LabML 1: Implementing a Machine Learning Model from Scratch

DNA program 2022-2023: course "Tools for ML"

Commentary: This lab is based on the assignments of the CPSC 340 course at UBC. Please refer to `https://www.cs.ubc.ca/~fwood/CS340/`. To succeed in the lab, you will need to know:

- Basic Python programming, including NumPy and plotting with matplotlib.
- Statistics, algorithms and data structures to the level of the course prerequisites.
- Some basic LaTeX and git skills so that you can typeset equations and submit your lab.

A note on the provided code: in the `code` directory we provide you with a file called `main.py`. This file, when run with different arguments, runs the code for different parts of the assignment. For example,

```
python main.py -q 3.2
```

runs the code for Question 3.2. At present, this should do nothing (throws a `NotImplementedError`), because the code for Question 3.2 still needs to be written (by you). But we do provide some of the bits and pieces to save you time, so that you can focus on the machine learning aspects. For example, you'll see that the provided code already loads the datasets for you. The file `utils.py` contains some helper functions. You don't need to modify the code in there. To complete your assignment, you will need to modify `main.py`, `main.py`, and other files like `decision_stump_generic.py`, `decision_stump_error.py` and file `decision_stump_info.py` (which you would need to complete).

Instructions

We use blue to highlight the deliverables that you must answer/do/submit with the lab. Please, commit your code to your github account and provide the repository link by email.

1 Algorithms and Data Structures Review

1.1 Trees

Answer the following questions in `main.py`. You do not need to show your work.

1. What is the minimum depth of a binary tree with 64 leaf nodes?
2. What is the minimum depth of binary tree with 64 nodes (includes leaves and all other nodes)?

Note: we'll use the standard convention that the leaves are not included in the depth, so a tree with depth 1 has 3 nodes with 2 leaves.

1.2 Running times of code

Your repository contains a file named `bigO.py`, which defines several functions that take an integer argument N. For each function, state the running time as a function of N, using big-O notation. Please include your answers in `main.py`. Do not write your answers inside `bigO.py`.

2 Data Exploration

Your repository contains the file `fluTrends.csv`, which contains estimates of the influenza-like illness percentage over 52 weeks on 2005-06 by Google Flu Trends. Your `main.py` loads this data for you and stores it in a pandas DataFrame `X`, where each row corresponds to a week and each column corresponds to a different region. If desired, you can convert from a DataFrame to a raw numpy array with `X.values()`.

2.1 Summary Statistics

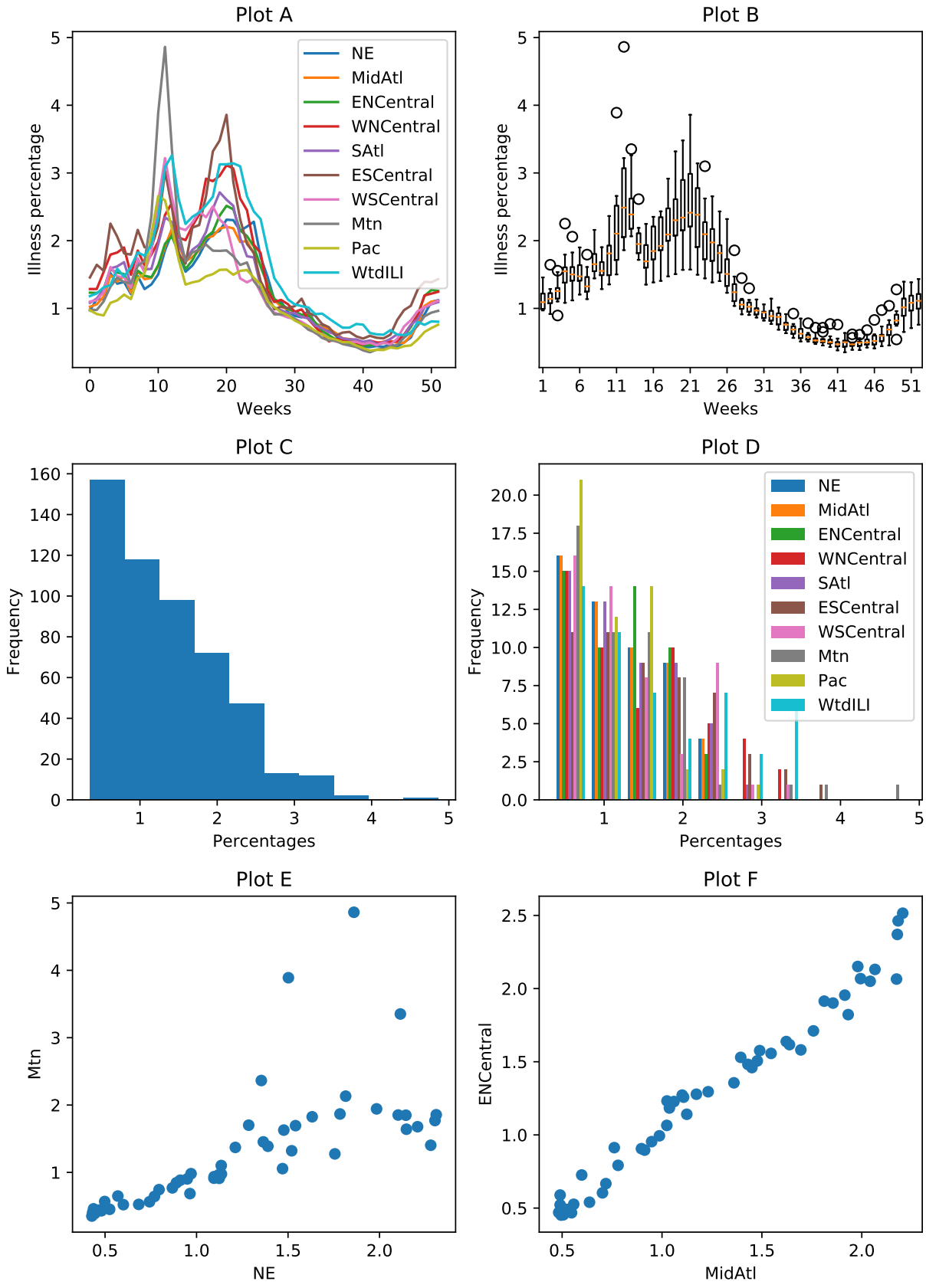
Report the following statistics in `main.py`:

1. The minimum, maximum, mean, median, and mode of all values across the dataset.
2. The 5%, 25%, 50%, 75%, and 95% quantiles of all values across the dataset.
3. The names of the regions with the highest and lowest means, and the highest and lowest variances.

In light of the above, is the mode a reliable estimate of the most "common" value? Describe another way we could give a meaningful "mode" measurement for this (continuous) data. Note: the function `utils.mode()` will compute the mode value of an array for you.

2.2 Data Visualization

Consider the figure below.


Plot A
Illness percentage
Weeks
NE
MidAtl
ENCentral
WNCentral
SAtl
ESCentral
WSCentral
Mtn
Pac
WtdILI
Plot B
Illness percentage
Weeks
Plot C
Frequency
Percentages
Plot D
Frequency
Percentages
NE
MidAtl
ENCentral
WNCentral
SAtl
ESCentral
WSCentral
Mtn
Pac
WtdILI
Plot E
Mtn
NE
Plot F
ENCentral
MidAtl

The figure contains the following plots, in a shuffled order:

1. A single histogram showing the distribution of *each* column in X.
2. A histogram showing the distribution of each the values in the matrix X.
3. A boxplot grouping data by weeks, showing the distribution across regions for each week.
4. A plot showing the illness percentages over time.
5. A scatterplot between the two regions with highest correlation.
6. A scatterplot between the two regions with lowest correlation.

Match the plots (labeled A-F) with the descriptions above (labeled 1-6), with an extremely brief (a few words is fine) explanation for each decision.

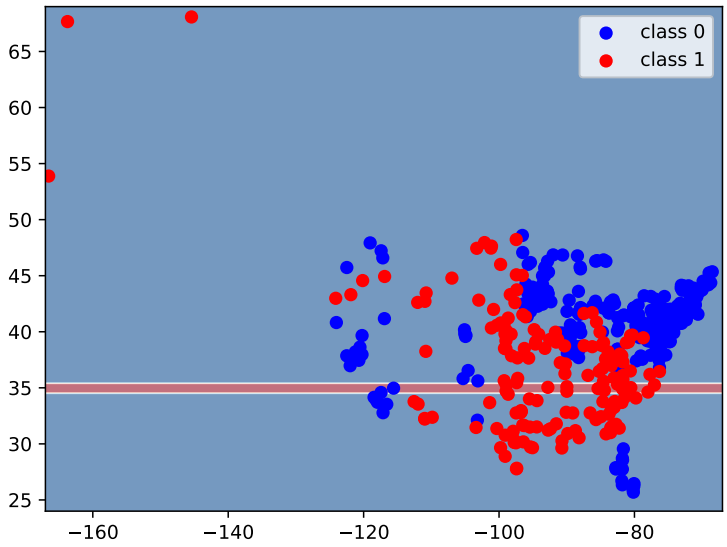
3 Decision Trees

The cities data corresponds to the US election in 2012. It was collected from Wikipedia and sampled from `http://simplemaps.com/static/demos/resources/us-cities/cities.csv`. It containing longitude and latitude data for 400 cities in the US, along with a class label indicating whether they were a "red" state or a "blue" state in the 2012 election. Specifically, the first column of the variable X contains the longitude and the second variable contains the latitude, while the variable y is set to 0 for blue states and 1 for red states.

3.1 Equal-based Decision Stump

The file `decision_stump.py` contains the class `DecisionStumpEquality` which finds the best decision stump using the equality rule and then makes predictions using that rule.

If you run `python main.py -q 3`, it will first load the cities dataset, plot the data and then fits two simple classifiers: a classifier that always predicts the most common label (0 in this case) and a decision stump that discretizes the features (by rounding to the nearest integer) and then finds the best equality-based rule (i.e., check if a feature is equal to some value). It reports the training error with these two classifiers, then plots the decision areas made by the decision stump. The plot is shown below:



As you can see, it is just checking whether the latitude equals 35 and, if so, predicting red (Republican). This is not a very good classifier.

Is there a particular type of features for which it makes sense to use an equality-based splitting rule rather than the threshold-based splits we discussed in class?

3.2 A more Generic Equality-based Decision Stump

We want to implement a more generic version de the decision stump equality in terms of:

1. Objective hyperparameter (`loss function`): choose the method for loss computation.
2. Scoring function (`score method`): report directly the score of validation.
3. Immediate prediction (`fit_predict method`): fit and predict in one step.

The file `decision_stump_generic.py` contains the class `DecisionStumpEqualityGeneric` (not yet implemented). Consider the implementation of this class as highlitted above. Also submit the generated figure of the classification boundary

Hint: you may want to start by copy/pasting the contents `DecisionStumpEquality` and then make modifications from there.

3.3 Inequality-based Decision Stump

Instead of discretizing the data and using a rule based on testing an equality for a single feature, we want to check whether a feature is above or below a threshold and split the data accordingly (this is a more sane approach, which we discussed in class). Create a `DecisionStumpErrorRate` class in `decison_stump_error.py` to do this, and report the updated error you obtain by using inequalities instead of discretizing and testing equality. Also submit the generated figure of the classification boundary.

Hint: you may want to start by copy/pasting the contents `DecisionStumpEquality` and then make modifications from there.

3.4 Decision Stump Info Gain

In `decision_stump_info.py`, create a `DecisionStumpInfoGain` class that fits using the information gain criterion discussed in lecture. Report the updated error you obtain, and submit the classification boundary figure.

Notice how the error rate changed. Are you surprised? If so, hang on until the end of this question!

Note: even though this data set only has 2 classes (red and blue), your implementation should work for any number of classes, just like `DecisionStumpEquality` and `DecisionStumpErrorRate`.

Hint: take a look at the documentation for `np.bincount`, at `https://docs.scipy.org/doc/numpy/reference/generated/numpy.bincount.html`. The `minlength` argument comes in handy here to deal with a tricky corner case: when you consider a split, you might not have any examples of a certain class, like class 1, going to one side of the split. Thus, when you call `np.bincount`, you'll get a shorter array by default, which is not what you want. Setting `minlength` to the number of classes solves this problem.

3.5 Constructing Decision Trees

Once your `DecisionStumpInfoGain` class is finished, running `python main.py -q 6.4` will fit a decision tree of depth 2 to the same dataset (which results in a lower training error). Look at how the decision tree is stored and how the (recursive) `predict` function works. Using the splits from the fitted depth-2 decision tree, write a hard-coded version of the `predict` function that classifies one example using simple if/else statements (see the Decision Trees lecture). Save your code in a new file called `simple_decision.py` (in the `code` directory) and make sure you link to this file from your README.

Note: this code should implement the specific, fixed decision tree which was learned after calling `fit` on this particular data set. It does not need to be a learnable model. You should just hard-code the split values directly into the code. Only the `predict` function is needed.

Hint: if you plot the decision boundary you can do a visual sanity check to see if your code is consistent with the plot.

3.6 Decision Tree Training Error

Running `python main.py -q 6.5` fits decision trees of different depths using the following different implementations:

1. A decision tree using `DecisionStump`
2. A decision tree using `DecisionStumpInfoGain`
3. The `DecisionTreeClassifier` from the popular Python ML library *scikit-learn*

Run the code and look at the figure. Describe what you observe. Can you explain the results? Why is approach (1) so disappointing? Also, submit a classification boundary plot of the model with the lowest training error.

Note: we set the `random_state` because sklearn's `DecisionTreeClassifier` is non-deterministic. This is probably because it breaks ties randomly.

Note: the code also prints out the amount of time spent. You'll notice that sklearn's implementation is substantially faster. This is because our implementation is based on the $O(n^2 d)$ decision stump learning algorithm and sklearn's implementation presumably uses the faster $O(nd \log n)$ decision stump learning algorithm that we discussed in lecture.

3.7 Comparing implementations

In the previous section you compared different implementations of a machine learning algorithm. Let's say that two approaches produce the exact same curve of classification error rate vs. tree depth. Does this conclusively demonstrate that the two implementations are the same? If so, why? If not, what other experiment might you perform to build confidence that the implementations are probably equivalent?