Machine Learning Cheat Sheet

Model Evaluation

- **Confusion_matrix**(y_test, y_pred, normalize = 'true') / Confusion_matrix(y_true, y_pred, normalize = 'pred')
- **Accuracy**: what proportion of data accurately predict. How close prediction are to true value.
  - (TP+TN)/(TP+TN+FN+FP) ⇔ *(y_train_5 == y_train_pred).mean()*
- **Precision**(Positive Predictive Value):out of retrieved elements, how many are actually positive. Used to limit FP. *Precision_score(y_test, y_ pred)*
- **Recall**(sensitivity / hit rate / true positive rate): ratio of positive instances correctly detected by classifier. Used to identify all positive samples and avoid false negatives.
- **F1 score**: harmonic mean of precision and recall, give more weights to low values. High when precision & recall is similar / high. *f1_score(y_test, y_ pred) / f1_score(y_test, pred, average="micro")* / *recall_score(y_test, y_ pred)*
- **Precision / Recall trade-off (curve) :** y_scores = sgd_clf.decision_function([some_digit]) *precision_score(y_test, y_ pred)* *y_probs = cross_val_predict(rnd_clf, X_train, y_train_5, cv=3, method='predict_proba')[:, 1]* *PrecisionRecallDisplay.from_predictions(y_test, y_prob)*
  Low threshold->increases recall, reduce precision. Higher threshold-> higher precision and lower recall.
- **ROC Curve**: true positive rate(recall) to false positive rate (1-specifity, fall out). **FPR ratio** = 1- TNR(specifity)
  - Specifity = TN / TN + FP, high specificity, it is good at identifying the negative cases
  - Straight line from (0,0) to (1,1) represents random guess. Higher recall(TPR), more false positive classifier produces. *RocCurveDisplay.from_predictions(y_test, y_proba)*
- **AUC Score**: rank most likely to least likely to be positive. 0.50 – random prediction / 1 – prefect prediction. Common for imbalanced data.
- Prefer the PR curve whenever the positive class is rare or when you care more about the false positives than the false negatives, and the ROC curve otherwise.
- **Classification Report:** number of samples in the class according to the ground truth.
  *Classification_report(y_test, y_ pred, target_names = ['not five','five]) / ConfusionMatrixDisplay.from_predictions(y_test, pred, normalize = "true"). if `'true'`, the confusion matrix is normalized over the true conditions (e.g. rows); if `'pred'`, the confusion matrix is normalized over the predicted conditions (e.g. columns); if `'all'`, the confusion matrix is normalized by the total number of samples; if `None` (default), the confusion matrix will not be normalized; / grid.best_score_*
  Macro avg: computes the average as an equal weight of each class (un-weighted per-class score - **treats each class equally**)
  Weighted avg: weighted average(mean of per-class score with weights) by support (**weights classes**)
  Micro: total FP/FN & TP over all classes, then compute precision, recall, f1-score using 3 counts(**treats sample/instance equally**)
- **Classification**: roc_auc, average_precision, f1, f1_macro, f1_weighted: binary f1 score, weighted variants
- **Regression**: r2, neg_mean_squared_error, neg_root_mean_squared_error, new_mean_absolute_error
- **Threshold based:** Accuracy, Precision, Recall F1 (balanced, equally important class); **Ranking Based**: Average precision, AUC (imbalanced, all possible threshold, models' ability to rank positive ones)

Neural Networks

- For linear regression: activation function, h(a) = a
- **Threshold Logic Unit / Linear Threshold Unit** - used for binary classification (like logistic regression and linear SVM)
  - Compute linear function(weighted sum) of inputs, $z = w^T x + b$, then apply
  - Activation function: step / Heaviside function to the sum, and output h(z) = 1, if z>=0 / 0, if z<0
- **Perceptron**: One or more TLU connected to input, dense layer. Perceptron is incapable of learning complex patterns
  - Output function: h(XW +b), X is input features(1 row per instance, 1 column per feature), W is weights(1 row per input, 1 column per neuron), b is bias vector(1 per neuron)
  - Learning Rule: $w_{i,j}$ (next step) = $w_{i,j} + n(y_j - y_j)\, x_{i.}$ Learning algorithm strongly resembles stochastic GD.
- Do not output class probability, make predication based on hard threshold.(LR over perceptron)
- **Single Layer Neural Network / multilayer perceptron** (input->hidden->output): Goal - learn model weights and biases.
- The connection weights between two neurons are increased whenever they have same output.
- Until the model is build, the layers will not have any weights. All weights must be initialized randomly.
- **Deep neural networks (2+ layers)** – Scaling is highly recommended
- Backpropagation: determine weights and bias terms through computations of gradients(derivative) in a repeated fashion to drop the neural network's error. Reverse mode autodiff
  - Forward pass making prediction(all results are preserved) -> measures network output error->contribution of each output connection to error(till input layer)->measures error gradient across all the connection weights in the network->computes GD step to tweak all connection weights in network.
- **Epoch**: Consists of forward pass for making prediction on existing params, backward pass for computing error gradients)

- Update weights in epoch i via gradient descent: $W^{i+1} = W^i - \varepsilon \nabla \mathrm{Loss}^i_W$, learning rate $\varepsilon>0$, Gradient $\nabla \mathrm{Loss}_W$
- **Activation Function**: Heaviside *(non-differentiable)* / Rectified Linear Unit (ReLU), h(a) = max(0,a) - *non-differentiable at 0, derivative 0 for a<0* / Sigmoid, $h(a) = 1 / (1+e^{-a})$ / Hyperbolic Tangent, $h(a) = e^a - e^{-a} / e^a + e^{-a}$
- **Regression**: one output neuron per output dimension**.** Prediction within range (singmoid[0,1]; tanh[-1,1])
- Eg. - To locate center of the image, you need 2D coordinates, so 2 output neurons.
  optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
  model.compile(loss="mse", optimizer=optimizer, metrics=["RootMeanSquaredError"])
  mse_test, rmse_test = model.evaluate(X_test, y_test)
  X_new = X_test[:3]
  y_pred = model.predict(X_new)

Hyperparameter	Typical Value
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem. Typically 1 to 5.
# neurons per hidden layer	Depends on the problem. Typically 10 to 100.
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None or ReLU/Softplus (if positive outputs) or Logistic/Tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

- **Classification -** Multiclass Classification: Output activation function encodes **softmax function** across all output nodes
  $f_m(X) = \Pr(Y = m \mid X) = \frac{e^{Z_m}}{\sum_{l=0}^{9} e^{Z_l}}$ z_m is raw output score; *softmax outputs a probability distribution*
- Fit the model by minimizing **cross entropy,** where y_{im} is 1 the true class for observation i is m, else 0(one-hot)
  $-\sum_{i=1}^{n}\sum_{m=0}^{9} y_{im}\log(f_m(x_i))$**;** y_{im} is binary indicator(0,1); One-hot encoding is used for the true class labels.

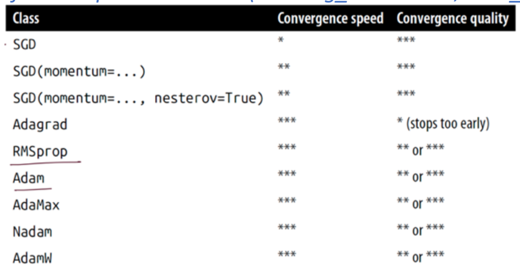
Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
# hidden layers	Typically 1 to 5 layers, depending on the task		
# output neurons	1	1 per binary label	1 per class
Output layer activation	Sigmoid	Sigmoid	Softmax
Loss function	X-entropy	X-entropy	X-entropy

- *callback = keras.callbacks.EarlyStopping(monitor='loss', patience=1)*
  history = model.fit(X_train, y_train, epochs = 50, validation_data = (X_valid, y_valid), callbacks = [callback]); history.params
  model.evaluate(X_test, y_test);   y_proba = model.predict(X_new)
- *categorical_crossentropy – one-hot / sparse_categorical_crossentropy – interger target*
  model.compile(loss = "categorical_crossentropy", optimizer = 'adam', metrics=['accuracy'])
  model.layers / model.get_layer / hidden1 = model.layers[1] / weights, biases = hidden1.get_weights() / history.params
  model.evaluate(X_test, y_test) / y_proba = model.predict(X_new)
- **Neural network tuning**:
  - Architecture: Layer arrangement, layer types; number of hidden layers, number of neurons per layer
  - Regularization: Batch Normalization, dropout; Optimizer: Number of epochs, learning rate, loss function
- **Issues:**
  - Slow training: vanishing(use non-saturating activation function) / exploding gradients(batch normalization)
  - Slow training: use other optimizers; Overfitting: dropout / early stopping
- **Vanishing gradients**: ReLU(z) = max(0,z) can help, but also try leaky ReLU(z) = max(αz, z), where α(derivative) >0, describes how much function leaks when z<0. e.g. sigmoid, when its value is close to 0 or 1
- **Exploding Gradients: Batch normalization**, add operation before / after activation function of each hidden layer to zero-center or normalize the output. Can allow increase learning rate(faster training) + regularize the model, may get faster convergence.
- Note- BatchNormalization before the activation function requires to create the activation function as separate layers after the dense layer and batch normalization layer. Having a mean close to 0 and a standard deviation close to 1.
  - Issue: adds complexity to the model, slower runtime per epoch
  - Adds 4 params: output scale(gamma), output offset(beta) vectors, final input means(moving_mean) and standard deviation(moving_var) vector. Allow the normalization process to maintain the representational capacity of the network.
- **Gradient Descent**: x+ = -learning_rate *dx, in opp. direction to minimize loss
- **Stochastic Gradient Descent with momentum**: adv – can roll past local minima; disadv – may roll past optimum and oscillate, 1 more hyperparameter to tune. V = mu *v -learning_rate * dx (integrate velocity) ; x+=v(integrate position)
- **Dropout**: ignore neuron with dropout rate p; typically, 10-50%; 20-30 for RNN; 40-50 for CNN
  - Scale remaining connection weights by 1/(1-p)(1/keep probability) to compensate dropout neurons
  - Randomly set input unit to 0 with frequency rate of each step during training time. Inputs not set to 0 are scaled up by 1/(1-rate) such that the sum over all the inputs is unchanged. Dropout is its own layer.

- Issue: may lead to slow convergence but lead to better model when tuned properly

- **Callbacks:** lets you specify the list of objects that keras will call during training at the start and the end of training, at start the end of each epoch, before and after processing each batch.
  - Save_best_only = True for model checkpoint
  - EarlyStopping will keep the track of the best weights and restore them at the end of training.

optimizer = tf.keras.optimizers.SGD(learning_rate=0.001) / optimizer = tf.keras.optimizers.SGD(learning_rate=0.001, momentum=0.9) / optimizer = tf.keras.optimizers.SGD(learning_rate=0.001, momentum=0.9, nesterov=True) / optimizer = tf.keras.optimizers.Adagrad(learning_rate=0.001) / optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9) / optimizer = tf.keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9,beta _2=0.999)



Class	Convergence speed	Convergence quality
SGD	*	***
SGD(momentum=...)	**	***
SGD(momentum=..., nesterov=True)	**	***
Adagrad	***	* (stops too early)
RMSprop	***	** or ***
Adam	***	** or ***
AdaMax	***	** or ***
Nadam	***	** or ***
AdamW	***	** or ***

Dimensionality Reduction

- **Curse of Dimensionality:** too many features – too long to train and makes it hard to find good solution. Training set is sparse.
- **Pros**: reduce overfitting, improve model efficiency, can remove correlated features and simpler data visualization. Image compression, finding pattern in high-dimensional data. Helps with modeling and reduce noise. Detect outliers
- **Cons**: PCA is linear, cannot work with polynomial. Can try kenelPCA. Need to find the dimension to keep. Less interpretable.
- PCA identifies the hyperplane that lies closest to the data and then project the data onto it. **Feature extraction** technique.
  - Selects the axis that amount for the maximum amount of variance, then second axis, orthogonal to the first that account for the largest amount of variance, so on. Summarizes the main insight for all the features.
  - Minimizes the mean squared distance between original dataset and the projection onto the dataset.
- PCA assumes the data is centered around the origin. Direction of the PCA is not stable. Scaling needed.
- By default, svd_solver is set to "auto": Scikit-Learn automatically uses the randomized PCA algorithm if m or n is greater than 500 and d is less than 80% of m or n, or else it uses the full SVD approach.
  rnd_pca = PCA(n_components=154, svd_solver="randomized"); Principal components are $Z1, \dots ZM$ representing $M < p$ linear combinations of our original p features. That is:
  $Zm = \phi_{1m}X_1 + \phi_{2m}X_2 + \dots + \phi_{pm}X_p$ for constants $\phi_{1m}, \phi_{2m}, \dots, \phi_{pm}$ and $m = 1, \dots, M$
  $\phi_{1m}, \phi_{2m}, \dots, \phi_{pm}$ are the weights (principal component loading vector) corresponding to each feature
- **Parameters**: n_components: features to keep, by default – min(n_samples, n_features)-1, all features are kept
  Attributes: components_, explained_variance_ and explained_variance_ratio_:amount/percentage of variance explained by each of the selected components *pca.components_.shape / pca.n_components_ / pca.components_.T[:, 0] #first component*
  cumsum = np.cumsum(pca.explained_variance_ratio_); varianceratio = pca.explained_variance_ratio_.sum();
  d = np.argmax(cumsum >= 0.95) + 1
- fit: finds the principal components, transform: applies rotation and dimensionality reduction
  pca = PCA(0.95); X_reduced = pca.fit_transform(X_train); X_recovered = pca.inverse_transform(X_reduced)

Clustering

- **Goals**: Data exploration(coherent group / number of groups) and unsupervised feature extraction. Create convex clusters.
- **Application**: customer segmentation; recommender system; search engines; semi-supervised learning; image segmentation; anomaly detection; data analysis; non-linear dimensionality reduction; feature engineering. Adv: fast and scalable.
- **Note**: Drop highly correlated attributes; can perform dimensionality reduction first then clustering. Standardize the data prior.
- **K-Means**: find centroids that minimizes the inertia. Variable can be categorical / numeric. Scaling needed(prevent stretching)

$$\min_{\substack{c_j\in\mathbb{R}^p, j=1\dots k \\ \text{closest point to } x_i}} \sum_{i=1}^{n} \underbrace{\left\| \overset{\text{every data point}}{x_i} - \underset{\text{centroid}}{c_{x_i}} \right\|^2}_{\text{Squared Euclidian Distance}}$$

- Hyperparameters:
  - n_clusters(default = 8)
  - init: {k-means++ / 'random'} (default(k-means++) *- can prevent converge to local optimum*
  - n_init (default = 10): run 10 times with different centroid seeds *- can prevent converge to local optimum*

- algorithm: {“lloyd”,”elkan”} default = “Lloyd”
- max_iter (default = 300)
  kmeans = KMeans(n_clusters = k, random_state = 42, n_init = 10) -> y_pred = kmeans.fit_predict(X), or
  good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]]). # When you know centroid
  kmeans = KMeans(n_clusters=5, init=good_init, n_init=1, random_state=42)
  Attributes: inertia is sum of squared distance of samples to their closest cluster center - *(X_dist[np.arange(len(X_dist)), kmeans.labels_] ** 2).sum() / kmeans.inertia_ / kmeans.score(X), will return negative inertia*
  *kmeans.labels_==y_pred / kmeans.cluster_centers_*
  X_dist = kmeans.transform(X). #measures distance from each instance to every centroid – soft clustering

- Identify n_clusters
  - **Inertia:** plot inertia to k and look for elbow, inflection point. (k increases inertia decreases)
  - **Silhouette Score:** mean silhouette coef. a(i) is intra mean cluster distance; b(i) mean nearest cluster distance
    silhouette_score(X, kmeans.labels_)

Silhouette Coefficient $$s_i = \frac{b(i)-a(i)}{\max(a(i),b(i))} = \begin{cases} 1-\frac{a(i)}{b(i)} & if\ a(i) < b(i) \\ 0 & if\ a(i) = b(i) \\ \frac{b(i)}{a(i)} - 1 & if\ a(i) > b(i) \end{cases}$$

-1 : wrong cluster
1 : inside right cluster
0 : decision boundary (close)
a<b, good solution

  - **Silhouette diagram:** shape height(num of instance in cluster), shape width (silhouette score coeff), vertical line (mean silhouette score for each number of clusters.
- **Limitations of K-means**: not well with cluster of varying size, different density or non-spherical. Need to specify number of cluster and run several times for suboptimal solution.
- **DBSCAN**: works well when cluster are separated by low-density regions. Does not require clusters number. Detects outliers.
- Hyperparameters:
  - Min_samples(default = 5): less dense region shall be outliers or have their own clusters
  - eps (default = 0.5): if too small, labeled -1 as “anomaly”) and if too large then 1 cluster
    dbscan = DBSCAN(eps = 0.2, min_samples=5)   #eps is most imp
    dbscan.fit(X) or clusters = dbscan.fit_predict(X)
    dbscan.labels_ (-1 for anamoly) / dbscan.core_sample_indices_(index of core) / dbscan.components_ (core instances)
    knn = KNeighborsClassifier(n_neighbors=50) , #input components, target labels of the core samples
    knn.fit(dbscan.components_, dbscan.labels_[dbscan.core_sample_indices_]) / knn.predict(X_new) / knn.predict_proba(X_new)
- If density varies significantly across clusters, it can be impossible for it to capture all the clusters properly. Scale less well for large dataset.

<u>Working with Text Data</u>

- **Bag of words**: discard input text structure and focus on how often each word appears in each text in corpus. Steps:
  - Tokenization: splitting document into words (tokens)
  - Vocabulary builder: numbered list of words appearing in the document
  - Encoding: for each document, count how often each vocab word appears.
- **Normalization(**reducing the vocab): Stemming and lemmatization
- **CountVectorizer**: Scikit-Learn implements Bag-of-Words in `CountVectorizer`. We call `.fit` on our text data and it will tokenize the data. *vect = CountVectorizer(stop_words='english', min_df=2, max_features=4)*
  *vect.fit(malory) / vect.get_feature_names_out() / vect.vocabulary_*
  bag_of_words = vect.transform(malory)   #get the updated text -> bag_of_words.toarray()
  vect.inverse_transform(bag_of_words)
  For classification: vect = CountVectorizer()
  X_train = vect.fit_transform(text_train)
  X_val = vect.transform(text_val) -> scores = cross_val_score(LogisticRegression(max_iter = 10000), X_train, y_train, cv=5)
- **Ngrams:** *cv = CountVectorizer(ngram_range=(1, 2)).fit(malory)*
- **TF-IDF:** Rescale features by how informative we expect them to be.

$$tfidf(t,d) = tf(t,d) \times idf(t)$$

$$idf(t) = \log\left(\frac{1+n_d}{1+df(d,t)}\right) + 1$$

If a word appears in every document, then idf = 1 (it is not important)

- **TfidfTransformer:** takes sparse matrix output from CountVectorizer and transforms it
  malory_tfidf = make_pipeline(CountVectorizer(), TfidfTransformer()).fit_transform(malory) / malory_tfidf.toarray()
- **TfidfVectorizer:** takes text data->bag of words->tf-idf transformation
  malory_tfidf = TfidfVectorizer().fit_transform(malory) / malory_tfidf.toarray()
  - After tf-idf representation L2 norm is applied. Each row is normalized to Euclidian norm.
  - Length of document doesn’t change the vectorized representation.