CSC411: Assignment #3

Due on Monday, March 19, 2018

Chawla Dhruv, Maham Shama Saila

March 18, 2018

Part 1

Dataset Description

The dataset is present in `clean_fake.txt` and `clean_real.txt`. Each line of the files contains a headline all in lower case. Example: `trump warns of vote flipping on machines` is present in `clean_fake.txt`.

It does seem feasible to predict whether a headline is real or fake. There are certain phrases whose presence seems to indicate if a headline is real or not:

1. `clean`: appears in 1 real headline and 5 fake headlines
2. `hillary`: appears in 24 real headlines and 150 fake headlines
3. `donald`: appears in 832 real headlines and 231 fake headlines

Part 2

Naive Bayes Algorithm

The Naive Bayes algorithm is applied on the dataset in `naive_bayes.py`.

Our goal is to compute $P(fake|w)$ given $P(w|fake)$.

For a test headline, assume $w_i = 1$ if headline contains the word w_i and $w_i = 0$ otherwise.

Then for training:

$$\begin{aligned}
\hat{P}(w_i = 1|fake) &= \frac{number_of_fake_headlines_containing_w_i + m\hat{p}}{number_of_fake_headlines + m} \\
\hat{P}(w_i = 0|fake) &= 1 - \hat{P}(w_i = 1|fake) \\
\hat{P}(w_i = 1|real) &= \frac{number_of_real_headlines_containing_w_i + m\hat{p}}{number_of_real_headlines + m} \\
\hat{P}(w_i = 0|real) &= 1 - \hat{P}(w_i = 1|real) \\
\hat{P}(fake) &= \frac{number_of_fake_headlines}{number_of_total_headlines} \\
\hat{P}(real) &= 1 - \hat{P}(fake)
\end{aligned}$$

For classifying:

$$\begin{aligned}
\hat{P}(fake|w_1, w_2, ..., w_n) &\propto \hat{P}(fake) \prod_{i=1}^{n} \hat{P}(w_i|fake) \\
\hat{P}(real|w_1, w_2, ..., w_n) &\propto \hat{P}(real) \prod_{i=1}^{n} \hat{P}(w_i|real) \\
\hat{P}(fake|w_1, w_2, ..., w_n) &= \frac{\hat{P}(fake) \prod_{i=1}^{n} \hat{P}(w_i|fake)}{\hat{P}(fake) \prod_{i=1}^{n} \hat{P}(w_i|fake) + \hat{P}(real) \prod_{i=1}^{n} \hat{P}(w_i|real)}
\end{aligned}$$

If $\hat{P}(fake|w_1, w_2, ..., w_n) >= 0.5$, the headline was classified as fake and otherwise, real.

Note: since $\prod_{i=1}^{n} \hat{P}(w_i|real)$ and $\prod_{i=1}^{n} \hat{P}(w_i|fake)$ invloves computing products of a lot of really small numbers (which might result in underflow), the property that $a_1, a_2, ..., a_n = exp(loga_1 + loga_2 + ... + loga_n)$ was used to compute the product.

The values of m and $\hat{p}$ were determined using random search over the performance of validation set.

m is the number of examples to be included in the prior calculation. The more number of examples, the more $\hat{p}$ influences the final probability calculation. Values of m were tried on a logarithmic scale of $1, 10, 100, 1000$. Out of these, $m = 1$ gave the best result.

$\hat{p}$ is the prior probability of the word being real or fake. Values of $\hat{p}$ were tried in $0.1, 0.5, 0.7, 1$. Out of these $\hat{p} = 1$ gave the best performance.

Note: $\hat{p}$ was used as prior in calculation for both word being real and fake. This finding (low m and $\hat{p}$) seems to imply that our prior assumptions in this case are not very accurate and it was best to have prior influence as minimal as possible.

The final results were as follows:

1. Training Set: 97.28%
2. Validation Set: 83.46%
3. Testing Set: 85.68%

Part 3

Naive Bayes Algorithm: Indicative Words

Part 4

Logistic Regression

PyTorch framework was used to create a Logistic Regression model. This model is constructed and trained in `logistic_classifier.py`. The code is reproduced below.

```
# Model
class LogisticRegression(nn.Module):
    def __init__(self, input_size, num_classes):
        super(LogisticRegression, self).__init__()
        self.linear = nn.Linear(input_size, num_classes)

    def forward(self, x):
        out = self.linear(x)
        return out

def train_LR_model(training_set, training_label, validation_set, validation_label,
                                                    total_unique_words):
    """
    Trains Logistic Regression Numpy model

    PARAMETERS
    ----------
    training_set, validation_set: numpy arrays [num_examples, total_unique_words]
        For each headline in a set:
        v[k] = 1 if kth word appears in the headline else 0

    training_label, validation_label: numpy arrays [num_examples, [0, 1] or [1, 0]]
        [0, 1] if headline is fake else [1, 0]

    total_unique_words: int
        total number of unique words in training_set, validation_set, testing_set

    RETURNS
    -------
    model: LogisticRegression instance
        fully trained Logistic Regression model

    REQUIRES
    --------
    LogisticRegression: PyTorch class defined
    """
    # Hyper Parameters
    input_size = total_unique_words
    num_classes = 2
    num_epochs = 800
    learning_rate = 0.001
    reg_lambda = 0.01

    model = LogisticRegression(input_size, num_classes)

    x = Variable(torch.from_numpy(training_set), requires_grad=False).type(torch.FloatTensor)
    y_classes = Variable(torch.from_numpy(np.argmax(training_label, 1)), requires_grad=False).
                                                        type(torch.LongTensor)

    # Loss and Optimizer
    # Softmax is internally computed.
    # Set parameters to be updated.
    loss_fn = nn.CrossEntropyLoss()
```

```
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    # Training the Model
    for epoch in range(num_epochs+1):
        # Forward + Backward + Optimize
        optimizer.zero_grad()
        outputs = model(x)
        l2_reg = Variable(torch.FloatTensor(1), requires_grad=True)
        for W in model.parameters():
            l2_reg = l2_reg + W.norm(2)
        loss = loss_fn(outputs, y_classes) + reg_lambda*l2_reg
        loss.backward()
        optimizer.step()

        if epoch % 100 == 0:
            print("Epoch: " + str(epoch))

            # Training Performance
            x_train = Variable(torch.from_numpy(training_set), requires_grad=False).type(torch.
                                                            FloatTensor)
            y_pred = model(x_train).data.numpy()
            train_perf_i = (np.mean(np.argmax(y_pred, 1) == np.argmax(training_label, 1))) * 100
            print("Training Set Performance  : " + str(train_perf_i) + "%")

            # Validation Performance
            x_valid = Variable(torch.from_numpy(validation_set), requires_grad=False).type(torch.
                                                            FloatTensor)
            y_pred = model(x_valid).data.numpy()
            valid_perf_i = (np.mean(np.argmax(y_pred, 1) == np.argmax(validation_label, 1))) *
                                                            100
            print("Validation Set Performance:  " + str(valid_perf_i) + "%\n")

    return model
```

The final results were as follows:

1. Training Set: 98.46%
2. Validation Set: 82.04%
3. Testing Set: 84.86%

The learning curves are shown in Figure 1.

For regularization parameters, both $L1$ and $L2$ regularization were tried. $L2$ regularization performed 10% better. The λ values for $L2$ regularization was varied for values $0.001, 0.001, 0.05, 0.01, 0.1, 0.5$. Out of these, 0.01 performed the best.

Part 5

Part 6

Logistic Regression: Indicative Words

Part 6(a)

This part can be reproduced by calling `part6()`. The results are reproduced here:

```
Top 10 positive thetas (including stop-words):
1: trumps
2: tax
3: australia
4: tapping
5: says
6: debate
7: latest
8: hacking
9: business
10: asia


Top 10 negative thetas (including stop-words):
1: victory
2: breaking
3: information
4: elect
5: veterans
6: predicts
7: black
8: watch
9: d
10: won


Top 10 positive thetas (excluding stop-words):
1: trumps
2: tax
3: australia
4: tapping
5: says
6: debate
7: latest
8: hacking
9: business
10: asia


Top 10 negative thetas (excluding stop-words):
1: victory
2: breaking
```

```
3: information
4: elect
5: veterans
6: predicts
7: black
8: watch
9: d
10: won
```

Part 7

Part 8