Python Functions

01. Linear Regression

The **zscore** function operates a standardization of its inputs:

```
from scipy.stats import zscore
x = zscore(dataset['petal-length'].values).reshape(-1, 1) # we reshape our
feature column as a (n_sample, n_features) matrix

y = zscore(dataset['petal-width'].values)
```

Let's use **scikit-learn** tools to do a linear regression:

```
from sklearn import linear_model
lin_model = linear_model.LinearRegression()

lin_model.fit(x, y)
```

To evaluate the quality of our regression we can analyse some metrics:

- RSS: it tells us how much of the prediction differs from the true value
- Coefficient of determination: it tells us how the fraction of the variance of the data explained by the model (how much better we are doing w.r.t. just using the mean of the target
- Mean Squared Error: it tells approximately how much error we get on a predicted data over the training set (i.e., a normalized version of the RSS)

```
from sklearn.metrics import mean_squared_error, r2_score

from sklearn.feature_selection import f_regression
# RSS
lin_model._residues
# Coefficient of determination
r2_score(y, y_pred)
# Mean Squared Error
mean_squared_error(y, y_pred)
f_regression(x, y) # it outputs a tuple: (value of the F-statistics, its p-
value)

# If one wants all the information about the output of a linear model in a
single instruction, just use the library statsmodels and use the function
summary() on the result of the Ordinary Least Square optimization procedure
```

```
from statsmodels import api as sm

lin_model2 = sm.OLS(y, x).fit()
print(lin_model2.summary())
lin_model2._results.params
lin_model2._results.k_constant
```

Custom Implementation

We can also implement Least-Squares from scratch, using its closed-form: $\hat{w}_{OLS} = (\Phi^T\Phi)^{-1}\Phi^T t$

```
from numpy.linalg import inv

n_samples = len(x)
Phi = np.ones((n_samples, 2))
Phi[:, 1] = x.flatten() # the second column is the feature
# the field 'T' represents the transposed matrix, @ is the matrix product,
the method 'dot' is the matrix product
w = inv(Phi.T @ Phi) @ (Phi.T.dot(y))
```

Regularization

If we need to mitigate over-fitting effects in a model we might resort to some regularization techniques, like Ridge regression or Lasso regression.

- Ridge Regression: Linear least squares with l2 regularization.

```
ridge_model = linear_model.Ridge(alpha=10)
ridge_model.fit(x, y)
```

- Lasso Regression: Linear Model trained with L1 prior as regularizer

```
lasso_model = linear_model.Lasso(alpha=10)
lasso_model.fit(x, y)
mean_squared_error(y, lasso_model.predict(x))
```

02.Classification

From lesson we learned that 3 approaches are possible for classification:

- **Discriminant function approach**:
  - model a *function* that maps inputs to classes

    - fit model to data
- **Probabilistic discriminative approach**:
    - model a *conditional probability* $P(C_k|x)$
    - fit model to data
- **Probabilistic generative approach**:
    - model *likelihood* $P(x|C_k)$ and *prior* $P(C_k)$
    - fit models to data
    - infer posterior $P(Ck|x) = \frac{P(C_k)P(x|C_k)}{P(x)}$

Preparing the dataset for classification

Let us start with discriminating between Setosa and non-Setosa flowers according to the sepal length and width.

```
from scipy.stats import zscore
from sklearn.utils import shuffle

X = zscore(dataset[['sepal-length', 'sepal-width']].values)
t = dataset['class'].values == 'Iris-setosa'
X, t = shuffle(X, t, random_state=0) # this time we have to do it!
setosa = X[t]
not_setosa = X[~t]
```

Discriminant Function Approach: the Perceptron

At first, let us perform a classification with a perceptron classifier:

- Hypothesis space: $y(x_n) = sgn(w^T x_n) = sgn(w_0 + x_{n1}w_1 + x_{n2}w_2)$;
- Loss measure: Distance of misclassified points from the separating surface $L_P(w) = -\sum_{n \in \mathcal{M}} w^T x_n C_n$;
- Optimization method: Online Gradient Descent;
  where sgn(·) is the sign function.

```
from sklearn.linear_model import Perceptron
perc_classifier = Perceptron(shuffle=False, random_state=0)
perc_classifier.fit(X, t)
```

Evaluating Classification

To evaluate the performances of the chosen method, we need to compute the *confusion matrix* which tells us the number of points which have been correctly classified and those which have been misclassified.
Based on this matrix we can evaluate:

- Accuracy: $Acc = \frac{tp+tn}{N}$ fraction of the samples correctly classified in the dataset;
- Precision $Pre = \frac{tp}{tp+fp}$ fraction of samples correctly classified in the positive class among the ones classified in the positive class;
- Recall: $Rec = \frac{tp}{tp+fn}$ fraction of samples correctly classified in the positive class among the ones belonging to the positive class;
- F1 score: $F1 = \frac{2 \cdot Pre \cdot Rec}{Pre+Rec}$ harmonic mean of the precision and recall;
  where tn is the number of true negatives, fp is the number of false positives, fn are the false negatives and tn are the true negatives.
  Remember that:
- The higher these figures of merits the better the algorithm is performing.
- These performance measures are **not** symmetric, but depends on the class we selected as positive.
- Depending on the **application** one might switch the classes to have measures which better evaluate the predictive power of the classifier.

```
from sklearn.metrics import confusion_matrix, accuracy_score,
precision_score, recall_score, f1_score
t_pred = perc_classifier.predict(X)
confusion_matrix(t, t_pred)
confusion_matrix(t, t_pred)
accuracy_score(t, t_pred)
precision_score(t, t_pred)
recall_score(t, t_pred)
f1_score(t, t_pred)
```

Implementing the Perceptron classifier

```
w = np.ones(3)
n_epochs = 10
for epoch in range(n_epochs):
        for i, (x_i, t_i) in enumerate(zip(X, t)):
        # correct t_i to be in {-1, 1}
                corr_t_i = 1 if t_i else -1
                ext_x = np.concatenate([np.ones(1), x_i.flatten()])
                if np.sign(w.dot(ext_x)) != corr_t_i:
                        w = w + ext_x * corr_t_i
```

Notice that this procedure will stop if the classes are linearly separable, while it does not stop if the two classes are overlapping.
Moreover, we do not know how long the procedure will take to reach convergence.
This makes impossible to distinguish between a procedure which is *slowly converging* to a *non-linearly separable* setting.

Probabilistic Discriminative Approach: Logistic Regression

Let us change the methods for the classification task and use a Logistic regression classifier with two classes:

- Hypothesis space: $y_n = y(x_n) = \sigma(w_0 + x_{n1}w_1 + x_{n2}w_2)$;
- Loss measure: Loglikelihood $L(w) = -\sum_{n=1}^{N}[C_n ln y_n + (1 - C_n) ln(1 - y_n)]$;
- Optimization method: Gradient Descent;
  where the sigmoid function is defined as $\sigma(x) = \frac{1}{1+e^{-x}}$.

```
from sklearn.linear_model import LogisticRegression

log_classifier = LogisticRegression(penalty=None) # regularization is applied
as default
log_classifier.fit(X, t)
```

If we perform the $logit(x) = log(\frac{x}{1-x})$ transformation to the output we have:

$$logit(y_n) = w_0 + x_{n1}w_1 + x_{n2}w_2$$

and, thus, we have the same statistical characterization of the parameters w as we had in the linear regression if we consider as output a specific transformation of the target, i.e., we can perform hypothesis testing on the significance of the parameters.

Multiple Classes

```
multi_t = dataset['class']
multi_log_classifier = LogisticRegression()
multi_log_classifier.fit(X, multi_t)
```

Probabilistic Generative Approach: Naive Bayes

Generative models have the purpose of modeling the joint pdf of the couple input/output $p(C_k, x)$, which allows us to generate also **new data** from what we learned.
This is different from the probabilistic discriminative models, in which we are only interested in computing the probabilities that a given input is coming from a specific class $p(C_k|x)$, which is not sufficient to produce new samples.
Conversely, we will see how it is possible to generate new samples if we are provided with an approximation of the joint input/output distribution $p(C_k, x)$.
In this case, the Naive Bayes method considers the **naive assumption** that each input is conditionally (w.r.t. the class) independent from each other. If we consider the Bayes formula we have:

$$p(C_k|x) = \frac{p(C_k)p(x|C_k)}{p(x)} = p(C_k)\prod_{j=1}^{M} p(x_j|C_k)$$

The decision function, which maximises the Maximum A Posteriori probability, is the following:

$$y(x) = \underset{k}{argmax}\, p(C_k) \prod_{j=1}^{M} p(x_j|C_k)$$

where as usual we do not consider the normalization factor p(x).
In a specific case we have to define a prior distribution for the classes $p(C_k)$ ∀k and a distribution to compute the likelihood of the considered samples p(x_j|C_k) ∀J, ∀k.

In the case of continuous variable one of the usual assumption is to use Gaussian distributions for each variable $p(x_j|C_k) = \mathcal{N}(x_j; \mu_{jk}, \sigma^2_{jk})$ and either a uniform prior $p(C_k) = \frac{1}{K}$ or a multinomial prior based on the samples proportions $p(C_k) = \frac{\sum_{i=1}^{N} I\{x_n \in C_k\}}{N}$, where I{·} is the indicator function.
The complete model of Naive Bayes is:

- Hypothesis space: $y_n = y(x_n) = \underset{k}{argmax}\, p(C_k) \prod_{j=1}^{M} p(x_j|C_k)$;
- Loss measure: Log likelihood;
- Optimization method: MLE.

```
from sklearn.naive_bayes import GaussianNB

gnb_classifier = GaussianNB()
gnb_classifier.fit(X, t)
t_pred = gnb_classifier.predict(X)
```

Generating new data

Using the estimated priors $p(C_k)$ and likelihoods $p(x_j|C_k) = \mathcal{N}(x_j; \mu_{jk}, \sigma^2_{jk})$ it is possible to generate new data.

```
N = 100
new_samples = np.empty((N, 2))
new_t = np.empty(N, dtype=bool)

for i in range(N):
        # Based on the class priors, we sample a class
        class_ = np.random.choice([0,1], p=gnb_classifier.class_prior_)
        new_t[i] = class_

        # For each feature, we have a normal distribution of its likelihood
given the class
        # theta: mean of each feature per class (n_classes, n_features)
        thetas = gnb_classifier.theta_[class_, :]

        # sigma: standard deviarion of each feature per class (n_classes,
n_features)
        sigmas = np.sqrt(gnb_classifier.var_[class_, :])

        # sample x1
```

```
        new_samples[i,0] = np.random.normal(thetas[0], sigmas[0], 1)
        # sample x2
        new_samples[i,1] = np.random.normal(thetas[1], sigmas[1], 1)

# divide samples by class
new_setosa = new_samples[new_t, :]
new_not_setosa = new_samples[~new_t, :]
```

Notice that the Naive Bayes is **not** a Bayesian method.
Indeed, the priors we compute are estimated from data, and not updated using likelihoods.
This makes Naive Bayes a method which uses the Bayes theorem to model the independence among the input, given the classes.

03. Bias-Variance Tradeoff