Topics For Today

• Supervised Learning Problems
• Naive Bayes
• Decision Trees and Random Forests

1 Supervised Learning

What makes a problem supervised? A supervised learning problem involves learning from pre-labeled data. That is, we are given data and their respective ground truth labels, then later asked to predict the labels of unseen data.

Some examples of supervised learning include:

• Predicting the integer value of handwritten digits
• Classifying Tumors with Array Data (looking at gene expression for tumor vs. non-tumor)
• Classifying whether a new protein is mitochondrial based on Targeting signal, protein domains, co-expression, homology, induction, motifs)

In practice, how do we develop a model for a predictive task?

1. Feature engineering - we need to generate problem-specific features that we expect to have predictive value over the ground-truth label
2. Training - we split our dataset into a training set and testing set, and train our model using the training set.

*Adapted from material by Connor Duffy, Tejas Sundaresan (2016), and Sitara Persad (2017)
3. Testing - we run our model on the test data and evaluate its performance

*For each of the above examples of supervised learning problems, think about what the features are, and what the labels might be.*

**Why do we evaluate the performance of our model on the test data?**

Consider the problem of fitting a polynomial in Figure 1. If we evaluate our model solely on the performance of the training data, we may decide the third model is best, whilst in practice it will not generalize well.

Significantly better performance on training data than test data is usually an indicator that your model has **overfit** the training data.

![Figure 1: Overfit](image)

On the other hand, poor performance on the training data may be an indicator that we may be **underfitting** the data (as in the first model of Figure 1). In this case, a better model (often a more complex model) is needed.

**Types of Supervised Learning Problems**

Fitting a polynomial function is an example of a **regression problem**, where we are trying to predict a continuous value as the output label. The other main type of supervised learning problem is **classification**, where we are trying to predict a category, or discrete label as the output label.

In general, we can think of a learning problem as the problem of trying to learn a function $f : X \to Y$ to make predictions $\hat{y} = f(x)$ which minimize the loss function $L(\hat{y}, y)$.

In addition to the generalized loss function, there are several metrics by which we can evaluate the performance of a model on a supervised learning problem.
1.1 Evaluating Models’ Performance

We will first describe methods of evaluating performance for classification problems:

1. Confusion Matrix

![Confusion Matrix Image](confusion_matrix.png)

Figure 2: Confusion matrix

2. Precision/Recall/Specificity/Sensitivity

![Evaluation Metrics Image](precisionetc.png)

Figure 3: Evaluation Metrics

3. AUC & AUROC - Area Under Curve & Area Under Receiver Operator Characteristic

![ROC Curve Image](receiver_operating_characteristic.png)

Figure 4: Receiver Operating Characteristic - Area Under the Curve

The ROC curve is a plot of the true positive rate versus the false positive rate for different thresholds of a probabilistic binary classifier. The higher the AUC score, the better the classifier.

We often prefer AUROC to simple accuracy, because the accuracy is strongly affected by how our data is distributed over classes.
For example, if we have a binary classification problem, and our dataset contains 90 positive examples and 10 negative examples, a classifier which always predicts positive can get 90% accuracy despite having learned nothing about the features of the data.

For evaluating regression problems, we typically use a loss function $L(y, \hat{y})$, which is a function of the true value, $y$ and the predicted value $\hat{y}$. Generally, the loss function for regression is a function of the residual, $r_i = y_i - \hat{y}_i$.

The most common error for regression is the mean squared error: $L(\hat{y}, y) = \sum_{(x,y)} (y - \hat{y})^2$

## 2 Naive Bayes

Recall (never forget!!) Bayes’ Rule:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

We can use Bayes’ Rule to describe the relationship between a set of features and the class those features belong to.

$$P(C|F) = \frac{P(F|C)P(C)}{P(F)}$$

In English, this means:

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{evidence}}$$

If we develop a model of the likelihood - $P(F|X)$ - and the prior - $P(C)$ - then we can predict the class of unseen data by computing $P(C|F)$ for all classes and predicting the class with the highest probability.

However, it is difficult to develop a model of $P(F|C)$. Recall that $F$ is actually a vector of features $F_1, ..., F_n$. Thus, we have to learn a joint distribution over all these features and the possible values they can take on.

Suppose we have 10 features which could each take on 10 different values. There are then $10^{10}$ possible features, and we would need to learn the joint distributions for all feature combinations.

This is not feasible because:

1. We might not have at least 10 billion data points to be able to calculate $P(F|C)$ - or we would have a lot of zeros.
2. This is not computationally feasible to do - storing all of these values requires a lot of memory.

Therefore, we make the **Naive Bayes assumption** that all of these features are independent.

Now, we can use the chain rule to expand:

\[
P(C|F_1, F_2, ..., F_n) = \frac{P(F_1|C) \times P(F_2|C) \times ... \times P(F_n|C) \times P(C)}{P(F_1)P(F_2)...P(F_n)}
\]

This greatly reduces the complexity of the problem from exponential to polynomial.

In order to train this model, we compute from the labelled data:
- The likelihoods \( P(F_i|C) \) based on frequency counts
- The prior \( P(C) \) by frequency

To test the model or apply it to unseen data points:
- Find \( C^* \) which maximizes \( P(C|F) \) and select \( C^* \) as the class label for that point.

**Example:**
See In-Class Exercise - Supervised Learning.

### 3 Decision Trees

A decision tree is a specific kind of tree where we make a **decision** about how to classify our data at each branching point (every time we split off into subtrees).

For example, suppose we wished to classify a nucleotide as A,T,C or G. We select of features which we think will have predictive power, for example:
- Is the nucleotide a purine or a pyrimidine?
- Does it form weak bonds or strong bonds?

We can use these features to construct a tree which correctly classifies all of our examples.
3.1 How to Build a Decision Tree

The generalized algorithm for constructing a decision tree is:

**Data:** Labelled examples with selected features  
**Result:** Decision tree which correctly classifies the examples  
**while all examples not correctly classified do**  
  Pick a decision criteria which *optimally splits data*;  
  Assign data to subtrees based on this split;  
  Recurse on subtrees;  
**end**

**Algorithm 1:** General approach to constructing decision trees.

The question then remains - what is an optimal split to the data? There are several metrics we can use to figure out which feature has the **best splitting power**. Two such metrics are:

1. Gini Index  
2. Average Disorder (Entropy)

Suppose we had a simple set of data \( \{x : y\} = \{1 : +1, 2 : +1, 3 : +1, 4 : -1, 5 : -1\} \). In this case, the feature with the best splitting power is \( x \) itself - the decision \( x < 4? \) splits the data exactly into the correct labels.

There is an example on how to construct a more complex decision tree using average disorder in the handout 'Supervised Learning - Decision Tree'.

---

Figure 5: A decision tree is a series of branching decisions, designed to separate members of different classes on different leaves of the branch.
3.2 Advantages of Decision Trees

1. White box - very interpretable. We can easily understand the rationale behind a classification by simply looking at the decisions used in the decision tree.

2. Feature selection - we can identify features that are important in splitting the data. These are the features near the top of the tree which have the best splitting power, and which are likely good predictors.

3. Overfit detection - we can identify features which might be associated with overfitting by looking at the features near the bottom of the tree.

3.3 Disadvantages of Decision Trees

As we have previously alluded to, decision trees have a tendency to overfit the data. Since the algorithm’s goal is to classify all examples correctly, the decision tree becomes very tailored to the training set. Such trees tend to be quite deep, using features that may not be generally useful for classifying the data.

Example: See In-Class Exercise - Supervised Learning

4 Random Forests

Random forests are an extension of decision tree classifiers which aim to provide more resistance to overfitting. Random forests typically have:

- More trees
- Shorter trees

There are many variations on random forest classifiers to suit different problems but, in general, the algorithm for constructing random forests from decision trees is as follows.

Decision forests are parametrized by:

- The number of trees in the forest, $T$
- The number of samples used to construct each tree, $N$
- The number of features used to construct each tree, $m$, out of a total possible $M$ features

Random forests are aggregate classifiers - to label an unseen data point, we give each tree in the forest a vote.
Select the number of trees, $T$;

for every decision tree do

    Choose $N$ samples (with replacement - bootstrap sample);
    Choose $m << M$ features to build tree;
    Construct decision tree;

end

Algorithm 2: General approach to constructing random forests from decision trees.

This vote is the class that decision tree outputs. Then, we use the class with the most votes as the label for that data point.

We can evaluate the performance of each decision tree by looking at how well it classifies data points not included in its sample of $N$ points. This is called the out of bag (OOB) error, since those data points were not included in the bag of points used to create the tree.

If we use a high value of $m$, then we get deep trees, which tend to overfit and which are all closely correlated. However, if we use $m$ too small, then we end up with a high OOB error and poor performance.