ML Notes

Jyotirmoy Banerjee

May 26, 2020

Dedicated to Ma and Baba

Preface

ML notes. May be some day this notes will transform into a book ....

Chapter 1

Introduction

Broadly, there are 3 types of Machine Learning Algorithms. **Supervised Learning:** This algorithm consist of a target/outcome variable (or dependent variable) which is to be predicted from a given set of predictors (independent variables). Using these set of variables, we generate a function that map inputs to desired outputs. The training process continues until the model achieves a desired level of accuracy on the training data. Examples of Supervised Learning: Regression, Decision Tree, Random Forest, KNN, Logistic Regression etc. **Unsupervised Learning:** In this algorithm, we do not have any target or outcome variable to predict / estimate. It is used for clustering population in different groups, which is widely used for segmenting customers in different groups for specific intervention. Examples of Unsupervised Learning: Apriori algorithm, K-means. **Reinforcement Learning:** Using this algorithm, the machine is trained to make specific decisions. It works this way: the machine is exposed to an environment where it trains itself continually using trial and error. This machine learns from past experience and tries to capture the best possible knowledge to make accurate business decisions. Example of Reinforcement Learning: Markov Decision Process.

1.1 Empirical risk minimization

Assume a non-negative real-valued loss function $\mathcal{L}(\hat{y}, y)$. The risk associated with hypothesis $f(x)$ is then defined as the expectation of the loss function:

$$\begin{aligned} R(f) &= E[\mathcal{L}(f(x), y)] \\ &= \int \mathcal{L}(f(x), y)\, dP(x, y) \end{aligned}$$

The goal of a learning algorithm is to find a hypothesis $\hat{f}$ among a fixed class of function $\mathcal{H}$ for which the risk $R(f)$ is minimal:

$$\hat{f} = \operatorname*{argmin}_{f \in \mathcal{H}} R(f)$$

The distribution $P(x, y)$ in usually unknown. However, we can compute an approximation, called empirical risk, by averaging the loss function on the training set:

$$R_{\text{emp}}(f) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(f(x_i), y_i)$$

The empirical risk minimization chooses a hypothesis $\hat{f}$ which minimizes the empirical risk:

$$\hat{f} = \operatorname*{argmin}_{f \in \mathcal{H}} R_{\text{emp}}(f)$$

1.2 Bias and variance

Let $y = f(x) + \epsilon$. We want to find the estimate $\hat{y}$, that approximates the *true function* f as well as possible, by means of some learning algorithm. Mean square error (MSE) for estimating f is given as:

$$\begin{aligned}
E[(f - \hat{y})^2] &= E[f^2 + \hat{y}^2 - 2f\hat{y}] \\
&= E[f^2] + E[\hat{y}^2] - E[2f\hat{y}] \\
&= \text{Var}[f] + E[f]^2 + \text{Var}[\hat{y}] + E[\hat{y}]^2 - E[2f\hat{y}] \\
&= \text{Var}[f] + \text{Var}[\hat{y}] + (E[f] - E[\hat{y}])^2 \\
&= \text{Var}[f] + (f - E[\hat{y}])^2 \\
&= \text{Var}[f] + \text{Bias}^2[f]
\end{aligned}$$

$$\begin{aligned}
\mathrm{Var}[x] &= E[(x-\mu)^2] \\
&= E[x^2] + E[E[x]^2] - E[2xE[x]] \\
&= E[x^2] - E[x]^2 \\
\mathrm{Var}[\hat{y}] &= 0 \\
E[f] &= f
\end{aligned}$$

1.3 Bayes theorem

The posterior probability can be written in the form as:

$$\begin{aligned}
p(x|y) &= \frac{p(y|x)p(x)}{p(y)} \\
\text{Posterior probability} &= \frac{\text{Likelihood} \times \text{Prior probability}}{\text{Evidence}} \\
&\propto \text{Likelihood} \times \text{Prior probability}
\end{aligned}$$

As per graph notation: $\mathrm{parent}(x) \rightarrow x \equiv p(x|\,\mathrm{parent}(x))$. The conditional probability of A given B, is usually written as $P(A|B)$, or sometimes $P_B(A)$, where A and B are random variables.

Chapter 2

Modelling

When creating a machine learning model, we are presented with design choices as to how to define your model architecture. Often times, we don't immediately know what the optimal feature set and model architecture should be, and thus we'd like to be able to explore a range of possibilities. Typically the following steps are explored before selecting a model:

- Feature engineering
- Feature selection
- Model selection

Parameters which define the model architecture are referred to as hyperparameters and thus this process of searching for the ideal model architecture is referred to as Hyperparameter tuning. In general, this process includes:

- Define a model
- Define the range of possible values for all hyperparameters
- Define a method for sampling hyperparameter values
- Define an evaluative criteria to judge the model
- Define a cross-validation method

The ultimate goal for any machine learning model is to learn from examples in such a manner that the model is capable of generalizing the learning to new instances which it has not yet seen. At a very basic level, you should train on a subset of your total dataset, holding out the remaining data for evaluation to gauge the model's ability to generalize - in other words, "how well will my model do on data which it hasn't directly learned from during training".

When you start exploring various model architectures (ie. different hyperparameter values), you also need a way to evaluate each model's ability to generalize to unseen data. However, if you use the testing data for this evaluation, you'll end up "fitting" the model architecture to the testing data - losing the ability to truly evaluate how the model performs on unseen data. This is sometimes referred to as "data leakage".

To mitigate this, we'll end up splitting the total dataset into three subsets: *training data*, *validation data*, and *testing data*. The introduction of a validation dataset allows us to evaluate the model on different data than it was trained on and select the best model architecture, while still holding out a subset of the data for the final evaluation at the end of our model development.

You can also leverage more advanced techniques such as *K-fold cross validation* in order to essentially combine training and validation data for both learning the model parameters and evaluating the model without introducing data leakage.

2.1 Feature selection

Feature selection have become the focus of much research in areas of application for which datasets with tens or hundreds of thousands of variables are available [2]. The 4 different automatic feature selection techniques are:

- **Univariate Selection:** Statistical tests can be used to select those features that have the strongest relationship with the output variable.

  The scikit-learn library provides the SelectKBest class that can be used with a suite of different statistical tests to select a specific number of features.

  The example below uses the chi squared (chi^2) statistical test for non-negative features to select 4 of the best features from the Pima Indians onset of diabetes dataset.

```
# Feature Extraction with Univariate Statistical Tests
(Chi-squared for classification)
import pandas
import numpy
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
# load data
```

```
url = "https://raw.githubusercontent.com/jbrownlee/Datasets/
    master/pima-indians-diabetes.data.csv"
names = ['preg','plas','pres','skin','test','mass','pedi','age','
    class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
# feature extraction
test = SelectKBest(score_func=chi2, k=4)
fit = test.fit(X, Y)
# summarize scores
numpy.set_printoptions(precision=3)
print(fit.scores_)
features = fit.transform(X)
# summarize selected features
print(features[0:5,:])
```

You can see the scores for each attribute and the 4 attributes chosen (those with the highest scores): plas, test, mass and age.

```
[  111.52    1411.887     17.605     53.108   2175.565    127.669
    5.393
   181.304]
[[ 148.      0.     33.6    50. ]
 [  85.      0.     26.6    31. ]
 [ 183.      0.     23.3    32. ]
 [  89.     94.     28.1    21. ]
 [ 137.    168.     43.1    33. ]]
```

- **Recursive Feature Elimination:** The Recursive Feature Elimination (or RFE) works by recursively removing attributes and building a model on those attributes that remain.

  It uses the model accuracy to identify which attributes (and combination of attributes) contribute the most to predicting the target attribute.

  You can learn more about the RFE class in the scikit-learn documentation.

The example below uses RFE with the logistic regression algorithm to select the top 3 features. The choice of algorithm does not matter too much as long as it is skillful and consistent.

```
# Feature Extraction with RFE
from pandas import read_csv
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression
# load data
url = "https://raw.githubusercontent.com/jbrownlee/Datasets/
    master/pima-indians-diabetes.data.csv"
names = ['preg','plas','pres','skin','test','mass','pedi','age','
    class']
dataframe = read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
# feature extraction
model = LogisticRegression()
rfe = RFE(model, 3)
fit = rfe.fit(X, Y)
print("Num Features: %d") % fit.n_features_
print("Selected Features: %s") % fit.support_
print("Feature Ranking: %s") % fit.ranking_
```

You can see that RFE chose the the top 3 features as preg, mass and pedi.

These are marked True in the `support_` array and marked with a choice "1" in the `ranking_` array.

```
Num Features: 3
Selected Features: [ True False False False False  True  True
    False]
Feature Ranking: [1 2 3 5 6 1 1 4]
```

- Principal Component Analysis

Principal Component Analysis (or PCA) uses linear algebra to transform the dataset into a compressed form.

Generally this is called a data reduction technique. A property of PCA is that you can choose the number of dimensions or principal component in the transformed result.

In the example below, we use PCA and select 3 principal components.

```
# Feature Extraction with PCA
import numpy
from pandas import read_csv
from sklearn.decomposition import PCA
# load data
url = "https://raw.githubusercontent.com/jbrownlee/Datasets/
    master/pima-indians-diabetes.data.csv"
names = ['preg','plas','pres','skin','test','mass','pedi','age','
    class']
dataframe = read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
# feature extraction
pca = PCA(n_components=3)
fit = pca.fit(X)
# summarize components
print("Explained Variance: %s") % fit.explained_variance_ratio_
print(fit.components_)
```

You can see that the transformed dataset (3 principal components) bare little resemblance to the source data.

```
Explained Variance: [ 0.88854663  0.06159078  0.02579012]
[[ -2.02176587e-03   9.78115765e-02   1.60930503e-02   6.07566861
    e-02
     9.93110844e-01   1.40108085e-02   5.37167919e-04  -3.56474430
        e-03]
 [   2.26488861e-02   9.72210040e-01   1.41909330e-01  -5.78614699
     e-02
```

```
  -9.46266913e-02   4.69729766e-02   8.16804621e-04   1.40168181
     e-01]
 [ -2.24649003e-02   1.43428710e-01  -9.22467192e-01  -3.07013055
    e-01
    2.09773019e-02  -1.32444542e-01  -6.39983017e-04  -1.25454310
      e-01]]
```

- Feature Importance Bagged decision trees like Random Forest and Extra Trees can be used to estimate the importance of features.

  In the example below we construct a `ExtraTreesClassifier` classifier for the Pima Indians onset of diabetes dataset. You can learn more about the `ExtraTreesClassifier` class in the scikit-learn API.

```
# Feature Importance with Extra Trees Classifier
from pandas import read_csv
from sklearn.ensemble import ExtraTreesClassifier
# load data
url = "https://raw.githubusercontent.com/jbrownlee/Datasets/
    master/pima-indians-diabetes.data.csv"
names = ['preg','plas','pres','skin','test','mass','pedi','age','
    class']
dataframe = read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
# feature extraction
model = ExtraTreesClassifier()
model.fit(X, Y)
print(model.feature_importances_)
```

  You can see that we are given an importance score for each attribute where the larger score the more important the attribute. The scores suggest at the importance of `plas`, `age` and `mass`.

```
[ 0.11070069  0.2213717   0.08824115  0.08068703
0.07281761  0.14548537 0.12654214  0.15415431]
```

2.2 Hyperparameters tuning

The best hyperparameters are usually impossible to determine ahead of time, and tuning a model is where machine learning turns from a science into trial-and-error based engineering. Let us discuss how to search for the optimal structure of a random forest classifier. Random forests are an ensemble model comprised of a collection of decision trees; when building such a model, two important hyperparameters to consider are:

- How many estimators (ie. decision trees) should I use?
- What should be the maximum allowable depth for each decision tree?

The hyperparameter tuning methods are:

- **Grid search:** Grid search is arguably the most basic hyperparameter tuning method. With this technique, we simply build a model for each possible combination of all of the hyperparameter values provided, evaluating each model, and selecting the architecture which produces the best results.

  For example, we would define a list of values to try for both `n_estimators` and `max_depth` and a grid search would build a model for each possible combination. Performing grid search over the defined hyperparameter space:

  ```
  n_estimators = [10, 50, 100, 200]
  max_depth = [3, 10, 20, 40]
  ```

  would yield the following models.

  ```
  RandomForestClassifier(n_estimators=10, max_depth=3)
  RandomForestClassifier(n_estimators=10, max_depth=10)
  RandomForestClassifier(n_estimators=10, max_depth=20)
  RandomForestClassifier(n_estimators=10, max_depth=40)

  RandomForestClassifier(n_estimators=50, max_depth=3)
  RandomForestClassifier(n_estimators=50, max_depth=10)
  RandomForestClassifier(n_estimators=50, max_depth=20)
  RandomForestClassifier(n_estimators=50, max_depth=40)

  RandomForestClassifier(n_estimators=100, max_depth=3)
  RandomForestClassifier(n_estimators=100, max_depth=10)
  ```

```
RandomForestClassifier(n_estimators=100, max_depth=20)
RandomForestClassifier(n_estimators=100, max_depth=40)

RandomForestClassifier(n_estimators=200, max_depth=3)
RandomForestClassifier(n_estimators=200, max_depth=10)
RandomForestClassifier(n_estimators=200, max_depth=20)
RandomForestClassifier(n_estimators=200, max_depth=40)
```

Each model would be fit to the training data and evaluated on the validation data. As you can see, this is an exhaustive sampling of the hyperparameter space and can be quite inefficient.

- **Random search:** Random search differs from grid search in that we longer provide a discrete set of values to explore for each hyperparameter; rather, we provide a statistical distribution for each hyperparameter from which values may be randomly sampled.

  We'll define a sampling distribution for each hyperparameter.

  ```
  from scipy.stats import expon as sp_expon
  from scipy.stats import randint as sp_randint
  n_estimators = sp_expon(scale=100)
  max_depth = sp_randint(1, 40)
  ```

  We can also define how many iterations we'd like to build when searching for the optimal model. For each iteration, the hyperparameter values of the model will be set by sampling the defined distributions above. The `scipy` distributions above may be sampled with the `rvs()` function - feel free to explore this in Python!

  One of the main theoretical backings to motivate the use of random search in place of grid search is the fact that for most cases, hyperparameters are not equally important.

  In the following example, we're searching over a hyperparameter space where the one hyperparameter has significantly more influence on optimizing the model score - the distributions shown on each axis represent the model's score. In each case, we're evaluating nine different models. The grid search strategy blatantly misses the optimal model and spends redundant time exploring the unimportant parameter. During this grid search, we isolated each hyperparameter and searched for the best possible value while holding all other hyperparameters constant. For cases where the hyperparameter being studied has little effect on the resulting model score, this results in wasted effort. Conversely, the random search has much improved exploratory power and can focus on finding the optimal value for the important hyperparameter.

- bandit methods

- Bayesian optimization - The previous two methods performed individual experiments building models with various hyperparameter values and recording the model performance for each. Because each experiment was performed in isolation, it's very easy to parallelize this process. However, because each experiment was performed in isolation, we're not able to use the information from one experiment to improve the next experiment. Bayesian optimization belongs to a class of *sequential model-based optimization* (SMBO) algorithms that allow for one to use the results of our previous iteration to improve our sampling method of the next experiment.

  We'll initially define a model constructed with hyperparameters λ which, after training, is scored v according to some evaluation metric. Next, we use the previously evaluated hyperparameter values to compute a posterior expectation of the hyperparameter space. We can then choose the optimal hyperparameter values according to this posterior expectation as our next model candidate. We iteratively repeat this process until converging to an optimum.

  Gaussian process is used to model the prior probability of model scores across the hyperparameter space. This model will essentially serve to use the hyperparameter values $\lambda_{1,\cdots,i}$ and corresponding scores $v_{1,\cdots,i}$ we have observed thus far to approximate a continuous score function over the hyperparameter space. This approximated function also includes the degree of certainty of our estimate, which we can use to identify the candidate hyperparameter values that would yield the largest expected improvement over the current score. The formulation for expected improvemenet is known as our acquisition function, which represents the posterior distribution of our score function across the hyperparameter space.

2.3 K-fold Cross Validation

The technique of cross validation is best explained by example using the most common method, K-fold cross validation. When we approach a machine learning problem, we make sure to split our data into a training and a testing set. In K-fold cross validation, we further split our training set into K number of subsets, called folds. We then iteratively fit the model K times, each time training the data on K-1 of the folds and evaluating on the Kth fold (called the validation data).

As an example, consider fitting a model with K = 5. The first iteration we train on the first four folds and evaluate on the fifth. The second time we train on the first, second, third, and fifth fold and evaluate on the fourth. We repeat this procedure 3 more times, each time evaluating on a different fold. At the very end of training, we average the performance on each of the folds to come up with final validation metrics for the model.

For hyperparameter tuning, we perform many iterations of the entire K-fold cross validation process, each time using different model settings. We then compare all of the models, select the best one, train it on the full training set, and then evaluate on the testing set.

2.4 Class imbalance

It is the problem in machine learning where the total number of a class of data (positive) is far less than the total number of another class of data (negative). This problem is extremely common in practice and can be observed in various disciplines including fraud detection, anomaly detection, medical diagnosis, oil spillage detection, facial recognition, etc.

Rare objects are often of great interest and great value. From a classification perspective, it is an extreme case of the class imbalance problem. Weiss et al. [7] in their article have discuss about mining rare classes.

Approach to handling Imbalanced Datasets -

- **Data Level approach -**
  - **Resampling Techniques -** Dealing with imbalanced datasets entails strategies such as balancing classes in the training data (data preprocessing) before providing the data as input to the machine learning algorithm using up-sample minority class or down-sample majority class. This is done in order to obtain approximately the same number of instances for both the classes.

    Up-sampling is the process of randomly duplicating observations from the minority class in order to reinforce its signal. There are several heuristics for doing so, but the most common way is to simply resample with replacement.

    Down-sampling involves randomly removing observations from the majority class to prevent its signal from dominating the learning algorithm. The most common heuristic for doing so is resampling without replacement.
  - **Data Augmentation -** Creating synthetic samples is a close cousin of up-sampling, and some people might categorize them together. For example, the SMOTE algorithm

is a method of resampling from the minority class while slightly perturbing feature values, thereby creating "new" samples.

- **Change Your Performance Metric** - For a general-purpose metric for classification, we recommend Area Under ROC Curve (AUROC). Intuitively, AUROC represents the likelihood of your model distinguishing observations from two classes. In other words, if you randomly select one observation from each class, what's the probability that your model will be able to "rank" them correctly?

- **Penalize Algorithms** - The next tactic is to use penalized learning algorithms that increase the cost of classification mistakes on the minority class.

- **Use Tree-Based Algorithms** - The final tactic we'll consider is using tree-based algorithms. Decision trees often perform well on imbalanced datasets because their hierarchical structure allows them to learn signals from both classes. In modern applied machine learning, tree ensembles (Random Forests, Gradient Boosted Trees, etc.) almost always outperform singular decision trees.

  Tree ensembles have become very popular because they perform extremely well on many real-world problems.

- **Anomaly Detection** - Anomaly detection, a.k.a. outlier detection, is for detecting outliers and rare events. Instead of building a classification model, you'd have a "profile" of a normal observation. If a new observation strays too far from that "normal profile", it would be flagged as an anomaly.

- **Learn only the Rare Class** - If we try to learn a set of classification rules for all classes, the rare classes may be largely ignored. One solution to this problem is to only learn classification rules that predict the rare class. One data mining system that utilizes this recognition-based approach is Hippo [3, 7]. Hippo uses a neural network and learns only from the positive (rare) examples, thus recognizing patterns amongst the positive examples, rather than differentiating between positive and negative examples.

2.5 Norm

L1 and L2 regularization add constraints to the optimization problem. The curve H_0 is the hypothesis. The solution to this system is the set of points where the H_0 meets the constraints.

Now, in the case of L2 regularization, in most cases, the the hypothesis is tangential to the $||w||_2$. The point of intersection has both x_1 and x_2 components. On the other hand, in L1, due to the nature of $||w||_1$, the viable solutions are limited to the corners, which are on one axis only - in the above case x_1. Value of $x_2 = 0$. This means that the solution has eliminated the role of x_2 leading to sparsity. Extend this to higher dimensions and you can see why L1 regularization leads to solutions to the optimization problem where many of the variables have value 0. In other words, L1 regularization leads to sparsity.

2.6 Loss functions for classification

1. Square loss: While more commonly used in regression, the square loss function can be re-written as a function $\phi(yf(\vec{x}))$ and utilized for classification. Defined as:

   $$V(f(\vec{x}), y) = (1 - yf(\vec{x}))^2$$

   the square loss function is both convex and smooth and matches the $0-1$ indicator function when $yf(\vec{x}) = 0$ and when $yf(\vec{x}) = 1$. However, the square loss function tends to penalize outliers excessively, leading to slower convergence rates (with regards to sample complexity) than for the logistic loss or hinge loss functions.

2. Hinge loss: The hinge loss function is defined as $V(f(\vec{x}), y) = \max(0, 1 - yf(\vec{x})) = |1 - yf(\vec{x})|_+$. The hinge loss provides a relatively tight, convex upper bound on the $0-1$ indicator function. Specifically, the hinge loss equals the $0-1$ indicator function when $\text{sgn}(f(\vec{x})) = y$ and $|yf(\vec{x})| \geq 1$. In addition, the empirical risk minimization of this loss is equivalent to the classical formulation for support vector machines (SVMs). Correctly classified points lying outside the margin boundaries of the support vectors are not penalized, whereas points within the margin boundaries or on the wrong side of the hyperplane are penalized in a linear fashion compared to their distance from the correct boundary.

3. Logistic loss: The logistic loss function is defined as

   $$V(f(\vec{x}), y) = \frac{1}{\ln 2} \ln(1 + e^{-yf(\vec{x})})$$

   This function displays a similar convergence rate to the hinge loss function, and since it is continuous, gradient descent methods can be utilized. However, the logistic loss function does not assign zero penalty to any points. Instead, functions that correctly classify points with high confidence (i.e., with high values of $|f(\vec{x})|$) are penalized less. This structure leads the logistic loss function to be sensitive to outliers in the data. The minimizer of $I[f]$ for the logistic loss function is

$$f^*_{\text{Logistic}} = \ln\left(\frac{p(1 \mid x)}{1 - p(1 \mid x)}\right).$$

4. Cross entropy loss: Using the alternative label convention $t = (1+y)/2$ so that $t \in \{0, 1\}$, the cross entropy loss is defined as

$$V(f(\vec{x}), t) = -t\ln(f(\vec{x})) - (1-t)\ln(1 - f(\vec{x}))$$

The cross entropy loss is closely related to the Kullback-Leibler divergence between the empirical distribution and the predicted distribution. This function is not naturally represented as a product of the true label and the predicted value, but is convex and can be minimized using stochastic gradient descent methods. The cross entropy loss is ubiquitous in modern deep neural networks.

2.7 Metrics

		True condition	
		Positive	Negative
Predicted condition	Positive	TP	FP (Type I error)
	Negative	FN (Type II error)	TN

$$\begin{aligned}
\text{precision} &= \frac{TP}{TP + FP} \\
\text{recall} &= \frac{TP}{TP + FN} \\
F1 &= \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \\
\text{Accuracy} &= \frac{TP + TN}{TP + TP + FP + FN}
\end{aligned}$$

Chapter 3

Logistic Regression

Logistic regression is a statistical method for analyzing a dataset in which there are one or more independent variables that determine an outcome. The outcome of a two class problem is measured with a dichotomous variable, in which there are only two possible outcomes 0 or 1. The expression/probabilities of the outcome of the two classes is given as:

$$\begin{aligned} P(G=1|X=x) &= \frac{\exp(\beta_0+\beta_1 x)}{\exp(\beta_0+\beta_1 x)+1} \\ &= \frac{1}{1+\exp(-(\beta_0+\beta_1 x))} \\ P(G=2|X=x) &= \frac{1}{\exp(\beta_0+\beta_1 x)+1} \end{aligned}$$

A simple calculation shows that -

$$\ln\left(\frac{p}{1-p}\right) = \beta_0+\beta_1 x$$

where $p = P(G=1|X=x)$ and $\text{logit}(p) = \ln\left(\frac{p}{1-p}\right)$. Extending this to a multi class problem, the logistic regression model arises from the desire to model the posterior probabilities of the K classes via linear functions in x, while at the same time ensuring that they sum to one and remain in [0, 1].

$$\begin{aligned} P(G=k|X=x) &= \frac{exp(\beta_{k0}+\beta_k^T x)}{1+\sum_{l=1}^{K-1} exp(\beta_{l0}+\beta_l^T x)}, \quad k=1,\cdots,K-1 \\ P(G=K|X=x) &= \frac{1}{1+\sum_{l=1}^{K-1} exp(\beta_{l0}+\beta_l^T x)} \end{aligned}$$

A simple calculation shows that model is specified in terms of $K-1$ log-odds or logit transforms and is of the form -

$$\begin{aligned}\log \frac{P(G=1|X=x)}{P(G=K|X=x)} &= \beta_{10} + \beta_1^T x \\ &\vdots \\ \log \frac{P(G=(K-1)|X=x)}{P(G=K|X=x)} &= \beta_{(K-1)0} + \beta_{K-1}^T x\end{aligned}$$

The log-likelihood for N observations is (more material is required) -

$$\mathcal{L}(\theta) = \sum_{i=1}^{N} \log p_{g_i}(x_i;\theta)$$

where $p_{g_i}(x_i;\theta) = P(G=k|X=x_i;\theta)$. For a two class problem g_i is encoded via a 0/1 response y_i, where $y_i = 1$ when $g_i = 1$, and $y_i = 0$ when $g_i = 2$. Let $p_1(x;\theta) = p(x;\theta)$ and $p_2(x;\theta) = 1 - p(x;\theta)$. The log-likelihood can be written as -

$$\begin{aligned}\mathcal{L}(\beta) &= \sum_{i=1}^{N} \Big\{ y_i \log p(x_i;\beta) + (1-y_i)\log(1-p(x_i;\beta)) \Big\} \\ &= \sum_{i=1}^{N} \Big\{ y_i \beta^T x_i - \log(1+e^{\beta^T x_i}) \Big\}\end{aligned}$$

Chapter 4

Frequent patterns mining

In medicine, comorbidity is the presence of one or more additional diseases or disorders co-occurring with a primary disease or disorder; in the countable sense of the term, a comorbidity is each additional disorder or disease. The additional disorder may be a behavioral or mental disorder. Frequent patterns mining approaches can be used to mine the co-morbidities from the patient records. Frequent patterns are item sets, sub sequences, or substructures that appear in a data set with frequency no less than a user-specified threshold.

Association rules - Association rules are if/then statements that help uncover relationships between seemingly unrelated data in an information repository. An example of an association rule would be 'If a customer buys a dozen eggs, he is 80% likely to also purchase milk'. An association rule has two parts, an antecedent (if) and a consequent (then). An antecedent is an item found in the data. A consequent is an item that is found in combination with the antecedent.

Association rules are created by analyzing data for frequent if/then patterns and using the criteria support and confidence to identify the most important relationships. Support is an indication of how frequently the items appear in the database. Confidence indicates the number of times the if/then statements have been found to be true.

$$X \Rightarrow Y$$
$$\text{Support} = \frac{\text{freq}(X, Y)}{N}$$
$$\text{Confidence} = \frac{\text{freq}(X, Y)}{\text{freq}(X)}$$

The Apriori algorithm - The Apriori algorithm is a technique of mining association rules from large databases. The algorithm takes advantage of the fact (downward closure lemma) that any subset of a frequent item set is also a frequent item set. The steps are as follows:

1. It uses a 'bottom up' approach, where frequent subsets are extended one item at a time (a step known as candidate generation). It generates candidate item sets of length k from item sets of length $k - 1$. The groups of candidates are tested against the data and candidates which have an infrequent sub pattern are pruned (based on support). The algorithm terminates when no further successful extensions are found. The candidate set contains all frequent k-length item sets.

2. Generate all non-empty subsets for each frequent item set. For every non-empty subset S of item set I, the output rule is $S \Rightarrow (I - S)$ is selected, if the confidence of the output rule is greater than a user defined threshold.

The algorithm uses breadth-first search and a Hash tree structure to count candidate item sets efficiently.

Data mining - Youtube

Chapter 5

Topic modelling

5.1 Non-negative matrix factorization

Non-negative matrix factorization (NMF), is a group of algorithms in multivariate analysis and linear algebra where a matrix V is factorized into two matrices W and H, with the property that all three matrices have no negative elements.

$$V = W * H$$

$$\text{Data Matrix}\,(n \times m) = \text{Basis Vectors}\,(n \times k) * \text{Coefficient Matrix}\,(k \times m)$$

The columns of W are the basis vectors. Each basis vector can be interpreted as a cluster. The memberships of items in these clusters is encoded by H. NMF is used for topic modelling, where the words and document matrix is factorized into words and topics basis vectors and topics and documents coefficient matrix:

$$[\text{words} \times \text{documents}] = [\text{words} \times \text{topics}] * [\text{topics} \times \text{documents}]$$

5.2 LDA

Chapter 6

Trees

6.1 Decision tree

Decision Trees are an important type of algorithm for predictive modelling machine learning. The classical decision tree algorithms have been around for decades and modern variations like random forest are among the most powerful techniques available. Decision tree algorithm is also known by it's more modern name CART which stands for Classification And Regression Trees. The representation for the CART model is a binary tree. Each root node represents a single input variable (x) and a split point on that variable (assuming the variable is numeric). The leaf nodes of the tree contain an output variable (y) which is used to make a prediction.

For example, given a dataset with two inputs (x) of height in centimetres and weight in kilograms the output of sex as male or female, is an example of a binary decision tree, see Figure 6.1. A learned binary tree is actually a partitioning of the input space. You can think of each input variable as a dimension on a p-dimensional space. The decision tree split this up into rectangles (when p=2 input variables) or some kind of hyper-rectangles with more inputs. New data is filtered through the tree and lands in one of the rectangles and the output value for

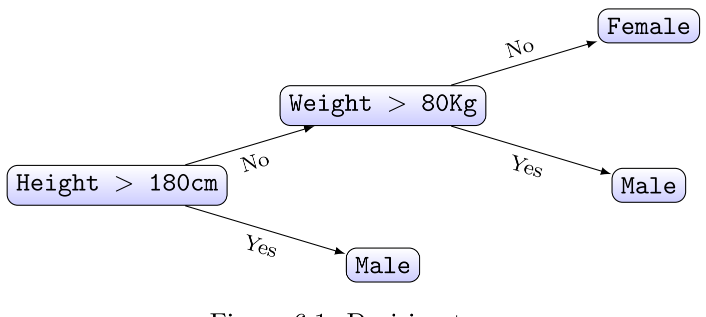


Figure 6.1: Decision tree

that rectangle is the prediction made by the model. This gives you some feeling for the type of decisions that a CART model is capable of making, e.g. boxy decision boundaries.

The selection of which input variable to use and the specific split or cut-point is chosen using a greedy algorithm to minimize a cost function. Tree construction ends using a predefined stopping criterion, such as a minimum number of training instances assigned to each leaf node of the tree.

6.1.1 Greedy splitting

Creating a binary decision tree is actually a process of dividing up the input space. A greedy approach is used to divide the space called recursive binary splitting. This is a numerical procedure where all the values are lined up and different split points are tried and tested using a cost function. The split with the best cost (lowest cost because we minimize cost) is selected. All input variables and all possible split points are evaluated and chosen in a greedy manner (e.g. the very best split point is chosen each time).

For regression predictive modelling problems the cost function that is minimized to choose split points is the sum squared error across all training samples that fall within the rectangle:

$$\sum (y - \text{prediction})^2$$

where y is the output for the training sample and prediction is the predicted output for the rectangle.

For classification the cost function provides an indication of how “pure” the leaf nodes are (how mixed the training data assigned to each node is). There are three commonly used impurity measures used in binary decision trees: Entropy, Gini index, and Classification Error.

$$\text{Entropy} = -\sum_j p_j \log p_j$$
$$\text{Gini} = 1 - \sum_j {p_j}^2$$
$$\text{Classification Error} = 1 - \max p_j$$

where p_j is the probability of class j. The entropy is 0 if all samples of a node belong to the same class, and the entropy is maximal if we have a uniform class distribution. In other words, the entropy of a node (consist of single class) is zero because the probability is 1 and $\log(1) = 0$. Entropy reaches maximum value (i.e. 1) when all classes in the node have equal probability.

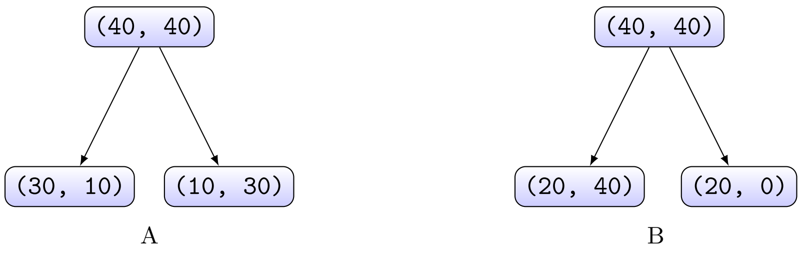


Figure 6.2: Information Gain - Example

Similar to entropy, the Gini index is maximal (i.e. 0.5) if the classes are perfectly mixed, for example, in a binary class.

We start at the tree root and split the data on the feature that results in the largest information gain. The Information Gain (IG) can be defined as follows:

$$IG(D_p) = I(D_p) - \frac{N_l}{N_p} I(D_l) - \frac{N_r}{N_p} I(D_r)$$

where I could be entropy, Gini index, or classification error, D_p, D_l, and D_r are the dataset of the parent, left and right child node.

As per the Information gain example in Figure 6.2 -

$$\begin{aligned}
\text{Classification Error} &= 1 - \max p_j \\
A : I_E(D_p) &= 1 - \frac{40}{80} = 0.5 \\
A : I_E(D_l) &= 1 - \frac{30}{40} = 0.25 \\
A : I_E(D_r) &= 1 - \frac{30}{40} = 0.25 \\
IG_E &= 0.5 - \frac{40}{80} * 0.25 - \frac{40}{80} * 0.25 = 0.25 \\
B : I_E(D_l) &= 1 - \frac{40}{60} = \frac{1}{3} \\
B : I_E(D_r) &= 1 - \frac{20}{20} = 0 \\
IG_E &= 0.5 - \frac{60}{80} * \frac{1}{3} - \frac{20}{80} * 0 = 0.25
\end{aligned}$$

The information gains using the classification error as a splitting criterion are the same (0.25) in both cases A and B.

$$
\begin{aligned}
\text{Gini index} &= 1 - \sum_j {p_j}^2 \\
A : I_E(D_p) &= 1 - \left(\left(\frac{40}{80}\right)^2 + \left(\frac{40}{80}\right)^2\right) = 0.5 \\
A : I_E(D_l) &= 1 - \left(\left(\frac{30}{40}\right)^2 + \left(\frac{10}{40}\right)^2\right) = 0.375 \\
A : I_E(D_r) &= 1 - \left(\left(\frac{10}{40}\right)^2 + \left(\frac{30}{40}\right)^2\right) = 0.375 \\
IG_G &= 0.5 - \frac{40}{80} * 0.25 - \frac{40}{80} * 0.25 = 0.25 \\
B : I_E(D_l) &= 1 - \left(\left(\frac{20}{60}\right)^2 + \left(\frac{40}{60}\right)^2\right) = 0.44 \\
B : I_E(D_r) &= 1 - \left(\left(\frac{20}{20}\right)^2 + \left(\frac{0}{20}\right)^2\right) = 0 \\
IG_G &= 0.5 - \frac{60}{80} * 0.44 - \frac{20}{80} * 0 = 0.17
\end{aligned}
$$

The information gains using the Gini index favors the split B.

$$
\begin{aligned}
\text{Entropy} &= -\sum_j p_j \log p_j \\
A : I_E(D_p) &= -\left(\left(\frac{40}{80}\right)\log\left(\frac{40}{80}\right) + \left(\frac{40}{80}\right)\log\left(\frac{40}{80}\right)\right) = 1 \\
A : I_E(D_l) &= -\left(\left(\frac{30}{40}\right)\log\left(\frac{30}{40}\right) + \left(\frac{10}{40}\right)\log\left(\frac{10}{40}\right)\right) = 0.81 \\
A : I_E(D_r) &= -\left(\left(\frac{10}{40}\right)\log\left(\frac{10}{40}\right) + \left(\frac{30}{40}\right)\log\left(\frac{30}{40}\right)\right) = 0.81 \\
IG_H &= 0.5 - \frac{40}{80} * 0.25 - \frac{40}{80} * 0.25 = 0.25 \\
B : I_E(D_l) &= -\left(\left(\frac{20}{60}\right)\log\left(\frac{20}{60}\right) + \left(\frac{40}{60}\right)\log\left(\frac{40}{60}\right)\right) = 0.92 \\
B : I_E(D_r) &= -\left(\left(\frac{20}{20}\right)\log\left(\frac{20}{20}\right) + \left(\frac{0}{20}\right)\log\left(\frac{0}{20}\right)\right) = 0 \\
IG_H &= 1 - \frac{60}{80} * 0.92 - \frac{20}{80} * 0 = 0.31
\end{aligned}
$$

The information gains using the entropy criterion favors B.

6.1.2 Stopping criterion

The recursive binary splitting procedure described above needs to know when to stop splitting as it works its way down the tree with the training data. The most common stopping procedure is to use a minimum count on the number of training instances assigned to each leaf node. If the count is less than some minimum then the split is not accepted and the node is taken as a final leaf node.

The count of training members is tuned to the dataset, e.g. 5 or 10. It defines how specific to the training data the tree will be. Too specific (e.g. a count of 1) and the tree will overfit the training data and likely have poor performance on the test set.

6.1.3 Pruning the tree

The stopping criterion is important as it strongly influences the performance of your tree. You can use pruning after learning your tree to further lift performance. The complexity of a decision tree is defined as the number of splits in the tree. Simpler trees are preferred. They are easy to understand (you can print them out and show them to subject matter experts), and they are less likely to overfit your data.

The fastest and simplest pruning method is to work through each leaf node in the tree and evaluate the effect of removing it using a hold-out test set. Leaf nodes are removed only if it results in a drop in the overall cost function on the entire test set. You stop removing nodes when no further improvements can be made.

More sophisticated pruning methods can be used such as cost complexity pruning (also called weakest link pruning) where a learning parameter (alpha) is used to weigh whether nodes can be removed based on the size of the sub-tree.

6.2 Bootstrapping

In machine learning, the *bootstrap* method refers to random sampling with replacement. This sample is referred to as a resample. This allows the model or algorithm to get a better understanding of the various biases, variances and features that exist in the resample. Taking a sample of the data allows the resample to contain different characteristics than it might have contained as a whole.

The reason to use the bootstrap method is because it can test the stability of a solution. By using multiple sample data sets and then testing multiple models, it can increase robustness. Perhaps one sample data set has a larger mean than another, or a different standard deviation.

This might break a model that was overfit, and not tested using data sets with different variations. Bootstrapping is used in both Bagging and Boosting.

6.3 Bagging

Bagging refers to bootstrap aggregation. Bagging predictors is a method for generating multiple versions of a predictor and using these to get an aggregated predictor. What Bagging does is help reduce variance from models that are might be very accurate, but only on the data they were trained on. This is also known as overfitting. Overfitting is when a function fits the data too well. Typically this is because the actual equation is much too complicated to take into account each data point and outlier. For example, decision tree like CART has high variance. Bagging gets around this by creating it's own variance amongst the data by sampling and replacing data while it tests multiple hypothesis (models). In turn, this reduces the noise by utilizing multiple samples that would most likely be made up of data with various attributes (median, average, etc).

Once each model has developed a hypothesis. The models use voting for classification or averaging for regression. This is where the 'Aggregating' in 'Bootstrap Aggregating' comes into play. Each hypothesis has the same weight as all the others. When we later discuss boosting, this is one of the places the two methodologies differ. Essentially, all these models run at the same time, and vote on which hypothesis is the most accurate. This helps to decrease variance i.e. reduce the overfit. Bagging is the application of the Bootstrap procedure to a high-variance machine learning algorithm, typically decision trees.

Let's assume we have a sample dataset of 1000 instances and we are using the CART algorithm. Bagging of the CART algorithm would work as follows:

1. Create many (e.g. 100) random sub-samples of our dataset with replacement.
2. Train a CART model on each sample.
3. Given a new dataset, calculate the average prediction from each model.

For example, if we had 5 bagged decision trees that made the following class predictions for a in input sample: blue, blue, red, blue and red, we would take the most frequent class and predict blue.

When bagging with decision trees, we are less concerned about individual trees overfitting the training data. For this reason and for efficiency, the individual decision trees are grown deep (e.g. few training samples at each leaf-node of the tree) and the trees are not pruned. These trees will have both high variance and low bias. These are important characterize of sub-models when combining predictions using bagging.

The only parameters when bagging decision trees is the number of samples and hence the number of trees to include. This can be chosen by increasing the number of trees on run after run until the accuracy begins to stop showing improvement (e.g. on a cross validation test harness). Very large numbers of models may take a long time to prepare, but will not overfit the training data. Just like the decision trees themselves, Bagging can be used for classification and regression problems.

6.3.1 Random forest

Random Forests are an improvement over bagged decision trees. A problem with decision trees like CART is that they are greedy. They choose which variable to split on using a greedy algorithm that minimizes error. As such, even with Bagging, the decision trees can have a lot of structural similarities and in turn have high correlation in their predictions.

Combining predictions from multiple models in ensembles works better if the predictions from the sub-models are uncorrelated or at best weakly correlated. Random forest changes the algorithm for the way that the sub-trees are learned so that the resulting predictions from all of the sub-trees have less correlation.

It is a simple tweak. In CART, when selecting a split point, the learning algorithm is allowed to look through all variables and all variable values in order to select the most optimal split-point. The random forest algorithm changes this procedure so that the learning algorithm is limited to a random sample of features of which to search.

The number of features that can be searched at each split point (m) must be specified as a parameter to the algorithm. You can try different values and tune it using cross validation.

1. For classification a good default is: $m = \sqrt{p}$
2. For regression a good default is: $m = p/3$

where m is the number of randomly selected features that can be searched at a split point and p is the number of input variables. For example, if a dataset had 25 input variables for a classification problem, then:

1. $m = \sqrt{25}$
2. $m = 5$

Hyperparameters

In the case of a random forest, hyperparameters include:

1. `n_estimators` = number of trees in the foreset
2. `max_features` = max number of features considered for splitting a node
3. `max_depth` = max number of levels in each decision tree
4. `min_samples_split` = min number of data points placed in a node before the node is split
5. `min_samples_leaf` = min number of data points allowed in a leaf node
6. `bootstrap` = method for sampling data points (with or without replacement)

6.4 Boosting

Boosting refers to a group of algorithms that utilize weighted averages to make weak learners into stronger learners. Unlike bagging that had each model run independently and then aggregate the outputs at the end without preference to any model. Boosting is all about 'teamwork'. Each model that runs, dictates what features the next model will focus on. Boosting also requires bootstrapping. However, there is another difference here. Unlike in bagging, boosting weights each sample of data. This means some samples will be run more often than others.

When boosting runs each model, it tracks which data samples are the most successful and which are not. The data sets with the most misclassified outputs are given heavier weights. These are considered to be data that have more complexity and requires more iterations to properly train the model. During the actual classification stage, there is also a difference in how boosting treats the models. In boosting, the model's error rates are kept track of because better models are given better weights. That way, when the 'voting' occurs, unlike in bagging, the models with better outcomes have a stronger pull on the final output.

Boosting and bagging are both great techniques to decrease variance. Ensemble methods generally out perform a single model. There are different reasons you would use one over the other. Bagging is great for decreasing variance when a model is overfit. However, boosting is much more likely to be a better pick of the two methods. Boosting also is much more likely to cause performance issues. It is also great for decreasing bias in an underfit model.

Boosting algorithms, such as AdaBoost, are iterative algorithms that place different weights on the training distribution each iteration. After each iteration boosting increases the weights associated with the incorrectly classified examples and decreases the weights associated with the correctly classified examples. This forces the learner to focus more on the incorrectly classified examples in the next iteration. Because rare classes/cases [7] are more error-prone than common classes/cases, it is reasonable to believe that boosting may improve their classification performance because, overall, it will increase the weights of the examples associated with these rare cases/classes. Note that because boosting effectively alters the distribution of the training data, one could consider it a type of advanced sampling technique.

6.4.1 Additive model

These models are fit by minimizing a loss function averaged over the training data, such as squared-error or a likelihood-based loss function,

$$\min_{\{\beta_m,\gamma_m\}} \sum_{i=1}^{N} L(y_i, f(x_i))$$
$$\text{where} \quad f(x) = \sum_{m=1}^{M} \beta_m \, b(x;\gamma_m)$$

where $\beta_m, m = 1, 2, \cdots, M$ are the expansion coefficients, and $b(x;\gamma) \in \mathbb{R}$ are usually simple functions of the multivariate argument x, characterized by a set of parameters γ.

The above function requires computationally intensive numerical optimization techniques. A simple alternative is to solve the subproblem of fitting just a single basis at a time. This is described in the algorithm called *forward stage-wise additive modelling.*

Algorithm 1 Forward stage-wise additive modelling

```
1. Initialize $f_0(x) = 0$.
2. For $m = 1$ to $M$:
    (a) Compute
        $(\beta_m, \gamma_m) = \operatorname{argmin}_{\beta,\gamma} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta \, b(x_i;\gamma))$
    (b) Set $f_m(x) = f_{m-1}(x) + \beta_m \, b(x;\gamma_m)$
```

6.4.2 AdaBoost

AdaBoost is equivalent to forward stage-wise additive modelling using the loss function:

$$L(y, f(x)) = \exp(-yf(x))$$

6.4.3 XGBoost

Chapter 7

Support vector machine

Hyperplane that partitions the two classes:

$$y = \vec{w}^T \vec{x} + b$$

Hyperplanes over the support vectors:

$$\vec{w}^T \vec{x} + b = +1$$
$$\vec{w}^T \vec{x} + b = -1$$

The best partition line is the line parallel and equidistant from the support vectors, where the distance between the support vectors is the maximum and the partition line separates the two classes well. Subtraction the above two lines:

$$\vec{w}^T (\vec{x_1} - \vec{x_2}) = +2$$
$$\frac{\vec{w}^T}{||\vec{w}||} (\vec{x_1} - \vec{x_2}) = \frac{2}{||\vec{w}||}$$

Distance between the support vectors is given as: $\frac{\vec{w}^T}{||\vec{w}||}(\vec{x_1} - \vec{x_2})$. Increasing the distance between them is equivalent to minimizing the term $\frac{2}{||\vec{w}||}$, i.e. maximizing the margins. The criteria that the partition line separates the two classes well is given as: $y(\vec{w}^T \vec{x}_j + b) \geq 1 \, \forall j$. As squaring preserves the order of the values (monotonic), maximizing $\frac{2}{||\vec{w}||}$ is equivalent to minimizing $\frac{||\vec{w}||^2}{2}$.

The resultant optimization problem is of the form:

$$\begin{aligned} \min_{\vec{w},b} \quad & \frac{1}{2}||\vec{w}||^2 \\ \text{where} \quad & (\vec{w}\cdot\vec{x}_j + b)y_j \geq 1, \quad \forall j \end{aligned}$$

The Lagrangian is:

$$L(\vec{w},\alpha) = \frac{1}{2}||\vec{w}||^2 - \sum_j \alpha_j[(\vec{w}\cdot\vec{x}_j + b)y_j - 1]$$

Our goal now is to solve:

$$\min_{\vec{w},b}\ \max_{\vec{\alpha}\geq 0} = \frac{1}{2}||\vec{w}||^2 - \sum_j \alpha_j[(\vec{w}\cdot\vec{x}_j + b)y_j - 1] \quad \text{(Primal)}$$

Swapping min and max, as the *Slater's condition* from convex optimization guarantees that these two optimization problems are equivalent.

$$\max_{\vec{\alpha}\geq 0}\ \min_{\vec{w},b} = \frac{1}{2}||\vec{w}||^2 - \sum_j \alpha_j[(\vec{w}\cdot\vec{x}_j + b)y_j - 1] \quad \text{(Dual)}$$

Let us solve for optimal $\vec{w}, b$ as a function of $\vec{\alpha}$:

$$\begin{aligned} \frac{\partial L}{\partial \vec{w}} &= \vec{w} - \sum_j \alpha_j y_j \vec{x}_j \quad \rightarrow \quad \vec{w} = \sum_j \alpha_j y_j \vec{x}_j \\ \frac{\partial L}{\partial b} &= -\sum_j \alpha_j y_j \quad \rightarrow \quad \sum_j \alpha_j y_j = 0 \end{aligned}$$

Substituting these values back in (and simplifying), we obtain:

$$\max_{\vec{\alpha}\geq 0, \sum_j \alpha_j y_j = 0} \sum_j \alpha_j - \frac{1}{2}\sum_{i,j} y_i y_j \alpha_i \alpha_j (\vec{x_i}\cdot\vec{x_j}) \quad \text{(Dual)}$$

Once α is estimated from the optimization function, we obtain $\vec{w}, b$. $\vec{w} = \sum_j \alpha_j y_j \vec{x}_j$ and $b = y_k - \vec{w}\cdot\vec{x}_k$, for any k where $\alpha_k > 0$. The steps used to obtain b are:

$$\begin{aligned} y_k(\vec{w}\cdot\vec{x}_k + b) &= 1 \\ y_k y_k(\vec{w}\cdot\vec{x}_k + b) &= y_k \\ \vec{w}\cdot\vec{x}_k + b &= y_k \end{aligned}$$

The classification rule is then $y \leftarrow \text{sign}(\vec{w} \cdot \vec{x} + b)$. Using the dual solution $y \leftarrow \text{sign}\big[\sum_i \alpha_i y_i (\vec{x_i} \cdot \vec{x}) + b\big]$, where $\vec{x}$ is the support vector.

7.0.1 Kernel trick

$$\max_{\vec{\alpha} \geq 0, \sum_j \alpha_j y_j = 0} \sum_j \alpha_j - \frac{1}{2} \sum_{i,j} y_i y_j \alpha_i \alpha_j K(\vec{x_i} \cdot \vec{x_j})$$

$$K(\vec{x_i} \cdot \vec{x_j}) = \Phi(\vec{x_i}) \cdot \Phi(\vec{x_j})$$

Chapter 8

Stochastic Gradient Descent

8.1 Convex function

A function f is convex if dom$f$ is a convex set and Jensen's inequality holds:

$$f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y) \quad \forall\, \theta \in [0,1]$$

First-order condition: for (continuously) differentiable f, Jensen's inequality can be replaced with:

$$f(y) \geq f(x) + \nabla f(x)^T(y-x) \quad \forall\, x, y \in \mathrm{dom}(f)$$

Proof: From the definition of the convex function we have -

$$\begin{aligned}
f(\theta x + (1-\theta)y) &\leq \theta f(x) + (1-\theta)f(y) \quad \forall\, \theta \in [0,1] \\
f(x + \theta(y-x)) &\leq f(x) + \theta(f(y) - f(x)) \\
f(y) - f(x) &\geq \frac{f(x+\theta(y-x)) - f(x)}{\theta} \\
f(y) - f(x) &\geq \frac{f(x+\theta(y-x)) - f(x)}{x + \theta(y-x) - x} \times (y-x) \\
f(y) - f(x) &\geq \nabla f(x)^T(y-x)
\end{aligned}$$

Second-order condition: for twice differentiable f, Jensen's inequality can be replaced with:

$$\nabla^2 f(x) \succeq 0 \quad \forall\, x \in \mathrm{dom}(f)$$

8.2 Gradient Descent

When f is convex

$$f(y) \geq f(x) + \nabla f(x)^T(y-x) \quad \forall\, x, y \in \mathrm{dom}(f)$$

Therefore, w close to $w^{(t)}$, we have $f(w) \approx f(w^{(t)}) + \langle (w - w^{(t)}), \nabla f(w^{(t)}) \rangle$. Hence we can minimize the approximation of $f(w)$. However, the approximation might become loose for w, which is far away from $w(t)$. Therefore, we would like to minimize jointly the distance between w and $w(t)$ and the approximation of f around $w(t)$. If the parameter η controls the tradeoff between the two terms, we obtain the update rule

$$w^{(t+1)} = \underset{w}{\operatorname{argmin}}\ \frac{1}{2}\left\|w - w^{(t)}\right\| + \eta\Big(f(w^{(t)}) + \langle w - w^{(t)}, \nabla f(w^{(t)})\rangle\Big)$$

Solving the preceding by taking the derivative with respect to w and comparing it to zero yields the update rule

$$w^{(t+1)} = w^t - \eta \nabla f(w^t)$$

Chapter 9

Neural network

- Epochs - One Epoch is when an ENTIRE dataset is passed forward and backward through the neural network only ONCE. Since, one epoch is too big to feed to the computer at once we divide it in several smaller batches.
- Batch Size - Total number of training examples present in a single batch.
- Iterations - Iterations is the number of batches needed to complete one epoch.

Let's say we have 2000 training examples that we are going to use. We can divide the dataset of 2000 examples into batches of 500 then it will take 4 iterations to complete 1 epoch.

- Stochastic Gradient Descent - Often abbreviated SGD, is a variation of the gradient descent algorithm that calculates the error and updates the model for each example in the training dataset.

  The update of the model for each training example means that stochastic gradient descent is often called an online machine learning algorithm.

- Batch gradient descent - It is a variation of the gradient descent algorithm that calculates the error for each example in the training dataset, but only updates the model after all training examples have been evaluated.

  One cycle through the entire training dataset is called a training epoch. Therefore, it is often said that batch gradient descent performs model updates at the end of each training epoch.

- Mini-batch gradient descent - It is a variation of the gradient descent algorithm that splits the training dataset into small batches that are used to calculate model error and update model coefficients.

Implementations may choose to sum the gradient over the mini-batch or take the average of the gradient which further reduces the variance of the gradient.

Mini-batch gradient descent seeks to find a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent. It is the most common implementation of gradient descent used in the field of deep learning.

Chapter 10

Convolutional neural network

Neural network equations:

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$$
$$a^{[l]} = g(z^{[l]})$$

Convolutional layer hyperparameter:

- $f^{[l]}$: filter size
- $p^{[l]}$: padding
- $s^{[l]}$: stride
- $\eta_c^{[l-1]}$: number of channels
- $\eta_c^{[l]}$: number of filters
- each filter is: $f^{[l]} \times f^{[l]} \times \eta_c^{[l-1]}$
- activation: $a^{[l]} \rightarrow \eta_h^{[l]} \times \eta_w^{[l]} \times \eta_c^{[l]}$
- weights: $f^{[l]} \times f^{[l]} \times \eta_c^{[l-1]} \times \eta_c^{[l]}$
- bias: $\eta_c^{[l]}$
- input: $\eta_h^{[l-1]} \times \eta_w^{[l-1]} \times \eta_c^{[l-1]}$
- output: $\eta_h^{[l]} \times \eta_w^{[l]} \times \eta_c^{[l]}$

where,

$$\eta_h^{[l]} = \left\lfloor \frac{\eta_h^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$
$$\eta_w^{[l]} = \left\lfloor \frac{\eta_w^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$

Pooling layer hyperparameter:

- f: filter size
- s: stride
- each filter is: $f^{[l]} \times f^{[l]} \times 1$
- input: $\eta_h^{[l-1]} \times \eta_w^{[l-1]} \times \eta_c^{[l-1]}$
- output: $\eta_h^{[l]} \times \eta_w^{[l]} \times \eta_c^{[l]}$

where,

$$\begin{aligned}
\eta_h^{[l]} &= \left\lfloor \frac{\eta_h^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor \\
\eta_w^{[l]} &= \left\lfloor \frac{\eta_w^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor \\
\eta_c^{[l]} &= \eta_c^{[l-1]}
\end{aligned}$$

Residual network has the following *short cut*: $a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$

10.1 Why convolutional layer?

The two main advantages are parameter sharing and sparsity of connection. Consider the following convolutional neural network:

$$\begin{aligned}
\text{Input} &: [\eta_h^{[l-1]} \times \eta_w^{[l-1]} \times \eta_c^{[l-1]}] \\
&= [32 \times 32 \times 3] \\
\text{Filter} &: [f^{[l]} \times f^{[l]} \times \eta_c^{[l-1]}] \times \eta_c^{[l]} \\
&= [5 \times 5 \times 3] \times 6 \\
\text{Bias} &: \eta_c^{[l]} \\
&= 6 \\
\text{Output} &: [\eta_h^{[l]} \times \eta_w^{[l]} \times \eta_c^{[l]}] \\
&= [28 \times 28 \times 6]
\end{aligned}$$

For the above network, the number of parameters of a convolutional neural network and a generic neural network are:

1. Parameters in a convolutional neural network

$$[f^{[l]} \times f^{[l]} \times \eta_c^{[l-1]}] \times \eta_c^{[l]} + \eta_c^{[l]}$$

$$[5 \times 5 \times 3] \times 6 + 6 = 456$$

2. Parameters in a neural network

$$[\eta_h^{[l-1]} \times \eta_w^{[l-1]} \times \eta_c^{[l-1]}] \times [\eta_h^{[l]} \times \eta_w^{[l]} \times \eta_c^{[l]}]$$
$$[32 \times 32 \times 3] \times [28 \times 28 \times 6] \approx 14\text{M}$$

The parameters of of a convolutional neural network are far less than a generic neural network.

10.2 Triplet loss

Triplet loss is used in the Siamese Neural Net (NN) for one shot learning. The Triplet loss tries to bring close the Anchor (current record) with the Positive (A record that is in theory similar with the Anchor) as far as possible from the Negative (A record that is different from the Anchor).

$$\begin{aligned}
\text{Step1}: &\quad ||f(A) - f(P)||^2 \leq ||f(A) - f(N)||^2 \\
&\quad \text{adding margin}\, \alpha \\
\text{Step2}: &\quad ||f(A) - f(P)||^2 + \alpha \leq ||f(A) - f(N)||^2 \\
\text{Triplet loss}: &\quad \mathcal{L}(A, P, N) = \max(||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + \alpha, 0)
\end{aligned}$$

Chapter 11

Sequence models

11.1 Recurrence neural network

Recurrence neural network equations:

$$z^{<t>} = W_{za}a^{<t-1>} + W_{zx}x^{<t>} + b_z$$
$$a^{<t>} = g_a(z^{<t>})$$
$$\hat{y}^{<t>} = g_y(W_{ya}a^{<t>} + b_y)$$

Simplified recurrence neural network notation:

$$a^{<t>} = g_a(W_a[a^{<t-1>}, x^{<t>}] + b_a)$$
$$\hat{y}^{<t>} = g_y(W_y a^{<t>} + b_y)$$

Types of recurrence neural network architecture

- one-to-one: Standard neural network
- one-to-many: Music/sequence generation
- many-to-one: Sentiment classification
- many-to-many: Name entity recognition
- many-to-many (encoder-decoder architecture): Machine translation

11.1.1 Vanishing or exploding gradient problem

Vanishing or exploding gradient is the problem where the derivative either decreases or grows exponentially as a function of the number of layers. The exploding gradient problem can be addressed by clipping the gradients. However, solving the vanishing gradient is challenging.

The vanishing gradient problem is a difficulty found in training artificial neural networks with gradient-based learning methods and backpropagation. In such methods, each of the neural network's weights receives an update proportional to the partial derivative of the error function with respect to the current weight in each iteration of training. The problem is that in some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training. As one example of the problem cause, traditional activation functions such as the hyperbolic tangent function have gradients in the range (0, 1), and backpropagation computes gradients by the chain rule. This has the effect of multiplying n of these small numbers to compute gradients of the "front" layers in an n-layer network, meaning that the gradient (error signal) decreases exponentially with n while the front layers train very slowly.

11.1.2 Gated recurrent unit

Gated recurrence unit (GRU) helps in long range interaction.

$$\begin{aligned}
\tilde{c}^{<t>} &= \tanh(W_c[c^{<t-1>}, x^{<t>}] + b_c) \\
\gamma_u &= \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u) \\
c^{<t>} &= \gamma_u * \tilde{c}^{<t>} + (1 - \gamma_u) * c^{<t-1>} \\
a^{<t>} &= c^{<t>} \\
\hat{y}^{<t>} &= g_y(W_y a^{<t>} + b_y)
\end{aligned}$$

where the subscript u stands for the update gate.

11.1.3 Long short-term memory

Long short-term memory (LSTM) helps in long range interaction.

$$
\begin{aligned}
\tilde{c}^{<t>} &= \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c) \\
\gamma_u &= \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u) \\
\gamma_f &= \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f) \\
\gamma_o &= \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_o) \\
c^{<t>} &= \gamma_u * \tilde{c}^{<t>} + \gamma_f * c^{<t-1>} \\
a^{<t>} &= \gamma_o * \tanh(c^{<t>}) \\
\hat{y}^{<t>} &= g_y(W_y a^{<t>} + b_y)
\end{aligned}
$$

where the subscripts u, f and o stands for the update, forget and output gates, respectively.

11.1.4 Bidirectional RNN

$$
\begin{aligned}
\overrightarrow{a}^{<t>} &= g_{\overrightarrow{a}}(W_{\overrightarrow{a}}[\overrightarrow{a}^{<t-1>}, x^{<t>}] + b_{\overrightarrow{a}}) \\
\overleftarrow{a}^{<t>} &= g_{\overleftarrow{a}}(W_{\overleftarrow{a}}[\overleftarrow{a}^{<t-1>}, x^{<t>}] + b_{\overleftarrow{a}}) \\
\hat{y}^{<t>} &= g_y(W_y[\overrightarrow{a}^{<t>}, \overleftarrow{a}^{<t>}] + b_y)
\end{aligned}
$$

Chapter 12

Regression

In linear regression we wish to fit a function (model) in this form:

$$\hat{Y} = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_n X_n$$

where X_i is the vector of features, and β_i are the coefficients we wish to learn.

Having more features may seem like a perfect way for improving the accuracy of our trained model (reducing the loss) - because the model that will be trained will be more flexible and will take into account more parameters. On the other hand, we need to be extremely careful about overfitting the data. As we know, every dataset has noisy samples. The inaccuracies can lead to a low-quality model if not trained carefully. The model might end up memorizing the noise instead of learning the trend of the data.

Because overfit is an extremely common issue in many machine learning problems, there are different approaches to solving it. The main concept behind avoiding overfit is simplifying the models as much as possible. Simple models do not (usually) overfit. On the other hand, we need to pay attention the to gentle trade-off between overfitting and underfitting a model.

There is a gentle trade-off between fitting the model, but not overfitting it. One of the most common mechanisms for avoiding overfit is called regularization. Regularized machine learning model, is a model that its loss function contains another element that should be minimized as well.

12.1 Ridge regression

Ridge regression is an extension for linear regression. It's basically a regularized linear regression model.

$$L = \sum(\hat{Y_i} - Y_i)^2 + \lambda \sum {\beta_i}^2$$

The second element though, a.k.a the regularization term, sums over squared β values and multiplies it by another parameter λ. The reason for doing that is to "punish" the loss function for high values of the coefficients β. By punishing the β values we add a constraint to minimize them as much as possible. The λ parameter is a scalar that is learned as well, using cross validation.

Ridge regression enforces the β coefficients to be lower, but it does not enforce them to be zero. That is, it will not get rid of irrelevant features but rather minimize their impact on the trained model.

12.2 Lasso

Lasso is another extension built on regularized linear regression, but with L1 norm. The loss function of Lasso is in the form:

$$L = \sum(\hat{Y_i} - Y_i)^2 + \lambda \sum |\beta_i|$$

The only difference from Ridge regression is that the regularization term is in absolute value. But this difference has a huge impact on the trade-off between bias and variance. Lasso method overcomes the disadvantage of Ridge regression by not only punishing high values of the coefficients β but actually setting them to zero if they are not relevant. Therefore, you might end up with fewer features included in the model than you started with, which is a huge advantage.

Chapter 13

Clustering

13.1 K-means clustering

The K-means clustering algorithm uses iterative refinement to produce a final result. The algorithm inputs are the number of clusters K and the data set. The data set is a collection of features for each data point. The algorithms starts with initial estimates for the K centroids, which can either be randomly generated or randomly selected from the data set. The algorithm then iterates between two steps:

1. Data assigment step: Each centroid defines one of the clusters. In this step, each data point is assigned to its nearest centroid, based on the squared Euclidean distance. More formally, if c_i is the collection of centroids in set C, then each data point x is assigned to a cluster based on

   $$\underset{c_i \in C}{\operatorname{argmin}} \operatorname{dist}(c_i, x)^2$$

   where dist(.) is the standard (L2) Euclidean distance. Let the set of data point assignments for each i^{th} cluster centroid be S_i.

2. Centroid update step: In this step, the centroids are recomputed. This is done by taking the mean of all data points assigned to that centroid's cluster.

   $$c_i = \frac{1}{|S_i|} \sum_{x_i \in S_i} x_i$$

The algorithm iterates between steps one and two until a stopping criteria is met (i.e., no data points change clusters, the sum of the distances is minimized, or some maximum number of iterations is reached). This algorithm is guaranteed to converge to a result. The result may be a local optimum (i.e. not necessarily the best possible outcome), meaning that assessing more than one run of the algorithm with randomized starting centroids may give a better outcome.

The algorithm described above finds the clusters and data set labels for a particular pre-chosen K. To find the number of clusters in the data, the user needs to run the K-means clustering algorithm for a range of K values and compare the results. In general, there is no method for determining exact value of K, but an accurate estimate can be obtained using the following techniques.

One of the metrics that is commonly used to compare results across different values of K is the mean distance between data points and their cluster centroid. Since increasing the number of clusters will always reduce the distance to data points, increasing K will always decrease this metric, to the extreme of reaching zero when K is the same as the number of data points. Thus, this metric cannot be used as the sole target. Instead, mean distance to the centroid as a function of K is plotted and the “elbow point”, where the rate of decrease sharply shifts, can be used to roughly determine K.

Chapter 14

Recommendations

A recommender system is a technology that is deployed in the environment where *items* (products, movies, events, articles) are to be recommended to *users* (customers, visitors, app users, readers) or the opposite. Typically, there are many items and many users present in the environment making the problem hard and expensive to solve. Imagine a shop. Good merchant knows personal preferences of customers. Her/His high quality recommendations make customers satisfied and increase profits. In case of online marketing and shopping, personal recommendations can be generated by an artificial merchant: the recommender system.

14.1 Knowledge based recommender systems

Both users and items have attributes. The more you know about your users and items, the better results can be expected.

Such attributes are very useful and data mining methods can be used to extract knowledge in forms of rules and patterns that are subsequently used for recommendation. For example, the item above is represented by several attributes that can be used to measure similarity of items. Even the long text description can be processed by advanced NLP tools. Then, recommendations are generated based on item similarity. When users are also described by similar attributes (e.g. text extracted from CVs of job applicants), you can recommend items based on user-item attributes similarities. Note that in this case we do not use past user interactions at all. This approach is therefore very efficient for so called “cold start” users and items. Those are typically new users and new items.

14.2 Content based recommender systems

Such systems are recommending items similar to those a given user has liked in the past, regardless of the preferences of other users. Basically, there are two different types of feedback.

Explicit feedback is intentionally provided by users in form of clicking the "like"/"dislike" buttons, rating an item by number of stars, etc. In many cases, it is hard to obtain explicit feedback data, simply because the users are not willing to provide it. Instead of clicking "dislike" for an item which the user does not consider interesting, he/she will rather leave the web page or switch to another TV channel.

Implicit feedback data, such as "user viewed an item", "user finished reading the article" or "user ordered a product", however, are often much easier to collect and can also help us to compute good recommendations.

Content based recommenders work solely with the past interactions of a given user and do not take other users into consideration. The prevailing approach is to compute attribute similarity of recent items and recommend similar items.

14.3 Collaborative filtering

Last group of recommendation algorithms is based on past interactions of the whole user-base. These algorithms are far more accurate than the algorithms described in previous sections, when a "neighborhood" is well defined and the interactions data are clean.

To construct a recommendation for a user, k-nearest neighbor users (with most similar ranked items) are examined. Then, top N extra items (non-overlapping with items ranked by the user) are recommended.

K-nearest neighbor algorithm is not only solution to collaborative filtering problem. The rule-based algorithm above uses 'Apriori algorithm' to generate set of rules from the interaction matrix. The rules with a sufficient support are subsequently used to generate candidate items for recommendations.

Important difference between K-NN and rule based algorithms is the speed of learning and recall. Machine learning models operate in two phases. In the learning phase, model is constructed and in the recall phase, model is applied to new data. Rule based algorithms are expensive to train but their recall is fast. K-NN algorithms are just the opposite, therefore they are also called lazy learners. In recommender systems, it is important to update model frequently (after each user interaction), to be able to generate new recommendations instantly. Whereas lazy learners are easy to update, rule based models have to be retrained, which is particularly challenging in large production environments. However, rules can be visualized and it is great tool to inspect quality of data and problems in your database.

14.3.1 Matrix factorization

Chapter 15

Natural language processing

15.1 Introduction

Topics in the area of natural language understanding:

1. Sentiment Analysis - Deriving sentiments in sentences (positive, negative, neutral), and also in articles. The future is to include emotions (attributes) in that, like the attributes now on Facebook posts - Love, Like, Angry, Surprised, Sad, Hilarious. These attributes make a lot more sense for sentiments going forward.
2. Text Summarization - Summarizing a single or many articles according to a particular theme.
3. Textual Entailment - Inferring directional causal relationships between textual fragments. This can be challenging in a long article.
4. Information Extraction - Find structured information from unstructured data, like entities, relationships, co-reference resolution. This at a basic level is very useful for algorithmic trading. An extension of this is a global form of extracting logic structures
5. Topic Segmentation - Topic Extraction (with regions). Normally, there will be overlapping regions.
6. Question Answering - Answer the questions to both closed (specific) and open questions (subjective). Answers to subjective questions is the main challenge for the likes of realistic Virtual Assistants.
7. Parsing - Parsing natural language generally in the form a tree. This involves hierarchical segmentation of the language involving the grammar rules.

8. Prediction - Given a short text, predict what happens next. The prediction problem is beginning to be targeted in vision, but it has never ever gained paths for realistic products. For closed and deterministic prediction, this can be a useful task for prediction of future events based on past evidences and analysis. This can be then very useful for finance sectors.

9. Part of Speech Tagging (POS) - Tagging words whether they are nouns, verbs or adjectives.

10. Translation - Translate one language to another. This can be very challenging given the nature of the language, and the grammar. Normally, under probabilistic models, this assumes that the underlying grammar is mostly the same, and thus, models normally fail for Sanskrit.

11. Query Expansion - Expand query in possible ways for making the search results more meaningful. This is normally an issue with search engines, where people do not know what all keywords (or query sentences) to include to cover the entire gamut of relevancy.

12. Argumentation Mining - Evolving field of NLP, where one wants to analyse discussions and arguments.

13. Interestingness - Most interesting portion of text in an article. This can be done very much on the same lines as in images, where one ranks the likeness of images.

15.2 Language model

A statistical language model is a probability distribution over sequences of words. Given such a sequence, say of length m, it assigns a probability $P(w_1, \ldots, w_m)$ to the whole sequence. In an *n-gram* model, the probability $P(w_1, \ldots, w_m)$ of observing the sentence $w_1, \ldots, w_m$ is approximated as

$$P(w_1, \ldots, w_m) = \prod_{i=1}^{m} P(w_i \mid w_1, \ldots, w_{i-1}) \approx \prod_{i=1}^{m} P(w_i \mid w_{i-(n-1)}, \ldots, w_{i-1})$$

Here, it is assumed that the probability of observing the i^{th} word w_i in the context history of the preceding $i-1$ words can be approximated by the probability of observing it in the shortened context history of the preceding $n-1$ words (n^{th} order Markov property). The conditional probability can be calculated from n-gram model frequency counts:

$$P(w_i \mid w_{i-(n-1)}, \ldots, w_{i-1}) = \frac{\text{count}(w_{i-(n-1)}, \ldots, w_{i-1}, w_i)}{\text{count}(w_{i-(n-1)}, \ldots, w_{i-1})}$$

The words bigram and trigram language model denote *n-gram* model language models with $n = 2$ and $n = 3$, respectively.

Chapter 16

Learning to rank

At a high level, pointwise, pairwise and listwise approaches differ in how many documents you consider at a time in your loss function when training your model. Pointwise approaches

Pointwise approaches look at a single document at a time in the loss function. They essentially take a single document and train a classifier / regressor on it to predict how relevant it is for the current query. The final ranking is achieved by simply sorting the result list by these document scores. For pointwise approaches, the score for each document is independent of the other documents that are in the result list for the query.

All the standard regression and classification algorithms can be directly used for pointwise learning to rank. Pairwise approaches

Pairwise approaches look at a pair of documents at a time in the loss function. Given a pair of documents, they try and come up with the optimal ordering for that pair and compare it to the ground truth. The goal for the ranker is to minimize the number of inversions in ranking i.e. cases where the pair of results are in the wrong order relative to the ground truth.

Pairwise approaches work better in practice than pointwise approaches because predicting relative order is closer to the nature of ranking than predicting class label or relevance score. Some of the most popular Learning to Rank algorithms like RankNet, LambdaRank and LambdaMART [1] [2] are pairwise approaches. Listwise approaches

Listwise approaches directly look at the entire list of documents and try to come up with the optimal ordering for it. There are 2 main sub-techniques for doing listwise Learning to Rank:

1. Direct optimization of IR measures such as NDCG. E.g. SoftRank [3], AdaRank [4]

2. Minimize a loss function that is defined based on understanding the unique properties of the kind of ranking you are trying to achieve. E.g. ListNet [5], ListMLE [6]

Listwise approaches can get fairly complex compared to pointwise or pairwise approaches.

Chapter 17

Click-through rate

Search engine advertising has become a significant element of the Web browsing experience. Choosing the right ads for the query and the order in which they are displayed greatly affects the probability that a user will see and click on each ad. This ranking has a strong impact on the revenue the search engine receives from the ads. Further, showing the user an ad that they prefer to click on improves user satisfaction. For these reasons, it is important to be able to accurately estimate the click-through rate of ads in the system. For ads that have been displayed repeatedly, this is empirically measurable, but for new ads, other means must be used.

Though there are many forms of online advertising, in this paper we will restrict ourselves to the most common model: pay-per-performance with a cost-per-click (CPC) billing, which means the search engine is paid every time the ad is clicked by a user (other models include cost-per-impression, where advertisers are charged according to the number of times their ad was shown, and cost-per-action, where advertisers are charged only when the ad dis- play leads to some desired action by the user, such as purchasing a product or signing up for a newsletter). Google, Yahoo, and Microsoft all primarily use this model.

In order to maximize ad quality (as measured by user clicks) and total revenue, most search engines today order their ads primarily based on expected revenue:

$$E_{ad}[revenue] = p_{ad}(click) \times CPC_{ad}$$

Thus, to ideally order a set of ads, it is important to be able to accurately estimate the p(click) (CTR) for a given ad. For ads that have been shown to users many times (ads that have many impressions), this estimate is simply the binomial MLE (maximum likelihood estimation), #clicks / #impressions.

Whenever an ad is displayed on the search results page, it has some chance of being viewed by the user. The farther down the page an ad is displayed, the less likely it is to be viewed. As a simplification, we consider the probability that an ad is clicked on to be dependent on two factors: a) the probability that it is viewed, and b) the probability that it is clicked on, given that it is viewed:

$$p(\mathit{click}|\mathit{ad}, \mathit{pos}) = p(\mathit{click}|\mathit{ad}, \mathit{pos}, \mathit{seen})p(\mathit{seen}|\mathit{ad}, \mathit{pos})$$

(Note that we are assuming that the probability that it is clicked on but not viewed is zero). We also make the simplifying assumptions that the probability an ad is clicked is independent of its position, given that it was viewed, and that the probability an ad is viewed is independent of the ad, given the position, and independent of the other ads shown:

$$p(\mathit{click}|\mathit{ad}, \mathit{pos}) = p(\mathit{click}|\mathit{ad}, \mathit{seen})p(\mathit{seen}|\mathit{pos})$$

Let the CTR of an ad be defined as the probability it would be clicked if it was seen, or $p(\mathit{click}|\mathit{ad}, \mathit{seen})$. From the CTR of an ad, and the discounting curve $p(\mathit{seen}|\mathit{pos})$, we can then estimate the probability an ad would be clicked at any position. This is the value we want to estimate, since it provides a simple basis for comparison of competing ads.

For any ad that has been displayed a significant number of times, we can easily estimate its CTR. Whenever the ad was clicked, it was seen. Whenever the ad was not clicked, it may have been seen with some probability (e.g. heat map of search page viewership intensity for different ad position). Thus, the number of views of an ad is the number of times it was clicked, plus the number of times it was estimated to have been seen but not clicked. The CTR of the ad is simply the number of clicks divided by the total number of views.

Since our goal is to predict a real-value (the CTR of an *ad*), we cast it as a regression problem - that is, to predict the CTR given a set of features. We chose to use logistic regression [5], which is ideally suited for probabilities as it always predicts a value between 0 and 1:

$$CTR = \frac{1}{1 + e^{-Z}}$$
$$\text{where} \quad Z = \sum_i w_i f_i(ad)$$

where $f_i(ad)$ is the value of the i^{th} feature for the ad, and w_i is the learned weight for that feature. Features may be anything, such as the number of words in the title, the existence of a word, etc.

Chapter 18

Graphical models

Graphical model

Chapter 19

Miscellaneous

- Curse of dimensionality (Youtube - ML Georgia Tech)
- Regression - Lasso, quantile and cardinal
- Recommendation
- Detect anomalies - class imbalance - recognition problem
- Distributions, Hessian, Jacobian
- R measure
- Loss functions and Logistic regression
- CNN - object detection
- RNN - vanishing gradient
- Boosting and XGBoost
- Neural language model
- Topic model - NMF and LDA. TopicRNN
- Sequential model
- Streaming data
- Probabilistic Graphical Models (Blog)

Chapter 20

Introduction to Bayesian methods and Conjugate priors

20.1 Think Bayesian and Statistics review

Probability mass function: A discrete random variable is a random variable whose range is finite or countably infinite. The probability mass function of a discrete random variable X is

$$f_X(x) = P\{X = x\}$$

The mass function has two basic properties:

- $f_X(x) \geq 0$ for all x in the state space
- $\sum_x f_X(x) = 1$

Probability density function: Let X be a random variable whose cumulative distribution function F_X has a derivative. The function f_X satisfying

$$F_X(x) = \int_{-\infty}^{x} f_X(t)\, dt$$

is called the probability density function and X is called a continuous random variable. By the fundamental theorem of calculus, $F'_X(x) = f_X(x)$. We can compute compute probabilities using

$$P\{a < X \leq b\} = F_X(b) - F_X(a) = \int_a^b f_X(t)\, dt$$

Independence: X and Y are independent if:

$$P(X,Y) = P(X)P(Y)$$
$$\text{Joint} = \text{Marginal} \times \text{Marginal}$$

Conditional probability: Probability of X given that Y happened:

$$\begin{aligned} P(X|Y) &= \frac{P(X,Y)}{P(Y)} \\ \text{Conditional} &= \frac{\text{Joint}}{\text{Marginal}} \end{aligned}$$

Chain rule:

$$\begin{aligned} P(X,Y) &= P(X|Y)P(Y) \\ P(X,Y,Z) &= P(X|Y,Z)P(Y|Z)P(Z) \\ P(X_1,X_2,\ldots,X_N) &= \prod_{i=1}^{N} P(X_i|X_1,X_2,\ldots,X_{i-1}) \end{aligned}$$

Sum rule: Marginalization

$$p(X) = \int_{-\infty}^{\infty} p(X,Y)\,dY$$

Bayes theorem

- θ - Parameters
- X - Observations

$$\begin{aligned} P(\theta|X) &= \frac{P(\theta,X)}{P(X)} \\ &= \frac{P(X|\theta)P(\theta)}{P(X)} \\ \text{Posterior} &= \frac{\text{Likelihood} \times \text{Prior}}{\text{Evidence}} \\ &\propto \text{Likelihood} \times \text{Prior} \end{aligned}$$

20.2 Bayesian approach to Statistics

Frequentist approach:

- X is random and θ is fixed
- $|X| \gg |\theta|$
- Maximum Likelihood:
$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}}\, P(X|\theta)$$

Bayesian approach:

- θ is random and X is fixed
- For any $|X|$
- Calculates the probability of parameters θ given the data X, using the Bayes theorem.
$$P(\theta|X) = \frac{P(X|\theta)P(\theta)}{P(X)}$$

Classification:

- Training:
$$P(\theta|X_{\text{tr}}, y_{\text{tr}}) = \frac{P(y_{\text{tr}}|X_{\text{tr}}, \theta)P(\theta)}{P(y_{\text{tr}}|X_{\text{tr}})}$$
- Prediction:
$$P(y_{\text{ts}}|X_{\text{ts}}, X_{\text{tr}}, y_{\text{tr}}) = \int P(y_{\text{ts}}|X_{\text{ts}}, \theta)P(\theta|X_{\text{tr}}, y_{\text{tr}})\, d\theta$$

Regularization: In Bayes theorem the prior $P(\theta)$ can be used as a regularizer.

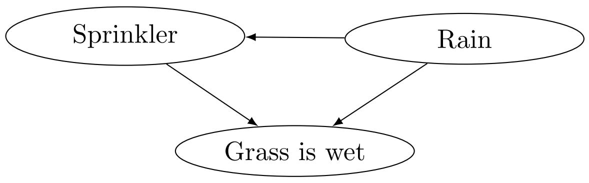
$$P(\theta|X) = \frac{P(X|\theta)P(\theta)}{P(X)}$$

On-line learning:

$$P_k(\theta) = P(\theta|X_k) = \frac{P(X_k|\theta)P_{k-1}(\theta)}{P(X_k)}$$
$$\text{New prior} = \text{Posterior} = \frac{\text{Likelihood} \times \text{Prior}}{\text{Evidence}}$$

20.3 How to define a model

Bayesian network: We see that the event "grass is wet" has two possible causes: either the water sprinker is on or it is raining. Further the sprinkler is off when it is raining. Note, we can't



have interdependent variables in Bayesian network. For those cases we can either join random variables into one random variable or use Markov Random Fields (MRF).

Probabilistic model from BN:

$$P(X_1, \ldots, X_n) = \prod_{k=1}^{n} P(X_k|\text{Pa}(X_k))$$

$$\text{Pa}(G) = \{R, S\}$$

$$P(S, R, G) = P(G|S, R)P(S|R)P(R)$$

Naïve Bayes:

$$\begin{aligned}
P(c|f_1, f_2, \ldots, f_N) &\propto P(f_1, f_2, f_3, \ldots, f_N, c) \\
&= P(f_1|f_2, f_3, \cdots, f_N, c)P(f_2, f_3, \cdots, f_N, c) \\
&= P(f_1|f_2, f_3, \cdots, f_N, c)P(f_2|f_3, \cdots, f_N, c)P(f_3, \cdots, f_N, c) \\
&= P(f_1|f_2, f_3, \cdots, f_N, c)P(f_2|f_3, \cdots, f_N, c) \cdots P(f_N|c)P(c) \\
&= P(c)P(f_1|c)P(f_2|c) \cdots P(f_N|c) \\
&= P(c) \prod_{i=1}^{N} P(f_i|c)
\end{aligned}$$

$$P(c|f_1, f_2, \cdots, f_N) = P(c) \prod_{i=1}^{N} P(f_i|c)$$

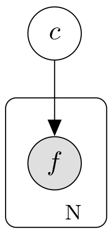
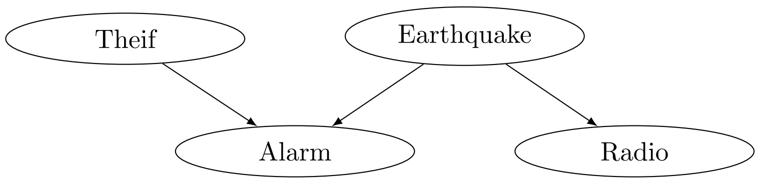


Figure 20.1: Plate notation

20.4 Example: thief and alarm

Imagine that you buy an alarm to a house to prevent thief from going into it. Either the thief goes into a house, and the alarm will go off and they will get, for example, SMS notification. However, the alarm may give a false alarm in case of an earthquake. Also, if there is a strong earthquake, the radio will report about it and so you get another source of them notification. Here's a graphical model for it. What is the general probability of the four and the variables, thief, alarm, earthquake, and the radio is given by the following formula. To fully define our model, we need to define these four probabilities.



$$P(T, A, E, R) = P(T)P(E)P(A|T, E)P(R|E)$$

Priors	
$P(T)$	10^{-3}
$P(E)$	10^{-2}

$P(A\|T, E)$	$\overline{E}$	E
$\overline{T}$	0	1/10
T	1	1

$P(R\|E)$	
$\overline{E}$	0
E	1/2

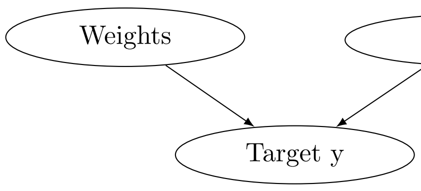
$$\begin{aligned}
P(T|A) &= \frac{P(T, A)}{P(A)} \\
&= \frac{P(T, A, E) + P(T, A, \overline{E})}{P(T, A, E) + P(T, A, \overline{E}) + P(\overline{T}, A, E) + P(\overline{T}, A, \overline{E})} \\
&= \frac{P(T)P(E)P(A|T, E) + P(T)P(\overline{E})P(A|T, \overline{E})}{P(T)P(E)P(A|T, E) + P(T)P(\overline{E})P(A|T, \overline{E}) + P(\overline{T})P(E)P(A|\overline{T}, E) + P(\overline{T})P(\overline{E})P(A|\overline{T}, \overline{E})}
\end{aligned}$$

$$\begin{aligned} P(T|A,R) &= \frac{P(T,A,R)}{P(A,R)} \\ &= \frac{P(T,A,R,E)+P(T,A,R,\overline{E})}{P(T,A,R,E)+P(T,A,R,\overline{E})+P(\overline{T},A,R,E)+P(\overline{T},A,R,\overline{E})} \end{aligned}$$

20.5 Linear regression

In linear regression, we want to feed a straight line into data. We feed it in the following way. You want to minimize the errors, and those are, the red line is the prediction and the blue points are the true values. And you want, somehow, to minimize those black lines. The line is usually found with so-called least squares problem. The prediction of each point is computed as w transposed times x_i, where x_i is our point. Then, we compute the total sum squares, that is, the difference between the prediction and the true value square. And we try to find the vector w that minimizes this function.

Let's see how this one works for the Bayesian perspective. Here's our model. We have three random variables, the weights, the data, and the target.



We have three random variables, the weights, the data, and the target. We're not interested in modeling the data, so we can write down the joint probability of the weights and the target, given the data. This will be given by the following formula.

$$P(w, y \mid X) = P(y \mid X, w)P(w)$$

It would be the probability of target given the weights of the data, and the probability of the weights. Now we need to define these two distributions. Let's assume them to be normal.

$$P(y \mid X, w) = \mathcal{N}(y \mid w^T X, \sigma^2 I)$$
$$P(w) = \mathcal{N}(w \mid 0, \gamma^2 I)$$

Let's compute the posterior probability over the weights, given the data.

$$P(w \mid y, X) = \frac{P(w, y \mid X)}{P(y \mid X)}$$

We want to maximum of this posterior distribution with respect to the weights. Notice that the denominator does not depend on the weights, and so we can maximize only the numerator.

$$\begin{aligned}
\operatorname*{argmax}_w P(w \mid y, X) &= \operatorname*{argmax}_w P(w, y \mid X) \\
&= \operatorname*{argmax}_w P(y \mid X, w) P(w) \\
\operatorname*{argmax}_w \log P(w \mid y, X) &= \operatorname*{argmax}_w \ \big[\log P(y \mid X, w) + \log P(w)\big] \\
&= \operatorname*{argmax}_w \ \Big[\log C_1 \exp\Big(-\frac{1}{2}(y - w^T X)^T [\sigma^2 I]^{-1} (y - w^T X)\Big) + \log C_2 \exp\Big(-\frac{1}{2} w^T [\gamma^2 I]^{-1} w\Big)\Big] \\
&= \operatorname*{argmax}_w \ \Big[\frac{1}{2\sigma^2}(y - w^T X)^T (y - w^T X) + \frac{1}{2\gamma^2} w^T w\Big] \\
&= \operatorname*{argmax}_w \ \big\|y - w^T X\big\| + \lambda \|w\|
\end{aligned}$$

Notice the first term is sum of squares and the second term is a L2 regularizer.

20.6 Likelihood

Given a statistical probability mass function or density, say $f(x, \theta)$, where θ is an unknown parameter, the *likelihood* is f viewed as a function of θ for a fixed, observed value of x of the random variable X. The probability density function is the continuous analogue of probability mass function.

Given the outcome x of the random variable X, the likelihood function, which is a function of θ, is given as:

$$\mathcal{L}(\theta|x) = f_\theta(x) = P_\theta(X = x) = P(X = x \mid \theta)$$

For example -

1. Suppose we flip a coin with success probability of θ
2. Recall that the mass function for x

$$f(x, \theta) = \theta^x (1 - \theta)^{1-x} \quad \text{for} \quad \theta \in [0, 1]$$

where x is either 0 (tails) or 1 (heads)

3. Suppose that the result is a head. The likelihood is

$$\mathcal{L}(\theta|1) = \theta^1(1-\theta)^{1-1} = \theta \quad \text{for} \quad \theta \in [0,1]$$

4. Therefore, $\mathcal{L}(0.5|1)/\mathcal{L}(0.25|1) = 2$

5. There is twice as much evidence supporting the hypothesis that $\theta = 0.5$ to the hypothesis that $\theta = 0.25$

20.7 MLE and MAP estimation

Maximum a posteriori (MAP) and maximum likelihood estimation (MLE) are stated as follows

$$\begin{aligned}\theta_{\mathcal{MAP}} &= \underset{\theta}{\operatorname{argmax}}\, P(\theta|X) \\ &= \underset{\theta}{\operatorname{argmax}}\, P(X|\theta)P(\theta) \\ \theta_{\mathcal{MLE}} &= \underset{\theta}{\operatorname{argmax}}\, P(X|\theta)\end{aligned}$$

where X is the input data and θ is the output parameter.

20.8 Analytical inference

There are many problems with Bayesian inference. In Bayes theorem, we have the likelihood times the prior over the evidence. However, what is the evidence? Imagine that they are working with images. For example, working with images of Van Gogh. You will have, for example, a Starry night, the Starry night over the Rhone. And if you model the probability of these images, you would also be able to draw new paintings that Van Gogh could have drawn. And so modelling this distribution is usually really hard.

We'll try to come up with ideas how we can avoid computing the evidence. It is called a Maximum a posteriori principle. We try to find the value of the parameters that maximizes the posterior probability. In Bayes theorem note that evidence does not depend on theta. And so we can remove it. And so, by computing the Maximum a posteriori, we avoid conputing the evidence.

20.9 Probability distributions

20.9.1 Normal distribution

The probability density of the normal distribution is

$$\mathcal{N}(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where μ is the mean or expectation of the distribution (and also its median and mode), σ is the standard deviation, and σ^2 is the variance.

The multivariate normal distribution is said to be "non-degenerate" when the symmetric covariance matrix $\mathbf{\Sigma}$ is positive definite. In this case the distribution has density

$$\mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}, \mathbf{\Sigma}) = \frac{1}{\sqrt{(2\pi)^k |\mathbf{\Sigma}|}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\mathbf{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)$$

where $\mathbf{x}$ is a real k-dimensional column vector and $|\mathbf{\Sigma}| \equiv \det \mathbf{\Sigma}$ is the determinant of $\mathbf{\Sigma}$. The equation above reduces to that of the univariate normal distribution if $\mathbf{\Sigma}$ is a 1×1 matrix (i.e. a single real number).

20.9.2 Binomial Distribution

Problem: A dice is rolled six times. What is the probability that we will get exactly two 3's?

Solution: The probability of pattern $*, 3, *, *, 3, *$ is $\frac{5}{6} \times \frac{1}{6} \times \frac{5}{6} \times \frac{5}{6} \times \frac{1}{6} \times \frac{5}{6}$. We conclude that the probability of getting exactly two 3's is ${}^6C_2 \left(\frac{5}{6}\right)^4 \left(\frac{1}{6}\right)^2$.

Definition: We perform a trial independently for n times, and on each trail an event we call 'success' has probability θ. Then the probability of x successes out of n trails is as follows:

$$f(x; n, \theta) = {}^nC_x \theta^x (1-\theta)^{n-x}$$

- n trial
- probability of success θ
- x successes $\implies$ $n - x$ failures

20.9.3 Multinomial Distribution

Problem: Consider a die with 1 painted on three sides, 2 painted on two sides, and 3 painted on one side. If we roll this die ten times what is the probability we get five 1's, three 2's and two

3's?

Solution: The probability of the pattern $1, 1, 1, 1, 1, 2, 2, 2, 3, 3$ is $\left(\frac{3}{6}\right)^5 \times \left(\frac{2}{6}\right)^3 \times \left(\frac{1}{6}\right)^2$. We conclude that the probability of getting five 1's, three 2's and two 3's is ${}^{10}C_5\,{}^{5}C_3\left(\frac{3}{6}\right)^5\left(\frac{2}{6}\right)^3\left(\frac{1}{6}\right)^2$ i.e. $\frac{10!}{5!3!2!}\left(\frac{3}{6}\right)^5\left(\frac{2}{6}\right)^3\left(\frac{1}{6}\right)^2$.

Definition: We see that if we have k possible outcomes for our experiment with probabilities $\theta_1, \cdots, \theta_k$, then the probability of getting exactly x_i outcomes of type i in $n = \sum_{i=0}^{k} x_i$ trails is as follows:

$$f(x_1, \ldots, x_k; n, \theta_1, \cdots, \theta_k) = \frac{n!}{x_1! x_2! \cdots x_k!} {\theta_1}^{x_1}, \ldots, {\theta_k}^{x_k}$$

- n trial
- k bins
- bin probabilities $\theta_1, \cdots, \theta_k$
- x_1 items in bin-1, x_2 items in bin-2, $\ldots$, x_k items in bin-k

20.9.4 Poisson Distribution

We use Poisson distribution to approximate binomial distribution when n is large.

20.9.5 Bernoulli Distribution

It is a special case of the Binomial distribution where a single trail is conducted $(n = 1)$.

$$f(k; p) = p^k (1 - p)^{1-k} \quad \text{for}\, k \in \{0, 1\}$$

The distribution of heads $(k = 1)$ and tails $(k = 0)$ in coin tossing is an example of a Bernoulli distribution with $(p = 1/2)$.

20.9.6 Beta distribution

Beta distribution is a type of statistical distribution, which has two free parameters. It is used as a prior distribution in Bayesian inference, due to the fact that it is the conjugate prior distribution for the binomial distribution, which means that the posterior distribution and the prior distribution are in the same family.

20.9.7 Dirichlet distribution

The Dirichlet distribution is a family of continuous multivariate probability distributions parameterised by a vector α of positive reals. It is the multivariate generalisation of the beta distribution. It is often used as the prior distribution in Bayesian inference and it is the conjugate prior of the categorical distribution and multinomial distribution.

20.9.8 Gamma distribution

Range $[0, \infty)$

20.10 Conjugate prior

In Bayesian probability theory, if the posterior distributions $p(\theta \mid x)$ are in the same probability distribution family as the prior probability distribution $p(\theta)$, the prior and posterior are then called conjugate distributions, and the prior is called a conjugate prior for the likelihood function $p(x \mid \theta)$.

Imagine you have a vectors of counts $\mathbf{n}$, that come from a multinomial distribution θ. This multinomial comes from a Dirichlet distribution with a parameter α. Using chain rule we get $p(\mathbf{n}) = p(\mathbf{n} \mid \theta)p(\theta \mid \alpha)$. $p(\mathbf{n})$ is also Dirichlet distribution.

If $\theta \sim \mathrm{Dir}(\alpha)$, $\mathbf{w} \sim \mathrm{Dir}(\theta)$ and $n_k = |\{w_i : w_i = k\}|$, i.e. is count of w_k.

$$\begin{aligned} p(\theta \mid \alpha, \mathbf{w}) &\propto p(\mathbf{w} \mid \theta)p(\theta \mid \alpha) \\ &\propto \prod_k \theta^{n_k} \prod_k \theta^{\alpha_k - 1} \\ &\propto \prod_k \theta^{\alpha_k + n_k - 1} \end{aligned}$$

20.11 Expectation maximization

20.11.1 Gaussian mixture model

$$\begin{aligned} p(x|\theta) &= \pi_1 \mathcal{N}(x \mid \mu_1, \Sigma_1) + \pi_2 \mathcal{N}(x \mid \mu_2, \Sigma_2) + \pi_3 \mathcal{N}(x \mid \mu_3, \Sigma_3) \\ \theta &= \{\pi_1, \pi_2, \pi_3, \mu_1, \mu_2, \mu_3, \Sigma_1, \Sigma_2, \Sigma_3\} \end{aligned}$$

Training GMM

$$\begin{aligned}\underset{\theta}{\operatorname{argmax}}\; p(X|\theta) &= \underset{\theta}{\operatorname{argmax}} \prod_{i=1}^{N} p(x_i|\theta)\\ &= \underset{\theta}{\operatorname{argmax}} \prod_{i=1}^{N} (\pi_1 \mathcal{N}(x_i \mid \mu_1, \Sigma_1) + \dots)\end{aligned}$$

subjected to $\sum_i \pi_i = 1$; $\pi_k \geq 0$ and $\Sigma_k \succ 0$.

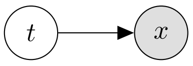


Figure 20.2: Latent variable

Introducing latent variable

$$\begin{aligned}p(t = c|\theta) &= \pi_c\\ p(x|t = c, \theta) &= \mathcal{N}(x \mid \mu_c, \Sigma_c)\\ p(x|\theta) &= \sum_{c=1}^{N} p(x|t = c, \theta)p(t = c|\theta)\end{aligned}$$

Parameter estimation

$$\begin{aligned}p(x|t = c, \theta) &= \mathcal{N}(x \mid \mu_c, \sigma_c^2)\\ \mu_c &= \frac{\sum_i^N p(t_i = c|x_i, \theta)x_i}{\sum_i^N p(t_i = c|x_i, \theta)}\\ \sigma_c^2 &= \frac{\sum_i^N p(t_i = c|x_i, \theta)(x_i - \mu_c)^2}{\sum_i^N p(t_i = c|x_i, \theta)}\\ \pi_c &= \sum_i^N \frac{p(t_i = c|x_i, \theta)}{N}\end{aligned}$$

From Bayes theorem

$$p(t = c|x, \theta) = \frac{p(x|t = c, \theta)p(t = c|\theta)}{Z}$$

EM Algorithm

- Start with randomly placed parameters $\theta = \{\pi_1, \dots, \mu_1, \dots, \Sigma_1, \dots\}$
- Until convergence repeat:
  1. For each point x_i calculate $p(t = c|x_i, \theta)$.
  2. Update parameters $\theta = \{\pi_1, \dots, \mu_1, \dots, \Sigma_1, \dots\}$ to fit points assigned to them.

20.12 Expectation Maximization

Statistical inference involves finding the right model and parameters that represent the distribution of observations well. Let $\mathbf{x}$ be the observations and θ be the unknown parameters of a ML model. In maximum likelihood estimation, we try to find the θ_{ML} that maximizes the probability of the observations using the ML model with the parameters:

$$\hat{\theta}_{\mathrm{ML}} = \underset{\theta}{\operatorname{argmax}}\ p(\mathbf{x}, \theta) \tag{20.1}$$

Typically, the problem requires few assumptions to solve the above optimization efficiently. One trick is to introduce latent variables $\mathbf{z}$ that break down the problem into smaller subproblems. For instance, in the Gaussian Mixture Model, we can introduce the cluster membership assignment as random variables z_i for each datum x_i, which greatly simplifies the model ($p(x_i|z_i = k) \sim \mathcal{N}(\mu_k, \sigma_k)$).

$$p(\mathbf{x}; \theta) = \int p(\mathbf{x}, \mathbf{z}; \theta) \mathrm{d}z \tag{20.2}$$

However, the above integration is, in many cases, intractable and can be either approximated using stochastic sampling (Monte Carlo methods) or we can simply bypass the computation using few assumptions. The second method is called variational inference, coined after the calculus of variations, which we will go over.

20.12.1 Evidence Lower Bound (ELBO)

There are many great tutorials for variational inference, but I found the tutorial by Tzikas et al. [6] to be the most helpful. It follows the steps of Bishop et al. [1] and Neal et al. [4] and starts the introduction by formulating the inference as the Expectation Maximization. Here, we will summarize the steps in Tzikas et al. [6] and elaborate some steps missing in the paper. Let $q(z)$ be a probability distribution on z. Then,

20.13 Variational inference vs. expectation maximization

The difference lies mainly in that EM algorithm is a generic maximization algorithm that can be used for both frequentist inference and Bayesian inference. However, in contrast to the EM algorithm which only gives you a point estimate, it is always better for Bayesians if, hopefully, the whole posterior distribution is available. This is different from just obtaining one point estimate because then, you donot have any measure of uncertainty that your estimate conveys with it.

This is where the variational Bayes (or variational inference, variational approximations) kicks in.

The difference of EM and VB is the kind of results they provide, EM is just a point, VB is a distribution. However, they also have similarities. EM and VB can both be interpreted as minimizing some sort of distance between the true value and our estimate, which is the Kullback-Leibler divergence.

So EM and VB are not really distinguished as to how complex they are used for is, but rather what kind of result it returns in the end.

Chapter 21

Maths

21.1 Hessian

21.2 Jacobian

Contents

1 Introduction **3**

1.1 Empirical risk minimization 3

1.2 Bias and variance 4

1.3 Bayes theorem 5

2 Modelling **7**

2.1 Feature selection 8

2.2 Hyperparameters tuning 13

2.3 K-fold Cross Validation 15

2.4 Class imbalance 16

2.5 Norm 17

2.6 Loss functions for classification 18

2.7 Metrics 19

3 Logistic Regression **21**

4 Frequent patterns mining **23**

5 Topic modelling **25**

5.1 Non-negative matrix factorization 25

5.2 LDA 25

6 Trees **27**

6.1 Decision tree 27

6.1.1 Greedy splitting 28

6.1.2 Stopping criterion 31

6.1.3 Pruning the tree 31

6.2 Bootstrapping 31

6.3 Bagging 32

6.3.1 Random forest 33

6.4 Boosting 34

6.4.1 Additive model 35

6.4.2 AdaBoost 36

6.4.3 XGBoost 36

7 Support vector machine 37

7.0.1 Kernel trick 39

8 Stochastic Gradient Descent 41

8.1 Convex function 41

8.2 Gradient Descent 42

9 Neural network 43

10 Convolutional neural network 45

10.1 Why convolutional layer? 46

10.2 Triplet loss 47

11 Sequence models 49

11.1 Recurrence neural network 49

11.1.1 Vanishing or exploding gradient problem 49

11.1.2 Gated recurrent unit 50

11.1.3 Long short-term memory 51

11.1.4 Bidirectional RNN 51

12 Regression 53

12.1 Ridge regression 54

12.2 Lasso 54

13 Clustering 55

13.1 K-means clustering 55

14 Recommendations 57

14.1 Knowledge based recommender systems 57

14.2 Content based recommender systems 58

14.3 Collaborative filtering 58

14.3.1 Matrix factorization 59

15 Natural language processing 61

15.1 Introduction 61

15.2 Language model 62

16 Learning to rank 65

17 Click-through rate 67

18 Graphical models 71

19 Miscellaneous 73

20 Introduction to Bayesian methods and Conjugate priors 75

20.1 Think Bayesian and Statistics review 75

20.2 Bayesian approach to Statistics 76

20.3 How to define a model 77

20.4 Example: thief and alarm 79

20.5 Linear regression 80

20.6 Likelihood 81

20.7 MLE and MAP estimation 82

20.8 Analytical inference 82

20.9 Probability distributions 83

20.9.1 Normal distribution 83

20.9.2 Binomial Distribution 83

20.9.3 Multinomial Distribution 83

20.9.4 Poisson Distribution 84

20.9.5 Bernoulli Distribution 84

20.9.6 Beta distribution 84

20.9.7 Dirichlet distribution 85

20.9.8 Gamma distribution 85

20.10Conjugate prior 85

20.11Expectation maximization 85

20.11.1Gaussian mixture model 85

20.12Expectation Maximization 87

20.12.1Evidence Lower Bound (ELBO) 87

20.13Variational inference vs. expectation maximization 87

21 Maths 89

21.1 Hessian 89

21.2 Jacobian 89

Bibliography

[1] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[2] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[3] N. Japkowicz, C. Myers, and M. A. Gluck. A novelty detection approach to classification. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, pages 518–523, 1995.

[4] R. M. Neal and G. E. Hinton. Learning in graphical models. chapter A View of the EM Algorithm That Justifies Incremental, Sparse, and Other Variants, pages 355–368. MIT Press, Cambridge, MA, USA, 1999.

[5] M. Richardson, E. Dominowska, and R. Ragno. Predicting clicks: estimating the click-through rate for new ads. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 521–530, 2007.

[6] D. G. Tzikas, A. C. Likas, and N. P. Galatsanos. The variational approximation for bayesian inference. *IEEE Signal Processing Magazine*, 25(6):131–146, November 2008.

[7] G. M. Weiss. Mining with rarity: a unifying framework. *SIGKDD Explorations*, 6(1):7–19, 2004.