CO553 - Neural Networks Coursework

November 2019

1 Overview

Please read this manual **THOROUGHLY** as it contains crucial information about the assignment, as well as the support provided.

In this assignment, you will implement a mini neural network library and then use it (or an alternate high-level library of your choice) to solve a practical problem inspired by the insurance industry. This assignment is split into 3 parts:

- **Part 1: Creating a neural network mini-library**:
  1. Create a low-level implementation of a multi-layered neural network, including a basic implementation of the backpropagation algorithm.
  2. Implement data preprocessing, training, and evaluation utilities.
- **Part 2: Use your library for classification**
  1. Use the mini-library you have just developed (or any other NN framework) to solve a classification task in insurance pricing.
  2. Document your work in a report.
- **Part 3: A free for all competition in insurance pricing**
  1. Create an insurance pricing model trained on real data to compete with you classmates!
  2. Your models will have to pass certain tests to qualify
  3. Optionally create a linear model in addition to your model of choice

The mini-library in Part 1 must be designed using NumPy, and will require you to implement a linear layer class, activation functions, a multi-layer network class, a trainer class, and a data preprocessing class. Parts 2 and 3 of the assignment will be primarily assessed on the basis of your report - we will not be assessing your code, unlike for Part 1 (although you will still be required to submit your code for Parts 2 and 3, and we will run some of your code on a hidden dataset).

All implementations should be completed in Python 3. We do NOT provide any code or support for any other programming languages.

In addition to the instructions provided in this document, we provide a suite of tests for you to test your code prior to submission using the LabTS platform. To test your code, push to your GitLab coursework repository (that will be generated for your group) and access the tests through the LabTS portal available at:

https://teaching.doc.ic.ac.uk/labts/ .

The tests are only an indication of the status of your code, and may not reflect your final mark in any part of this coursework.

Please note that the test suite is expected to evolve between the release and the submission dates of this coursework.

Deadline: Monday - 25 of November, 2019 (7pm) on CATE.

2 Setup

2.1 Working on DoC lab workstations (recommended)

You can either work from home or use the lab workstations. See this list https://www.doc.ic.ac.uk/csg/facilities/lab/workstations to ssh into one of the machines. In order to load all the packages that you might need for the course work, you can run the following command:

```
export PYTHONUSERBASE=/vol/lab/ml/pypi
```

We have installed any packages you might need for all the work required in CO553. If you don't want to type that command every time that you connect to your lab machine, you can add it to your bashrc:

```
echo "export PYTHONUSERBASE=/vol/lab/ml/pypi" >> ~/.bashrc
```

This way, every terminal you open will have that environment variable set. It is recommended to use "python3" exclusively. The current python3 version in lab machines is 3.6.7. To test the configuration:

```
python3 -c "import numpy as np; print(np)"
```

This should print:

```
<module 'numpy' from '/vol/lab/ml/pypi/lib/python3.6/site-packages/numpy/__init__.py'>
```

2.2 Working on your own system

If you decide to work locally on your machine, then you must make sure that your code also runs on the lab machines using the above environment. **Anything we cannot run will result in marks for the question being reduced by 30%.**

All provided code has been tested on Python versions 3.5 or 3.6. Make sure to install Python version 3.5 or 3.6 on your local machine, otherwise you might encounter errors! If you are working on Mac OSX, you can use Homebrew to brew install python3.

Part 1: Creating a neural network mini-library

In this part, you will implement your own modular neural network mini-library using NumPy. This will require you to complete a number of classes in `src/nn\_lib.py`.

A simple dataset (https://en.wikipedia.org/wiki/Iris_flower_data_set, provided in the file `iris.dat`) and sample code demonstrating intended usage is provided for debugging purposes as you work through this part of the coursework.

Q1.1: Implement a linear layer:

For this question, you must implement a linear layer (which performs an affine transformation $XW + B$ on a batch of inputs X) by completing the following methods of the `LinearLayer` class:

Constructor: Define the following attributes in the constructor:

- NumPy arrays representing the learnable parameters of the layer, initialized in a sensible manner (hint: you can use the provided `xavier_init` function). Use the attributes `_W, _b` to refer to your weights matrix and bias respectively.

Forward pass method: Implement the `forward` method to do the following:

- Return the outputs of the layer given a NumPy array representing a batch of inputs.
- Store any data necessary for computing the gradients when later performing the backward pass in the `_cache_current` attribute.
- You are NOT allowed to use Python loops in this method (for efficiency reasons); use vectorized operations instead.

Backward pass method: Implement the `backward` method to do the following:

- Given the gradient of some scalar function with respect to the outputs of the layer as input, compute the gradient of the function with respect to the parameters of the layer

and store them in the relevant attributes defined in the constructor (`_grad_W_current` and `_grad_b_current`).

- Compute and return the gradient of the function with respect to the inputs of the layer.
- You are NOT allowed to use Python loops in this method (for efficiency reasons); use vectorized operations instead.

Parameter update method: Implement the `update_params` method to perform one step of gradient descent on the parameters of the layer (using the stored gradients and the learning rate provided as argument).

Here is an example of how this class can be used:

```
layer = LinearLayer(n_in=3, n_out=42)
# `inputs` shape: (batch_size, 3)
# `outputs` shape: (batch_size, 42)
outputs = layer(inputs)
# `grad_loss_wrt_outputs` shape: (batch_size, 42)
# `grad_loss_wrt_inputs` shape: (batch_size, 3)
grad_loss_wrt_inputs = layer.backward(grad_loss_wrt_outputs)
layer.update_params(learning_rate)
```

Q1.2: Implement activation function classes:

For this question, you must implement the `SigmoidLayer` and `ReluLayer` activation function classes (note: linear activation can be achieved without an activation class). In each case, complete the following:

Forward pass method: Implement the `forward` method to do the following:

- Returns the element-wise transformation of the inputs using the activation function.
- Store any data necessary for computing the gradients when later performing the backward pass in the `_cache_current` attribute.
- You are NOT allowed to use Python loops in this method (for efficiency reasons); use vectorized operations instead.

Backward pass method: Implement the `backward` method to do the following:

- Compute and return the gradient of the function with respect to the inputs of the layer.
- You are NOT allowed to use Python loops in this method (for efficiency reasons); use vectorized operations instead.

Q1.3: Implement a multi-layer network:

For this question, you must implement a multi-layer network (consisting of stacked linear layers and activation functions) by completing the following methods of the `MultiLayerNetwork` class:

Constructor: Define the following in the constructor:

- Attribute/s containing instances of the `LinearLayer` class and activation classes, as specified by the arguments:
  - `input_dim`: an integer specifying the number of input neurons in the first linear layer,
  - `neurons`: a list specifying the number of output neurons in each linear layer,
  - `activations`: a list specifying which activation functions to apply to the output of each linear layer.
- Store your layer instances in the `_layers` attribute.

Forward pass method: Implement the `forward` method to do the following:

- Return the outputs of the network given a NumPy array representing a batch of inputs (note: the instances of the `LinearLayer` classes created in the constructor should automatically handle any storage of data needed to compute gradients).

Backward pass method: Implement the `backward` method to do the following:

- Given the gradient of some scalar function with respect to the outputs of the network as input, compute the gradient of the function with respect to the parameters of the network. (note: the instances of the `LinearLayer` classes created in the constructor should automatically handle any storage of computed gradients).
- Return the gradient of the function with respect to the inputs of the network.

Parameter update method: Implement the `update_params` method to perform one step of gradient descent on the parameters of the network (using the stored gradients and the learning rate provided as argument).

Here is an example of how this class can be used:

```
# The following command will create a MultiLayerNetwork object
# consisting of the following stack of layers:
#   - LinearLayer(4, 16)
#   - ReluLayer()
#   - LinearLayer(16, 2)
#   - SigmoidLayer()
network = MultiLayerNetwork(
```

```
    input_dim=4, neurons=[16, 2], activations=["relu", "identity"]
)
# `inputs` shape: (batch_size, 4)
# `outputs` shape: (batch_size, 2)
outputs = network(inputs)
# `grad_loss_wrt_outputs` shape: (batch_size, 2)
# `grad_loss_wrt_inputs` shape: (batch_size, 4)
grad_loss_wrt_inputs = network.backward(grad_loss_wrt_outputs)
network.update_params(learning_rate)
```

Q1.4: Implement a trainer:

For this question, you must implement a "Trainer" class which handles data shuffling, training a given network using minibatch gradient descent on a given training dataset, as well as computing the loss on a validation dataset. To do so, complete the following methods of the `Trainer` class:

Constructor: Define the following in the constructor:

- An attribute (`_loss_layer`) referencing an instance of a loss layer class as specified by the `loss_fun` argument (which can take values `"mse"` or `"cross_entropy"`, corresponding to the mean-squared error and binary cross-entropy losses respectively.).

Data shuffling: Implement the `shuffle` method to return a randomly reordered version of the data observations provided as arguments.

Main training loop: Implement the `train` method to carry out the training loop for the network. It should loop over the following `nb_epoch` times:

- If `shuffle_flag = True`, shuffle the provided dataset using the `shuffle` method.
- Split the provided dataset into minibatches of size `batch_size` and train the network using minibatch gradient descent.

Computing evaluation loss: Implement the `eval_loss` method to compute and return the loss on the provided evaluation dataset.

Here is an example of how this class can be used:

```
trainer = Trainer(
    network=network,
    batch_size=32,
    nb_epoch=10,
    learning_rate=1.0e-3,
    shuffle_flag=True,
    loss_fun="mse",
)
trainer.train(train_inputs, train_targets)
print("Validation loss = ", trainer.eval_loss(val_inputs, val_targets))
```

Q1.5: Implement a preprocessor:

Data normalization can be crucial for effectively training neural networks. For this question, you will need to implement a "Preprocessor" class which performs min-max scaling such that the data is scaled to lie in the interval $[0, 1]$. Same as before, complete the methods of the `Preprocessor` class.

Constructor: Should compute and store data normalization parameters according to the provided dataset. This function does **not** modify the provided dataset. **Apply method:** Complete the `apply` method such that it applies the pre-processing operations to the provided dataset and returns the preprocessed version.

Revert method: Complete the `revert` method such that it reverts the pre-processing operations that have been applied to the provided dataset and returns the reverted dataset. For any dataset `A`, `prep.revert(prep.apply(A))` should return `A`.

Here is an example of how this class can be used:

```
prep = Preprocessor(dataset)
normalized_dataset = prep.apply(dataset)
original_dataset = prep.revert(normalized_dataset)
```

Part 2: Use your (or another) library for a classification task

Now that we have a mini-library, we can use it to train a deep neural network to solve an insurance pricing problem (alternatively, you are free to use another library of your choice to implement and train your models in the rest of the assignment).

Motor insurance pricing, is a supervised learning problem. Here, an insurer receives a person's record (attributes such as the age of the driver, the size of the car, the region, ...). This record also includes the claim amount which indicates the cost of a claim if a claim has been made. The insurer then develops a model that determines a price for the contract.

One of the fundamental parts of a pricing model is predicting whether or not a claim has occurred in the first place. This is a binary classification task, and the purpose of this part of the coursework.

For this task you will be using a simplified insurance data set, sourced from real historical data. You can find the data dictionary in the appendix.

PLEASE NOTE: FOR THIS EXERCISE YOUR TARGET VARIABLE IS ONLY THE COLUMN made_claim.[1]

THE INPUT FEATURES TO YOUR MODEL SHOULD NOT INCLUDE THE COLUMN claim_amount FOR OBVIOUS REASONS.

[1] see the data dictionary in the appendix for more information.

Q2.1: Implement a claim classifier

To predict the probability of a contract making a claim (e.g. being involved in an accident), the typical way is to build a classifier that tries to predict whether a person will make a claim or not from their record.

What you need to do

For this question you need to:

1. Open the `part2_claim_classifier.py` file and complete template file.
2. Among other comments that you find interesting, report the initial architecture used (number of layers, number of neurons per layer, activation function etc.) and the obtained performance.

Q2.2: Evaluate your architecture

The aim of this question is for you to show evidence of how you are evaluating your architecture.

What you need to do

1. In the file `part2_claim_classifier.py` populate the function `evaluate_architecture`
   - This must print or return indicators about the performance of your network
2. Explain in your report the methodology that you use

You are allowed to use utilities from libraries such as `scikit-learn` to evaluate your model.

Q2.3: Fine tune your architecture

Using the tools you have developed so far, perform a hyper-parameter search using a well thought methodology that you will detail in your report. You are allowed to use utilities from libraries such as `scikit-learn`.

What you need to do

1. Find and save your best performing model (hint: use the provided `save_model` function to generate a pickle file which will be used to load your model on labTS).
2. Report on your methodology.

3. Ensure that your code works. Any code that does not run or does not precisely adhere to the above specification will result in a loss of marks.

Note: We will be using a secret, hidden dataset to validate your reported results!

Part 3: A free for all competition in insurance pricing

In this last part of the coursework, you are asked to develop a complete pricing model (see below) that will be used in a competitive insurance market.

In part 2 your aim was to construct a classifier that makes correct *classification decisions*. In reality, the aim of a pricing model is to estimate the *expected cost* for each contract. Typically this is done by having two models:

1. **Frequency model $f(X)$:** A classifier to predict the *probability* of a claim (same as what you did in part2).
2. **Severity model $s(X)$:** A regressor to predict the amount of a claim. Creating a severity model is usually challenging, for this coursework we can consider the severity mode as a constant (i.e., same value for each client), for instance built from the mean amount of the claims. You will have to find the most appropriate constant for your implementation.

The final price $p(X)$ offered to a contract X is the product of the two components:

$$p(X) = f(X) \cdot s(X)$$

However, in this course, we will focus only on the frequency models. A constant will be used as the severity model for all submissions.

In this part of the coursework you are required to:

1. Build a frequency model that outputs the probability of a claim being made, a proxy for risk levels.[2].
2. Ensure your frequency model has an AUC of the ROC curve of over 0.6
3. **Optional** Try your hand at building a simpler, *linear* pricing model[3].
4. Test out your models in a competitive insurance market!

[2] As an extra note, see the section on probability calibration in the appendix.
[3] If you are not aware of what linear models are in classification, this short post explains the process in detail.

Q3.1: Build a frequency model (classifier) that predicts correct probabilities

What you need to do

For this coursework you will need to:

1. Use the dataset for Part3
2. Implement a frequency model by filling in `part3_pricing_model.py`
3. Find and save your best performing model (hint: use the provided `save_model` function to generate a pickle file which will be used to load your model on labTS).
4. The column of the dataset, called made_claim, should be used to create your frequency model as it indicates whether this client made a claim.
5. **(Optional)** Repeat the above steps with a linear model (such as a GLM or Logistic regression) and compare your results. Create the file `part3_pricing_model_linear.py` with your model following the same template as `part3_pricing_model.py`.

Q3.2 Engage in price competition

You can use LabTS to see the rank of your model in an AI market. You may consider tweaking your pricing model from Q3.1 so that it stands the best chance of making the most profit (see below for more information).

Overview of the market simulation

We are in essence, simulating a car insurance market, with N different companies competing fro K different customers.

In this market simulation:

1. Your pricing model will be one of N different models
2. Your pricing model will offer a price to each of the K different contracts in the market (i.e. test data)
3. Each of the K contracts then pick the cheapest of the N prices offered to them
4. Each of the N models will then be left with a fraction of the K contracts for which they offered the cheapest price
5. The model with the most profit wins (see below)

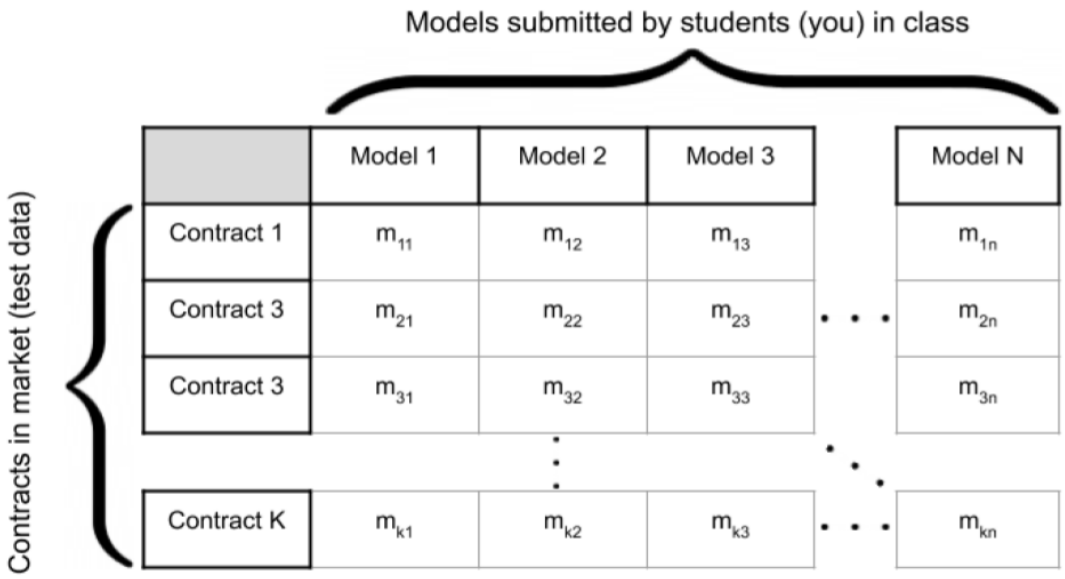

Figure 1: Each cell (i, j) of this matrix represent the price offered to contract i by model j

This insurance market simulation can be summarised best in a matrix representation where the cells of the matrix M represent the prices offered by each model (see figure 1 below).

As mentioned above, the contracts will pick the cheapest price offered to them. Therefore, we can define the set of contracts S_j won by the model j as:

$$S_j = \{\text{all } i \text{ such that } M_{ij} \text{ is the minimum of row } i \text{ of matrix } M\}$$

Note: If there are multiple minimum prices in a particular row, then one is chosen at random.

The profit for each model j is then computed as:

$$P_j = \sum_{i \in S_j} \left(M_{ij} - C_i\right)$$

Where C_i is the claim amount associated with the contract i.

The AI market

We recognise that most of you will never have played in such a game before. We also realise that how much profit a model makes in a monopoly, is very different to how much profit it

makes when it is competing with other models. **There is a balance between offering cheap prices to get more customers and making sure to offer expensive prices to the *bad, risky* customers that will end up making a claim.**

In order to help you get a *feel* for how your pricing model might rank in a competitive environment, we are offering you a fake AI market with which is populate with 10 other secret pricing models. You can use this tool via LabTS as mentioned above.

What you need to do

To be able to submit your model you must:

1. Make sure your pricing model has an AUC for the ROC curve of 0.6 or above
2. Ensure that your model consistently ranks in the top 50% of the AI market[4]

NOTES ON GRADING:

1. We will check the ROC AUC of your models on a separate hidden dataset which is used for grading
2. The performance of your model is evaluated on a separate hidden AI market with models that are similar to the AI market provided to you

Once you are satisfied with your model, you can submit it and you're done!

Two final notes:

- Your performance against your classmates in the market competition (i.e. your market profit / rank) will not contribute to your final grade on the coursework
- Your models will be used as part of a research project in the Data Science Institute, if you do not wish to participate, please speak with one of the GTAs.

Deliverables

You will have to submit two things on CATE:

- The SHA1 corresponding to the commit on you gitlab repository (used with LabTS). In this repository, you should have completed and pushed:
  - Completed version of `nn_lib.py`

[4]The AI market has been trained on the first 60,000 rows of the training data. You may want to consider this when using the AI market.

  - Completed version of `part2_claim_classifier.py`
  - Uploaded a `part2_claim_classifier.pickle`
  - Completed version of `part3_pricing_model.py`
  - Uploaded a `part3_pricing_model.pickle`
  - (optional) any other code that you created for the optional parts
  - (optional) a readme explaining how to run your code, if needed.
- Report (PDF) containing answers to written questions in each part of this coursework

Grading scheme

Total = 100 marks.

- Part 1 (40 marks):
  - Linear layer (10 marks)
  - Implement activation function classes (5 marks)
  - Multilayer network (10 marks)
  - Trainer (10 marks)
  - Preprocessor (5 marks)
- Part 2 (25 marks):
  - Implement claim classifier (10 marks)
  - Evaluate your architecture (5 marks)
  - Fine tune your architecture (5 marks)
  - Evaluation on hidden dataset (5 marks)
- Part 3 (20 marks):
  - Pass the AUC-ROC requirement (10 marks)
  - Rank in the top 50% of the AI market (10 marks)
- Report presentation quality (15 marks)

A Appendix

A.1 Data dictionary

Below is the full data dictionary. However, note that in part 2 you are receiving a subset of the full data, which you receive in part 3.

NOTE: Variables in RED appear in both the data for part 2 and part 3.

1. **id_policy**: A unique ID for contracts

2. **pol_bonus**: The bonus/malus system is compulsory in France. The coefficient is attached to the driver. It starts at 1 for young drivers (i.e. first year of insurance). Then, every year without claim, the bonus decreases by 5% until it reaches its minimum of 0.5. Without any claim, the bonus evolution would then be: 1, 0.95, 0.9, 0.85, 0.8, 0.76, 0.72, 0.68, 0.64, 0.6, 0.57, 0.54, 0.51, 0.5. Every time the driver causes a claim (only certain types of claims are taken into account), the coefficient increases by 25%, with a maximum of 3.5. Thus, the range of pol bonus extends from 0.5 to 3.5 in the dataset.

3. **pol_coverage**: The coverage are of 4 types : Mini, Median1, Median2 and Maxi, in this order. Mini policies cover only Third Party Liability claims, whereas Maxi policies covers all claims, including Damage, Theft, Windshield Breaking, Assistance, etc.

4. **pol_duration**: Policy duration represents how old the policy is. It is expressed in year, accounted from the beginning of the current year i.

5. **pol_sit_duration**: Represent how old the current policy characteristics are. This is different from pol_duration, because the same insurance policy could have evolved in the past (e.g. coverage, or vehicle, or drivers, ...).

6. **pol_pay_freq**: The price of the insurance coverage can be paid annually, bi-annually, quarterly or monthly. Be aware that you must provide a yearly cotation in your answer to the pricing game.

7. **pol_payd**: The pol_payd is a boolean (i.e. a string with Yes or No), which indicates whether the client has subscribed a mileage-based policy or not.

8. **pol_usage**: The policy use describes what usage the driver makes from the vehicle, most of time. There are 4 possible values : WorkPrivate which is the most common, Retired which is presumed to be aimed at retired people (who also are presumed to drive less). Professional which denotes a professional usage of the vehicle, and AllTrips which is quite similar to Professional (including professional tours).

9. **pol_insee_code**. INSEE code is 5-digits alphanumeric code used by the French National Institute for Statistics and Economic Studies (hence INSEE) to identify communes and departments in France. There are about 36,000 'communes' in France, but not every one of them is present in the dataset . The first 2 digits of insee code identifies the 'department' (they are 96, not including overseas departments).

10. **drv_drv2**: This boolean (Yes/No) identifies the presence of a secondary driver on the contract. There is always a first driver, for whom characteristics (age, sex, licence) are provided, but a secondary driver is optional, and is present in roughly a third of contracts.

11. **drv_age1**: This is the age of the first driver in years.

12. **drv_age2**: When drv_drv2 is Yes, then the secondary driver's age is present. When not, this is 0.

13. **drv_sex1**: European rules force insurers to charge the same price for women and men. But gender can still be used in academic studies, and that's why drv_sex1 is still available in the datasets. For our purposes, let's assume that this can be used as discriminatory variable in this pricing game.

14. **drv_sex2**: The gender of the second driver if present in the data.
15. **drv_age_lic1**: The age of the first driver's driving license in years.
16. **drv_age_lic2**: The age of the second driver's license. Be wary of outliers in the data.
17. **vh_age**: This variable is the vehicle's age, the difference between the year of production and the current year. One can consider values of 1 or 2 to correspond to new vehicles.
18. **vh_cyl**: The engine cylinder displacement is expressed in ml in a continuous scale.
19. **vh_din**: A representation of the engine power. Don't be surprised to find correlations between cylinder displacement, power, speed or even the value of the vehicle.
20. **vh_fuel**: Diesel, Gasoline or Hybrid. This is the fuel type of the vehicle.
21. **vh_make**: The make (brand) of the vehicle. As the database is built from a French insurance set, the three major brands are Renault, Peugeot and Citroën.
22. **vh_model**: As a subdivision of the make, vehicle is identified by its model name. The are about 100 different make names in the datasets, and about 1,000 different models. If you used the, consider maybe concatenation of the two variables.
23. **vh_sale_begin**: The difference between the current year and the year in which the vehicle first went into production.
24. **vh_sale_end**: The difference between the current year and the year in which the vehicle officially went out of production.
25. **vh_speed**: This is the maximum speed of the vehicle, as stated by the manufacturer.
26. **vh_type**: Tourism or Commercial. You'll find more Commercial types for Professional policy usage than for WorkPrivate.
27. **vh_value**: The vehicle's value (replacement value) is expressed in euros.
28. **vh_weight**: The weight (in kg) of the vehicle.
29. **town_mean_altitude**: The altitude of the town centre.
30. **town_surface_area**: Approximate surface area of the town.
31. **population**: The population of the town
32. **commune_code**: Commune, canton, city district, and regional department codes can act as lookup code to figure out exactly which area the policy belongs too.
33. **canton_code**
34. **city_district_code**
35. **regional_department_code**
36. (RELATED TO TARGET VARIABLE) **claim_amount**: The amount of a claim expressed in euros for the year (a value of zero indicates that no claim was made).
37. (TARGET VARIABLE) **made_claim**: A binary variable with a value of 1 indicating that a claim has been made.

A.2 Probability calibration

Probability calibration is a niche but interesting part of classifier tuning that will be partially introduced here. But before we start:

DISCLAIMER: You are not required to implement probability calibration in your models, doing so will not affect your grade in any way. If you wish to use probability calibration, the template file provided for part 3 contains a function that does so for you. See template for instructions.

Now that that has been said, let's begin.

Many classifiers use some sort of internal *score* to make classification decisions (e.g. the distribution of classes in the leaf node of a decision tree). However, these don't always correspond to probabilities. How can we check?

To answer this, let's ask ourselves and illustrative question in a specific example: *How would we check if the probability estimate of 0.2 is correct, as given by our classifier?*

When the classifier says the probability of a positive class is 0.2, then we expect that 20% of the time, those instances that are assigned this probability estimate will belong to the positive class. This intuition is quantified in a *reliability* or **calibration** curve (see figure 2).

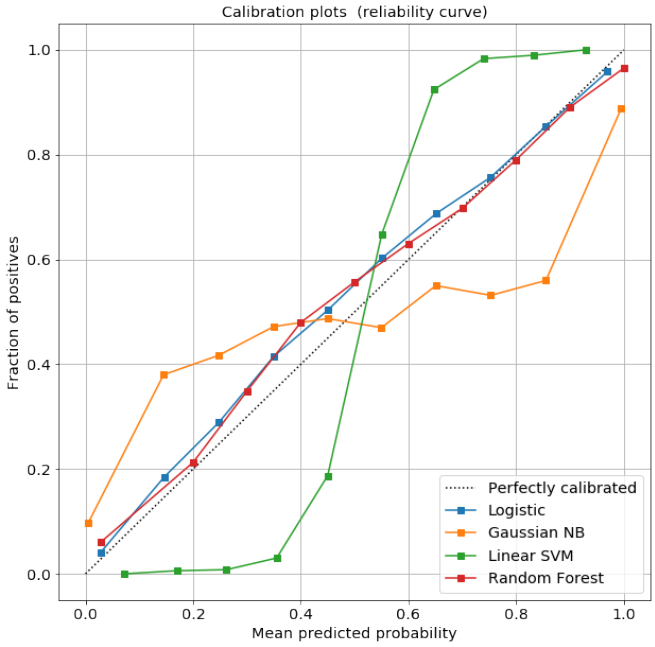

Figure 2: The reliability curve of several well known classifiers are compared.

For example, in figure 2 we can see that logistic regression, and random forests, in this artificial toy data set are well calibrated. An SVM with a linear kernel function on this data set tends to overestimate probabilities below 50% and underestimate them above 50% (e.g. when it says

40% the probability is closer to 10%).

Now, let's say we have a classifier already, we know it is not calibrated. What can we do? The answer is of course, to calibrate it. See figure 3 to see the results (we will discuss how below).

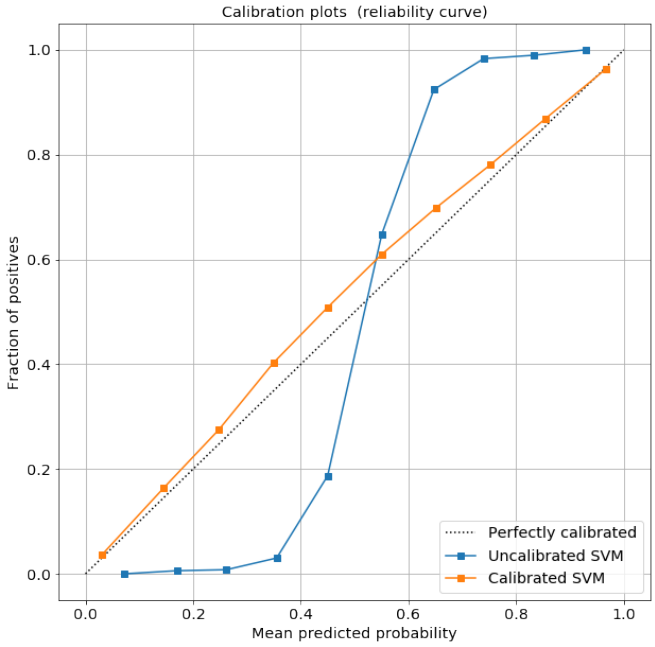

Figure 3: The effect of probability calibration is seen on an SVM classifier.

Calibrating probabilities

A classifier $f(X)$ typically outputs a set of scores $\hat{y}_{\text{scores}}$ (with y being the true class labels). Due to a variety of reasons such as data set imbalance, or even some details of the objective function, these scores might not represent calibrated probabilities. In order to turn them into calibrated probabilities some method of probability scaling is used. One example is Platt Scaling where the scores $\hat{y}_{\text{scores}}$ are fed into a logistic regression with y as the target. The output of the logistic regression is then used as probability estimates. For more on probability calibration please visit this link and references therein.

DONT' PANIC: You will not be required to implement any probability calibration in this coursework, we will provide the code for you, and are only mentioning probability calibration for your benefit and curiosity.