Linear Classification

by

Samirah Amadu & Inti Gabriel Mendoza Estrada

Neural Networks KU WS19 - Exercise Sheet 2

Assoc. Prof. Dipl.-Ing. Dr. techn. Robert Legenstein
Name and title of the supervisor

Date of Submission: November 23, 2019

TU Graz — Computer Science Masters Programme

Contents

1 Introduction

Given a dataset `vehicle.pkl`, we aim to classify a given silhouette as one of two types of vehicles, i.e., `SAAB` or `VAN`. To do so, we use a *Probabilistic Generative Model* approach that allows us to classify our dataset as well as the `Iterative Reweighted Least Squares (IRLS)` algorithm. We will compare and contrast each model's performance on our `vehicle.pkl` dataset.

The dataset consists of 270 training examples and 146 test examples which each have 18 features characterizing the object. There are two classes (`vehicle.pkl` has 4 'classes' but we only extract 2 of them) of interest: SAAB (Class tag 2) and VAN (Class tag 4).

We aim to minimize misclassification rate on the test set after training with either algorithm.

2 Implementation

We used Python 3.7.0 to develop our implementation of the algorithms. To extract the dataset from `vehicle.pkl` we use:

```
# Training set
X = vehicle_data['train']['X']  # features; X[i,j]...feature j of example i
C = vehicle_data['train']['C']  # classes; C[i]...class of example i
# Test set
Xtst = vehicle_data['test']['X']  # features
Ctst = vehicle_data['test']['C']  # classes

# extract examples of class SAAB (2) and VAN (4)
indices = np.flatnonzero((C == 4) | (C == 2))
C = C[indices]
X = X[indices]

indices_tst = np.flatnonzero((Ctst == 4) | (Ctst == 2))
Ctst = Ctst[indices_tst]
Xtst = Xtst[indices_tst]
```

2.1 Probabilistic Generative Model

We use the *Probabilistic Generative Model* approach to classify the data set, assuming Gaussian class-conditional distributions with a common covariance matrix. To estimate the class prior probability, covariance matrix, the means, and the posterior distribution, from the training data, we do:

```
# find prior probability
nr_training_examples = C.size

unique, examples_per_class = np.unique(C, return_counts=True)  # counts .. number of examples per class, unique .. classes
prior = examples_per_class / nr_training_examples  # convert count into percentage
priors = dict(zip(unique, prior))

# find mean for each class
mean = {}
i = 0
for cls in unique:
    mean[cls] = (1/examples_per_class[i]) * np.sum(X[np.flatnonzero(C == cls)], axis=0)
    mean[cls] = mean[cls].reshape((-1, 1))
    i = i+1

# compute covariance matrix
s = {}
normalized_features = {}
for cls in unique:
    indices = np.flatnonzero(C == cls)
    normalized_features[cls] = X[indices].T - mean[cls]

j = 0
for cls in unique:
    s[cls] = (1/examples_per_class[j]) * (normalized_features[cls] @ normalized_features[cls].T)
```

```
26 |       j = j+1
27 |
28 |   covariance = (examples_per_class[0]/nr_training_examples) * s[unique
       [0]] + \
29 |                (examples_per_class[1]/nr_training_examples) * s[unique
       [1]]
30 |
31 |   # compute posterior probability
32 |   sigmoid = lambda a: np.where(a >= 0, 1 / (1 + np.exp(-a)), np.exp(a)
        / (1 + np.exp(a))) # numerically stable
```

From the lecture notes we get the *Gaussian-Class Conditionals With Common Covariance Matrix*:

If $p(\mathbf{x}|C_k) \sim \mathcal{N}(\mu_k, \Sigma)$, then we arrive at:

$$\begin{aligned} p(C_1|x) &= \sigma(\mathbf{w}^T\mathbf{x} + w_0) \text{ with} \\ \mathbf{w} &= \Sigma^{-1}(\mu_1 - \mu_2), \\ w_0 &= -\frac{1}{2}\mu_1^T\Sigma^{-1}\mu_1 + \frac{1}{2}\mu_2^T\Sigma^{-1}\mu_2 + ln\frac{p(C_1)}{p(C_2)}. \end{aligned}$$

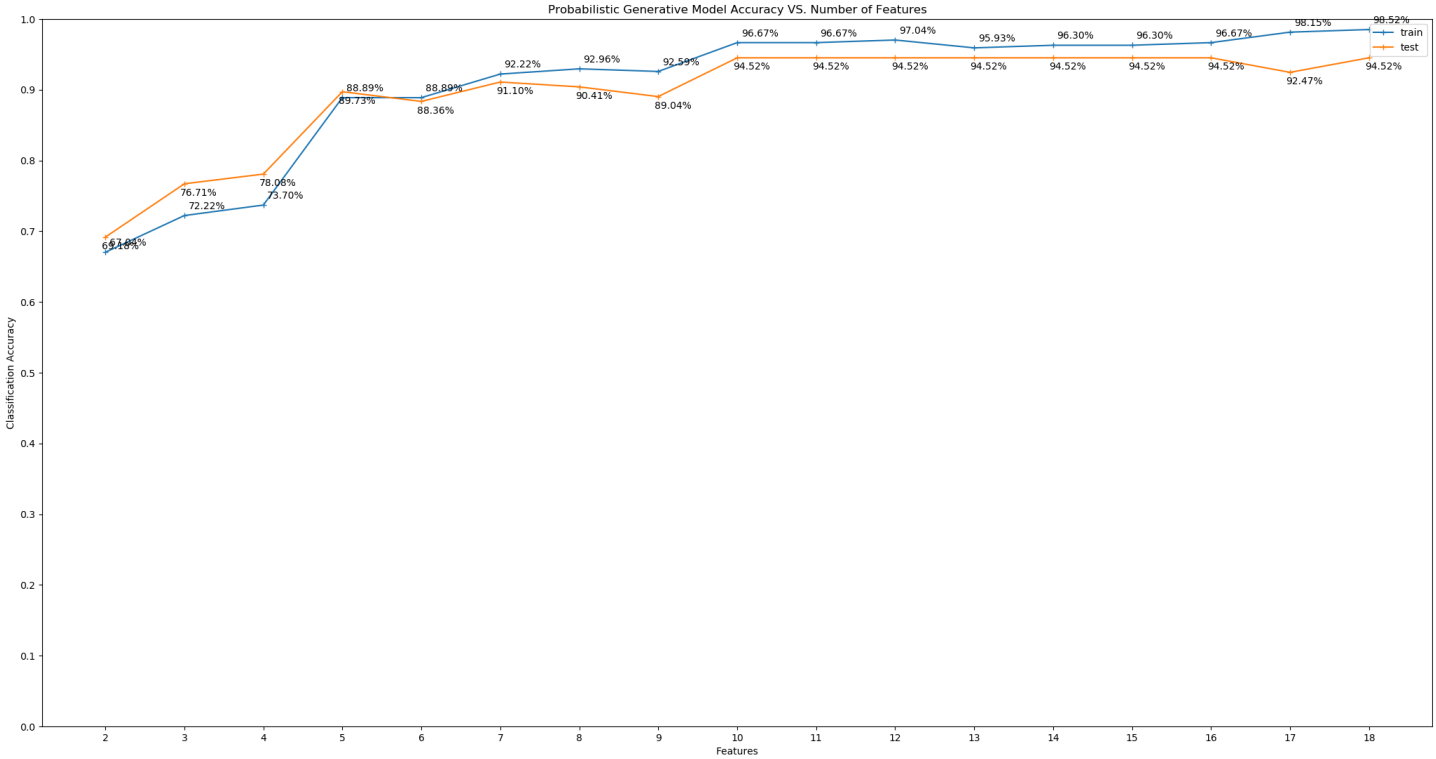
We implement this in Python as:

```
01 |  def classify(X, mean, covariance, target, flag="train"):
02 |      result = np.zeros((X.shape[1]-1, 2), dtype=float)
03 |      for features in range(2, X.shape[1]+1):
04 |          sub_m1 = mean[2][0:features]
05 |          sub_m2 = mean[4][0:features]
06 |          covinverse = np.linalg.pinv(covariance[0:features, 0:
       features]) # Simga^-1
07 |          weights = covinverse @ (sub_m1 - sub_m2) # w
08 |          bias = (-(1/2) * sub_m1.reshape((1, -1)) @ covinverse @
       sub_m1) \
09 |                 + ((1/2) * sub_m2.reshape((1, -1)) @ covinverse @
       sub_m2) \
10 |                 + np.log(priors[2]/priors[4]) # w_0
11 |          input = X[:, 0:features]
12 |          prediction = (sigmoid(weights[0:features].reshape((1, -1)) @
        input.T + bias)) # p(C_1|x)
13 |          [...] # out of scope code
14 |
15 |      return result
```

If our `prediction` variable (seen above) is smaller than 0.5, the object is classified as `VAN`, else as `SAAB`. We are able to do the classification using only the first 2, then 3, etc...up to all features of the dataset through the `for` loop. The dataset is a table in which the columns are the features, therefore using `[0:features]` allows us to partition the array/table by the specified amount of features for each iteration.

We are able, then, to plot the classification accuracy (percentage of correctly classified examples) as a function of the number of input features for both the training set and the test set.

The training correct classification percentage on the training set reaches a 98.52% and 94.52% on the test set 'at' 18 features. Note that the misclassification rate is just (1.0 - `Accuracy`). The full graph can be seen below:



2.2 Iterative Reweighted Least Squares

We implemented the *Iterative Reweighted Least Squares(IRLS)* algorithm and applied it to the dataset.

The *IRLS* update formula is:

$$\begin{aligned} \mathbf{w}^{\text{new}} &= \mathbf{w}^{\text{old}} - H^{-1} \cdot \nabla_{\mathbf{w}} E \\ &= \mathbf{w}^{\text{old}} - (X^T R X)^{-1} \cdot X^T (\mathbf{y} - \mathbf{t}). \end{aligned}$$

We initialize the weights $\mathbf{w}_0$ as a column vector of Gaussian-distributed random numbers between $(-1 \times 10^{-4}, 1 \times 10^{-4})$ with |features| elements through the code:

```
01 |  weights = np.random.uniform(low=-0.0001, high=0.0001, size=input.
       shape[1])[:, np.newaxis]
```

We obtain H by computing $X^T R X$ where R is the diagonal matrix with $R_{mm} = y_m(1-y_m)$ where $y_m = \sigma(\mathbf{w}^T \mathbf{x}^{(m)})$ and $\sigma(a) = \frac{1}{1+exp(a)}$. The code is as follows:

```
01 |  # numerically stable sigmoid function
02 |  sigmoid = lambda a: np.where(a >= 0, 1 / (1 + np.exp(-a)), np.exp(a)
       / (1 + np.exp(a)))
03 |
04 |  y = sigmoid(weights.reshape((1, -1)) @ input.T) # w = input
05 |
06 |  def hessian(X, y):
07 |      R = np.diag(y[0]*(1-y[0]))
08 |      return X.T @ R @ X
```

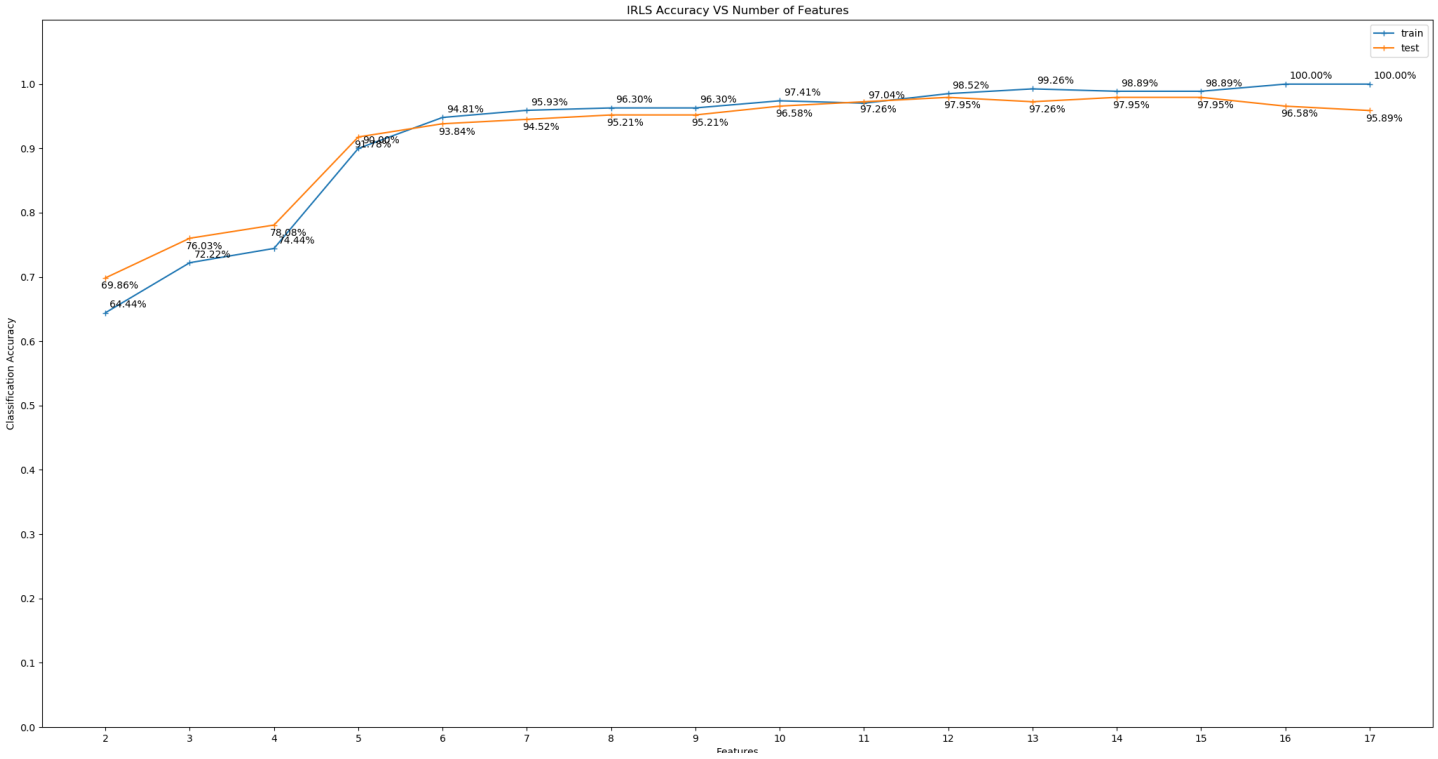
We obtain $\nabla_{\mathbf{w}} E$ by computing $X^T(\mathbf{y} - \mathbf{t}))$ where y is computed as seen above, and t being the normal target column vector. The code is as follows:

```
01 |  def nabla_e(X, y, target):
02 |      return X @ (y - target)
```

We obtain $\mathbf{w}^{\text{new}}$ by updating $\mathbf{w}^{\text{old}}$ (initially $\mathbf{w}_0$ computed above) with the *IRLS* update formula. We update the weights until the new *negative log-likelihood error* (calculated at each iteration) differs with the previous iteration's *negative log-likelihood error* by 1×10^{-4} or less. Reducing the misclassification error any further is no longer efficient. The code is as follows:

```
01 |  def update(weights, hessian, gradient): # IRLS update formula
02 |      return weights - (np.linalg.inv(hessian) @ gradient)
03 |
04 |  while np.abs(error_old - error_new) > 0.0001:
05 |          error_old = error_new
06 |
07 |          # update until error converges
08 |          hess = hessian(input, y)
09 |          gradient = nabla_e(input.T, y.reshape((-1, 1)), target)
10 |          weights = update(weights, hess, gradient)
11 |
12 |          y = sigmoid(weights.reshape((1, -1)) @ input.T)
13 |          error_new = log_likelihood(y.reshape((-1, 1)), target)
14 |          [...] # out of scope code
```

The training correct classification percentage on the training set reaches a 100.00% and 95.89% on the test set 'at' 18 features. Note that the misclassification rate is just (1.0 - `Accuracy`). The full graph can be seen below:

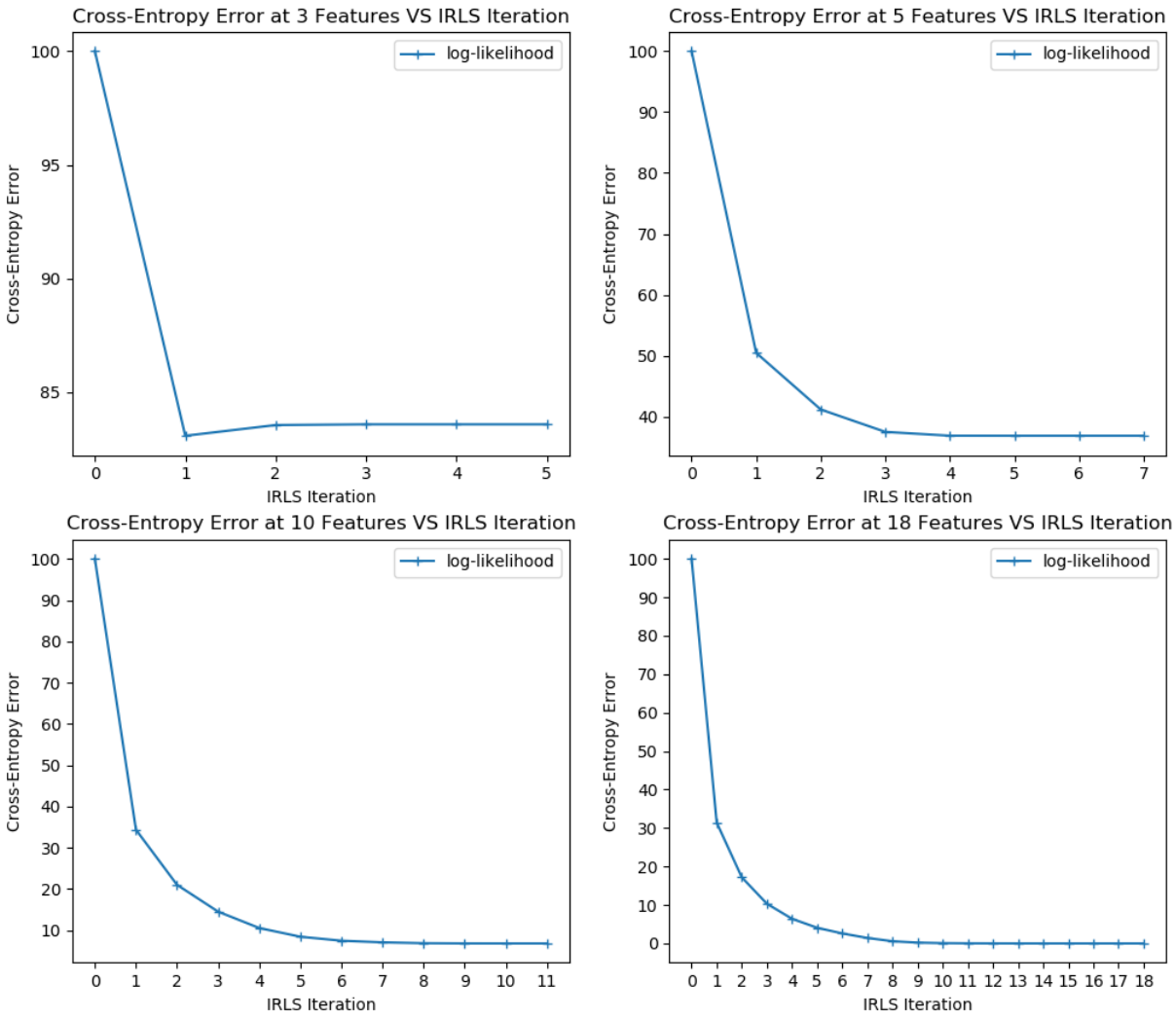


Furthermore, we can observe the evolution of the cross-entropy error as *IRLS* iterations are performed at different feature-numbers. Below we show the *Cross-Entropy Error* at 3, 5, 10, and 18 features at every iteration. The last computed cross-entropies are:

Features	Negative Log-Likelihood
3	[83.58347373]
5	[36.85420504]
10	[6.81522355]
18	[2.56824073e-05]

These are computed at every iteration of the *IRLS* algorithm with the code:

```
01 | def log_likelihood(estimate, target):
02 |     err_per_example = np.where(target != 0, target * np.log(estimate
       ), 0)
03 |     err_per_example = np.where((1-target) != 0, err_per_example +
       (1-target) * np.log(1 - estimate), 0)
04 |     return -np.sum(err_per_example, axis=0)
```



3 Conclusion

We have classified the given `vehicle.pkl` dataset with the *Probabilistic Generative Model* and in parallel applied the *Iterative Reweighted Least Squares (IRLS)* algorithm to it. Individually, we notice the following things from each approach:

- *Probabilistic Generative Model*: Only after including 5 features in the classification model do we get a significant jump in accuracy. However, only until the 10th feature is added do we jump to a 'respectable' ($< 8\%$) misclassification rate (on the test data). However, even though the accuracy on the train data increases (as expected) with the inclusion of more features, it never reaches 100% accuracy. The misclassification rate on the test data does not decrease (strangely at 17 features it increases but is quickly corrected at the 18th feature).
- *Iterative Reweighted Least Squares (IRLS)*: Only after including 6 features in the algorithm do we get a significant jump in accuracy. Furthermore, at the 6th added feature do we jump to a 'respectable' ($< 8\%$) misclassification rate (on the test data). We reach 100% accuracy on the train set at 16 added features and our lowest misclassification rate is encountered at 12, 14, and 15 added features, not after all are added, peculiarly.

When comparing both methods, we can clearly see that *IRLS* is better performing than the *Probabilistic Generative Model*. *IRLS* reaches a 'respectable' ($< 8\%$) misclassification rate much earlier than the *Probabilistic Generative Model*. Having less features required to reach this misclassification point makes it much more efficient - computationally cheap. At the 8th added feature, *IRLS* reaches a misclassification rate the *Probabilistic Generative Model* will not be able to beat. Furthermore, as more features are added from this point, *IRLS* does in fact continue to improve to achieve a peak of 97.95% accuracy. 3.43% higher than the *Probabilistic Generative Model*'s peak accuracy.