CS188 - Homework 3

Homework Partners:

- Gautam Nambiar: 004-933-460
- Stewart Dulaney: 904-064-791

2. Overfitting

Overfitting is a common problem when doing data science work.

A. How can you tell if a model you have trained is overfitting?

The goal of a supervised machine learning model is to generalize well across the training dataset. Therefore, if the model performs very well on certain evaluation metrics (like accuracy and recall) on the training set, but performs very poorly on unseen data (validation and test data), then we can conclude that the model has overfit, that is, it has failed to generalize across the training data and learn the general ideas/trends in the dataset. Instead, it has learned trends specific to our training data that don't reflect the general trends across similar data of the same kind.

B. Why would it be bad to deploy a model that's been overfitted to your training data?

Since it doesn't learn general trends and has instead overfitted, if the underlying distribution of the data sent to the model for predictions/inference changes, the model will perform extremely poorly. This could have severe consequences depending on the use cases (and particularly in cases where machine learning is being used for disease prediction/detection) because the model will not perform as expected - in fact, its predictions will almost seem random. For example, if the model is learning to classify cats vs dogs, and the learns that all cats are white rather than the fact that cats generally have "whiskers", then if we show it an image of a brown cat, it will probably predict it as a dog rather than a cat. While this is a fairly trivial example, the consequences can be extremely severe in cases where decisions are actively being made purely based on a model's predictions.

C. Briefly describe how overfitting can occur.

It usually occurs when a model captures the noise of the data, that is, it has a low bias but high variance. Intuitively, overfitting occurs when the model or the algorithm fits the data too well and is often a result of an excessively complicated model to model the data. This can usually happen due to a variety of factors, such as too many parameters for the model to learn, training the model for too long when the loss starts to increase (especially in the case of ANN where loss can be tracked after each epoch), or when the underlying distribution of the train and test data are extremely different from each other, leading us to believe our model has overfit.

D. In lecture and discussion, we've discussed multiple methods for dealing with overfitting. One such technique was regularization. Please describe two different regularization techniques and explain how they help to mitigate overfitting.

Both of my examples of regularization are specific to neural networks since they are especially susceptible to overfitting.

1. **Dropout**: This method approximates training a large number of neural networks with different architectures in parallel. During training, some number of layer outputs are randomly ignored or "*dropped out*." with probability p. This has the effect of making the layer look-like and be treated-like a layer with a different number of nodes and connectivity to the prior layer. In effect, each update to a layer during training is performed with a different "*view*" of the configured layer. Dropout has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs. In other words, the result is that no particular node gets relied on heavily, and all the weights, in general, are optimized for different architectural patterns, resulting in a "generalized" adjustment of weights in the model.

2. **L2 Regularization**: Instead of using the loss function L directly, we'll add a regularization term (norm penalty) $\Omega(\theta)$ to the objective function. To minimize the regularized loss function, the algorithm should minimize both the original loss function plus the regularization term, which depends on the square of the weights. In effect, we're adding the constraints to the original loss function, such that the weights of the network don't grow too large. If the weights grow too large, the overall value of the regularized loss function would increase, meaning the network's training loss would be more. By adding the regularized term, we're fooling the model such that it won't drive the training error to zero, which in turn reduces the complexity of the model. Therefore, L2 regularization helps reduce the overfitting of data.

3. Overfitting Mitigations

For each of the below strategies, state whether or not it might help to mitigate overfitting and explain why.

A. Using a smaller training dataset

No, it will not help mitigate overfitting. If we already overfit our data, cutting down the number of rows only makes the problem easier, especially if the number of parameters stays constant. Therefore, a complex model will only learn more complex overfitted curves that will almost perfectly fit the training dataset but possibly will perform poorly over the test dataset.

B. Restricting the maximum value any parameter can take on.

Yes, it will help mitigate overfitting. If we allow a parameter to take on any values, we're effectively giving the model the power to approximate the data too well, that is, for example, to draw a very precise hyperplane around the data points. By restricting the values, the model cannot approximate as thoroughly, and therefore at some level, is forced to find the best "general" approximation.

C. Training your neural network for longer (more iterations).

No, it will not help mitigate overfitting. The more iterations we train it for, the more the weights get optimized to fit the particular training dataset, causing the model to overfitting more and more with each epoch. At some point, this may cause the test loss to start increasing after reaching a minimum, indicating the model is overfitting the training data.

D. Training a model with more parameters.

No, it will not help mitigate overfitting. While adding more parameters allows the model to approximately more complex relationships, it also lets the model fit the training data more tightly, resulting in a model with weights trained to be too specific to the training dataset itself. Increased parameters also let the model map more complex non-linearities (normally), which is what really causes it to overfit.

E. Randomly setting the outputs of 50% of the nodes in your neural network to zero.

Yes, this is the equivalent of 'dropouts' in neural networks (described above) - During training, some number of layer outputs are randomly ignored or "*dropped out*." with probability p. This has the effect of making the layer look-like and be

treated-like a layer with a different number of nodes and connectivity to the prior layer. In effect, each update to a layer during training is performed with a different "*view*" of the configured layer. Dropout has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs. In other words, the result is that no particular node gets relied on heavily, and all the weights, in general, are optimized for different architectural patterns, resulting in a "generalized" adjustment of weights in the model.

F. Incorporating additional sparse features into your model.

No, it will not help mitigate overfitting. This is because increasing the number of features typically results in an increase in the number of parameters required to process that feature, and as mentioned above, increasing the number of parameters leads to overfitting. Very simply put, simple models are the best models, and adding more sparse features increases the complexity of the model, giving it more than enough ways to fit your training data too perfectly.

G. Initializing your parameters randomly instead of to zero.

No, it will not help mitigate overfitting because initializing weights to zero results in a "linear" model. Therefore, initializing weights randomly actually increases the chance of overfitting because the model has more parameters to work with, and therefore can use higher-order polynomial models to fit the training dataset.

H. Training your model on a graphics processing unit or specialized accelerator chip instead of a CPU.

No, this does not help mitigate overfitting. Training on a GPU tends to increase the speed of training with no real consequence on the performance of the model itself. This is because a GPU has thousands of processing units that can do matrix multiplication (and often, tensor multiplication) much more quickly than a CPU's limited number of ALUs.

4. K-Nearest Neighbors

A. Why is it important to normalize your data when using the k-nearest neighbors algorithm (KNN)?

Suppose you had a dataset (m "examples" by n "features") and all but one feature dimension had values strictly between 0 and 1, while a single feature dimension had values that range from -1000000 to 1000000. When taking the euclidean distance between pairs of "examples", the values of the feature dimensions that range between 0 and 1 may become uninformative and the algorithm would essentially rely on the single dimension whose values are substantially larger. By normalizing the data, no single feature dominates the Euclidean distance calculation.

B. When doing k-nearest neighbors, an odd value for k is typically used. Why is this?

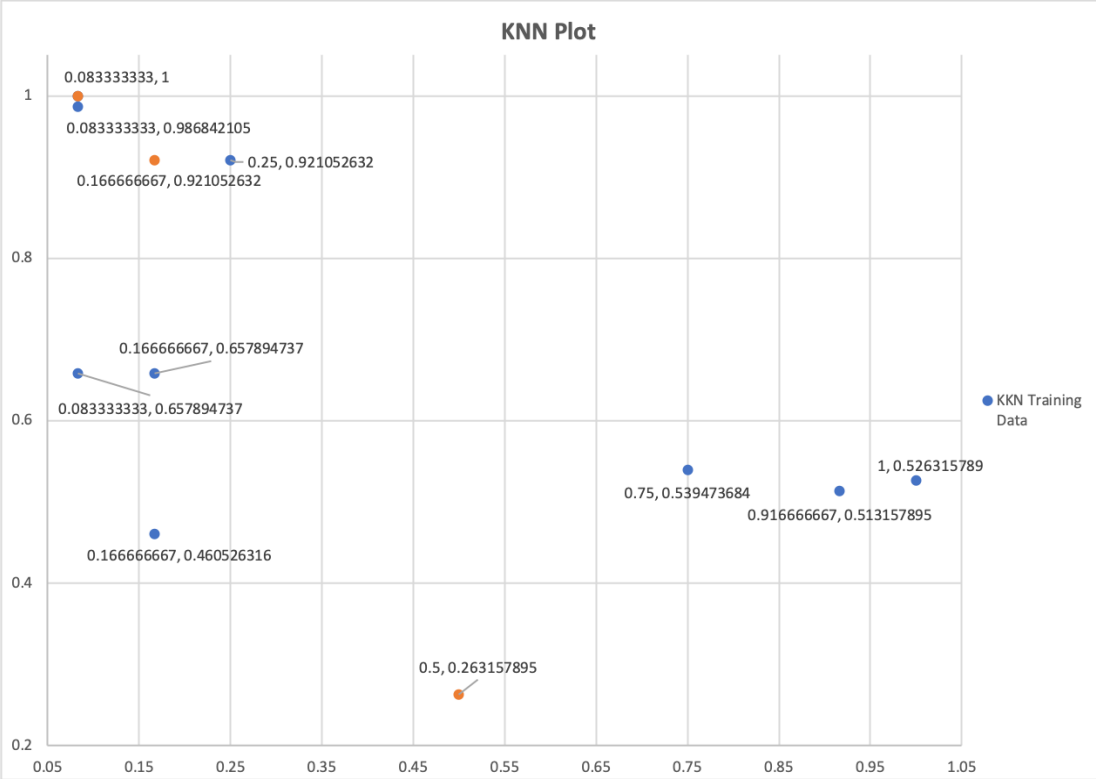
The k-nearest neighbor algorithm relies on majority voting based on class membership of 'k' nearest samples for a given test point. Since it relies on majority voting, picking an even k could result in no clear majority (Eg. k = 4, with 2 positive and 2 negative neighbors). Picking an odd value of k ensures there are no ties and always a clear majority.

C. Say you have the dataset in Table 1, where x and y are features and L is the label. Normalize the features by scaling them so that all values in a column lie in the range [0;1] so that they can be used with KNN.

x	y	L	x_normalized	y_normalized
0.2	350	0	0.166666667	0.460526316
0.1	750	0	0.083333333	0.986842105
0.3	700	0	0.25	0.921052632
0.1	500	0	0.083333333	0.657894737
0.2	500	0	0.166666667	0.657894737
1.2	400	1	1	0.526315789
0.9	410	1	0.75	0.539473684
1.1	390	1	0.916666667	0.513157895
0.1	760	1	0.083333333	1

Table 2: KNN Data for Training

x	y	x_normalized	y_normalized
0.1	760	0.083333333	1
0.2	700	0.166666667	0.921052632
0.6	200	0.5	0.263157895



D. Use KNN with K=3 to make predictions for the data points in Table 2.

x	y	x_normalized	y_normalized	Prediction
0.1	760	0.083333333	1	0
0.2	700	0.166666667	0.921052632	0
0.6	200	0.5	0.263157895	1

E. Use KNN with k = 1 to make predictions for the data points in Table 2.

x	y	x_normalized	y_normalized	Prediction
0.1	760	0.083333333	1	1
0.2	700	0.166666667	0.921052632	0
0.6	200	0.5	0.263157895	1

F. Use KNN with k = 5 to make predictions for the data points in Table 2.

x	y	x_normalized	y_normalized	Prediction
0.1	760	0.083333333	1	0
0.2	700	0.166666667	0.921052632	0
0.6	200	0.5	0.263157895	1

G. What is a potential issue with using a low k for KNN?

A small value of k means that noise will have a higher influence on the result. As a result, the predicted label is likely to not be accurate. Additionally, it does not factor a large enough localization area of the data itself, which creates inaccuracies in predictions since and is likely to lead to lucky predictions.

H. What is a potential issue with selecting a k that is too high when doing KNN?

While a large value for k could work, it would be computationally expensive. Additionally, it defeats the purpose of the algorithm itself, which is to find the class membership of 'k' nearest samples for a given test point. This might result in a very generalized localization area, which doesn't capture the true label well. In particular, it would not be good for edge cases like x=0.6, y=200 in the example above, which lie in between "clusters". We will get a smooth decision surface, where almost everything will fall within a single class resulting in a lot of potential misclassifications.

I. Say you had a dataset and wanted to understand how well KNN with a k of 3 performed on it. How could you quantify its performance? Assume you do not have access to any samples beyond those in your dataset.

We need to use K-fold Cross-validation. Specifically, for KNN with k=3, we would need a minimum of 4 samples in the training dataset so we could do 2-fold CV. Then, the performance can be quantified by fitting the model twice (with the 1st fold as train and the 2nd as test, then switching them), and summarizing the skill of the model using this sample of evaluation scores.

J. **Say you had a dataset consisting of 200 samples. How might you select the optimal k to use for KNN?**

Since we don't know the true labels for our validation data, I would use k-fold cross-validation on the training data, and probably set number of folds = 10. This would result in using 90% of the data for training and 10% for validation. We can repeat this for a variety of k values, and pick the k that maximizes our desired metric (could be accuracy, recall, etc.). Alternatively, it is common practice to set K = sqrt(N) for KNN's to run optimally. This would equal sqrt(200) ~= 15, so k=15.

5. Principal Component Analysis

For each of the below situations, state whether or not PCA would work well, and briefly explain why.

A. Data with a linear structure

Will work well, since PCA works optimally only in the situation where the correlations are linear, which is most of the time an approximation. The correlations will be linear if the data has a linear structure.

B. Data lying on a hyperbolic plane

Data lying on a hyperbolic plane will have a non-linear structure. Therefore, the correlations are also bound to be non-linear. Given that PCA works optimally only in the situation where the correlations are linear, PCA will not work well in this situation. This can be fixed by finding a way to transform the data into some linear approximation and then running PCA on top of it.

C. A dataset containing non-normalized features

Will not work well, since PCA works well on normalized features. In PCA we are interested in the components that maximize the variance. If one component (e.g. human height) varies less than another (e.g. weight) because of their respective scales (meters vs. kilos), PCA might determine that the direction of maximal variance more closely corresponds with the ‘weight’ axis, if those features are not scaled. As a change in height of one meter can be considered much more important than the change in weight of one kilogram, this is clearly incorrect.

D. A dataset where each feature is statistically independent of all others.

It will not work well (or is pointless), since the features are already independent of each other, and PCA essentially transforms dependent features into independent features.

6 Artificial Neural Networks

Consider the following computation graph for a simple neural network for binary classification. Here x1 and x2 are input features and y is their associated class. The network has multiple parameters, including weights w and biases b, as well as non-linearity function g. The network will output a value ypred, representing the probability that a sample belongs to class 1. We use a loss function Loss to help train our model. The network is initialized with the parameters in Table 3.

You will first train the model using some sample data points and then evaluate its performance. For any questions that ask for performance metrics, generate them using the samples in Table 4.

(a)

(Rest of the work is attached at the end of this document)

x1	x2	b0	b1	b2	w0	w1	w2	w3	w4	w5	a1	a2	a3	y_pred	y
0	0	1	-6	-3.93	3	4	6	5	2	4	0.7311	2.47E-03	0.07886	0	0
1	1	1	-6	-3.93	3	4	6	5	2	4	0.99967	0.9933	0.8852	1	1
0	1	1	-6	-3.93	3	4	6	5	2	4	0.9933	0.2689	0.2957	0	1

Last input prediction were wrong - here are the updated weights:

b0_new	b1_new	b2_new	w0_new	w1_new	w2_new	w3_new	w4_new	w5_new
1.00019523	-5.9884656	-3.9153321	3	4.00019523	6	5.01153438	2.01456958	4.00394419

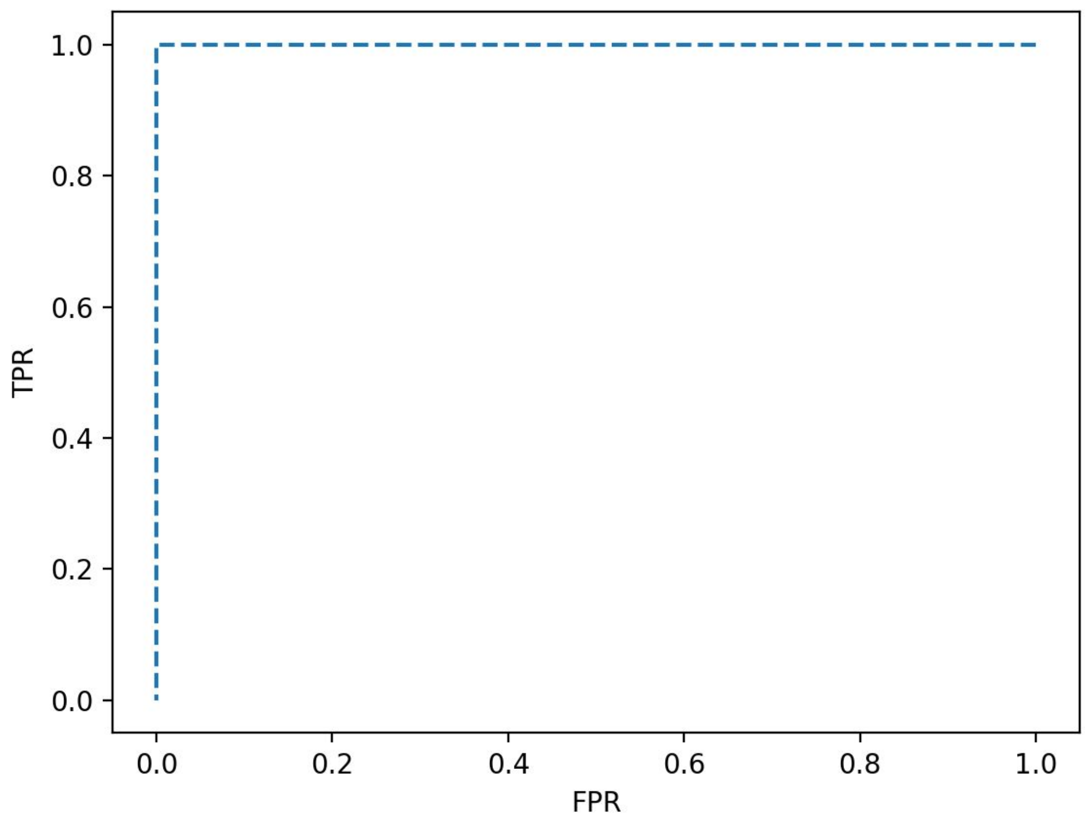
(b) Accuracy = TP+TN/FP+FN+TP+TN = ⅔ = 66.67%
(c) Precision = TP/TP+FP = 1/1+0 = 100%
(d) Recall = TP/TP+FN = ½ = 50%
(e) F1-Score = 2PR(P+R) = 2(100)(50)/(100+50) = 66.67%
(f) **I wrote code to generate the scores**:

```
from sklearn.metrics import confusion_matrix
import numpy as np
```

```python
weights = [0.07886, 0.8852, 0.2957]
true_labels = [0,1,1]

thresholds = [i for i in np.arange(0,1,0.01)]
print(thresholds)

tprs = []
fprs = []

for thresh in thresholds:
   test_labels = []
   for weight in weights:
       if weight > thresh:
           test_labels.append(1)
       else:
           test_labels.append(0)
   tn, fp, fn, tp = confusion_matrix(true_labels, test_labels).ravel()
   tpr = round(tp/(tp+fn),2)
   fpr = round(fp/(fp+tn),2)
   tprs.append(tpr)
   fprs.append(fpr)
   print("Threshold: {0}, TPR: {1}, FPR: {2}".format(thresh, tpr, fpr))

print(tprs)
print(fprs)

import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.plot(fprs, tprs, linestyle='--')
ax.set_xlabel('FPR')
ax.set_ylabel('TPR')
plt.show()
```

TPRS = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

FPRS = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]



(g) AUC-ROC - Area approximately equals 1.

8 (Money)ball So Hard

(a) Main Problem: Oakland A's mananger has a small budget but needs to find new players for the season after their best players are bought out. The scout's are trying to figure out who to pick based on intuition. Manager instead looks at standard metrics of moneyball theory which is numerical and uses different metrics. Problem change - from buying players to essentially "buying" wins. So, they need to by runs and in baseball, that means people who can make it one of the bases successfully. Analyze players using:

(i) Slugging Percentage (total bases divided by bats)

(ii) On-base percentage (the rate at which a batter gets on base)

Pick undervalued players based on these metrics.

End up doing well.

Steps to solve:

- Data Collection: Many websites have players stats and performance metrics. Can be used to further predict player performance. Can find APIs from 538 or other such websites, otherwise build a web scraper for the same.
- Data Cleaning: Make sure players don't have missing stats (otherwise find a meaningful way to impute them). Some players may also need to be dropped. Might want to drop very historic performance data since it does not transcend over time.
- Feature Engineering: Potential strong correlations between balls hit and strikes - Can use PCA to convert data into lower dimensions and such that they are independent. Can classify players into groups of values and bucket them into price ranges. Also try and replicate metrics like slugging percentage and on-base percentage.
- Machine Learning: Since it is numeric value we are trying to predict, we would use regression instead of classification. Perform some more EDA, determine the nature of the data, and try decision trees/gbm.
- Metrics: Would use RMSE to get the standard dev. of the prediction errors to determine how well the model was able to predict the important winning chars of a player (especially in the case of victories).

(b) Potentially, we may want to consider other advanced stats and metrics that we did not know about. May potentially have features that are too correlated in a way that they negatively affect the model. Need to make sure we're smart about what features we drop - may be smart to consult professionals. Can perform further

EDA with players, coaches etc. to potentially create a new, more refined “moneyball” theory, and therefore come up with a fresh set of more realistic metrics.