CSC411: Assignment #3

Due on Monday, March 19, 2018

Chawla Dhruv, Maham Shama Saila

March 19, 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.

```python
# 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.

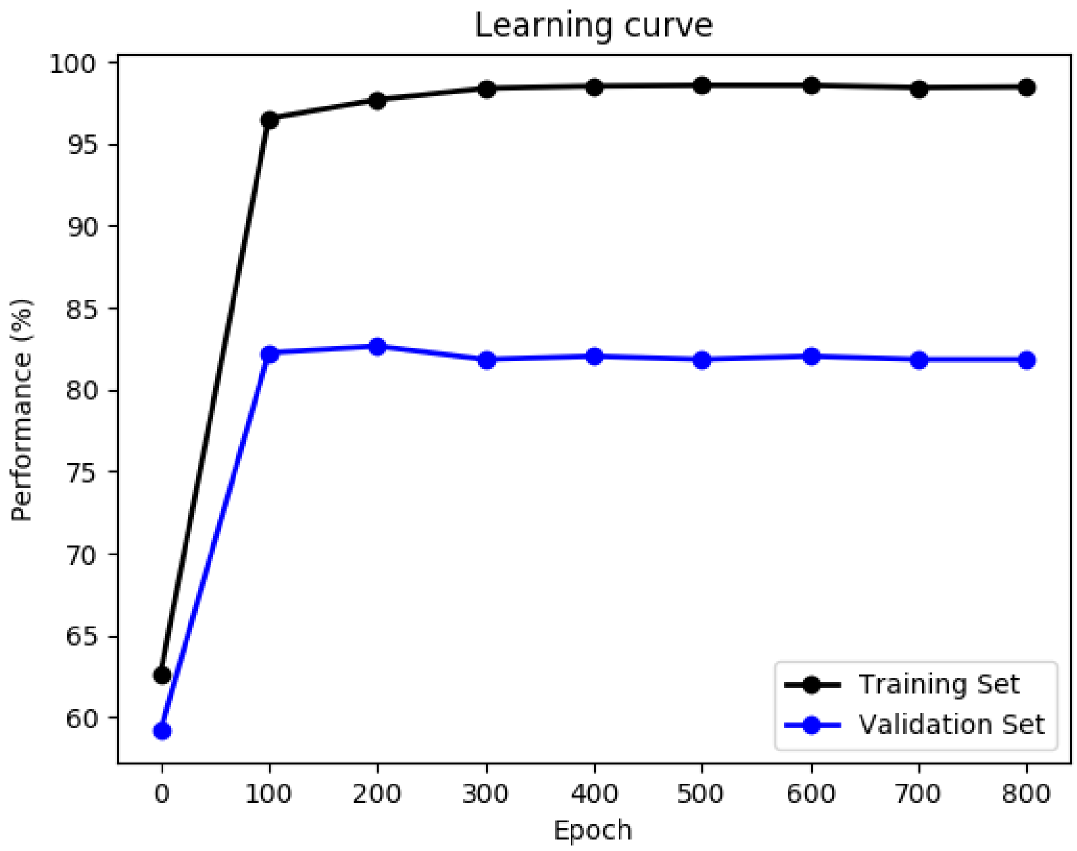

Figure 1: Logistic Regression Learning Curve

Part 5

Naive Bayes vs Logistic Regression

$$\theta_0 + \theta_1 I_1(x) + \theta_2 I_2(x) + \ldots + \theta_k I_k(x) > thr$$

k is the total number of unique words that appear in all headlines.
$I_i(x)$ is equal to 1 if x_i is present in the headline under consideration, 0 otherwise.

Naive Bayes
θ_i is equal to $log\frac{P(x_i=1|y=fake)}{P(x_i=1|y=real)} - log\frac{P(x_i=0|y=fake)}{P(x_i=0|y=real)}$

Logistic Regression
θ_i is equal to $max(log\frac{P(x_i=1|y=fake)}{P(x_i=1|y=real)}, log\frac{P(x_i=0|y=fake)}{P(x_i=0|y=real)})$

Part 6

Logistic Regression: Indicative Words

Part 6(a) and 6(b)

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 6(a)` does contain similar words to `Part 3(a)`. Words like `australia, business, asia` appear in `real_under_absence.txt` while words like `breaking, d` appear in `real_under_presence.txt`

The exlcusion of stopwords in both lists in Parts 6 and 3 have no effect on the top 10 lists.

Part 6(c)

Using magnitude of the logistic regression parameters to indicate importance of a feature can be a bad idea in general if features are not normalized.

Example: for a predictor of the selling price there are two factors: land area and width of the plot. Now, generally the land area is bigger in pure magnitude compared to the width of the plot, so unless the features are normalized, land area will influence the classifier more than the plot width.

However, it is reasonable to use logistic regression in this problem because each feature is either 0 or 1 so no one feature carries more importance than the rest. It's akin to the features already being normalized.

Part 7

Decision Trees

Part 7(a)

The relationship between between maximum depth of the tree and the training/validation accuracy can be seen below:

```
Depth: 2
Training Set Accuracy  : 70.6165282029
Validation Set Accuracy: 69.1836734694


Depth: 5
Training Set Accuracy  : 73.2837778749
Validation Set Accuracy: 67.5510204082


Depth: 10
Training Set Accuracy  : 78.1810231745
Validation Set Accuracy: 69.7959183673


Depth: 20
Training Set Accuracy  : 85.5268911237
Validation Set Accuracy: 71.4285714286


Depth: 50
Training Set Accuracy  : 95.2339309139
Validation Set Accuracy: 73.4693877551


Depth: 75
Training Set Accuracy  : 97.9011805859
Validation Set Accuracy: 73.8775510204


Depth: 100
Training Set Accuracy  : 99.0380411019
Validation Set Accuracy: 73.8775510204


Depth: 150
Training Set Accuracy  : 100.0
Validation Set Accuracy: 73.8775510204
```

```
Depth: 200
Training Set Accuracy  : 100.0
Validation Set Accuracy: 72.6530612245


Depth: 500
Training Set Accuracy  : 100.0
Validation Set Accuracy: 73.2653061224
```

As can be seen, with increasing depth, the performance increases at first, and then overfits.

The final testing accuracy (on $max\_depth = 150$ comes out to be 75.66%).

This can be reproduced by running `part7()` in `fake.py`.

In addition, we did not use any other parameters for which we are using non-default values, although non-default parameters were experimented with for `criterion, splitter,` and `max_features` and their performance measured on the validation set.

Part 7(b)

The first two layers of the tree are visualized here:

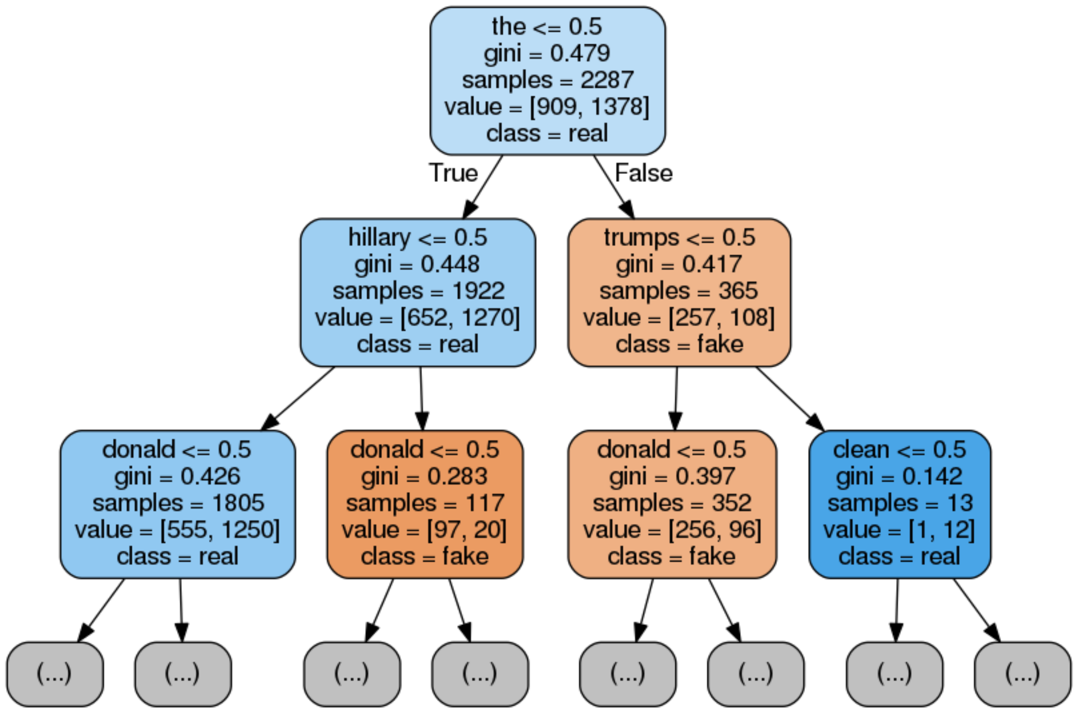

Figure 2: First two layers of the decision tree

This can be reproduced by running `part7()` in `fake.py`.

The only important feature common between the first two levels of the decision tree visualization and Parts 3 and 6 seems to be the word `trumps`. The lack of common words may be attributed to the fact that NB and Logistic Regression are classifying words important to classifying headlines as real or fake whereas the decision tree is objectified towards identifying a split with highest information split on either side of tree which may not directly correspond to the word as being indicative of one class or the other.

Part 7(c)

Method	Training Set Accuracy	Validation Set Accuracy	Testing Set Accuracy
Naive Bayes	97.28%	83.46%	85.68%
Logistic Regression	98.46%	82.04%	84.86%
Decision Tree	100.0%	73.87%	75.66%

The performance of the three classifiers is summarized in the table above. As can be seen, Naive Bayes and Logistic Regression had similar performances, while Decision Trees had the most overfitting.

Part 8