INF8245E (Fall 2025) : Machine Learning — Given on: Sep 26^{th}

Assignment #2 — Due on : Oct 26^{th}, 10 pm

- This is an individual assignment. You are not allowed to discuss the problems with other students.
- Make sure to write your code only in the .py files provided. Avoid creating new files. Do not rename the files or functions as it will interfere with the autograder's ability to evaluate your work correctly. Also, do not change the input or output structure of the functions.
- When Submitting to GradeScope, be sure to submit
  1. A '.zip' file containing all your python codes (in .py format) to the 'Assignment 2 - Code' section on Gradescope. Do not submit data, environment files, or any other files.
  2. A 'pdf' file that contains your answers to the questions to the 'Assignment 2 - Report' entry.
- Part of this assignment will be auto-graded by Gradescope. You can use it as immediate feedback to improve your answers. You can resubmit as many times as you want. We provide some tests that you can use to check that your code will be executed properly on Gradescope. These are **not** meant to check the correctness of your answers. We encourage you to write your own tests for this.
- Please provide the title and the labels of the axes for all the figures your are plotting.
- If you have questions regarding the assignment, you can ask for clarifications in Piazza. You should use the corresponding tag for this assignment.
- Before starting the assignment, make sure that you have downloaded all the data and tests related for the assignment and put them in the appropriate locations.
- **Warning:** Throughout the assignment, you will be asked to implement certain algorithms and find optimal values. In the solution you submit, do not simply call a library function which performs the entire algorithm for you, this is forbidden, as it would obviously defeat the purpose. For example, if you were asked to implement logistic regression, or KNN, do not simply call an outside package (such as scikit-learn) for help.
- You cannot use ChatGPT or any other code assistance tool for the programming part, however if you use ChatGPT to edit grammar in your report, you have to explicitly state it in the report.

In this assignment, you will implement three classification algorithms and test them on several datasets: the standard hand-written digit classification dataset, MNIST, which has 10 classes, a credit card fraud dataset, with 2 target classes and a much smaller dataset about iris flowers with 3 different species.

You will be implementing K-Nearest Neighbors, Gaussian Naive Bayes and Logistic Regression, along with answering theoretical questions about all three algorithms. The code that you will have to complete, and that will download the datasets is provided alongside this assignment.

K-Nearest Neighbors (55 Points)

Implementation (30 points)

In this first section, you will implement K-Nearest Neighbors. Question 1 deals with basic functions needed to perform classifications with KNN. In Question 2, you will use these functions to run the KNN algorithm and evaluate its performance on a withheld test split. You will be expected to discuss this performance based on the only hyper-parameter, k. Question 3 will make you implement a smarter data-structure for identifying the nearest neighbors of a point. You will analyze how both your implementations scale as data increases.

1. (10 points) KNN utility functions implementation.
   (a) (2 Points) Run the `data_download.py` script. This will fetch the data online for the whole assignment and decompress it.

   Implement the function `preprocess_mnist_data` in `data_process`. The expected signature of the function is indicated in the file. The function should:

   - take a path to a pair of csv files as input (train and test).
   - open them using pandas (careful, there is no header in the csv file).
   - extract the first column for both, which are the classes for each row.
   - perform scaling on the remaining columns (in order pixel (1, 1), pixel (1, 2), etc...the images are already vectorized) by removing the dataset-wide mean pixel value and dividing by the dataset-wide pixel standard deviation. These values should only be computed on the train split.
   - return as `numpy` arrays: the training features, the training classes, the test features, the test classes, the mean and standard deviation you computed.

   If you wish to visualize the data, we provide a `visualize_image` function in `utils.py`, taking in a (784,) shaped array and plotting the associated image, potentially saving it.

   (b) (2 points) Using `scikit-learn`, implement `euclidean_distance` and `cosine_distance`.

   (c) (3 points) Implement the `get_k_nearest_neighbors` function that retrieves the k-nearest neighbors labesl from a training set.

   (d) (3 points) Implement the `majority_voting` function which finds the most frequently occurring label from the k-nearest neighbors.

2. Using the utility functions you have implemented, we put together a `knn_classifier` function.

(a) (7 point)

- We don't have a validation set with our MNIST dataset. The train split is already randomized, select the last 10000 rows of the train set as a validation set.
- Using this validation set, run the classifier above with different values of $k \in \{1, 2, 3, 4, 5, 10, 20\}$, for the euclidean distance.
- In your report, write the different accuracies you get on the **validation set** by executing the KNN algorithm on different values of k. This is an hyper-parameter search. Comment on the trend you observe.
- Report the best value of k, and, for this value, compute the accuracy on the **test set**.

(b) (3 points) Answer the same question with the cosine distance.

3. (10 points) This initial implementation of the KNN algorithm is fairly fast, mainly because we are using optimized distance computations, and the dataset is small.

   What would happen with a dataset with one million of rows? Even for a modest 10000 test set, this would require 40GB of RAM (assuming float 32 precision), just to store all the pairwise distances and it would take very long to compute this matrix. In fact, the time complexity of the implementation above is $\mathcal{O}(N_{\text{train}}N_{\text{test}}d + N_{\text{train}}\log(N_{\text{train}})N_{\text{test}})$ and the space complexity is $\mathcal{O}(N_{\text{train}}N_{\text{test}})$. Can we do better and how?

   This question is exactly about that. We are going to be implementing a smarter data structure, called k-d trees, that allows us to significantly reduce the number of distance computations.

   The name might be confusing but k-d tree means k dimensional tree (and the k in question is unrelated to our k nearest hyper-parameter).

   For more information about k-d trees, here are lecture notes from the University of Maryland and a video from Computerphile.

   (a) (1 points) If you ran the `data_download.py` script, you should also have in your data folder a `credit_card_fraud_train.csv` file.

   Implement the function `preprocess_credit_card` in `data_process.py`. This function should be very similar to the `preprocess_mnist_data` function except that we have a header, the classes column has name `Class` and you can discard the colum:

   - Open the pair of csv files as input using pandas, keeping the header and using column id as our index.
   - extract the column Class for both.
   - perform scaling *column wise*: substract, for each column, its mean and divide by its standard deviation, as computed on the train set.
   - return the result as `numpy` arrays.

(b) (4 points) Creating the k-d data structure. k-d trees are binary trees, whose nodes (or leaves) represent points in a multi-dimensional point cloud. A k-d tree recursively splits the point cloud into two equal parts, at the median of coordinate along a certain dimension. In `knn_kdtree.py`:

- We have implemented the `KDNode` dataclass. This class should represent a single node in the stree and store the data point's features and labels, the splitting dimension index for this node and reference to it's left and right children.
- Complete the `KDTree` class, which contains a root node and a `_build_tree` method.
- The `_build_tree` method recursively builds the tree, selecting the splitting dimensions in increasing order $0, 1, 2, \ldots$. It finds the median along the selected dimension and the point corresponding to this median is selected as the node. The points less than the median will go in the left child's sub-tree, and the points greater or equal than the median in the right sub-tree. The recursion stops, for a branch, when a node contains only one data point and has no children.

In your report, discuss the time and space complexity of building the tree.

(c) (4 points) Implement the `_find_nearest` method in the `KDTree` class. This method will perform a recursive search for the nearest neighbor to a given query point. The search should:

- Start from the root and traverse down the tree, moving left or right based on the query point's value in the current splitting dimension.
- When the search reaches a leaf node, compare the distance of the leaf's point to the query point and update the current best guess for the nearest neighbor.
- As the search unwinds (recursively returns up the tree), check if the current `best_distance` is greater than the distance to the splitting plane. If it is, the nearest neighbor could potentially exist in the other branch of the tree, and you must recursively search that branch as well. This is the critical step that ensures correctness.
- Return the closest point and its label.

In your report, discuss the time complexity of traversing the tree. Explain why it is needed to store the splitting dimension in `KDNode`.

(d) (1 points) Putting it all together for 1-NN. Implement the `kdtree_1nn_predict` function that takes as input a "trained" 1-NN tree and a test array and predicts, for each element the target class.

Report the final accuracy you get on the credit card fraud dataset.

Note: You might notice than the runtime is very large, suggesting that our implementation is not optimized. If you were to compare that with some optimized python optimization, like scikit-learn `KNNeighborsClassifier`, the second option would be significantly faster. They parallelize operations, both when building the tree and when searching through it.

(e) **Bonus question (3 points). Your total is capped to 100 for the whole assignment.** Explain, using pseudo code if you need, how you would use a k-d tree to identify the k nearest neighbors, for $k > 1$. Discuss the complexity.

Theory: a special case of Stone's theorem (25 points)

In this second section, we will study theoretically the performance of the KNN algorithm as the number of data points available to us scales to infinity. We will prove a simplified version of Stone's theorem, which, in our case, will state that if $k \longrightarrow +\infty$ and $k/N_{\text{train}} \longrightarrow 0$ then k-NN becomes consistent, i.e. we get optimal performance. Question 4 summarizes a few simple decision theory / empirical risk minimization results, giving a meaning to "optimal performance" and setting up the rest of the section. Question 5 finishes the proof using inequality computations.

Notations: we restrict ourself to binary classification and assume features live in $\mathbb{R}^d$ for some positive integer d. The data is assumed to be i.i.d (independent, identically distributed) for some distribution ν. We also assume we have access to an infinite amount of samples from ν: $(X_i, g_i)_{i\in\mathbb{N}}$ where $X_i \in \mathbb{R}^d$ are called *features* and $g_i \in \{0, 1\}$ are the *labels*. Each label belong to class (here $\{0, 1\}$) .

The general goal of risk minimization is to build a classifier $G : \mathbb{R}^d \to \{0, 1\}$ which minimizes the probability of classifying an example wrong:

$$P_{X,g\sim\nu}(G(X) \neq g) =: R(G) = \mathbb{E}_{X,g\sim\nu}\left[\mathbf{1}_{G(X)\neq g}\right]$$

Where $\mathbf{1}_A$ is the indicator function for A, a function worth 1 if A is true and 0 if it's false. In this section, we will build $k(n)$-NN estimators $\hat{G}_n$ "trained" on $(X_i, g_i)_{i\in[|1,n|]}$ for some value $k(n)$ and we are interested in the behavior of $R(\hat{G}_n)$

4. (a) (2 points) We denote $\eta(x) = P(g = 1|X = x)$. Show that for any classifier G:

$$R(G) = \mathbb{E}\left[\eta(X)\mathbf{1}_{G(X)=0} + (1 - \eta(X))\mathbf{1}_{G(X)=1}\right]$$

(b) (5 points) We define the Bayes classifier as

$$G^*(x) = \begin{cases} 1 & \text{if } \eta(x) \geq 1/2 \\ 0 & \text{elsewhere.} \end{cases}$$

Show that

$$R^* := R(G^*) = \mathbb{E}\left[\min(\eta(X), 1 - \eta(X)\right]$$

and that for any classifier G:

$$R(G) - R^* = \mathbb{E}\left[|2\eta(X) - 1|\mathbf{1}_{G(X)\neq G^*(X)}\right]$$

What can we say about G^* compared to any other classifier, is G^* computable in practice?

We say a sequence of estimators $\hat{G}_n$ is consistent if:

$$\mathbb{E}_{(X_i,g_i)_{i\in[|1,n|]}}[R(\hat{G}_n)] \to R^*$$

Note that the expected value here is taken over the data on which $\hat{G}_n$ is built, and R contains itself an expected value, but over the whole data distribution.

(c) (3 points) Because we don't have access to η, a good way of building a classifier would be to approximate η via some estimator $\hat{\eta}_n$ and then plug in $\hat{\eta}_n$ in the Bayes classifier decision rule, instead of η. In practice, this is what most, if not all machine learning methods do. Show that if we define $\hat{G}_n$ using some approximated probability function $\hat{\eta}_n$:

$$R(\hat{G}_n) - R^* \leq 2\mathbb{E}[|\eta(X) - \hat{\eta}_n(X)|]$$

Hint: Starting from the equality in question 4.(a), $G(X) \neq G^*(X)$ is completely equivalent to $\eta(X)$ and $\hat{\eta}_n(X)$ being on "opposite sides" of $1/2$.

5. For some data $D_n = (X_i, g_i)_{i\in[|1,n|]}$, we can rewrite the k-NN estimator of $\eta(x)$ as:

$$\hat{\eta}_n(x) = \sum_{i=1}^{n} w_{n,i}(x) g_i$$

where the weights $w_{n,i}(x)$ are dependent on D_n and sum to 1:

$$w_{n,i}(x) = \begin{cases} 1/k & \text{if } \mathrm{X_i} \text{ is a k nearest of x} \\ 0 & \text{elsewhere.} \end{cases}$$

Let's also introduce a (purely theoretical) "oracle" k-NN estimator $\tilde{\eta}$, which is purely theoretical and "corrects" errors in $\hat{\eta}$ due to the randomness of $g|X$:

$$\tilde{\eta}_n(x) = \sum_{i=1}^{n} w_{n,i}(x) \eta(X_i)$$

.

Note that both $\hat{\eta}_n$ **and** $\tilde{\eta}_n$ **are random functions** dependent, for $\hat{\eta}_n$ on draws of both the features (X_i) and the labels (g_i) and for $\tilde{\eta}_n$, just the features. Expected values are taken over both X and the (X_i, g_i) and the subscript of the expected value symbol is implied based on context.

(a) (2 points) Show that

$$\mathbb{E}\left[(\eta(x) - \hat{\eta}_n(X))^2\right] \leq 2\mathbb{E}\left[(\eta(x) - \tilde{\eta}_n(X))^2\right] + 2\mathbb{E}\left[(\tilde{\eta}_n(x) - \hat{\eta}_n(X))^2\right]$$

(b) (5 points) Let's assume now that $k(n) \to \infty$ and $k(n)/n \to 0$. Let $a > 0$ be a fixed positive radius. We will assume that

$$\mathbb{E}\left[\sum_{i=1}^{n} w_{n,i}(X)\mathbf{1}_{\|X_i - X\| \geq a}\right] \longrightarrow 0.$$

This essentially tells us that for any ball of radius a, asymptotically, the points outside this ball don't influence the decision for the center of that ball. Let $\epsilon > 0$. Writing $\left[(\eta(x) - \tilde{\eta}_n(X))^2\right]$ as a single sum and using Jensen's inequality, the absolute continuity of η and the assumption above, show that, for some constant $\delta > 0$:

$$\mathbb{E}\left[(\eta(x) - \tilde{\eta}_n(X))^2\right] \leq \epsilon + \mathbb{E}\left[\sum_{i=1}^{n} w_{n,i}(X)\mathbf{1}_{\|X_i - X\| \geq \delta}\right]$$

What can you conclude about this first term?

(c) (5 points) Let's now focus on the second term $\mathbb{E}\left[(\tilde{\eta}(x) - \hat{\eta}_n(X))^2\right]$. Show that:

$$\mathbb{E}\left[(\tilde{\eta}(x) - \hat{\eta}_n(X))^2\right] = \mathbb{E}\left[\sum_{i=1}^{n} w_{n,i}^2(X)(g_i - \eta(X_i))^2\right]$$

Then show that

$$\mathbb{E}\left[(\tilde{\eta}(x) - \hat{\eta}_n(X))^2\right] \leq \mathbb{E}\left[\max_i w_{n,i}(X)\right]$$

What can you say about the second term?

(d) (3 points) Conclude the proof for consistency.

6. **Bonus question (5 points, your total is capped to 100 points for the whole assignment).** Show the main result we assumed for kNN.

For$k(n) \to \infty$ and $k(n)/n \to 0$. Let $a > 0$ be a fixed positive radius. Show that:

$$\mathbb{E}\left[\sum_{i=1}^{n} w_{n,i}(X)\mathbf{1}_{\|X_i - X\| \geq a}\right] \longrightarrow 0.$$

Logistic Regression (20 Points)

In this section, you will implement a logistic regression classifier for the credit card fraud datasets. A logistic regression model assumes that the log likelihoods of all the classes are affine in the features, for some matrix W and bias vector b:

$$\log P(g = k|X) = W_k^\top X + b_k - C$$

for some regularization constant C that does not depend on k (but depends on X), such that the probabilities sum to 1.

The probabilities for each class can then be computed using a softmax function:

$$P(g = k|X) = \frac{\exp(W_k^\top X + b_k)}{\sum_{j=1}^{K} \exp(W_j^\top X + b_j)}$$

We train such a model by maximizing the likelihood of our data under the model. In this case, it's equivalent to minimizing the cross entropy, also knwon as log loss between the predicted probabilities and the true labels:

$$\mathcal{L}(W, b) = -\frac{1}{N} \sum_{i=1}^{N} \log P(g_i|X_i)$$

The optimization will be performed using gradient descent, a method you implemented in assignment 1 and will not be re-implemented here (we provide you with a working version).

1. (a) (5 points) Derive the gradient of $\mathcal{L}$ with respect to W and b.
   (b) (2 point) Implement the `forward_probabilities` function that computes the probabilities of each class for each element of a feature array, given a pair (W, b).
   (c) (3 point) Implement the `logistic_regression_loss` function that computes the loss $\mathcal{L}$ for a given pair (W, b) and a dataset. In our implementation, we add an optional regularization term on W. This line is already completed in the code.
   (d) (4 points) Implement the `logistic_regression_grad` function that computes the gradient of $\mathcal{L}$ with respect to W and b. The corresponding gradient correction is also included in the provided code.
   (e) (6 points)Run the function `logistic_regression_fit` we provide you with for different values of the regularization parameter $\lambda \in \{0, 0.01, 0.1, 1, 10\}$. What is the role of this parameter? Is it necessary in this case? Our function performs inference on the validation set every 10 iterations. Plot the evolution of the accuracy on the validation set as a function of the number of iterations for different values of λ. Picking the best λ, report the accuracy on the test set of the credit card fraud dataset.

Gaussian Naive Bayes (20 points)

In this last section, you will implement a Gaussian Naive Bayes (GNB) classifier for the MNIST and Iris datasets. A GNB model assumes that class conditional densities correspond to a multivariate Gaussian distribution:

$$P(X|g=k) = \mathcal{N}(X|\mu_k, \Sigma_k)$$

for a pair of class specific mean and covariance μ_k, Σ_k. The *naive* adjective comes from the assumption that class conditioned features are independent from one another. This is equivalent to having all Σ_k diagonal. Given such a prior, one wants to compute the posterior distribution over classes: $P(g=k|X)$. We assume that we have access to a dataset of i.i.d samples: $(X_i, g_i)_{i\in\{1,\dots,N\}}$

1. (a) (1 point)Using Bayes theorem show that $\log P(g=k|X)$ can be written as:

$$\log P(g=k|X) = \log P(X|g=k) + \log P(g=k) - C(X)$$

. Where C is a constant only dependent on X, not k. Do we need to compute this constant? What parameters are to be "trained" or computed in a GNB model

   (b) (4 points) You will now derive the formulas for all $(\mu_k)_{k\in\{1,\dots,K\}}$ and $(\Sigma_k)_{k\in\{1,\dots,K\}}$. We will find the values for μ_k, Σ_k that maximize the likelihood of our dataset, i.e we want our data to be as likely as possible under model we are building, a central idea in machine learning.

   The likelihood of our data is $\mathcal{L} = P((X_i, g_i)_{i\in\{1,\dots,N\}}$ and this is equal to $\mathcal{L} = \prod_{i=1}^{N} P((X_i, g_i)) = \prod_{k=1}^{K}\prod_{i|g_i=k} P(X_i|g_i = k)$. This allows us to split the optimization problem in k independent problems. In practice, we are also going to be optimizing $\log \mathcal{L}$, which is easier.

   Derive the class dependent optimization problem for μ_k, Σ_k, assuming that the Σ_k are diagonal $\Sigma_k = \text{diag}(\sigma_1^{(k)} \dots \sigma_d^{(k)})$ and, via differentiation, find the maximum.

   (c) (4 points) Some pixels in MNIST are constant (such as always at 0). These dimensions are redundant and do not offer any information to the model. We could discard them. To avoid any numerical issues when dividing by the standard deviation for a pixel, regularize any standard deviation with $\sigma_{reg}^2 = \sigma^2 + \varepsilon$ with $\varepsilon = 10^{-3}$ when implementing the function `gnb_fit_model` that fits the GNB model on the training data

   (d) (4 points) Implement the `gnb_predict` function that takes as input a fitted GNB model and run predictions on a test set.

   (e) (7 points) Run GNB on both MNIST and the iris dataset test splits. Report the accuracy for both. Also report the error rate over MNIST test split for each one of the 10 classes.

   Why do you think the accuracy for the 1 class is so high? In your opinion, why does GNB works so well for the irises and so poorly on MNIST compared to KNN? To help you, and illustrate your answer, you can, and should visualize the iris dataset.