Comp 551: Applied Machine Learning

Shereen Elaidi

Winter 2020 Term

Contents

1 Week 1

1.1 Learning Objectives

2 Week 2

2.1 Learning Objectives

1. Linear Regression
    a) Linear model
    b) Evaluation Criteria
    c) How to find the best fit
    d) Geometric interpretation
2. Logistic Regression

a) Linear Classifier
b) Logistic regression – model
c) Logistic regression – loss function
d) Maximum likelihood view
e) Multi-class classification

2.2 Linear Regression

The whole idea behind linear regression is that we want our output to be an *linear* combination of the features. We don't care about the linearity of the features themselves; for all we care the features themselves can encode non-linear relations. The objective of **linear regression** is to learn a function of the form:

$$f_W(X) = w_0 + \sum_{j=1}^{m} w_j x_j \tag{1}$$

Where X is a matrix encodes our features (each row represents one feature) and W encodes the weights that we are supposed to learn. w_0 is the bias/offset term. In linear regression, the most common choice of error function is the **squared error**:

$$\text{Err}(W) := \sum_{i=1}^{n} (y_i - w^t x_i)^2$$

Some remarks on the squared error:

- We take the *squared* error since we want to take gradients. There are also other more statistical reasons behind why this is the most common choice of error function in regression.
- The other most common error is something called the **cross-entropy** error. A situation where we would want to use cross-entropy error over squared error is if the output is bounded; in this case, this is not the right thing to do. Why?
  - Least squared error assumes that the noise is Gaussian distributed. Gaussian distributions are NOT bounded. So, if we were trying to learn a Bernoulli model, in which the output is bounded, this would not be correct.

The **least-squared closed form solution** is what we obtain when we take the derivative of the error and set it equal to zero. The closed form solution is:

$$\hat{W} = (X^t X)^{-1} X^t Y \tag{2}$$

- The run time to compute $\hat{W}$ using the closed-form solution (2) is polynomial, which is quite reasonable.
- There are two primary failure modes of this way:
  1. $(X^t x)$ is a singular matrix (i.e., not invertible).
  2. A linear function is not the right fit.

We will now address these failure modes in more detail.

2.2.1 Failure Mode # 1: Avoiding Singularities

We require that X has full column-rank, since we are inverting matrices here. This means that *features cannot be linearly correlated with each other*. This would happen if our features are redundant; this redundancy would break the model. There are two cases in which this can occur:

1. The Weights are Not Uniquely defined:
   a) This can occur when *one feature is a linear function of the other*. The way you deal with this is by scrapping out redundant features.

b) A concrete example of this is if you had a model that included the number of clicks per day and the rate of clicks per hour in X.

2. The number of features (m) exceeds the number of training examples.
   a) In this case, you must either drop features or use *regularisation* (will discuss this later).
   b) Interesting from a ML perspective.

2.2.2 Failure Mode # 2: Linear Function is a Bad Fit

In this case, we would need to either:

1. Pick a better function.
2. Use more features.
3. Get more data

This failure mode leads into the idea of transforming features. Maybe we can preserve a linear model, but perhaps use non-linear bases?

2.3 Input Features for Linear Regression – Feature Engineering

Suppose that our original quantitative variables are $X_1, ..., X_n$. For various reasons, we may wish to take a transformation of variables.

1. We may want to take the logarithm $\log(X_i)$. This is useful when we are dealing with count data. These data points tend to come from what are called **heavy-tailed** distributions. If we do not transform these data points, the model may become confused.
2. We can also do **basis expansions**: suppose that instead of having X_m, we can set $X_{m+} = X_m^2$.
3. We can create **interaction terms**: this is useful when both terms are important. For example, we can set $X_{m+1} = X_i X_j$, which means we only care when both X_i and X_j are happening at the same time. Since this is a non-linear relation between X_i and X_j, this is okay since it will not create linear dependencies amongst our features.
4. Numeric coding of qualitative features or one-hot encoding.

Variable coding is not always straightforward. Suppose we have $X_{\text{cat}} = \{$ green, blue, yellow, red $\}$. A first naive way to one-hot encode this is to make four separate binary variables. However, this could lead to singularities; we actually only need three binary variables. What we need to do is arbitrarily choose one of the categories to be the "basis" category.

If our input is raw text, then we need to get creative:

1. Word counts?
2. Average word length?
3. Number of words per sentence?

All of the above are examples of **feature design**; which will be a recurrent theme throughout the course. In general, we should avoid categorical encoding such as $\{1, 2, 3, 4, ...\}$, as these can create artificial closeness between categorical features in feature space, which may mislead the model.

2.4 Overfitting & Generalisation

If we find a model that is able to perfectly fit the training data but does not generalise to new data, we are observing **overfitting**.

1. Very important problem in ML.
2. We want to minimise the **true** error. The only way we can do this is if we use the training set as a proxy for data that we don't have.
3. We can *formally express overfitting* as: suppose we compare hypotheses f_1 and f_1 and suppose that f_1 has a lower error on the training set. Then, if f_2 has a lower true error, then our model is overfitting.

With that in mind, the goal in ML is to develop models that **generalise** previously unseen data. *How do we evaluate this*?

1. We need to evaluate on held-out data. This introduces the concept of a **train/test/validate-split**:
   a) **Training set** → use this to fit the model (i.e., choose the best hypothesis in the class).
      i. Most of the data will be used for this.
   b) **Validation set** → compare different choices of feature selection. This also gives us a rough idea of how well our model is *actually* doing.
      i. This is like our "sandbox"
      ii. Use to compare different models during development, e.g.: should.I use a first- or second- order polynomial fit?
      iii. Also can be used to compare **hyper-parameters** (i.e., a parameter that is NOT learned but can impact the performance, such as the learning rate).
   c) **Test set** → we should only use this dataset to report our final accuracy. Do not use this dataset to train!

2.4.1 K-Fold Cross Validation

This is a very effective approach to see how the model is generalising. It is also a good technique when you have a small dataset, and so don't want to throw out a lot of data.

The way **k-fold cross validation** is as follows: consider k partitions of the training or non-test data. For each partition, train with $k-1$ subsets, and then validate on the kth. Repeat this k times. You then average the prediction error over the k rounds or folds. **Run-time remark**: this increases the computation by a factor of k.

A more extreme version of k-fold cross validation is **leave one out cross-validation**. Let $k = n$, the size of the training set. For each model or hyper-parameter setting: for n times, set aside one instance from the training set. Use all the other data points to find w (this is the optimisation step), and measure the prediction error on the held out datapoint. Then, average the prediction over all n subsets. **Run-time remark**: this gets pretty computationally expensive.

3 Week 3

3.1 Learning Objectives

1. Naive Bayes Classifier
   a) Generative vs. discriminative classifier
   b) Naive Bayes classifier – assumption
   c) Naive Bayes classifier – different design choices

4 Week 4

4.1 Learning Objects

1. Regularisation
   a) Overfitting and underfitting
   b) Regularisation (L1 and L2)
   c) MLE vs MAP estimator
   d) Bias and variance tradeoff
   e) Evaluation metrics and cross-validation

5 Week 5

5.1 Learning Objectives

1. Gradient Descent Methods
    a) Gradient Descent
    b) Stochastic Gradient Descent
    c) Method of momentum
    d) Sub-gradient
    e) Application to linear regression and classification
2. Evaluation
    a) Different types of error
    b) Common evaluation metrics
    c) Cross validation

6 Week 6

Topics covered: perceptron, max margin classifier, support vector machines, soft margin constraints, hinge loss, decision trees, greedy heuristic, entropy, mutual information, gini index, and overfitting.

6.1 Learning Objectives

1. Perceptron and Support Vector Machines
    a) The geometry of linear classification
    b) Perceptron learning algorithm
    c) Margin maximisation and support vectors
    d) Hinge loss and relation to logistic regression
2. Decision Trees
    a) Decision Trees: model, cost function, and how it is optimised.
    b) How to grow a tree and why you should prune.

6.2 Perceptron

The **perceptron** is the first machine learning algorithm. The assumption made by the perceptron is that given the data, there must be a hyperplane that separates one class from another. It additionally assumes that the data is binary, however we can later extend this to multi-class predictions. That is, for a binary classifier, all the points of one class lie on one side of a hyperplane, and the other points lie on the other side of a hyperplane. In high dimensional spaces, this almost always holds. This doesn't really happen often in lower-dimensional spaces. You can actually prove that this always holds in *infinite-dimensional* spaces.

In some sense, the perceptron is the opposite of the KNN. The KNN is very nice in low-dimensional spaces, because you don't need to deal with the "curse of dimensionality" and it's also much faster in lower-dimensional spaces. The perceptron is optimal for high-dimensional spaces. Moreover, KNN is not a linear classifier. For KNN, the decision boundaries are not necessarily linear (as we saw on the quiz); however, for the perceptron method it *must* be linear.

Assuming that such a hyperplane exists, the perceptron tries to find it. To mathematically define a hyperplane, you need a normal vector w and a bias term b:

$$\mathcal{H} := \{x \in \mathbb{R}^n \mid w^t x + b = 0\} \tag{3}$$

it has one less dimension than the ambient space. During test time, if you get an unknown point x, you classify it based on which side of the hyperplane it lies on. This is nice since computing which side of the hyperplane that the point lies on is always the same; all you need to do is compute $\text{sgn}(w^t x + b)$.

6.2.1 How can we find the hyperplane?

The way that we formalise that our label set is binary is by writing:

$$\mathcal{Y} = \{-1, +1\}$$

We have to learn two things: w and b. We want to learn only one thing, so assume that there is no offset. So, compute the following maps:

$$x_i \mapsto \begin{bmatrix} x_i \\ 1 \end{bmatrix}$$
$$w \mapsto \begin{bmatrix} w \\ b \end{bmatrix}$$

This is valid since

$$\left\langle \begin{bmatrix} x_i \\ 1 \end{bmatrix}, \begin{bmatrix} w \\ b \end{bmatrix} \right\rangle = w^t x + b$$

This transformation just absorbs b into the data. This is reflected in the new hyperplane:

$$\mathcal{H} := \{x \in \mathbb{R}^{n+1} \mid w^t x = 0\}$$

Geometrically, what we are doing is insisting that our hyperplane now goes through the origin. It will still be the same solution, just in a higher dimension.

6.2.2 The Algorithm

High-level intuition: Every time you get a point wrong, you adjust your hyperplane. Loop over the dataset. Once you make no more mistakes, you know you've found a hyperplane that separates the data, and then you stop.

Algorithm 1: Perceptron Algorithm

```
Result: Perceptron Algorithm
w = 0 (set weights to zero, this will get everything wrong);
while true do
    m = 0 (the counter of how many pts mis-classified);
    for ∀ (x, y) ∈ D do
        (checks which side of the hyperplane you are on:);
        if y w^t x ≤ 0 then
            (this is the case if and only if the point is wrong, since you want the signs to
             align.);
            w ← w^t y x ;
            (the above piece of code tries to reinforce the points if positive to make positive
             larger inner products, negative points have small inner products);
            m ← m + 1 ;
        end
        if m = 0 then
            (do this until you make a full pass over the dataset without getting anything
             wrong);
            break;
        end
    end
end
```

Some observations: the algorithm will take a long time if there is a lot of "wiggle room" between the data points (this is encoded by what is called the **margin size**, which will be discussed later), since it's easy to continue to offshoot the ideal hyperplane.

The **Perceptron Convergence Theorem** asserts that the algorithm is guaranteed to converge in a finite number of steps if our data is linearly separable. For a given x, the amount of times at which we get x incorrectly, encoded by k, is at most:

$$k \leq \frac{w^t x}{x^t x} \tag{4}$$