ECE20875 SPRING 2024 EXHAUSTIVE NOTES

RYAN WANS

Contents

1. Python

1.1. **List Syntax.** We define the following syntax when using list stuctures.

```
list[0:2]       # returns values between indexes 0 to 1
list[-1]        # returns last index
[0]*n           # creates 0 array of length n
list(range(n)) # array from 0 to n-1
```

1.2. **Tuples.** Are ordered, unchangable, and allow duplicate entries.

1.3. **Sets.** Are unordered, changable, and prevent duplicate entries.

1.4. **Filter.** Updates a list with removed/filtered entries. Requires two inputs; a boolean function to evaluate each element upon, and a list to be filtered.

```
listA = [5,7,22,97,54,62,77,23,73,61]
filtA = list(filter(lambda x: (x%2 != 0), listA))
==> [5,7,97,77,23,73,61]
```

1.5. **Map.** Applies a function to all elements of a list. Requires two inputs; a function to apply, and a list to be mapped.

```
listA = [1,2,3,4,5]
mapA  = list(map(lambda x: x**2), listA)
==> [1,4,9,16,25]
```

1.6. **Reduce.** Acts as a functional; performs computation on a list and returns to a scalar result value. Computation is done sequentially to pairs of values. Resides in functools package.

```
listA = [5,8,10,20,50,100]
redA  = reduce((lambda x, y: x + y), listA)
==> 193
```

1.7. **List Comprehensions.** A new list can be constructed iteratively and condionally as needed. Typically follows the form [**output** for item in iterable **condition**].

```
squares = [n**2 for n in range(11) if n > 2]
```

1.8. **File IO.** We start by opening the file, which returns a File object when completed. We can specify to "r" read, "w" (over)write, or "a" append. We must close the file when finished with operations.

```
f = f.open("file.txt", "a")
f.write("my-first-file\n")      # overwrites the file to this
f.write("This-file\n")          # this gets appended
f.read(4)                       # reads first 4 characters
f.readline()                    # reads the i-th line
f.readlines()                   # all lines as an array
```

2. Regular Expressions

A (shitty) means of imposing a structure upon a corpus of text. Firstly, an empty string is a valid regular expression. Absorb the following RegEx syntax.

Symbol/Identifier	Description	Symbol/Identifier	Description
.	Any character but newline	^	Beginning of a line
[ ]	Any characters specified within	$	End of a line
{ }	Matches the preceding element exactly the specified number of times	*	Zero or more times
+	One or more times	?	Zero or one time
( )	Groups expressions and captures matched text	|	OR operator
\	Escapes a special character	^	Beginning of a string
$	End of a string	\d	Any digit
\D	Any non-digit character	\w	Any alphanumeric character
\W	Any non-alphanumeric character	\s	Any whitespace character
\S	Any non-whitespace character	\b	Word boundary
\B	Non-word boundary	\A	Start of the string
\Z	End of the string or before newline at the end	\z	End of the string
[^abc]	Any charaters except a,b,c once	[a-z]	Any character between a and z once
(?=abc)	Matches if "abc" comes next	(?!abc)	Matches if "abc" does not come next
(abc)*	Repeat "abc"	(?P⟨name⟩\w+)	Named group of one/-more chars together
\k	Refers to k-th group	(?P=name)	Content of the group named name

2.1. Grouping.

An example could be `x = "dog = (?P<name>\w+), cat = (?P=name)"`

2.2. Python Implementation.

The following functions can be used in Python to implement RegEx sytax for structuring, conditioning, searching, etc.

```
import re
exp = re.compile(r'ece-(264|20875|368)')
exp.match("ece-264")              # returns a match object
exp.match("hello-ece-2087")       # returns None
exp.search("hello-ece-368")       # returns a match object
exp.group()                       # returns "ece 368"
exp.span()                        # returns (6, 13) where match lives
p = re.compile(r'hello-(\w*)')    # matches "hello ..."
p.sub(r'goodbye-\1', 'well-hello-ece')
# returns 'well goodbye hello'
p.sub(r'\1-goodbye-\1', 'well-hello-X')
# returns 'well X goodbye X'
```

I hate regex a lot. Here comes the fun!

3. Statistics & Probability

3.1. **Histograms.** A discrete representation of empirical, bin-ified data. The count of each bin represents the number of observations in that domain. The empirical frequency of each bin is the portion of data in that bin to the total amount of data. For some bin i, we have that

$$\hat{p_i} = \frac{x_i}{\sum_k x_k}, \quad \sum_k \hat{p_k} = 1$$

where $\hat{p_i}$ is the probability that a data point lands in bin i. We are representing a sample of the true distribution. Larger samples means we can use finer granularity. But, finer granularity can reveal more noise. The count of a bin is the amount of data in it. The density of a bin $\hat{d_i}$ is the ratio of it's probability to it's width $\hat{p_i}/w$. The area under the histogram must integrate to 1. We approximate the number of bins used to represent m data points as $n \approx \sqrt{m}$. The Integrated Sqaure Error of the histogram is expressed as

$$L(w) = \int \left(\hat{f_m}(x) - f(x)\right)^2 dx$$

where $f(x)$ is the value for the underlying distribution $\mathcal{D}$.

3.2. **Probability Space.** A tuple consisting of a *sample space* Ω, an *events set* $\mathcal{F}$, and a *probability distribution* $\mathcal{D}$. In greater detail,

- Sample Space Ω is the set of all elementary outcomes possible in a single trial.
- Events Set $\mathcal{F}$ is a σ-algebra closed under countable unions and complementation. It is a set of subsets of Ω. If we want to express all possile outcomes, we have $\mathcal{F} := 2^\Omega$.
- Probability Distribution $\mathcal{D}$ defines the behavior of $\mathbb{P} : \mathcal{F} \to [0, 1]$ such that $\mathbb{P}[\Omega] = 1$, $\mathbb{P}[\emptyset] = 1$, and, for all mutually exclusive events $A_1, \ldots, A_n$,

  $$\mathbb{P}\left[\bigcup A_i\right] = \sum_i \mathbb{P}[A_i]$$

  for $i \in [1, n]$. At the risk of generality, the *discrete* probability space is a quantized version of this.

3.3. **Random Variables.** A random variable is a functional $X : \Omega \to \mathbb{R}$ that is measurable s.t. for some interval I, the set $\{\omega \in \Omega \mid X(\omega) \in I\} \subset \Omega$ is an event.

3.4. **Probability Density Function.** A PDF is some $f_{X\sim\mathcal{D}} : X \to \mathbb{R}$ for a random variable X and distribution $\mathcal{D}$ if the following holds:

$$\mathbb{P}[X \leq a] = \int_{-\infty}^{a} f_{X\sim\mathcal{D}}(x)dx$$

for all $a \in \Omega$. The PDF must be nonnegative, measureable, and have a total integral of one.

3.5. **Probability Mass Function.** A PMF is similar to a PDF but it acts on discrete probability distributions, not continuous ones. Similarly, we define it as the functional $x \mapsto \mathbb{P}[X = x]$. The *joint probability mass function* of discrete random variables X and Y is defined as $(x, y) \mapsto \mathbb{P}[X = x \wedge Y = y]$. We are able to obtain quasi-continuous behavior when integrating via the Dirac delta function.

3.6. **Cumulative Distribution Function.** Given some PDF, the corresponding CDF is a function $F : X \to \mathbb{R}$ such that $F(a) := \mathbb{P}[X \leq a]$. The CDF is given by the integral of it's respective PDF. For example, we have that

$$\int_a^b f_{X\sim\mathcal{D}}dx \equiv F(b) - F(a)$$

Suppose we have $\mathbb{P}[X \in [a, b] \vee X \in [c, d]]$. If these sets are disjoint, we would simply add the probabilities together. The discretized version of this would be a ramp function that adds the empirical probability as each event occurs sequentially.

3.7. **Conditional Probability.** The conditional probability of event A occuring given event B is given by the expression

$$\mathbb{P}[A \mid B] = \frac{\mathbb{P}[A \cap B]}{\mathbb{P}[B]}$$

where $\mathbb{P}[B] \neq 0$.

3.8. **Independence.** Two events A and B are said to be independent iff $\mathbb{P}[A \cup B] = \mathbb{P}[A]\mathbb{P}[B]$. Equivalently, we have that $\mathbb{P}[A \mid B] = \mathbb{P}[A]$. Two random variables are said to be i.i.d if they are independent and follow the same distribution.

3.9. **Expectation.** The expectation, or mean of a random variable X is denoted and given by

$$\mathop{\mathbb{E}}_{X\sim\mathcal{D}}[X] = \sum_x x\mathbb{P}[X = x]$$

We also have that the expectation is a linear operator such that for two rando variables X and Y and any $a, b \in \mathbb{R}$, the following holds: $\mathbb{E}[aX + bY] = a\mathbb{E}[X] + b\mathbb{E}[Y]$.

3.10. **Markov's Inequality.** Let is be shown without proof that for nonnegative some random variable X with finite expectation, then $\forall\, t > 0$

$$\mathbb{P}[X \geq t\mathbb{E}[X]] \leq \frac{1}{t}$$

3.11. **Variance & Standard Deviation.** The variance of a random variable X, denoted by $\mathrm{Var}[X]$, is defined by

$$\mathrm{Var}[X] = \mathbb{E}[(X - \mathbb{E}[X])^2] = \frac{\sum(x_i - \bar{x})^2}{n-1}$$

The standard deviation of X, denoted by σ_X, is defined by

$$\sigma_X = \sqrt{\mathrm{Var}[X]} = \sqrt{\frac{\sum(x_i - \mu)^2}{n}}$$

Note the difference between $\bar{x}$, the mean of the sample, and μ, the population (underlying distribution) mean. We also truncate $n \to n-1$ when using the empircal form.

3.12. **Covariance.** The covariance of two random variable X and Y is denoted by $\mathrm{Cov}(X, Y)$ and is defined by the expression

$$\mathrm{Cov}(X, Y) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])]$$

Two random variables X and Y are said to be uncorrelated when their covariance is zero. The covariance is defined by a positive, semidefinite, symmetric, bilinear form.

3.13. **Weak Law of Large Numbers.** For some $(X_n)_{n\in\mathbb{N}}$ sequence of i.i.d random variables (same μ and σ^2), and $\overline{X}_n = \frac{1}{n}\sum_{i=1}^n X_i$, then for any $\epsilon > 0$,

$$\lim_{n\to\infty} \mathbb{P}[\left|\overline{X}_n - \mu\right| \geq \epsilon] = 0$$

3.14. **Central Limit Theorem.** Let $(X_n)_{n\in\mathbb{N}}$ be a sequence of i.i.d random variables with mean μ and standard deviation σ again. Adopt the same aforementioned definition for $\overline{X}_n$ and let $\overline{\sigma}_n^2 = \sigma^2/n$. Then, we have that $(\overline{X}_n - \mu)/\overline{\sigma}_n$ converges to the $N(0, 1)$ distribution. Formally, for any $t \in \mathbb{R}$, we have

$$\lim_{n\to\infty} \mathbb{P}\left[\frac{\overline{X}_n - \mu}{\overline{\sigma}_n} \leq t\right] = \int_{-\infty}^{t} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) dx$$

In english, as $n \to \infty$ (but, sufficiently, $n \geq 30$), $\overline{X}_n$ will follow a normal distribution around μ.

3.15. **QQ Plots.** A plot between the CDF of the hypothesized distribution and the measured CDF (see CDF section). A quantile q is the value of x s.t. $\mathbb{P}[X \leq x] = q$. It's represented by the coordinate $F(x) = q$ on the CDF and the corresponding integral on the PDF. With the superimposition in place, for each data point, we find the quantile w.r.t the dataset (q_D) and the quantile w.r.t. the model (q_M). Add each point (q_M, q_D) to a scatter plot. Similar distributions will form a line near $y = x$; this means the hypothesized distribution is close the empiral distribution (a good match).

4. Distributions

4.1. **Bernoulli.** We have $\Omega = \{0, 1\}$, a binary output, where $p = \mathbb{P}[X = 1]$. We denote this $\text{Ber}(p)$

4.2. **Binomial.** A random variable X is said to follow a binomial distribution $B(n, p)$ with $n \in \mathbb{N}$ and $p \in [0, 1]$ if for any $k \in \{0, 1, \ldots, n\}$,

$$\mathbb{P}[X = k] = \binom{n}{k} p^k (1 - p)^{n-k}$$

We are essentially repeating a Bernoulli sample n times. Recall that a permutation is the arrangement of items in which order matters (how many different ways you can arrange n binary outputs), and a combination is the selection of items in which order does not matter (get 50 truths from 100 tests).

4.3. **Normal.** A random variable X is said to follow a normal distribution $N(\mu, \sigma^2)$ with $\mu \in \mathbb{R}$ and $\sigma > 0$ if its PDF is given by

$$f_{X \sim N}(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right)$$

The standard normal distribution $N(0, 1)$ is the normal distribution with zero mean and unit variance. We often use this to approximate binomial distributions. Recall that $\sigma \equiv 0.68, 2\sigma \equiv 0.95, 3\sigma \equiv 0.997$.

4.4. **Notes.** Exponential distributions are used to model decay processes, inter-arrival times, and occurrence of events. Geometric distributions answer the question of "how many times do I need to flip a coin to get heads?". Uniform is self-explanatory; flat. Student's T distribution models the behavior of the normal distribution but with fewer samples (smaller dataset, different tial). Poisson distribution is a discrete version of the exponential distribution.

5. Sampling

5.1. **Classifications.** A *Simple Random Sample* works by selecting S elements from a population P so that each element of P is equally like to appear in S. A *Stratified Sample* works by partitioning P by some attribute that distinguishes them and performing SRS within the partitions. A *Cluster Sample* works by grouping P into random subsets and selecting a subset at random to sample from.

5.2. **Statistics vs Parameters.** Parameters describe the underlying distribution; variables like the populations mean μ, variance σ^2, expectation, etc. Statistics describe that sample(s); variables such as sample mean $\bar{x}$, variance s^2, median, etc.

6. Hypothesis Testing & Confidence Intervals

6.1. **Hypotheses.** A *null hypothesis* assumes the mean of the underlying distribution μ is true. An *alternative hypothesis* assumes that μ has changed.

6.2. **Hypothesis Test.** If we suppose the null hypothesis is true, then the empirical distribution should have a standard error of $\sigma/\sqrt{n}$. If we don't have σ, we may have to use s. The greater the standard error, the more unlikely it is to have come from the underlying distribution.

6.3. **Z-Test.** With a known population mean and sufficiently large sample size, can determine how different the means are via

$$Z_{\text{score}} = \frac{\overline{X} - \mu}{\sigma/\sqrt{n}}$$

We've now "normalized" to $N(0, 1)$. From here, we find the p-value from the z-score (via look-up table of from `p = 2 * stats.norm.cdf(-abs(z))`). Reject $H_0 \iff p < \alpha$.

6.4. **Statistical Signifigance.** We set a signifigance level α, which manifests as the $\alpha/2$-portion of each tail, in order to determine where to reject the null hypothesis (underlying distribution is capping). If the sample mean falls into the region, we reject H_0 under statistical signifigance.

6.5. **Confidence Intervals.** A range placed around the mean to determine the validity of samples. We can interpret as "If I were to repeat the experiment a large number of times, 95 percent of confidence intervals would contain the population mean". Formally, if the parameter is outside of a **c% confidence interval**, then the event had a probability of less than $(100 - c)\%$ of occuring. Wide CI means a high variance; narrow CI means low variance. If c is the desired decimal CI, what z_c do we need s.t. $p \leq (1 - c)$? In python; `z_c = stats.norm.ppf(1 - (1 - c)/2)`. What range of μ would we expect most samples to fall in at a $c\%$ CI? Given by

$$z_c = \left|\frac{\bar{x} - \mu}{\sigma/\sqrt{n}}\right|, \quad \mu \in \left(\bar{x} - \frac{z_c\sigma}{\sqrt{n}}, \bar{x} + \frac{z_c\sigma}{\sqrt{n}}\right) =: (l_c, u_c)$$

6.6. **T-Test.** Same as Z-Test but using s in place of σ. Godspeed.

---

7. Linear Regression

Our goal is to make predications about **target** variables from **feature** variables using a model. A model is just a learned relationship between the two.

7.1. **Mean Sqaured Error.** We have the following error expression:

$$E(a, b) = \frac{1}{n}\sum_{i\in\mathbb{N}}(y_i - (ax_i + b))^2.$$

For large enough n, this will converge to the population mean of the underlying distribution such that $E(a, b) \searrow \mathbb{E}\left[\epsilon^2\right]$. If we minimize this, we get the following coefficients for our model:

$$a = \frac{\sum x_i y_i - n\overline{xy}}{\sum x_i^2 - n\bar{x}^2}, \quad b = \bar{y} - a\bar{x}$$

7.2. **Simple Linear Regression.** A model using a single explanatory variable can be modelled by the equation

$$y_n = ax_n + b + \epsilon_n$$

where y_n is the target variable and ϵ_n is the error term; a random variable that we want minimized. We express this now in matrix notation to cover the case of *more than one explanatory variable*. We get that

$$\mathbf{X} = \begin{bmatrix} x_1 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{bmatrix}, \quad \beta = \begin{bmatrix} a \\ b \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

with our model now becoming $\mathbf{y} = \mathbf{X}\beta + \epsilon$. Our goal is to solve for β; the coefficients of our model. Accordingly, $\beta = \mathbf{X}^{-1}\mathbf{y} - \mathbf{X}^{-1}\epsilon$. If we want to generalize to the case of $\mathbb{R}^{m>2}$, we have that β now contains m a coefficients and one b coefficient, and $\mathbf{X}$ is now $n \times (m + 1)$ to account for the column of 1's at the end.

7.3. **Least Squares Error.** We can minimize the error of convex models using the form

$$\min_\beta \frac{1}{n}||\mathbf{y} - \mathbf{x}\beta||^2 \equiv \min_\beta \frac{1}{n}\sum_{i\in\mathbb{N}}\left(y_i - \mathbf{x}_i^{\mathrm{T}}\beta\right)^2$$

In the multidimensional case, we apply the same principle; $\nabla\left(\frac{1}{n}||\mathbf{y} - \mathbf{x}\beta||^2\right) = 0$. An extension to case of matrix results in the equation

$$\mathbf{X}^{\mathrm{T}}\mathbf{X}\beta = \mathbf{X}^{\mathrm{T}}\mathbf{y} \implies \beta = \left(\mathbf{X}^{\mathrm{T}}\mathbf{X}\right)^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y}$$

If $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ is noninvertible, we have infintely many solutions and should choose the one that minimizes β.

8. Regularization & Ridge Regression

8.1. **Normalization.** The goal is to standardize the range of the feature values (e.g. min-max). We normalize data before modeling so that the scale of coefficients for each feature are comparable. This allows us to easily assess how important features are to the model.

8.2. **Standard Renormalization.** We start with centering the values by subtracting the column average from each feature sample. We than scale the values by dividing each feature sample by the column standard deviation. Thus, $\tilde{\mathbf{x}}_m = (\mathbf{x}_m - \bar{x}_m)/s_m$.

8.3. **Coefficient of Determination.** Given by the following expression:

$$r^2 := 1 - \frac{\text{SSE}}{\text{SST}} = 1 - \frac{\sum(y_i - f(y_i))^2}{\sum(y_i - \bar{y})^2}$$

It describes the fraction of variance in the data that is explained by the model (higher better).

8.4. **Implementation.** We can implement the aforementioned concepts into a development pipline outlines by the following algorithm. We then implement this in python

Algorithm 1 Development of Linear model

1: Normalize data that the model will fit to
2: Train (fit) model using noramalized training data
3: Normalize features by the same constraints as the data
4: Apply the model
5: Denormalize the target data
6: Evaluate fit using r^2 or ther loss metrics

```
from sklearn import linear_model
from sklearn.metrics import mean_squared_error, r2_score
regr    = linear_model.LinearRegression(fit_intercept = True)
regr.fit(X_train, Y_train)
regr.coef_       # coefficients of trained model
regr.intercept_ # b value
y_pred  = regr.predict(X_test)
r2_score(Y_true, Y_pred)
```

8.5. **Feature Trimming.** The objective is to remove features from the model that have negligable impact on the output. Normalization allows us to accomplish this. If some a_m is close to zero, we can trim it from the model.

8.6. **Overfitting.** This occurs when the model performs very well for the training dataset but performs poorly for the testing dataset. We avoid this becuase it fails to correctly generalize the dataset; generally, the simple model will be the best. Moreover, it will occur when we have too many parameters than can be justified by the data.

8.7. **Regularization.** Allows us to penalize the model for adding too many features and threatening to overfit. The goal is to achieve $\min\{\text{model error} + \lambda(\text{coef. weights})\}$. We say that $\lambda \geq 0$ is the regularization parameter. Higher λ means that we want ot minimize model parameters whereas lower λ means that we want to minimize model error. This will be a hyperparameter.

8.8. **Nonlinear Extension.** We can apply the same principles of linear regression to polynomials by simply extending the $\mathbf{X}$ and β matrices accordingly. In other words, $y_n = a_1 x_n^2 + a_2 x_n + b$ becomes

$$\mathbf{X} = \begin{bmatrix} x_1^2 & x_1 & 1 \\ \vdots & \vdots & \vdots \\ x_n^2 & x_n & 1 \end{bmatrix}, \quad \beta = \begin{bmatrix} a_1 \\ a_2 \\ b \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

This allows us to fit n-degree polynomials to our dataset.

8.9. **Ridge Regression.** Regularization is just a technique we employ to alleviate overfitting (see above). In ridge regression, the regularization term will be the sum of squares of the coefficients.

$$\min_{\beta} ||\mathbf{X}\beta - \mathbf{y}||_2^2 + \lambda||\beta||_2^2$$

where $||\cdot||_2$ denotes the L^2 norm. Notice that this is the same formula as in (8.3) but with the appended, weighted sum of sqaure coefficients. Our new matrix form becomes

$$\beta^\star = \left(\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda\mathbf{I}\right)^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y}$$

We can implement this in python via the following

```
from sklearn import linear_model
reg = linear_model.Ridge(alpha = 0.1, fit_intercept = True)
```

where the boxed parameter is our λ or regularization hyperparameter. It's trivial to observe that raising λ will have a sort of smoothing effect on the model.

9. Cross Validation

Often times when faced with training a model, we only have access to one labelled dataset. However, labelled data is required for both training and testing. With cross validation, we will split this dataset into two; one for training and one for testing. We require that the training dataset be much larger, usually around 80% to 90%.

9.1. **k-fold Cross Validation.** We repeat the partitioning of a dataset into train/test splits k times across different folds of the data. Each time we split, we choose a different partition as the training set, and thus in the end we have trained k models. We can average the performance metrics at the end.

9.2. **Model Selection.** The question of "what λ value?" will often arise. We can repeat our k-fold cross validation some n times, each with different values for our hyperparameter λ, and evaluate the effect. Thus, each CV run will train k models, but if we test with N different λ values, we will have trained nk different models.

Algorithm 2 Cross Validation for Model Selection

$\lambda \leftarrow \{\ldots, 0.01, 0.1, 1, 10, \ldots\}$ ▷ log-scale typically used
for $\lambda_i \in \lambda$ **do**
    Train k folds with λ_i
    Average the performance
end for
$\lambda^\star \leftarrow \lambda_i$ with highest CV performance
Train final model with all data using $\lambda^\star$

10. NLP: N-Grams & TF-IDF

We start with some basic implementations in `numpy` for matrix opertions.

Command	Description
`.arange(n)`	Gives you 0 to $n-1$ in a vector
`.reshape(n, m)`	Reshapes the vector into an $n \times m$ matrix
`.shape`	Returns a tuple of the dimensions
`X@Y` or `.matmul`	Multiplies matrices $\mathbf{X}$ and $\mathbf{Y}$
`.eye(n)`	Returns an $n \times n$ identity matrix
`X*Y`	Performs point-wise multiplication
`.randn(n, m)`	Returns a matrix with unit-Normal-distr. random numbers
`.linalg.inv`	Inverts a matrix if possible
`.sum(A, axis=0)`	Collapses the columns into a single summed row. Grandsum if no axis specified.
`.mean`	Returns the average of the array elements

Note that `axis=0` will collapse columns into a row vector and `axis=1` will collapse rows into a column vector. What ensues will concern the basics of NLP.

10.1. **Bag-of-N-Grams.** Creates a vector of n-grams and their frequency without order. Recall that an n-gram is just a concatentation of n characters (or words) chosen consecutively from a corpus.

10.2. **TF-IDF.** We define t to be a term/n-gram and d as a document from some D corpus (set of ds). Then we define the Term Frequency, Inverse Document Frequency (TF-IDF) score as

$$\text{tfidf}(t, d, D) = \text{tf}(t, d) \cdot \text{idf}(t, D), \quad \text{tf}(t, d) = \frac{f_{t,d}}{\sum f_{t',d}}, \quad \text{idf} = \log_{10} \frac{N}{n_t}$$

where $N = |D|$, n_t is the number of documents where t occurs, and $f_{t,d}$ is the occurences of t in d. In this sense, tf represents the fraction of terms in document d which are term t and idf represents a meaure of how rare term t is across the corpus D.

10.3. **Tokenization.** Turns a document into n-grams. Implemented using `nltk.word_tokenize(string)`.

10.4. **Stopwords.** Words which contribute nothing to our analysis of the text. E.g. "A", "for", "can", "be", "I", "so", etc. Implemented using `stopwords.words('english')`. Don't forget to lowercase everything using `s.lower()`.

10.5. **Stemming.** Reduces inflected words to their stem. For example, the stem of "studies" would be "studi", which could also map to "studious".

10.6. **Lemmatization.** Maps words to their dictionary form.

11. Objects & Classes

Every object has an `id(o)` and `type(o)`. The ID is comparable to its unique place in memory, and its type dictates the operations and values that the object can take. I find the properties are best encapsulated in examples.

```
class MultipleLists():
        def __init__(self):
                self.lists = []
        def __add__(self, a):
                newlists = MultipleLists()
                newlists.lists = self.lists.copy()
                newlists.lists.append(a)
                return newlists
        def __len__(self):
                return sum([len(a) for a in self.lists])
        def __str__(self):
                return ',-'.join([
                        f'L{i+1}={a}'
                        for i, a in enumerate(self.lists)
                ])

many_lists = MultipleLists()
print(many_lists)       # ''
print(len(many_lists))  # 0
many_lists = many_lists + [3,5,1]
print(many_lists) # L1=[3, 5, 1]
print(len(many_lists))  # 3
many_lists += [8, 4]
print(many_lists) # L1=[3, 5, 1], L2=[8, 4]
print(len(many_lists))  # 5
```

Manipulating this object will invoke the built-in functions provided, which are overwritten from the default. The following example details the workings of accessing and modifying variables that are stored in a class both from the inside and the outside.

```
class SimpleClass():
        def __init__(self, x):
                # internal created
                self.myx = x
        def add(self, y):
                # internal access and update
                self.myx = self.myx + y

my_object = SimpleClass(10)
# external access
print(my_object.myx) # 10
# internal update
```

```
my_object.add(15)
print(my_object.myx) # 25
# external update
my_object.myx = 200
print(my_object.myx) # 200
# external variable creation
my_object.myz = 18
print(my_object.myz) # 18
# external variable deletion
del my_object.myz
print(my_object.myz) # Error
```

---

12. Clustering

Clustering provides a means of classification. If we are provided with unlabelled data, we are performing **unsupervised** learning. Our goal is to cluser this data into classes/groups. **Hierarchical** groups datapoints together by closeness. **Centroid-based** groups points by closeness to a cluser center. And then there's distribution-based.

12.1. **k-Means Clustering.** We are provided n unlaballed data points $\mathbf{x}_n$ where each $\mathbf{x}$ is a feature. Goal is to divide the dataset into k clusters S_k. Each cluster S_i has a centroid μ_i; the mean of every point in the cluster. Our goal is the following:

$$\arg\min_{S_k} \sum_{i\in\mathbb{N}} \sum_{\mathbf{x}\in\{\mathbf{x}\}} ||\mathbf{x} - \mu_i||^2$$

We can define this algorithmically with the following:

Algorithm 3 k-Means Clustering

1: $\mu \leftarrow k$ random centroids
2: **for** $\mathbf{x}_i \in \{\mathbf{x}\}$ **do**
3: Assign $\mathbf{x}_i$ to the closest S_k based on Euclidean distance from μ_k
4: **end for**
5: Move each μ_k to the center of it's cluster
6: Repeat assignment loop
7: Repeat until convergence

We implement this in Python using the following:

```
from sklearn.cluster import KMeans
kmeans = KMeans(num_clusters, num_init, random_state, ...)
kmeans.fit(X)
kmeans.labels_
kmeans.cluster_centers_
```

Choosing the right k value involves cross validating over k and finding the "elbow point" of the k vs. Error plot. We don't want to overfit.

12.2. **Gaussian Mixture Models.** A distribution-based approach to clustering. We define a Gaussian Mixture Model (GMM) with k components (clusters) as a probability distribution that is a weight sumer of k Gaussians. In other words, $p_X(x) = \sum \pi_i \mathcal{N}(x \mid \mu_i, \sigma_i^2)$ gives us the probability of x being in one of the k distributions. $\mathcal{N}$ represents a specific Gaussian with mean μ_i and variance σ_i^2 and weight π_i, where $\sum \pi_i = 1$ (must integrate to 1). We utilize the following method for fitting.

12.3. **Expectation Maximization.** We apply an iterative approach to fit Gaussian parameters. We are trying to discover the cluster labels; latent variables. In other words, we start with a random guess

Algorithm 4 Expectation Maximization Algorithm

Input: Data $\mathbf{X}$, Model parameters θ, Number of iterations T
Output: Maximum likelihood estimates of θ
Initialize $\theta^{(0)}$
for $t = 1$ to T **do**
  E-step: Compute the posterior distribution over the latent variables:
    For each data point x_i:
      Compute $Q_i(z) = p(z|x_i, \theta^{(t-1)})$, where z denotes latent variables
      (For example, in Gaussian Mixture Model, z represents cluster assignments)
  M-step: Update the model parameters by maximizing the expected log-likelihood:
    Update $\theta^{(t)}$ by solving:
    $\theta^{(t)} = \arg\max_\theta \sum_{i=1}^{N} \mathbb{E}_{Q_i}[\log p(x_i, z; \theta)]$
    (In many cases, this involves maximizing the expected complete-data log-likelihood)
end for
Return: $\theta^{(T)}$

for the Gaussian paramters. We compute the E-step. This involves finding the likelihood that each point comes from a particular Gaussian using the current parameters. For each point x_j, we find the likelihood that is comes from Gaussian i's random variable G_i using

$$\gamma_{ij} = \mathbb{P}[G_i \mid x_j] = \frac{\mathbb{P}[x_j \mid G_i]\mathbb{P}[G_i]}{\mathbb{P}[x_j]} = \frac{\pi_i \mathcal{N}(x_j \mid \mu_i, \sigma_i^2)}{\sum_g \pi_g \mathcal{N}(x_j \mid \mu_g, \sigma_g^2)}$$

Next we perform the maximization, or M-Step. Given these (new) likelihoods (weights), update the means, covariances, and weights of the Gaussians using weighted averages.

$$N_i = \sum_j \gamma_{ij} \quad \pi_i' = \frac{N_i}{N} \quad \mu_i' = \frac{\sum_j \gamma_{ij} x_j}{N_i} \quad \sigma_i'^2 = \frac{\sum_j \gamma_{ij}(x_j - \mu_i')^2}{N_i}$$

The parameters are the total likelihood of points in this Gaussian, proportion of points hat come from this Gaussian, weighted mean of this Gaussian, and weighted variance of this Gaussian, respectively. For each parameter, we are setting the derivative of $\log \prod_j p_X(x_k)$ to zero. We repeat the E-step and M-step until convergence. We deterine the convergence using the log-likelihood of the data give the parameters. The log-likelihood is a measure of how well a model fits data, and is defined as the natural logarithm of the likelihood function:

$$F(\theta) = \sum_{i \in \mathbb{N}} \ln f_i(y_i \mid \theta) \quad \text{where} \quad \mathcal{L}(\theta) = \prod_{i \in \mathbb{N}} f_i(y_i \mid \theta)$$

13. Inheritance

Lowkey easy, see ensuing example. Also recall the `super().whatever` function.

```
class Person:
        def __init__(self, name_:
        self.name = name
def getName(self):
        return self.name
class AgePerson(Person):
        def __init__(self, name, age):
        self.age = age
        self.name = name
def getAge(self):
        return self.age
```

14. Classification

Unlike clustering, we are starting with labelled data (supervised learning). The goal is, given a new data point, which class will it fall under?

14.1. **Naive Bayes.** Consider data in a d-dimensional space where each data point $\mathbf{x}_i$ has d features. The idea is that each class of data has it's own probability distribution. Given some point, which class is it more likely to have come from? We apply Bayes' Theorem to find the posterior probability using quantities from the Naive Bayes model. In words, our goal is to compare $\mathbb{P}[C_0 \mid x]$ with $\mathbb{P}[C_1 \mid x]$, etc. Recall that from Bayes' Rule, we get that we can actuall compare

$$\mathbb{P}[x \mid C_0]\frac{\mathbb{P}[C_0]}{\mathbb{P}[C_1]} \quad ? \quad \mathbb{P}[X \mid C_1]$$

where our goal is to discover the equivalence relation ?. In higher dimensions, we can apply a Multivariate Gaussian (MG) in the following manner:

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

The pros of this algorithm are that it's easy to build the classifier and compute likelihoods, and is good for missing data. However, you need to choose the correct model for the data (can be hard) and requires prior knowledge.

Algorithm 5 Naive Bayes Classification Algorithm

Input: Training data $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, where $\mathbf{x}_i$ is the feature vector and y_i is the class label
Output: Predicted class label for new data points
Training Phase:
for each class C_k **do**
  Compute class prior probability: $P(C_k) = \frac{\text{Number of samples in class } C_k}{\text{Total number of samples}}$
  For each feature x_j:
    Compute mean and variance of x_j for samples in class C_k
    (For categorical features, compute probability of each category in class C_k)
end for
Prediction Phase:
for each new data point **x do**
  for each class C_k **do**
    Compute class conditional probability:
      For continuous features:
        $P(\mathbf{x} \mid C_k) = \prod_{j=1}^d \frac{1}{\sqrt{2\pi\sigma_{jk}^2}} \exp\left(-\frac{(x_j-\mu_{jk})^2}{2\sigma_{jk}^2}\right)$
      For categorical features:
        $P(\mathbf{x} \mid C_k) = \prod_{j=1}^d P(x_j \mid C_k)$
  end for
  Use Bayes' theorem to compute posterior probability:
    $P(C_k \mid \mathbf{x}) = \frac{P(C_k)\times P(\mathbf{x}|C_k)}{\sum_{k=1}^K P(C_k)\times P(\mathbf{x}|C_k)}$
  Predict the class label for $\mathbf{x}$:
    $\hat{y} = \arg\max_k P(C_k \mid \mathbf{x})$
end for
Return: Predicted class labels for new data points

14.2. **k-Nearest Neighbor.** Naive Bayes is cool and all (allegedely), but it's parametric. We will noe deal with labelled training data and no initial assumptions. We want to observe the classes of the k-nearest points and pick the most frequent one. We typically choose k from cross-validation (it's a hyperparameter). It's sumple and makes no prior assumptions, but is computationally expensive (many dist. calcs) and does not help with missing data. Also... overfitting.

14.3. **Logistic Regression.** Pretty sure my Grandma knows how to do normal/boring linear regression, so let's step it up a notch, shall we? No longer will we use a linear model between features and targets; it will treat classes as numbers and cannot be interpreted as a probability. Instead of fitting a hyperplane, we use the logistic function $g(v) = (1+\exp(-v))^{-1}$. This will convert outputs to probabilities, is bounded between 0 and 1, and thus has a decision boundary at $y = 0.5$. Note that $1 - g(v) = \exp(-v)/(1 + \exp(-v))$. In order to interpret the nonlinear effect of coefficients, we inspect the odds ratio

$$\frac{\mathbb{P}[y=1 \mid x]}{\mathbb{P}[y=0 \mid x]} = \frac{1}{\exp(-(\beta_0 + \beta_1 x_1 + \cdots + \beta_m x_m))} = \exp(\beta_0 + \beta_1 x_1 + \cdots + \beta_m x_m)$$

Variables that bring $\hat{y}$ closer to 1 will increase the odds, and the coefficient of each variable determines impact. Our actual function becomes

$$f_\beta(x) = g(\beta_0 + \beta^T x) = \mathbb{P}[y = 1 \mid x] = \frac{1}{1 + \exp(-(\beta_0 + \beta_1 x_1 + \cdots + \beta_m x_m))}$$

When we consider the ratio of the odds when x_j is increased by 1:

$$\frac{\text{odds}_{x_j+1}}{\text{odds}_{x_j}} = \frac{\exp(\cdots + \beta_j(x_j + 1) + \cdots)}{\exp(\cdots + \beta_j x_j + \cdots)} = e^{\beta_j}$$

Our decision boundary is given by $\hat{y}(x) \geq 0.5 \to \hat{y} = 1, \quad \hat{y}(x) < 0.5 \to \hat{y} = 0$. My yapping boils down to; a unit change in x_{ij} corresponds to a factor e^{β_j} change in odds. And thus $e^{\beta_j} > 1$ means x_j increases the odds and $e^{\beta_j} < 1$ means x_j decreases the odds. Positive coefficients increase odds, negative decrease odds. The greater the coefficient, the greater the impact. Let's talk about training. The likelihood of the model

$$L(\beta) = \prod_{i \in \mathbb{N}} (f_\beta(x_i))^{y_i} \cdot (1 - f_\beta(x_i))^{1-y_i}$$

will maximize $f_\beta(x_i)$ when $y_i = 1$ and maximize $1 - f_\beta(x_i)$ when $y_i = 0$. Similar to GMMs, we want to optimize the log likelihood

$$l(\beta) = \sum_{i \in \mathbb{N}} [y_i \log f_\beta(x_i) + (1 - y_i) \log(1 - f_\beta(x_i))]$$

We want to apply gradient descent (or ascent) to find $\arg\max_\beta l(\beta)$, but there is no closed form $\ddot{\frown}$. Since we know that $l(\beta)$ is concave, we utilize ascent, and thus

$$\beta_j^{t+1} = \beta_j^t + \alpha^t \frac{\partial}{\partial \beta_j} l(\beta^t)$$

will help us to find a global maximum for some sufficient step size α^t. An example ensues. Support we have single parameter b for some model we are trying to train. For this model, we find a log-likelihood function of $l(b) = -((b - m)/s)^2$ for $m, s \in \mathbb{R}$. We wil derive the iteratuve procedure for termining the model parameters as a function of the step size α^t and run the procedure for different values of α^t until $t = 10$ and compare results. The solution follows. Let b^t be the value of b after the t-th iteration, where our update procedure is $b^{t+1} = b^t + \alpha^t \frac{d}{db} l(b^t)$. The derivative gives $b^{t+1} = b^t - 2\alpha^t((b^t - m)/s^2)$. Suppose $\alpha = 0.1, m = 5, s = 0.7$ and we start at $b^0 = 1.1$, we get $b^1 = 1.1 - 2 \cdot 0.1 \cdot ((1.1 - 5)/0.7^2) = 2.692$. This shit is lowkey hell, hopefully this is not on the exam $\ddot{\smile}$.

15. Binary Evaluation Metrics

With linear regression we used MSE and r^2 as our evaluation metrics. In classification we can't use this since prediction is a binary right/wrong answer. The confusion matrix is provided in Fig 1. Accuracy focuses on the false positives and false negatives that occur. Precision focuses on true positives (correct positives) and false positives (incorrec positives) which is given by $\frac{TP}{TP+FP}$. Recall inspects a similar case with false negatives (incorrect negatives) given by $\frac{TP}{TP+FN}$.

16. Neural Networks

We will focus on the use of neural networks specifically for classification. Our goal is to both learn a decision boundary and classify new points using it.

16.1. **Perceptrons.** Similar to their biological counterparts, neurons in NNs have an activation function which "activate" given enough stimulus. These are graphed as nodes. The perceptron is the simplist realization of a neuron for our purpose. The activation function is the unit step function (on/off); a linear decision boundary. However, because this step is not analytic, we use functions such as sigmoidal, tanh, ReLU, or Leaky ReLU instead (basically logistic regr. function).

16.2. **Decision Boundaries.** A single perceptron will induce a straight line decision boundary that seperates two or more classes of data. With neural networks, changes the layers and width of layers allows us to get nonlinear decision boundaries with the smoothness determines by the activation functions. We need these because perceptrons cannot learn nonlinear decision boundaries on their own.

		Predicted condition			
	Total population $= P + N$	**Predicted Positive (PP)**	**Predicted Negative (PN)**	Informedness, bookmaker informedness (BM) $= TPR + TNR - 1$	Prevalence threshold (PT) $= \frac{\sqrt{TPR \times FPR} - FPR}{TPR - FPR}$
Actual condition	**Positive (P)** [a]	**True positive** (TP), hit[b]	**False negative** (FN), miss, underestimation	True positive rate (TPR), recall, sensitivity (SEN), probability of detection, hit rate, power $= \frac{TP}{P} = 1 - FNR$	False negative rate (FNR), miss rate, type II error [c] $= \frac{FN}{P} = 1 - TPR$
	Negative (N)[d]	**False positive** (FP), false alarm, overestimation	**True negative** (TN), correct rejection[e]	False positive rate (FPR), probability of false alarm, fall-out, type I error [f] $= \frac{FP}{N} = 1 - TNR$	True negative rate (TNR), specificity (SPC), selectivity $= \frac{TN}{N} = 1 - FPR$
	Prevalence $= \frac{P}{P + N}$	Positive predictive value (PPV), precision $= \frac{TP}{PP} = 1 - FDR$	False omission rate (FOR) $= \frac{FN}{PN} = 1 - NPV$	Positive likelihood ratio (LR+) $= \frac{TPR}{FPR}$	Negative likelihood ratio (LR−) $= \frac{FNR}{TNR}$
	Accuracy (ACC) $= \frac{TP + TN}{P + N}$	False discovery rate (FDR) $= \frac{FP}{PP} = 1 - PPV$	Negative predictive value (NPV) $= \frac{TN}{PN} = 1 - FOR$	Markedness (MK), deltaP (Δp) $= PPV + NPV - 1$	Diagnostic odds ratio (DOR) $= \frac{LR+}{LR-}$
	Balanced accuracy (BA) $= \frac{TPR + TNR}{2}$	F_1 score $= \frac{2\,PPV \times TPR}{PPV + TPR} = \frac{2\,TP}{2\,TP + FP + FN}$	Fowlkes–Mallows index (FM) $= \sqrt{PPV \times TPR}$	Matthews correlation coefficient (MCC) $= \sqrt{TPR \times TNR \times PPV \times NPV} - \sqrt{FNR \times FPR \times FOR \times FDR}$	Threat score (TS), critical success index (CSI), Jaccard index $= \frac{TP}{TP + FN + FP}$

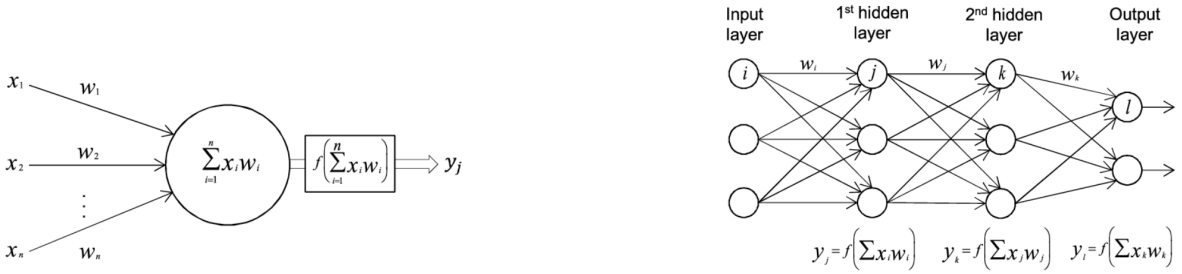
Sources: [1][2] [3][4][5][6][7][8] view · talk · edit

FIGURE 1. Binary Evaluation Metrics (credit: Wikipedia)

16.3. **Multilayer NN Structure.** Each perceptron is going to act like a mini logistic regression model. It will evaluate the weighted sum of n inputs ($z = \sum_{i\in\{n\}} w_i x_i$) and then activate accordingly ($y = f(z)$). We can put these together in layers to form a *feed-forward neural network*. In this, each perceptron computes the weighted sum and activation function, feeding the output to one layer as the input to the next layer (deep). This is modelled in Fig 2. We can express the weighted sum in vector form as $\mathbf{w}_j^{\mathrm{T}}\mathbf{x}$. A vertical stack (layer) can be exressed as $\mathbf{z} = W\mathbf{x}$ where $\mathbf{z}$ is the weighted sum for each neuron and W is the column vector of each weight vector transposed. Our activation function becomes $\mathbf{y} = f(\mathbf{z})$. Thus,

$$\mathrm{DNN}(\mathbf{x}) = \mathbf{f}(W^{(3)}\mathbf{f}(W^{(2)}\mathbf{f}(W^{(1)}\mathbf{x})))$$

would express some 3-layer network.



(A) Perceptron input activation (B) Feeding forwards

FIGURE 2. Structure of a Deep Neural Network

As an example, consider the example of classifying an XOR function. If you plot the output in $(x_1 \times x_2)$-space, it is clearly not a linear decision boundary. We will apply a simple two-layer NN to this. Our activatoin function will be given by $\mathbf{h} = \mathrm{ReLU}(W\mathbf{x} + \mathbf{c}) = \max\{0, W\mathbf{x} + \mathbf{c}\}$ and the output is given by $y = \mathbf{w}^{\mathrm{T}}\mathbf{h}$. The is modeled in Fig 3. We will prove that

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix} \quad \mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$

works as a solution.

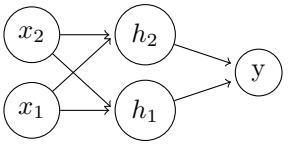


FIGURE 3. Graph of example provided

In escence, we have that neural networks map the input space which is very complicated and not easily or at all separable by a line (classified) to an output space which is also high dimension (maybe smaller) than can be separated by a line (classified).

16.4. **Batch & Stochastic Grad. Desc.** Batch requires evaluating the gradient for every sample, whereas stochastic moves 1 step at a time but in a random direction (weighted walk?). We define the error of the output of a specific input as

$$E(\mathbf{x}) = \frac{1}{2}(y_i - f(\text{sum}))^2 = \frac{1}{2}(y_i - f(\mathbf{w}^T\mathbf{x}_i))^2$$

For SGD specifically, we onl need the partial derivative for one specific input

$$\frac{\partial E(\mathbf{x}_i)}{\partial w_j} = -(y_i - f((sum)) \cdot f(\text{sum})(1 - f(\text{sum})) \cdot x_{ij}$$

since $\partial f(x)/\partial x = f(x)(1 - f(x))$ when f is a sigmoid. For SGD, we get the update rule $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \alpha \cdot \delta_0 \cdot \mathbf{x}_i$.

16.5. **CNNs.** The goal is to vectorize large matrices like an image for example. An effectie way to reduce model parameters is to have each neuron only process inputs from a local region and have neurons in the same layer share weights. Blah blah blah. We can apply a convolution filter. Convolve a filter with the image, computing at each position a dot product between the filter and a small chunck of the image $\mathbf{W}^T\mathbf{X} + b$. The dot product goes though an activation function to produce output. Convolution is often followed by pooling; creating a smaller and more manageable repreenttion whil reatining the most improtant information. Operates over each feature map independently. Max Pooling is a pooling operation that calculates the maximum value for patches of a feature map.

I RAN OUT OF TIME so im going to just print what I have and hope/pray for the best

Update after exam: I got cooked.