Report by Scharmensey Fine and Parker Blume

Exploring The Humor Behind Superbowl Commercials

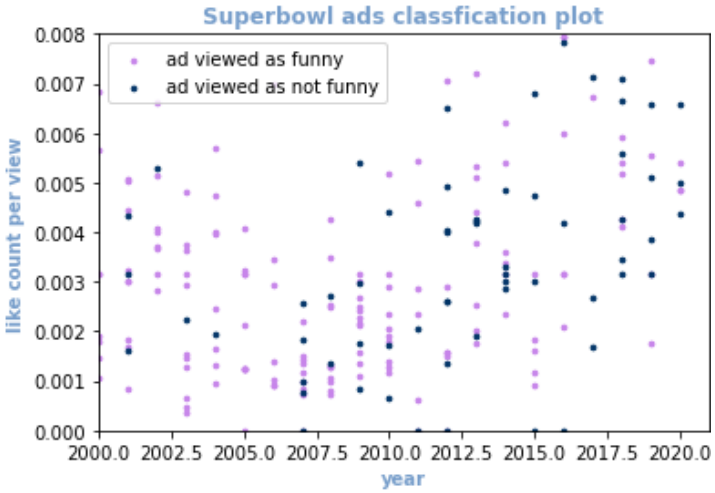
Introduction

The Superbowl is extremely popular for three things: the halftime show, the commercials, and of course the football game. However, fixating on its commercial aspect, its popularity is due to the viewer's high expectation of Super Bowl ads being funnier and more entertaining than regular ads. This brings a good reason to research & investigate methods in classifying which ads would appeal funny to a viewer. The dataset being used for this research is made up of data from Superbowl ads from the year 2000 to the year 2020.[1] The goal is to predict whether an ad will be found funny (the target variable), using two different machine learning algorithms and seeing which gives us the best results. The features in this dataset consist of 6 nominal predictors such as whether the ad shows the product quickly, uses sex, a celebrity, patriotism, danger, or animals. It also possesses some numeric predictors such as, dislike, like and comment counts from YouTube. These features will help identify the relations/patterns within the target, however we must find which methods will produce the most reliable results while minimizing the loss function.

Discussion

The two classification methods in examination are Linear Support Vector Machines (SVMs) & Neural Networks (NNs). Both excel highly in classification problems; however, both approach these problems differently. SVMs find an optimal hyperplane (decision boundaries) to accurately separate data points.[2] The labeling of these data points will be based on which side of the hyperplane they end up on. We will be using SGD as an optimizer, in which all batches of the full data are selected for each iteration to calculate the gradient and update the parameters. This selection introduces randomness into the optimization process.

Neural Networks use deep learning which operates much like the neurons in our brain. There is an input layer containing all features, following subsequent hidden layers, each neuron in the hidden layers receives inputs from the previous layer, performs a weighted sum of inputs (using weights and biases), and applies an activation function to the sum.[3] This is designed to learn complex patterns between the input data to distinctively classify data points in the output layer (the last layer).



Both algorithms can handle non-linear relationships and learn complex patterns/relationships. Although SVMs work effectively on both small & big datasets, NNs are preferred for bigger datasets as they can be computationally intensive and require substantial data for training, they're also superior in handling overlapping classes.[3] On the downside, overfitting can be an issue especially with smaller datasets, as SVMs tend to generalize well even with smaller datasets, primarily due to the margin maximization principle, which helps in reducing overfitting. SVMs also provide better interpretability on

1. Provided data: https://github.com/rfordatascience/tidytuesday/blob/master/data/2021/2021-03-02/youtube.csv
2. "Comparison of Support Vector Machine": https://arxiv.org/ftp/arxiv/papers/1007/1007.5133.pdf
3. "Designing Neural Networks": https://www.eecs.mit.edu/a-method-for-designing-neural-networks-optimally-suited-for-certain-tasks/

the significant features in predicting the outcome variables. Cons of SVMs are when using non-linear kernels or high-dimensional spaces, they can become memory-intensive, and they are also more affected by noisy data.[2]

Which algorithm would be better suited for our research purposes? Our dataset is limited which would make SVMs a better option, however, the target variable seems to have overlapping classes (figure 1) which NNs work better on. The only fear is that NNs could overfit the data since the number of observations is only below 300. The initial thought is that SVMs will give us more reliable results (less likely to overfit), but why completely rule out NNs when they may yield a higher accuracy score? We will build, train, and examine how both models interpret and predict the given data, making sure to tune their parameters to attain the best results possible.

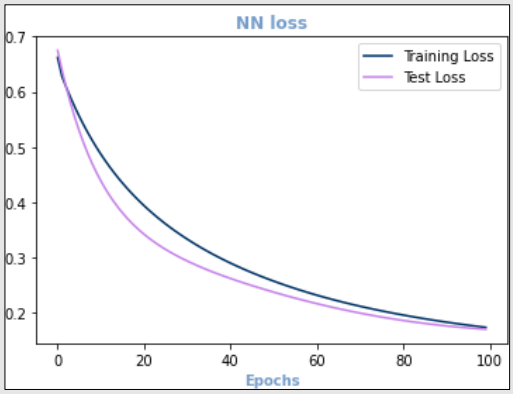
Implementation

```
        #----------------------------- SVM Implementation ----------------------------#
# The call to train the SVM (provided deeper in the code, brought up here for attention)
train_losses, test_losses, train_accs, test_accs = train_model_svm(n_iters=500)

def train_model_svm(n_iters):
        train_losses = []
        test_losses = []
        train_accs = []
        test_accs = []

        for i in range(n_iters):
              # partial fitting for SGD optimization
              model_svm.partial_fit(x_train, y_train, classes=np.unique(y_train))

              # Calculate train loss using the hinge function & accuracy
              train_loss = hinge_loss(y_train,
                                      model_svm.decision_function(x_train))
              train_acc = accuracy_score(y_train, model_svm.predict(x_train))

              # Calc the validation loss and acc
              test_loss = hinge_loss(y_test, model_svm.decision_function(x_test))
              test_acc = accuracy_score(y_test, model_svm.predict(x_test))

              # Add to our loss to keep track and plot later
              train_losses.append(train_loss)
              test_losses.append(test_loss)
              train_accs.append(train_acc)
              test_accs.append(test_acc)


        return train_losses, test_losses, train_accs, test_accs
```

SVM: This SVM was made with a built-in function definer where before-hand we find the best-parameters for tuning. In the training module it uses the hinge-loss function, and partial fitting to account for SGD optimization. We are using built-in functions for the SVM to do all the optimization and loss calculations for us during its training period.

```
#Tuning parameters with cross validation

model_svm = SGDClassifier(loss='hinge',learning_rate='optimal',random_state=0, **best_params)

grid_search = GridSearchCV(SGDClassifier(loss='hinge', random_state=0), param_grid=param_grid,
cv=5, n_jobs=-1)

grid_search.fit(x_train, y_train)
```

```
   #---------------------------- Neural Network Implementation ----------------------------#
# Defined Neural Network Module
class Net(nn.Module):

    def __init__(self, in_, width, out_):
        super().__init__()
        self.linear1 = nn.Linear(in_features= in_ , out_features = width, bias=True) #input layer
        self.relu = nn.ReLU() #Activation function
        self.dropout = nn.Dropout(p=0.2) # helps prevents overffitting (between 0.2-0.5)
        self.linear2 = nn.Linear(in_features=width,out_features = out_, bias=False) #output layer

    def forward(self,x):
        x = x.float()
        x = self.linear1(x)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.linear2(x)

        return x

# The Training Method
def train(net, X_train, Y_train, X_test, lr, epoch):
    # Loss function & SGD optimizer
    optimizer = optim.SGD(net.parameters(), lr=lr)
    loss_f = nn.BCEWithLogitsLoss() # logits for binary classification
    train_losses = []
    test_losses = []
    predictions = []

    for epoch in range(epoch):

        optimizer.zero_grad()
        output = net(X_train).float()
        loss = loss_f(output, Y_train.float().view(-1,1))
        loss.backward() #puts gradient in original W1
        optimizer.step()

        train_losses.append(loss.item())

        # Evaluate model on the test set
        with torch.no_grad():
            net.eval()
            test_output = net(X_test)
            test_loss = loss_f(test_output, Y_test.float().view(-1,1))
            test_losses.append(test_loss.item())

            test_predictions = (torch.sigmoid(test_output) > 0.5).float().numpy()
            predictions.append(test_predictions)

    return net, train_losses, test_losses, predictions

# Define hyperparameters and initialize our neural network
in_ = features.shape[1] #the number of features will be the number of neurons in the input layer
width = 25 # Num of neurons in the hidden layer
out_ = 1 # Binary classification
lr = 0.007 #Learning rate
epochs = 105 #number of grad descent steps

#Model
model_net = Net(in_, width, out_) # the neural network
```



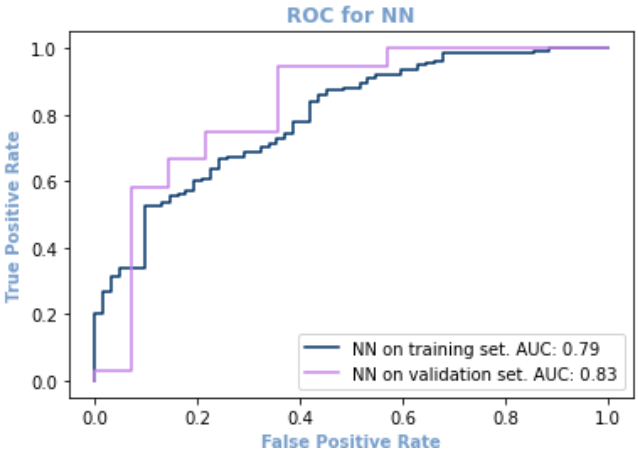
Results

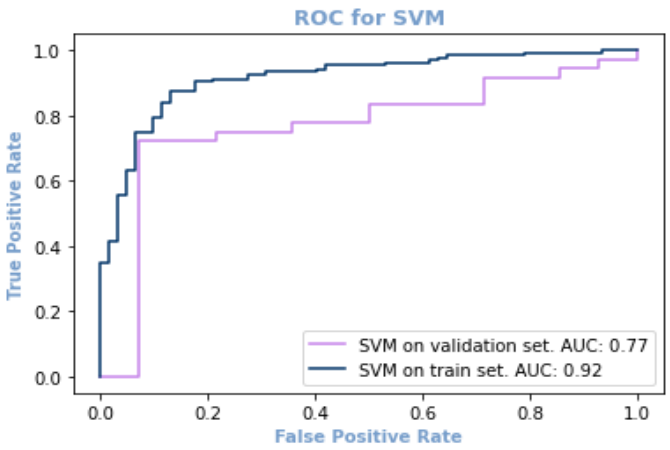
Beginning with the SVM approach, coming right out of the package of built-in libraries provided by sklearn, it was consistent at maintaining a decent accuracy. All it took was cross validation to simply find the best parameters to tune the SVM and define our own training method that uses SGD to optimize for the best results. After making predictions with the trained model, it came out to be fairly accurate, with acceptable metrics shown here:

```
~~ Metrics for SVM: ~~
   Accuracy: 0.72
   Recall: 0.81
   Precision: 0.81
   F1-Score: 0.81
----------------------
~~ Metrics for NN: ~~
   Accuracy:  0.74
   Recall: 1.00
   Precision: 0.72
   F1-Score: 0.84
```

The Neural Network was extremely unstable at first, its accuracy & AUC changing after every run showing extreme signs of overfitting. But through hyperparameter tuning, the optimal consistent performance was achieved with epochs between 95 and 105, around 25 to 30 neurons in the hidden layer (width), and a learning rate from 0.007 to 0.01 (However, in most cases it seemed the learning rate at 0.01 provided the most consistent results). After predicting funny ads with the trained model, the average accuracy averaged 72%-78% (with a wild card of 68%).

Comparing both algorithm's metrics, The Neural Network tended to accurately classify the funny ads consistently better than the SVMs, other than that, they share somewhat similar performances. However, observing their ROC curves, the SVM surprisingly overfitted the data far worse than the NN. The NN was far more complex and took more resources to train but came out being the most accurate. On the downside, there is no way of





Interpreting which features have relevance in ads being considered funny by viewers. SVMs have the upper hand here, and with the trained SVM model it found whether an ad contained danger, the year the ad came out, patriotic content, featured animals, the brand type, & number of dislikes to all be respectively significant in predicting an ad to be funny. The number of dislikes, patriotic content, & year made all have a negative relation with ads being found funny contradicting each other, while the rest have positive relations. The only exception was the brand type, as it's the only categorical, some brands were found to have positive associations while others have negative associations with an ad being funny.

Conclusion

Taking everything into account, for a fair comparison between the two in making predictions on ads, the Neural Network needed to be stable which took more time & resources to balance the parameters to successfully compare with the SVM. Because of such complexity, there were previous concerns about the Neural Network overfitting on a smaller dataset, but the opposite ended up happening. The SVM overfitted the data, and this could have to do with trying to differentiate between the excessive overlapping classes within the training process. Although the SVM still showed promising results for predicting ads to be funny or not, the Neural Network provided the best results for predicting, showing the significance in hyperparameter tuning in the training process (which will always vary depending on the task at hand). However, if wanting to explore the reasoning behind why an ad would be classified as funny or not, SVMs would be a better option as Neural Networks are black box and don't have this information available. Overall, we can conclude that using a Neural Network or SVM to predict the classification of Super Bowl ads being funny or not, have their trade-offs, the NN was more complex but came out more accurate with less overfitting, but we can't get any interpretability out of the model, such as we did with the SVM. The best model would be dependent on the researchers needs and resources.

References

1. Lee, Ming-Chang, and Chang To. "Comparison of Support Vector Machine and Back Propagation Neural Network in Evaluating the Enterprise Financial Distress." arXiv.org e-Print archive, July 2010. https://arxiv.org/.
2. Srikumar, Vivek. n.d. "Support Vector Machines: Training with Stochastic Gradient Descent the Slides Are Mainly from Vivek Srikumar." https://users.cs.utah.edu/~zhe/pdf/lec-19-2-svm-sgd-upload.pdf.
3. Zewe, Adam. 2023. "A Method for Designing Neural Networks Optimally Suited for Certain Tasks – MIT EECS." Www.eecs.mit.edu. March 31, 2023. https://www.eecs.mit.edu/a-method-for-designing-neural-networks-optimally-suited-for-certain-tasks/.