Project Guidelines

The goal of this project is to give students experience in the following aspects of applied machine learning:

- collecting data sets that are given in diverse formats;
- preparing data sets for training and testing;
- training many types of classification and regression models;
- performing hyperparameter search for each type of model;
- evaluating and summarizing classification and regression performance;
- generating plots that summarize performance;
- different approaches to interpretability; and
- writing about machine learning concepts, experiments, and results.

The specifics of these skills are not spelled out in the project guidelines, but rather something you should acquire by drawing upon lectures, labs, papers, lots of practice, and feedback from course staff.

Project Overview

1. Classification

The idea here is to train and evaluate 8 classification methods across 10 classification datasets.

Classification models

You should evaluate the following classification models:

- *k*-nearest neighbours classification
- Support vector classification
- Decision tree classification
- Random forest classification
- AdaBoost classification
- Logistic regression (for classification)
- Gaussian naive Bayes classification

- Neural network classification

Each of these is provided by scikit-learn under a unified interface. For example, MLPClassifier implements a fully-connected neural network classifier (also called a multi-layer perceptron, or MLP), and GaussianNB implements a Gaussian naive Bayes classifier. The AdaBoostClassifier implements AdaBoost for classification, for which using the default *base_estimator* is OK to use. Even though a model like logistic regression is not strictly a classifier, the scikit-learn implementation will still predict class labels.

Hyperparameters. Some types of models have more hyperparameters than others. You do not need to try every hyperparameter. Just choose 1–3 hyperparameters that are likely to have impact, such as C and γ for SVM, or *max_depth* and *n_estimators* for random forests, or *hidden_layer_sizes* and *learning_rate* and *max_iter* for neural networks. You need to choose and justify your strategy for picking hyperparameter ranges and for sampling the hyperparameters during hyperparameter search, and you should specify how you trained your final model once the best hyperparameters were found.

Classification datasets

You should evaluate each of the above classification model families on each the following UCI repository datasets:

1. Diabetic Retinopathy
2. Default of credit card clients
3. Breast Cancer Wisconsin
4. Statlog (Australian credit approval)
5. Statlog (German credit data) (recommend `german.doc` for instructions and `german-numeric` for data.)
6. Steel Plates Faults
7. Adult
8. Yeast
9. Thoracic Surgery Data
10. Seismic-Bumps

For these datasets you'll need to read the data set descriptions and discern which fields are intended to be features and which are the class labels to be predicted. If a dataset does not come with an explicit train/test split, then you will have to ensure your methodology can still evaluate the performance of the model on held-out data. Your conclusions regarding classification should draw from training and evaluating on the above datasets.

2. Regression

The idea here is to train and evaluate 7 regression methods across 10 regression datasets.

Regression models

You should evaluate the following regression models:

- Support vector regression
- Decision tree regression
- Random forest regression
- AdaBoost regression
- Gaussian process regression
- Linear regression
- Neural network regression

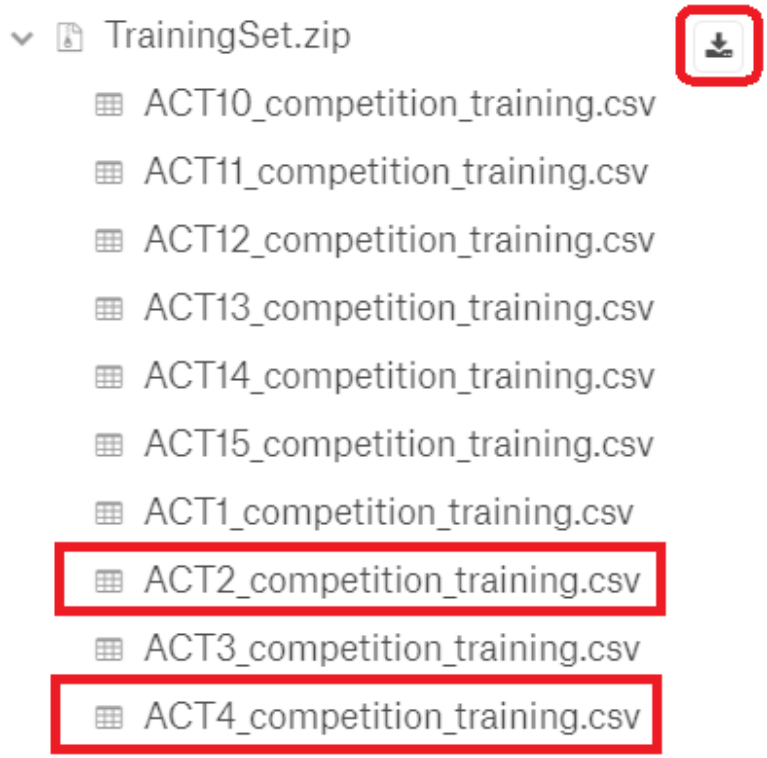
Each of these is provided by scikit-learn. For example, the MLPRegressor class implements a fully-connected neural network for regression. The SVR class implements support vector regression. The AdaBoostRegressor implements AdaBoost for regression, for which using the default *base_estimator* is OK. The GaussianProcessRegressor implements Gaussian process regression.

Regression datasets

You should evaluate each of the above regression model families on each the following datasets, which are mostly again from the UCI repository:

1. Wine Quality
2. Communities and Crime
3. QSAR aquatic toxicity
4. Parkinson Speech (extract RAR files with 7zip for Windows/Linux or Unarchiver for Mac.)
5. Facebook metrics
6. Bike Sharing (use `hour` data)
7. Student Performance (use just `student-por.csv` if you do not know how to merge the math grades)
8. Concrete Compressive Strength
9. SGEMM GPU kernel performance
10. Merck Molecular Activity Challenge (from Kaggle)

Merck Molecular Activity challege data. The Merck Molecular Activity challege is a much larger regression dataset than the others. The $40,000 prize went to a "deep multi-task neural network," but you do not need to evaluate that kind of model here. To download the training data, one of your group members must create a Kaggle account. Once registered, you should see the following download option:



To represent this dataset in your experiments, you should use the two training files shown above (and only those two), and use the evaluation metric of the original challenge which is described here. This metric is just the average of the squared Pearson correlation between predictions $\hat{y}_i$ and targets y_i.

The actual test data for the Merck challenge was never released, and you are not required to actually participate in the challenge itself (it is closed). So you in terms of evaluating performance can treat this like you would any dataset that does not have explicit an train/test split.

The Merck features are all small integers, and each file has a different number of features. I suggest loading the initial file as follows:

```
with open(MERCK_FILE) as f:
    cols = f.readline().rstrip('\n').split(',')  # Read the header line and get list of column names

# Load the actual data, ignoring first column and using second column as targets.
X = np.loadtxt(MERCK_FILE, delimiter=',', usecols=range(2, len(cols)), skiprows=1, dtype=np.uint8)
y = np.loadtxt(MERCK_FILE, delimiter=',', usecols=[1], skiprows=1)
```

and then caching the Numpy arrays *X* and *y* to a file using np.savez so that the data can be loaded faster when ready to train a model.

Remember, if an algorithm takes more than 3 minutes to train, you have the choice of either letting the method train longer in the hopes that it will finish, or you can declare (in the report) that the method did not finish training on that dataset.

3. Classifier interpretability

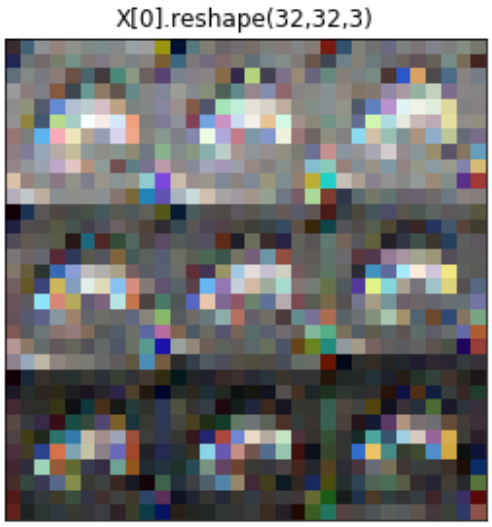
Here you will learn how to:

- load and train models on standard computer vision dataset called CIFAR-10;
- train a convolutional neural network (CNN) using PyTorch to classify images in the dataset;
- train a decision tree to classify images in the dataset;
- try to interpret the decision tree; and
- try to interpret the CNN using the 'activation maximization' technique.

Your team should report examples of what you found and state your current opinions on which of these models is most interpretable for vision data, if either of them are.

Visit the CIFAR dataset website, read about what the dataset contains, and then download the "CIFAR-10 python version." To extract the archive, use `tar -xvf cifar-10-python.tar.gz` on Linux/Mac or use `7z x cifar-10-python.tar.gz` with 7zip on Windows. Once you see the separate files (`data_batch_1` etc), follow the instructions on the CIFAR website for loading (unpickling) each data file in Python. Use the Python 3 version of the unpickling instructions. Note that to access the `data` and `labels` fields of the unpickled dictionary, you'll need to use byte keys `b'data'` and `b'labels'` rather than string keys `'data'` and `'labels'` .

Take note of the CIFAR website's description of how the images are represented. Load the first batch of images (say, into an ndarray called *X*) and try to plot the first image *X[0]*. You might think to simply reshape the length-3072 vector into shape $(32, 32, 3)$ (a 32×32 RGB image), but that would be wrong (see below).



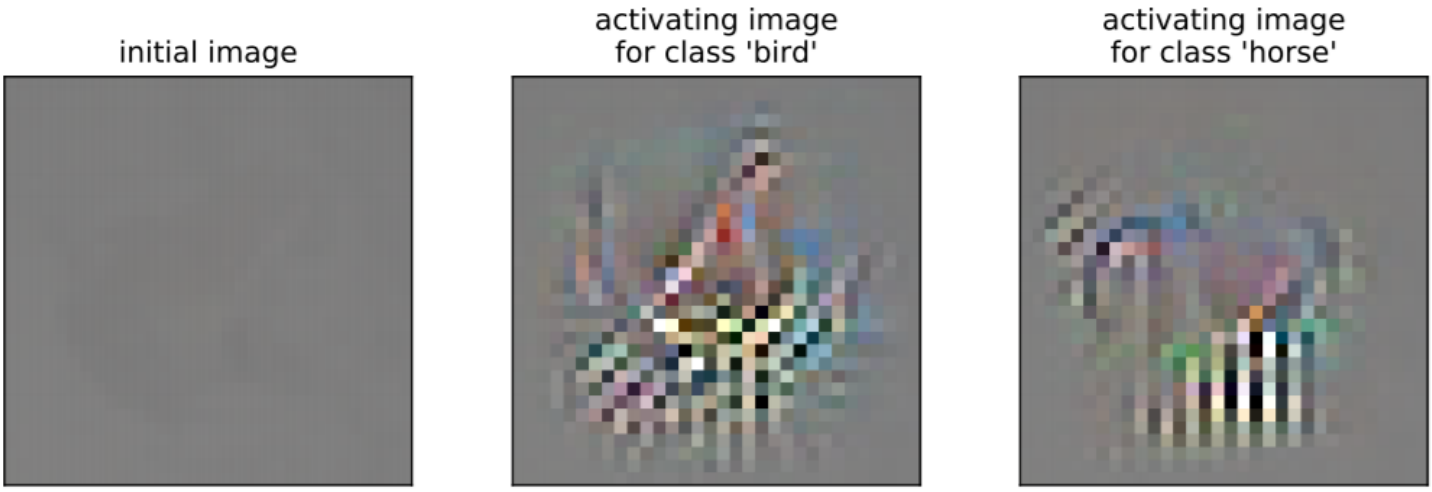
Convolutional neural networks typically expect colour image data to be in the format (N, C, H, W) which means the outermost dimension is over images $0, \ldots, N-1$, the next dimension is over colour channels $0, \ldots, C-1$ (in this case R, G, B), and the innermost dimensions are the height H and width W. So, if we want to take our (C, H, W)-formatted image and visualize it with Matplotlib (*imshow*), we need to transpose the colour axis to be the inner-most one, giving the image format (H, W, C) that Matplotlib expects. As we can see, this image probably belongs to class *frog*.

Before training a model, first write a script that combines the training batches (each of size 10,000) into a single giant batch (of size 50,000). You should end up with a 50000×3072 ndarray of dtype *np.uint8* as the features and a length-50000 ndarray *y* of dtype *np.int32* as the target labels. Likewise load the test batch and convert its targets from a `list` to an ndarray.

Decision tree classifier. Use these ndarrays to train a decision tree classifier on the entire training set. You may have to limit the tree depth for training to complete in reasonable time. Report the decision tree's accuracy on held-out test data. You can try plotting the tree like you did in lab, but use the plotting function's *max_depth* option to only plot the top few layers of the tree.

Convolutional neural network classifier. This part of the project you may not be able to complete until learning about convolutional neural networks in November. You must train a convolutional neural network on the CIFAR dataset using PyTorch. PyTorch is already installed in the lab machines, but can also be installed on your laptop. You will need to learn how to center and rescale your pixel values from $0, \ldots, 255$ of dtype *np.uint8* to be $[-1, 1]$ of dtype *np.float32*. Use any architecture you like, but keep trying until you achieve at least 75% accuracy on the test set. Train using *CrossEntropIt may take you a few attempts and each training attempt may 15-20 minutes on a laptop. If necessary, you can use techniques like data augmentation (the `torchvision` package can help with that) in order to achieve higher accuracy. Ensure that your script saves your trained model.

Activation maximization. Once your CNN is trained and saved, you can load it and start interrogating it. The idea of "activation maximization" is to ask the question: what input image would most strongly activate a particular output class prediction? Specifically, suppose we have an input image $\mathbf{x}$, which could be blank, or noise, or some other image. We feed this image through the CNN to get class activation $\hat{\mathbf{y}}$. We then choose a particular class i and ask: how should I perturb the pixels in $\mathbf{x}$ so as to increase $\hat{y}_i$? Deep learning frameworks allow us to compute the gradient of a particular class activation $\hat{y}_i$ with respect to all image pixels $\mathbf{x}$ by simply backpropagating. We can iterate this process, performing "gradient ascent" in pixel space. Below are two examples of an initial image $\mathbf{x}$ and a new image after maximizing either the 'bird' class label or the 'horse' class label. You may not see these exact results. Report what you are able to see.



To understand how this works in PyTorch, try to understand the code snippet below:

```
# Define some transformation of x, in this case a quadratic with maximum at +3
def f(x):
    return -(x - 3)**2

# Create a single scalar variable that stars with value 0. Also and notify PyTorch
# that we're eventually going to be asking for a gradient with respect to x.
x = torch.tensor(0.0, requires_grad=True)

print("%.3f" % x)
for i in range(10):
    # Evaluate our function transforming x into some quantity y that we want to maximize
    y = f(x)

    # Backpropagate a gradient of y with respect to x; this fills the x.grad variable
    y.backward()

    # Move x a small step in a direction that would increase y
    x.data += 0.2*x.grad.data

    # Reset the gradient for the next iteration, since backward() always adds to the current grad value.
    x.grad.data.zero_()

    print("%.3f" % x)
```

The above code outputs the following sequence of numbers for *x*, which converge to the maximum value that $f(x)$ can take:

```
0.000
1.200
1.920
2.352
2.611
2.767
2.860
2.916
2.950
2.970
2.982
```

In activation maximization, we are doing the same thing but where $\mathbf{x}$ is an image, $f(\mathbf{x})$ is a CNN, and y is some class activation score (the value being fed into the softmax for that particular class). You must implement activation maximization yourself. How meaningful are the patterns you see? Are you convinced that the model is learning high-level concepts like the appearance of "dog" and "cat"?

4. Novelty component

Try to introduce a novel aspect to your analysis of classifiers and regressors or to your investigation of interpretability. For example, you could add a new dataset to the study, or do a 'deep dive' one one particular dataset that's already included. You could do something as simple as try to inspect your convolutional filters as a secondary means of trying to interpret the model, and report on what patterns you saw and what they might mean. Or you could try applying interpretability techniques to one of the UCI datasets already analyzed, and report your experience.

Guidance on specific issues

Models not yet covered in lecture. Some of the models you are asked to evaluate will not be covered until later in the course, such as neural networks and Gaussian processes. That is OK, for now you can use them as a black box and take advantage of scikit-learn's unified interface (`model.fit(X,y)` , `model.predict(X)` , etc) to still apply these models to data. Or, you can wait until they're covered in the course and incorporate them into your project code. However, later on in the course you may come to understand the hyperparameters of these models better, and may want to re-run your experiments with that in mind, so making sure your experiments are automated and reproducible is important here (see "Tips on getting good marks").

Families of models. The suggested evaluation breaks models down into broad classes, such as "SVMs" rather than the more granular breakdown of "linear SVM" separate from "RBF SVM." So, we could make a statement like "SVM is the best classifier" when in fact the truth could be "RBF SVM worked best on dataset 1, whereas linear SVM worked best on dataset 2." Since these two example models belong to the SVM family, we'll count them as a win for the SVM family overall. We could likewise group models even more coarsely, as "parametric" or "non-parametric" and try to draw conclusions like "non-parametric models are the best," but this is too broad a claim to be useful. Similarly the "neural networks" family includes a broad spectrum of possible architectures (networks with many or few layers, with many or few hidden units per layer, with different activation functions, etc) and, if any one of the attempted architectures wins out on a particular data set, this individual win can be counted as a win for neural networks family as a whole. In other words, the number of layers and hidden units in a neural network can be treated as hyperparameters of "neural networks" rather than as separate types of models, and so they may need more computational investment in hyperparameter search than would a family of models that is more constrained.

Getting data sets. Data for this project comes from the UCI repository, a Kaggle challenge, and the University of Toronto CIFAR dataset website. The purpose of this project is to have you work with real data, including some minimal processing such as loading tabular data from text files. That means that, to receive full marks, you should use basic functions like `numpy.loadtxt` or load the raw data yourself using Python. If you find a specialized Python packages to fetch your data for you. You should instead download the data from its original

source, either in a web browser or with `wget`, and learn how to load it and process it from that form. For example, the `torchvision` Python package comes with a CIFAR10 DataSet class that will automatically download the raw binary data from the University of Toronto servers and serve the images as PyTorch tensors in (N, C, H, W) tensor format. Instead, you should be downloading "CIFAR-10 python version" from the CIFAR website and learn to understand and load the raw data as it was originally stored. For example, the CIFAR-10 training features are stored in five pickle files, each containing a 10000×3072 narray of dtype *np.uint8* (10000 32×32 RGB images); see the Python 3 `pickle.load` example on that website. That being said, if you are running out of time on your project and do not want to get stuck on the data loading aspect, you can try using specialized data loaders anyway and you won't lose many marks for that.

Missing values. Real-world datasets (*e.g.* from social media, businesses, hospitals) often contain "missing values" in their features. This means that the feature's actual value was unknown (unobserved). Throwing such data away is possible for the training set, but not for the test set. To still learn and to make predictions from the remaining values, one strategy is to 'impute' the missing values. The simplest way to impute a missing value is to assume (possibly incorrectly) that it probably took the most 'typical' value of that feature across the training set. This is considered a type of preprocessing, to prepare the data for training. See the scikit-learn guide for imputing missing values.

Long training times. Most of the datasets are small and training algorithms will complete in a matter of seconds. But a few datasets are of moderate size, such as the Merck and CIFAR datasets. Some models and learning algorithms do not scale to larger datasets, and so when you try to train them they will get stuck. Each time you try to train a model you should give it **at least 3 minutes** of training time, not including data loading and preprocessing. However, if the method takes longer than 3 minutes, you may decide to record that the method has "failed" to train for that particular dataset, rather than waiting to find the classification/regression performance. Models that take a flexible amount of time, such as MLPs (In real life we would give training algorithms much, much more time to complete than this, but for the sake of letting you try things.) The only exception is that when training a deep neural network on CIFAR-10 data, you should expect training time to be several minutes to an hour on a CPU, depending on the architecture and regularization you chose.

Reporting model performance. You need to propose a way of evaluating and comparing these models. Your evaluation methodology needs to be explained in the report. You can propose a methodology that you find convincing or cite an existing research paper (a benchmark) as inspiration for your chosen methodology. Your project code must ultimately evaluate every (model type, dataset) pair and then compare those individual results in such a way as to draw conclusions about which model (or models) you believe to be 'best', if any. It would help the