Classifier) . . . . .	86
7.42.1	Module usecase . . . . .	86
7.42.2	Predict class probability . . . . .	86
7.43	Agglomerative Clustering . . . . .	87
7.43.1	Silhouette score . . . . .	87
7.44	DBSCAN . . . . .	87
7.44.1	Description . . . . .	87
7.44.2	Fit . . . . .	87
7.45	Decision trees . . . . .	87
7.45.1	DTR (Decision Tree Regressor) . . . . .	88
7.45.2	DTC (Decision Tree Classifier) . . . . .	88
7.45.3	CART (Classification and Regression Trees) . . . . .	88
7.45.4	Gini measure of impurity . . . . .	88
7.46	Random forest . . . . .	90
7.46.1	Description . . . . .	90
7.46.2	RandomForestRegressor . . . . .	91
7.46.3	RandomForestClassifier . . . . .	91
7.47	Extra trees (Extremely Randomized Trees) . . . . .	91
7.48	GBDT (Gradient Boosting Decision Trees) . . . . .	92
7.48.1	Gradient Boosting Machines (GBM) . . . . .	92
7.48.2	AdaBoost (Adaptive Boosting) . . . . .	92
7.48.3	SGB (Stochastic Gradient Boosting) . . . . .	92
7.49	GBR (Gradient Boosting Regressor) . . . . .	92
7.50	SVM (Support Vector Machines) . . . . .	93
7.50.1	Description . . . . .	93
7.50.2	Kernel tricks . . . . .	95
7.50.3	Linear kernel . . . . .	95
7.50.4	Polynomial kernel . . . . .	95
7.50.5	Gaussian kernel . . . . .	95
7.50.6	Gaussian RBF kernel (Gaussian Radial Basis Function kernel) . . . . .	95
7.50.7	RBF kernel . . . . .	95
7.50.8	Sigmoid kernel . . . . .	100

7.50.9	SVR (Support Vector Regression)	100
7.50.10	SVC (Support Vector Classification)	100
7.50.11	SMV (Soft Margin Classification)	101
7.50.12	Polynomial SVM	101
7.51	Naive Bayes	104
7.52	Deep Learning	104
7.52.1	Whazit?	104
7.53	ANN (Artificial Neural Networks)	104
7.53.1	Description	104
7.53.2	Activation functions	104
7.53.3	Optimizers	104
7.53.4	Weights and biases	104
7.53.5	Backpropagation	104
7.53.6	Neurons weights	105
7.53.7	Weight-initialisation	105
7.53.8	Training loop	105
7.53.9	Dropout	105
7.53.10	Learning rate	105
7.53.11	Vanishing vs. exploding gradient	105
7.53.12	Gradient clipping	105
7.53.13	Transfer learning	105
7.53.14	Sparse models	105
7.53.15	Loss functions	106
7.54	Activation functions	106
7.54.1	ReLU (Rectified Linear Unit)	106
7.54.2	RReLU (Randomized leaky ReLU)	106
7.54.3	PRELU (Parametric leaky ReLU)	106
7.54.4	Sigmoid	106
7.54.5	Hyperbolic Tangent	106
7.54.6	Softmax	106
7.54.7	Softplus	106
7.54.8	ELU (Exponential Linear Unit)	106
7.54.9	SELU (Scaled ELU)	106
7.55	Optimizers	106
7.55.1	SGD (Stochastic Gradient Descent)	106
7.55.2	Mini-batch SGD	107
7.55.3	Batch SGD	107
7.55.4	Momentum	107
7.55.5	NAG (Nesterov Accelerated Gradient)	107
7.55.6	AdaGrad	107
7.55.7	RMSProp (Root Mean Square Propagation)	107
7.55.8	Adam (ADAPtive Moment estimation)	107
7.55.9	AdaMax	107
7.55.10	Nadam (Nesterov Adam)	107
7.56	Loss Functions	107
7.56.1	Binary crossentropy	107
7.57	RNNs (Recurring Neural Networks)	107
7.57.1	Unstable gradients	108
7.57.2	Memory cells	108
7.57.3	Limited memory	108
7.57.4	Encoder/decoder	108
7.57.5	Stateless	108
7.58	CNNs (Convolutional Neural Networks)	108
7.58.1	Description	108
7.58.2	Architectures	108
7.58.3	Convolutional layers	108
7.58.4	Pooling layers	109
7.58.5	MaxPooling	109
7.58.6	Filters	109
7.58.7	Transfer learning	109
7.58.8	Localization	110



7.58.9	Object detection	110
7.59	GANs (Generative Adversial Networks)	110
7.60	ANN (Artificial Neural Network) models & configuration	110
7.60.1	Import layers	110
7.60.2	Import models	110
7.60.3	Input layer	110
7.60.4	Multimodal Input	110
7.60.5	layers.Flatten(input_shape(x,y))	110
7.60.6	Dense layer	111
7.60.7	LSTM layer	111
7.60.8	Output layer	111
7.60.9	Compile model	112
7.60.10	Fit model	112
7.60.11	Plot fit results	112
7.60.12	Save model to HDF5	112
7.60.13	Load model	112
7.60.14	Checkpoints	112
7.60.15	Checkpoints versioned	113
7.60.16	Adaptive learning rate	113
7.60.17	Transfer learning	113
7.60.18	Early stopping	113
7.60.19	Regularization techniques	114
7.60.20	Initializer	114
7.60.21	Dropout	114
7.60.22	Constraint	114
7.60.23	Custom activation/initializer/regularizer/constraint	115
7.60.24	Compile	115
7.60.25		115
7.60.26	RNN	115
7.60.27	Autoencoders	115
7.60.28	Hopfield network	115
7.60.29	RBM (Restricting Boltzmann techniques)	115
7.60.30	GAN (Generative Adversial Network)	115
7.60.31	LSTM (Long Short-Term Memory)	116
7.60.32	WaveNet	116
7.60.33	Gru (Gated Recurrent Unit) cells	116
7.60.34	LeNet-5	116
7.60.35	AlexNet	116
7.60.36	ZF Net	116
7.60.37	GoogLeNet	116
7.60.38	VGGNet (Visual Geometry Group Net)	116
7.60.39	ResNet (Residual Network)	116
7.60.40	Xception (Extreme Inception)	117
7.60.41	SENet (Squeeze-and-Excitation Network)	117
7.60.42	FCN (Fully Convolutional Networks)	117
7.60.43	YOLO (You Only Look Once)	117
7.61	Reinforcement Learning	117
7.61.1	Overview	117
7.61.2	Agent	117
7.61.3	Environment	117
7.61.4	State	118
7.61.5	Reward	118
7.61.6	Action	118
7.61.7	Policy	118
7.61.8	Value Function	118
7.61.9	Q-Learning	118
7.61.10	Markov decision processes	118
7.61.11	Model based vs model free	118
7.61.12	Model	118
7.61.13	Model tuning	118
7.61.14	Policy gradient methods	119

7.62	Policy Gradient Methods	119
7.62.1	Stochastic policy	119
7.62.2	Genetic policy	119
7.62.3	Deep Deterministic Policy Gradient (DDPG)	119
7.62.4	Epsilon greedy policy	119
7.62.5	SARSA	119
7.62.6	Credit assignment problem	119
7.63	Reinforcement Learning models	119
7.63.1	REINFORCE	119
7.63.2	Q-Learning	119
7.63.3	Deep Q-Networks (DQN)	119
7.63.4	Double DQN	119
7.63.5	DDQN (Dueling DQN)	120
7.63.6	Proximal Policy Optimization (PPO)	120
7.63.7	Asynchronous Advantage Actor-Critic (A3C)	120
7.63.8	Advantage Actor-Critic (A2C)	120
7.63.9	Soft Actor-Critic (SAC)	120
7.63.10	Trust Region Policy Optimization (TRPO)	120
7.63.11	Proximal Policy Optimization (PPO)	120
7.63.12	Curiosity-based exploration	120
<b>8</b>	<b>Statistics Appendix (Math 001)</b>	<b>120</b>
8.1	Mean	120
8.2	Median	120
8.3	Mode	120
8.4	Variance	120
8.5	Covariance	120
8.6	Beta	120
8.7	Standard-Deviation	120
8.8	Standard Error	121
8.9	Relative Standard Error	121
8.10	Pearson Correlation	121
8.11	Standard Correlation Coefficient Pearson's r	121
8.12	Convolution	121
8.13	Fourier Transform	121
8.14	Laplace Transform	121
8.15	Autocovariance	121
8.16	Gaussian Distribution	121
8.17	Time Series	121
<b>9</b>	<b>Probability Appendix (Math 002)</b>	<b>122</b>
9.1	Picking Probability	122
9.1.1	Clusters	122
9.2	Martingale	122
9.3	Bayes's Theorem	122
9.4	Laplace Smoothing	122
9.5	PDF (Probability Density Function)	122
<b>10</b>	<b>Algebra Appendix (Math 003)</b>	<b>122</b>
10.1	Matrices	122
10.1.1	Sparse Matrix:	122
10.1.2	Dense Matrix:	122
10.2	Tensors	122
10.2.1	Axes (rank)	122
10.2.2	Shape	123
10.2.3	Scalars (rank-0 tensors)	123
10.2.4	Vectors (rank-1 tensors)	123
10.2.5	Matrices (rank-2 tensors)	123
10.2.6	Higher rank (rank-3 and more)	123
10.3	Useful Calculations	123
10.3.1	Harmonic Mean:	123

10.3.2 Distances . . . . .	123
10.4 Boolean Algebra . . . . .	123
10.4.1 Logical Operators . . . . .	123
10.4.2 Truth Tables . . . . .	124
<b>11 Analysis Appendix (Math 004)</b>	<b>124</b>
11.1 Bellman equations . . . . .	124

# 1 Syntax & Fundamentals

## 1.1 Methods

Dunder method `__getitem__` is used for example is used with Double UNDERscore before and aftER. List of available methods:

### 1.1.1 `__getitem__`

### 1.1.2 `__repr__`

With repr this only represents object with `'iVector object at 0x10e100070:'`

### 1.1.3 `__abs__`

### 1.1.4 `__add__`

### 1.1.5 `__mul__`

### 1.1.6 `__and__`

Used to represent  $a \& b$  which is the intersection of two sets.

### 1.1.7 `__len__`

- ◇ Collections
- ◇ Attribute access
- ◇ Iteration
- ◇ Async (Asynchronous Iteration)
- ◇ Operator overloading
- ◇ Function Invocation
- ◇ String representation and Formatting
- ◇ Asynchronous programming (await)
- ◇ Object creation and destruction
- ◇ Managed contexts (async)

### 1.1.8 Emulating numeric types

```
class MyInt:
    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        return MyInt(self.value + other.value)

# Usage
a = MyInt(5)
b = MyInt(3)
c = a + b # This will call the __add__ method
```

### 1.1.9 String representation of objects

```
class MyClass:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return f'Hello, {self.name}!'

    def __repr__(self):
        return f'MyClass({self.name})'

# Usage
obj = MyClass('Alice')
print(obj) # This will call the __str__ method
print(repr(obj)) # This will call the __repr__ method
```

### 1.1.10 Boolean value of an object

```
class MyList:
    def __init__(self, items):
        self.items = items

    def __bool__(self):
        return bool(self.items)

# Usage
obj = MyList([])
if obj:
    print('The object is truthy')
else:
    print('The object is falsy') # This will be printed
```

### 1.1.11 Implementing collections

A collection is a container that holds other objects (known as elements/items), python has different available collection types such as list/tuple/set/dictionary

```
from collections.abc import MutableSequence

class MyList(MutableSequence):
    def __init__(self, items):
        self._items = items

    def __getitem__(self, index):
        return self._items[index]

    def __setitem__(self, index, value):
        self._items[index] = value

    def __delitem__(self, index):
        del self._items[index]

    def __len__(self):
        return len(self._items)

    def insert(self, index, value):
```

```

        self._items.insert(index, value)

# Usage
obj = MyList([1, 2, 3])
obj.insert(1, 'a')
print(obj) # This will print [1, 'a', 2, 3]

```

## 1.2 Formatting

### 1.2.1 f''

```

name = "John"
print(f'Hi my name is {name}')
```

### 1.2.2 .format()

```

>>> a = 3
>>> print("Hi, do you know the number {}".format(a))
Hi, do you know the number 3
>>>

```

## 1.3 Functions

### 1.3.1 Declaration structure

```

def f(x, y, z):
    """
    Purpose of the function description

    :param x: description of the parameter x
    :param y: description of the parameter y
    :param z: description of the parameter z
    :return: what the function returns
    """

```

### 1.3.2 lambda method

```

f = lambda x, y: x + y

```

$f$  is the function's name,  $x, y$  are the input variables and  $x + y$  is the result to return.

### 1.3.3 Decorators

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
#####
#Output:
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

## 1.4 Classes

### 1.4.1 \_\_init\_\_

```
class State:
    def __init__(self):
        self.counter = 0
    def __call__(self, x):
        self.counter += x
        print(self.counter)

a = State()
a(2) => 2
a(2) => 4
```

### 1.4.2 \_\_call\_\_

Integer division

```
a // b # 2//3 = 0 (rounded division)
```

Slicing list

```
var_name[i:j] # returns |j-i| elements
```

Create a class

```
class Person:
    def __init__(self, age, name):
        self.name = name
    def greet(self):
        print("Hi: ", self.name)
```

## 2 Major Modules

### 2.1 Itertools

#### 2.1.1 Import

```
from itertools import ...
```

#### 2.1.2 combinations

```
from itertools import combinations
all_samples = [sample for sample in combinations("ABCDEFGG", 3)]
print(all_samples)
print("Number of Samples:", len(all_samples))

#####
[('A', 'B', 'C'), ('A', 'B', 'D'), ('A', 'B', 'E'), ('A', 'B', 'F'), ('A', 'B', 'G'),
 ('A', 'C', 'D'), ('A', 'C', 'E'), ('A', 'C', 'F'), ('A', 'C', 'G'), ('A', 'D',
 'E'), ('A', 'D', 'F'), ('A', 'D', 'G'), ('A', 'E', 'F'), ('A', 'E', 'G'), ('A',
 'F', 'G'), ('B', 'C', 'D'), ('B', 'C', 'E'), ('B', 'C', 'F'), ('B', 'C', 'G'),
 ('B', 'D', 'E'), ('B', 'D', 'F'), ('B', 'D', 'G'), ('B', 'E', 'F'), ('B', 'E',
 'G'), ('B', 'F', 'G'), ('C', 'D', 'E'), ('C', 'D', 'F'), ('C', 'D', 'G'), ('C',
 'E', 'F'), ('C', 'E', 'G'), ('C', 'F', 'G'), ('D', 'E', 'F'), ('D', 'E', 'G'), ('
 D', 'F', 'G'), ('E', 'F', 'G')]
Number of Samples: 35
```

#### 2.1.3 permutations

```
from itertools import permutations
print(["".join(sample) for sample in permutations("ABC")])

#####
['ABC', 'ACB', 'BAC', 'BCA', 'CAB', 'CBA']
```

## 2.2 OS

### 2.2.1 Import

```
import os
```

### 2.2.2 Execute command in prompt

```
os.system("command to execute")
```



### 2.2.3 Get current path

```
os.path.abspath(os.path.filename(__file__))
```

### 2.2.4 Join paths

```
os.path.join(path_1, path_2)
```

## 2.3 Requests

### 2.3.1 Import

```
import requests
```

### 2.3.2 Get request to url

```
response = requests.get(url)
```

## 2.4 JSON

### 2.4.1 Save json file

```
with open(path, "w") as file:  
    json.dump(data, file, indent=4)
```

### 2.4.2 Load json file

```
with open(path, "r") as file:  
    data = json.load(file)
```

## 2.5 Pickle

### 2.5.1 Dump

```
from pickle import dump  
dump(model, open(filename, 'wb'))
```

## 2.5.2 Load

```
from pickle import load
loaded_model = load(open(filename, 'rb'))
```

## 2.6 Matplotlib

### 2.6.1 Import

```
import matplotlib as plt
import matplotlib.pyplot as plt
```

### 2.6.2 Save figure

```
plt.savefig(<file_path>, format="png", dpi=300)
```

### 2.6.3 Multiple Graphs

#### First method

```
# Create a new figure
fig = plt.figure(figsize=(10, 6))

# Add subplots
ax1 = fig.add_subplot(221) # 2 rows, 2 columns, index 1
ax2 = fig.add_subplot(222) # 2 rows, 2 columns, index 2
ax3 = fig.add_subplot(223) # 2 rows, 2 columns, index 3
ax4 = fig.add_subplot(224) # 2 rows, 2 columns, index 4

# Plot on each subplot
ax1.plot([1, 2, 3], [1, 2, 3])
ax1.set_title('Plot 1')

ax2.plot([1, 2, 3], [3, 2, 1])
ax2.set_title('Plot 2')

ax3.plot([1, 2, 3], [2, 1, 2])
ax3.set_title('Plot 3')

ax4.plot([1, 2, 3], [1, 3, 2])
ax4.set_title('Plot 4')

plt.tight_layout()
plt.show()
```

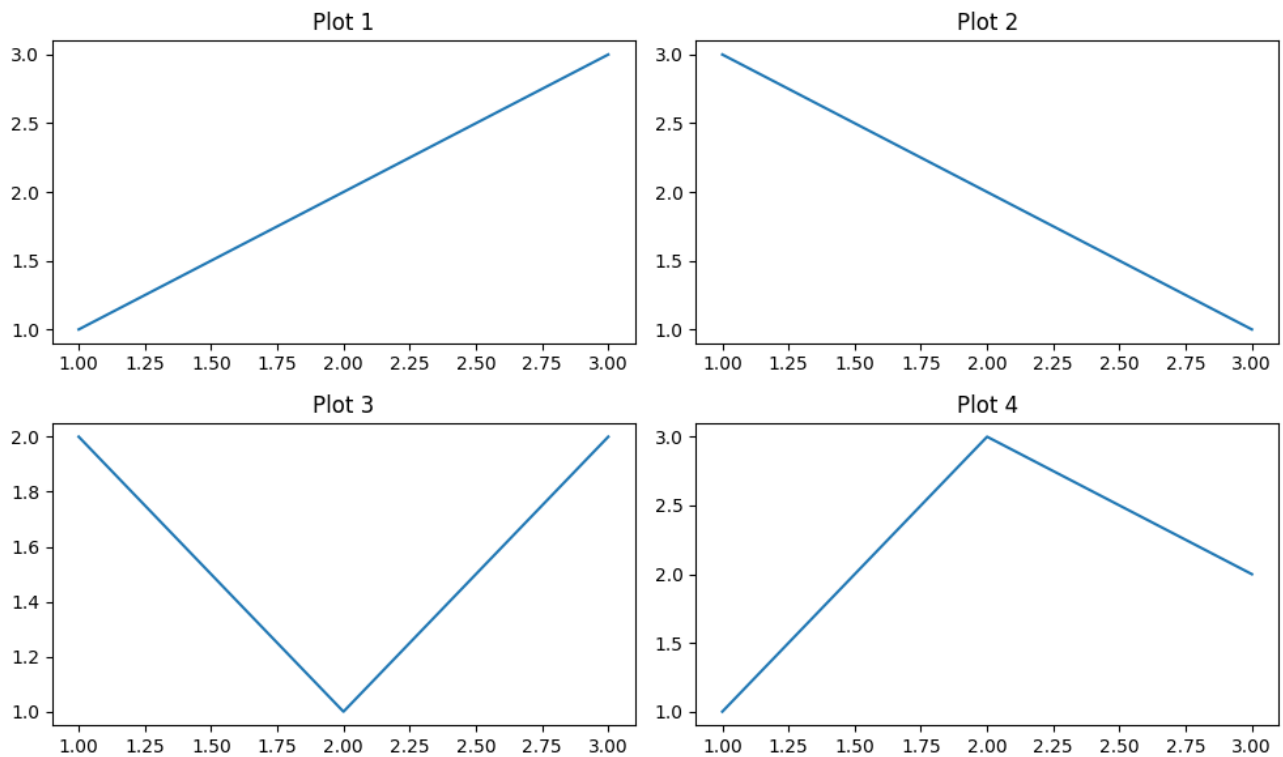


Figure 1: Different graphs stacked on matplotlib

## Second method

```
# Create a figure and subplots
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 8))

# Plot prices on the top subplot
ax1.plot(A, color='blue')
ax1.set_title('Dataset A')
ax1.grid(True)

# Plot RSI values on the bottom subplot
ax2.plot(B, color='green')
ax2.set_title('Dataset B')
ax2.grid(True)
```

### 2.6.4 Plot dataframe column in graph

```
goog_data['Close'].plot(ax=ax1, color='r', lw=2.)
```

`ax1` is the subplot on which to plot the data, `color` is the hyperparameter for color and `lw` is the hyperparameter for the line width.

### 2.6.5 Buy and sell markers

```
# Fetch Tesla stock data
tesla = yf.Ticker("TSLA")
data = tesla.history(period="1y")
```

```

# Calculate 50-day simple moving average
data['SMA_50'] = data['Close'].rolling(window=50).mean()

# Generate buy and sell signals based on SMA strategy
data['Signal'] = 0
data.loc[data['Close'] > data['SMA_50'], 'Signal'] = 1 # Buy signal
data.loc[data['Close'] < data['SMA_50'], 'Signal'] = -1 # Sell signal

# Filter buy and sell signals
buy_signal = data[data['Signal'] == 1]
sell_signal = data[data['Signal'] == -1]

plt.plot(data.index, data['Close'], label='Tesla Stock Price')
plt.plot(data.index, data['SMA_50'], label='50-Day SMA', linestyle='--', color='orange')
plt.scatter(buy_signal.index, buy_signal['Close'], marker='^', color='green', label='Buy Signal', s=100)
plt.scatter(sell_signal.index, sell_signal['Close'], marker='v', color='red', label='Sell Signal', s=100)

```

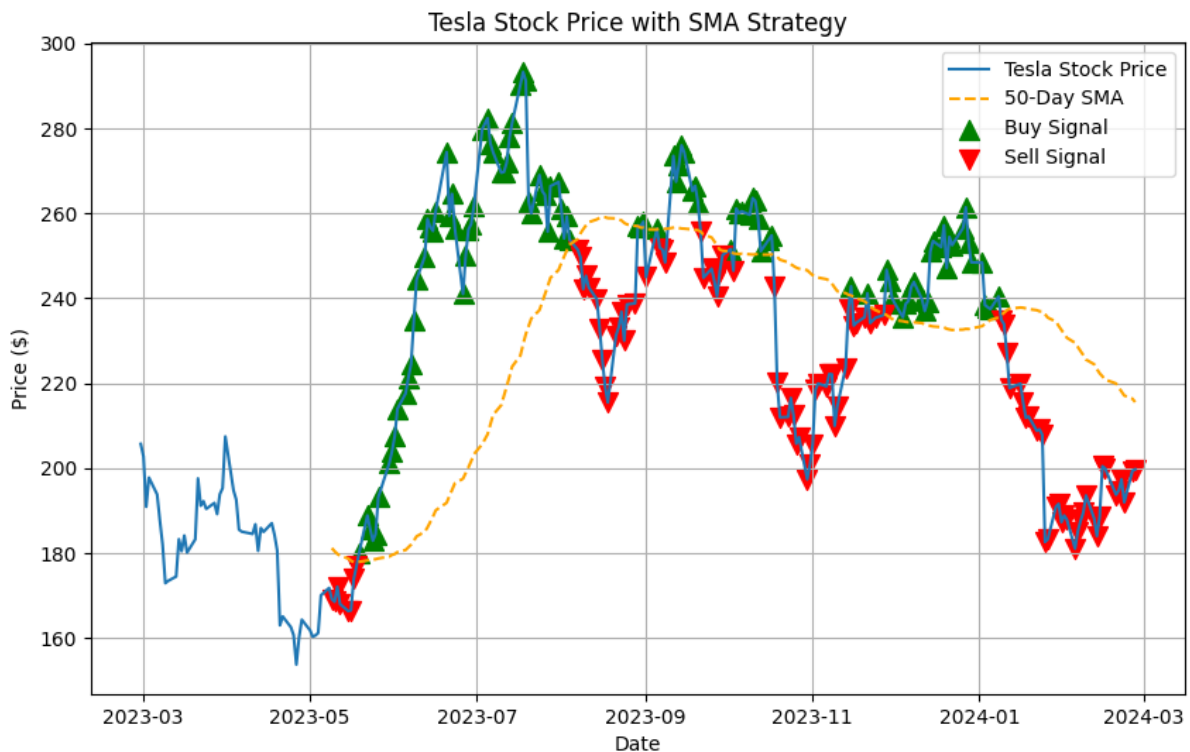


Figure 2: Buying and selling points on graph

### 2.6.6 Modern graph

```

import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# Generate random data
np.random.seed(0)
n_samples = 100

```

```

n_features = 5
n_clusters = 3
X = np.random.randn(n_samples, n_features)
Y_train = np.random.randint(0, n_clusters, n_samples)

# Create DataFrame from random data
dfsvd = pd.DataFrame(X, columns=['c{}'.format(c) for c in range(n_features)])

# Select columns for plotting
plotdims = 2
dfsvdplot = dfsvd.iloc[:, :plotdims]
dfsvdplot['Close_pred'] = Y_train

# Create pairplot using seaborn
ax = sns.pairplot(dfsvdplot, hue='Close_pred', height=2.5)

# Ensure tight layout for better visualization
plt.tight_layout()

# Display the plot
plt.show()

```

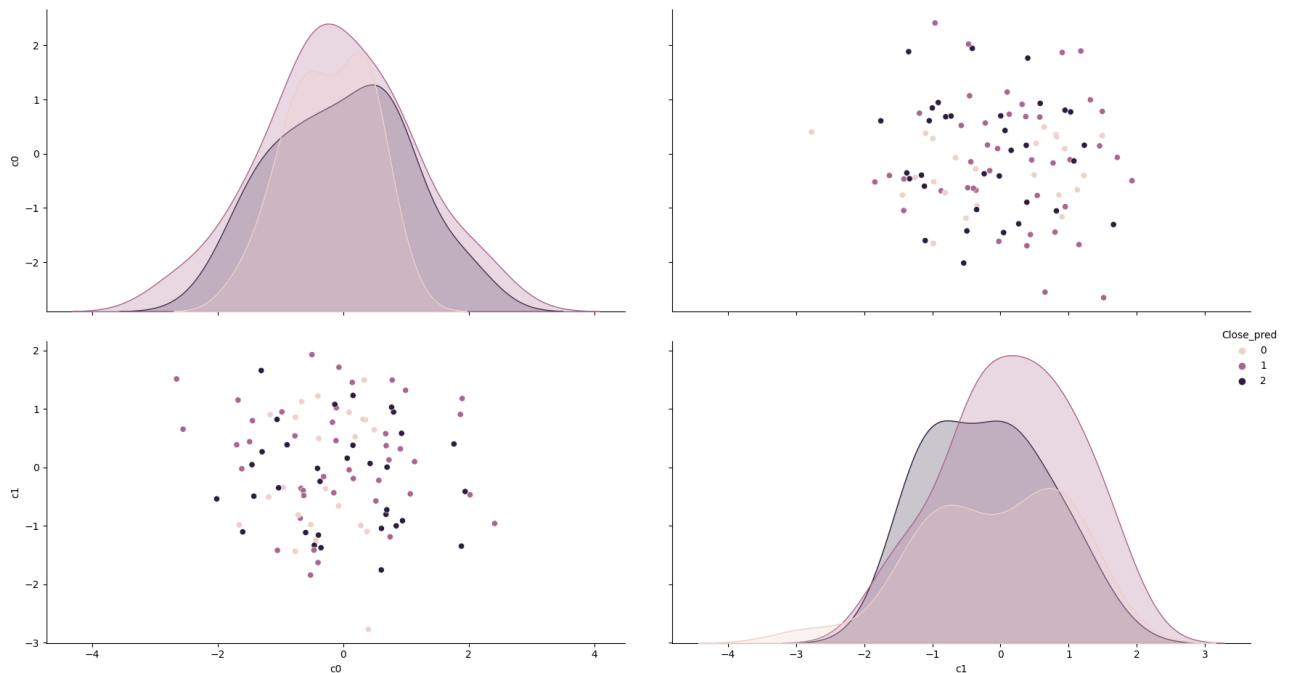


Figure 3: Modern scatter and line graph

### 2.6.7 3D Scatter Plot

```

np.random.seed(42)

def randrange(n, vmin, vmax):
    """
    Helper function to make an array of random numbers having shape (n, )
    with each number distributed Uniform(vmin, vmax).
    """
    return (vmax - vmin)*np.random.rand(n) + vmin

fig = plt.figure(figsize=(16, 10))
ax = fig.add_subplot(projection='3d')

```

```

n = 100

# For each set of style and range settings, plot n random points in the box
# defined by x in [23, 32], y in [0, 100], z in [zlow, zhigh].
for m, zlow, zhigh in [('o', -50, -25), ('v', -30, -5), ('^', -30, -5)]:
    xs = randrange(n, 23, 32)
    ys = randrange(n, 0, 100)
    zs = randrange(n, zlow, zhigh)
    ax.scatter(xs, ys, zs, marker=m)

ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')

plt.show()

```

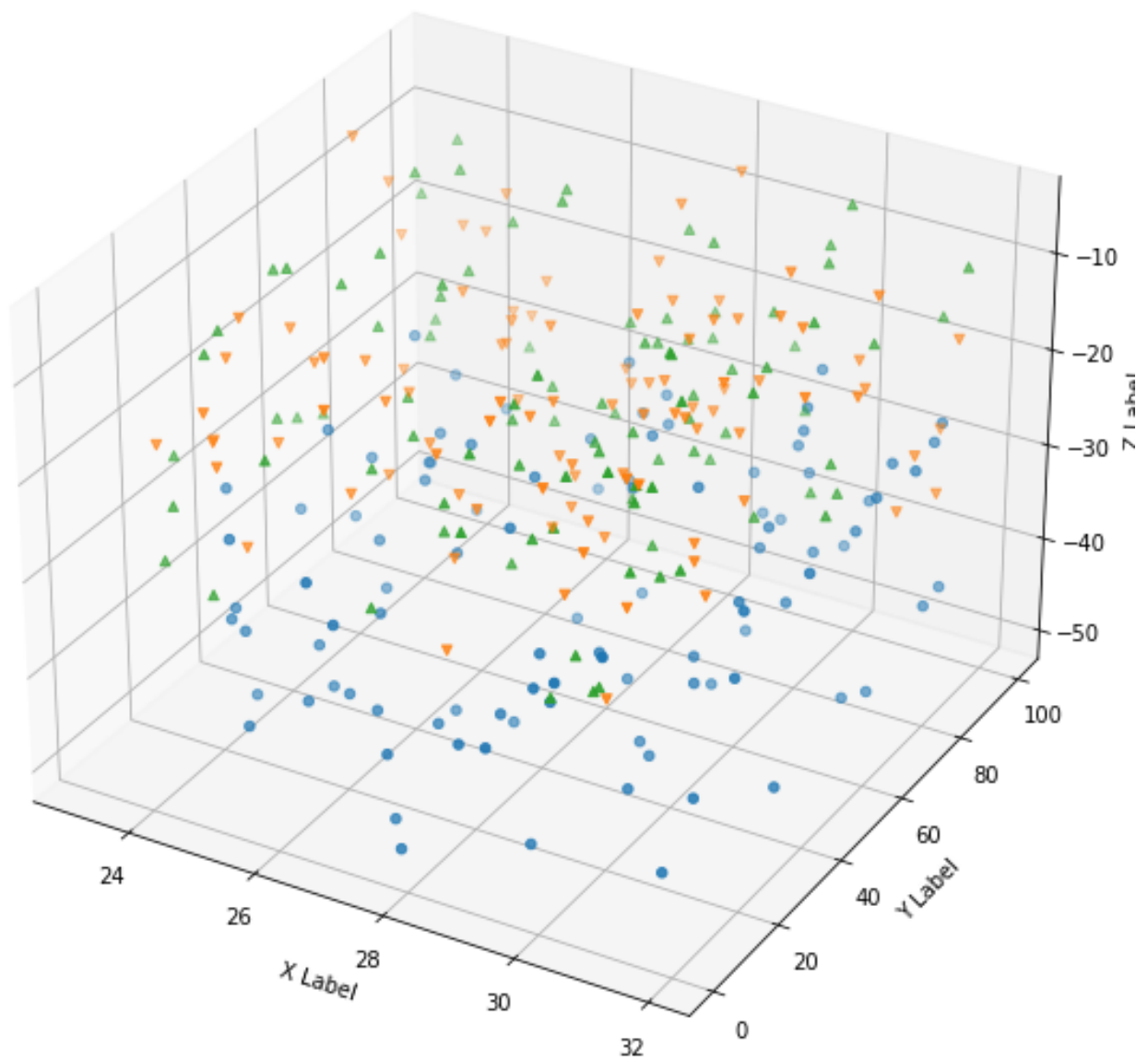


Figure 4: 3D Scatter graph

### 2.6.8 Graph settings

```
plt.xlabel("text for the x label") # label for the x axis
plt.ylabel("text for the y label")
plt.show() # display graph
```

### 2.6.9 Line plot

```
plt.plot(x, y)
```

### 2.6.10 Histogram

```
plt.hist(data, number_bins, color="blue")
```

### 2.6.11 Scatter plot

```
plt.scatter(x, y, alpha=0.1, s=50)
```

Where  $x$  and  $y$  are the coordinates of the points, *alpha* is the transparency of it,  $s$  is the size of the point. To plot based on classes, make use of:

```
test_points = np.array([[1, 2], [2, 1], [3, 4], [4, 5]])
classes = np.array([0, 0, 1, 1])
plt.scatter(test_points[:, 0], test_points[:, 1], c=classes, cmap='cool')
```

Which results in the following:

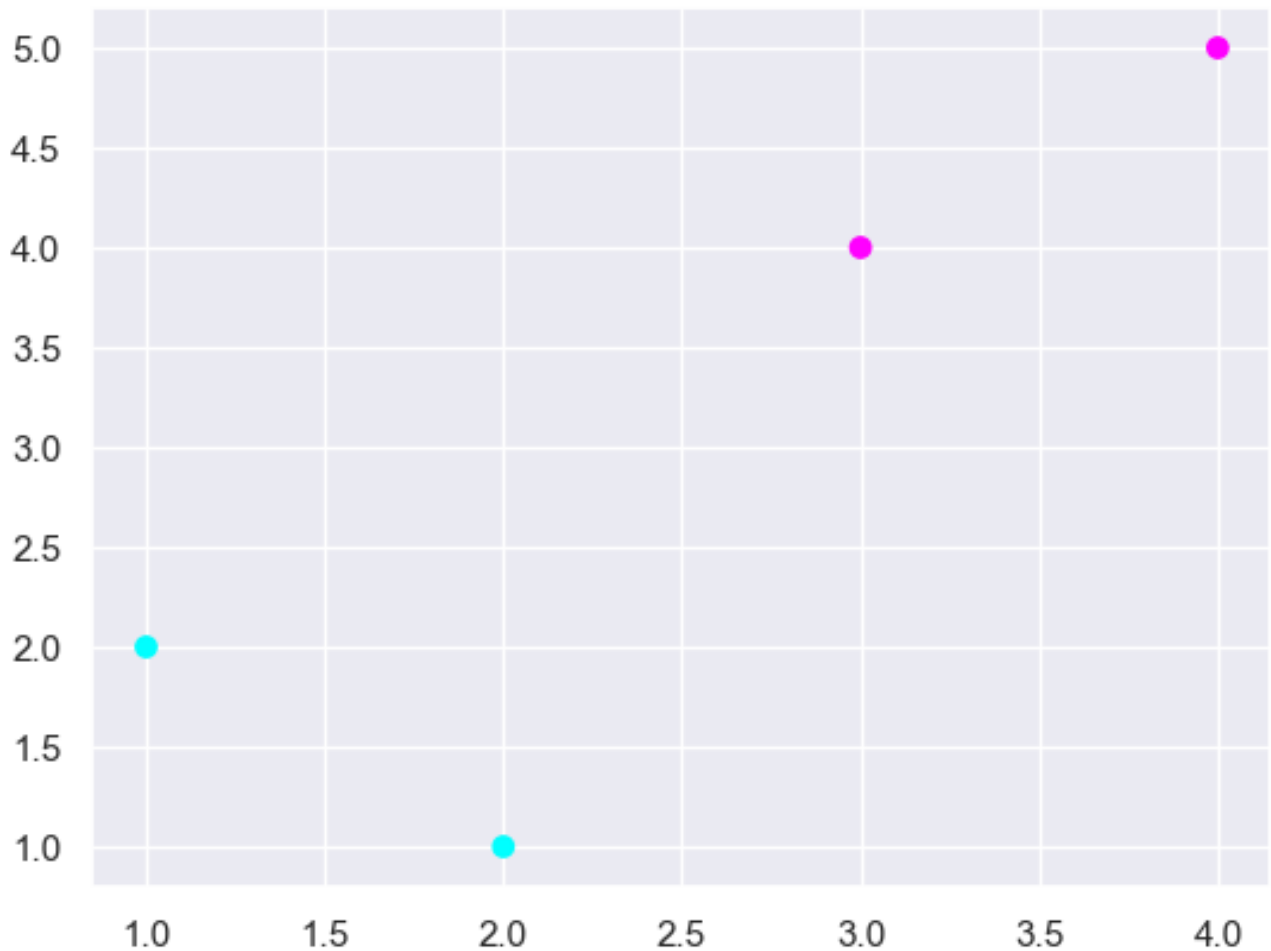


Figure 5: Scatter plot with color based on classes

### 2.6.12 Heatmap

```
plt.imshow(dataset, cmap="gray_r")
```

## 2.7 Seaborn

### 2.7.1 Pair plot

For higher than 3 dimensions impossible to plot all the data on a single graph.

```
import seaborn as sns
sns.pairplot(df)
```

With *df* the dataframe containing all the data.

An example showcasing it would be with the iris dataset:

```
import pandas as pd
from sklearn.datasets import load_iris
import seaborn as sns

iris = load_iris()
df = pd.DataFrame(iris.data, columns=iris.feature_names)
```



```
df['class'] = iris.target
df['class name'] = iris.target_names[iris['target']]
df.head()
sns.pairplot(df)
```

Noticeable that ignores textual classes (strings) because they can't be plotted. Result should look like this:

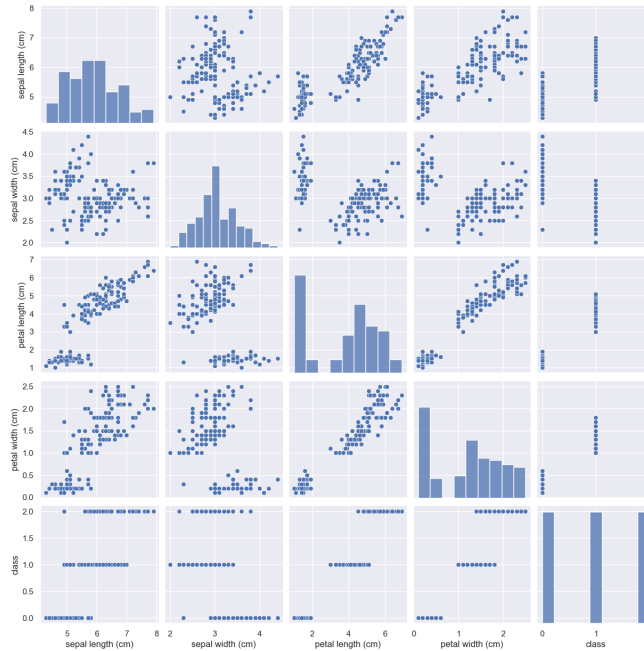


Figure 6: Pair plot of the iris setosa dataset

pairplot seaborn iris setosa

## 2.8 SciPy

## 2.9 Numpy

### 2.9.1 Import

```
import numpy as np
```

### 2.9.2 Operators

```
arr = np.array(-1, 1)
arr > 0 # => array([False, True])
```

### 2.9.3 Vertical slicing

```
a = np.array([[1, 2], [1, 2], [3, 4]])
# [[1, 2],
#  [1, 2],
#  [3, 4]]
# ]
```

```
first_column = a[:, 0] # => [1, 1, 3]
```

#### 2.9.4 np.astype(type)

```
arr = np.array([-1, 0.2, 2.43])
arr.astype(int) # => array([-1, 0, 2])
```

#### 2.9.5 np.trunc()

```
np.trunc(8.99) # => 8.0
np.trunc([8.3, 3.7]) # => [8.0, 3.0]
```

#### 2.9.6 np.choice(sample, size, replace)

```
urn = ["b", "b", "b", "w", "w"]
chosen = np.random.choice(urn, size=2, replace=False)
```

*Size* represents the number of elements in the list returned, *replace* set to false means that once the marble is selected it is not returned to the urn (one by one, not in bulk, consider a size=4 in an urn of 3 elements, won't throw an error if *replace* is set to *True* but will if it is set to *False*), with *urn* being the list from which to pick selected items. Returns an array.

#### 2.9.7 np.mean()

```
valeurs = [True, False, True]
np.mean(valeurs) # => 0.66 because True counts as a one and False as a zero
```

#### 2.9.8 np.unique()

```
sample = [1, 2, 2, 3, 4, 4, 4, 5, 6]
unique_elem, counts_elem = np.unique(sample, return_counts=True)
#####
>>> unique_elem
array([1, 2, 3, 4, 5, 6])
>>> counts_elem
array([1, 2, 1, 3, 1, 1], dtype=int64)
```

Setting *return\_counts* to false returns a list containing all the elements that appear at least once.

### 2.9.9 np.where()

Useful to iterate through pandas dataframe and create new column of data

```
df['signal'] = np.where(condition, a, b)
```

Where the 'signal' column is created and a is put in the row if the condition is True otherwise writes b in the dataframe. *condition* might be dataset['SMA.short'] & dataset['SMA.long']

### 2.9.10 np.eye()

Create an identity matrix.

```
eye = np.eye(4)
```

### 2.9.11 np.reshape(x, y)

```
array.reshape(x, y)
```

Where x and y are desired dimensions (row x column).

### 2.9.12 np.reshape(-1, 1)

Reshape as column vector.

```
arr = np.array([1, 2, 3, 4, 5, 6])
reshaped_arr = arr.reshape(-1, 1)
#####
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6]])
```

### 2.9.13 np.reshape(1, -1)

Reshape as horizontal array.

```
arr = np.array([1, 2, 3, 4, 5, 6])
reshaped_arr = arr.reshape(1, -1)
#####
array([1, 2, 3, 4, 5, 6])
```

### 2.9.14 `np.c_[a, b]`

Turn two arrays into columns of bigger array:

```
a = [1, 2, 3]
b = [4, 5, 6]
np.c_[a, b]
#####
array([[1, 4],
       [2, 5],
       [3, 6]])
```

### 2.9.15 `np.linspace(start, end, steps)`

```
np.linspace(0, 10, 100)
#####
array([ 0.          ,  0.1010101 ,  0.2020202 ,  0.3030303 ,  0.4040404 ,
        0.50505051,  0.60606061,  0.70707071,  0.80808081,  0.90909091,
        1.01010101,  1.11111111,  1.21212121,  1.31313131,  1.41414141,
        1.51515152,  1.61616162,  1.71717172,  1.81818182,  1.91919192,
        2.02020202,  2.12121212,  2.22222222,  2.32323232,  2.42424242,
        2.52525253,  2.62626263,  2.72727273,  2.82828283,  2.92929293,
        3.03030303,  3.13131313,  3.23232323,  3.33333333,  3.43434343,
        3.53535354,  3.63636364,  3.73737374,  3.83838384,  3.93939394,
        4.04040404,  4.14141414,  4.24242424,  4.34343434,  4.44444444,
        4.54545455,  4.64646465,  4.74747475,  4.84848485,  4.94949495,
        5.05050505,  5.15151515,  5.25252525,  5.35353535,  5.45454545,
        5.55555556,  5.65656566,  5.75757576,  5.85858586,  5.95959596,
        6.06060606,  6.16161616,  6.26262626,  6.36363636,  6.46464646,
        6.56565657,  6.66666667,  6.76767677,  6.86868687,  6.96969697,
        7.07070707,  7.17171717,  7.27272727,  7.37373737,  7.47474747,
        7.57575758,  7.67676768,  7.77777778,  7.87878788,  7.97979798,
        8.08080808,  8.18181818,  8.28282828,  8.38383838,  8.48484848,
        8.58585859,  8.68686869,  8.78787879,  8.88888889,  8.98989899,
        9.09090909,  9.19191919,  9.29292929,  9.39393939,  9.49494949,
        9.59595959,  9.69696969,  9.79797979,  9.89898989, 10.          ])
```

### 2.9.16 `np.cov(x, y)`

```
x = np.array([1, 2, 3, 4, 5])
y = np.array([5, 4, 3, 2, 1])

covariance_matrix = np.cov(x, y)
#####
array([[ 2.5, -2.5],
       [-2.5,  2.5]])
```

Covariance matrix can be displayed as such:

$$\begin{bmatrix} \text{cov}(x, x) & \text{cov}(x, y) \\ \text{cov}(y, x) & \text{cov}(y, y) \end{bmatrix}$$

### 2.9.17 np.var(x, y, ddof=k)

```
x = [1, 2, 3, 1, 0]
np.var(x)
#####
1.04
```

Where  $k$  is 0 by default and represents the divider  $N - k$ .

### 2.9.18 np.cumsum()

```
np.cumsum(data_array)
```

### 2.9.19 np.random.randn(n)

Generate  $n$  random numbers, thus comparable to noise generation.

```
noise = np.random.randn(100)
```

### 2.9.20 np.random.standard\_normal((rows, columns))

Generate random matrix.

```
np.random.standard_normal((5, 4))
```

### 2.9.21 np.eye()

Meshgrid to be used for 3D graphing.

```
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])
X, Y = np.meshgrid(x, y)
#####
X = [[1, 2, 3],
     [1, 2, 3],
     [1, 2, 3]]

Y = [[4, 4, 4],
     [5, 5, 5],
     [6, 6, 6]]
```

Import module

```
import numpy as np
```

Create identity matrix (I)

```
eye = np.eye(4)
```

Create array

```
x = np.array([[1,2,3][4,5,6]])
```

### 2.9.22 np.repeat(array, iterations)

```
arr = np.array([1, 2, 3])
repeated_arr = np.repeat(arr, 3)
print("Original Array:", arr)
print("Repeated Array:", repeated_arr)
#####
Original Array: [1 2 3]
Repeated Array: [1 1 1 2 2 2 3 3 3]
```

### 2.9.23 np.convolve()

mathematical appendix explain how convolution works?

```
a = np.array([i for i in range(10)])
b = np.array([-i for i in range(10)])
conv_result = np.convolve(a, b)

print("Array a:", a)
print("Array b:", b)
print("Convolution Result:", conv_result)
#####
Array a: [0 1 2 3 4 5 6 7 8 9]
Array b: [0 -1 -2 -3 -4 -5 -6 -7 -8 -9]
Convolution Result: [0 0 -1 -4 -10 -20 -35 -56 -84 -120 -165 -200 -224 -236 -235
-220 -190 -144 -81]
```

### 2.9.24 np.ones()

```
np.ones(10)
#####
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

### 2.9.25 np.linalg.inv(A)

Calculate inverse of matrix.

```
np.linalg.inv()
```

### 2.9.26 np.linalg.pinv(A)

Moore-Penrose inverse calculation.

```
np.linalg.pinv()
```

## 2.10 Pandas

### 2.10.1 Import

```
import pandas as pd
```

### 2.10.2 Axis representation

Axis = 0 represents a row, axis = 1 represents a column.

### 2.10.3 Load dataframe

Load dataframe from file path.

```
df = pd.read_csv(<file_path>, header=0)
```

*header* = 0 to use first column as column titles

### 2.10.4 Create dataframe

```
data = {"column_name": [row1, row2, row3, ...]}  
df = pd.DataFrame(data)
```

### 2.10.5 Create dataframe with dict

```
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
```

### 2.10.6 Get dataframe shape

```
df.shape()
```

### 2.10.7 Iterate through rows

```
for index, row in df.iterrows():  
    print(f'Index: {index}, A: {row["A"]}, B: {row["B"]}'))
```

## 2.10.8 Change column datatype

```
df['Close'] = df['Close'].astype(float)
```

## 2.10.9 Get rows specific index

Get index of rows with specific Contents

```
tesla = yf.Ticker("TSLA")
data = tesla.history(period="1y")
data['SMA_50'] = data['Close'].rolling(window=50).mean()
data['Signal'] = 0
data.loc[data['Close'] > data['SMA_50'], 'Signal'] = 1 # Buy signal
data.loc[data['Close'] < data['SMA_50'], 'Signal'] = -1 # Sell signal

buy_signal = data[data['Signal'] == 1]
sell_signal = data[data['Signal'] == -1]
```

Where the content of the buy\_signal and sell\_signal dataframes are the following:

```
>>> buy_signal
Date                Open      High      Low      Close      Volume  Dividends  Stock Splits  SMA_50  Signal
2023-05-19 00:00:00-04:00  177.169998  181.949997  176.309998  180.139999  136024200    0.0          0.0  177.951599    1
2023-05-22 00:00:00-04:00  180.699997  189.320007  180.110001  188.869995  132001400    0.0          0.0  178.260199    1
2023-05-23 00:00:00-04:00  186.199997  192.960007  185.259995  185.770004  156952100    0.0          0.0  178.485999    1
2023-05-24 00:00:00-04:00  182.229996  184.220001  178.220001  182.899994  137605100    0.0          0.0  178.478799    1
2023-05-25 00:00:00-04:00  186.539993  186.779999  180.580002  184.470001  96870700    0.0          0.0  178.559200    1
...
2024-01-02 00:00:00-05:00  250.080002  251.250000  244.410004  248.419998  104654200    0.0          0.0  233.462000    1
2024-01-03 00:00:00-05:00  244.979996  245.679993  236.320007  238.449997  121082600    0.0          0.0  233.991199    1
2024-01-04 00:00:00-05:00  239.250000  242.699997  237.729996  237.929993  102629300    0.0          0.0  234.508199    1
2024-01-05 00:00:00-05:00  236.860001  240.119995  234.899994  237.490005  92379400    0.0          0.0  234.927599    1
2024-01-08 00:00:00-05:00  236.139999  241.250000  235.300003  240.449997  85166600    0.0          0.0  235.488199    1
[106 rows x 9 columns]
>>> sell_signal
Date                Open      High      Low      Close      Volume  Dividends  Stock Splits  SMA_50  Signal
2023-05-09 00:00:00-04:00  168.949997  169.820007  166.559998  169.149994  88965000    0.0          0.0  181.176599   -1
2023-05-10 00:00:00-04:00  172.550003  174.429993  166.679993  168.539993  119840700    0.0          0.0  180.433199   -1
2023-05-11 00:00:00-04:00  168.699997  173.570007  166.789993  172.080002  103889900    0.0          0.0  179.819399   -1
2023-05-12 00:00:00-04:00  176.070007  177.380005  167.229996  167.979996  157577100    0.0          0.0  179.360999   -1
2023-05-15 00:00:00-04:00  167.660004  169.759995  164.550003  166.350006  105592500    0.0          0.0  178.732199   -1
...
2024-02-21 00:00:00-05:00  193.360001  199.440002  191.949997  194.770004  103844000    0.0          0.0  218.995800   -1
2024-02-22 00:00:00-05:00  194.000000  198.320007  191.360001  197.410004  92739500    0.0          0.0  218.067200   -1
2024-02-23 00:00:00-05:00  195.309998  197.570007  191.500000  191.970001  78670300    0.0          0.0  217.111800   -1
2024-02-26 00:00:00-05:00  192.289993  201.779999  192.000000  199.399994  111747100    0.0          0.0  216.359600   -1
2024-02-27 00:00:00-05:00  204.039993  205.600006  198.259995  199.729996  108521300    0.0          0.0  215.568400   -1
[96 rows x 9 columns]
```

Figure 7: CNN layers with rectangular receptive fields

## 2.10.10 Shuffle dataframe

```
data = {
    'A': [1, 2, 3, 4, 5],
    'B': ['a', 'b', 'c', 'd', 'e']
}
df = pd.DataFrame(data)
shuffled_df = df.sample(frac=1, random_state=42)
```



### 2.10.11 Confusion matrix

```
df_cm = pd.DataFrame(confusion_matrix(y_test_actual, y_test_pred), \
                      columns=np.unique(y_test_actual), index = np.unique(y_test_actual))
df_cm.index.name = 'Actual'
df_cm.columns.name = 'Predicted'
sns.heatmap(df_cm, cmap="Blues", annot=True, annot_kws={"size": 16})
```

### 2.10.12 Load from the web

```
url = "https://raw.githubusercontent.com/jeffprosise/Applied-Machine-Learning/main/Chapter%201/Data/customers.csv"
customers = pd.read_csv(url)
print(customers.head())
```

### 2.10.13 .loc(row, column)

```
df.loc(row, column)
```

*row* can be set to  $x_i : x_f$  where  $x_i$  is the initial row and  $x_f$  is the final one (e.g. `df.loc( $x_i : x_f$ , column)`), the number of elements collected can be calculated by  $x_f - x_i$ , using `:` will select the whole column (indeed all the rows). Multiple columns can be sliced by listing them in a list (e.g. `df.loc(row, [column1, column2, ...])`). To get a whole column `df[column_name]` also works.

### 2.10.14 .iloc()

```
df.iloc[:, 1:5]
```

Returns column 1 to 4.

### 2.10.15 .values

```
df.iloc[:, 1:5].values
```

	Gender	Age	Annual Income (k\$)	Spending Score (1-100)
0	1	19	15	39
1	1	21	15	81
2	0	20	16	6
3	0	23	16	77
4	0	31	17	40
...	...	...	...	...

(a) Before OHE

```
array([[ 1, 19, 15, 39],
       [ 1, 21, 15, 81],
       [ 0, 20, 16,  6],
       [ 0, 23, 16, 77],
       [ 0, 31, 17, 40],
       [ 0, 22, 17, 76],
       [ 0, 35, 18,  6],
       [ 0, 23, 18, 94],
       [ 1, 64, 19,  3],
       [ 0, 30, 19, 72],
       [ 1, 67, 19, 14],
       [ 0, 35, 19, 99],
       [ 0, 58, 20, 15],
       [ 0, 24, 20, 77],
       [ 1, 37, 20, 13],
       [ 1, 22, 20, 79],
       [ 0, 35, 21, 35],
       [ 1, 20, 21, 66],
       [ 1, 52, 23, 29],
       [ 0, 35, 23, 98],
       [ 1, 35, 24, 25],
```

(b) After OHE

Figure 8: Dataframe values extraction

### 2.10.16 `.merge(df1, df2, left_on, right_on, how)`

```
result = pd.merge(df1, df2, left_on='df1col', right_on='df2col', how='inner')
```

*how* set to inner will only match if existing matching keys; *left\_on* is the column from the *df1* to merge with the *right\_on* columns from *df2*.

### 2.10.17 `.drop([column1, column2, ...])`

```
df_bis = df.drop([column1, column2, ...], axis=1)
```

*Axis* set to 1 to drop columns.

### 2.10.18 .info()

```
df.info()
```

Gives important information about **size and data type**.

### 2.10.19 .describe()

```
df.describe()
```

	Time	V1	V2	V3	V4	V5	V6
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05
mean	94813.859575	1.168375e-15	3.416908e-16	-1.379537e-15	2.074095e-15	9.604066e-16	1.487313e-15
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00	1.332271e+00
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02	-2.616051e+01
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01	-7.682956e-01
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02	-2.741871e-01
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01	3.985649e-01
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01	7.330163e+01

Figure 9: Dataset description

### 2.10.20 .diff()

Difference between two rows

```
df["c"] = df["c"].diff()
```

### 2.10.21 .sort\_values

```
sorted_dataframe = df.sort_values(by='<column_name>', ascending=False)
```

### 2.10.22 shuffle(df, random\_state)

```
df = shuffle(df, random_state=42)
```

### 2.10.23 .concat([df1, df2])

Used to concatenate datasets.

```
df_conc = pd.concat([df1, df2])
```

### 2.10.24 .sample(frac)

```
sample_df = df.sample(frac=0.05)
```

### 2.10.25 .shift()

```
s = pd.Series([10, 20, 30, 40, 50])
shifted_s = s.shift(-1)
#####
Original Series:
0      10
1      20
2      30
3      40
4      50
dtype: int64

Shifted Series:
0      20.0
1      30.0
2      40.0
3      50.0
4         NaN
dtype: float64
```

### 2.10.26 .count()

Count classified instances.

```
class_names = {0: 'Not Fraud', 1: 'Fraud'}
print(dataset.Class.value_counts().rename(index = class_names)) ### count class
      columns to see how many are fraud and how many are not
```

Bar chart of instances count.

```
fig = plt.figure()
plot = dataset.groupby(['parsed_signals']).size().plot(kind='barh', color='red')
plt.show()
```

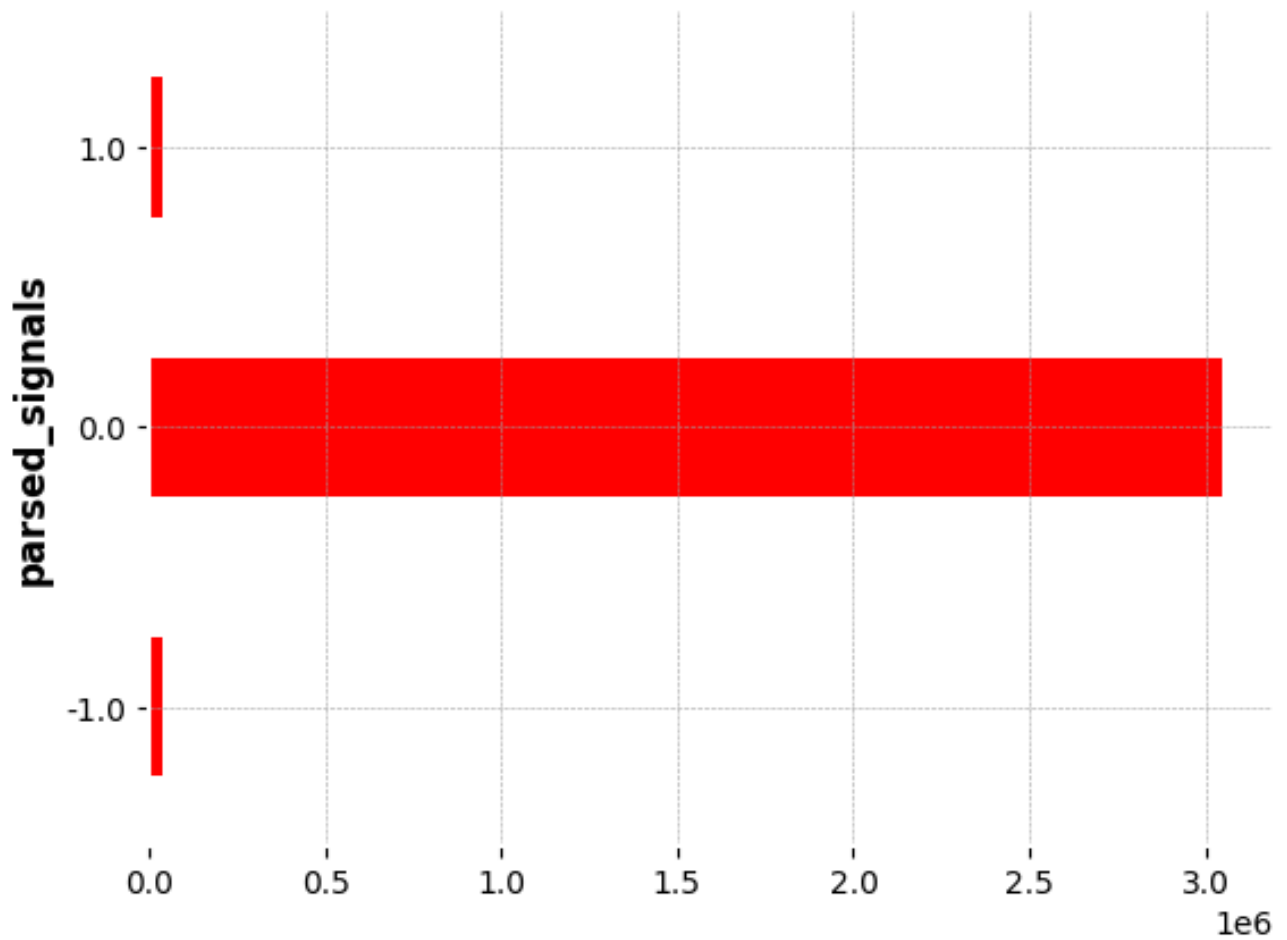


Figure 10: Signal count

#### 2.10.27 .head()

Prints out the head of the dataset

```
df.head()
```

#### 2.10.28 .ffill()

Forward fill (fill missing future value with previous value).

```
df.ffill()
```

#### 2.10.29 .get\_dummies()

Sex
male
female
female
female
male

(a) Before dummies split

Sex_female	Sex_male
False	True
True	False
True	False
True	False
False	True
...	...
False	True
True	False
True	False
False	True
False	True

(b) After dummies split

Figure 11: Dummies split resulting effect

### 2.10.30 .set\_index()

Define index column based on existing column.

```
df.set_index('Time', inplace=True)
```

### 2.10.31 .reset\_index()

```
df.reset_index(drop=True, inplace=True)
#####
A  B  C
0  1  4  7
1  2  5  8
2  3  6  9
```

### 2.10.32 .pct\_change()

Change from previous row

```
a = pd.DataFrame([2, 4, 2, 1])
a.pct_change()
#####
0  NaN
1  1.0
2 -0.5
3 -0.5
```

### 2.10.33 .corr()

```
data = {
    'A': [1, 2, 3, 4, 5],
    'B': [2, 3, 4, 5, 6],
    'C': [3, 4, 5, 6, 7]
}
df = pd.DataFrame(data)

# Select columns for correlation
columns_for_correlation = ['A', 'B', 'C']

# Compute correlation matrix
correlation_matrix = df[columns_for_correlation].corr()
#####
      A      B      C
A  1.0  1.0  1.0
B  1.0  1.0  1.0
C  1.0  1.0  1.0
```

Correlation matrix of various cryptocurrency assets.

```
correlation = dataset.corr()
plt.figure(figsize=(20, 20))
plt.title('Correlation Matrix')
sns.heatmap(correlation, vmax=1, square=True, annot=True, cmap='cubehelix')
```

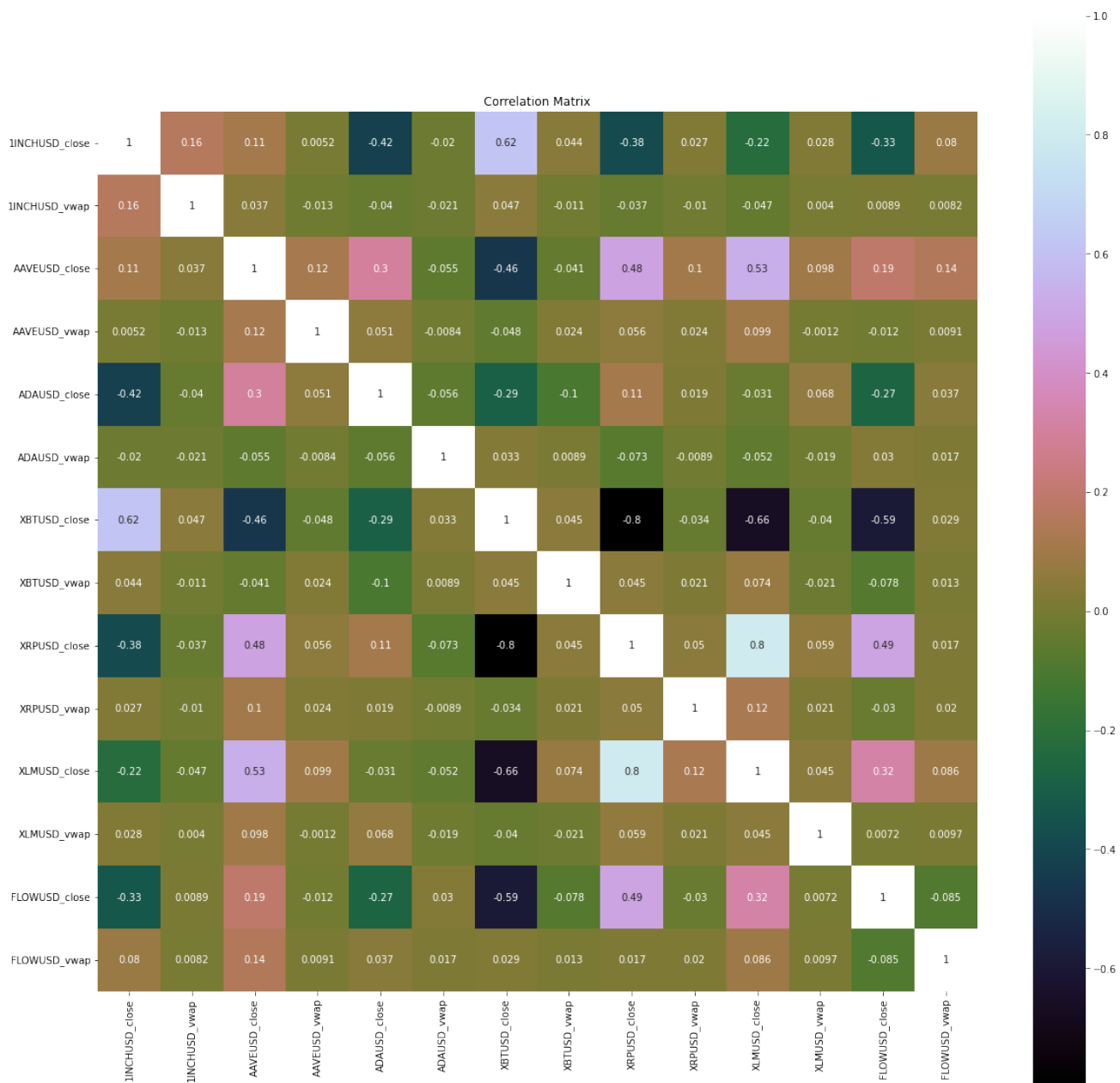


Figure 12: Correlation of various cryptocurrencies

### 2.10.34 .rolling()

Used for moving average as follows:

```
data["sma_size"] = data['Close'].rolling(window=window_size).mean()
```

### 2.10.35 .isin()

Keep only rows containing specific data

```
data = {'A': [1, 2, 3, 4, 5],
        'B': ['foo', 'bar', 'foo', 'bar', 'foo']}
selected_data_isin = df[df['B'].isin(['foo'])]
#####
A      B
```



```
0 1 foo
2 3 foo
4 5 foo
```

### 2.10.36 .at()

Get element at [row, column]

```
# Creating a sample DataFrame
data = {
    'A': [1, 2, 3],
    'B': [4, 5, 6],
    'C': [7, 8, 9]
}
df = pd.DataFrame(data, index=['X', 'Y', 'Z'])

# Accessing a single value using .at[]
value = df.at['Y', 'B']
print("Value at row 'Y' and column 'B':", value)
#####
Value at row 'Y' and column 'B': 5
```

### 2.10.37 .merge()

Outer merge might be the most useful.

```
# Creating two sample DataFrames
df1 = pd.DataFrame({'key': ['A', 'B', 'C', 'D'],
                    'value': [1, 2, 3, 4]})
df2 = pd.DataFrame({'key': ['B', 'D', 'E', 'F'],
                    'value': [5, 6, 7, 8]})

# Performing an inner merge based on the 'key' column
inner_merge = pd.merge(df1, df2, on='key', how='inner')
print("Inner Merge:")
print(inner_merge)

# Performing a left merge based on the 'key' column
left_merge = pd.merge(df1, df2, on='key', how='left')
print("\nLeft Merge:")
print(left_merge)

# Performing a right merge based on the 'key' column
right_merge = pd.merge(df1, df2, on='key', how='right')
print("\nRight Merge:")
print(right_merge)

# Performing an outer merge based on the 'key' column
outer_merge = pd.merge(df1, df2, on='key', how='outer')
print("\nOuter Merge:")
print(outer_merge)
#####
Inner Merge:
  key  value_x  value_y
0  B         2         5
1  D         4         6

Left Merge:
  key  value_x  value_y
0  A         1      NaN
```

1	B	2	5.0
2	C	3	NaN
3	D	4	6.0

Right Merge:

	key	value_x	value_y
0	B	2.0	5
1	D	4.0	6
2	E	NaN	7
3	F	NaN	8

Outer Merge:

	key	value_x	value_y
0	A	1.0	NaN
1	B	2.0	5.0
2	C	3.0	NaN
3	D	4.0	6.0
4	E	NaN	7.0
5	F	NaN	8.0

### 2.10.38 .sort\_values()

```
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Emma'],
    'Age': [25, 35, 30, 28, 40],
    'Salary': [50000, 60000, 55000, 52000, 65000]
}
df = pd.DataFrame(data)
df.sort_values(by='Age', inplace=True)
df
#####
Name  Age  Salary
0   Alice   25   50000
3   David   28   52000
2  Charlie   30   55000
1     Bob   35   60000
4     Emma   40   65000
```

### 2.10.39 .iloc()

The dataframe used to showcase is the following:

```
data = {
    'A': [10, 20, 30, 40, 50],
    'B': [100, 200, 300, 400, 500],
    'C': [1000, 2000, 3000, 4000, 5000]
}
df = pd.DataFrame(data)
```

Access row:

```
row = df.iloc[0]      # Accessing a single row
rows = df.iloc[1:4]   # Accessing multiple rows
specific_rows = df.iloc[[0, 2, 4]] # Accessing specific rows by list of indices
#####
A      10
```

```

B      100
C      1000
Name: 0, dtype: int64
   A      B      C
1  20    200   2000
2  30    300   3000
3  40    400   4000
   A      B      C
0  10    100   1000
2  30    300   3000
4  50    500   5000

```

#### 2.10.40 .ewm() (Exponentially Weighted Moving Average)

```

data = pd.Series([10, 12, 15, 18, 22, 25, 28])
ewm = data.ewm(span=3, adjust=False)
ewma = ewm.mean()

def EMA(df, n):
    EMA = pd.Series(df['Close'].ewm(span=n, min_periods=n).mean(), name='EMA_' + str
                    (n))
    return EMA

```

Access column:

```

column = df.iloc[:, 0]          # Accessing a single column
columns = df.iloc[:, [0, 2]]    # Accessing multiple columns
range_of_columns = df.iloc[:, 1:4] # Accessing a range of columns
#####
0      10
1      20
2      30
3      40
4      50
Name: A, dtype: int64
   A      C
0  10  1000
1  20  2000
2  30  3000
3  40  4000
4  50  5000
   B      C
0  100  1000
1  200  2000
2  300  3000
3  400  4000
4  500  5000

```

Access specific element:

```

element = df.iloc[0, 1]          # Accessing a specific element
subset = df.iloc[1:4, [0, 2]]    # Accessing a subset of rows and columns
#####
100
   A      C
1  20  2000
2  30  3000

```

```
3 40 4000
```

#### 2.10.41 MACD

```
short_ema = df['Close'].ewm(span=short_window, adjust=False).mean() # ewm =  
    exponentially weighted .mean() to turn it into EMA, adjust=False stands for  
    equal weight, otherwise recent observations have more weight  
long_ema = df['Close'].ewm(span=long_window, adjust=False).mean() # if adjust = True  
    weights are adjusted to account for 1 when summed, e.g. SMA of 20 days on the  
    first 5 days will be lower  
macd = short_ema - long_ema  
signal_line = macd.ewm(span=signal_window, adjust=False).mean()  
macd_histogram = macd - signal_line
```

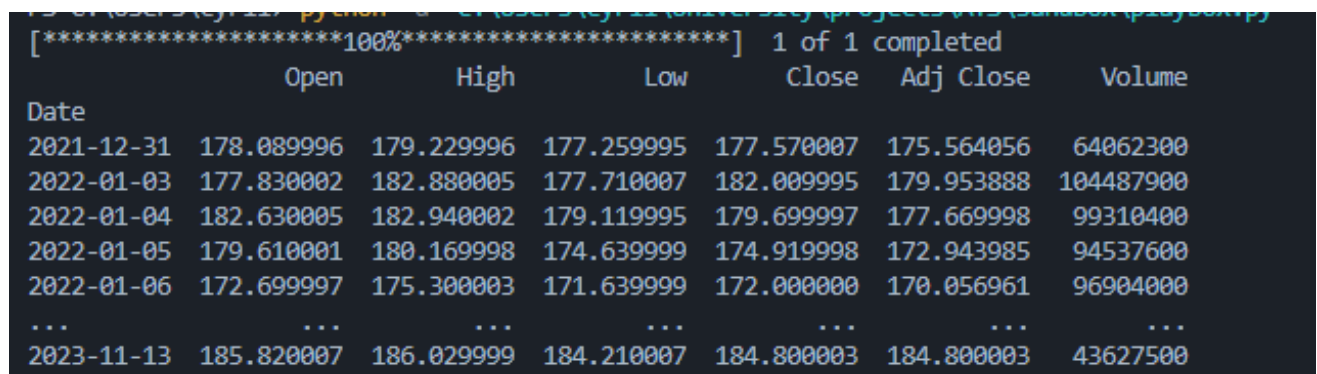
### 2.11 YFinance

Import module

```
import yfinance as yf
```

Gather stock price data with time interval

```
# Define symbol  
stock_symbols = ["AAPL"]  
  
# Define start and end date  
start_date = "2022-01-01"  
end_date = "2023-11-30"  
  
# Fetch stock data from Yahoo Finance  
stock_data = yf.download(stock_symbols, start=start_date, end=end_date)  
  
# format it to a Pandas DataFrame  
stock_df = pd.DataFrame(stock_data)
```



[*****100%*****] 1 of 1 completed						
Date	Open	High	Low	Close	Adj Close	Volume
2021-12-31	178.089996	179.229996	177.259995	177.570007	175.564056	64062300
2022-01-03	177.830002	182.880005	177.710007	182.009995	179.953888	104487900
2022-01-04	182.630005	182.940002	179.119995	179.699997	177.669998	99310400
2022-01-05	179.610001	180.169998	174.639999	174.919998	172.943985	94537600
2022-01-06	172.699997	175.300003	171.639999	172.000000	170.056961	96904000
...	...	...	...	...	...	...
2023-11-13	185.820007	186.029999	184.210007	184.800003	184.800003	43627500

Figure 13: YFinance results for scraping AAPL stock prices

## 3 Major APIs

### 3.1 Kraken

#### 3.1.1 Required modules

```
import requests
import json
```

#### 3.1.2 Pair info

```
resp = requests.get('https://api.kraken.com/0/public/Ticker?pair=TIAUSD')
resp.json()
#####
{'error': [], 'result': {'TIAUSD': {'a': ['16.94520000', '60', '60.000'], 'b': ['16.94510000', '1', '1.000'], 'c': ['16.97500000', '53.88471'], 'v': ['4869.80066', '197239.90133'], 'p': ['17.01820934', '17.22717403'], 't': [59, 3192], 'l': ['16.97500000', '16.69990000'], 'h': ['17.14220000', '17.81600000'], 'o': '16.99000000'}}}}
```

#### 3.1.3 OHLC data

```
resp = requests.get('https://api.kraken.com/0/public/OHLC?pair=TIAUSD&interval=1440')
resp.json()
#####
[ ... , '0.00000', 0], [1709082840, '16.9381', '16.9381', '16.9381', '16.9381', '0.0000', '0.00000', 0]], 'last': 1709082780}}
```

URL parameters are: (to be separated with &)

- Pair: seems obvious
- Interval: time interval in minutes

#### 3.1.4 Historical dataframe

```
csv_file_path = "C:\\Users\\cyril\\University\\projects\\QFT\\datasets\\Kraken_OHLCVT\\TIAUSD_1.csv"
ohlcv_data = pd.read_csv(csv_file_path, header=0, names=["DateTime", "Open", "High", "Low", "Close", "Volume", "Trades"])
ohlcv_data.head()
```

#### 3.1.5 Pair price dataframe

```
pair = "TIAUSD"
url = f'https://api.kraken.com/0/public/OHLC?pair={pair}&interval=1'
response = requests.get(url)
data = response.json()
```

```
data = data['result']['TIAUSD']
df = pd.DataFrame(data, columns=['Time', 'Open', 'High', 'Low', 'Close', 'VWAP', 'Volume', 'Count']) # Convert to DataFrame
print(df)
#####
Time      Open      High      Low      Close      VWAP      Volume      Count
0      1709040360      17.3824      17.3824      17.3130      17.3130      17.3635      692.20434      27
1      1709040420      17.3255      17.3255      17.3062      17.3062      17.3202      11.47762      3
2      1709040480      17.2873      17.3330      17.2819      17.3220      17.2846      526.38073      7
3      1709040540      17.2807      17.2807      17.2806      17.2806      17.2806      24.57475      2
4      1709040600      17.3506      17.3721      17.3506      17.3663      17.3666      309.78807      3
..      ...      ...      ...      ...      ...      ...      ...      ...
715      1709083260      16.9507      16.9507      16.9507      16.9507      0.0000      0.00000      0
716      1709083320      16.9589      16.9601      16.9589      16.9601      16.9597      235.80317      3
717      1709083380      16.9601      16.9601      16.9601      16.9601      0.0000      0.00000      0
718      1709083440      16.9601      16.9601      16.9601      16.9601      0.0000      0.00000      0
719      1709083500      16.9389      16.9389      16.9389      16.9389      16.9389      2.36142      1
```

### 3.1.6 Status code

Get the status code of the api call (prey for 200)

```
response = requests.get(url)
response.status_code
```

## 4 Algorithms

### 4.1 Hashing

#### 4.1.1 Hash function

Mathematical function that takes an input/key and produces a fixed-size string of byte. The output (a number) is called hash code/value.

#### 4.1.2 Hash function characteristics

**Deterministic** because the same input produces the same output, the output is of **fixed size**, it must be collision resistant because it is **hard to find different inputs that produce the same output**, must be prone to avalanche effect, in other words a **small change in input changes drastically the output**.

#### 4.1.3 Hash buckets

## 5 Finance

### 5.1 Drawdown

```
df['cumulative_max'] = df['values'].cummax()
df['cumulative_sum'] = df['values'].cumsum()
df['drawdown'] = df['cumulative_max'] - df['cumulative_sum']
```

## 5.2 Technical Indicators

### 5.2.1 SMA

```
dataset['short_mavg'] = dataset['Close'].rolling(window=10, min_periods=1, center=False).mean()
```

### 5.2.2 EMA

```
def EMA(df, n):
    EMA = pd.Series(df['Close'].ewm(span=n, min_periods=n).mean(), name='EMA_' + str(n))
    return EMA

dataset['EMA10'] = EMA(dataset, 10)
dataset['EMA30'] = EMA(dataset, 30)
dataset['EMA200'] = EMA(dataset, 200)
```

### 5.2.3 ROC (Rate of Change)

```
#calculation of rate of change
def ROC(df, n):
    M = df.diff(n - 1)
    N = df.shift(n - 1)
    ROC = pd.Series(((M / N) * 100), name = 'ROC_' + str(n))
    return ROC

dataset['ROC10'] = ROC(dataset['Close'], 10)
dataset['ROC30'] = ROC(dataset['Close'], 30)
```

### 5.2.4 BBands (Bollinger Bands)

```
window = 20
def calculate_bollinger_bands(prices, window=window, num_std_dev): # lookback period
    and standard deviation factor
    # Calculate rolling mean (middle band)
    rolling_mean = np.convolve(prices, np.ones(window) / window, mode='valid') ####
        create serie of ones of lenght "window"

    # Calculate rolling standard deviation
    rolling_std = np.std([prices[i:i+window] for i in range(len(prices) - window +
        1)], axis=1) #### standard deviation using numpy

    # Calculate upper and lower bands
    upper_band = rolling_mean + num_std_dev * rolling_std
    lower_band = rolling_mean - num_std_dev * rolling_std

    return rolling_mean, upper_band, lower_band

# Calculate Bollinger Bands
middle_band, upper_band, lower_band = calculate_bollinger_bands(df["Close"])
```

### 5.2.5 RSI

```
avg_gain_values = [] # track avg gains for visualization purposes
avg_loss_values = [] # track avg losses for visualization purposes
rsi_values = [] # track computed RSI values
last_price = 0 # current_price - last_price > 0 => gain. current_price -
last_price < 0
for close_price in prices:
    if last_price == 0:
        last_price = close_price

    gain_history.append(max(0, close_price - last_price))
    loss_history.append(max(0, last_price - close_price))
    last_price = close_price

    if len(gain_history) > time_period: # maximum observations is equal to lookback
        period
        del (gain_history[0])
        del (loss_history[0])

    avg_gain = stats.mean(gain_history) # average gain over lookback period
    avg_loss = stats.mean(loss_history) # average loss over lookback period
    avg_gain_values.append(avg_gain)
    avg_loss_values.append(avg_loss)
    rs = 0
    if avg_loss > 0: # to avoid division by 0, which is undefined
        rs = avg_gain / avg_loss

    rsi = 100 - (100 / (1 + rs))
    rsi_values.append(rsi)
```

### 5.2.6 MOM (MOMentum)

```
prices = df['High']
momentum = []

for i in range(1, len(prices)):
    momentum.append(prices[i]-prices[i-1])
```

### 5.2.7 STOK & STOD

```
def STOK(close, low, high, n):
    STOK = ((close - low.rolling(n).min()) / (high.rolling(n).max() - \
    low.rolling(n).min())) * 100
    return STOK
def STOD(close, low, high, n):
    STOK = ((close - low.rolling(n).min()) / (high.rolling(n).max() - \
    low.rolling(n).min())) * 100
    STOD = STOK.rolling(3).mean()
    return STOD

dataset['%K10'] = STOK(dataset['Close'], dataset['Low'], dataset['High'], 10)
dataset['%D10'] = STOD(dataset['Close'], dataset['Low'], dataset['High'], 10)
dataset['%K30'] = STOK(dataset['Close'], dataset['Low'], dataset['High'], 30)
dataset['%D30'] = STOD(dataset['Close'], dataset['Low'], dataset['High'], 30)
dataset['%K200'] = STOK(dataset['Close'], dataset['Low'], dataset['High'], 200)
```



```
dataset['%D200'] = STOD(dataset['Close'], dataset['Low'], dataset['High'], 200)
```

## 6 Data Science

### 6.1 Lifecycle

#### 6.1.1 Stages

- ◇ Ask a question: must narrow a broad question into one that can be answered with data, amongst four categories (descriptive (e.g. how have house prices changed over time), exploratory (e.g. which aspects of a house drive its price), inferential, predictive)
- ◇ Obtain the data: most of the time expensive/hard to gather
- ◇ Understand the data: plots to uncover patterns, summarize visually
- ◇ Understand the world:

#### 6.1.2 Data scope

Range of questions that can be answered is limited with a given amount of data.

#### 6.1.3 Target population, access frame, sample

Target population: collection of elements we intend to describe/draw conclusion about (of which one element is called an atom/unit)

Access frame: collection of elements accessible for measurement and observation (e.g. in a vote, asking by phone to someone who doesn't vote is in the frame but trying to phone someone that doesn't answer to unknown numbers is out of frame) Sample: subset of units taken from the access frame to observe and measure.

#### 6.1.4 Protocols

Instrument being used to take the measurements and the procedure for taking them.

#### 6.1.5 Accuracy

Bias:

Precision:

### 6.2 Bias

#### 6.2.1 Coverage bias

Target frame doesn't include every target population (units) (e.g. survey by phone can't reach those without phones).

#### 6.2.2 Selection bias

Mechanism used to choose units for the sample tends to select more of certain units rather than others (e.g. self selection with volunteers to participate in the study).

#### 6.2.3 Nonresponse bias

Unit: someone from selected sample is unwilling to participate (e.g. someone who doesn't answer phone calls)

Item: someone is willing to participate but not to answer (e.g. someone answers the phone but doesn't answer)

#### 6.2.4 Measurement bias

Instrument systematically misses the target (in the same direction otherwise correction occurs) (e.g. can occur with sensors but also when making a survey if people are not at ease giving honest answers or even just questions wrongly formulated).

## 6.3 Variations

### 6.3.1 Sampling variation

Using chance to select a sample.

### 6.3.2 Assignment variation

When splitting units into different treatment groups. Thus changing the way we split into treatment groups will affect results.

### 6.3.3 Measurement variation

If an instrument commits errors that are normally distributed around the the truth, the result is also centered around the truth.

## 6.4 Sampling

### 6.4.1 Stratified sampling

Divide the population into nonoverlapping groups (named strata of which one group is named stratum)

### 6.4.2 Cluster sampling

Divide the population into nonoverlapping subgroups

## 6.5 Distributions

### 6.5.1 Hypergeometric distribution

Drawing without replacement.

### 6.5.2 Multivariate Hypergeometric distribution

For example a poll with 3 possible votes.

### 6.5.3 Binomial distribution

Drawing with replacement.

## 7 Machine Learning

### 7.1 Lingo

- ◆ Collaborative filtering: guess what user might like based on what other similar user liked.
- ◆ Dataset: row = sample, column = properties/features
- ◆ Deterministic: can be perfectly determined based on initial conditions, no stochasticity implied
- ◆ Feature = Predictor: input data (column wise)
- ◆ Holdout set: test set
- ◆ Validation set: intermediate holdout set
- ◆ Inference time: time required to make a prediction
- ◆ Label: output data
- ◆ Lazy learner: no learning required (such as KNN)
- ◆ Nested vs flat dataset: nested is a dataset that contains datasets itself and a flat one is one that does not
- ◆ Multiclass: pick one choice amongst multiple classes to belong to (e.g. a digit can be a 3 or a 2)
- ◆ Multilabel: pick multiple choices amongst multiple classes to belong to (e.g. photo containing cat or dog)

## 7.2 Bias

### 7.2.1 Data snooping bias

Adapting machine learning model to specific spotted pattern, thus exposing ourselves to overfitting. In other words using data multiple times and pretend the data is independent.

### 7.2.2 Selection bias

Certain group of individuals systematically excluded from sample in non-random manner. Can happen by self-selection where individuals want or do not want to bring data to the dataset. Stratified sampling (taking random sample representative of each part of possible sample) solves this problematic issue.

### 7.2.3 Sampling bias

Selected sample overrepresents or underrepresents group of individuals. Stratified sampling (taking random sample representative of each part of possible sample) solves this problematic issue.

## 7.3 Classification vs. Regression

### 7.3.1 Classification

Binary if two possible classes, multiclass if more than two. Discrete prediction problems.

### 7.3.2 Multiclass classification

Model picks on group amongst many.

#### One-vs-one

**One-vs-rest** Suppose we have multiple labels such as [l1, l2, l3], one-vs-rest consists of taking each label and creating a model to ask whether it is or not the said label (here: l1 or not, l2 or not, l3 or not). Thus allowing binary models to solve multiclass classification problems.

### 7.3.3 Singlelabel vs. Multilabel

**Singlelabel** Each instance assigned to only one class.

**Multilabel** Instance belongs to multiple classes.

### 7.3.4 Regression

Continuous data to predict.

## 7.4 Supervised vs. Unsupervised

### 7.4.1 Supervised

Goal is to minimize cost function. Learn from input/output pairs, easier to measure performance ex: zip code reading on envelope, determine if tumor is benign, fraudulent activity in credit card transactions

### 7.4.2 Unsupervised

Only input is known, harder to evaluate ex: stock market mood, grouping customers by preferences and potential purchases, ...

## 7.5 Supervised Models

Data is labeled (i.e. there is an expected answer), most common supervised tasks are regression and classification. Clustering if expected labels are unknown but classification if those are known.

Supervised models are either about regression or about classification. While classification problems are probability, regression problems on the other hand output a continuous set of possible outcomes.

### 7.5.1 Logistic regression

Primarily a **classification model**, despite confusing name. Fits an equation to the distribution of data.

### 7.5.2 LDA (Linear Discriminant Analysis)

### 7.5.3 Polynomial regression

Import:

```
from sklearn.preprocessing import PolynomialFeatures
```

Fit:

```
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
```

### 7.5.4 DecisionTreeRegressor

Works on non-linear data relationships. Can be used both for regression and classification

Import:

```
from sklearn.tree import DecisionTreeRegressor
```

Build model:

```
tree_regressor = DecisionTreeRegressor()
```

Fit:

```
tree_regressor.fit(X, Y)
```

Predict: input X values outputs y

```
prediction = tree_regressor.predict(X)
```

## 7.6 Semisupervised Models

### 7.6.1 deep belief networks (DBNs)

Based on unsupervised restricted Boltzmann machines (RBMs) stack on top of one another then system as a whole is fine-tuned using supervised learning techniques.

## 7.7 Unsupervised Models

Data is unlabeled (i.e. there is no expected answer), most of the data in today's world are this way, we have  $X$  label but not  $y$  label. Common unsupervised tasks include clustering, visualization, dimensionality reduction, and association rule learning. Permits to save a lot of time (e.g. a factory gets pictures of thousands of manufactured pictures but are not labeled whether or not they are defective)

**Clustering** Group similar instances together into clusters. Commonly used in finance to find homogenous classes of assets.

**Anomaly Detection** Learn what normal is and then based on this "normal" template detect outliers/abnormal samples.

**Density Estimation** Task is to estimate the PDF (Probability Density Function) of the random process that generated the dataset (commonly used for anomaly detection, low probability instances might be outliers).

### 7.7.1 K-means

For clustering. Lloyd-Forgy method, bad for non-spherical clusters, works better with similar-size clusters. Finds  $K$  centroids and assigns each data point to exactly one cluster, the goal of the model is to minimize the within-cluster variance (inertia) based on euclidean distance.

**Overview** Initialization is made with random centroids selection. Each data point is assigned to a cluster's center (put simply a centroid). Cluster centers are updates to the mean of the assigned points. Process is repeated until stabilization is acquired. Advantages are fast convergence and simplicity as well as scalability to large dataset and construction of evenly sized clusters. Only default is knowing the number of clusters.

#### Hyperparameters

- ◇ number of clusters:
- ◇ maximum iterations: if doesn't converge to solution must stop somewhere
- ◇ number initial: number of times the algorithm will be run with different centroids seeds (default is at least 10 iterations).

#### Finding optimal number of clusters in kmeans

- Elbow method: Based on the sum of squared errors (SSE) within clusters.
- Silhouette method: Based on the silhouette score. Measures similarity between a point and its cluster (cohesion) vs other clusters (separation).

**Hard clustering** Assigns to a single cluster

**Soft clustering** Assigns a score per cluster (e.g. score can be distance to centroid, GRBF (Gaussian Radial Basis Function))

### 7.7.2 Hierarchical Clustering

Clusters have predominant ordering from top to bottom **tf does it even mean?**. No need to provide the number of clusters.

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=5)
y_pred = kmeans.fit_predict(X)
kmeans.cluster_centers_
```

Disadvantage is having to explicitly specify the number of clusters to be found. Then to make predictions the following can be used:

```
kmeans.predict(X_new)
```

Returns the distance to all the cluster centroids.

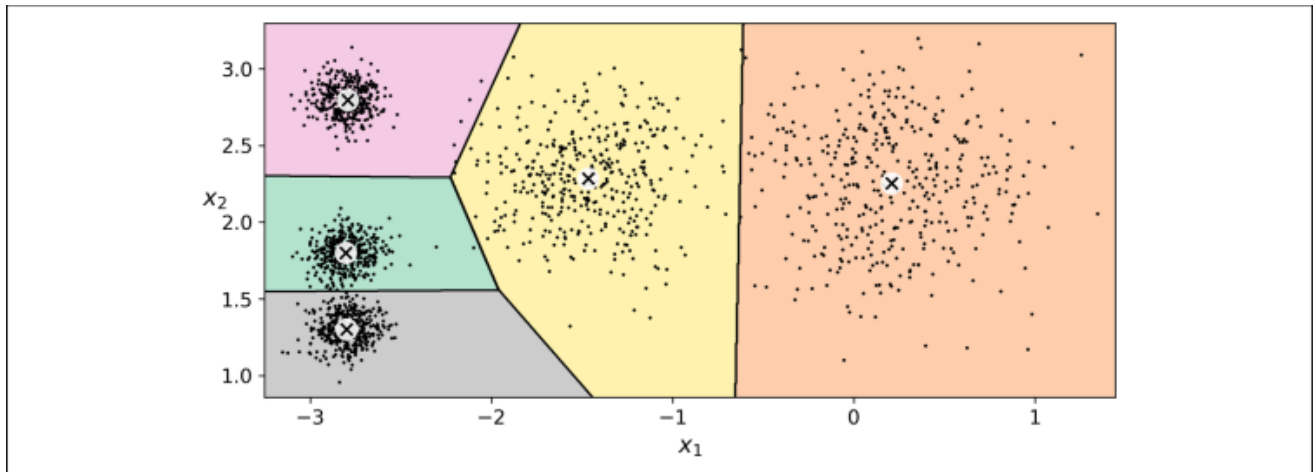


Figure 14: KMeans Clustering, where rounded x represents centroids

Define initial centroids

```
good_init = np.array([[ -3, 3], [ -3, 2], [ -3, 1], [ -1, 2], [ 0, 2]])
kmeans = KMeans(n_clusters=5, init=good_init, n_init=1)
```

*init = "random"* can be used to avoid doing this step.

Charles Elkan proposed to avoid some distance calculations and this model is used by default but original model can be used by setting hyperparameter `algorithm="full"`.

### 7.7.3 K-Means mini-batches

David Sculley, useful if dataset does not fit in memory.

```
from sklearn.cluster import MiniBatchKMeans
minibatch_kmeans = MiniBatchKMeans(n_clusters=5)
minibatch_kmeans.fit(X)
```

### 7.7.4 Agglomerative Clustering

Most common type, group instances based on similarity. Algorithms works by creating  $N$  clusters (where each instances is a cluster itself (to put simply  $N$  is the number of instances)), then the two closest data points are taken and merged into one bigger cluster (proceeding to agglomeration).

```
from sklearn.cluster import AgglomerativeClustering
from scipy.cluster.hierarchy import dendrogram, linkage
import matplotlib.pyplot as plt

# Fit the model
model = AgglomerativeClustering(n_clusters=4, linkage='ward')
clust_labels1 = model.fit_predict(dataset[-50:])

# Generate linkage matrix
```

```

linkage_matrix = linkage(dataset, method='ward', metric='euclidean') #### ward, We use ward as the method since it minimizes the variance of distances between the clusters

# Plot dendrogram
plt.figure(figsize=(10, 7))
dendrogram(linkage_matrix)
plt.title('Hierarchical Clustering Dendrogram')
plt.xlabel('sample index')
plt.ylabel('distance')
plt.show()

```

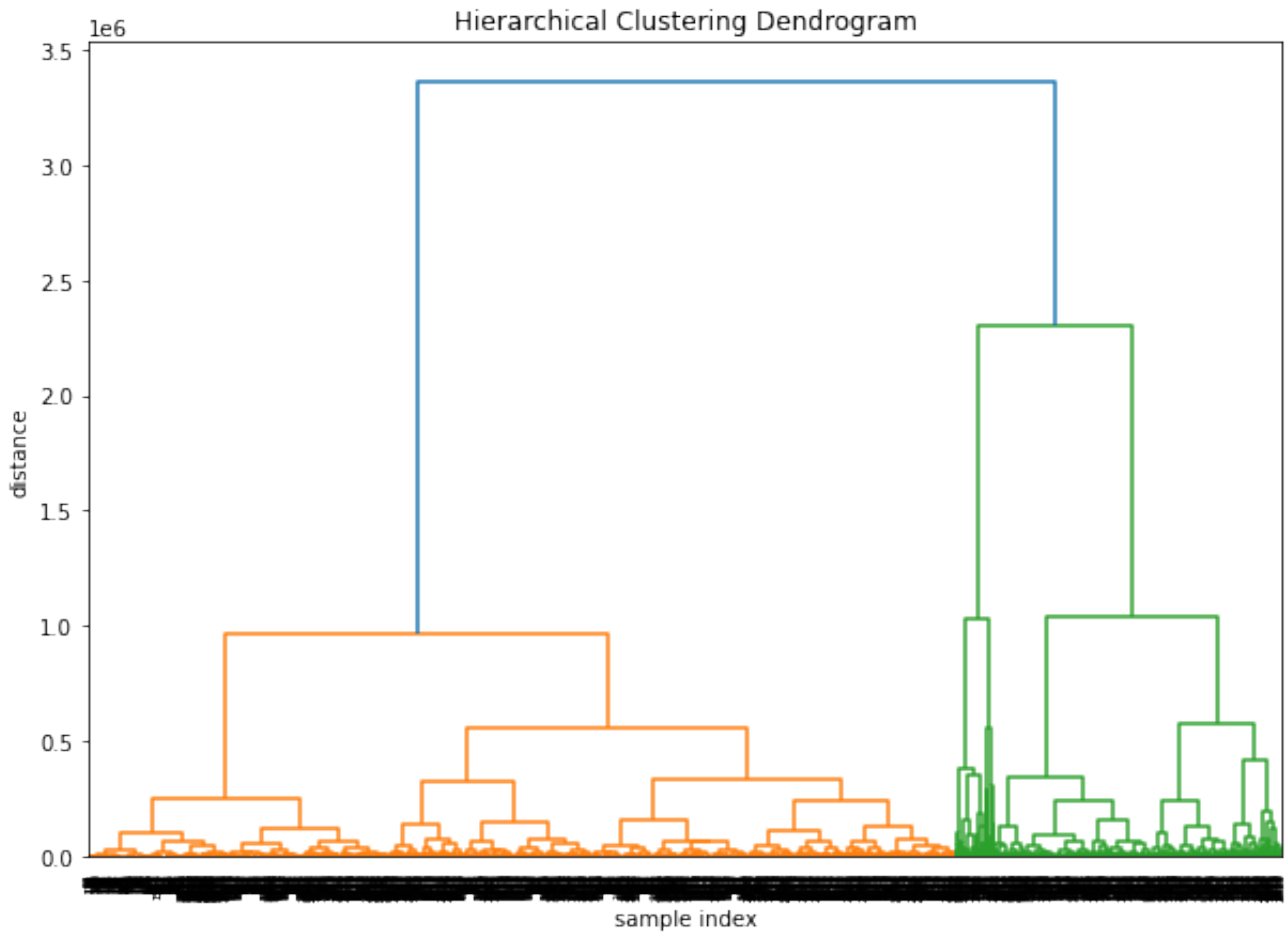


Figure 15: Same dataset rotated 45° shows high sensitivity

**Divisive hierarchial clustering** Oppostive of agglomerative.

#### 7.7.5 AffinityPropagation

```

from sklearn.cluster import AffinityPropagation
# Initialize the algorithm and set the number of PC's
ap = AffinityPropagation()
ap.fit(dataset)

```

### 7.7.6 DBSCAN (Density-Based Spatial Clustering of Applications with Noise)

Clustering for arbitrary shape. The algorithm takes a point/instance and sees the neighborhood  $\epsilon$ , if the  $\epsilon$  has at least *min - samples* instances in its neighborhood then  $\epsilon$  is considered core instance (thus they are located in dense regions). Works well is clusters are dense enough and separated by low density regions. *eps* is considered the maximum distance between two instances to be considered of the same class. Based on density of points.

```
from sklearn.cluster import DBSCAN
dbscan = DBSCAN(eps=0.05, min_samples=5)
dbscan.fit(X)
dbscan.labels_
```

```
dbscan.core_sample_indices_
808
```

```
>>> dbscan.core_sample_indices_
array([ 0,  4,  5,  6,  7,  8, 10, 11, ..., 992, 993, 995, 997, 998, 999])
```

```
>>> dbscan.components_
array([[ -0.02137124,  0.40618608],
       [ -0.84192557,  0.53058695],
       ...,
       [ -0.94355873,  0.3278936 ],
       [  0.79419406,  0.60777171]])
```

### 7.7.7 HDBSCAN (Hierarchical DBSCAN)

<https://github.com/scikit-learn-contrib/hdbscan/>

### 7.7.8 Agglomeration Clustering

Small bubbles attached to one another until it forms a bigger cluster (like merging a tree's branches).

### 7.7.9 BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies)

Designed for big datasets with few features (¡20).

### 7.7.10 Mean-Shift

This algorithm starts by placing a circle centered on each instance; then for each circle it computes the mean of all the instances located within it, and it shifts the circle so that it is centered on the mean. Next, it iterates this mean-shifting step until all the circles stop moving (i.e., until each of them is centered on the mean of the instances it contains)

### 7.7.11 Affinity Propagation

Voting system, does not require number of clusters to be known beforehand.

### 7.7.12 Spectral Clustering

Similarity matrix between instances to create low-dimensional matrix of it then uses other clustering method in the low-dimensional space.



### 7.7.13 GMM (Gaussian Mixtures Model)

Assumption that instances were generated from a mixture of several Gaussian distributions. Anomaly detection using GMM can be performed before training another model on the cleaned data. To do so density threshold is defined and instances in the low density region are kicked. **MLE (Maximum Likelihood Estimation), MAP (Maximum A-Posteriori), BIC, AIC with `gm.big()` and `gm.aic()`** Advance is that shape can be imposed.

```
from sklearn.mixture import GaussianMixture
gm = GaussianMixture(n_components=3, n_init=10)
gm.fit(X)
>>> gm.weights_
array([0.20965228, 0.4000662 , 0.39028152])
>>> gm.means_
array([[ 3.39909717, 1.05933727],
       [-1.40763984, 1.42710194],
       [ 0.05135313, 0.07524095]])
>>> gm.covariances_
array([[[ 1.14807234, -0.03270354],
        [-0.03270354, 0.95496237]],
       [[ 0.63478101, 0.72969804],
        [ 0.72969804, 1.1609872 ]],
       [[ 0.68809572, 0.79608475],
        [ 0.79608475, 1.21234145]]])
```

Shape can be imposed: "spherical" all clusters must be spherical, but they can have different diameters (i.e., different variances). "diag" clusters can take on any ellipsoidal shape of any size, but the ellipsoid's axes must be parallel to the coordinate axes (i.e., the covariance matrices must be diagonal). "tied" All clusters must have the same ellipsoidal shape, size, and orientation (i.e., all clusters share the same covariance matrix).  
Outliers identification

```
densities = gm.score_samples(X)
density_threshold = np.percentile(densities, 4)
anomalies = X[densities < density_threshold]
```

### 7.7.14 Fast-MCD (Fast-Minimum Covariance Determinant)

#### 7.7.15 Isolation forest

#### 7.7.16 LOF (Local Outlier Factor)

#### 7.7.17 One-Class SVM

#### 7.7.18 Hierarchical Cluster Analysis (HCA)

Clustering

#### 7.7.19 One-class SVM

Anomaly detection and novelty detection

#### 7.7.20 Isolation Forest

Anomaly detection and novelty detection

#### 7.7.21 Apriori

Association rule learning

#### 7.7.22 Eclat

Association rule learning

## 7.8 Ensemble Learning

”there is wisdom in the crowd”

### 7.8.1 Voting classifiers

Suppose we have 4 models, each with 80% accuracy in their predictions, comparing their predictions and keeping the one occurring the most often is called hard voting (majority-vote). Accuracy somehow often reaches higher level than with the best individual model. Even a sufficient list of weak learners (less precise than random) can become a strong learner if enough models are combined (cf. law of large numbers), for example a list of models with each an accuracy of 51% can expect a global accuracy of around 75% (nevertheless models must be independent to make uncorrelated errors). Soft voting can also be used if all models have a `predict_proba` method, soft-voting often leads to higher accuracy predictions because it gives more weight to highly confident votes, to set up soft proba, hyperparameters arguments must be the following: `voting=’soft’` and `probability=True` (to have probability between 0 and 1).

### 7.8.2 Bagging (Bootstrap Aggregating)

Used for models with high variance. Training same algorithm on different subsets of the training set then combining as if they were different predictors (can be done in parallel). Resulting algorithm has an equivalent bias but lower variance. Another advantage is that each small subset can be trained on a separated core/server. Automatically uses soft voting if probability is available (unlike voting classifiers)

```
from sklearn.ensemble import BaggingClassifier
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1)
```

Where `n_jobs` sets the number of cores to use, -1 uses as many cores as available.

### 7.8.3 Boosting

Emphasis on learning the examples the previous model got wrong. Residual models (correcting the error) are typically not very complicated.

### 7.8.4 AdaBoost (Adaptive Boosting)

Boosting method (referenced in 7.8.3) algorithm. Correct precursor by paying attention to instances it underfitted and tweaking weights to adjust result. Default is it doesn’t scale well. (sklearn uses SAMME (Stagewise Additive Modeling using a Multiclass Exponential loss function) AdaBoost algorithm) **mathematics behind: negative weights allowed?**

### 7.8.5 Gradient Boosting

Boosting method (referenced in 7.8.3) algorithm. Same principle as AdaBoost but instead fits to residual errors made by the previous predictor **GBRT is supervised and makes use of this but other algorithms also do, which?**

### 7.8.6 XGBoost

Boosting method (referenced in 7.8.3) algorithm.

```
import xgboost

# train an XGBoost model
X, y = shap.datasets.california()
model = xgboost.XGBRegressor().fit(X, y)
```

### 7.8.7 Pasting

**i believe there is a misconception with bagging**

### 7.8.8 Stacking (Stacked Generalization)

Combines output of a collection of models to make a prediction. All models are trained then a model is trained using output of all models as feature. The upper-level model's goal is to learn how to best combine the outcomes of the initial models. No hard-voting but model is trained to perform best aggregation possible. The way it works is by gathering all predictions and blending them into a single output prediction.

### 7.8.9 Out-of-Bag (oob)

With bagging some instances of training set can be used multiple times (or not at all). Set `oob_score=True` to get score without having to create separated training/testing set.

### 7.8.10 Random patches

too complicated for now but need to learn about it

### 7.8.11 Random subspaces

too complicated for now but need to learn about it

## 7.9 Ensemble Models

### 7.9.1 GBRT (Gradient Boosted Regression Trees)

### 7.9.2 Voting Classifier

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()
voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
voting_clf.fit(X_train, y_train)

from sklearn.metrics import accuracy_score

for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
#####
LogisticRegression 0.87
RandomForestClassifier 0.825
SVC 0.875
VotingClassifier 0.875
```

**Hyperparameters** Set voting to *soft* for probability assigned to each class and hard to go binary.

### 7.9.3 GDM (Gradient Boosting Machine)

### 7.9.4 AdaBoost (Adaptive Boosting)

```
from sklearn.ensemble import AdaBoostClassifier
ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5)
```

```
ada_clf.fit(X_train, y_train)
ada_clf.fit(X_train, y_train)
predictions = ada_clf.predict(X_test)
```

## 7.10 Batch Learning vs. Online Learning

### ◇ Batch:

Incapable of learning incrementally, trained then used. Called offline learning. Therefore this solution is not advised for usecases where fast reaction is required such as for stock prices prediction. Useful when training everyday is not sustainable or carrying a lot of data to train on the go is impossible e.g. a rover on mars or a smartphone.

### ◇ Online (incremental learning):

Consists of training model incrementally with mini-batches (advantage is that learning step is fast and cheap). To save space this method allows throwing data away once used to train.

Allows usage of "out-of-core" learning where algo runs training step on slice of data and repeat until it has trained on all the data, practical to handle vast quantities of data a machine can't handle alone.

Learning rate can be set to decide how fast or slow model must learn, thus to be less or more sensitive to sudden outliers.

## 7.11 Instance-Based vs. Model-Based Learning

Inference: make predictions on new cases.

### ◇ Instance-Based:

Based on measure of similarity it makes prediction. e.g. the count of words in a mail

### ◇ Model-Based Learning:

Search for an optimal value for the model parameters such that the model will generalize well to new instances. We usually train such systems by minimizing a cost function that measures how bad the system is at making predictions on the training data, plus a penalty for model complexity if the model is regularized. To make predictions, we feed the new instance's features into the model's prediction function, using the parameter values found by the learning algorithm.

## 7.12 Performance Evaluation

### 7.12.1 No Free Lunch (NFL) theorem

If we make no assumption about the data, nothing happens, that is why we must try to think how the data behaves and thus use the most convenient model (e.g. the data evolves in a linear way)

### 7.12.2 Bias

Low performance because model is **not fitting well**.

### 7.12.3 Variance

Due to model's excessive adaptation to small variations in training data (high degree polynomial regression **tend to overfit** thus have more variance).

### 7.12.4 Bias-variance trade-off

Increasing model complexity will increase bias and reduce variance but cannot get both at the same time.

### 7.12.5 Testing and validating

Using split of data around 80/20 ratio, can show how model performs on "never seen before data". Few errors on training but a lot in testing is an indicator of overfitting. **holdout validation**

### 7.12.6 Cross-validation

Comes at cost of training multiple times.

### 7.12.7 Irreducible error

Due to inherent noise in data, only way to reduce it is to clean the data (remove outliers, ...).

### 7.12.8 Data drift

Current data with previously fitted model doesn't work anymore because data changed.

### 7.12.9 Cost/Loss Function

Used to calculate error committed by the model.

### 7.12.10 Evaluation Metrics

evaluation metrics

## 7.13 Cost/Loss Function Algorithms

Calculations for cost functions are detailed in 7.12.9

### 7.13.1 MSE (Mean Squared Error)

More sensitive to unusually large errors.

```
from sklearn.metrics import mean_squared_error
mse = mean_squared_error(y_test, y_pred)
```

And is calculated as follows:

$$L(\theta, \mathbf{y}) = \frac{1}{n} \sum_{i=1}^n (y_i - \theta)^2$$

Where  $y_i$  is the given output and  $\theta$  is the expected value.

### 7.13.2 RMSE (Root Mean Squared Error)

Typical for regression problems, higher weight given to large errors. Or simply Euclidean norm, commonly noted  $\|\mathbf{x}\|$ .

$$RMSE(X, h) = \sqrt{\frac{1}{m} \sum_{i=1}^m (h(x^{(i)}) - y^{(i)})^2}$$

Where  $m$  is the number of instances (e.g. 2000 cars data for estimating car values),  $x^{(i)}$  is the feature vector of the  $i^{th}$  instance of the dataset and  $y^{(i)}$  the expected output.

Feature vector is written like mathematical vector, indeed is vertical like this:

$$\mathbf{x}^{(i)} = \begin{pmatrix} 118.29 \\ 33.91 \\ 1833.29 \end{pmatrix}$$

As well as expected output is (e.g):

$$y^{(i)} = 156$$

The  $X$  matrix is the matrix of all feature vectors

$$X = \begin{pmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ \vdots \\ (x^{(n)})^T \end{pmatrix}$$

And finally,  $h$  being the prediction function, indeed the output of  $h(x^{(i)})$  is noted  $\hat{y}^{(i)}$ .

```
from sklearn.metrics import mean_squared_error
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
```

### 7.13.3 MAE (Mean Absolute Error)

Less sensitive to unusually large errors.

Also called average absolute deviation. Less dependencies to outliers. Nicknamed Manhattan error, gives sum of errors but doesn't indicate if over or under fitting.

$$MAE(\theta, \mathbf{y}) = \frac{1}{n} \sum_{i=1}^n |y_i - \theta|$$

Where  $\theta$  is the expected value and  $y_i$  is the  $i$ th output given by the model in a sample of  $n$  predictions. The minimizing value is the median.

### 7.13.4 Huber loss

$$L(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{for } |y - \hat{y}| \leq \delta \\ \delta(|y - \hat{y}| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

Where  $y$  is the true value,  $\hat{y}$  is the predicted value  $\delta$  is the threshold

## 7.14 Distance Measurement

Calculations for cost functions are detailed in 7.12.9, distance measurement is used for cost functions/metrics calculations but various exist.

### 7.14.1 Euclidean

### 7.14.2 Manhattan

### 7.14.3 Minkowski

### 7.14.4 Cosine similarity

Mathematical mean to compute the similarity between pairs of vectors (in machine learning refers to feature vector). In the example of a sentence, works by computing one more dimension per type of word and is not impacted by the length of a line but the position between different vectors. More similarity implies **closer to 1**. Basically it is a transformation of the scalar product of two vectors:

$$A.B = \|A\| \|B\| \cos(\theta)$$

And then isolate  $\theta$

$$\frac{A.B}{\|A\| \|B\|} = \cos(\theta)$$

Now it is only needed to isolate  $\theta$  by using  $\arccos$

```
from sklearn.metrics.pairwise import cosine_similarity

# Declaring vectors
data = [[1, 2], [2, 3]]

# Calculate similarity
cosine_similarity(data) # => [0][1] to get element of vector 1 compared to vector 2
                           otherwise calculated twice
# otherwise result is: array([[1.          , 0.99227788], [0.99227788, 1.          ]])
```

## 7.15 Optimization Algorithms

Calculations for cost functions are detailed in 7.12.9

### 7.15.1 GD (Gradient Descent)

Work by (repeats the following until sufficiently optimized):

- ◇ intializing (guess parameters)
- ◇ calculate gradient of cost function
- ◇ update parameters

Step size can be defined in hyperparameters as "learning rate" (equilibrium between too much time to get trained and too fast to adapt). Other issues encountered with GD are plateau and local minimum which might lead GD to stop somewhere where minimum was not reached.

Fortunately MSE is a convex function (a function is convex is taking two random points on the graph, the segment joining both points is over the graph). Indeed, approaching global minimum can be guaranteed. Default is converging to local minimum instead of global minimum, both can be done.

Types of GD:

- Batch:

$$\frac{\partial}{\partial \theta_j} MSE(\theta) = \frac{2}{m} \sum_{i=1}^m (\theta^T \mathbf{x}^i - y^i) x_j^{(i)}$$

As a reminder but  $\theta_j$  is the  $j^{th}$  parameter.

- Stochastic (SGD): Random step size but if iterates a lot will never converge so step size is reduced over time.  
Function defining learning rate based on number of iterations is called the "learning schedule".
- Mini-Batch:

## 7.16 Evaluation Metrics

Details on evaluation metrics at section 7.12.10

### 7.16.1 Confusion matrix

- TP (True Positive): predicted positive, is positive
- FP (False Positive): predicts positive, not positive
- TN (True Negative): predicts negative, is negative
- FN (False Negative): predicts negative, not negative

```
from sklearn.metrics import confusion_matrix
y_pred = model.predict(x_test)
cm = confusion_matrix(y_test, y_pred)
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, cmap='Blues', fmt='d',
            xticklabels=['Predicted Negative', 'Predicted Positive'],
            yticklabels=['Actual Negative', 'Actual Positive'])
plt.xlabel('Predicted label')
plt.ylabel('True label')
plt.title('Confusion Matrix')
plt.show()
```

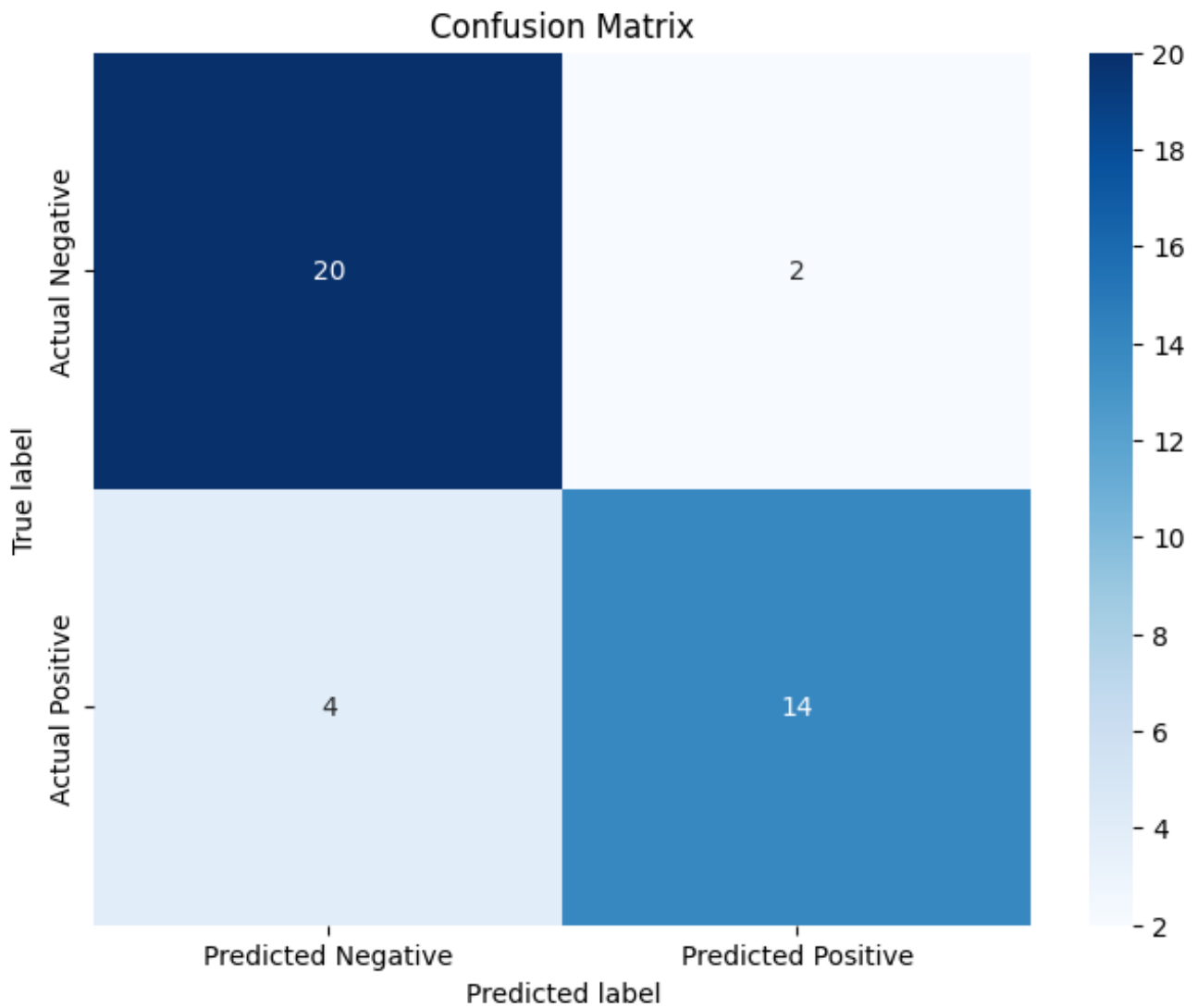


Figure 16: Signal count

Newly supported confusion matrix method:

```
from sklearn.metrics import ConfusionMatrixDisplay as cmd
labels = ['Legitimate', 'Fraudulent']
cmd.from_estimator(lr_model, x_test, y_test, display_labels=labels, cmap='Blues',
                  xticks_rotation='vertical')
```

### 7.16.2 Accuracy

Ratio of correct predictions amongst all predictions made.

$$Accuracy = \frac{TP + TN}{Total}$$

### 7.16.3 Precision (sensitivity)

Positive instances amongst predicted positive instances.

$$Precision = \frac{TP}{ActualResults} = \frac{TP}{TP + FP}$$



#### 7.16.4 Recall (specificity)

$$Recall = \frac{TP}{PredictiveResults} = \frac{TP}{TP + FN}$$

In other words  $TP + FN$  are all the positive cases, predicted or not.

```
from sklearn.metrics import recall_score

score = recall_score(y_test, y_pred, pos_label=0)
```

Where *pos\_label* represents the positive label or the label for which we want to know the recall.

#### 7.16.5 F1

Optimal scoring at 1

$$F1 = 2 * \frac{precision * recall}{precision + recall}$$

#### 7.16.6 Classification report

Provides an overview of precision/recall/f1-score/support

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report

# Load the Iris dataset
data = load_iris()
X = data.data
y = data.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    random_state=42)

# Train a Random Forest Classifier
classifier = RandomForestClassifier(n_estimators=100, random_state=42)
classifier.fit(X_train, y_train)

# Make Predictions
y_pred = classifier.predict(X_test)

# Generate the Classification Report
report = classification_report(y_test, y_pred, target_names=data.target_names)
print(report)
```

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	19
versicolor	1.00	1.00	1.00	13
virginica	1.00	1.00	1.00	13
accuracy			1.00	45
macro avg	1.00	1.00	1.00	45
weighted avg	1.00	1.00	1.00	45

Figure 17: Classification report

### 7.16.7 R-squared

Commonly used **for regression problems**, provides an indication of the "goodness of fit", between 0 and 1 (for not fit or perfect fit)

```
from sklearn.metrics import r2_score
r2_score(y_val_test, y_pred_test)
```

And is calculated with:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

With  $SS_{res} = \sum_{i=1}^n (y_i - \hat{y}_i)^2$ , where  $y_i$  the actual value and  $\hat{y}_i$  is the predicted one, we also have and  $SS_{tot} = \sum_{i=1}^n (y_i - \bar{y})^2$  where  $y_i$  is the actual value and  $\bar{y}$  the mean of actual values.

### 7.16.8 adjusted R-squared

$Adj - R^2$  is calculated by:

$$R_{adj}^2 = 1 - \frac{(1 - R^2)(n - 1)}{n - k - 1}$$

Where  $n$  is the number of observations and  $k$  the number of predictors.

R-squared **close to 1 means a perfect fit**.

### 7.16.9 ROC curve (Receiver Operating Characteristic curve)

Plots the TPR (True Positive Rate) against the FPR (False Positive Rate). Model is **most accurate when curves arches toward upper-left corner**.

### 7.16.10 AUC (Area Under Curve)

Used with 7.16.9, varies between 0 and 1.

### 7.16.11 OOB (Out-of-Bag)

Specifically used for Random Forests and related algorithms. Technique to estimate the performance of a model without the need for an additional validation set. Often used for random forests.

```
score = fitted_model.oob_score_
```

### 7.16.12 mAP (mean Average Precision)

### 7.16.13 K-Means scoring

Inertia:

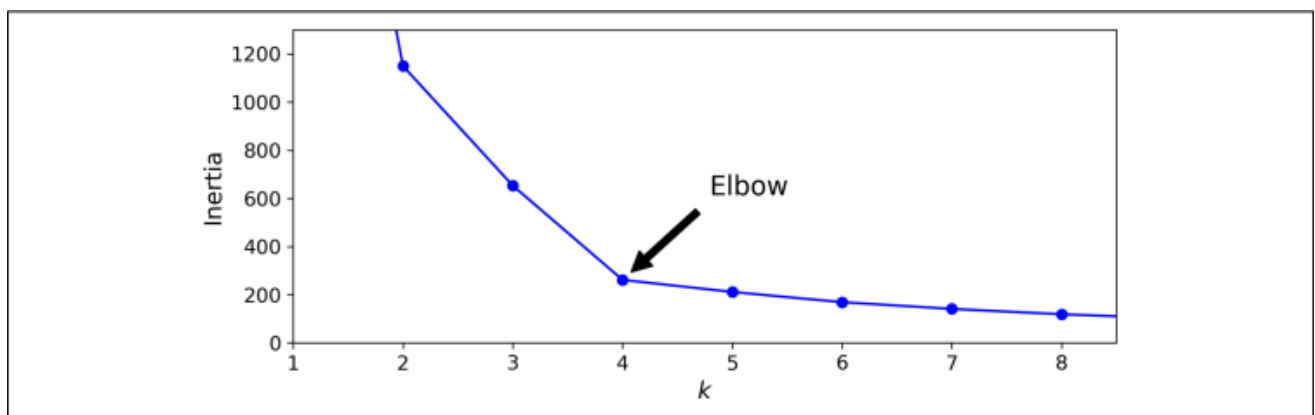


Figure 18: KMeans inertia elbow

Silhouette:

```
from sklearn.metrics import silhouette_score
silhouette_score(X, kmeans.labels_), a is the mean intra-cluster distance
and b is the mean near-cluster distance, varies between [-1:1]
```

## 7.17 Overfitting

May detect patterns where there is none e.g. country's name in data might consider the name with letter 'w' in the name to be richer without real causation. Solving overfitting can be done this in multiple ways through what is called regularization:

- ◇ simplify model (select less features)
- ◇ gather more data
- ◇ reduce noise in training data (remove outliers, fix errors, ...)

### 7.17.1 Regularization

## 7.18 Regularization Algorithms

Regularization itself is detailed in 7.17.1

### 7.18.1 Lasso (L1)

$$CostFunction = RSS + \lambda * \sum_{j=1}^p |\beta_j|$$

Where the larger the  $\lambda$  the more parameters tend to approach 0. A  $\lambda$  value of 0 produces a basic linear regression. Only creates high weights (low weights are turned into 0 thus creating a sparse matrix).

Import: import module

```
from sklearn import linear_model
```

Initiate model:

```
lasso = linear_model.Lasso(alpha=0.1)
```

Fit:

```
lasso.fit(X, Y)
```

Predict:

```
prediction = lasso.predict(X)
```

### 7.18.2 Ridge (L2)

$$CostFunction = RSS + \lambda * \sum_{j=1}^p \beta_j^2$$

Import: import module

```
from sklearn.linear_model import Ridge
```

Initiate model:

```
ridge = linear_model.Ridge(alpha=10000)
```

Fit:

```
ridge.fit(X, Y)
```

Predict:

```
prediction = ridge.predict(X)
```

### 7.18.3 ElasticNet

Combination of Ridge and Lasso regression.

Import:

```
from sklearn.linear_model import ElasticNet
```

Initiate model:

```
elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
```

Fit:

```
elastic_net.fit(X, y)
```

Predict:

```
elastic_net.predict(X)
```

## 7.19 Hyperparameters Tuning

### 7.19.1 Grid search

Try a set of parameters and keep best results, to apply using scikit-learn, refer to 7.35.2.

### 7.19.2 Random search

Try random set of hyperparameters.

### 7.19.3 Bayesian optimization

Train model as few times as possible as it is costly. By making use of a **surrogate function** surrogate/objective functions?

### 7.19.4 Genetic algorithms

Highest performing is kept. Based on Darwin's theory where the fittest survives. We thus need to define a **fitness function**

## 7.20 Distributed Training

### 7.20.1 Map reduce

To parallelize training. spark?

## 7.21 Feature Engineering

Process of using domain knowledge to create new features.

### 7.21.1 Feature selection

Selecting the most useful features to train on among existing features.

### 7.21.2 Feature extraction

Feature extraction: combining features to create a more useful one (e.g. age of a car and mileage)

### 7.21.3 Feature cross

Feature engineering technique based on the idea of creating a synthetic feature with two or more categorical features by capturing the interaction between them.

### 7.21.4 LabelEncoder

```
from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder()
df['Gender'] = encoder.fit_transform(df['Gender'])
```

CustomerID	Gender	Age	Annual Income (k\$)	Spending Score (1-100)	
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40
...	...	...	...	...	...

(a) Before OHE

CustomerID	Gender	Age	Annual Income (k\$)	Spending Score (1-100)	
0	1	1	19	15	39
1	2	1	21	15	81
2	3	0	20	16	6
3	4	0	23	16	77
4	5	0	31	17	40

(b) After OHE

Figure 19: Comparison of images before and after OHE

But can also be applied to a list.

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
labels = ['cat', 'dog', 'bird', 'bird', 'dog', 'cat']
labels = np.array(labels)
encoded_labels = le.fit_transform(labels) # => array([1, 2, 0, 0, 2, 1])
```

### 7.21.5 One-Hot encoding

Represent categorical data in numerical format (e.g. small is 0 and tall is 1). Used over LabelEncoding (7.21.4) because gives equal weight to each value. Doesn't work for very variable data such as id because leads to sparse matrix.

### 7.21.6 Data augmentation

Artificially increase size of the training dataset by generating realistic variants of each training instance. Making in a very regularization technique. By realistic it is meant that augmented data (images/...) should not be distinguished from original data.

### 7.21.7 Dimensionality reduction

The main purpose of DR is to simplify the data for computing without losing much information, to speed up training, this can also remove noise and redundant features, to make the model perform better or even saving space. Therefore it adds a certain level of complexity to the pipeline and turns the dataset into more of a blackbox. Dimensionality Reduction (DR) models are in fact unsupervised models. Main problem with dimensionality reduction is the fact that it makes data less easily interpretable.

**Curse of dimensionality** each instance of training dataset contains sometimes millions of features, thus making it much harder to find a good solution, dropping features or changing them might be a good solution to solve this problem (e.g. sides of picture are always white, dropping these pixels might be a good solution, other solution to calculate average of two neighbor pixels)

**Manifold learning** 2D or higher dimensional figure that can be twisted into a higher dimension-dimensional space. Relies therefore on the manifold hypothesis that states that most real-world high-dimensional datasets lie close to a lower-dimension space (e.g. MNIST dataset most letters are connected, of black color, ...).

### Variance preserved after scalar

```
# Truncated svd to save as much data as possible
from sklearn.decomposition import TruncatedSVD

ncomps = 4
svd = TruncatedSVD(n_components=ncomps)
svd_fit = svd.fit(dataset)
Y_pred = svd.fit_transform(dataset)
ax = pd.Series(svd_fit.explained_variance_ratio_.cumsum()).plot(kind='line', figsize=(10, 3))
ax.set_xlabel("Eigenvalues")
ax.set_ylabel("Percentage Explained")
print('Variance preserved by first 5 components == {:.2%}'.format(svd_fit.explained_variance_ratio_.cumsum()[-1]))
```

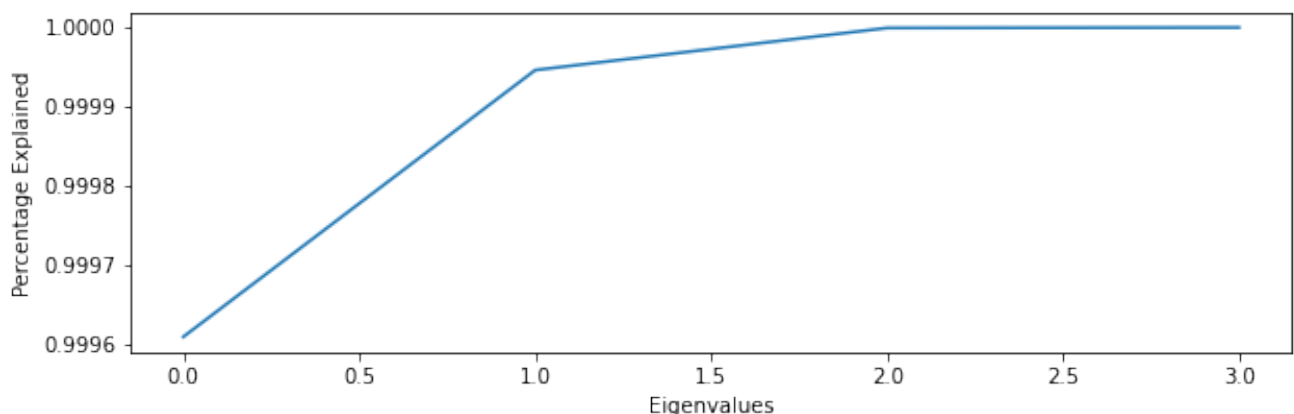


Figure 20: Variance preserved with number of features

### 7.21.8 Data scaling

One of the most important transformations, ML algorithms usually don't perform well when the numerical attributes have very different scales. Most parametric models require parametrization to avoid giving weights to certain features but non-parametric ones do not require such thing.

## 7.22 Dimensionality Reduction Algorithms

Dimensionality reduction itself is detailed in section 7.21.7

### 7.22.1 Principal Component Analysis (PCA)

DR (Dimensionality Reduction) model. Reduce dimensionality of dataset with many variables while retainin as much variance as possible. PCA finds new variables through a linear combination called PC (Principal Components), those are orthogonal (or independant). The end goal is to find the directions that preserve the most variance, the two most common approaches are Eigen Decomposition and SVD (referred in the subsection about

dimensionality reduction) PCA (Principal Component Analysis): Works by projecting data on the hyperplane that lies closest to the data. Decision trees are sensitive to rotation, **PCA can help solving this**

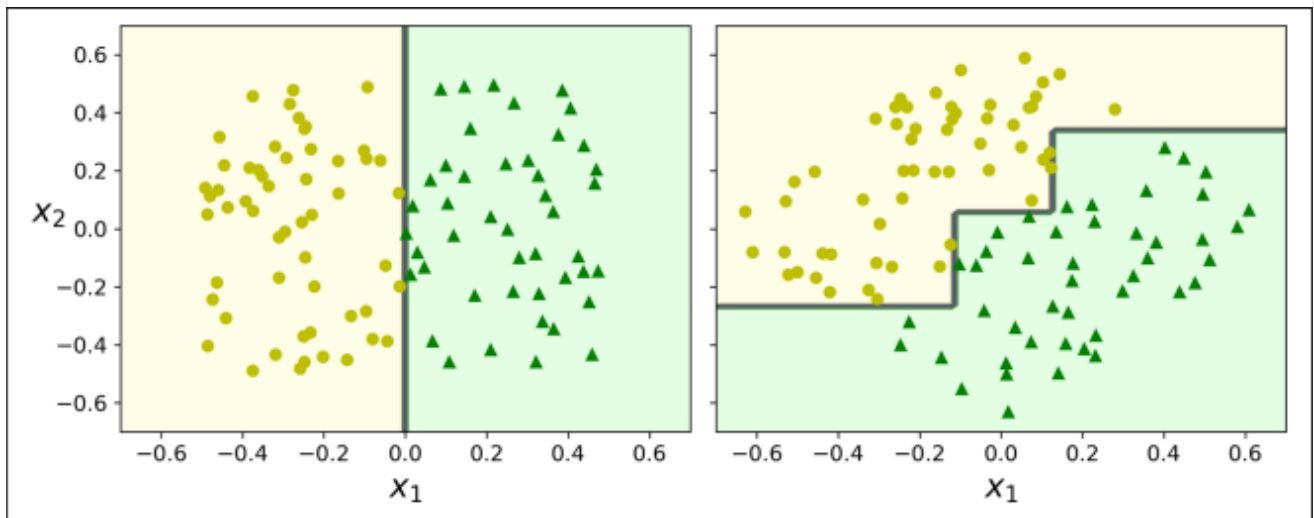


Figure 21: Same dataset rotated 45° shows high sensitivity

Major issue with PCA is not being able to apply to nonlinear data.

```
from sklearn.decomposition import PCA
pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)

pca.explained_variance_ratio_
```

**hyperparameters explained: svd\_solver="randomized", n\_components = output dimension**

### 7.22.2 IPCA (Incremental PCA)

```
from sklearn.decomposition import IncrementalPCA
n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_train, n_batches):
    inc_pca.partial_fit(X_batch)
X_reduced = inc_pca.transform(X_train)
```

### 7.22.3 t-distributed Stochastic Neighbor Embedding (t-SNE)

#### 7.22.4 kernel PCA

Solves major issue with PCA by handling nonlinearity.

```
from sklearn.decomposition import KernelPCA
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
```

**hyperparameters explained: svd, n\_components, kernel, gamma**

#### 7.22.5 sparse PCA

**nonlinear data?**



### 7.22.6 ED (Eigen Decomposition)

Steps are as follow: covariance matrix is created for the features then eigenvectors of the covariance matrix are created (directions of maximum variance), eigenvalues are finally created, those define the magnitude of the principal components.

### 7.22.7 SVD (Singular Value Decomposition)

Goal is to get a lower rank matrix.

$$A = U\Sigma V^T$$

Where  $A$  is a  $m \times n$  matrix,  $U$  is a  $m \times n$  orthogonal matrix and  $\Sigma$  is a  $m \times n$  diagonal matrix. And  $V$  is also an orthogonal matrix.

### 7.22.8 TSVD (Truncated SVD)

SVD but only for large singular values.

### 7.22.9 LLE (Locally Linear Embedding)

```
from sklearn.manifold import LocallyLinearEmbedding
lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10)
X_reduced = lle.fit_transform(X)
```

### 7.22.10 K-Means

```
from sklearn.pipeline import Pipeline
pipeline = Pipeline([
    ("kmeans", KMeans(n_clusters=50)),
    ("log_reg", LogisticRegression()),
])
pipeline.fit(X_train, y_train)
from sklearn.model_selection import GridSearchCV
param_grid = dict(kmeans__n_clusters=range(2, 100))
grid_clf = GridSearchCV(pipeline, param_grid, cv=3, verbose=2)
grid_clf.fit(X_train, y_train)
>>> grid_clf.best_params_
{'kmeans__n_clusters': 99}
>>> grid_clf.score(X_test, y_test)
0.9822222222222222
```

verbose?

### 7.22.11 t-SNE (t-Distributed Stochastic Neighbor Embedding)

Reduces dimension by modeling the probability of neighbors (set of points around a given point) around each point.

```
from sklearn.manifold import TSNE
X_valid_compressed = stacked_encoder.predict(X_valid)
tsne = TSNE()
X_valid_2D = tsne.fit_transform(X_valid_compressed)
```

Scatter classification:

```
plt.scatter(X_valid_2D[:, 0], X_valid_2D[:, 1], c=y_valid, s=10, cmap="tab10")
```

### 7.22.12 LLE (Locally Linear Embedding)

Visualization and dimensionality reduction

### 7.22.13 Random projection

### 7.22.14 Manifold learning

### 7.22.15 Isomap

### 7.22.16 Other unsupervised algorithms

The following algorithms will be mentioned as they are unsupervised classification algorithms but won't be detailed here but rather in the section dedicated to unsupervised algorithms.

Hierarchical clustering, DBSCAN

## 7.23 Data Scaling Algorithms

Further details on data transformation in section 7.21.8

### 7.23.1 Standard scaling

Consists of turning dataset into dataset with mean of 0 and variance of 1.

$$standardization = \frac{x - mean}{\sigma}$$

Where  $\sigma$  represents the standard deviation, resulting distribution has unit variance.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler().fit(x)
StandardisedX = pd.DataFrame(scaler.fit_transform(X))
```

### 7.23.2 MinMaxScaler

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))
rescaledX = pd.DataFrame(scaler.fit_transform(X))
```

Between 0 and 1

$$s = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Between -1 and 1

$$s = \frac{2x - x_{max} - x_{min}}{x_{max} - x_{min}}$$

### 7.23.3 Clipping

To reduce outlier effect, consider MinMaxScaling and use -1 or 1 for outliers.

#### 7.23.4 Z-score normalization

$$s = \frac{x - \bar{x}}{\sigma}$$

Where  $\sigma$  is the standard deviation and  $\bar{x}$  is the mean. Lies between -1 and 1 the majority of the time but not always.

#### 7.23.5 LRN (Local Response Normalization)

Based on observations made on biological neurons, the most strongly activated neuron inhibit other neurons located at the same position in neighboring feature maps.

### 7.24 Feature Selection Algorithms

Reference on feature selection at 7.21.1

#### 7.24.1 Chi-squared $\chi^2$

Used to find the best features.

```
from sklearn.feature_selection import chi2
selector = SelectKBest(score_func=chi2, k=2)
X_new = selector.fit_transform(X, y)
```

Where  $k$  is the number of features to keep.

### 7.25 Embedding

#### 7.25.1 Word embedding

Turn words into numbers and get similar words to have similar numbers. Same word with different meanings is assigned different numbers (e.g. "it's great we could see each other" or "my phone is broken, great"). Neurons weights are vector components.

#### 7.25.2 Word2Vec

Word embedding (7.25.1) by making use of neural networks. Introduced by Tomas Mikolov et al. at Google in 2013.

- ◇ Continuous Bag of Words (CBOW): Predicts the current word given its context (surrounding words). It aims to maximize the probability of the target word given the context words. (more context on BOW in section 7.33.7)
- ◇ Skip-gram: Predicts surrounding words given the current word. It aims to maximize the probability of context words given the target word.

Negative sampling: ignore unconsidered weights.

### 7.26 Cascade

Machine learning task where output of one model is input of the other.

### 7.27 Rebalancing

Used when only one class is represented in a dataset (consider fraud vs non fraud with 1% fraud and 99% non-fraud). Solutions are:

#### 7.27.1 Downsampling

Throw away instances of majority class. Models might work with 75/25 ratio but some require a net 50/50 ratio.

#### 7.27.2 Upsampling

Generating synthetic examples. **SMOTE?**

## 7.28 Model Explainability

### 7.28.1 Attribution values

Tells how much features influence predictions.

## 7.29 Shap

Created for model explainability.

learn more about it <https://github.com/shap/shap?tab=readme-ov-file>

### 7.29.1 Feature importance

```
import xgboost
import shap

# train an XGBoost model
X, y = shap.datasets.california()
model = xgboost.XGBRegressor().fit(X, y)

# explain the model's predictions using SHAP
# (same syntax works for LightGBM, CatBoost, scikit-learn, transformers, Spark, etc
.)
explainer = shap.Explainer(model)
shap_values = explainer(X)

# visualize the first prediction's explanation
shap.plots.waterfall(shap_values[0])
```

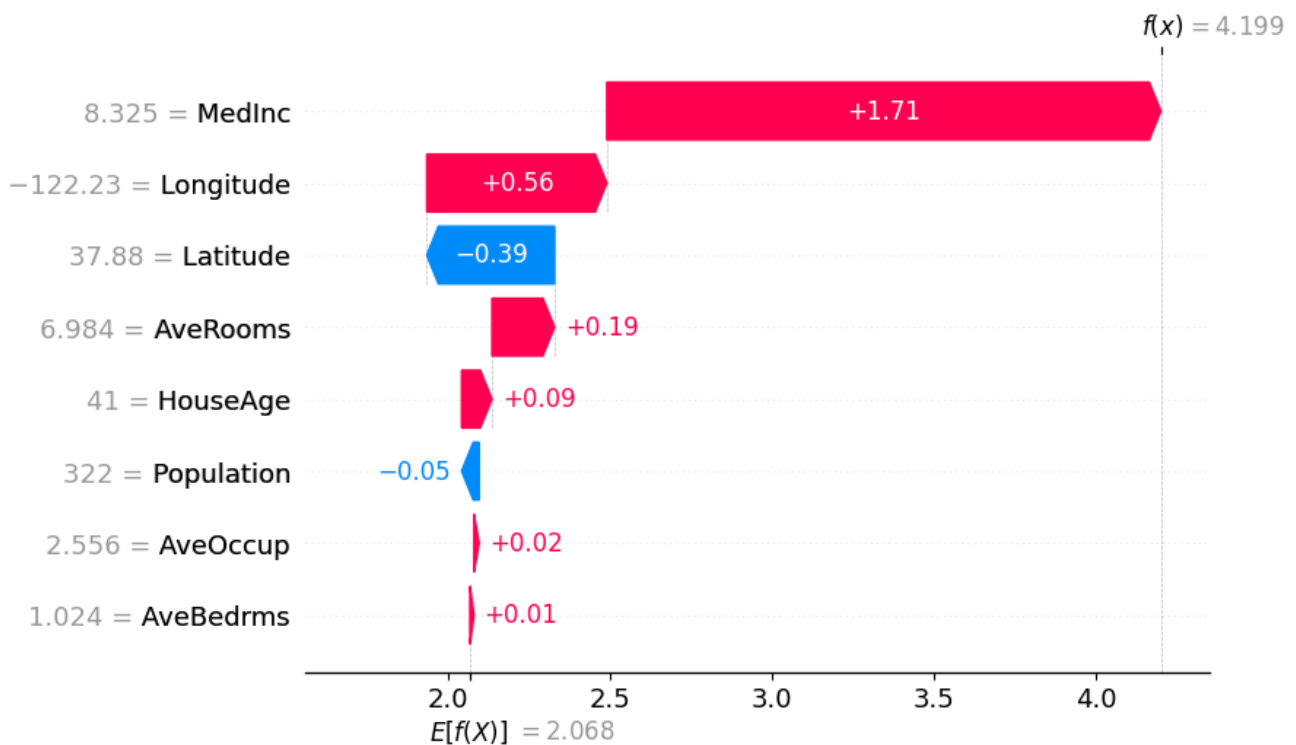


Figure 22: Feature importance in house price prediction

## 7.30 TPOT

Used for genetic algorithm

```

import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_digits
from tpot import TPOTClassifier

# Load dataset
digits = load_digits()
X, y = digits.data, digits.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=42)

# Initialize TPOT
tpot = TPOTClassifier(generations=5, population_size=20, verbosity=2, random_state
                      =42)

# Fit the TPOT classifier on the training data
tpot.fit(X_train, y_train)

# Evaluate the model on the test data
print(tpot.score(X_test, y_test))

# Export the optimized pipeline
tpot.export('tpot_optimized_pipeline.py')

```

### 7.31 Resilient Serving

Little human intervention required.

### 7.32 Distributed Learning

#### 7.32.1 Synchronous training

Workers train on different slices of input data in parallel and values are aggregated.

#### 7.32.2 Asynchronous training

Workers train on different slices of input data independently and weights are updated asynchronously.

### 7.33 Natural Language Processing

#### 7.33.1 Vectorization

Converting text into numbers.

#### 7.33.2 CountVectorizer

Dictionary from the corpus of words in the training text and generates a matrix of word counts.

#### 7.33.3 HashingVectorizer

Uses word hashes to be more memory efficient

#### 7.33.4 TfidfVectorizer

Dictionary containing frequency of words (term frequency-inverse document frequency (TFIDF)).

#### 7.33.5 Stemming

(e.g. equals = equal when text has been stemmed)

### 7.33.6 Lemmatizing

### 7.33.7 Bag of words

Based on normalized text (e.g. awesome = AweSoME = Awesome).

## 7.34 Recommender Systems

### 7.34.1 Popularity based systems

Presents what is popular right now.

### 7.34.2 Collaborative systems

Recommendations based on what others have selected.

### 7.34.3 Content-based systems

If you liked this content you might like this other content.

## 7.35 Scikit-learn

Design principles are well built to use their API. Those are the following:

Consistency:

- ◇ Estimators: estimate parameters based on dataset (e.g. imputer), uses fit() method
- ◇ Transformers: returns a transformed dataset, by relying on transformed dataset
- ◇ Predictors: predicts using the predict() method and has a score() method to evaluate it's performance, in other words it's an algorithm

Inspection: estimators's hyperparameters can be accessed through public instances

Nonproliferation of classes: datasets are numpy classes or scipy matrices instead of homemade classes

Composition: reuse building blocks as much as possible, by creating pipeline

Sensible defaults: default values for most parameters, making it quick to create a baseline working system

get model fit scores and so on (calculate own error function, ...)

### 7.35.1 .train\_test\_split()

Train and test set to verify that the model **learned** and **generalizes** the data well.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=42)
```

### 7.35.2 GridSearchCV (GridSearch Cross Validation)

```
model = <model_name>
param_grid = {'fit_intercept': [True, False], 'n_estimators': [100, 150, 200]}
grid_search = GridSearchCV(model, param_grid, cv=3, scoring='r2')
grid_search.fit(X_train, y_train)
print(grid_search.best_params_)
```

Where *scoring* is the metric to be optimized. Not a good idea on big datasets as it might lead to a **combinatorial explosion**.

### 7.35.3 KFold

```
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.svm import SVC

# Load a sample dataset (for demonstration purposes)
iris = load_iris()
X, y = iris.data, iris.target

# Initialize the KFold cross-validation object
kf = KFold(n_splits=5, shuffle=True, random_state=42) # Define number of splits (k)
, shuffle data, and random state

# Loop over each fold and train/test the model
for fold, (train_index, test_index) in enumerate(kf.split(X)):
    print(f'Fold {fold+1}')
    X_train, X_test = X[train_index], X[test_index] # Split data into train and
    test sets
    y_train, y_test = y[train_index], y[test_index]

    # Initialize and train your model (for demonstration purposes, using Support
    Vector Classifier)
    model = SVC(kernel='linear')
    model.fit(X_train, y_train)

    # Evaluate the model on the test set
    accuracy = model.score(X_test, y_test)
    print(f'Accuracy: {accuracy:.4f}\n')
```

### 7.35.4 .score()

### 7.35.5 cross\_val\_score()

Uses  $R^2$  score, detailed in 7.16.7.

```
from sklearn.model_selection import cross_val_score
cvs = cross_val_score(model, x, y, cv=5) # e.g. array([0.6453678 , 0.347992 ,
0.49169531, 0.53649706, 0.60955012])
cvs.mean() # what really matters
```

### 7.35.6 .summary()

```
model.summary()
```

Layer (type)	Output Shape	Param #
flatten_3 (Flatten)	(None, 784)	0
dense_10 (Dense)	(None, 300)	235500
dense_11 (Dense)	(None, 100)	30100
dense_12 (Dense)	(None, 10)	1010
Total params: 266610 (1.02 MB)		
Trainable params: 266610 (1.02 MB)		
Non-trainable params: 0 (0.00 Byte)		

Figure 23: Model summary after compiling

#### 7.35.7 accuracy\_score(actual, predicted)

```
accuracy_train = accuracy_score(Y_train, knn.predict(X_train))
accuracy_test = accuracy_score(Y_test, knn.predict(X_test))
```

#### 7.35.8 make\_classification()

Create a classification dataset.

```
from sklearn.datasets import make_classification

# Generate pseudo-random dataset
X, y = make_classification(n_samples=25, n_features=2, n_informative=2,
                          n_redundant=0, n_classes=2, random_state=42)

print(X)
print("=====")
print(y)
#####
[[ 1.51892967  2.06893996]
 [-0.63261556 -1.36951734]
 [-1.38312222  1.68462133]
 [-1.12450988 -0.75541669]
 [-1.86225266  1.80856609]
 ...
 [-2.76108267  0.87454572]
 [ 0.02320915  0.76512646]
 [ 0.45424212 -2.71686178]
 [-1.40610823  1.39801155]
 [-2.78712335  0.70177458]
 [ 0.0213132  -0.02560141]]
=====
```



```
[1 0 1 0 1 0 0 0 1 1 1 1 0 0 0 1 0 1 1 0 0 0 1 0 1]
```

### 7.35.9 Feature importance bar chart

```
model = RandomForestRegressor(n_estimators= 200,n_jobs=-1)
model.fit(X_train,Y_train)
#use inbuilt class feature_importances of tree based classifiers
feat_importances = pd.Series(model.feature_importances_, index=X.columns)
feat_importances.nlargest(10).plot(kind='barh', color='r')
```

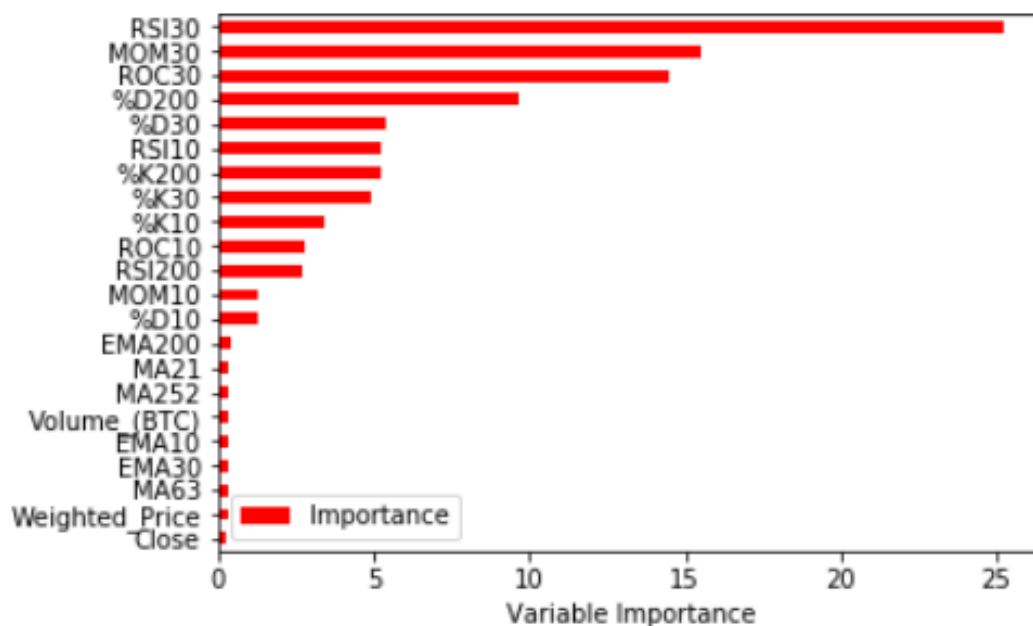


Figure 24: Different graphs stacked on matplotlib

## 7.36 TensorFlow

### 7.37 Keras

High-level TensorFlow api for building neural networks.

#### 7.37.1 Convolutional layer

```
Conv2D(filters=16, kernel_size=3, activation='relu', input_shape=(28,28,3))
```

Input images divided into 3x3 chunks before passing through a max pooling layer.

## 7.38 Generate Random Data

### 7.38.1 Blobs

```

from sklearn.datasets import make_blobs
points, cluster_indexes = make_blobs(n_samples=300, centers=4, cluster_std=0.8,
    random_state=0)
x = points[:, 0] # x-coordinates
y = points[:, 1] # y-coordinates
plt.scatter(x, y, s=50, alpha=0.7)

```

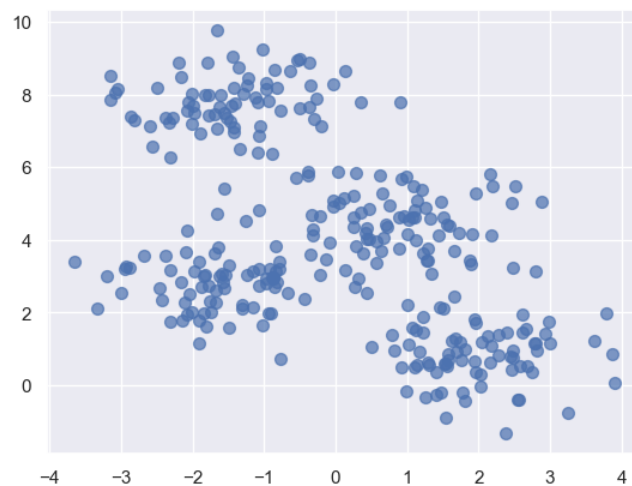


Figure 25: Blobs scatter plot

### 7.38.2 Moons

```

from sklearn.datasets import make_moons
x, y = make_moons(n_samples=200, noise=0.3, random_state=42)

```

### 7.38.3 Regression

```

from sklearn.datasets import make_regression
x, y = make_regression(n_samples=300, n_features=1, noise=20, random_state=0)

```

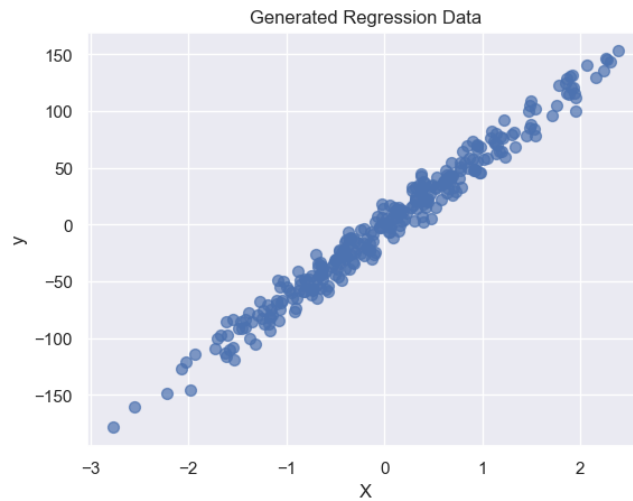


Figure 26: Regression data from sklearn

## 7.39 Linear regression

### 7.39.1 OLS (Ordinary Least Squares)

Based on assumption that target variable is a linear combination of feature values, the goal is the find a function as such:

$$f : \mathbb{R} \rightarrow \mathbb{R}, \quad y = \alpha + \beta x$$

Where this linear function is found by minimizing error which is calculated with distance to the expected value.

$$\min_{\alpha, \beta} \frac{1}{N} \sum_{n=1}^N (y_n - f(x_n))^2$$

In this expression  $y_n$  is the expected value and  $f(x_n)$  is the value of the function, to minimize the global value of the error we use the following calculation:

$$\beta^* = \frac{Cov(x, y)}{Var(x)}$$

$$\alpha^* = \bar{y} - \beta \bar{x}$$

Normal equation permits to immediatly calculate the searched terms.

$$\hat{\theta} = (X^T X)^{-1}$$

$\hat{\theta}$  is the value that minimizes cost function and  $\mathbf{y}$  vector of target values.

Import: import module

```
import sklearn.linear_model
```

Fit: X are features y is expected value

```
model = sklearn.linear_model.LinearRegression()
model.fit(X, y)
```

Predict: input X values outputs y

```
model.predict(X_new)
```

**RSS (Sum of Squared Residuals)**

$$RSS = \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

## 7.40 KNN (K-Nearest Neighbors)

Number of neighbors too low overfits and too high underfits the data.

### 7.40.1 Regressor

```
# Modules import
import sklearn.neighbors
from sklearn.datasets import make_regression
import matplotlib.pyplot as plt

# Generate regression data
x, y = make_regression(n_samples=200, n_features=1, noise=20, random_state=0)

# Display data
model = sklearn.neighbors.KNeighborsRegressor(n_neighbors=3)
model.fit(x, y)
predictions = model.predict(x)
plt.scatter(x, predictions, alpha=0.7, c="blue")
plt.scatter(x, y, alpha=0.5, c="red")
```

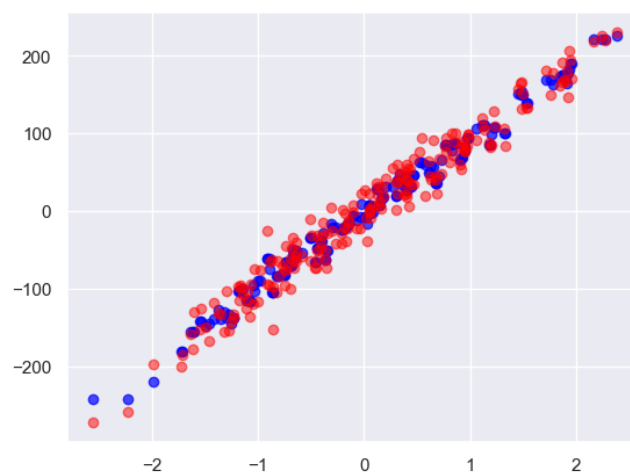


Figure 27: KNN regression fitting

## 7.40.2 Classifier

```
# Modules import
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=10) # n_neighbors default to 5
model.fit(x_train, y_train)
model.score(x_test, y_test) # => accuracy: correct predictions/total predictions
model.predict([[15.2, 3.5, 1.2, 6.51]])
```

## 7.41 K-means

### 7.41.1 Description

Unsupervised learning algorithm, requires to provide number of clusters desired in predictions.

### 7.41.2 Fit clusters

```
from sklearn.cluster import KMeans
model = KMeans(n_clusters=4, random_state=0)
model.fit(points)
predicted_cluster_indexes = model.predict(points)
plt.scatter(x, y, c=predicted_cluster_indexes, s=50, alpha=0.7, cmap='viridis')
```

### 7.41.3 Cluster centers

```
centers = model.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='red', s=100)
```

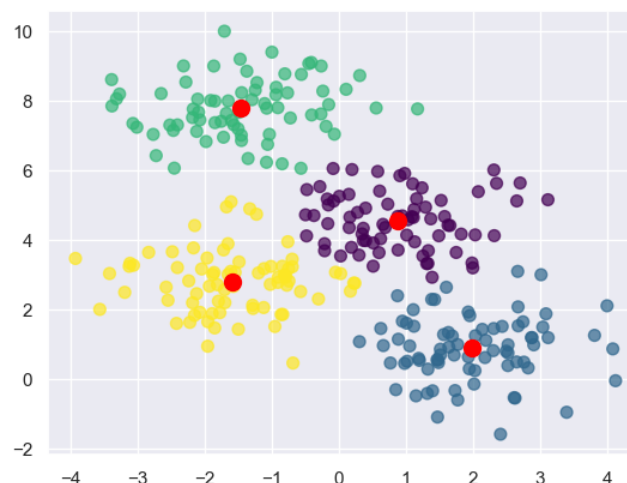


Figure 28: Clusters and respective centroids

### 7.41.4 Elbow method for number of clusters

```

inertias = []
for i in range(1, 10):
    kmeans = KMeans(n_clusters=i, random_state=0)
    kmeans.fit(points)
    inertias.append(kmeans.inertia_)
plt.plot(range(1, 10), inertias)
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')

```

### 7.41.5 Inertia calculation

Also known as the within-cluster sum of squares, measures how close points are to their centroid.

$$\sum_{i=1}^k \sum_{x \in C_i} (\|x - \mu_i\|)^2$$

For each cluster, take every "point" inside and calculate the distance to the centroid of cluster  $C_i$ .

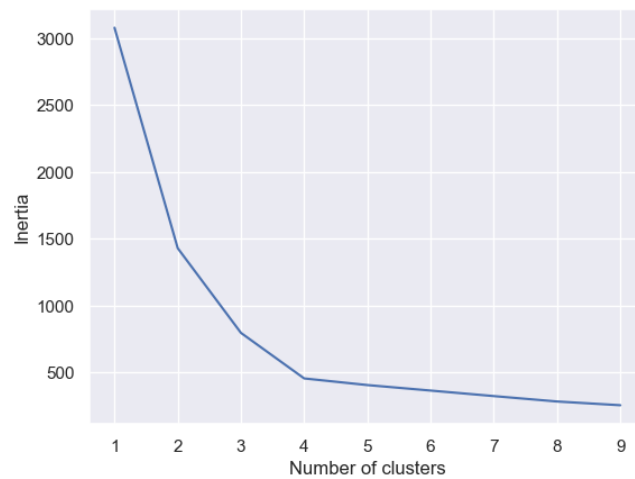


Figure 29: Elbow method describing inertia with number of clusters

## 7.42 RNC (Radius Neighbors Classifier)

### 7.42.1 Model usecase

```

from sklearn.neighbors import RadiusNeighborsClassifier
model = RadiusNeighborsClassifier(radius=1.0)
model.fit(x_train, y_train)
model.score(x_test, y_test)

```

### 7.42.2 Predict class probability

```

model.predict_proba([[6.3, 2.5, 5. , 1.9]]) # => array([[0. , 0.31428571,
0.68571429]])

```

## 7.43 Agglomerative Clustering

### 7.43.1 Silhouette score

Optimal number of clusters maximizes the silhouette score.

```
cluster_range = range(2, 10)
silhouette_scores = []

for i in cluster_range:
    model = AgglomerativeClustering(n_clusters=i)
    cluster_labels = model.fit_predict(points) # Fit and predict in one step
    silhouette_avg = silhouette_score(points, cluster_labels)
    silhouette_scores.append(silhouette_avg)

plt.plot(range(2, 10), silhouette_scores)
```

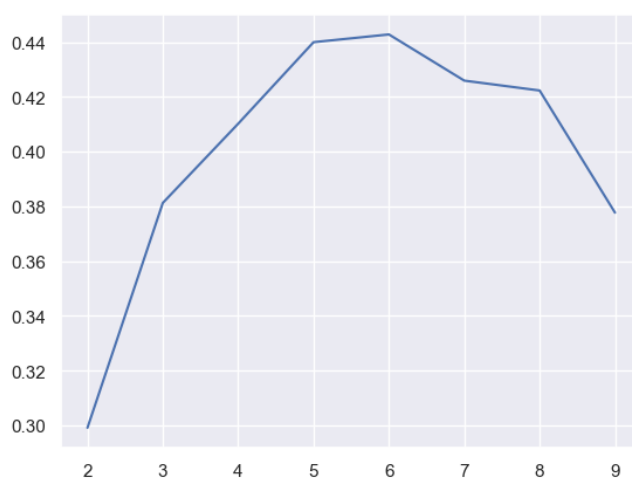


Figure 30: Silhouette score

## 7.44 DBSCAN

### 7.44.1 Description

Unsupervised learning model for classification that doesn't require number of clusters.

### 7.44.2 Fit

```
from sklearn.cluster import DBSCAN

model = DBSCAN(eps=0.01, min_samples=50)
predicted_cluster_indexes = model.fit_predict(points)
plt.scatter(x, y, c=predicted_cluster_indexes, s=50, alpha=0.7, cmap='viridis')
```

## 7.45 Decision trees

Useful both for classification and regression tasks. Root node (depth 0) set binary criterion to classify (e.g. sepal length over 2.45cm for a flower) and proceeds to binary classification until there is no more child nodes to make predictions about. Unlike linear regression and other models, no assumptions are made by the human side but rather automatically by the model adapting itself to the data (nonparametric model, despite having hyperparameters), thus producing either a linear or nonlinear model which might be a cause of overfitting, this is partly due to the fact that the model has no pre-determined number of parameters so its degree of freedom is

unlimited (unlike linear regression, therefore a good balance must be found, setting the depth is an example of regularization that can be easily implemented). Very little data preparation is required and feature scaling is not required at all. Optimal depth is calculated with number of leaves (suppose  $m$  leaves), thus the optimal depth is calculated by  $\log_2(m)^2$  rounded up. Computational complexity is  $O(nxm \log(m))$ . To work the model counts number of instances of similar feature have the same attributes (e.g. Iris Virginica has a sepal length of 2.45cm) then counts number of other instances with same feature, no outlier means gini = 0 (e.g.  $1 - (0/54)^2 - (43/54)^2 \dots$  is used to establish gini impurity). Probability is based on number of instances with current feature. When doing a regression with a decision tree output won't be continuous because leaf node (last one) will be the regression value, in a similar classification problem, that node will be the class.

#### 7.45.1 DTR (Decision Tree Regressor)

#### 7.45.2 DTC (Decision Tree Classifier)

#### 7.45.3 CART (Classification and Regression Trees)

Decision trees both for classification as well as for regression are based on this algorithm to make predictions. Scikit-learn uses binary trees because it makes use of the CART algorithm, thus nonleaf nodes always have two children. The CART algorithm minimizes the following cost function:

$$J = \frac{m_{left}}{m} G_{left} + \frac{m_{right}}{m} G_{right}$$

Where  $J$  is the cost function,  $G$  is the gini measure of impurity (discussed in: 7.45.4) and  $m_{left/right}$  is the number of instances respectively of the *left* and *right* subsets,  $m$  indeed is the total number of instances.

#### 7.45.4 Gini measure of impurity

$$G = 1 - \sum_{i=1}^m p_i^2$$

Where  $p_i$  is the probability/proportion of a given class, consider the following example: class A, B and C with respectively 50%, 30% and 20% as frequency. Thus gini can be calculated by:  $1 - 0.5^2 - 0.3^2 - 0.2^2 = 0.62$ .

#### ID3 comparison to CART (because ID3 can have more than two children)

**Chi-squared test** Is used to estimate the probability that improvement in predictions made by tree leafs are based on statistical chances not real improvements, thus we can remove them. Probability here is qualified as p-value, threshold is set (usually around 5%) and if values goes over then node is considered unnecessary and children are deleted. Pruning continues until all unnecessary nodes have been pruned.

#### Gini Impurity

$$G_i = 1 - \sum_{k=1}^n p_{i,k}^2$$

Where  $p_{i,k}$  is the number of instances of class k among training instances of node i. Gini is the default binary classifier but Entropy can also be used, both often lead to similar results but Gini is slightly faster, therefore entropy produces more balanced trees.

**Entropy** Same idea as in thermodynamics, where order is almost perfect when entropy approaches 0. Entropy function is evaluated as such:

$$H_i = \sum_{k=1}^n p_{i,k} \log_2(p_{i,k})$$

Import: import module



```
from sklearn.tree import DecisionTreeClassifier
```

Initiate model:

```
tree = DecisionTreeClassifier(max_depth=1)
```

Fit:

```
tree.fit(X_train, y_train)
```

Predict:

```
tree.predict([X])
```

Predict proba:

```
tree.predict_proba(X)
```

```
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier

# Generate moon dataset
X, y = make_moons(n_samples=200, noise=0.4)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42, test_size=0.2)

# Decision tree
tree = DecisionTreeClassifier(max_depth=1)
tree.fit(X_train, y_train)

# Plotting functions
def plot_dataset(X, y, axes):
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs") # Plot different classes
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^")
    plt.axis(axes)
    plt.grid(True, which='both')
    plt.xlabel(r"$x_1$", fontsize=20)
    plt.ylabel(r"$x_2$", fontsize=20, rotation=0)

def plot_predictions(clf, axes):
    x0s = np.linspace(axes[0], axes[1], 100)
    x1s = np.linspace(axes[2], axes[3], 100)
    x0, x1 = np.meshgrid(x0s, x1s)
    X = np.c_[x0.ravel(), x1.ravel()]
    y_hat = clf.predict(X)
    y_hat = y_hat.reshape(x0s.shape)
    plt.plot(x0s, x1s, y_hat, "b.")
```

```

# Obtain probabilities instead of predictions
y_proba = clf.predict_proba(X)[: , 1].reshape(x0.shape)
# Plot probability estimates as a color gradient
plt.contourf(x0, x1, y_proba, cmap=plt.cm.brg, alpha=0.2)

# Plot predictions
plot_predictions(tree, [-1.5, 2.5, -1, 1.5])

# Plot dataset
plot_dataset(X_train, y_train, [-1.5, 2.5, -1, 1.5])

# Show the plot
plt.show()

```

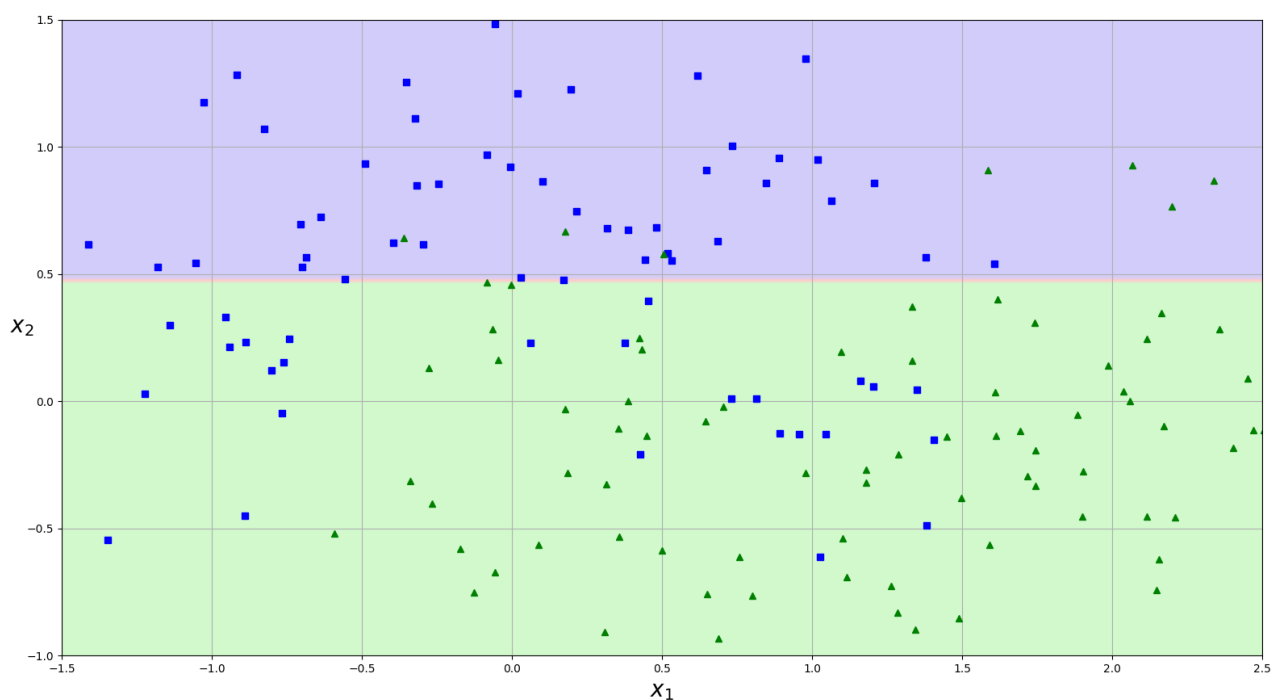
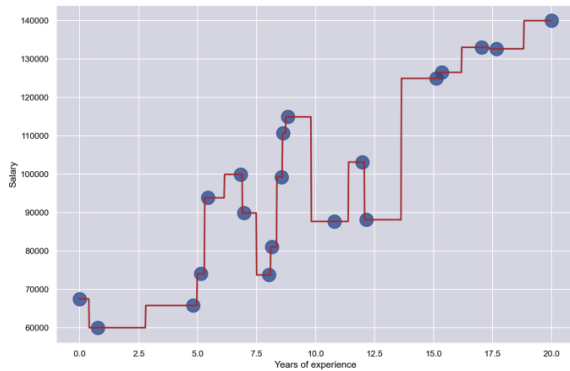


Figure 31

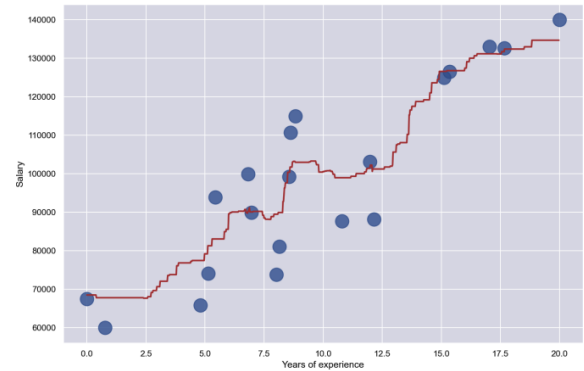
## 7.46 Random forest

### 7.46.1 Description

Decision trees are too much prone to overfitting thus random forests solves this problem. Number of predictors (trees) for each split is usually the square root of the total number of predictors (features). When the model makes a prediction it is the average of all the trees (each one trained on random rows of the dataframe fed as input). Because the trees are constructed independently, they can each be trained (thus random forests supports parallelized training). Prevents quite well overfitting compared to a single decision tree.



(a) Decision tree



(b) Random forest

Figure 32: Comparison of decision tree versus random forest

### 7.46.2 RandomForestRegressor

Training many Decision Trees on random features subsets and averaging out predictions.

### 7.46.3 RandomForestClassifier

```
from sklearn.ensemble import RandomForestClassifier
forest = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1)
forest.fit(X_train, y_train)
forest.predict(X)
forest.predict_proba(X)
```

### Hyperparameters

*n\_estimators*: the number of trees 100, 200, ...,

*criterion*: gini or entropy

*min\_impurity\_decrease*: 0.0, 0.1, 0.2, 0.3

*max\_leaf\_nodes*: [16, 32, 48, 64]

*max\_depth* = maximum number of children/branches for each tree

*max\_samples* = maximum number of rows of the dataset that can be seen by a single tree

*n\_jobs* = number of cpu cores for parallel processing, set to -1 to maximize

### 7.47 Extra trees (Extremely Randomized Trees)

Instances trained on a random sample of the training data,

```
clf = ExtraTreesClassifier(n_estimators=10, max_depth=2, random_state=0)
clf.fit(X, y)
```

Classification:

Import:

```
from sklearn.ensemble import ExtraTreesClassifier
```

Build model:

```
model = ExtraTreesClassifier()
```

Fit:

```
model.fit(X, Y)
```

Regression

Import:

```
from sklearn.ensemble import ExtraTreesRegressor
```

Build model:

```
model = ExtraTreesRegressor()
```

Fit:

```
model.fit(X, Y)
```

## 7.48 GBDT (Gradient Boosting Decision Trees)

### 7.48.1 Gradient Boosting Machines (GBM)

### 7.48.2 AdaBoost (Adaptive Boosting)

### 7.48.3 SGB (Stochastic Gradient Boosting)

## 7.49 GBR (Gradient Boosting Regressor)

```
from sklearn.ensemble import GradientBoostingRegressor
gbr = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0)
cross_val_score(gbr, x, y, cv=5).mean
gbr.fit(X, y)
```

Hyperparameters `max_depth=2, n_estimators=3, learning_rate=1.0`

## 7.50 SVM (Support Vector Machines)

### 7.50.1 Description

**Mainly used for classification** whether linear or non-linear, regression and outlier detection. Well suited for mid-size or lower datasets. Purpose of SVM is to fit widest possible "street" between classes (thus it is called large margin classification for that reason). Adding more training instances "off the street" doesn't affect the decision boundary but the model is sensitive to feature scaling, indeed is known to be parametric. Purpose of SVM is to find compromise between largest possible street and perfectly separating both classes. Support vectors are instances located on the "street", as long as new instances are not on the support vector they won't affect the model.

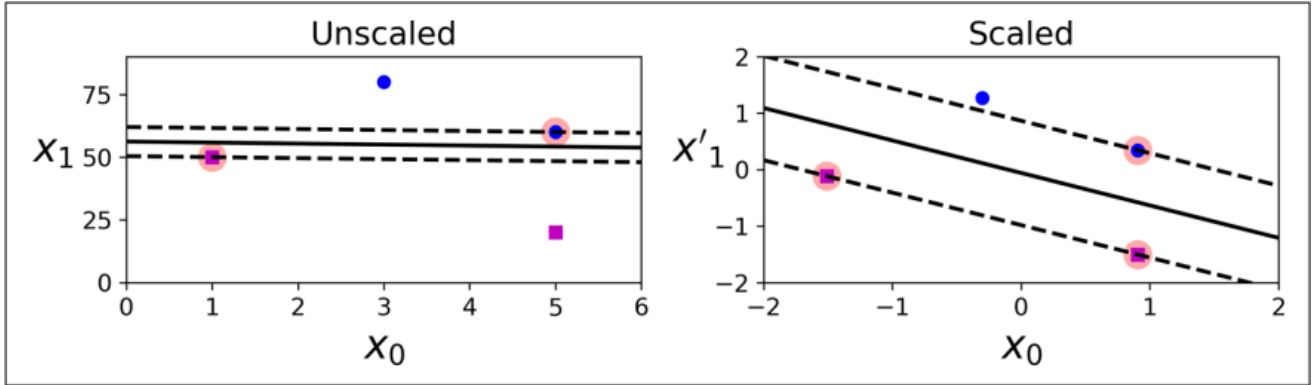


Figure 33: Sensitivity of SVM to feature scaling

To understand this sensitivity, compare vertical scale to horizontal scale (y axis vs x axis).

The output of SVM models are not probabilities but rather what class the model supposes it is. Another fact about sensitivity of this model is that small instances tend to be neglected if not scaled. However distance between decision boundary and instance can be used as confidence score.

To represent model's hyperparameters  $\theta$  is commonly used, the bias term is represented using  $\theta_0$  (intercept term, value added to weighted sum of input features as a constant term allow to make predictions when all input features are zeros) and input feature weights  $\theta_1$  and  $\theta_0$ .

$$y^2 = \dots$$

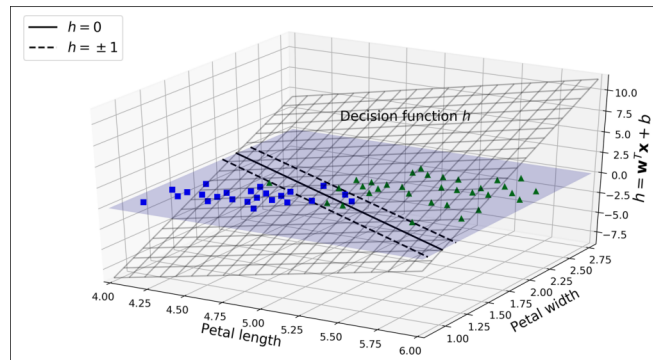


Figure 34: SVM decision function

If over the separation function 1 otherwise 0. Dashed line represents points where decision function is equal to 1 or -1 (those are parallel at equal distance to decision boundary). The purpose of training the model is to find optimal values of  $w$  and  $b$  to make the margin -1/1 as wide as possible while avoiding margin violations (hard margin) or limiting them (soft margin).

◇ Linear SVM Classification

◇ Nonlinear SVM Classification To handle nonlinear data separation (because some datasets are not linearly separable) the creation of new features can be very useful, like here where a new feature is created  $x_2$  comes from  $(x_1)^2$ .

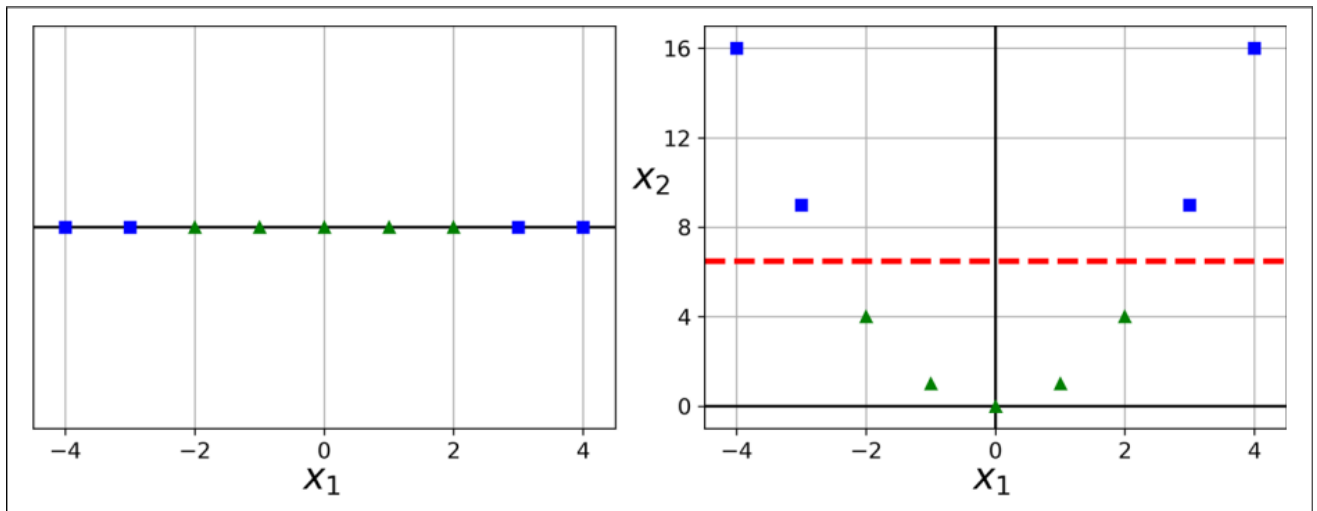


Figure 35: Linear data separation creating new features

◇ SVM Regression

◇ online SVM

SVM hyperparameters:

◇  $c$ : margin violation to generalize better, to avoid overfitting reduce  $C$  (regularization indeed)

◇  $\gamma$ : **gamma? rbf?**

Reducing  $c$  and  $\gamma$  can help reducing overfitting.

Import: import module

```
from sklearn.svm import SVC
```

Initiate model:

```
svc = SVC(kernel='linear', C=1.0)
```

Fit:

```
svc.fit(X_train, Y_train)
```

Predict:

```
prediction = lasso.predict(X)
```

Class	Time complexity	Out-of-core support	Scaling required	Kernel trick
LinearSVC	$O(m \times n)$	No	Yes	No
SGDClassifier	$O(m \times n)$	Yes	Yes	No
SVC	$O(m^2 \times n)$ to $O(m^3 \times n)$	No	Yes	Yes

Figure 36: Computational complexity of different SVM models

## Computational Complexity

### 7.50.2 Kernel tricks

Adds dimensions to data.

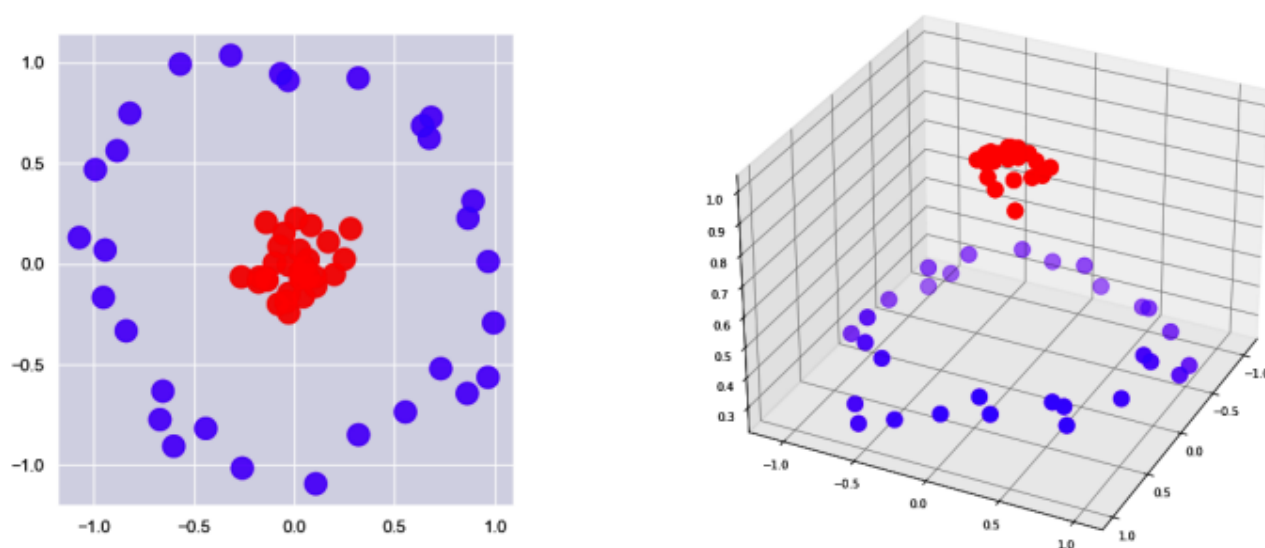


Figure 37: Kernel tricks

Always use linear kernel first (Linear SVC much faster than `SVC(kernel="linear")`) then switch to Gaussian RBF if dataset is not too large. **Kernel trick?, different types of kernels (linear, polynomial, gaussian rbf, sigmoid)**

### 7.50.3 Linear kernel

```
from sklearn.svm import LinearSVC
```

### 7.50.4 Polynomial kernel

### 7.50.5 Gaussian kernel

### 7.50.6 Gaussian RBF kernel (Gaussian Radial Basis Function kernel)

**Gaussian RBF** low gamma values, increasing gamma makes bell-shaped curve narrower ()

### 7.50.7 RBF kernel

Classification algorithm. Submodel of SVC.

Import: import module

```
from sklearn.svm import SVC
```

Initiate model:

```
rbf_kernel_svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="rbf", gamma=5, C=1))
])
```

Fit:

```
rbf_kernel_svm_clf.fit(X, y)
```

Predict:

Predict proba:

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

# Generate synthetic data
X, y = make_classification(n_samples=100, n_features=2, n_classes=2,
                          random_state=42)

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=42)

# Create SVC model with RBF kernel and probability estimates
svm_rbf = SVC(kernel='rbf', gamma='auto', probability=True)

# Train the model
svm_rbf.fit(X_train, y_train)

# Obtain probability estimates for the test set
probabilities = svm_rbf.predict_proba(X_test)

print("Probability estimates for the first 5 samples in the test set:")
print(probabilities[:5])
```

```
X, y = make_moons(n_samples=20, noise=0.2, random_state=42)

rbf_kernel_svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="rbf", gamma=5, C=1))
])
```



```
rbf_kernel_svm_clf.fit(X, y)

def plot_dataset(X, y, axes):
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs") # plot different classes
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^")
    plt.axis(axes)
    plt.grid(True, which='both')
    plt.xlabel(r"$x_1$", fontsize=20)
    plt.ylabel(r"$x_2$", fontsize=20, rotation=0)

def plot_predictions(clf, axes):
    x0s = np.linspace(axes[0], axes[1], 100)
    x1s = np.linspace(axes[2], axes[3], 100)
    x0, x1 = np.meshgrid(x0s, x1s)
    X = np.c_[x0.ravel(), x1.ravel()]
    # clf.predict(X) is under that format [1 1 1 ... 1 1 1]
    y_pred = clf.predict(X).reshape(x0.shape)
    # changed format to [[1 1 0 1 0 ... 1 0]]
    y_decision = clf.decision_function(X).reshape(x0.shape)
    plt.contourf(x0, x1, y_pred, cmap=plt.cm.brg, alpha=0.2) #contourf?
    plt.contourf(x0, x1, y_decision, cmap=plt.cm.brg, alpha=0.1)

plot_predictions(rbf_kernel_svm_clf, [-1.5, 2.5, -1, 1.5])
plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
plt.show()
```

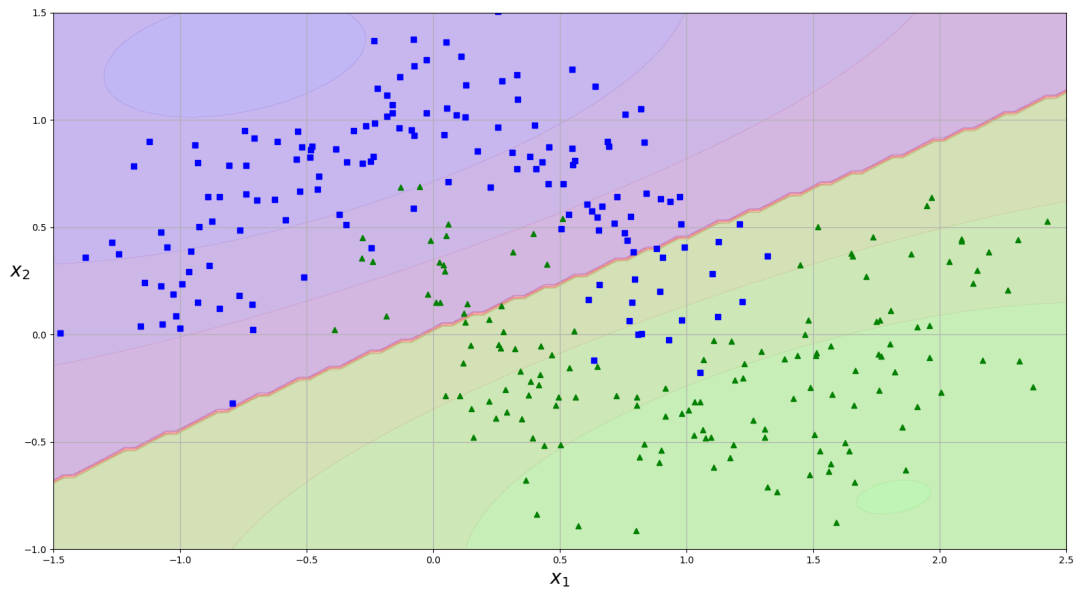


Figure 38

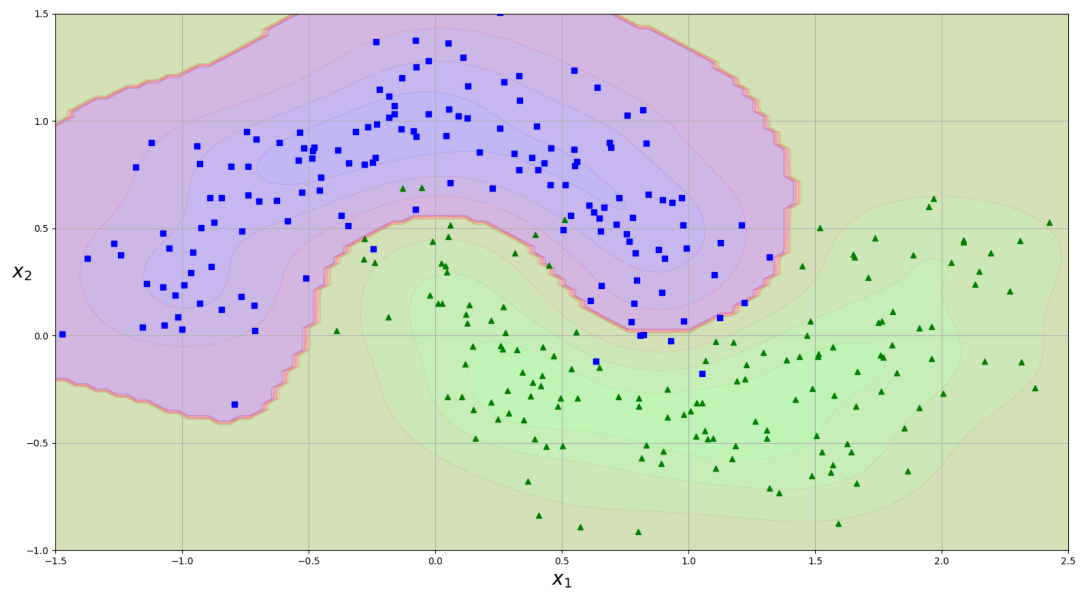


Figure 39

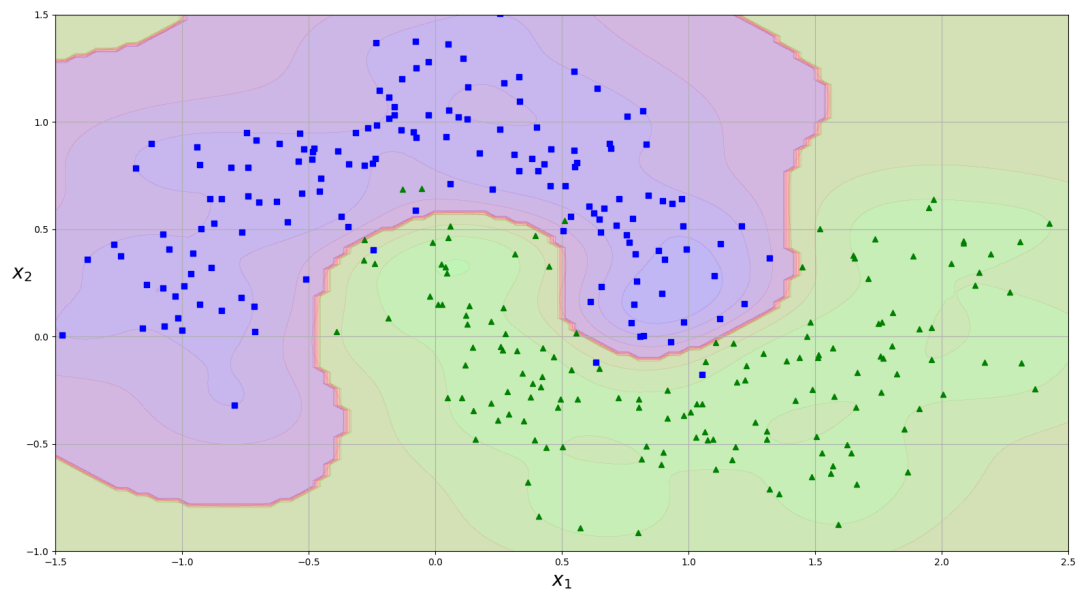


Figure 40

```
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
import numpy as np
import matplotlib.pyplot as plt
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

# Generate moon dataset
```

```

X, y = make_moons(n_samples=200, noise=0.2, random_state=42)

# Create an SVM classifier with RBF kernel
rbf_kernel_svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="rbf", gamma=5, C=1, probability=True)) # Enable
        probability estimates
])
rbf_kernel_svm_clf.fit(X, y)

# Plotting functions
def plot_dataset(X, y, axes):
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs") # Plot different classes
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^")
    plt.axis(axes)
    plt.grid(True, which='both')
    plt.xlabel(r"$x_1$", fontsize=20)
    plt.ylabel(r"$x_2$", fontsize=20, rotation=0)

def plot_predictions(clf, axes):
    x0s = np.linspace(axes[0], axes[1], 100)
    x1s = np.linspace(axes[2], axes[3], 100)
    x0, x1 = np.meshgrid(x0s, x1s)
    X = np.c_[x0.ravel(), x1.ravel()]
    # Obtain probabilities instead of predictions
    y_proba = clf.predict_proba(X)[: , 1].reshape(x0.shape)
    # Plot probability estimates as a color gradient
    plt.contourf(x0, x1, y_proba, cmap=plt.cm.brg, alpha=0.2)

# Plot predictions
plot_predictions(rbf_kernel_svm_clf, [-1.5, 2.5, -1, 1.5])

# Plot dataset
plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])

# Show the plot
plt.show()

```

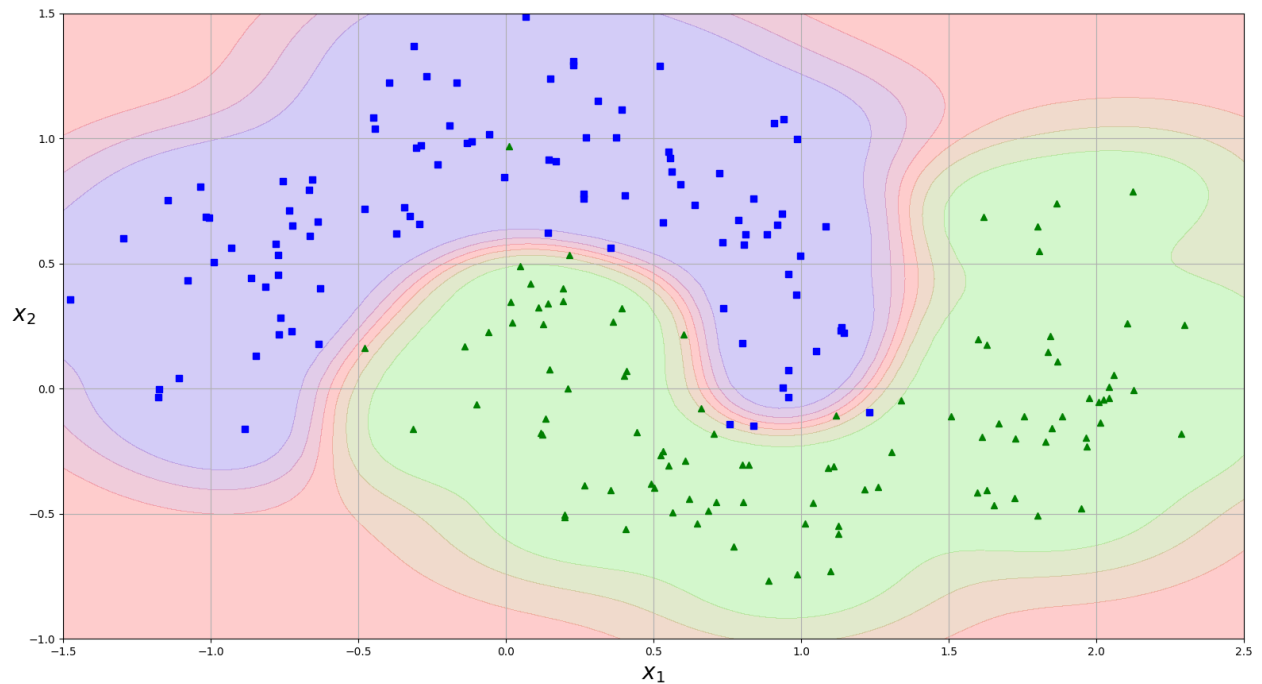


Figure 41

#### 7.50.8 Sigmoid kernel

#### 7.50.9 SVR (Support Vector Regression)

```
svm_reg = SVR()
```

#### 7.50.10 SVC (Support Vector Classification)

```
# Required Libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.svm import SVC
from mpl_toolkits.mplot3d import Axes3D

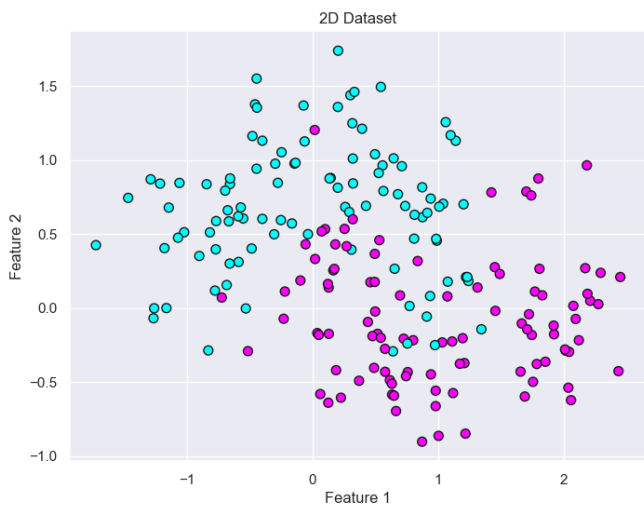
# Generate a 2D dataset using make_moons
x, y = make_moons(n_samples=200, noise=0.3, random_state=42)

# Plot the 2D dataset
plt.figure(figsize=(8, 6))
plt.scatter(x[:, 0], x[:, 1], c=y, cmap='cool', edgecolor='k', s=50)
plt.title('2D Dataset')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()

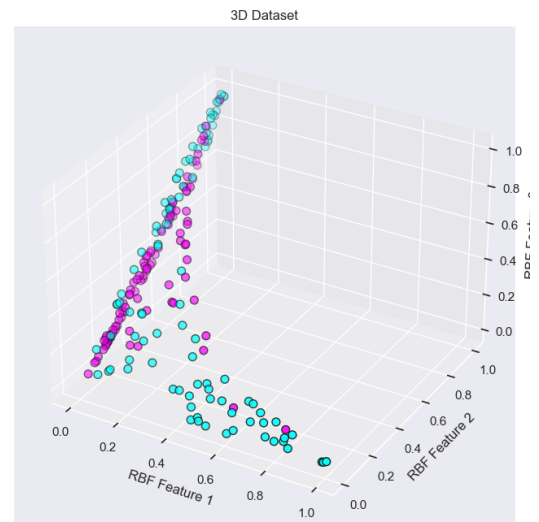
# Define an RBF kernel transformation function
def rbf_kernel(X, gamma=1.0):
    # Compute the pairwise squared Euclidean distances
    pairwise_sq_dists = np.square(X[:, np.newaxis] - X).sum(axis=2)
    # Apply the RBF kernel
    return np.exp(-gamma * pairwise_sq_dists)
```

```
# Transform the data into a higher-dimensional space using the RBF kernel
Z = rbf_kernel(X)

# Plot the transformed dataset in 3D
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(Z[:, 0], Z[:, 1], Z[:, 2], c=y, cmap='cool', edgecolor='k', s=50)
ax.set_title('3D Dataset')
ax.set_xlabel('RBF Feature 1')
ax.set_ylabel('RBF Feature 2')
ax.set_zlabel('RBF Feature 3')
plt.show()
```



(a) Before kernel trick



(b) After kernel trick

Figure 42: Dataframe values extraction

### 7.50.11 SMV (Soft Margin Classification)

Allows for misclassifications to achieve better generalization by introducing a margin of violation.

### 7.50.12 Polynomial SVM

Classification algorithm.

Make moons classification:

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

X, y = make_moons(n_samples=500, noise=0.3, random_state=42)
polynomial_svm_clf = Pipeline([
    ("poly_features", PolynomialFeatures(degree=3)), #/#
    ("scaler", StandardScaler()), #/#
    ("svm_clf", LinearSVC(C=100, loss="hinge")) #/#
])
polynomial_svm_clf.fit(X, y)

def plot_dataset(X, y, axes):
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs")
```

```

plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^")
plt.axis(axes)
plt.grid(True, which='both')
plt.xlabel(r"$x_1$", fontsize=20) #### x label
plt.ylabel(r"$x_2$", fontsize=20, rotation=0) #### y label

def plot_predictions(clf, axes):
    x0s = np.linspace(axes[0], axes[1], 100)
    x1s = np.linspace(axes[2], axes[3], 100)
    x0, x1 = np.meshgrid(x0s, x1s)
    X = np.c_[x0.ravel(), x1.ravel()] #### c_? & ravel?
    y_pred = clf.predict(X).reshape(x0.shape)
    y_decision = clf.decision_function(X).reshape(x0.shape)
    plt.contourf(x0, x1, y_pred, cmap=plt.cm.brg, alpha=0.2) #### contourf?
    plt.contourf(x0, x1, y_decision, cmap=plt.cm.brg, alpha=0.1)

plot_predictions(polynomial_svm_clf, [-1.5, 2.5, -1, 1.5])
plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
plt.show()

```

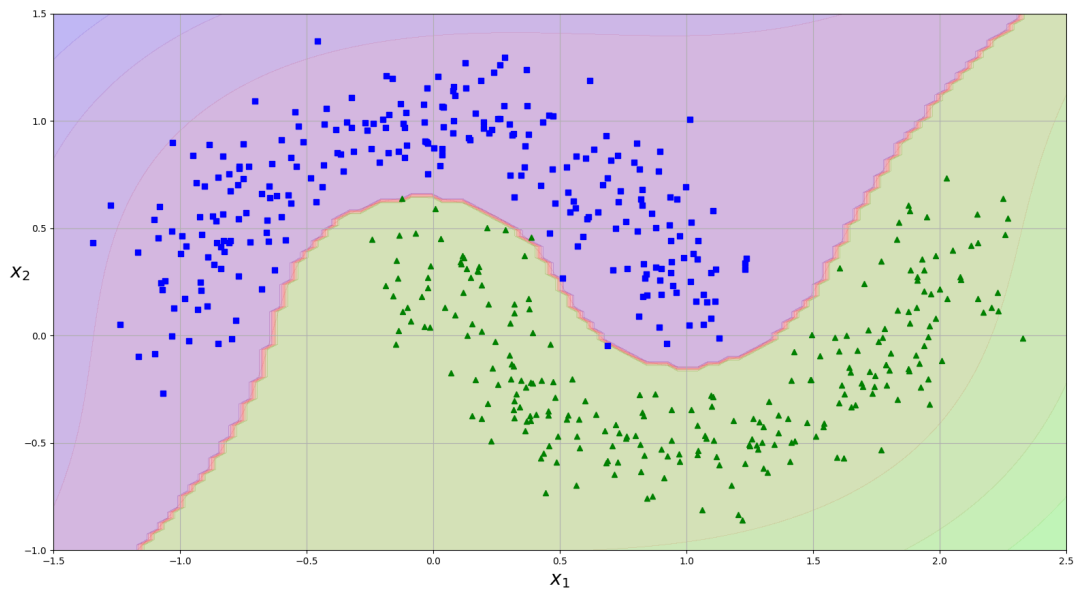


Figure 43

After introducing noisy data:

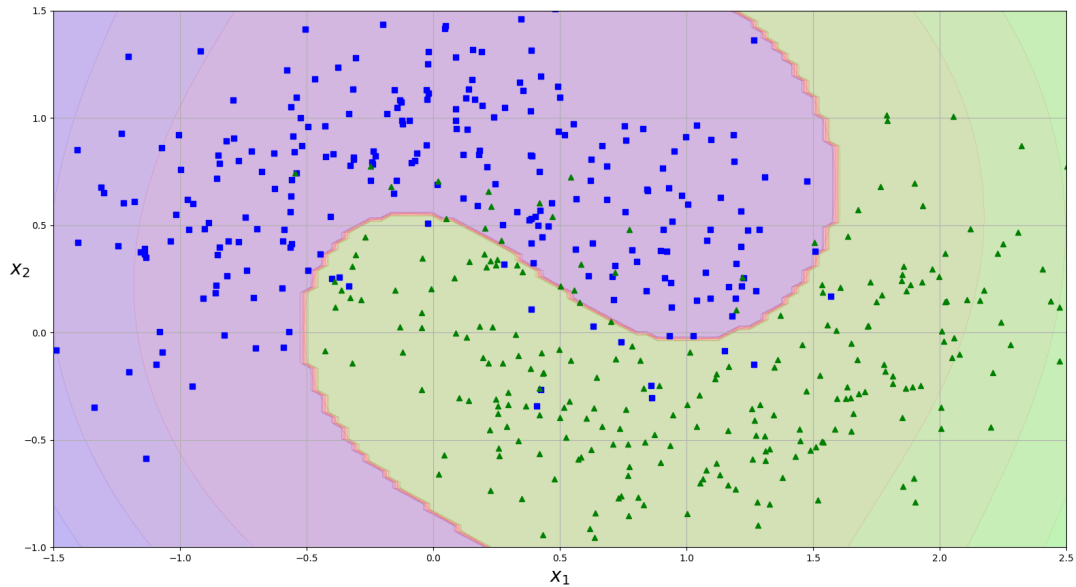


Figure 44

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Generate pseudo-random dataset with 3 informative features
X, y = make_classification(n_samples=500, n_features=3, n_informative=3,
                           n_redundant=0, n_classes=2, random_state=42)

# Split dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=42)

# Create Polynomial SVM classifier pipeline
polynomial_svm_clf = Pipeline([
    ("poly_features", PolynomialFeatures(degree=10)),
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="poly", degree=10, C=10000))
])

# Train the classifier
polynomial_svm_clf.fit(X_train, y_train)

# Make predictions
y_pred = polynomial_svm_clf.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

```

# Generate grid points for the features
x0_min, x0_max = X[:, 0].min() - 1, X[:, 0].max() + 1
x1_min, x1_max = X[:, 1].min() - 1, X[:, 1].max() + 1
x0, x1 = np.meshgrid(np.arange(x0_min, x0_max, 0.1),
                     np.arange(x1_min, x1_max, 0.1))

# Flatten the grid points and create a third feature with zeros
x2 = np.zeros_like(x0)
X_grid = np.c_[x0.ravel(), x1.ravel(), x2.ravel()]

# Predict the class labels for the grid points
y_grid = polynomial_svm_clf.predict(X_grid)
y_grid = y_grid.reshape(x0.shape)

# Create the 3D scatter plot
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# Plot the data points
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=y, cmap=plt.cm.Paired)

# Plot the decision boundary
ax.plot_surface(x0, x1, y_grid, alpha=0.2)

ax.set_xlabel('Feature 1')
ax.set_ylabel('Feature 2')
ax.set_zlabel('Feature 3')
ax.set_title('Polynomial SVM Classification')
plt.show()

```

## 7.51 Naive Bayes

Often used for text classification.

## 7.52 Deep Learning

### 7.52.1 Whazit?

Neural networks, Deep Boltzmann Machines (DBM).

## 7.53 ANN (Artificial Neural Networks)

### 7.53.1 Description

Neural networks are 'cells' linked by mathematical weight to pass information. **DNN** (aka deep neural networks) are neural networks with multiple layers,

### 7.53.2 Activation functions

Input is a value between 0 and 1. [documentation here: https://keras.io/api/layers/activations/](https://keras.io/api/layers/activations/) try a few

### 7.53.3 Optimizers

Update weights parameters to reduce loss function. <https://keras.io/api/optimizers/>  
<https://keras.io/api/losses/>

### 7.53.4 Weights and biases

### 7.53.5 Backpropagation

[i know the word at least ...](#) backpropagation uses reverse-mode autodiff



### 7.53.6 Neurons weights

Initialized at random.

### 7.53.7 Weight-initialisation

Generated at random

`keras.layers.Dense(10, activation='relu', kernel_initializer='he_normal')`

### 7.53.8 Training loop

Consider an input vector  $\mathbf{x}$  (a sample from the dataset), a matrix  $\mathbf{w}$  (containing the weights of a model, those being between 0 and 1) and a target value  $y$ .

**Batch**

Draw a batch of training samples.

**Forward pass**

Run the model to obtain predictions.

**Loss**

Compute the loss to update weights based on the loss from the batch used in forward pass.

**Backward pass**

Come back to update the weights.

### 7.53.9 Dropout

At each iteration a random subset of neurons are dropped out (they return 0). Forces network not to rely on the same networks. Tends to slow down convergence to solution but builds stronger models. Slows down training time (by a factor of 2 or around but has no impact on inference time).

### 7.53.10 Learning rate

Update weights based on the gradient but slower to reduce bouncing.

$$w = learning\_rate * gradient$$

### 7.53.11 Vanishing vs. exploding gradient

When performing backpropagation and going from output neuron to input neuron weights are updated using cost function of each neuron in a Gradient Descent step.

**Vanishing** Lower layer's weights virtually unchanged during backpropagation.

**Exploding** Algorithm diverges during backpropagation because of excessive weights.

### 7.53.12 Gradient clipping

To reduce exploding gradient problem by setting threshold.

### 7.53.13 Transfer learning

In order to save time and computing power, reusing models already able to do things might be considerable.

### 7.53.14 Sparse models

**purpose? other than speed of prediction** To get one, zero out tiny weights, apply L1 regularization, use TensorFlow model optimization toolkit.

### 7.53.15 Loss functions

## 7.54 Activation functions

Reference to 7.53.2

### 7.54.1 ReLU (Rectified Linear Unit)

$$\text{ReLU}(z) = \max(0, z)$$

### 7.54.2 RReLU (Randomized leaky ReLU)

### 7.54.3 PReLU (Parametric leaky ReLU)

Outperforms ReLU on large datasets but on small datasets tends to overfit.

### 7.54.4 Sigmoid

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

### 7.54.5 Hyperbolic Tangent

$$\tanh(z) = 2\sigma(2z) - 1 = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Where  $\sigma(z)$  is still the sigmoid function.

### 7.54.6 Softmax

$$\frac{1}{1}$$

detail?

### 7.54.7 Softplus

$$\text{softplus}(z) = \log(1 + \exp(z))$$

### 7.54.8 ELU (Exponential Linear Unit)

### 7.54.9 SELU (Scaled ELU)

A few advantages over ReLU such as: it can take on negative values, always has a nonzero derivative, ensures the model is self-normalized to solve exploding/ vanishing gradients

## 7.55 Optimizers

Reference to 7.53.3.

### 7.55.1 SGD (Stochastic Gradient Descent)

To calculate optimal weight update. **Stochastic** refers to random batch sampling.

```
from sklearn.linear_model import SGDRegressor

# ... (build model)

optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy', metrics=[
    'accuracy'])
```

capivalue, lr, momentum, nesterov

### 7.55.2 Mini-batch SGD

Using SGD algorithm (mentioned in 7.55.1), on a sample of the dataset.

### 7.55.3 Batch SGD

Using SGD algorithm (mentioned in 7.55.1), on the whole dataset.

### 7.55.4 Momentum

Momentum: compare previous change in gradient, if they move in the same direction will take more speed (bigger than previous step).

### 7.55.5 NAG (Nesterov Accelerated Gradient)

### 7.55.6 AdaGrad

Risk of slowing too fast, small updates with features occurring often and big ones with features occurring often.

### 7.55.7 RMSProp (Root Mean Square Propagation)

Different learning rate for each parameter.

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

rho, lr

### 7.55.8 Adam (ADaptive Moment estimation)

Combination of momentum and RMSProp with AdaGrad

### 7.55.9 AdaMax

### 7.55.10 Nadam (Nesterov Adam)

```
optimizer = keras.optimizers.Nadam(learning_rate=0.001)
```

## 7.56 Loss Functions

Reference to 7.53.15

### 7.56.1 Binary crossentropy

## 7.57 RNNs (Recurring Neural Networks)

"The batter hits the ball. The outfielder immediately starts running, anticipating the ball's trajectory. He tracks it, adapts his movements, and finally catches it (under a thunder of applause). Predicting the future is something you do all the time, whether you are finishing a friend's sentence or anticipating the smell of coffee at breakfast. In this chapter we will discuss recurrent neural networks (RNNs), a class of nets that can predict the future (well, up to a point, of course). They can analyze time series data such as stock prices, and tell you when to buy or sell. In autonomous driving systems, they can anticipate car trajectories and help avoid accidents. More generally, they can work on sequences of arbitrary lengths, rather than on fixed-sized inputs like all the nets we have considered so far. For example, they can take sentences, documents, or audio samples as input, making them extremely useful for natural language processing applications such as automatic translation or speech-to-text." - Aurelien Geron, "Hands-on ML".

To summarize, RNNs are a good fit for time series time series forecasting. By that it is meant that this neural network model might be a good fit to predict future values or intermediate missing values (e.g. number of active users on a website is known before February 13th and after February 15th but not for the 14th, time series can help finding this missing value, this process is called imputation). Note that  $\tanh(x)$  function is preferred to ReLU in RNNs. Two main problems to be faced are unstable gradients and limited memory.

### 7.57.1 Unstable gradients

Recurrent dropouts and recurrent layer normalization. Neuron sending to itself cumulates (e.g. 1.05 change on and on makes  $1.05^{10}$  if ten times)

### 7.57.2 Memory cells

Input is output from the same cell at time  $t - 1$ , indeed this behavior is consider as the one of a memory. **cell initialization (distributing data? and forward knowledge?)** Typically neurons can memorize as much as 10 steps of data for most of them but some are able to memorize up to 100. Particularity is to get sequence as input and sequence as output which is indeed a good fit for time series forecasting.

### 7.57.3 Limited memory

### 7.57.4 Encoder/decoder

- ◇ Encoder: sequence-to-vector
- ◇ Decoder: vector-to-sequence

Useful for translation.

### 7.57.5 Stateless

Network's hidden layer is reset after processing each input sequence.

## 7.58 CNNs (Convolutional Neural Networks)

### 7.58.1 Description

Useful for visual perception, voice recognition and natural language processing. Scientists (David H. Hubel and Torsten Wiesel) noticed that neurons dedicated to vision spark only when a specific slice of the eye's view has input, in other words they only react in a limited region of the visual cortex. They noticed too that certain neurons only react with specific images features such as horizontal images, different orientation images and so on. Originally CNN were called neocognitron (specificity was not being sensitive to image shift). Issue with CNN is the memory requirement and ressources consumption making it hard to proceed to search grid to find optimal parameters (such as number of filters and so on). Basic idea is to take tiles from images.

### 7.58.2 Architectures

Stack of a few convolutional layers each one generally followed by a ReLU layer, then a pooling layer then another convolutional layer with ReLU then another pooling layer and so on. The purpose of layering like this is to emphasis on smaller and smaller parts of the image as we advance in the network.

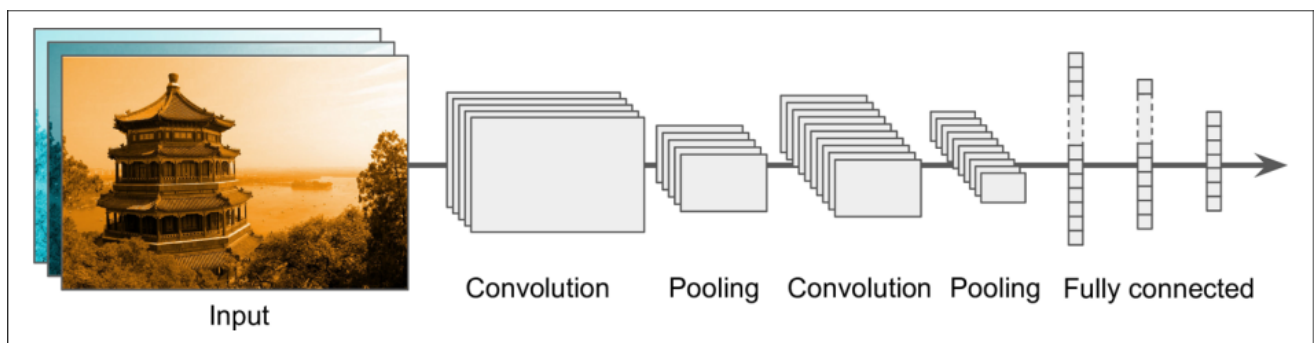


Figure 45: CNN layers with rectangular receptive fields

### 7.58.3 Convolutional layers

Neurons in the first convolutional layer are not connected with every pixel on the input image.

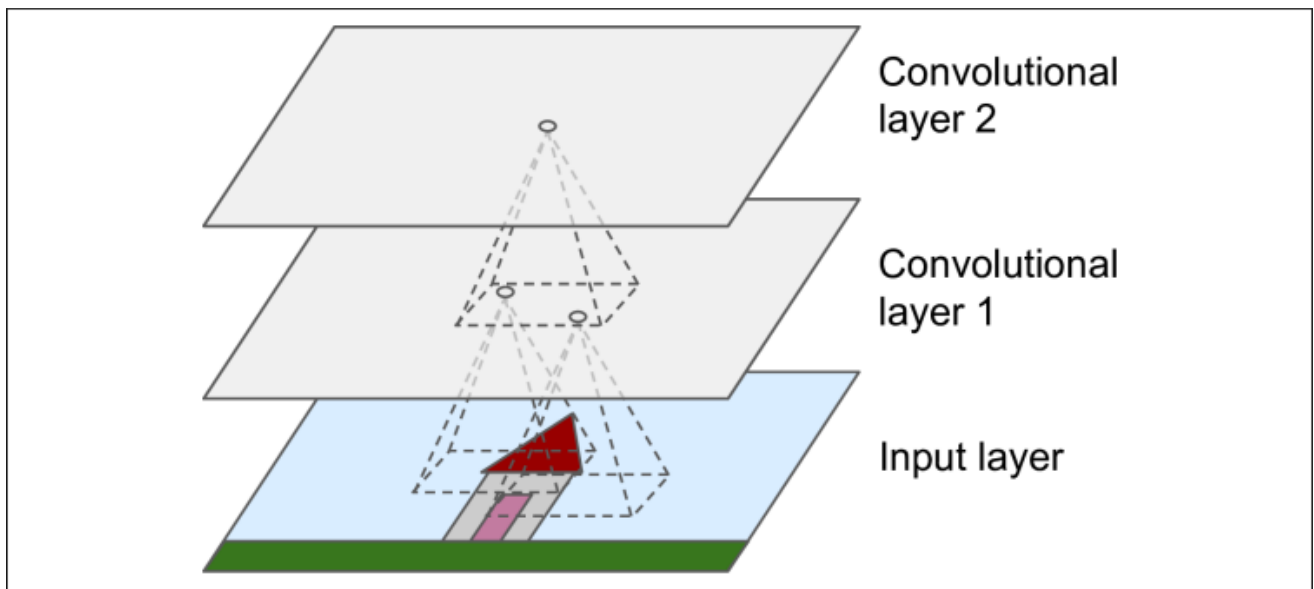


Figure 46: CNN layers with rectangular receptive fields

#### 7.58.4 Pooling layers

Purpose is to subsample the input image to limit the risk of overfitting. Here image pixels take the most intensive one.

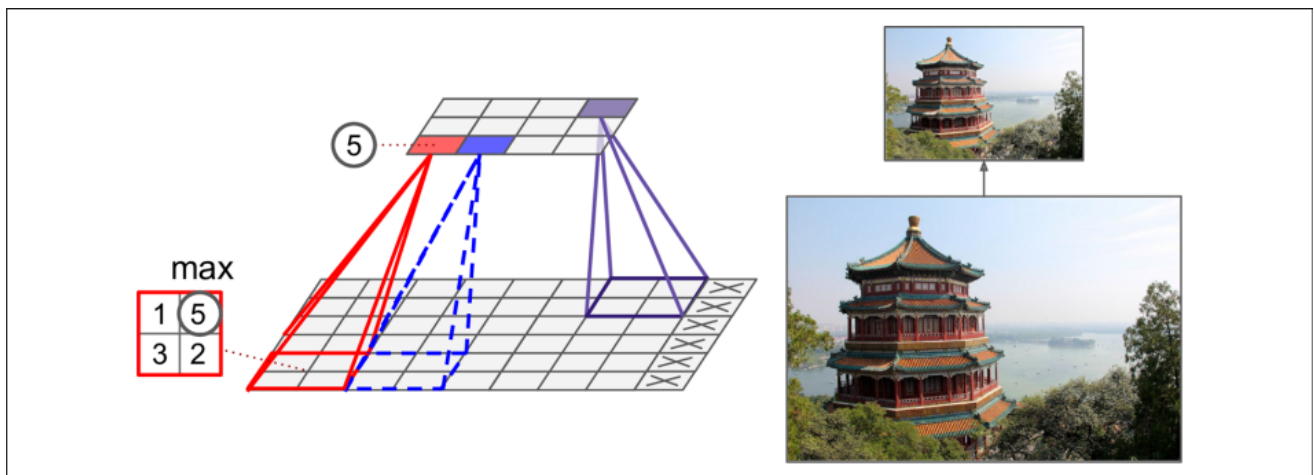


Figure 47: Max pooling layer (2 x 2 pooling kernel, stride 2, no padding)

Size of final image is half the original one (rounded down because there is no padding as can be seen on the side cases)

#### 7.58.5 MaxPooling

#### 7.58.6 Filters

vertical/horizontal lines

#### 7.58.7 Transfer learning

When not enough training data is available, might be smart to use lower layers of a pretrained model.

### 7.58.8 Localization

### 7.58.9 Object detection

## 7.59 GANs (Generative Adversial Networks)

Useful to generate artificial data. (e.g. train on faces to generate fake faces). Composed of two neural networks, one to generate data and the other to discriminate (between true data and generated data).

## 7.60 ANN (Artificial Neural Network) models & configuration

Those can be either supervised or unsupervised but are often both. `sequential`, `RNN (Recurring Neural Networks)` not a specific model but some ANNs work this way, `DNN (Deep Neural Networks)`

### 7.60.1 Import layers

```
from tensorflow.keras.layers import LSTM, Dense
```

### 7.60.2 Import models

```
from tensorflow.keras.models import Sequential
```

### 7.60.3 Input layer

```
input = keras.layers.Input(shape=(<shape_x>, <shape_y>))
```

*shape*: input table shape

### 7.60.4 Multimodal Input

### 7.60.5 `layers.Flatten(input_shape(x,y))`

Input image for example with dimensions of 28 by 28:

```
import numpy as np
from tensorflow.keras import Sequential, layers

# Generate data
random_matrix = np.random.rand(4, 4)

# Create model
model = Sequential()
model.add(layers.Flatten(input_shape=(4, 4)))
```

With the matrix: `array([[0.60872335, 0.99446746, 0.28128227, 0.64169977], [0.14730914, 0.90998328, 0.33204108, 0.85083919], [0.80004922, 0.29794655, 0.10482782, 0.30146641], [0.18502237, 0.0376533 , 0.44252062, 0.02067095]])`  
For colored images:

```
import numpy as np
from tensorflow.keras import Sequential, layers

# Generate data
random_rgb_matrix = np.random.rand(4, 4, 3)

# Create model
model = Sequential()
model.add(layers.Flatten(input_shape=(4, 4, 3)))
model.summary()
```

where the matrix looks like this:`array([[0.63192061, 0.18524456, 0.83385466], [0.79491155, 0.92942268, 0.5088006 ], [0.5981006 , 0.56562488, 0.88500774], [0.79289695, 0.9659789 , 0.1461061 ]], [[0.24680856, 0.75861799, 0.64804456], [0.76327029, 0.29416709, 0.15149997], [0.40057238, 0.99504103, 0.19209781], [0.54279693, 0.55907711, 0.75802485]], [[0.75123018, 0.52956408, 0.2450316 ], [0.09417945, 0.04883064, 0.34909757], [0.71825547, 0.54400082, 0.81305528], [0.63590995, 0.39435362, 0.47630061]], [[0.7187529 , 0.2487458 , 0.64103506], [0.26499111, 0.92787077, 0.46668161], [0.9789001 , 0.53370229, 0.70885125], [0.43296265, 0.37520249, 0.71286807]]])` , because each pixel has 3 components, note that for 3D greyscaled images it works decently well but for colorful images it becomes less efficient because it is hard for the model to learn about details of images.

#### 7.60.6 Dense layer

```
hidden = keras.layers.Dense(30, activation="relu")(input_)
```

#### 7.60.7 LSTM layer

```
hidden = keras.layers.LSTM(30, activation="relu")(input_)
```

#### 7.60.8 Output layer

```
output = keras.layers.Dense(1)(concat)
```

### 7.60.9 Compile model

```
model.compile(loss=["mse", "mse"],
              loss_weights=[0.9, 0.1],
              optimizer="sgd",
              metrics=["accuracy"])
```

Use different weights for loss functions with the hyperparameter *loss\_weights*.

### 7.60.10 Fit model

```
history = model.fit(X_train, y_train, epochs=20, validation_data=X_test, y_valid))
```

### 7.60.11 Plot fit results

```
pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()
```

```
pyplot.plot(<model>.history['loss'], label='train', )
pyplot.plot(<model>.history['val_loss'], '--', label='test',)
```

### 7.60.12 Save model to HDF5

```
model.save("<file_path.h5>")
```

### 7.60.13 Load model

```
model = keras.models.load_model(<file_path.h5>)
```

### 7.60.14 Checkpoints

```
checkpoint_cb = keras.callbacks.ModelCheckpoint(<file_path.h5>)
history = model.fit(X_train, y_train, epochs=10, callbacks=[checkpoint_cb])
```

*callbacks* is the destination at which to save models. Set *save\_best\_only = True* to only save the best models (might be useful when gradient explodes, to avoid model from being saved at that moment).



### 7.60.15 Checkpoints versioned

```
class SaveModelCallback(keras.callbacks.Callback):
    def __init__(self, model_index):
        super().__init__()
        self.model_index = model_index

    def on_epoch_end(self, epoch, logs=None):
        model_filename = f"model_{self.model_index}.h5"
        self.model.save(model_filename)
        print(f"Model {self.model_index} saved as {model_filename}")

checkpoint_callback = SaveModelCallback(model_index=i)
history = model.fit(X_train, y_train, epochs=10, callbacks=[checkpoint_callback])
```

### 7.60.16 Adaptive learning rate

Reduce on plateau:

```
lr_scheduler = keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=5)
```

factor, patience

Exponential decay:

```
learning_rate = keras.optimizers.schedules.ExponentialDecay(0.01, s, 0.1)
optimizer = keras.optimizers.SGD(learning_rate)
```

### 7.60.17 Transfer learning

```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

Use this code to do **not** save **in** A when saving B:  
model\_A\_clone = keras.models.clone\_model(model\_A)  
model\_A\_clone.set\_weights(model\_A.get\_weights())

### 7.60.18 Early stopping

```
checkpoint_cb = keras.callbacks.ModelCheckpoint(<file_path.h5>)
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10, restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb, early_stopping_cb])
```

patience what exactly, restore\_best\_weights signification?

### 7.60.19 Regularization techniques

L2 (Ridge):

```
layer = keras.layers.Dense(100, activation="elu", kernel_regularizer=keras.
    regularizers.l2(0.01))
```

0.01 but what?

Template layer:

```
from functools import partial
RegularizedDense = partial(keras.layers.Dense,
    activation="elu",
    kernel_initializer="he_normal",
    kernel_regularizer=keras.regularizers.l2(0.01))

RegularizedDense(10, activation="softmax", kernel_initializer="glorot_uniform")
```

### 7.60.20 Initializer

He normal:

```
layer = keras.layers.Dense(100, activation="elu", kernel_initializer="
    he_normal")
```

glorot uniform:

```
kernel_initializer="glorot_uniform"
```

### 7.60.21 Dropout

```
keras.layers.Dropout(rate=0.2)
```

Drop weights about random neurons.

### 7.60.22 Constraint

```
layer = keras.layers.Dense(100, kernel_constraint=keras.constraints.max_norm(1.))
```

### 7.60.23 Custom activation/initializer/regularizer/constraint

```
def my_softplus(z):
    return tf.math.log(tf.exp(z) + 1.0)
def my_glorot_initializer(shape, dtype=tf.float32):
    stddev = tf.sqrt(2. / (shape[0] + shape[1]))
    return tf.random.normal(shape, stddev=stddev, dtype=dtype)
def my_l1_regularizer(weights):
    return tf.reduce_sum(tf.abs(0.01 * weights))
def my_positive_weights(weights): # return value is just tf.nn.relu(weights)
    return tf.where(weights < 0., tf.zeros_like(weights), weights)

layer = keras.layers.Dense(30, activation=my_softplus,
    kernel_initializer=my_glorot_initializer,
    kernel_regularizer=my_l1_regularizer,
    kernel_constraint=my_positive_weights)
```

describe activation functions

### 7.60.24 Compile

```
model.compile(optimizer=optimizer, loss=loss, metrics=metrics)
```

loss = 'sparse\_categorical\_crossentropy'

### 7.60.25

### 7.60.26 RNN

Import RNN layer.

```
from keras.layers import SimpleRNN
```

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, input_shape=[None, 1])
    keras.layers.SimpleRNN(20),
    keras.layers.SimpleRNN(1)
])
```

### 7.60.27 Autoencoders

Neural networks designed for dimensionality reduction, image denoising, anomaly detection, feature extraction and data generation.

### 7.60.28 Hopfield network

### 7.60.29 RBM (Restricting Boltzmann techniques)

neither do i know what this is

### 7.60.30 GAN (Generative Adversial Network)

oh yeah ian's thing

### 7.60.31 LSTM (Long Short-Term Memory)

Unsupervised & supervised, designed to avoid long-term dependencies problems. RNN based.

- Forget: output between 0 and 1, decides whether past should be forgotten or kept
- Input: chooses which data need to be stored
- Output: decides what will yield out of each cell

Converges faster and detects long-term behavior with more accuracy.

### 7.60.32 WaveNet

### 7.60.33 Gru (Gated Recurrent Unit) cells

### 7.60.34 LeNet-5

Mainly used in the banking system to recognize handwritten digits, introduced by Yann LeCun.

### 7.60.35 AlexNet

Convolutional layers directly on top of one another instead of stacking a pooling layer on top of each convolutional layer. To reduce overfitting two major regularization techniques were used (dropout at rate of 50% and data augmentation with random image shift/offsets/flipping horizontally/light dimming/...).

### 7.60.36 ZF Net

Subclass of AlexNet network, developed by Matthew Zeiler and Rob Fergus (won ILSVRC 2013). Basically it is an AlexNet with a few tweaked hyperparameters such as number of feature maps/kernel size/stride/...

### 7.60.37 GoogLeNet

Christian Szegedy et al. from Google (won ILSVRC 2014). Makes use of a subnetwork called "inception module" using thus 10 times less hyperparameters (roughly 6 millions instead of 60 millions). Insensitive to the position of the object on the image. Later versions of their inception function were proposed over time.

### 7.60.38 VGGNet (Visual Geometry Group Net)

Developed by Karen Simonyan and Andrew Zisserman (won ILSVRC 2014) works based on a classical architecture, with 2 or 3 convolutional layers and a pooling layer, then again 2 or 3 convolutional layers and a pooling layer, and so on (reaching a total of just 16 or 19 convolutional layers, depending on the VGG variant), plus a final dense network with 2 hidden layers and the output layer. It used only 3x3 filters, but many filters.

### 7.60.39 ResNet (Residual Network)

Extremely deep CNN of 152 layers with other variants of 34, 50 and 101 layers (won the ILSVRC 2015). Key to be able to train such networks is to skip connections using shortcuts.

**Connections skip/shortcut connections** The signal feeding into a layer is also added to the output of the layer located a bit higher. Up on the stack. When you initialize a regular neural network, its weights are close to zero, so the network just outputs values close to zero. If you add a skip connection, the resulting network just outputs a copy of its inputs; in other words, it initially models the identity function. If the target function is fairly close to the identity function (which is often the case), this will speed up training considerably. If many skip connections are added, progress can be seen without even having all the layers being trained yet (because the signals can make their way through the whole network easily).

**RUs (Residual Units)** The deep network with skipped and untrained connections can become a stack of residual units where each residual unit is a small neural network with a skip connection.

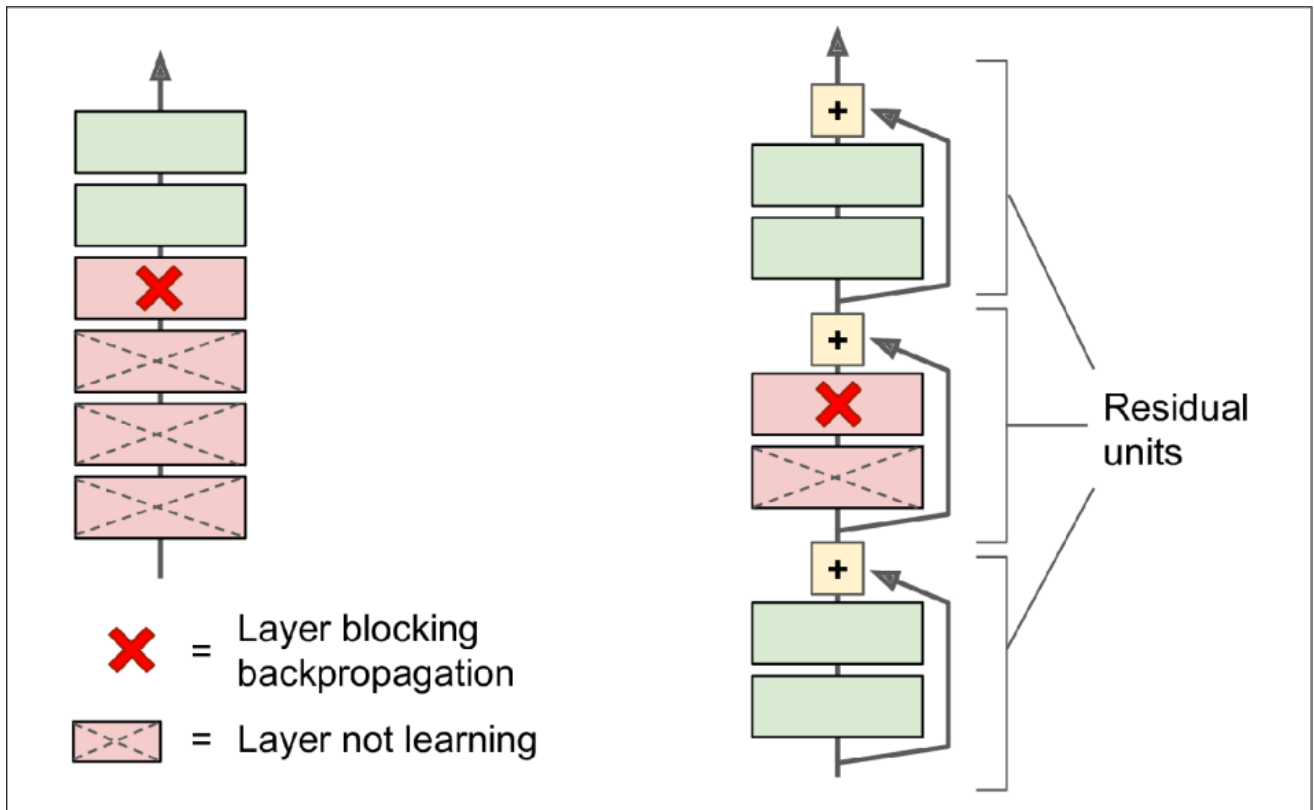


Figure 48: Regular DNN (Deep Neural Network) vs. DRN (Deep Residual Network)

#### 7.60.40 Xception (Extreme Inception)

Variant of Google's GoogLeNet, proposed in 2016 by François Chollet. Replaces the inception modules with a special type of layer called a depthwise separable convolution layer (or separable convolution layer for short). Composed of two parts (one that applies a single filter to each input feature map and the second which only searches for cross-channel patterns).

#### 7.60.41 SENet (Squeeze-and-Excitation Network)

(won ILSVRC 2017)

#### 7.60.42 FCN (Fully Convolutional Networks)

#### 7.60.43 YOLO (You Only Look Once)

Proposed by Joseph Redmon in 2015, so fast that it is used to run in real time on videos.

### 7.61 Reinforcement Learning

#### 7.61.1 Overview

Agent makes observations and takes actions within an environment to maximize rewards. Some of these factors can be omitted (e.g. a vacuum doesn't care about its environment and cares mostly about the amount of dust it can gather, to escape a maze a robot only gets penalty at each step it makes because it is making the path longer so the end goal is only to minimize cost).

#### 7.61.2 Agent

Entity that performs the action.

#### 7.61.3 Environment

The world in which the agent resides.

#### 7.61.4 State

Current environment. not sure about this one

#### 7.61.5 Reward

Immediate return sent by the environment to evaluate the last action of the agent.

#### 7.61.6 Action

Things agent can do.

#### 7.61.7 Policy

Can be neural network taking observations as input and outputting the actions to take. Defined as  $\pi(S)$ , it is the behavior depending upon the current status.

#### 7.61.8 Value Function

The goal being to maximize future rewards not immediate next step can be mathematically explained by maximizing the cumulative discounted rewards which can be expressed by following equations:

$$G_t = R_{t+1} + \gamma R_{t+1} + \dots = \sum_0^{\infty} \gamma^k R_{t+k+1}$$

Where  $\gamma$  is the discounting factors and resides between 0 and 1.

#### 7.61.9 Q-Learning

Value function based, works by picking actions depending upon Q-values with:  $a_t = \max_a Q(s_t, a)$ , by applying e-greedy approach to select random action with probability  $\epsilon$ . Q-values might be hard to calculate due to the fact that states and actions spaces can be very large, thus ANNs can be used to approximate these values.

#### 7.61.10 Markov decision processes

Action leads to state

**Optimal state value** sum of all discounted future rewards the agent can expect on average after it reaches a state s

**Partially Observable Markov Decision Process (POMDP)** Not everything is observable but what is observed is a derivation of it.

#### 7.61.11 Model based vs model free

Determined whether rewards and probabilities for each step are readily accessible. Model-based ends there but model-free makes use of policy and value functions.

#### 7.61.12 Model

transition probability function (P) and reward function

#### 7.61.13 Model tuning

Few parameters can be optimized but requires a lot of resources.

**Gamma (discount factor)** Decaying gamma will prioritize short-term learning.

**Epsilon** Drives exploration/exploitation tendencies.

**Episodes and batch size** A higher number of episodes and larger batch size in the training set will lead to better training and a more optimal Q-value.

**Number of layers and nodes of the deep learning model** Better training and more optimal Q-values.

#### 7.61.14 Policy gradient methods

### 7.62 Policy Gradient Methods

More information about what is policy gradient method in section 7.61.14

#### 7.62.1 Stochastic policy

#### 7.62.2 Genetic policy

Kill worst policies and keep best performing ones.

#### 7.62.3 Deep Deterministic Policy Gradient (DDPG)

#### 7.62.4 Epsilon greedy policy

#### 7.62.5 SARSA

Takes its name from iterating over  $S_t, A_t, R_t, S_{t+1}, A_{t+1}$ .

#### 7.62.6 Credit assignment problem

If last action caused failure of the system in the environment last action will be taken as responsible but surely is not completely behind failure.

**Discount factor** Solution was found in evaluating action based on the sum of all rewards that come after it with a discount factor  $\gamma$ .

**Action advantage** If a good action is followed by very bad actions, discount factor will creditate lower than deserved, action advantages tries to solve this. To perform this method normalization is used amongst many episodes.

### 7.63 Reinforcement Learning models

#### 7.63.1 REINFORCE

#### 7.63.2 Q-Learning

Adaptation of the Q-Value Iteration algorithm to the situation where the transition probabilities and the rewards are initially unknown. Q-Learning works by watching an agent play (e.g., randomly) and gradually improving its estimates of the Q-Values accurate Q-Value estimates (or close enough), then the optimal policy is choosing the action that has the highest Q-Value (i.e., the greedy policy). Advantage is being able to learn from agent acting randomly. Takes its name from the function  $Q(s, a)$  that returns expected reward based on the state  $s$  and by executing action  $a$ .

#### 7.63.3 Deep Q-Networks (DQN)

#### 7.63.4 Double DQN

Based on the observation that the target network is prone to overestimating Q-Values.

- 7.63.5 DDQN (Dueling DQN)
- 7.63.6 Proximal Policy Optimization (PPO)
- 7.63.7 Asynchronous Advantage Actor-Critic (A3C)
- 7.63.8 Advantage Actor-Critic (A2C)
- 7.63.9 Soft Actor-Critic (SAC)
- 7.63.10 Trust Region Policy Optimization (TRPO)
- 7.63.11 Proximal Policy Optimization (PPO)
- 7.63.12 Curiosity-based exploration

## 8 Statistics Appendix (Math 001)

### 8.1 Mean

$$m = \frac{(x_1 + x_2 + \dots + x_n)}{N}$$

### 8.2 Median

$$M =$$

### 8.3 Mode

Value with highest frequency.

### 8.4 Variance

Measurement of how much points are scattered from the mean.

$$Var(x) = \sigma^2 = \frac{1}{n-1} \sum (x_i - \bar{x})^2$$

Where  $\bar{x}$  is the average and the whole thing is squared because it is a measure of distance.  $\sigma$  itself is the standard-deviation, which is detailed further in 8.7 **why n-1?**

### 8.5 Covariance

$$Covar(x) = \frac{1}{n-1} \sum (x_i - \bar{x})(y_i - \bar{y})$$

### 8.6 Beta

$$\beta = \frac{cov(x, y)}{var(x)}$$

### 8.7 Standard-Deviation

$$\sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n-1}}$$

Where  $n$  is the size of the sample,  $x_i$  is an individual element,  $\bar{x}$  is the average value.



## 8.8 Standard Error

$$SE = \frac{s}{\sqrt{n}}$$

Where  $s$  is the sample standard deviation,  $n$  is the sample size.

## 8.9 Relative Standard Error

Used for instruments, calculated as the standard deviation as a percentage of the mean.

$$RSE = \frac{SE}{\theta} * 100$$

$SE$  is the standard error,  $\theta$  is the sample mean.

Lower the RSE, the more precise the measurement.

## 8.10 Pearson Correlation

$$Corr(x, y) = \frac{Cov(x, y)}{\sqrt{Var(x)Var(y)}}$$

## 8.11 Standard Correlation Coefficient Pearson's r

## 8.12 Convolution

## 8.13 Fourier Transform

## 8.14 Laplace Transform

## 8.15 Autocovariance

Autocovariance: Autocovariance is a special case of covariance where both variables are from the same time series, but they are at different time points. It measures the degree of linear dependence between values of a time series at different lags. In other words, it quantifies the relationship between a time series and a lagged version of itself.

$$ACov = (X_t, X_{t+k}) = \frac{1}{N} \sum_{t=1}^{N-k} (X_t - \bar{X})(X_{t+k} - \bar{X})$$

Where:

$X_t$  = value at time  $t$

$X_{t+k}$  = value at time  $t + k$ , where  $k$  is the lag (period)

$\bar{X}$  = average value

## 8.16 Gaussian Distribution

The “68-95-99.7” rule. Empirical distribution.

- ◇ **68% Rule:** Approximately 68% of the data falls within 1 standard deviation ( $\sigma$ ) from the mean ( $\mu$ ).
- ◇ **95% Rule:** Approximately 95% of the data falls within 2 standard deviations ( $2\sigma$ ) from the mean ( $\mu$ ).
- ◇ **99.7% Rule:** Approximately 99.7% of the data falls within 3 standard deviations ( $3\sigma$ ) from the mean ( $\mu$ ).

## 8.17 Time Series

time serie = collection of data points at constant time intervals

Time series properties:

- ◇ constant mean
- ◇ constant variance

◇ time-independant autocovariance

If time series is strictly stationary without dependencies amongst values, regular linear regression can be used to forecast values. If dependency amongst values exists, ARIMA (AutoRegressionIntegrated Moving Averages) is another choice of statistical model that can be used

## 9 Probability Appendix (Math 002)

### 9.1 Picking Probability

#### 9.1.1 Clusters

Probability to pick an item in a cluster is the same as picking that cluster, consider (A, B), (C, D), (E, F, G), then:

$$P(A) = P((A, B))$$

### 9.2 Martingale

Gains and losses accumulated by player during a game, concept of probability.

### 9.3 Bayes's Theorem

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

### 9.4 Laplace Smoothing

### 9.5 PDF (Probability Density Function)

## 10 Algebra Appendix (Math 003)

Mapping function = function

### 10.1 Matrices

#### 10.1.1 Sparse Matrix:

- Contains mostly zero values.
- Efficient for representing matrices with a large number of zero elements.
- Requires less memory compared to dense matrices.
- Suitable for sparse data structures and applications where memory usage is critical.

#### 10.1.2 Dense Matrix:

- Contains mostly non-zero values.
- Requires more memory compared to sparse matrices.
- Suitable for dense data structures and operations where all elements are significant.

### 10.2 Tensors

#### 10.2.1 Axes (rank)

Dimensions of a tensor are called axis, defined by key attributes.

### 10.2.2 Shape

### 10.2.3 Scalars (rank-0 tensors)

### 10.2.4 Vectors (rank-1 tensors)

$$\mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{pmatrix}$$

### 10.2.5 Matrices (rank-2 tensors)

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \\ a_{51} & a_{52} & a_{53} \end{pmatrix}$$

### 10.2.6 Higher rank (rank-3 and more)

$$\mathbf{A} = \left( \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \\ a_{51} & a_{52} & a_{53} \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \\ a_{51} & a_{52} & a_{53} \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \\ a_{51} & a_{52} & a_{53} \end{pmatrix} \right)$$

## 10.3 Useful Calculations

### 10.3.1 Harmonic Mean:

$$H = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \dots + \frac{1}{x_n}}$$

### 10.3.2 Distances

◇ Euclidean:

$$d(a, b) = \sqrt{\sum_{i=1}^n (a_i - b_i)^2}$$

◇ Manhattan:

$$\sum_{i=1}^n |x_i - y_i|$$

◇ Geodesic: shortest number of nodes between two points

**Lipschitz continuous convex function**

## 10.4 Boolean Algebra

### 10.4.1 Logical Operators

◇ AND operator:  $\wedge$

◇ OR operator:  $\vee$

◇ NOT operator:  $\neg$

◇ XOR (exclusive OR) operator:  $\oplus$

#### 10.4.2 Truth Tables

## 11 Analysis Appendix (Math 004)

### 11.1 Bellman equations

Used in dynamic programming, states that optimal policy (sequence of decisions) has the property that the decisions are independent of the initial state. In other words, if we have the solution to a small problem, we can find the solution to a larger problem.

$$V^*(s) = \max_a \left( R(s, a) + \gamma \sum_{s'} P(s'|s, a) V^*(s') \right)$$

Where:  $R(s, a)$  is the immediate reward obtained after taking action  $a$  in state  $s$ .  $P(s'|s, a)$  is the probability of transitioning to state  $s'$  given that action  $a$  is taken in state  $s$ .  $\gamma$  is the discount factor, representing the importance of future rewards relative to immediate rewards.  $\max_a$  denotes the maximum over all possible actions.