UDACITY

AI FOR TRADING: TERM II

OFFLINE INSTRUCTIONS

Another poster:
Zips all the files in the workspace so you can download them all at the same time:

```
zip -r workspace.zip ./*
```

Zips just data:

```
zip -r data.zip ../../data/*
```

Rama Krishna B:
1)First Download the data/module* folder. This depends on your exercise. Try something line " !tar -cvf my.tar /data/ " and download and extract my.tar to your local workstation.

2)Change os.environ['ZIPLINE_ROOT'] variable in your python notebook. Set it to appropriate place based on where you extracted your data folder from the my.tar. Look at the change in my iPython notebook and do something similar. One easy way to do is to print the existing os.environ['ZIPLINE_ROOT'] and work accordingly.

3)If there is any reference to any file in the data folder, change the path accordingly.

4)Look at https://colab.research.google.com/drive/1Dj1fLiEA122N95vjgfLNOSRda5IbpIlH That is an exercise file from from one of our lessons.

Rama Krishna:

```
!tar -vczf pandas2008.tar.gz ../../data/project_8_barra/pandas-frames.2008.pickle
```

Try the above for every pickle file individually , download the tar and then delete the tar and repeat for all the pickle files. The pickle files are large and hit the Udacity limits. The other group of files can be downloaded together after compressing them.

To download twits data:
run the following in your notebook:
Copy the data to the workspake dir with: !cp -r ../../data/ ../workspace/
Zip the data using: !zip -r data.zip data/
— I used json.dump() once the data was already loaded into notebook

```
with open('file_name.json', 'w') as f:
    json.dump(twits, f)
```

The below codes will copy the entire original data directory into `/home/workspace/data` directory.

```
import shutil
shutil.copytree('/home/workspace/../../data/', '/home/workspace/data')
```

I didn't realize that the folder can't be downloaded. I think the number of data files are few so you can download the data files individually. In case there are large number of files then you can zip the folder using the terminal embedded in the workspace or inside the notebook using shutil as well.

```
shutil.make_archive(output_filename, 'zip', dir_name)
```

MODULE 5: NLP ON FINANCIAL STATEMENTS

Lesson 3: Text Processing

Capturing Text Data

PLAIN TEXT

```
import os
  # Read in a plain text file
with open(os.path.join("data", "hieroglyph.txt"), "r") as f:
    text = f.read()
    print(text)
```

TABULAR DATA

```
import pandas as pd
  # Extract text column from a dataframe
df = pd.read_csv(os.path.join("data", "news.csv"))
df.head()[['publisher', 'title']]
  # Convert text column to lowercase
df['title'] = df['title'].str.lower()
df.head()[['publisher', 'title']]
```

ONLINE RESOURCE

```
import requests
import json
  # Fetch data from a REST API
r = requests.get(
    "https://quotes.rest/qod.json")
res = r.json()
print(json.dumps(res, indent=4))
  # Extract relevant object and field
q = res["contents"]["quotes"][0]
print(q["quote"], "\n--", q["author"])
```

Normalization, Tokenization, Cleaning, Stopwords

NORMALIZATION

A part of converting all different representations of a word to one token: conventions such as capitalizing words at the beginning of a sentence, or all-caps for emphasis, may be helpful for humans but not for computers

- Normalize to lowercase
  - `text.lower()`
- In some situations you may need/want to remove special characters like punctuation marks
  - `re.sub(r'[^a-zA-Z0-9]',” “, text.lower())`

TOKENIZATION

Splitting sentences/text into words

```
# Split text into tokens (words)
words = text.split()
```

NLTK

```
import os
import nltk
```

```
# http://www.nltk.org/_modules/nltk/data.html
# Functions to find and load NLTK resource files, such as corpora, grammars, and saved
        processing objects.
nltk.data.path.append(os.path.join(os.getcwd(), "nltk_data"))

# Split text into words using NLTK (same as .split but smarter)
from nltk.tokenize import word_tokenize
words = word_tokenize(text)
print(words)

# Split text into sentences
from nltk.tokenize import sent_tokenize
sentences = sent_tokenize(text)
print(sentences)
```

CLEANING

```
# Remove HTML tags using RegEx
pattern = re.compile(r'<.*?>')  # tags look like <...>
print(pattern.sub('', r.text))  # replace them with blank

# Remove HTML tags using Beautiful Soup library
soup = BeautifulSoup(r.text, "html5lib")
print(soup.get_text())
```

STOP WORD REMOVAL

```
# Remove stop words that are common but uninformative, like is a the ….
words = [w for w in words if w not in stopwords.words("english")]
```

Part-of-Speech Tagging; Named Entity Recognition

POS

Nouns, pronouns, verbs, adverbs, et cetera. Identifying how words are being used in a sentence can help us better understand what is being said. It can also point out relationships between words and recognize cross references. NLTK, again, makes things pretty easy for us.

NAMED ENTITY RECOGNITION

Used to identify company names etc.

```
from nltk import ne_chunk
ne_chunck(pos_tag(word_tokenize('Antonio joined Udacity Inc. in L.A.')))
```

Stemming and Lemmatization

STEMMING

Stemming converts all variations of a word to its root (ie: removes -ed, -ing, -s, etc..). NLTK has a few different stemmer to choose from, including PorterStemmer, SnoballStemmer, and more.

```
# Reduce words to their stems
from nltk.stem.porter import PorterStemmer
stemmed = [PorterStemmer().stem(w) for w in words]
```

LEMMATIZATION

Similar to stemming, but uses a dictionary to map different variant of a word to its root. WordNet is the default database used by NLTK library. A lemmatizer needs to know or make an assumption about the part of speech for each word it's trying to transform. In this case, WordNetLemmatizer defaults to nouns, but we can override that by specifying the PoS parameter, such as pos='v' for verbs.

```
# Reduce words to their root form
from nltk.stem.wordnet import WordNetLemmatizer
lemmed = [WordNetLemmatizer().lemmatize(w) for w in words]
# Lemmatize verbs by specifying pos
lemmed = [WordNetLemmatizer().lemmatize(w, pos='v') for w in lemmed]
```

Exercise

Get tweets from https://twitter.com/AIForTrading1 assign to 'all_tweets'

Step-by-step to show how each library works

```
# Convert to lowercase:
all_tweets = [tweet.lower() for tweet in all_tweets]

# Remove punctuation with regex:
counter = 0
For tweet in all_tweets:
      all_tweets[counter] = re.sub(r'[^a-zA-z0-9]', ' ', tweet)
      counter += 1
print(all_tweets)

???
import os
import nltk
nltk.data.path.append(os.path.join(os.getcwd(), "nltk_data"))

# Tokenize text
from nltk.tokenize import TweetTokenizer
tknzr = TweetTokenizer(preserve_case=False)
tokens = [tknzr.tokenize(tweet) for tweet in all_tweets]

# Remove stopwords:
from nltk.corpus import stopwords
nltk.download('stopwords')
for tweet in all_tweets:
    tokens = tknzr.tokenize(tweet)
    print([w for w in tokens if w not in stopwords.words('english')])

# Perform stemming on all tweets:
from nltk.stem.porter import PorterStemmer
for tweet in all_tweets:
      words = tweet.split()  # or use tokens method in previous…
      new_words = [w for w in words if w not in stopwords.words('english')])
      print([PorterStemmer().stem(w) for w in new_words])

# Lemmatize on all tweets: and for part-of-speech as verbs…
from nltk.stem.wordnet import WordNetLemmatizer
for tweet in all_tweets:
      words = tweet.split()
      new_words = [w for w in words if w not in
            stopwords.words('english')
      print([WordNetLemmatizer().lemmatize(w, pos='v') for w in
new_words])
```

Single complete workflow should look something like this:

```
# Import all dependencies
import os
import nltk
from nltk.tokenize import TweetTokenizer
```

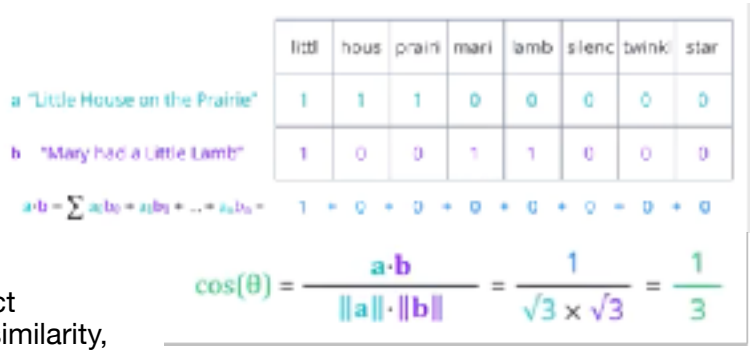
```
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer, WordNetLemmatizer

nltk.download('stopwords')
nltk.download('wordnet') ????? why not?

nltk.data.path.append(os.path.join(os.getcwd(), "nltk_data"))

# Pre-process tweets:
tknzr = TweetTokenizer(preserve_case=False)

For tweet in all_tweets:
        # tokenize text
        tokens = tknzr.tokenize(tweet)
        # remove stopwords
        new_words = [t for t in tokens if t not in
                stopwords.words('english')
        # perform stemming
        stemmed = ([PorterStemmer().stem(w) for w in new_words])
        # lemmatize
        lemmed = ([WordNetLemmatizer().lemmatize(w) for w in stemmed])
        print(lemmed)
```

Lesson 4: Feature Extraction

Bag of Words

Treats each document as an unordered bag of words. First you would do all the cleaning steps of the previous lesson and treat the resulting tokenized words as unordered. Then take the words across the corpus (ie: set of documents) and turn them into vectors of numbers. To do this, first collect all of the unique words across the corpus to form your vocabulary. The vocabulary set can be the headings across the top of a table and each document is a row in the table, and then record the frequency of occurrences of each word in each document to form the vector.

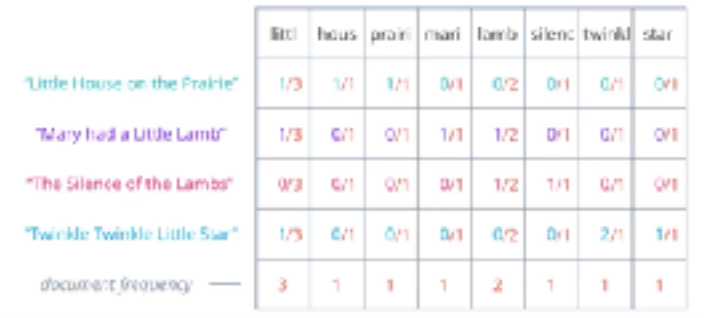
Then you can do things like compare things like how many terms (words) they have in common etc.. One way to do this is to take the dot product between two row vectors, which is the sum of the products of the corresponding elements. The greater the dot product, the more similar the two vectors are.



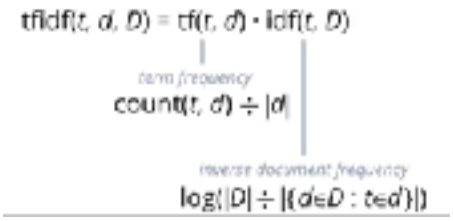
However, dot product only captures direct overlaps. A better method is the cosine similarity, which divides the dot product by the product of the vectors' Euclidean norms. Identical vectors have a cosine similarity of 1, orthogonal vectors have cos = 0, and for vectors that are exactly opposite, cos = -1.

TF-IDF

A limitation of Bag-of-Words is that is treats each word with equal importance, when some words may be frequent but not necessarily important. We can adjust for this by counting a word's frequency in each document, called 'document frequency', and then dividing each document (row) by the document frequency of that term. This gives the proportion of the frequency of a term in a document, but inversely proportional to the number of documents it appears in. It highlights the words that are more unique to a document and thus better for characterizing the document.



TF-IDF stands for term frequency and inverse document frequency. It is a formula that multiplies the term frequency with the inverse document frequency. Term frequency is calculated as the raw count of a term 't' in a document 'd' divided by the total number of terms in 'd' (ie: the vocabulary set in 'd'). Inverse document frequency is calculated as the log of the total number of documents in the collection (corpus) 'D' divided by the number of documents where 't' is present.



So TF-IDF is a way to assign weight to words that signify their relevance in documents.

One-Hot Encoding

For a deeper analysis of text we need a numerical representation of each word, which is what one-hot encoding does. As done in multiple classification, simply treat each word as a class by assigning that word a value of one in a pre-determined position in a vector that has zeroes everywhere else. It's just like the bag-of-words representation, only that we keep a single word in each bag and build a vector for it.

	litl	hous	prairi	mari	lamb	silenc	twinkl	star
hous	0	1	0	0	0	0	0	0
lamb	0	0	0	0	1	0	0	0
silenc	0	0	0	0	0	1	0	0
twinkl	0	0	0	0	0	0	1	0

Word embeddings

One-hot encoding breaks down though when we have a large vocabulary as the size of our word representation grows with the number of words. To combat this, we need to limit our word representation to a fixed-size vector, ie: we want to find an embedding for each word in some vector space (??) and we want it to exhibit some desired properties. For example, if two words are similar in meaning, we want them to be closer to each other in the embedded space than to words that are disimilar. And if two sets of words have a similar difference in their meanings, they should be approximately equally separated in the embedded space. This can be used to help find synonyms, analogies, identifying concepts around which words are clustered, classifying words are positive, negative, neutral etc.

Word2Vec

Is a model that is able to predict a word given neighboring words, or vice-versa, which means it is able to capture the contextual meanings of words very well.

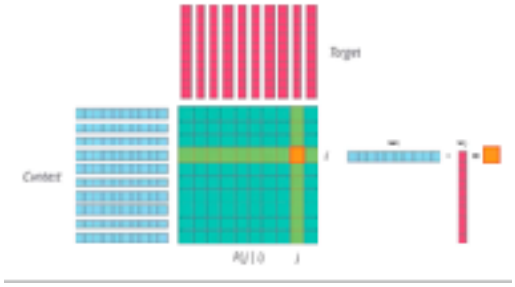
Continuous bag-of-words (CBoW) is where you are given neighboring word and predict a word.

Skip-gram is where you are given the middle word and predict neighboring words. You pick any word from a sentence, convert into a one-hot encoded vector, then feed it into a DNN which is designed to predict a few surrounding words (ie: its context).

GloVe

GloVe (global vectors for word representation) is another variation on word2vec. It tries to directly optimize the vector representation of each word just using co- occurrence statistics, unlike word2vec which sets up an ancillary prediction task. First, the probably that word j appears in the context of word i is computed, pj given i for all word pairs ij in a given corpus (context, meaning the words are close to each other in vector space). We count all such occurrences of i and j in our text collection, and then normalize account to get a probability. Then, a random vector is initialized for each word, actually two vectors. One for the word when it is acting as a context, and one when it is acting as the target.

Now, for any pair of words, ij, we want the dot product of their word vectors, w_i times w_j, to be equal to their co-occurrence probability. Using this as our goal and a suitable last function, we can iteratively optimize these word vectors. The result should be a set of vectors that capture the similarities and differences between individual words.



If you look at it from another point of view, we are essentially factorizing the co-occurrence probability matrix into two smaller matrices. This is the basic idea behind GloVe.

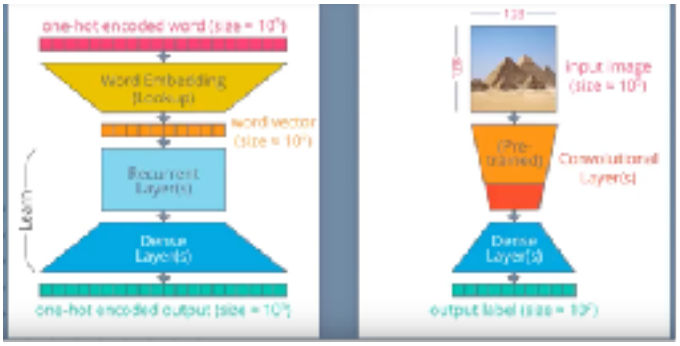
But why co-occurrence probabilities? Consider two context words, say ice and steam, and two target words, solid and water. You would come across solid more often in the context of ice than steam, right? But water could occur in either context with roughly equal probability. At least, that's what we would expect and that's exactly what co-occurrence probabilities reflect. Given a large corpus, you'll find that the ratio of P(solid | ice) to P(solid | steam) is much greater than one, while the ratio of P(water | ice) and P(water | steam) is close to one. Thus, we see that co-occurrence probabilities already exhibit some of the properties we want to capture. In fact, one refinement over using raw probability values is to optimize for the *ratio of probabilities*.

Now, there are a lot of subtleties here, not the least of which is the fact that the co-occurence probability matrix is huge. At the same time, co-occurrence probability values are typically very low, so it makes sense to work with the log of these values.

Embeddings for Deep Learning

To account for similarities between words in one context yet differences in other contexts, you need to add more dimensions to the embedded vector space. Example coffee and tea: one dimension will capture, say, the similarity that they are both beverages, while another dimension might capture that one is made from grounds and the other from leaves. Its common for a word prediction NLP deep network to use a word embedding layer with a few hundred dimensions, which is still much smaller than a one-hot encoded method that has a vector for each word in the vocabulary (sometimes 10's of thousands).

Also, if you learn the embedding as part of the model training process, you can obtain a representation that captures the dimensions that are most relevant for your task. This often adds complexity. So unless you're building a model for a very narrow application like one that deals with medical terminology, you can use a pre-trained embedding as a look-up. For example, word2vec or GloVe. Then you only need to train the layer specific to your task.



Convolutional layers, which exploit the spatial relationships (like in words or image data), are used to reduce dimensionality (a small 128x128 res image has 16,000 pixels). Early stages and visual processing are often transferable across tasks, so it is common to use pre-trained layers (AlexNet, BTG16) and only learn the later layers.

An embedding look up for NLP is not unlike using pre-trained layers for computer vision — both are examples of transfer learning.

t-SNE

t-Distributed Stochastic Neighbor Embedding (t-SNE) is used for dimensionality reduction. It can map high dimensional vectors to lower dimensional space, like PCA, but with one amazing property: it tries to maintain relative distances between objects. The output can visualize clusters or groupings of words which is useful for understanding what the network has learned and debugging.

Lesson 5: Regular Expressions

RAW STRINGS
Put an r before the 'text string' to convert it to a raw string so no \n \t for new line or tab are interpreted but just read as raw string.

```
print(r'Hello\n\tWorld')
Hello\n\tWorld
```

Regexes use raw strings because regexes themselves use \ characters to indicate their own special characters.

The `re.compile(pattern)` function converts a regular expression `pattern` into a regular expression object. This allows us to save our regular expressions into objects that can be used later to perform pattern matching using various methods, such as `.match()`, `.search()`, `.findall()`, and `.finditer()`.

Method/Attribute	Purpose
match()	Determine if the RE matches at the beginning of the string.
search()	Scan through a string, looking for any location where this RE matches.
findall()	Find all substrings where the RE matches, and returns them as a list.
finditer()	Find all substrings where the RE matches, and returns them as an iterator.

Method/Attribute	Purpose
group()	Return the string matched by the RE
start()	Return the starting position of the match
end()	Return the ending position of the match
span()	Return a tuple containing the (start, end) positions of the match

```
import re
sample_text = 'Alice and WaLtEr BroWN are talking with wAlTer Jackson.'
regex = re.compile(r'WaLtEr BroWN')
matches = regex.finditer(sample_text)
for match in matches:
    print(match)
        # Using the span information from the match, print the match from the original string
        print('\nMatch from the original text:', sample_text[match.span()
[0]:match.span()[1]])
```

MetaCharacters: \

. ^ $ * + ? { } [ ] \ | ( )

To search directly for metacharacters as text, you need to place '\' before a metacharacter:

```
        regex = re.compile(r'\$25\.99')
```

The backslash can also be followed by various characters to signal various special sequences:

- `\d` - Matches any decimal digit; this is equivalent to the set [0-9]
- `\D` - Matches any non-digit character; this is equivalent to the set [^0-9]
- `\s` - Matches any whitespace character, this is equivalent to the set [\t\n\r\f\v]

    - white spaces, tabs (`\t`), newlines (`\n`), carriage returns (`\r`), form feeds (`\f`), and vertical tabs (`\v`). Notice that form feeds appear as `\x0c` and vertical tabs as `\x0b`.
- `\S` - Matches any non-whitespace character; this is equivalent to the set [^ \t\n\r\f\v]
- `\w` - Matches any alphanumeric character and the underscore; this is equivalent to the set [a-zA-Z0-9_]
- `\W` - Matches any non-alphanumeric character; this is equivalent to the set [^a-zA-Z0-9_]

eg: Print numbers between whitespaces

Sample text
```
sample_text = '''
123\t45\t7895
1\t222\t33
'''
```
Print sample_text
```
print('Sample Text:\n', sample_text)
```

Find numbers by finding white spaces around them:
```
regex = re.compile(r'\s')
matches = regex.finditer(sample_text)
```
Write a loop to print all the numbers found in the original string
```
counter = 0
for match in matches:
    print(match)
    if counter != 0:
        start_idx = match.start()
        print('\nNumber from text: ', sample_text[end_idx:start_idx])
    end_idx = match.end()
    counter += 1
```

eg: Find emails using \w
```
sample_text = '''
John Sanders: j.s@email.com
Alice Walters: a.w@email.com
Mary Jones: m.j@email.com
'''

regex = re.compile(r'\w\.\w@email.com')
matches = regex.finditer(sample_text)
[match for match in matches]
```

WORD BOUNDARIES: \b

A boundary is defined as a white space, a non-alphanumeric character, or the beginning or end of a string.

Sample text
```
sample_text = 'The biology class will meet in the first floor classroom to learn about
Theria, a subclass of mammals.'
```

Searching the word directly (without word boundaries \b):
```
regex = re.compile(r'class')
matches = regex.finditer(sample_text)
[match for match in matches]     # returns 3 matches
```

With a leading boundary:
```
regex = re.compile(r'\bclass')
matches = regex.finditer(sample_text)
[match for match in matches]     # returns 2 matches
```

With a trailing boundary:
```
regex = re.compile(r'class\b')
```

```
matches = regex.finditer(sample_text)
[match for match in matches]     # returns 2 matches
```

With a both boundaries:

```
regex = re.compile(r'\bclass\b')
matches = regex.finditer(sample_text)
[match for match in matches]     # returns 1 match
```

eg: Match 3-letter words:

Sample text

```
sample_text = 'John went to the store in his car, but forgot to buy bread.'
regex = re.compile(r'\b\w\w\w\b')
matches = regex.finditer(sample_text)
[match for match in matches]
```

NOT A WORD BOUNDARY: \B

To return a match at the end of word, use \B at the beginning and vice versa. T get a match in the *middle* of a word, use \B…\B

eg: Finding last digits

Write code that uses a regular expression to count how many numbers (greater than 3), have 3 as their last digit. For example, 93 is greater than 3 and its last digit is 3, so your code should count this number as a match. However, the number 3 by itself should not be counted as a match.

Sample text

```
sample_text = '203 3 403 687 283 234 983 345 23 3 74 978'
regex = re.compile(r'\B3\b')
matches = regex.finditer(sample_text)
```

Print all the matches

```
counter = 0
for match in matches:
    print(match)
    counter += 1
```

Print the total number of matches

```
print('\nTotal matches: ', counter)
```

MetaCharacters: . ^ $

The dot (`.`) matches any character except for newline (`\n`) characters.

Sample text

```
sample_text = '''
Mr. Brown: 555-123-4567
Mrs. Smith: 455 555 4549
Mr. Jackson: 655-777-7346
Ms. Wilson: (555)999-8464
'''
```

Use . to match any character in between the digits:

```
regex = re.compile(r'\d\d\d.\d\d\d.\d\d\d\d')
```

The caret (`^`) is used to match a sequence of characters when they appear at the **beginning** of a string.

- **Outside** of a character set, the caret matches a sequence of characters when they are located at the beginning of a string.
    - `^[6-9]` will match any beginning character that is a `6`, `7`, `8`, or `9`.
- When the caret (`^`) appears at the **beginning** of a character set it **negates** the set. This means it matches everything that is **not** in that character set.

- For example, the regular expression `[^6-9]` will match any character that is **not** a `6`, `7`, `8`, or `9`. Similarly, the regular expression `[^a-zA-Z]`will match any character that is **not** a lowercase or uppercase letter.

The dollar sign (`$`) is used to match a sequence of characters when they appear at the **end** of a string.

Character Sets: { } [ ]

`{#}` is used to specify the number of copies of the previous regex to match.

```
sample_text = as above…
regex = re.compile(r'\d{3}.\d{3}.\d{4}')
```

- You can use the qualifier `{m,n}` in your regular expression. This qualifier means there must be at least `m` repetitions, and at most `n` repetitions of the previous regular expression. For example, `a/{1,3}b` will match `a/b`, `a//b`, and `a///b`. It won't match `ab`, which has no slashes, or `a////b`, which has four slashes.

Character sets are specified using the `[]` metacharacters and are used to indicate a set of characters that you wish to match.

Suppose we only wanted to find phone numbers in which the groups of digits were separated by either a dash (–) or a white space ( ):

```
sample_text = as above…
regex = re.compile(r'\d{3}[- ]\d{3}[- ]\d{4}')
```

Find phone numbers with area codes that either begin with 4 or 6:

```
sample_text = as above…
regex = re.compile(r'[46]/d{2}[- ]\d{3}[- ]\d{4}')
```

Find phone numbers with specific last digits 6, 7, 8 or 9:

```
sample_text = as above…
regex = re.compile(r'\d{3}.\d{3}.\d{3}[6-9]')
```

Find phone numbers that DO NOT end in 6, 7, 8 or 9:

```
sample_text = as above…
regex = re.compile(r'\d{3}.\d{3}.\d{3}[^6-9]')
```

Find phone numbers with country codes:

```
# Sample text
sample_text = '''
Mr. Brown: +1-555-123-4567
Mrs. Smith: +61 455 555 4549
Mr. Jackson: +375-655-777-7346
Ms. Wilson: +213(555)999-8464
'''
regex = re.compile(r'\+\d{1,3}.\d{3}.\d{3}.\d{4}')
```

MetaCharacters: * + ? | ( )

The `?` metacharacter signifies the character to be found is optional

- The `?` will match 0 or 1 repetitions of the preceding regular expression.
- For example, the regular expression `ab?` will match either `a` or `ab`.

```
# Sample text
sample_text = '''
Mt Everest: Height 8,848 m
Mt. K2: Height 8,611 m
Mt Kangchenjunga: Height 8,586 m
Mt. Lhotse: Height 8,516 m
```

```
'''
```

Find all abbreviations of mountain:

```
regex = re.compile(r'Mt\.?')
```

The `*` metacharacter, matches 0 or more repetitions of the preceding regular expression. In other words, it matches 0 or as many repetitions as possible of the preceding regular expression.

- For example, the regular expression `ab*` will match `a` or `a` followed by any number of `b`'s, such as `ab` or `abbbbb`.

Continue to match all mountain names:

```
regex = re.compile(r'Mt\.?\s\[A-Z]\w*')
```

The `( )` metacharacters group together the expressions contained inside of them.

- For example, we saw before that `ab*` will match `a` or `a` followed by any number of `b`'s, such as `ab` or `abbbbb`.
- But, if you put `ab` inside a parenthesis to define the **group** `(ab)`, then `(ab)*` will match zero or more repetitions of `ab`, for example `ab` or `abababab`.
- You can repeat the contents of a group with any repeating qualifier, such as `*, ?, or {m}` that we have seen before.

```
>>> p = re.compile('(ab)*')
>>> print p.match('abababababab').span()
(0, 10)
```

The OR `|` metacharacter can be used within a group to be able to select between two expressions:

Sample text

```
sample_text = '''
Mt Everest: Height 8,848 m
Mt. K2: Height 8,611 m
Mt Kangchenjunga: Height 8,586 m
Mt. Lhotse: Height 8,516 m
Mnt makalu: Height 8,485 m
'''
```

```
regex = re.compile(r'(Mt|Mnt)\.?\s\[a-zA-Z]\w*')
```

```
>>> p = re.compile('(a(b)c)d')
>>> m = p.match('abcd')
>>> m.group(0)
'abcd'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
>>> m.group(2,1,2)
('b', 'abc', 'b')
```

eg: Match all emails

```
sample_text = '''
fake_email@fake-email.edu
fakeemail43@fake_email.com
fake891_email@fakemail.gov
52fake_email@FAKE_email.com.nl
'''
```

```
regex = re.compile(r'[a-z_0-9]+@[A-Za-z_-]+\.[a-z]+\.?[a-z]+')
```

Substitutions

Regex objects have the `.sub()` method that allows us to replace patterns within a string.

eg: Substitute all & in the sample_text with 'and'

```
sample_text = '''
Ben & Jerry
Jack & Jill
'''

regex = re.compile(r'&')
new_text = regex.sub(r'and', sample_text)
```

Method/Attribute	Purpose
`split()`	Split the string into a list, splitting it wherever the RE matches
`sub()`	Find all substrings where the RE matches, and replace them with a different string
`subn()`	Does the same thing as `sub()`, but returns the new string and the number of replacements

We can do more sophisticated substitutions by using groups.

eg: Group all the first, middle and last names separately

```
sample_text = '''
John David Smith
Alice Jackson
Mary Elizabeth Wilson
Mike Brown
'''
```

First brackets contain first name, [ ] and next ( ) are optional second name, last ( ) is last name:

```
regex = re.compile(r'([A-Za-z]+)[ ]?([A-Za-z]+)?[ ]([A-Za-z]+)')
```

We can now reference the groups individually from the Match Objects using .group(#). So match.(1) would return first name etc. Also, `.group(0)` (or equivalently `.group()`) selects all the groups at once.

```
for match in matches:
    print('\nFirst Name: '+ match.group(1))
    if match.group(2) is None:
        print('Middle Name: None')
    else:
        print('Middle Name: '+ match.group(2))
    print('Last Name: '+ match.group(3))
```

Now, that we know how to select groups individually for each match, we are ready to use the `.sub()` method to make substitutions.

eg: Replace all names with only first and last names

`regex.sub(r'string', sample_text)` will replace every match of the regex expression in the `sample_text` with the raw string `string`. We can refer to each group in the `string` by using the backslash. For example, `regex.sub(r'\1', sample_text)` will replace every match with the first group.

```
new_text = regex.sub(r'\1 \3', sample_text)  #gets first/last names only
```

Flags

The `re.compile(pattern, flags)` function, has a `flag` keyword that can be used to allow more flexibility. For example, the `re.IGNORECASE` flag can be used to perform **case-insensitive** matching.

```
sample_text = 'Alice and WaLtEr Brown are talking with wAlTer Jackson.'
regex = re.compile(r'walter', re.IGNORECASE)
```

Flag	Meaning
DOTALL, S	Make . match any character, including newlines
IGNORECASE, I	Do case-insensitive matches
LOCALE, L	Do a locale-aware match
MULTILINE, M	Multi-line matching, affecting ^ and $
VERBOSE, X	Enable verbose REs, which can be organized more cleanly and understandably.
UNICODE, U	Makes several escapes like \w, \b, \s and \d dependent on the Unicode character database.

https://docs.python.org/3/howto/regex.html

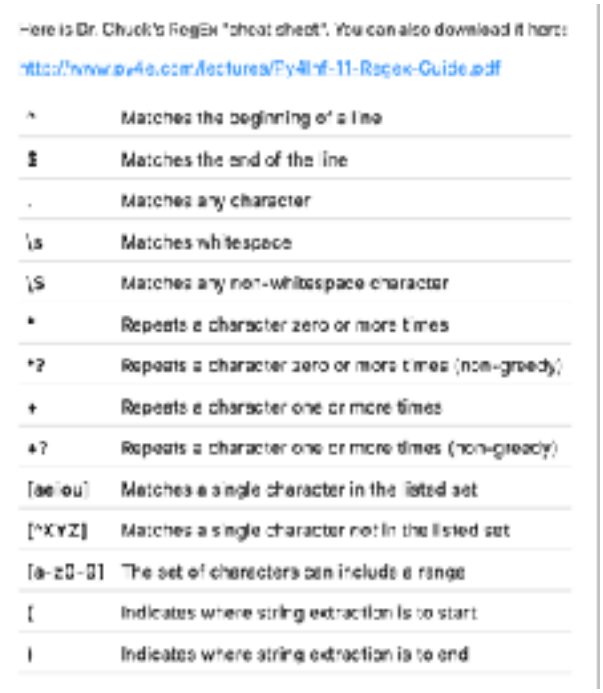
Accelerate notes:

```
######### DATA CLEANING ########################################
# .search - search whole string for match
# .match - match from the start of the string
# .findall - finds all matches and returns a list of strings
# .finditer - finds all matches but returns a iterable object
# .sub - can be used to substitute text
# .split - used to split using a regex

m = re.search(pattern,s) # returns the match object, has info we might want
m.group() # returns the matched pattern
m.span() # returns the index of the position
re.findall(r"[csm]at",s) # finds all combinatinons of cat/sat/mat
[ m.span() for m in re.finditer(r"[csm]at",s) ] # list comp to use finditer
      # returns start/end index tuples
re.findall(r"[^c]at",s) # match any chars except c which is what the ^ does
#-------------------
import string
alphabet = string.ascii_letters # upper and lowercase
re.search("[e-z]+",alphabet) # only matches e-z lowercase
re.search("[f-zA-C]+",alphabet) #match lower case f to z and upper case A-C
#-------------------
s1 = "The rainbow has many colors."
s2 = "The rainbow has many colours."
re.findall('color|colour',s1) # returns ['color']
#-------------------

######### QUANTIFIERS ########################################
# `*` - zero of more  times
# `+` - one or more times
# `?` -  one or zero times.
# `{ n }` - match exactly n times
# `{ n , }` - match at least n times
# `{ n , m }` - match between n to m times
s = "The rainbow has many colors but not the colour silver"
```

re.findall('colou?r',s) # returns ['color', 'colour']

######### META CHARACTERS ##
`\d` - match any digit same as [0-9]
`\w` - match any word char (a-z, A-Z, 0-9 and _'s)
`\s` - match white space (spaces, tabs...)
`\t` - match tab only
`\D` - match anything but digits same, is true of the above expresions i.e \W is anything but word chars.
`.` - match any characters

######### ANCHORS ##
`^` - starts with
`$` - ends with
`\b` - word boundary



Here is Dr. Chuck's RegEx "cheat sheet". You can also download it here:

http://www.py4e.com/lectures/Py4Inf-11-Regex-Guide.pdf

^	Matches the beginning of a line
$	Matches the end of the line
.	Matches any character
\s	Matches whitespace
\S	Matches any non-whitespace character
*	Repeats a character zero or more times
*?	Repeats a character zero or more times (non-greedy)
+	Repeats a character one or more times
+?	Repeats a character one or more times (non-greedy)
[aeiou]	Matches a single character in the listed set
[^XYZ]	Matches a single character not in the listed set
[a-z0-9]	The set of characters can include a range
(	Indicates where string extraction is to start
)	Indicates where string extraction is to end

Regular Expression Basics	
.	Any character except newline
a	The character a
ab	The string ab
a|b	a or b
a*	0 or more a's
\	Escapes a special character

Regular Expression Quantifiers	
*	0 or more
+	1 or more
?	0 or 1
{2}	Exactly 2
{2, 5}	Between 2 and 5
{2,}	2 or more
{,5}	Up to 5

Default is greedy. Append ? for reluctant.

Regular Expression Groups	
(...)	Capturing group
(?P<Y>...)	Capturing group named Y
(?:...)	Non-capturing group
\Y	Match the Y'th captured group
(?P=Y)	Match the named group Y
(?#...)	Comment

Regular Expression Character Classes	
[ab-d]	One character of: a, b, c, d
[^ab-d]	One character except: a, b, c, d
[\b]	Backspace character
\d	One digit
\D	One non-digit
\s	One whitespace
\S	One non-whitespace
\w	One word character
\W	One non-word character

Regular Expression Assertions	
^	Start of string
\A	Start of string, ignores m flag
$	End of string
\Z	End of string, ignores m flag
\b	Word boundary
\B	Non-word boundary
(?=...)	Positive lookahead
(?!...)	Negative lookahead
(?<=...)	Positive lookbehind
(?<!...)	Negative lookbehind
(?(i)	Conditional

Regular Expression Flags	
i	Ignore case
m	^ and $ match start and end of line
s	. matches newline as well
x	Allow spaces and comments
L	Locale character classes
u	Unicode character classes
(?iLmsux)	Set flags within regex

Regular Expression Special Characters	
\n	Newline
\r	Carriage return
\t	Tab
\YYY	Octal character YYY
\xYY	Hexadecimal character YY

Regular Expression Replacement	
\g<0>	Insert entire match
\g<Y>	Insert match Y (name or number)
\Y	Insert group numbered Y

New to Debuggex? Check out the regex tester!

Using re.VERBOSE

By now you've probably noticed that regular expressions are a very compact notation, but they're not terribly readable. REs of moderate complexity can become lengthy collections of backslashes, parentheses, and metacharacters, making them difficult to read and understand.

For such REs, specifying the `re.VERBOSE` flag when compiling the regular expression can be helpful, because it allows you to format the regular expression more clearly.

The `re.VERBOSE` flag has several effects. Whitespace in the regular expression that *isn't* inside a character class is ignored. This means that an expression such as `dog | cat` is equivalent to the less readable `dog|cat`, but `[a b]` will still match the characters `'a'`, `'b'`, or a space. In addition, you can also put comments inside a RE; comments extend from a `#` character to the next newline. When used with triple-quoted strings, this enables REs to be formatted more neatly:

```
pat = re.compile(r"""
 \s*                 # Skip leading whitespace
 (?P<header>[^:]+)   # Header name
 \s* :               # Whitespace, and a colon
 (?P<value>.*?)      # The header's value -- *? used to
                     # lose the following trailing whitespace
 \s*$                # Trailing whitespace to end-of-line
""", re.VERBOSE)
```

This is far more readable than:

```
pat = re.compile(r"\s*(?P<header>[^:]+)\s*:(?P<value>.*?)\s*$")
```

Lesson 5: Beautiful Soup

In BeautifulSoup, the **parser** is a piece of software whose primary job is to build a data structure in the form of a hierarchical tree that gives a structural representation of the HTML or XML file. In other words, the parser divides these complex files into simpler parts while keeping track of how these parts are related to each other.

The `lxml` parser can be used to parse both HTML and XML files and has the advantage of being very fast. In order to use the `lxml` parser, you must have `lxml` installed. You can install the `lxml` parser by using the following command in your terminal:

```
$ pip install lxml
```

If you're working with perfectly formatted HTML or XML files (*i.e.* files that don't contain any missing information or mistakes) then, in the majority of cases, your choice of parser shouldn't really matter. However, if the files you are working with have missing information or mistakes, then your choice of parser will matter because each parser has different rules for dealing with missing information or mistakes. Consequently, in these cases, different parsers will create different parse trees for the same document. You can take a look at the **differences between parsers**, in the BeautifulSoup documentation, for details.

Parsing HTML

1. Pass the document into a Beautiful Soup constructor:

```
# Import BeautifulSoup
from bs4 import BeautifulSoup
# Open the HTML file and create a BeautifulSoup Object
with open('./sample.html') as f:
    page_content = BeautifulSoup(f, 'lxml')
# Print the BeautifulSoup Object (if you want)
print(page_content.prettify())  # .prettify add all the indents etc.
```

Navigating the Parse Tree

2. Navigate the parse tree created by BeautifulSoup by accessing the HTML or XML tags:
- We can access the tags as if they were attributes of the BeautifulSoup object.
- We will access the `<head>` tag in our `page_content` object by using `page_content.head`
- This returns a Tag object that we will save in the `page_head` variable.

```
# Access the head tag
page_head = page_content.head
# Print the Tag Object
print(page_head.prettify())
```

```
<head>
 <title>
  AI For Trading
 </title>
 <meta charset="utf-8"/>
 <link href="./teststyle.css" rel="stylesheet"/>
 <style>
  .h2style {background-color: tomato;color: white;padding: 10px;}
 </style>
</head>
```

We can access **child tags** within the `<head>` tag as if they were attributes of the `page_head` object. For example, if we wanted to access the `<title>` tag within the `<head>` tag, we can use `page_head.title` or directly as `page_content.head.title`

```
<title>AI For Trading</title>
```

Print only text within tags:

```
print(page_content.head.title.get_text())
AI For Trading
```

GETTING ATTRIBUTES

An HTML or XML tag can have many attributes. For example, the tag:

```
<h1 id='intro'>
```

has the attribute `id` whose value is `'intro'`. BeautifulSoup allows us to get the value of a tag's attribute by treating the tag like a dictionary. For example, in the code below we get the value of the `id` attribute of the `<h1>` tag by using:

```
page_h1['id']
```

where `page_h1` is the Tag object that holds the contents of the `<h1>` tag. Let's see how this works in the code below:

```
h1_id_attr = page_h1[‘id']
print(h1_id_attr)

intro
```

eg: Get hyperlink

```
# Open the HTML file and create a BeautifulSoup Object
with open('./sample.html') as f:
    page_content = BeautifulSoup(f, 'lxml')
# Access the a tag
page_hyperlink = page_content.a
# Get the href attribute from the a tag
href_attr = page_hyperlink['href']
# Print the href attribute
print(href_attr)
```

Searching the Parse Tree

FINDING ALL TAGS

The `.find_all(filter)` method will search an entire document for the given `filter`. The `filter` can be a string containing the HTML or XML tag name, a tag attribute, or even a regular expression.

```
# Find all the h2 tags
h2_list = page_content.find_all('h2')
# Print the h2_list
print(h2_list)

[<h2 class="h2style" id="hub">Student Hub</h2>, <h2 class="h2style"
id="know">Knowledge</h2>]

# Print each tag in the h2_list
```

```
for tag in h2_list:
    print(tag)

<h2 class="h2style" id="hub">Student Hub</h2>
<h2 class="h2style" id="know">Knowledge</h2>
```

eg: Print all <p> tags

```
# Find all the p tags
p_list = page_content.find_all('p')
# Print each tag in the p_list
for tag in p_list:
    print(tag.prettify())
```

SEARCHING FOR MULTIPLE TAGS

```
# Print all the h2 and p tags
for tag in page_content.find_all(['h2', 'p']):
    print(tag.prettify())
```

SEARCHING FOR TAGS WITH PARTICULAR ATTRIBUTES

The `.find_all()` method also allows us to pass some arguments, such as the attribute of a tag, so that we can search the entire document for the exact tag we want. The first `<h2>` tag has the attribute `id="hub"`, while the second `<h2>` tag has the attribute `id="know"`. Let's suppose, we only wanted to search our `sample.html` document for the `<h2>` tag that had `id="know"`.

```
# Find the h2 tag with id = know
h2_know = page_content.find_all('h2', id = 'know')
# Print each item in the h2_know
for tag in h2_know:
    print(tag)

<h2 class="h2style" id="know">Knowledge</h2>
```

eg: Find all <h1> tags with the attribute id='intro'

```
# Print all the h1 tags with id = intro
for tag in page_content.find_all('h1', id='intro'):
    print(tag)
```

SEARCHING FOR ATTRIBUTES DIRECTLY

```
# Print all the h1 tags with id = intro
for tag in page_content.find_all(id='intro'):
    print(tag)
```

SEARCHING BY `CLASS`

Let's suppose we wanted to find all the tags that had the attribute `class="h2style"`. Unfortunately, in this case, we can't simply pass this attribute to the `.find_all()` method. The reason is that the **CSS** attribute, `class`, is a reserved word in Python. Therefore, using `class` as a keyword argument in the `.find_all()` method, will give you a syntax error. To get around this problem, BeautifulSoup has implemented the keyword `class_`

```
# Print the tags that have the attribute class_ = 'h2style'
for tag in page_content.find_all(class_ = 'h2style'):
    print(tag)

<h2 class="h2style" id="hub">Student Hub</h2>
<h2 class="h2style" id="know">Knowledge</h2>
```

eg: Find all attributes class=‘section’

```
# Print the tags that have the attribute class_ = 'section'
for tag in page_content.find_all(class_='section'):
    print(tag.prettify())
```

SEARCHING WITH REGULAR EXPRESSIONS

Use regex to find all the tags whose names contain the letter `i` and print tag names:

```
# Print only the tag names of all the tags whose name contain the letter i
for tag in page_content.find_all(re.compile(r'i')):
    print(tag.name)
```

eg: Find All Tags The Start With The Letter `h`

```
# Print only the tag names of all the tags whose names start with the letter h
for tag in page_content.find_all(re.compile(r'^h')):
    print(tag.name)
```

CHILDREN TAGS

```
# Open the HTML file and print the head tag
with open('./sample2.html') as f:
    print(BeautifulSoup(f, ‘lxml').prettify())
```

The `<html>` tag contains some child tags. For example, the `<head>` tag is a direct child of the `<html>` tag. Similarly, the `<title>` tag is a direct child of the `<head>` tag. We also see that the `<title>` tag itself has a child, namely the string `'AI For Trading'`.

We already saw we can access a child directly this way:

```
with open('./sample2.html') as f:
    print(BeautifulSoup(f, 'lxml').head.title.get_text())
```

```
AI For Trading
```

We can view a tag's children by using the `.contents` attribute of the Tag object. The `.contents` attribute returns a list with all the tag's children.

```
# Access the head tag
page_head = page_content.head
# Print the children of the head tag
print(page_head.contents)
# Print the number of children of the head tag
print('\nThe <head> tag contains {} children'.format(len(page_head.contents)))
```

```
[<title>AI For Trading</title>, <meta charset="utf-8"/>, <link href="./
teststyle.css" rel="stylesheet"/>, <style>.h2style {background-color:
tomato;color: white;padding: 10px;}</style>]

The <head> tag contains 4 children
```

Instead of getting a tag's children as a list, we can also get an iterator that we loop over by using the `.children` attribute.

```
# Print the children of the head tag
for child in page_content.head.children:
    print(child)
```

```
<title>AI For Trading</title>
<meta charset="utf-8"/>
<link href="./teststyle.css" rel="stylesheet"/>
```

```
<style>.h2style {background-color: tomato;color: white;padding: 10px;}</
style>
```

RECURSIVE SEARCH

The tag `.find_all()` method will search all the tag's children, its children's children, and so on. However, there will be times where you only want BeautifulSoup to search a tag's direct children. To do this, we can pass the `recursive=False` argument.

The `<head>` tag is directly beneath the `<html>` tag and that the `<title>` tag is directly beneath the `<head>` tag. Even though the `<title>` tag is beneath the `<html>` tag, it's **not** directly beneath it, because the `<head>` tag is in between them.

Let's restrict our search to only look at the `<html>` tag's direct children, by using the `recursive=False` argument:

```
# Search the html tag's direct children for the title tag
for tag in page_content.html.find_all('title', recursive=False):
    print(tag)
```

no matches found….

eg: `recursive=False` search For The `<head>` Tag

```
# Search the html tag's direct children for the head tag
for tag in page_content.html.find_all('head', recursive=False):
    print(tag.prettify())
```

```
<head>
 <title>
  AI For Trading
 </title>
 <meta charset="utf-8"/>
 <link href="./teststyle.css" rel="stylesheet"/>
 <style>
  .h2style {background-color: tomato;color: white;padding: 10px;}
 </style>
</head>
```

The Requests Library

Unfortunately, Beautifulsoup can't access websites directly. Therefore, in order to access websites, we will use Python's `requests` library. The `requests` library allows us to send web requests and get a website's HTML data. Once the `requests` library gets us the HTML data, we can use Beautifulsoup, just as we did before, to extract the data we want.

```
import requests
# Create a Response object:
r = requests.get('https://twitter.com/AIForTrading1')
# Get HTML data:
html_data = r.text

# Create a BeautifulSoup Object
page_content = BeautifulSoup(html_data, 'lxml')
print(page_content.prettify())
```

We need to know which tags contain our tweets. In order to figure this out, we need to inspect our webpage using our web browser. Hover over a tweet, right-click and select 'Inspect Element'. Our

tweet is contained within a `<p>` tag; and that this `<p>` tag is a child tag of a `<div>` tag with `class="js-tweet-text-container"`

```
# Find all the <div> tags that have a class="js-tweet-text-container" attribute
tweets = page_content.find_all('div', class_='js-tweet-text-container')
# Set counter
counter = 1
# Print each tweet by accessing the <p> tag inside the above <div> tags using the `get_text()` method
for tweet in tweets:
    print('{}. {}'.format(counter, tweet.p.get_text()))
    counter += 1
```

Twitter will only return 20 tweets per request. In order to get more than 20 tweets we need to use the `count` parameter in our web request. The `requests` library allows you to provide arguments to a web request by using the `params` keyword argument in the `.get()` function. So to get say 50 tweets, we need to set `count=50` to our web request:

```
requests.get('https://twitter.com/AIForTrading1', params={'count':'50'})
```

eg: Get all tweets from 'https://twitter.com/AIForTrading1'

```
# Get HTML data
html_data = requests.get('https://twitter.com/AIForTrading1',
params={'count':'50'}).text

# Create a BeautifulSoup Object
page_content = BeautifulSoup(html_data, 'lxml')

# Find all the <div> tags that have a class="js-tweet-text-container" attribute
tweets = page_content.find_all('div', class_='js-tweet-text-container')

# Print all tweets. Print the counter next to each tweet as well
counter = 1
for tweet in tweets:
    print('{}. {}'.format(counter, tweet.p.get_text()))
    counter +=1
```

****

In order to parse a document as XML, you can use `lxml`'s XML parser by passing in `xml` as the second argument to the BeautifulSoup constructor:

```
page_content = BeautifulSoup(xml_file, 'xml')
```

Lesson 6: Basic NLP Analysis

Readability

MEASURES OF COMPLEXITY

Longer sentences tend to be less readable.
Longer words tend to be less readable.

Two common readability indices are:

- The Flesch-Kincaid Grade index
- The Gunning-Fog Grade index

FLESCH-KINCAID GRADE INDEX

Combines the average length of sentences and average syllable counts of all the words:

$$= 0.39\left(\frac{\#\text{ words}}{\#\text{ sentences}}\right) + 11.8\left(\frac{\#\text{ syllables}}{\#\text{ words}}\right) - 15.59$$

GUNNING-FOG GRADE INDEX

Combines the average length of sentences and a fraction of hard words (defined as 3 syllables or more):

$$= 0.4\left[\frac{\#\text{ words}}{\#\text{ sentences}} + 100\left(\frac{\#\text{ hard words}}{\#\text{ words}}\right)\right]$$

Harry Potter has a grade around 8, while a Phd physics paper could exceed 16.

EXERCISE

```
# import dependencies
import nltk
from nltk.stem import WordNetLemmatizer, SnoballStemmer
from collection import Counter
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.tokenize import sent_tokenize
from syllable_count import syllable_count
from nltk.tokenize import RegexpTokenizer
# downloads
nltk.download('wordnet')
nltk.download('punkt')  # for sent_tokenize
nltk.download('stopwords')
# assign objects
sno = SnowballStemmer('english')
wnl = WordNetLemmatizer()
word_tokenizer = RegexpTokenizer(r'[^\d\W]+')  # no non-letters; symbols

# what the hell is this?
def word_tokenize(sent):
      return [w for w in word_tokenizer.tokenize(sent) if w.isalpha()]

# Define Functions
# Functions for Flesch-Kincaid index
def sentence_count(text):
    return len(sent_tokenizer.tokenize(text))
def word_count(sent):
    return len([ w for w in word_tokenize(sent)])
def hard_word_count(sent):
    return len([ w for w in word_tokenize(sent) \
                if syllable_count(wnl.lemmatize(w, pos='v'))>=3 ])
# Flesch-Kincaid Grade index calculation
def flesch_index(text):
    sentences = sent_tokenize(text)
    total_sentences = len(sentences)
    total_words = np.sum([ word_count(s) for s in sentences ])
```

```
    total_syllables = np.sum([ np.sum([ syllable_count(w) for w in
                      word_tokenize(s) ])for s in sentences ])
    return 0.39*(total_words/total_sentences) + \
            11.8*(total_syllables/total_words) - 15.59
```

```
# Gunning-Fog Grade index
def fog_index(text):
    sentences = sent_tokenize(text)
    total_sentences = len(sentences)
    total_words = np.sum([ word_count(s) for s in sentences ])
    total_hard_words = np.sum([ hard_word_count(s) for s in sentences ])
    return 0.4*((total_words/total_sentences) + \
            100.0*(total_hard_words/total_words))
```

Bag-of-Words

Counting frequency of words (like a histogram) but including similar words grouped in the bag.

EXERCISES

Term Frequency-Inverse Document Frequency (TF-IDF)

Some words are naturally more frequent than others, so it is good to use TF-IDF to reduce the importance of words solely based on their frequency (ie: TD-IDF is a method to normalize the importance of words in the bag-of-words).

The tf–idf value increases proportionally to the number of times a word appears in the document and is offset by the number of documents in the corpus that contain the word, which helps to adjust for the fact that some words appear more frequently in general.

Term frequency can be calculated as a raw count, which we be the same as bag-of-words, but is frequently log-normalized as $\log(1 + f_{t,d})$.

With finance documents, we often don't care how long the document is, but we still need a way to normalize the word importance. He then says we can use the average word count for a document, then can divide the 'term freqeuncy' by the average 'term freq for the average word frequency. (wtf !!!!!??????)

$$\frac{1+\log f_{w,d}}{1+\log a_a}$$

Inverse document frequency is a measure of how much information the word provides, i.e., if it's common or rare across all documents. It is obtained by dividing the total number of documents by the number of documents containing the term, and then taking the logarithm of that quotient.

$$\mathrm{idf}(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|} \qquad \mathrm{idf}(w) = \log \frac{N_d}{\mathrm{df}_w}$$

with

- N: total number of documents in the corpus $N = |D|$
- $|\{d \in D : t \in d\}|$: number of documents where the term t appears (i.e., $\mathrm{tf}(t, d) \neq 0$). If the term is not in the corpus, this will lead to a division-by-zero. It is therefore common to adjust the denominator to $1 + |\{d \in D : t \in d\}|$.

TF-IDF is calculated as: TF x IDF

Similarity Metrics

Comparing documents over time is useful, beyond just sentiment or readability.

TF-IDF can be used to compare document similarity over time because TF-IDF are number vectors. One method is to use cosine similarity (ie: comparing two vectors in space, the angle between them is a measure of similarity — if the angle is small, the vectors are close to each other).

$$\text{Cosine Similarity} = \cos\theta = \frac{u \cdot v}{|u||v|}$$

This website has a nice plain-english with step-by-step concrete examples:
http://blog.christianperone.com/2011/09/machine-learning-text-feature-extraction-tf-idf-part-i/

The **first step** in modeling the document into a vector space is to **create a dictionary of terms present in documents** (ie: the vocabulary). To do that, you can simple select all terms from the document and convert it to a dimension in the vector space, but we know that there are some kind of words (stop words) that are not important and can be removed/excluded.

Now, we're going to use the **term-frequency** to represent each term in our vector space; the term-frequency is nothing more than a measure of how many times the terms present in our vocabulary $\mathrm{E}(t)$ are present in the documents $d3$ or $d4$.

term-frequency is a counting function which returns how many times the term 't' appears in the document 'd'.:

$$\mathrm{tf}(t, d) = \sum_{x \in d} \mathrm{fr}(x, t)$$

where the $\mathrm{fr}(x, t)$ is a simple function defined as:

$$\mathrm{fr}(x, t) = \begin{cases} 1, & \text{if } x = t \\ 0, & \text{otherwise} \end{cases}$$

Now we can **create the document vector**, which is represented by:

$$\vec{v_{d_n}} = (\mathrm{tf}(t_1, d_n), \mathrm{tf}(t_2, d_n), \mathrm{tf}(t_3, d_n), \ldots, \mathrm{tf}(t_n, d_n))$$

Each dimension of the document vector is represented by the term of the vocabulary, for example, the $\mathrm{tf}(t_1, d_2)$ represents the frequency-term of the term 1 or t_1 (which is our "blue" term of the vocabulary) in the document d_2.

Let's now show a concrete example of how the documents d_3 and d_4 are represented as vectors:

$$\vec{v_{d_3}} = (\mathrm{tf}(t_1, d_3), \mathrm{tf}(t_2, d_3), \mathrm{tf}(t_3, d_3), \ldots, \mathrm{tf}(t_m, d_3))$$
$$\vec{v_{d_4}} = (\mathrm{tf}(t_1, d_4), \mathrm{tf}(t_2, d_4), \mathrm{tf}(t_3, d_4), \ldots, \mathrm{tf}(t_m, d_4))$$

which evaluates to:

$$\vec{v_{d_3}} = (0, 1, 1, 1)$$
$$\vec{v_{d_4}} = (0, 2, 1, 0)$$

Since we have a collection of documents, now represented by vectors, we can represent them as a matrix with $|D| \times t$.

where: $|D|$ is the cardinality of the document space (how many documents we have)
F is the number of features (in our case represented by the vocabulary size)

An example of the matrix representation of the vectors described above is:

$$M_{|D| \times F} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 2 & 1 & 0 \end{bmatrix}$$

As you may have noted, these matrices representing the term frequencies tend to be very sparse (with majority of terms zeroed), and that's why you'll see a common representation of these matrix as sparse matrices.

Project 5: NLP on Financial Statements

Student Hub Formatting Tips

Hi @RyanR, of course! Here are a few Student Hub tips for writing code and text:

- For short code statements you can wrap your code in between this quotation marks (`). e.g. `code=cool` will result in: `code=cool`.
- For long code statements or errors wrap your code in between three quotation marks (```) Note that I put spaces in between them to be able to write them without creating a code snippet, but it shouldn't have spaces. Here is an example:

```
code=cool
long_code=cooler
```

- If you want to create cool unordered lists you can use an asterisk (*) followed by a space at the beginning of your sentence. For ordered lists use (1.).

Reviewer Notes | Comments

`GET_DOCUMENT_TYPE`

The function `get_document_type` returns the document type lowercased.

You can do it directly as well without using any for loop:

```
type_pattern = re.compile(r'<TYPE>([^\n]+)\b')
doc_type = type_pattern.findall(doc)
return doc_type[0].lower()
```

`GET_COSINE_SIMILARITY`

The function `get_cosine_similarity` calculates the cosine similarities for each neighboring TFIDF vector/document.

You can also do it using np.diag:

```
list(np.diag(cosine_similarity(tfidf_matrix, tfidf_matrix), k=1))
```

Student Hub | Knowledge Discussions

GET_CLEAN_FACTOR_AND_FORWARD_RETURNS

In Project 5, I want to intuitively understand the part of how the sentiment data is actually used to generate returns. In the code, we call get_clean_factor_and_forward_returns. I'm not sure how the sentiment data is manipulated, and how it is used in the next step.

Hi Hahnsang, the `alphalens.utils.get_clean_factor_and_forward_returns` is indeed a bit of a complicated function in alphalens. On the other hand, however, it lets you create an alphalens-friendly formatted dataframe you can then use to calculate factor returns, mean returns by quantiles, etc. etc.

In `get_clean_factor_and_forward_returns` you will make sure to get a date-aligned dataframe with the following metrics:

- **forward returns** (this is the same as running alphalens.utils.compute_forward_returns) for the periods you specify, e.g. [1, 5, 10] would calculate, by default, the forward returns for next day, next 5 days and next 10 days;
- **factor**, i.e. the value for the factor itself. In the specific case of Project 5, that would correspond to the cosine similarity metric you have calculated;
- **factor_quintile**, i.e. the quintile to which the factor metric belongs to (e.g. if the factor value is among the top 20%, you would have a quintile value of 5).

Please bear in mind that the **periods** argument can be tricky sometime, since it is affected by the time index of the pricing dataframe. In the example of Project 5, you will see that the clean_factor_and_forward returns dataframe will have a column named **1D**, suggesting that is the 1-day forward return (since the periods argument is set to [1]). **However**, since our pricing dataframe includes yearly price, the forward return actually corresponds to a 1-year return.

I also suggest you have a look at the documentation page. Let me know if this helps.

The **get_clean_factor_and_forward_returns** is indeed a quite complicated method in alphalens. Ultimately, however, it can be one of the best tools you can use, since it lets you create an alphalens-friendly dataframe you can then use to calculate factor returns, mean returns by quintiles, tear sheets, etc - **very very useful** if you don't want to struggle to format the data in a way that works for alphalens.

First of all you should notice that **get_clean_factor_and_forward_returns** returns a Multi-Index pandas Dataframe. The first index corresponds to dates (in the case of project 5 is the first of January of every year starting from 1994). The second one corresponds to the tickers of the securities for which we have a factor value in that specific date.

The columns of the dataframes are:

- **factor**, i.e. the factor value associated to a given stock at a given date. In the case of project 5, that corresponds to the cosine similarity between two consecutive 10-K documents;
- **forward return** - in the case of project 5 you can see a column called **1D** which actually corresponds to the 1-year (since we have daily data) forward return for a given stock.
- **quintile**, i.e. the bucket to which the factor value belongs to. For example, if a stock's factor value is in the top 20%, then quintile would be 5.

Part 2 answer:
Hi again! These are excellent questions - I had to go through the same effort to fully understand what we are doing. Let me structure my answer a bit more.

1. Alpha research
In this project we are not building a portfolio, but we are ultimately evaluating whether a certain hypothesis is true. In this specific case, we are assuming that if a company has a stable 10-K sentiment content (measured as tf-idf bag of words using a Loughran McDonald sentiment dictionary) over time (measured as cosine/jaccard similarity between consecutive 10-K documents), the company should have a higher return than another one whose 10-K sentiment content changed (measured by a lower cosine/jaccard similarity with the previous 10-K).

This means that the 10-K similarity is our factor. Remember that we have an alpha value for each stock in our universe and this value is intended to be proportional to the future performance of a stock. In other words, the higher alpha value, the higher the expectation for a stock's future performance. Suppose we have the following cosine similarities for three stocks A, B and C (at a given date d):

```
alpha_cos_sim =
  'A' : 0.96,
  'B' : 0.78,
  'C' : 0.33
```

This means that stock A's 10-K at date d is quite similar to the 10-K at date d-1 (i.e. the 10-K published one year before). As of now, these numbers only give us expectations in terms of future performance, but they are not associated to any weight (yet). In this particular case, our hypothesis tells us that we should expect stock A to have a better performance than stock B (and stock C). This is ultimately our hypothesis.

As explained in my previous comment, get_clean_factor_and_forward_returns will build a dataframe that, at every period, will give you the factor value (like in my example above), the future forward returns (for the period/s specified) and the quantiles' labels.

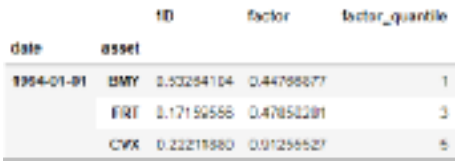
At this stage, you simply want to have a quick check on whether your hypothesis makes sense. Quantile analysis is actually a good way to do it. If your hypothesis is sound, in fact, you should expect stocks in the fifth quintile (i.e. top 20% by cosine similarity) to have performed better (in terms of returns) than stocks in, for example, the first quintile (i.e. bottom 20% by cosine similarity).

date	asset	1D	factor	factor_quantile
1994-01-01	BMY	0.53264104	0.44766877	1
	CVX	0.22211880	0.91255527	5
	FRT	0.17159556	0.47852201	3
1995-01-01	BMY	0.32152010	0.82108386	1
	CVX	0.28478159	0.91611711	3
	HON	0.45139755	0.94140116	5

In the picture beside, you can see the results of get_clean_factor_and_forward_returns for negative-sentiment 10-K cosine similarity for the stocks in the universe on Jan 1994 and Jan 1995.

Let's focus on the first date. In Jan 1994, company CVX had a negative-sentiment cosine similarity between its 1994 10-K and its 1993 10-K of 0.91255527. Because you only have three stocks per day (in this case), you can see that we are missing second and fourth quintile, but that doesn't really effect our reasoning.
Under column 1D you see the forward return for each stock. In this case the name 1D is a bit misleading, since we have yearly prices and 1D actually refers to a 1-year forward return. To double-check these numbers, let me show you the prices for CVX in Jan 1994 and Jan 1995.

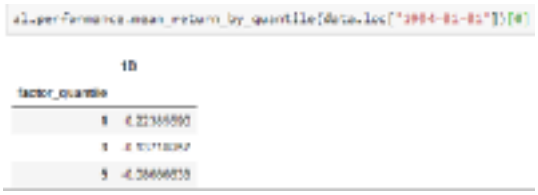
2. Mean returns by quantiles
Let's now concentrate on the values for Jan 1994. If we were to simply calculate the mean return for quantile, since we have only three stocks and three quantiles (with one stock in each), the mean return per quantile would simply be the return of each stock. In other words, the mean return for quantile 1 would correspond to return of BMY (i.e. 0.53264104).



date	asset	1D	factor	factor_quantile
1994-01-01	BMY	0.53264104	0.44766877	1
	FRT	0.17159556	0.47852201	3
	CVX	0.22211880	0.91255527	5

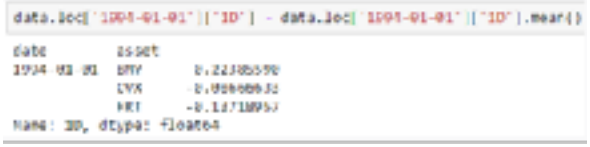
However, as you correctly pointed out, our factor should indeed tell us which stocks to buy/sell. Remember however that here we are in the context of **alpha research**, and we are not really interested in which weight we should give a certain stock. Our factor values tell us that CVX should do **relatively** better than FRT (and BMY), and FRT should do relatively better than BMY. If our hypothesis holds, we should then expect our CVX forward return to be higher than that of FRT and BMY (same reasoning). In the documentation for **mean_return_by_quantile** (here), you can see there is a parameter named *demeaned* (default is True).

This is a very smart trick that actually demean the returns (subtract sample mean from each return), allowing for longing/shorting stocks. Let me show you some numbers.



factor_quantile	1D
1	0.22366890
3	-0.13718953
5	-0.08648038

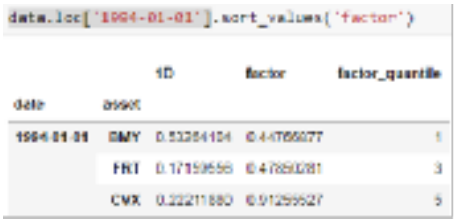
Beside you find the **mean returns** by quantile as calculated by alphalens, using *demeaned* = True. This is equivalent to manually demeaning the returns (see below).



```
data.loc['1994-01-01']['1D'] - data.loc['1994-01-01']['1D'].mean()

date        asset
1994-01-01  BMY      0.223886590
            CVX      0.089660832
            FRT     -0.137189543
Name: 1D, dtype: float64
```

Intuitively, you may have wanted to assign a proportionate weight to every stock (say w_i) and calculate the stock return (in this case corresponding to the quantile return) as w_i times the forward return of the stock. This however would already contain information about the weights you are willing to assign to each stock, but remember here we are interested in the context of

alpha research, and we purely want to compare quantiles (not assign a higher short weight to quintile 1 and a higher long weights to quintile 5). Read through this section again because I still find it quite complicated, but this is ultimately the most important part of this post.

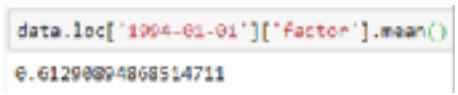


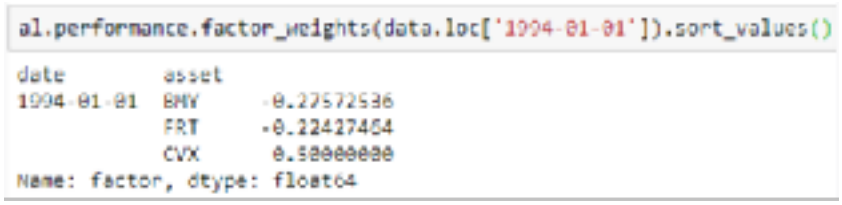
3. Factor returns
Ok, I hope you are still there. Now that we have evaluated our signal (and suppose we have found a nice monotonic relationships for our mean returns by quantile), we want to aggregate our numbers and calculate factor returns for our factor.

This is equivalent to creating a theoretical portfolio that longs and shorts stocks according to our factor value. In this case, you will have to assign a weight to each stock, and having the conditions that the portfolio is dollar neutral (i.e. the sum of the weights is zero).

Alphalens' performance_returns method (see documentation) set weights by using the method factor_weights (see documentation). Let's run it for our example (see below).

As you can see the weights are based on the demeaned factor values. CVX's factor value is largest than the factor mean for that date (i.e. 0.612908...), while the factor values for BMY and FRT are both smaller than the mean. Therefore, CVX will get all the long dollar amount (i.e. weight = 0.5), while BMY and FRT will get the short dollar amount (i.e. -0.5), in proportion to their factor value (BMY short weight is indeed larger than that of FRT).





The factor return for Jan 1994 would then simply be a weighted average of returns. Look beside to double check:

As you can see, the return of our factor on Jan 1994 is -0.0742977. This corresponds to the dot product (weighted average in this case) between the 1-year forward returns of our stock and their weights (defined proportionately to the alpha vector).

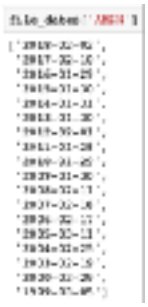
4. Alpha combination
As you correctly pointed out, in this project, we calculate multiple factors (negative, positive, litigious, etc.). In project 4 in term 1, if you remember, we had multiple factors and we combined them directly by doing a simple average. In future projects we are expected to employ machine learning algorithms to identify the optimal way to combine such alpha factors.
In general, remember that we will feed one single (combined) alpha vector to our optimization problem.

5. Multiple sentiments
In your last question, you ask how do we use different sentiments. Remember that in this case we are not really tracking the sentiment of the 10-Ks document, but rather how much they change from one period to another. The usage of sentiment words from Loughran McDonald dictionary makes the processing really convenient, because instead of using a very large vocabulary, we limit ourselves to words which are meant to carry a specific sentiment in the particular domain of financial markets.

I hope all of this helps! Thanks for the great question! (GiacomoS)

COSINE SIMILARITIES AND ALPHALENS - IS DATE ALIGNMENT CORRECT?

This question referts to **Project 5**. I am going to use an example to make myself - hopefully - more understandable. I will refer to the stock **AMZN** (i.e. Amazon).



After having transformed each 10-K in multiple sentiment tf-ifd vectors, we calculate the cosine similarity between one 10-K and the one before. Please notice that because in EDGAR the 10-K are listed from the newest to the oldest. In the case of AMZN, for example, the transformed 10-Ks are reported in the following order (side):

In my cosine similarity implementation (which passed the test, and the project's submission), I will return a list of numbers which has 1 value less than the list of dates for AMZN. This is because I am calculating the cosine similarity between 2018 and 2017, 2017 and 2016, ... and 2000 and 1999.

	date	sentiment	ticker	value
0	2017-01-01	negative	AMZN	0.95880691
1	2016-01-01	negative	AMZN	0.98130982
2	2015-01-01	negative	AMZN	0.99304193
3	2014-01-01	negative	AMZN	0.98852755
4	2013-01-01	negative	AMZN	0.97421473
5	2012-01-01	negative	AMZN	0.97580038
6	2011-01-01	negative	AMZN	0.97556871
7	2010-01-01	negative	AMZN	0.98098896
8	2009-01-01	negative	AMZN	0.93656818
9	2008-01-01	negative	AMZN	0.98794405
10	2007-01-01	negative	AMZN	0.98290655
11	2006-01-01	negative	AMZN	0.93571084
12	2005-01-01	negative	AMZN	0.95956395
13	2004-01-01	negative	AMZN	0.79791469
14	2003-01-01	negative	AMZN	0.95740880
15	2002-01-01	negative	AMZN	0.87578403
16	1999-01-01	negative	AMZN	0.92748994

Contrarily to my expectations, **however,** in the notebook, the assignment of a cosine similarity to value to its corresponding date seems off-set. In the example of AMZN we have the following dataframe (for the *negative* sentiment):

As you can see, we have a cosine similarity metric for 1999-01-01, but that value (0.92748994) actually corresponds to the similarity between the 1999 10-K and the 2000 10-K. In other words, we are claiming, in 1999, to know how much the 1999 10-K will be similar to that of 2000.

This mistake - but I am not sure it is such - has a **major impact** when we calculate the forward factor returns. For 1999, we get the following:

		1D	factor	factor_quantile
date	asset			
1999-01-01	AMZN	-0.76561277	0.92748994	5
	BMY	0.17108178	0.83884133	2
	CVX	0.06514705	0.83184781	4
	FRT	0.14572127	0.67885415	1

As explained in AINDT Term 1 - Lesson 22 - Zipline Pipeline, "when pipeline returns with a date of, e.g., 2016-01-07 this includes data that would be known as of before the **market open** on 2016-01-07. In this **specific case**, the table above would suggest that on 1999-01-01, AMZN would have had a 10-K whose cosine similarity with the previous year 10-K of 0.92748994. However, we know this **cannot be true** since Amazon wasn't listed in 1998. We also know that the value 0.92748994 corresponds to the similarity between the 1999 10-K and the 2000 10-K.

From a trading perspective, in addition, this would create a sort of lookahead bias since it means that in 1999 I know that the similarity of the following year (i.e. 2000) would be pretty high (~1.0).

I spent quite a long time reviewing this and I think there might be a misalignment of the date indexes.

Can you please have a look at this and let me know your thoughts?
Thanks a million in advance for your help!

Giacomo

JACCARD SIMILARITY
I would suggest you both to use sklearn function since it will be optimized for large data. But still for understanding, remember you are working on strings not so instead of minimum/maximum it will be intersection/union.

```
def DistJaccard(str1, str2):
    str1 = set(str1.split())
    str2 = set(str2.split())
    return float(len(str1 & str2)) / len(str1 | str2)
```

COSINE SIMILARITY
Replace the codes given in the template with the above codes for cosine similarity in the code template to fix latest 10-k release breaking the code.
Reference https://knowledge.udacity.com/questions/26587

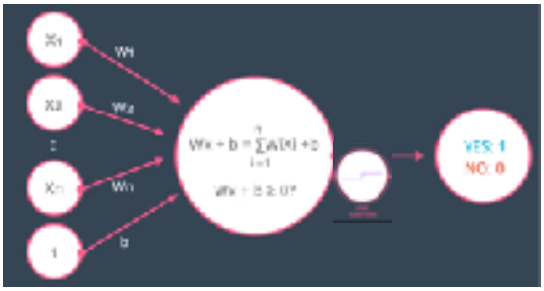
```
cosine_similarities_df_dict = {'date': [], 'ticker': [], 'sentiment': [], 'value': []}

for ticker, ten_k_sentiments in cosine_similarities.items():
    for sentiment_name, sentiment_values in ten_k_sentiments.items():
        for sentiment_values, sentiment_value in enumerate(sentiment_values[1:]): ###//
remove sentiment value for 2018
            cosine_similarities_df_dict['ticker'].append(ticker)
             cosine_similarities_df_dict['sentiment'].append(sentiment_name)
             cosine_similarities_df_dict[‘value'].append(sentiment_value)
             cosine_similarities_df_dict['date'].append(file_dates[ticker][2:]
[sentiment_values]) ###// exclude 2018

cosine_similarities_df = pd.DataFrame(cosine_similarities_df_dict)

cosine_similarities_df['date'] = pd.DatetimeIndex(cosine_similarities_df['date']).year

cosine_similarities_df['date'] = pd.to_datetime(cosine_similarities_df['date'],
format='%Y')

cosine_similarities_df.head()
```

MODULE 6: SENTIMENT ANALYSIS with NNs

Lesson 8: Neural Networks

Perceptrons

A node which calculates a linear regression or linear classification boundary taking the features as inputs (general case notation on right hand side).



Perceptrons as Logical Operators (AND, OR, NOT, XOR)

AND

OR

The OR perceptron is very similar to an AND perceptron. In the side image, the OR perceptron has the same line as the AND perceptron, except the line is shifted down. What can you do to the weights and/or bias to achieve this? — increase the weights (will move the dots up/right by multiplying them by a higher weight) or decrease the bias (will lower the line).

NOT

Unlike the other perceptrons we looked at, the NOT operation only cares about one input. The operation returns a `0` if the input is `1` and a `1` if it's a `0`. The other inputs to the perceptron are ignored.

XOR (EXCLUSIVE OR)

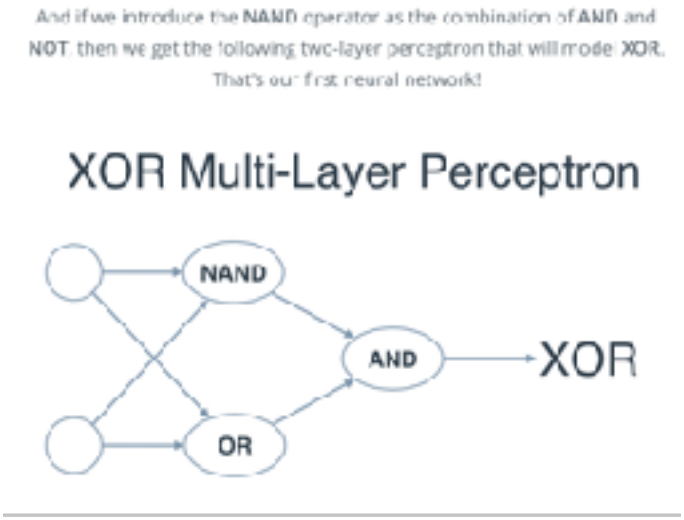
In the perceptron below we can see it is hard to separate the four points with a straight line. The XOR gate returns a true is one of the inputs is true while the other input is false (see right table). AND is both inputs true, OR is one input true, XOR is one true *and* one false (ie: not equal).

We can build a multi-layer perceptron from the AND, NOT, and OR perceptrons to create XOR logic. The neural network below contains 3 perceptrons, A, B, and C. The last one (AND) has been

given for you. The input to the neural network is from the first node. The output comes out of the last node.

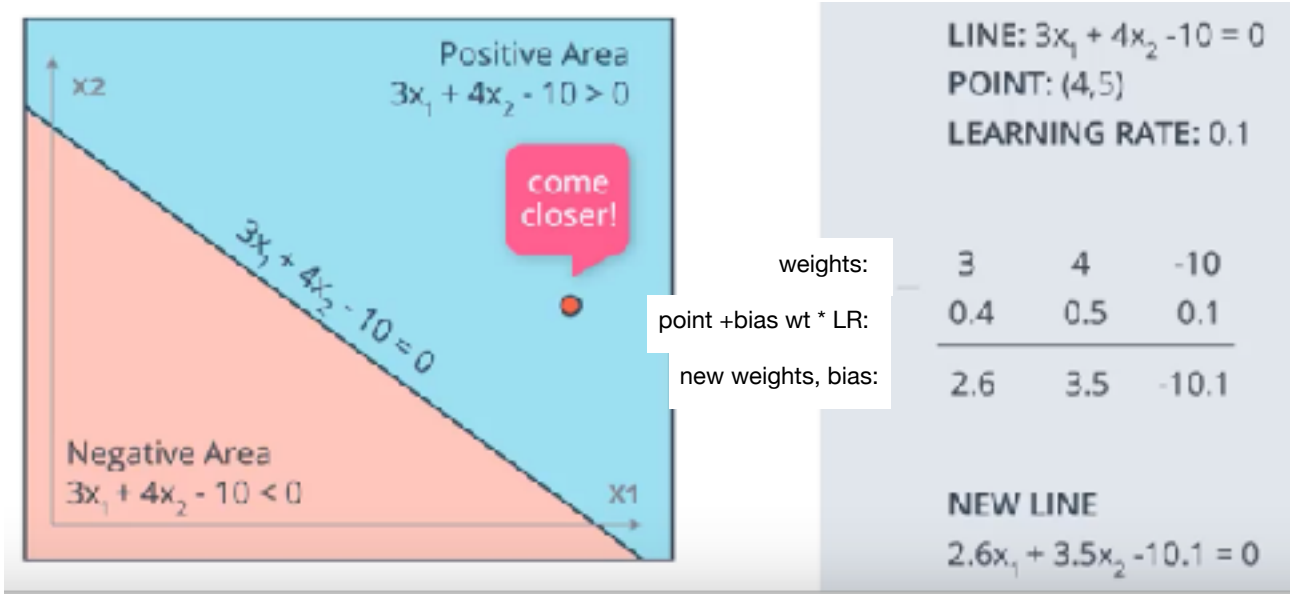
The multi-layer perceptron below calculates XOR. Each perceptron is a logic operation of AND, OR, and NOT. However, the perceptrons A, B, and C don't indicate their operation. In the following quiz, set the correct operations for the perceptrons to calculate XOR.

Good article/explanation: https://medium.com/@jayeshbahire/the-xor-problem-in-neural-networks-50006411840b



PERCEPTRON TRICK

The math to nudge a boundary line closer to a misclassified point adjusts the weights and bias from the line equation by adding(subtracting) the misclassified point coordinate values (4, 5) and the bias weight (=1) all multiplied by the learning rate (LR). If the misclassified point is above the line, subtract, otherwise add the adjustment.



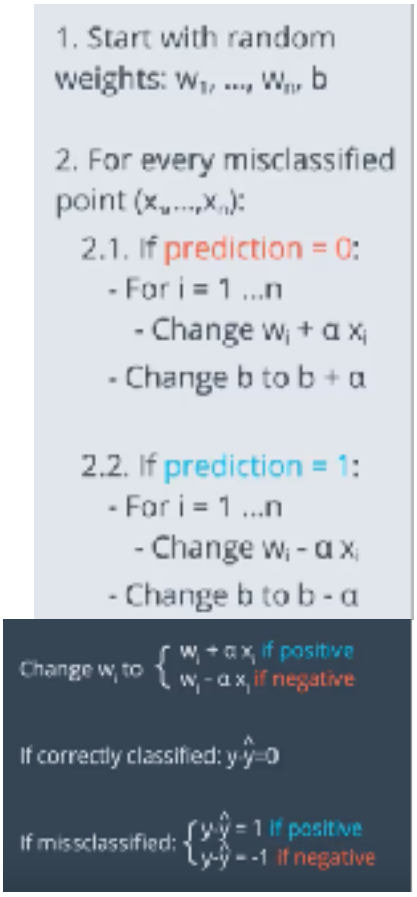
Perceptron Algorithm

PSEUDOCODE

1. Start with random weights: w1, w2, … wn, and the bias weight
   - this also gives us the boundary line and positive/negative areas

2. For every misclassified point (x1, x2, … xn):
   - <u>if prediction = 0</u>: (which means point is positive but in the negative area; we know this because we know it is misclassified. But how does computer know?)
     - For i = 1 … n:
       - update wi to wi+LR*xi (ie: wi -= LR*xi)
       - update b to b+LR
   - <u>if prediction = 1</u>: (which means point is negative but in the positive area; we know this because we know it is misclassified)
     - For i = 1 … n:
       - update wi to wi-LR*xi (ie: wi -= LR*xi)
       - update b to b-LR

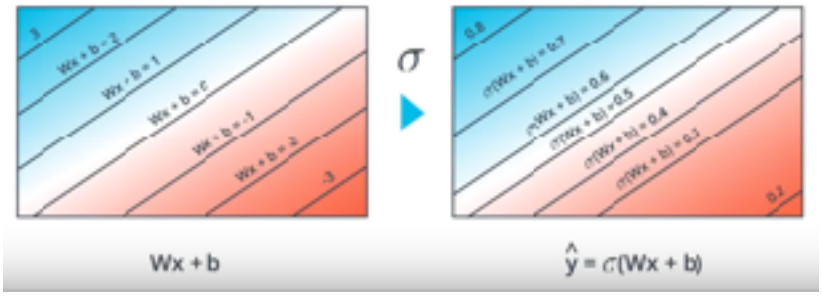
- repeat until some condition is met: eg, no more errors; errors meet min criteria; after 1000 attempts etc.
- Gray pseudocode is a bit confusing, this is clearer:



Error Function

SIGMOID ACTIVATION FUNCTION

The sigmoid function is defined as sigmoid(x) = 1/(1+e-x)



SOFTMAX ACTIVATION FUNCTION

Returns the probability of a classification for more than 2 classes. Probabilities must sum to one, so use softmax function to turn scores of each classes into probabilities. Softmax basically scales all scores by the total scores, but uses the exponential 'e' to keep all values positive:

Scores: 2, 1, 0; softmax for score 2 would be: $2 / (e^2 + e^1 + e^0)$

$$P(\text{class } i) = \frac{e^{Zi}}{e^{Z1} + \dots + e^{Zn}}$$

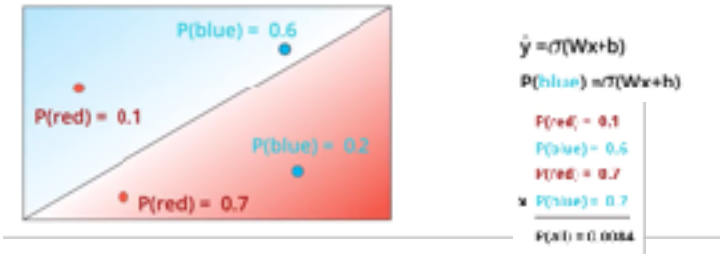
ONE-HOT ENCODING

Each class gets a vector of zeros and one 1 in a unique place in the vector to identify it.

Maximum Likelihood

Pick the model that gives the highest probabilities to the correct outcomes.

y_hat with sigmoid gives probability of each point. The product of all point probabilities is the probability of all points being correct.



So the model with more higher individual point probabilities is also better overall.

In this framework, the new objective function is to maximize P(all).

MAXIMIZING PROBABILITIES: CROSS-ENTROPY
Products have the problem of generating very small numbers when there are a lot of them. So use log to turn products into sums.

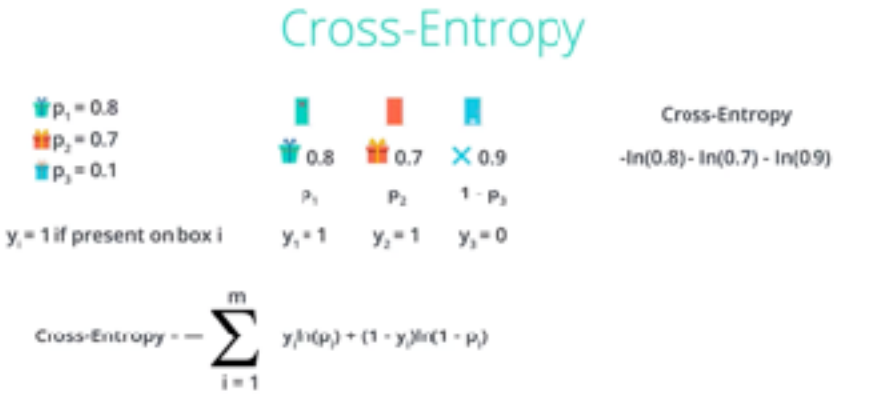
log(ab) = log(a) + log(b)

We will use ln (base 10) but it doesn't change anything. Also, log < 1 is a negative number, so need to take the negative ln.

The sum of negative ln is the '<u>cross-entropy</u>'

A good model has a low cross-entropy, bad has high, because the log of a large number is a small number and vice versa.

So now we can use cross-entropy as an objective function to minimize, as with other error functions.



Example:
p's are the probs of a present behind the door

The formula comprises the sum of negative cross-entropies.

So, if you calculate the cross-entropy (CE) of a gift being behind the doors (1, 1, 0) versus the probabilities (0.8, 0.7, 0.1):

CE[(1, 1, 0), (0.8, 0.7, 0.1)] = 0.69,

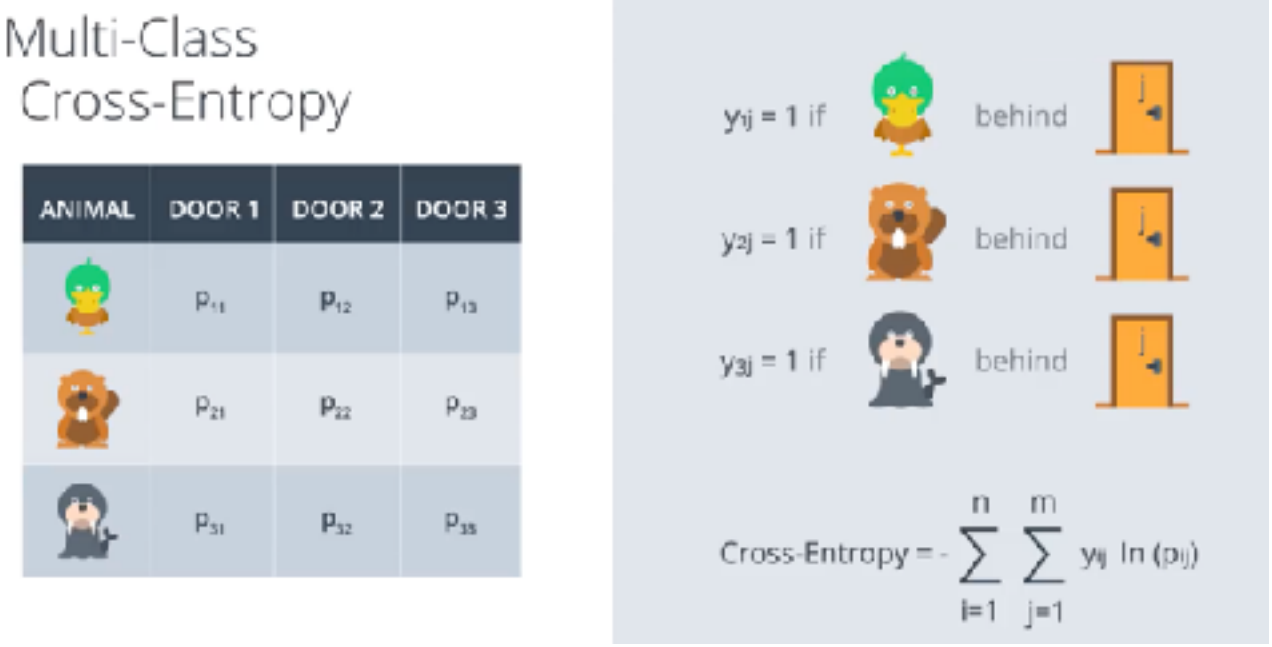
which is low because the gifts are where the probabilities indicate, and:

CE[(0, 1, 1), (0.8, 0.7, 0.1)] = 5.12 is high because they are not where the probs suggest

```
# Write a function that takes as input two lists Y, P,
# and returns the float corresponding to their cross-entropy.

def cross_entropy(Y, P):
    Y = np.float_(Y)
    P = np.float_(P)

    return -np.sum(Y*np.log(P) + (1 - Y)*np.log(1 - P))
```

MULTI-CLASS CROSS-ENTROPY
Instead of gift or no gift, you could have multiple type of things behind the doors. The sum of the probabilities of the classes for a given door must sum to one. The probabilities of a class behind each of the doors does not.

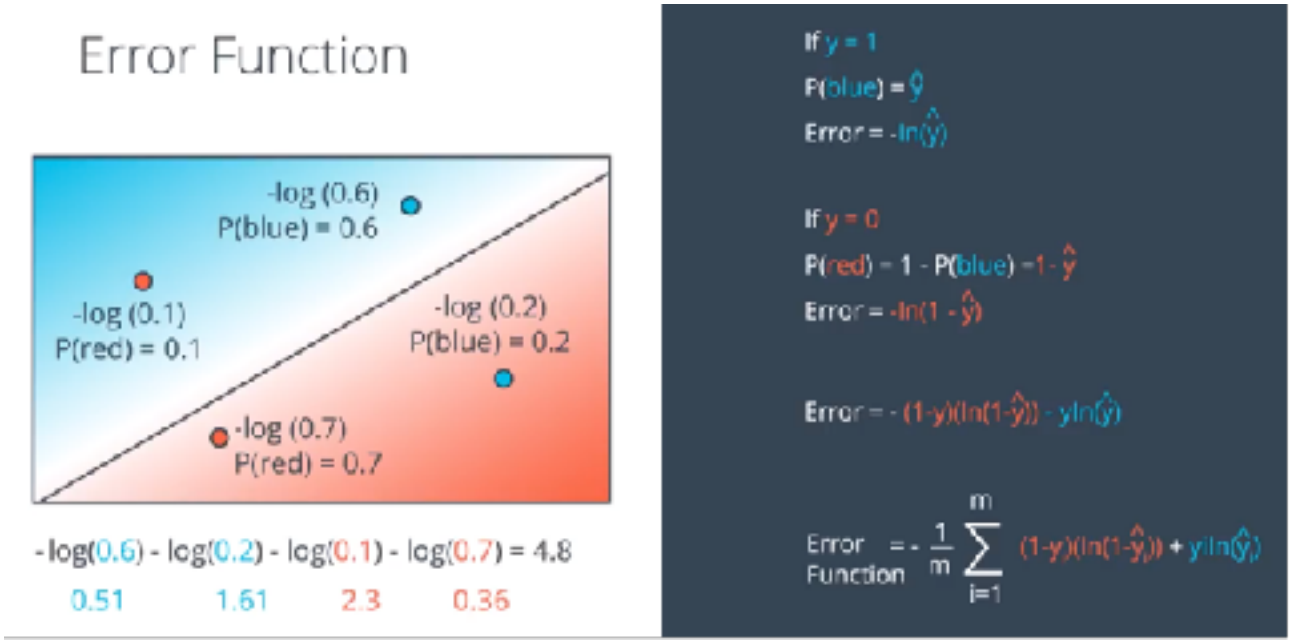
Generalized:



n doors; m classes

Logistic Regression

CALCULATING THE ERROR FUNCTION



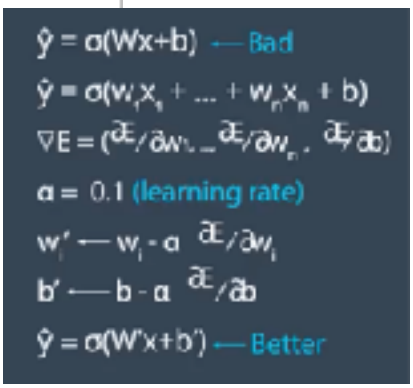
Substituting for y_hat, the full formula becomes:

$$E(W,b) = -\frac{1}{m}\sum_{i=1}^{m}(1-y_i)(\ln(1-\sigma(Wx^{(i)}+b)) + y_i\ln(\sigma(Wx^{(i)}+b))$$

For multi-class, the error function is:

$$-\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n} y_{ij}\ln(\hat{y}_{ij})$$

GRADIENT DESCENT
Take the negative of the gradient of the error function, which is found by calculating the partial derivatives of the error function wrt the weights and the bias.



<u>Compute the derivative of the error function</u>
The first thing to notice is that the sigmoid function has a really nice derivative, namely:

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

Our goal is to calculate the gradient of E, at a point x = (x1, … , xn), given by the partial derivatives:

$$\nabla E = \left(\frac{\partial}{\partial w_1}E, \cdots, \frac{\partial}{\partial w_n}E, \frac{\partial}{\partial b}E\right)$$

To simplify our calculations, we'll actually think of the error that each point produces, and calculate the derivative of this error. The total error, then, is the average of the errors at all the points. The error produced by each point is, simply:

$$E = -y \ln(\hat{y}) - (1-y) \ln(1-\hat{y})$$

The derivative of this error E wrt to the weights is:

$$\begin{aligned}
\frac{\partial}{\partial w_j} \hat{y} &= \frac{\partial}{\partial w_j} \sigma(Wx+b) \\
&= \sigma(Wx+b)(1-\sigma(Wx+b)) \cdot \frac{\partial}{\partial w_j}(Wx+b) \\
&= \hat{y}(1-\hat{y}) \cdot \frac{\partial}{\partial w_j}(Wx+b) \\
&= \hat{y}(1-\hat{y}) \cdot \frac{\partial}{\partial w_j}(w_1 x_1 + \cdots + w_j x_j + \cdots + w_n x_n - b) \\
&= \hat{y}(1-\hat{y}) \cdot x_j
\end{aligned}$$

Then calculate the derivative of the error E at a point x wrt the weight wj:

$$\begin{aligned}
\frac{\partial}{\partial w_j} E &= \frac{\partial}{\partial w_j} [-y \log(\hat{y}) - (1-y) \log(1-\hat{y})] \\
&= -y \frac{\partial}{\partial w_j} \log(\hat{y}) - (1-y) \frac{\partial}{\partial w_j} \log(1-\hat{y}) \\
&= -y \cdot \frac{1}{\hat{y}} \cdot \frac{\partial}{\partial w_j} \hat{y} - (1-y) \cdot \frac{1}{1-\hat{y}} \cdot \frac{\partial}{\partial w_j} (1-\hat{y}) \\
&= -y \cdot \frac{1}{\hat{y}} \cdot \hat{y}(1-\hat{y}) x_j - (1-y) \cdot \frac{1}{1-\hat{y}} \cdot (-1) \hat{y}(1-\hat{y}) x_j \\
&= -y(1-\hat{y}) \cdot x_j + (1-y) \hat{y} \cdot x_j \\
&= -(y-\hat{y}) x_j
\end{aligned}$$

And a similar derivation wrt to the bias ends up as:

$$\frac{\partial}{\partial b} E = -(y-\hat{y})$$

This actually tells us something very important. For a point with coordinates (x1, … , xn), label y and prediction y_hat, the gradient of the error function at that point is:

$$(-(y-\hat{y})x_1, \cdots, -(y-\hat{y})x_n, -(y-\hat{y}))$$

In summary, the gradient is:

$$\nabla E = -(y-\hat{y})(x_1, \ldots, x_n, 1).$$

If you think about it, this is fascinating. The gradient is actually a scalar times the coordinates of the point! And what is the scalar? Nothing less than a multiple of the difference between the label and the prediction. What significance does this have? It means larger errors (differences between y and y_hat) have steeper gradients, so get corrected more than smaller errors.

Gradient Descent Step

Since the gradient descent step simply consists in subtracting a multiple of the gradient of the error function at every point, then this updates the weights in the following way:

$$w_i' \leftarrow w_i - \alpha[-(y-\hat{y})x_i],$$

which is equivalent to

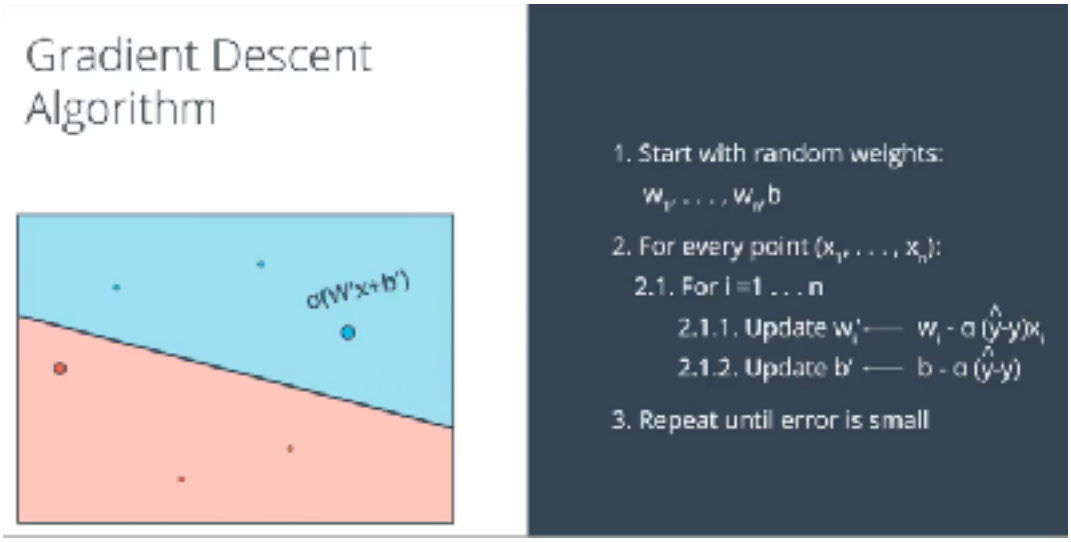
$$w_i' \leftarrow w_i + \alpha(y-\hat{y})x_i.$$

Similarly, it updates the bias in the following way:

$$b' \leftarrow b + \alpha(y-\hat{y}),$$

Note: Since we've taken the average of the errors, the term we are adding should be $\frac{1}{m} \cdot \alpha$ instead of α, but as α is a constant, then in order to simplify calculations, we'll just take $\frac{1}{m} \cdot \alpha$ to be our learning rate, and abuse the notation by just calling it α.

GRADIENT DESCENT ALGORITHM



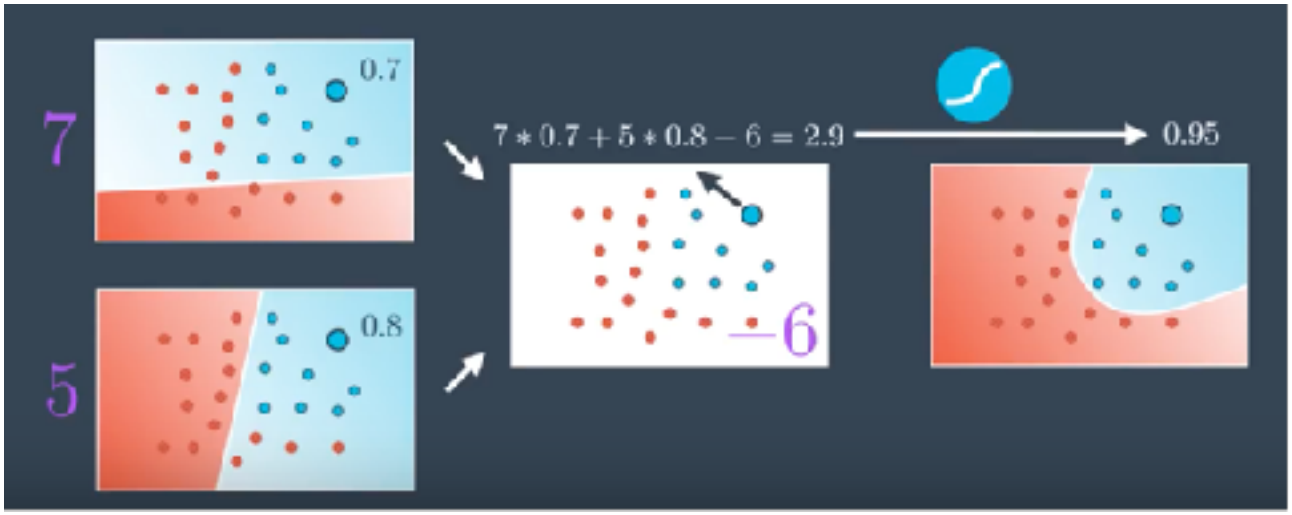
Compare/contrast GD vs. perceptron algo:

- Perceptron only takes y_hat values of zero or one, whereas gradient descent takes any range between zero and one
- Perceptron only adjust line by moving it closer to misclassified points, whereas GD does that and moves line away from correctly classified points

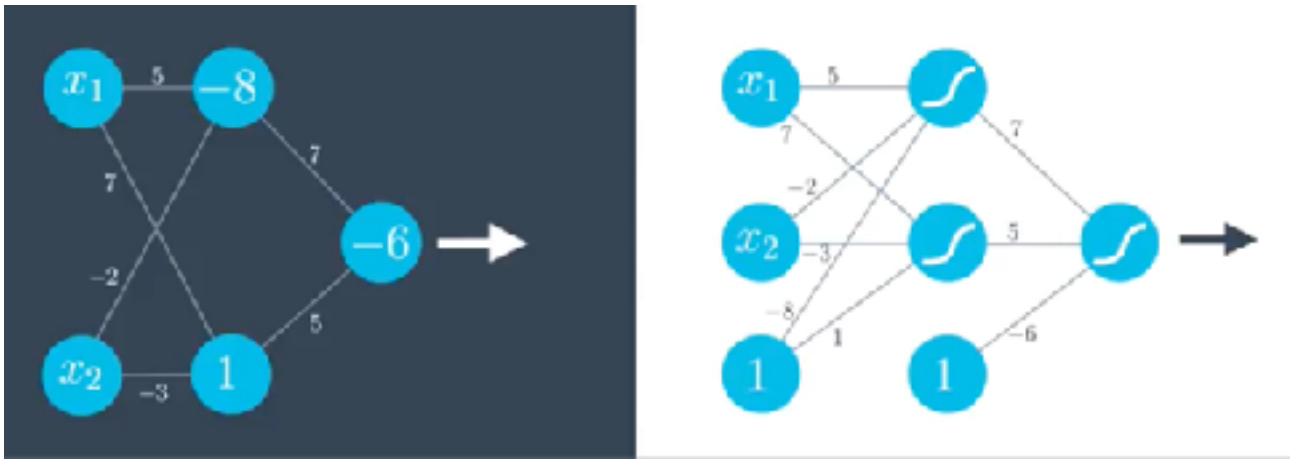
Non-Linear Models

NEURAL NETWORK ARCHITECTURE

NN's combine linear models (which determine the probability of a point being 'correct'), combine the models by by taking the weighted sum of the linear probabilities (and a bias), and finally apply the sigmoid function to return the combination back to a probability between 0 and 1 again.



Expressing above as a NN: left uses notation with bias weight inside the node; right uses notation with the bias weight outside (the constant ‘1’) and the ‘-8’ and ‘1’ bias are shown as edges coming from the bias node



DEEP NEURAL NETWORKS
Recapping above: a simple NN has:
- An *input layer* consisting of the n features and the bias (so n-dimensional space)
- A *hidden layer* consisting of each nodes containing the linear models created by the input layer
- An *output layer* which contains the non-linear model as output

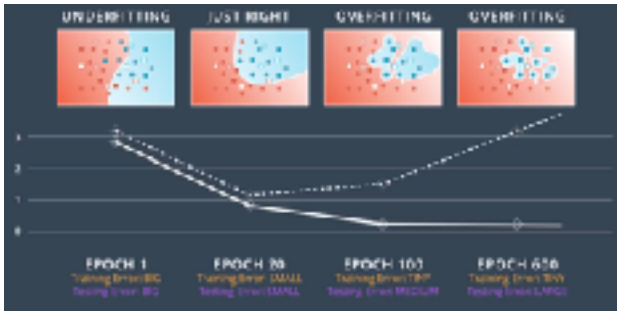
A deep neural network (DNN) has more than one hidden layer. Successive hidden layers after the first are analogous to an output layer (ie: now contain non-linear models in the nodes), but are subsequently treated as input layers for the next hidden layer. The final output layer is then a highly non-linear model.

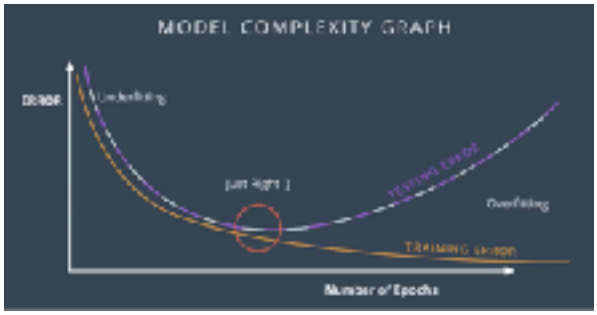
Backpropagation

In a nutshell, backpropagation will consist of:
- Doing a feedforward operation.
- Comparing the output of the model with the desired output.
- Calculating the error.
- Running the feedforward operation backwards (backpropagation) to spread the error to each of the weights. (using the chain rule)
- Use this to update the weights, and get a better model.
- Continue this until we have a model that is good.

Lesson 9: Training Neural Networks

EARLY STOPPING





REGULARIZATION

DROPOUT

LOCAL MINIMA

RANDOM RESTART

VANISHING GRADIENT

OTHER ACTIVATION FUNCTIONS

BATCH VS STOCHASTIC GRADIENT DESCENT

LEARNING RATE DECAY

MOMENTUM

Lesson 10: Deep Learning with PyTorch

Jupyter notebook part 3: MNIST has pretty good end-to-end workflow:
/Users/martinfoot/Documents/JUPYTER/AI4T/M6/L10 - PyTorch/Part 3 - Training Neural Networks (Solution).ipynb

Now we know how to use all the individual parts so it's time to see how they work together. Let's consider just one learning step before looping through all the data. The general process with PyTorch:

**** see fc_model.py file for more complete/proper implementation ****
/Users/martinfoot/Documents/JUPYTER/AI4T/M6/L10 - PyTorch/fc_model.py

1. Set up data:

```
import torch
from torch import nn
import torch.nn.functional as F
from torchvision import datasets, transforms

# Define a transform to normalize the data
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
0.5)),
                               ])
# Download and load the training data
trainset = datasets.MNIST('~/.pytorch/MNIST_data/', download=True, train=True,
transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
```

2. Make a forward pass through the network

```
model = nn.Sequential(nn.Linear(784, 128),
                      nn.ReLU(),
                      nn.Linear(128, 64),
                      nn.ReLU(),
                      nn.Linear(64, 10),
                      nn.LogSoftmax(dim=1))

# TODO: Define the loss
criterion = nn.CrossEntropyLoss() # he also uses nn.NLLLoss()
```

*** Softmax in combination with NLLLoss = CrossEntropy loss*

```
### Run this to check your work
# Get our data
images, labels = next(iter(trainloader))
# Flatten images
images = images.view(images.shape[0], -1)

# Forward pass, get our logits
log_probs = model(images)
```

3. Use the network output to calculate the loss

```
# Calculate the loss with the log_probs and the labels
loss = criterion(log_probs, labels)

print(loss)
```

4. Perform a backward pass through the network with `loss.backward()` to calculate the gradients

```
# model[0].weight.grad refers to first layer nn.Linear(784, 128) of model
print('Before backward pass: \n', model[0].weight.grad)

loss.backward()  # refers to criterion assigned in feed fwd step above
```

```
print('After backward pass: \n', model[0].weight.grad)
```

5. Take a step with the optimizer to update the weights

```
from torch import optim

# Optimizers require the parameters to optimize and a learning rate
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Take an update step and few the new weights
optimizer.step()
print('Updated weights - ', model[0].weight)
```

** above is a single backprop pass; this needs to be put in a loop to run a set number of epochs…

** optimizer accumulates the updated gradients, which will mess with subsequent passes forward and back, so when looping through each training pass, gradients need to be cleared:

```
optimizer.zero_grad()
```

Here is a full workflow for **training**:

```
model = nn.Sequential(nn.Linear(784, 128),
                      nn.ReLU(),
                      nn.Linear(128, 64),
                      nn.ReLU(),
                      nn.Linear(64, 10),
                      nn.LogSoftmax(dim=1))

criterion = nn.NLLLoss()  # or nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.003)
epochs = 5

for e in range(epochs):
    running_loss = 0

    for images, labels in trainloader:
        # Flatten MNIST images into a 784 long vector
        images = images.view(images.shape[0], -1)

        # TODO: Setup training pass
        optimizer.zero_grad()  # clear gradients
        log_probs = model.forward(images)  # output of forward feed
        loss = criterion(log_probs, labels)  # calculate the loss/error
        loss.backward()  # perform backprop
        optimizer.step()  # update weights
        running_loss += loss.item()  # here just to track of progress
    else:     # this is where testing section would go…
        print(f"Training loss: {running_loss/len(trainloader)}")
```

And here is the same but using a Class: with dropout…

```
from torch import nn, optim
import torch.nn.functional as F

class Classifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 128)
```

```
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, 10

      # Dropout module with 0.2 drop probability
      self.dropout = nn.Dropout(p=0.2)

    def forward(self, x):
        # make sure input tensor is flattened
        x = x.view(x.shape[0], -1)

        # Now with dropout
        x = self.dropout(F.relu(self.fc1(x)))
        x = self.dropout(F.relu(self.fc2(x)))
        x = self.dropout(F.relu(self.fc3(x)))

       # Output layer, so no dropout here and log_softmax replaces relu
        # dim=1 so for each row of data evaluated, output class probs
        x = F.log_softmax(self.fc4(x), dim=1)

        return x
```

…. and then running the Classifier model:

Important part when doing testing is that you want all data to be considered/included in the passes, so passes through the test set should turn off dropout. This is done with `model.eval()`, which sets the model to evaluation mode and ignores dropout instruction (ie: sets dropout probability to 0). To turn dropout back on, use `model.train()`. So, generally, the validation loop for testing will (i) turn off gradients using with `torch.no_grad():` (ii) use evaluation mode `model.eval()` (iii) then the validation pass as `for images, labels in test loader:` . Then outside/after the for loop, reset back to train mode with `model.train()`.

```
print(f'Epoch: {e+1, epochs}',
    f'\n  Train Loss: {running_loss/len(trainloader):.3f}',
    f'  Test Loss: {test_loss/len(testloader):.3f}',
    f'  Test Accy: {accuracy/len(testloader):.3f}')
```

Data Augmentation

Cropping, flipping, shearing, etc

Transfer Learning

Keep the feature parameters from the pre-trained model (ie: freeze them) so we don't backprop over them:

```
for param in model.parameters():
    param.requires_grad = False
```

Then need to replace the classifier from the pre-trained model with our own simple feed-forward classifier trained on our data:

```
# pass in an ordered dict to name each layer:
From collections import OrderedDict
classifier = nn.Sequential(OrderedDict([
```

```
                            ('fc1', nn.Linear(1024, 500)), # input 1024 determined by
last pre-trained layer; 500 he just randomly chose as next layer size…
                            ('relu', nn.ReLU()),
                            ('fc2', nn.Linear(500, 2)), # 2 is for dog/cat output
classes
                            ('output', nn.LogSoftmax(dim=1))
                            ]))

model.classifier = classifier      # attaches it to our model
```

USING GPU TO RUN MODEL
To move model, image tensor to GPU:

```
model.cuda()  # moves a model to the GPU
images.cuda()  # moves a tensor of images to the GPU
```

To move back to local computer:

```
model.cpu(), images.cpu()
```

Tips and Tricks

Watch those shapes
In general, you'll want to check that the tensors going through your model and other code are the correct shapes. Make use of the `.shape` method during debugging and development.

A few things to check if your network isn't training appropriately
Make sure you're clearing the gradients in the training loop with `optimizer.zero_grad()`. If you're doing a validation loop, be sure to set the network to evaluation mode with `model.eval()`, then back to training mode with `model.train()`.

CUDA errors
Sometimes you'll see this error:

```
RuntimeError: Expected object of type torch.FloatTensor but found type
torch.cuda.FloatTensor for argument #1 ‘mat1’
```

You'll notice the second type is `torch.cuda.FloatTensor`, this means it's a tensor that has been moved to the GPU. It's expecting a tensor with type `torch.FloatTensor`, no `.cuda` there, which means the tensor should be on the CPU. PyTorch can only perform operations on tensors that are on the same device, so either both CPU or both GPU. If you're trying to run your network on the GPU, check to make sure you've moved the model and all necessary tensors to the GPU with `.to(device)` where `device` is either `"cuda"` or `"cpu"`.

Lesson 11: Recurrent Neural Networks

LSTM: Long Short-Term Memory

The example they use is a TV show about nature and science. There have been lot of forest animals in the show. Recently there have been images of squirrels and trees. And currently there is an image of a dog or a wolf on the screen.

Long-term memory: show subject of nature and science and forest animals
Short-term memory: recent images of squirrels and trees
Event: appearance of unknown image of dog or a wolf

In the context of only short-term memory, a dog prediction would be more likely, but with context of LT memory, we would know the show is about forest animals so a wolf would be more likely prediction.

The next step in the model would incorporate the new short-term context and the event to update the long-term memory. So it might update as a show about nature (dropping science) and lots of forest animals and trees. And it would also update the short-term memory to forget about trees but keep squirrels.

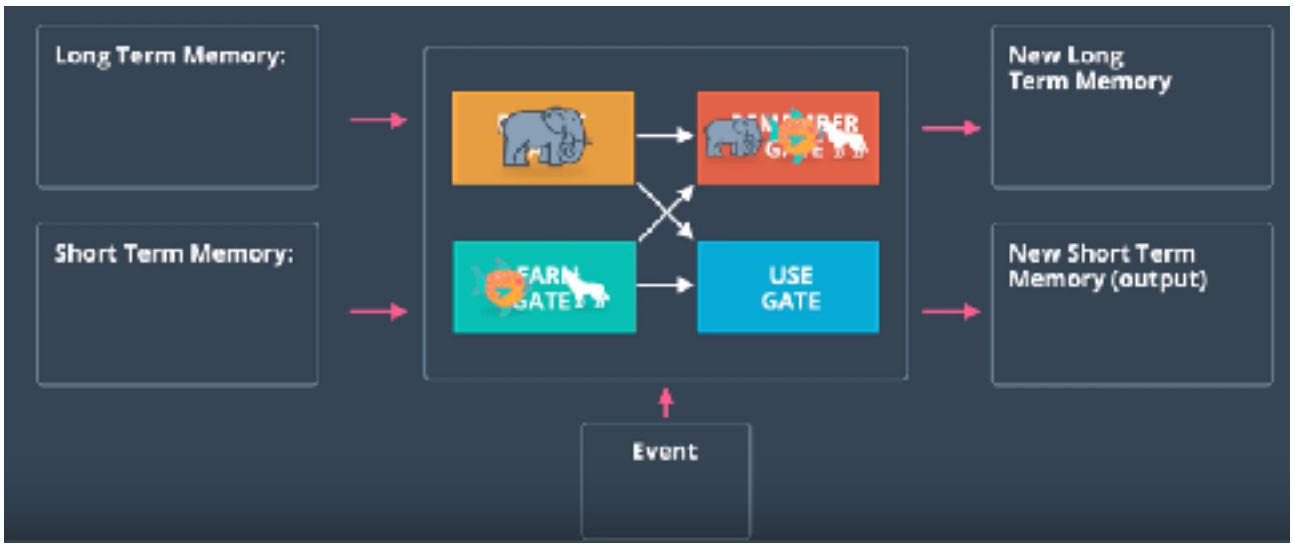
The LSTM architecture has multiple gates at each node: a forget gate, learn gate, remember gate, and use gate.

Forget gate: take previous LTM as input, decides what to keep, puts that in the remember gate.

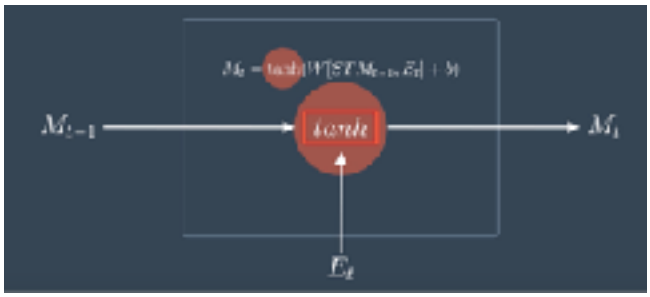
Learn gate: STM and event as input, keeps new info learned and removes unnecessary info.

Remember gate: uses knew information learned in the learn gate (from STM and event) and combines with updated info from the forget gate to decide what to keep as new LTM.

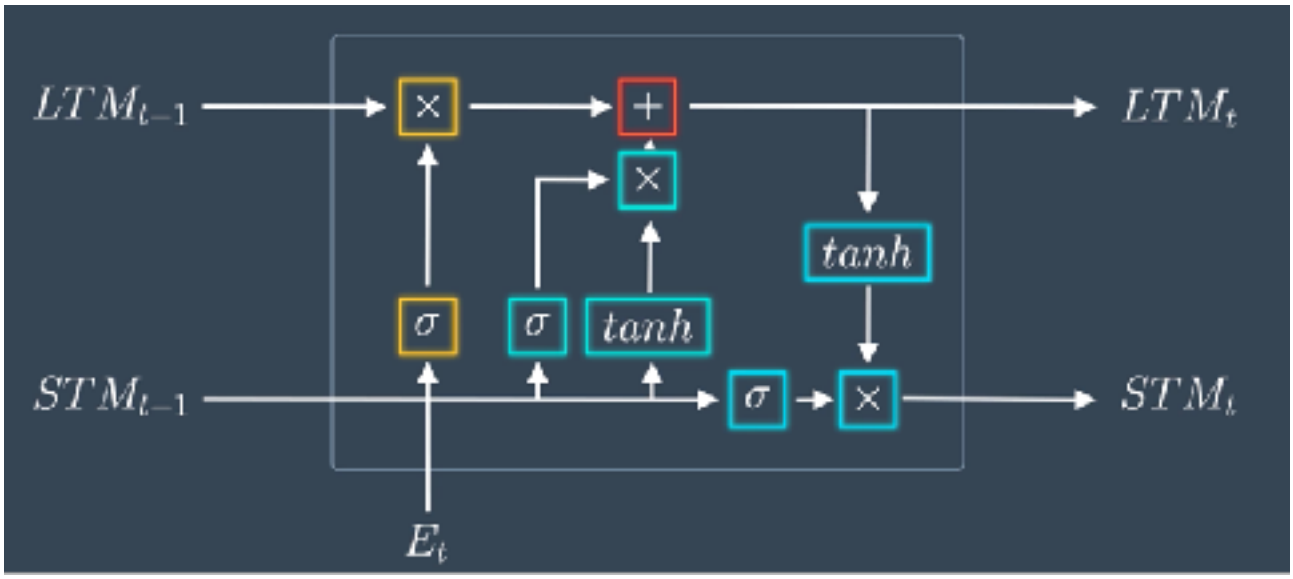
Use gate: takes info from forget gate and learn gate as input to decide to keep for new STM.



SIMPLE RNN ARCHITECTURE
Take previous memory (M[t-1]) and the event (E[t]) as input, multiplies by a weight (W), adds a bias (b), then applies the tanh activation function, which gives us the output M[t]. This is both the prediction of that node and the memory carried to the next RNN node:



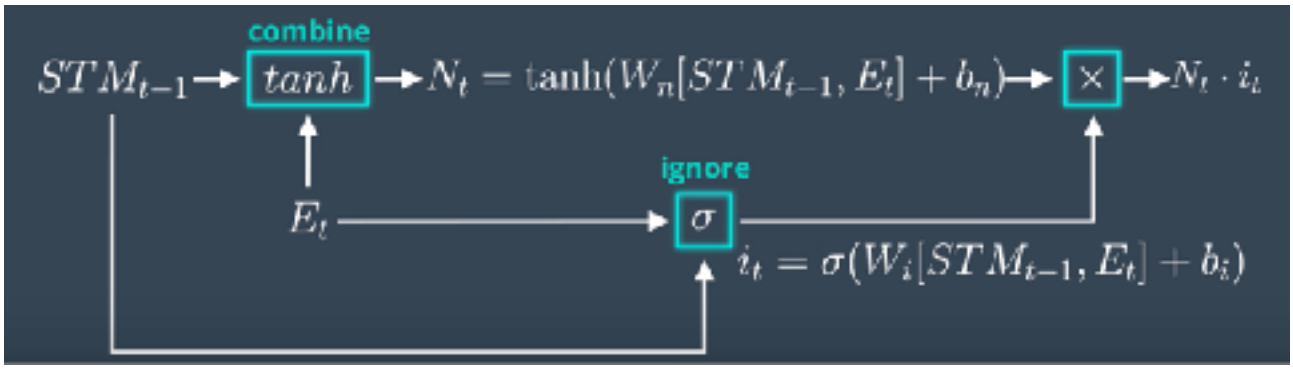
LTSM ARCHITECTURE
Is very similar to RNN except with more gates inside and two outputs since it keeps track of both the LTM and STM:



LEARN GATE
Takes its inputs and decides what to keep and what to forget.

Steps: combines prev STM with current event E[t] in a linear function with weight and bias and passed through the tanh activation function, which outputs 'new information' N[t]. N[t] is then multiplied by an ignore factor i[t]. The ignore factor i[t] is actually a vector, but it multiplies element-wise.

The ignore factor i[t] is itself calculated using STM and and the event E[t] as inputs to a linear gate with sigmoid as the activation function to give an output between 0 and 1.



The output of the *Learn Gate* is $N_t i_t$ where:

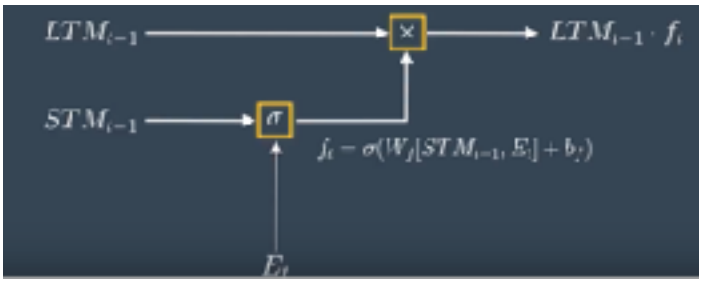
$$N_t = \tanh(W_n[STM_{t-1}, E_t] + b_n)$$

$$i_t = \sigma(W_i[STM_{t-1}, E_t] + b_i)$$

Equation 1

FORGET GATE
Takes LTM and decides what to forget and what to keep for continuing LTM.

Steps: prev LTM input is multiplied by a forget factor f[t]. The forget factor is calculated using prev STM and the current event inputs in a neural linear node with weights, bias and the sigmoid function.



The output of the *Forget Gate* is $LTM_{t-1} f_t$ where:

$$f_t = \sigma(W_f[STM_{t-1}, E_t] + b_f)$$

Equation 2

REMEMBER GATE
Simply adds together the outputs from the forget and learn gates — that's it.

USE GATE
Takes output of forget gate (ie, what to remember) and learn gate (what new and recent info kept) as inputs, same as remember gate, but applies a tanh node to the output of the forget gate, and a sigmoid node to the output of the learn gate, then multiplies these two together to get the outputs of a new STM and the prediction (which are the same thing).

The output of the *Use Gate* is $U_t V_t$ where:

$$U_t = \tanh(W_u LTM_{t-1} f_t + b_u)$$

$$V_t = \sigma(W_v[STM_{t-1}, E_t] + b_v)$$

Equation 4

RNN code
The input size should have dimensions: (batch_size, sequence length, input number of features)

The output size should have dimensions: (batch_size, sequence length)

The hidden state should have dimensions: (num_layers, batch_size, hidden_dim)

Getting Batches Right

RNNs look at sequences of data, like stock prices, text, audio waveforms etc. To make training more efficient, we can split a sequence into multiple shorter sequences to take advantage of matrix operations. In effect, this means the training can be done on multiple sequences in parallel.

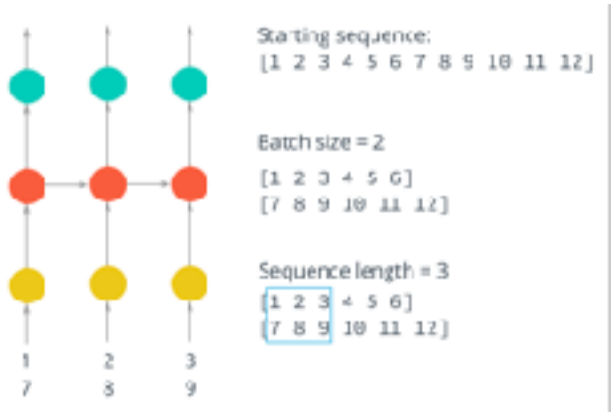
eg sequence: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

This could be passed directly to the RNN, or better, could be split into say 2 batches:

batch1: [1, 2, 3, 4, 5, 6] and
batch2: [7, 8, 9, 10, 11, 12]

We can also specify the sequence length to look at, for example seq_length=3 would look at the first 3 elements of each batch sequence, ie: 1, 2, 3 of batch 1, and 7, 8, 9 of batch 2. And so on.

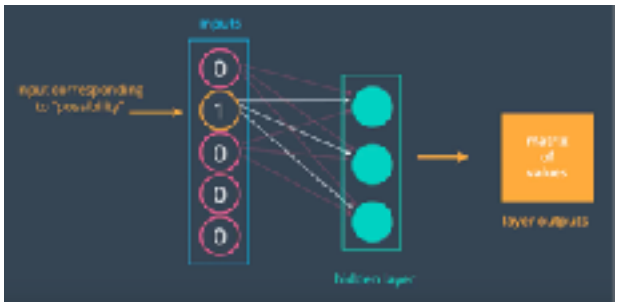
We can retain the hidden state of one batch and use it at the start of the next batch, thus transferring the sequence information across the batches.



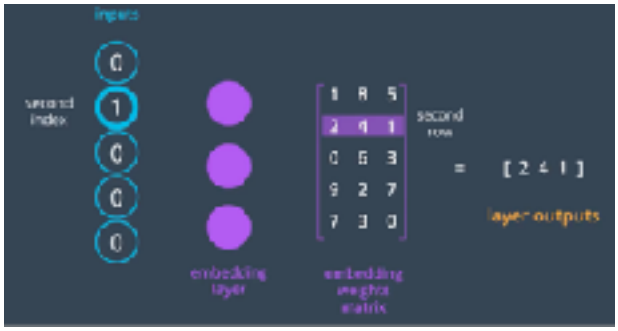
Lesson 12: Embeddings and Word2Vec

Word Embeddings

One-hot encode a vocabulary. The length of the resulting vector of zeros and a one for each word is equal to the number of words in the vocabulary. This word vector is the input layer that is then passed to a hidden layer network. The output of this is calculated by multiplying by the weights and bias. The result is another huge matrix of values, also mostly zeros (ie: very sparse). This essentially means thousands of calculations have been spent only to generate more zeros which are of no use. Embeddings solve this inefficiency of one-hot encoding.

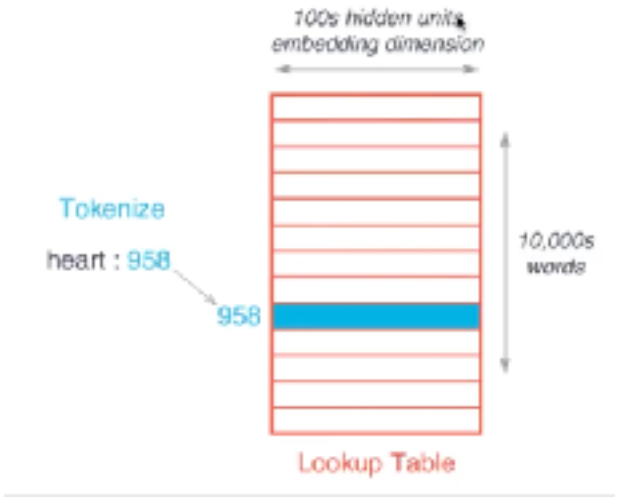


Embeddings basically provide a shortcut for doing the multiplications above by adding in an additional 'embedding' layer. To 'learn' word embeddings, we use a fully connected linear layer (the embedding layer) which has its own matrix of embedding weights learned during training of the embedding layer. These weights serve as a lookup table for the weights of a particular word, since when multiplied by a word's one-hot vector, all that is returned is the weight matrix row corresponding to the '1' in the one-hot vector. That is, the index position of the '1' in the one-hot vector is all that is required to lookup the weights for that particular word from the embedding weights matrix.



This serves the purpose of encoding each word in a vocabulary with a unique integer ID, instead of a massively long one-hot vector. This is the essence of word embedding. For example, if the word heart was given a number 958 from the embedding process, we can simply lookup the 958th row in the embedding weights matrix to find its weights for the layer outputs.

This process is called an embedding lookup. The number of hidden units (??) is the embedding dimension.



The way I understand it is: each row of the embedded weights matrix is the vector describing the position of a word in vector space.

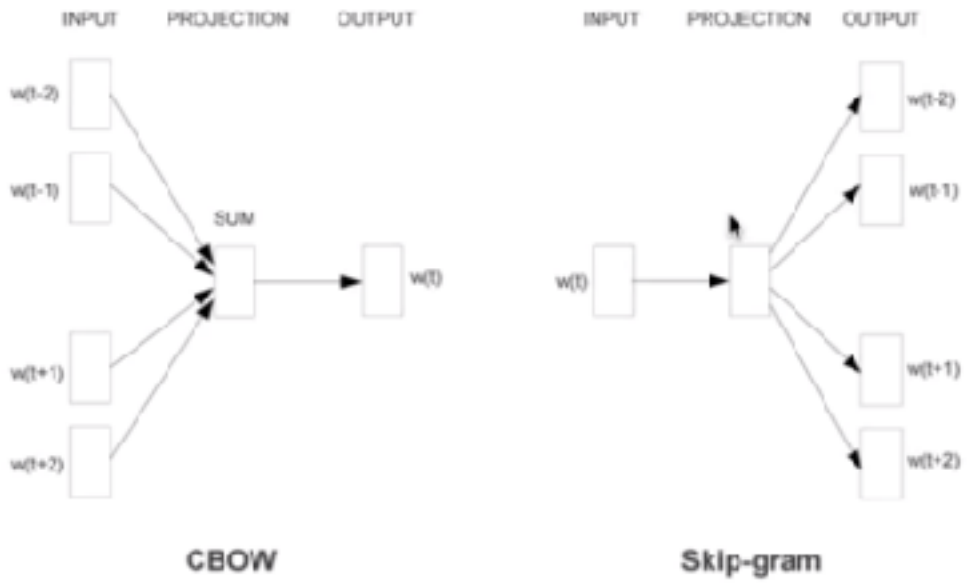
The dimensions of the vector space is defined by the embedding dimension, ie: the number of columns in the embedded weights matrix (or the number of elements in a word's row vector). This dimension is the result of the output layer dimensions chosen in the learning of the embedding layer network.

The resulting vector space and positional information for each word allows relationships between words to be learned/understood.

Implementing Word2Vec

There are 2 architectures for implementing Word2Vec:

- CBOW (Continuous Bag-Of-Words) and
- Skip-gram



CBOW takes as input the context (ie: surrounding words) and tries to output the missing word.

Skip-gram does the opposite: it inputs a word and tries to guess the context or surrounding words. Skip-gram has been shown to work a bit better than CBOW.

NOTEBOOKS
Negative_Sampling (for the SkipGramNeg model lessons)

Lesson 13: Sentiment Analysis with an RNN

NOTEBOOKS
These are to be found in the Lesson 12 master directory for all RNN / DL notebooks:
sentiment_rnn

Tensor Dataset and Batching Data

OMISSION: SHUFFLING DATA
Make sure to shuffle your data, so that your model doesn't learn anything about the ordering of the data, and instead can focus on the *content*. We can do this with a DataLoader by setting `shuffle=True`. You'll find this updated code in the exercise and solution notebooks.

```
# make sure to SHUFFLE your data
train_loader = DataLoader(train_data, shuffle=True, batch_size=batch_size)
valid_loader = DataLoader(valid_data, shuffle=True, batch_size=batch_size)
test_loader = DataLoader(test_data, shuffle=True, batch_size=batch_size)
```

TENSORDATASET
Take a look at the source code for the TensorDataset class, you can see that it's "purpose" is to provide an easy way to create a dataset out of standard data structures.

Complete Sentiment RNN

COMPLETE RNN CLASS
I hope you tried out defining this model on your own and got it to work! Below, is how I completed this model.

I know I want an embedding layer, a recurrent layer, and a final, linear layer with a sigmoid applied; I defined all of those in the `__init__` function, according to passed in parameters.

```
def __init__(self, vocab_size, output_size, embedding_dim, hidden_dim, n_layers,
drop_prob=0.5):
        """
        Initialize the model by setting up the layers.
        """
        super(SentimentRNN, self).__init__()

        self.output_size = output_size
        self.n_layers = n_layers
        self.hidden_dim = hidden_dim

        # embedding and LSTM layers
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers,
                            dropout=drop_prob, batch_first=True)

        # dropout layer
        self.dropout = nn.Dropout(0.3)

        # linear and sigmoid layers
        self.fc = nn.Linear(hidden_dim, output_size)
        self.sig = nn.Sigmoid()
```

`__init__` explanation

First I have an **embedding layer**, which should take in the size of our vocabulary (our number of integer tokens) and produce an embedding of `embedding_dim` size. So, as this model trains, this is going to create and embedding lookup table that has as many rows as we have word integers, and as many columns as the embedding dimension.

Then, I have an **LSTM layer**, which takes in inputs of `embedding_dim` size. So, it's accepting embeddings as inputs, and producing an output and hidden state of a hidden size. I am also specifying a number of layers, and a dropout value, and finally, I'm setting `batch_first` to True because we are using DataLoaders to batch our data like that!

Then, the LSTM outputs are passed to a dropout layer and then a fully-connected, linear layer that will produce `output_size` number of outputs. And finally, I've defined a sigmoid layer to convert the output to a value between 0-1.

FEEDFORWARD BEHAVIOR

Moving on to the `forward` function, which takes in an input `x` and a `hidden` state, I am going to pass an input through these layers in sequence.

```
def forward(self, x, hidden):
        """
        Perform a forward pass of our model on some input and hidden state.
        """
        batch_size = x.size(0)

        # embeddings and lstm_out
        embeds = self.embedding(x)
        lstm_out, hidden = self.lstm(embeds, hidden)

        # stack up lstm outputs
        lstm_out = lstm_out.contiguous().view(-1, self.hidden_dim)

        # dropout and fully-connected layer
        out = self.dropout(lstm_out)
        out = self.fc(out)

        # sigmoid function
        sig_out = self.sig(out)

        # reshape to be batch_size first
        sig_out = sig_out.view(batch_size, -1)
        sig_out = sig_out[:, -1] # get last batch of labels

        # return last sigmoid output and hidden state
        return sig_out, hidden
```

`forward` explanation

So, first, I'm getting the `batch_size` of my input x, which I'll use for shaping my data. Then, I'm passing x through the embedding layer first, to get my embeddings as output.

These embeddings are passed to my lstm layer, alongside a hidden state, and this returns an `lstm_output` and a new `hidden` state! Then I'm going to stack up the outputs of my LSTM to pass to my last linear layer.

Then I keep going, passing the reshaped `lstm_output` to a dropout layer and my linear layer, which should return a specified number of outputs that I will pass to my sigmoid activation function.

Now, I want to make sure that I'm returning *only* the **last** of these sigmoid outputs for a batch of input data, so, I'm going to shape these outputs into a shape that is `batch_size` first. Then I'm getting the last bacth by called `sig_out[:, -1], and that's going to give me the batch of last labels that I want!

Finally, I am returning that output and the hidden state produced by the LSTM layer.

`init_hidden`

That completes my forward function and then I have one more: `init_hidden` and this is just the same as you've seen before. The hidden and cell states of an LSTM are a tuple of values and each of these is size (n_layers by batch_size, by hidden_dim). I'm initializing these hidden weights to all zeros, and moving to a gpu if available.

```
def init_hidden(self, batch_size):
        ''' Initializes hidden state '''
        # Create two new tensors with sizes n_layers x batch_size x hidden_dim,
        # initialized to zero, for hidden state and cell state of LSTM
        weight = next(self.parameters()).data

        if (train_on_gpu):
            hidden = (weight.new(self.n_layers, batch_size,
self.hidden_dim).zero_().cuda(),
                  weight.new(self.n_layers, batch_size,
self.hidden_dim).zero_().cuda())
        else:
            hidden = (weight.new(self.n_layers, batch_size,
self.hidden_dim).zero_(),
                      weight.new(self.n_layers, batch_size,
self.hidden_dim).zero_())

        return hidden
```

After this, I'm ready to instantiate and train this model, you should see if you can decide on good hyperparameters of your own, and then check out the solution code, next!

HYPERPARAMETERS

After defining my model, next I should instantiate it with some hyperparameters.

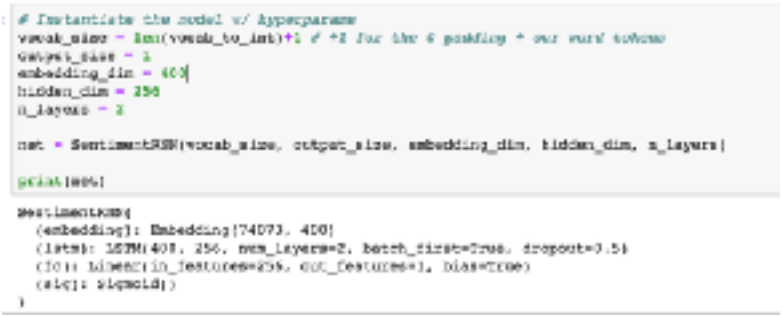
```
# Instantiate the model w/ hyperparams
vocab_size = len(vocab_to_int)+1 # +1 for the 0 padding + our word tokens
output_size = 1
embedding_dim = 400
hidden_dim = 256
n_layers = 2

net = SentimentRNN(vocab_size, output_size, embedding_dim, hidden_dim, n_layers)

print(net)
```

This should look familiar, but the main thing to note here is our `vocab_size`.
This is actually the length of our `vocab_to_int` dictionary (all our unique words) **plus one** to account for the `0`-token that we added, when we padded our input features. So, if you do data pre-processing, you may end up with one or two extra, special tokens that you'll need to account for, in this parameter!

Then, I want my `output_size` to be 1; this will be a sigmoid value between 0 and 1, indicating whether a review is positive or negative.

Then I have my embedding and hidden dimension. The embedding dimension is just a smaller representation of my vocabulary of 70k words and I think any value between like 200 and 500 or so would work, here. I've chosen 400. Similarly, for our hidden dimension, I think 256 hidden features should be enough to distinguish between positive and negative reviews.

I'm also choosing to make a 2 layer LSTM. Finally, I'm instantiating my model and printing it out to make sure everything looks good.



TRAINING AND OPTIMIZATION
The training code, should look pretty familiar. One new detail is that, we'll be using a new kind of cross entropy loss that is designed to work with a single Sigmoid output.

BCELoss, or **Binary Cross Entropy Loss**, applies cross entropy loss to a single value between 0 and 1.

We'll define an Adam optimizer, as usual.

```
# loss and optimization functions
lr=0.001

criterion = nn.BCELoss()
optimizer = torch.optim.Adam(net.parameters(), lr=lr)
```

OUTPUT, TARGET FORMAT
You should also notice that, in the training loop, we are making sure that our outputs are squeezed so that they do not have an empty dimension `output.squeeze()` and the labels are float tensors, `labels.float()`. Then we perform backpropagation as usual.

Train and eval mode
Below, you can also see that we switch between train and evaluation mode when the model is training versus when it is being evaluated on validation data!

TRAINING LOOP
I'm actually only going to do four epochs of training because that's about when I noticed the validation loss stop decreasing.

- You can see that I am initializing my hidden state before entering the batch loop then have my usual detachment from history for the hidden state and backpropagation steps.
- I'm getting my input and label data from my train_dataloader. Then applying my model to the inputs and comparing the outputs and the true labels.
- I also have some code that checks performance on my validation set, which, if you want, may be a great thing to use to decide when to stop training or which best model to save!

```
# training params
epochs = 4 # 3-4 is approx where I noticed the validation loss stop decreasing
counter = 0
print_every = 100
clip=5 # gradient clipping

# move model to GPU, if available
if(train_on_gpu):
    net.cuda()
```

```
net.train()
# train for some number of epochs
for e in range(epochs):
    # initialize hidden state
    h = net.init_hidden(batch_size)

    # batch loop
    for inputs, labels in train_loader:
        counter += 1
        if(train_on_gpu):
            inputs, labels = inputs.cuda(), labels.cuda()

        # Creating new variables for the hidden state, otherwise
        # we'd backprop through the entire training history
        h = tuple([each.data for each in h])

        # zero accumulated gradients
        net.zero_grad()

        # get the output from the model
        output, h = net(inputs, h)

        # calculate the loss and perform backprop
        loss = criterion(output.squeeze(), labels.float())
        loss.backward()
        # `clip_grad_norm` helps prevent the exploding gradient problem in RNNs /
LSTMs.
        nn.utils.clip_grad_norm_(net.parameters(), clip)
        optimizer.step()

        # loss stats
        if counter % print_every == 0:
            # Get validation loss
            val_h = net.init_hidden(batch_size)
            val_losses = []
            net.eval()
            for inputs, labels in valid_loader:

                # Creating new variables for the hidden state, otherwise
                # we'd backprop through the entire training history
                val_h = tuple([each.data for each in val_h])

                if(train_on_gpu):
                    inputs, labels = inputs.cuda(), labels.cuda()

                output, val_h = net(inputs, val_h)
                val_loss = criterion(output.squeeze(), labels.float())

                val_losses.append(val_loss.item())

            net.train()
            print("Epoch: {}/{}...".format(e+1, epochs),
                  "Step: {}...".format(counter),
                  "Loss: {:.6f}...".format(loss.item()),
                  "Val Loss: {:.6f}".format(np.mean(val_losses)))
```

Make sure to take a look at how training **and** validation loss decrease during training! Then, once you're satisfied with your trained model, you can test it out in a couple ways to see how it behaves on new data!

Project 6: Sentiment Analysis with Neural Networks

Reviewer Notes | Comments

LogSoftmax(dim=1)
My understanding is that the dim should =1 which represents each row we need to make probabilities on which class to predict. @RicardoD is this correct?

*Hi **@MartinF**, that's correct, you want to apply the log softmax in the `dim=1` because your outputs have dimensions `(batch_size, n_classes)`, so dim=0 is `batch_size` and dim=1 is `n_classes` and you want to calculate the log softmax in the classes.*

Remove most common and rare words by defining the following variables: `freqs`, `low_cotoff`, `high_cutoff`, `K_most_common`.

***Tips:** There are a few things you can do to improve this implementation, I'll list some of them:*

- *Instead of `bow.values()` you could have used `bow.items()` to get both the value and the key and use that to create the frequencies using dictionary comprehensions. e.g.*
  ```
  {word: count/len(tokenized) for word, count in bow.items()}
  ```
- *Your `K_most_common` variable is a list, and when using the conditional `word not in K_most_common` it takes longer to compute than using a `dict` or `set` that have a constant lookup time in python, so instead of looping through the k_most, wrap it in a `dict` object like this `dict(k_most)`.*

Defining the variables : 'vocab', 'id2vocab' and 'filtered' correctly.
***Note:** Remember that the `filtered_words` come from the `bow` variable, this means they are unique. Because of this, There's no need to count them nor sort them in order to create the vocab.*

Correctly split the data into `train_features`, `valid_features`, `train_labels`, and `valid_labels`.
you could also use a larger split like `0.9` so you can have more data in your training.

Train your model with dropout and clip the gradient. Print out the training progress with the loss and accuracy.
***Note:** Your test loss and accuracy are jumping to much because you are not using the average loss and average accuracy from the validation batches, if you'd like to see a more stable result, sum and average all the losses and accuracies that you get per step.*

The `predict` function correctly prints out the prediction vector from the trained model.
***Tip:** To convert python's list to torch tensors use `torch.tensor` instead of converting it into a numpy array.*

Student Hub | Knowledge Discussions

To download twits data:
run the following in your notebook:
Copy the data to the workspake dir with: !cp -r ../../data/ ../workspace/
Zip the data using: !zip -r data.zip data/

— I used json.dump() once the data was already loaded into notebook

```
with open('file_name.json', 'w') as f:
    json.dump(twits, f)
```

Another poster:
Zips all the files in the workspace so you can download them all at the same time:

```
zip -r workspace.zip ./*
```

Zips just data:

```
zip -r data.zip ../../data/*
```

Basically for **"Updating Vocabulary by Removing Filtered Words"**, you have to create a `vocab` that maps the words you filtered (filtered_words) to an id (i.e. {'this':1, 'is':2, 'the':3, 'vocab':4}) and then you want to use that `vocab` to filter the words that are inside your `tokenized` list.
Tip: You want to loop through the tokens in `tokenized` and only keep the words that are in the `vocab`

I didn't read all the text in the notebook and I might have missed something, but a couple of feedback points:

1) People jumping directly into this project aren't immediately aware that vocab indices should start from 1 and leave 0 for the pad token
2) People who have done the sentiment RNN classroom notebook might be thrown off by the fact that the project switches to the (seq_length, batch_size,...) input format instead of the (batch_size, seq_length,...) format used in sentiment RNN.

NN SIZE MISMATCH
1. problem with batch sizing, sequence length being messed up:

```
batch_size=nn_input.size(1)
    embed_out=self.embedding(nn_input)
    r_out,hidden_state=self.lstm(embed_out,hidden_state)

    out = self.fc(r_out)

    softmax_out = self.softmax(out)

    return softmax_out, hidden_state
```

remember here you do not have batch_first=True*, so accordingly shape your input. Moreover, since now input is* seq_length x batch *you just need to transform* lstm_out = lstm_out[-1, : , :]*.*

*This is just a **pytorch** preference, some libraries vary the input shapes so we decided to use the default shape (seq_len, batch_size) for the LSTM in this project. You can change this with the nn.LSTM parameter batch_first, if you set it to batch_first=True it'll accept the shape (batch_size, seq_len) and if you use batch_first=False it'll be the opposite.*

2. I keep getting a RuntimeError: Expected hidden[0] size (2, 20, 512), got (1, 20, 128) when trying to get the output on the model. For the life of me I cannot figure out what is wrong. Any idea why this may be happening on this line: `output, hidden = model(text_batch, hidden)` ?

sounds like an error with your hidden state. Remember you have to initialize your hidden state with the batch_size that comes from the `text_batch` *because the last batch doesn't always have the same size as the* `batch_size` *you set. To fix this, initialize the hidden state inside your batch loop with size:* `text_batch.size(1)` *.*

I had similar issues when I did the project for the first time, it would be good if you can see how data size changes after each layer. Also remember that in the LSTM layer in the project batch_first=False

text_batch, labels = text_batch.to(device), labels.to(device) <== seems should be correct…

3. When running the below code segment, I receive the following error when my init_hidden function is implemented as in the Sentiment RNN lesson: RuntimeError: Expected hidden[0] size (2, 5, 6), got (2, 4, 6). As the batch_size is simply defined in the code as 4, I'm a bit lost on why I'm getting the error unless I'm missing something. Thanks. Code snippet:

```
model = TextClassifier(len(vocab), 10, 6, 5, dropout=0.1, lstm_layers=2)
model.embedding.weight.data.uniform_(-1, 1) input = torch.randint(0, 1000, (5, 4),
dtype=torch.int64) hidden = model.init_hidden(4)
logps, _ = model.forward(input, hidden) print(logps)
```

you may want to check how you are grabbing the batch_size in forward pass? should be batch_size==nn_input.size(1)

thanks. I did have that line of code. There error occurs in the forward pass at the line:

```
lstm_out, hidden_state = self.lstm(embeds, hidden_state)
```

if I change where it expects a hidden_state size of (2,5,6) as opposed to (2,4,6) which the 4 gets set by the code in the test cell at the line:

```
hidden = model.init_hidden(4).
```

If I simply, change this to 5 instead of 4, it seems to run without the runtime error. Not sure this is correct however.

The error is because the LSTM model for this project takes input as `seq_length x batch_size`*, You need to ensure that your data loaders return input in this form, moreover this also means you do need to do any transformations in forward, excecp [-1, : , :]. In case the error is from other point please share your code, so that we can help better.*

VIEW MODEL OUTPUT

What should the output of the cell titled "View Model" be? The code following it is:

```
model = TextClassifier(len(vocab), 10, 6, 5, dropout=0.1, lstm_layers=2)
model.embedding.weight.data.uniform_(-1, 1)
input = torch.randint(0, 1000, (5, 4), dtype=torch.int64)
hidden = model.init_hidden(4)

logps, _ = model.forward(input, hidden)
print(logps)
```

the output shape should be the output of your model (forward pass) for the random `input`
Since output are probabilities they should lie between 0-1 if you are using softmax

yes the shape of the output tensor should be (batch_size, n_classes)

Yeah you should use the LogSoftmax activation function for your model outputs

TRAINING LOSS CALLS / ACCURACY

when doing training loss calculations, I'm doing

```
output_valid, hidden_valid = model(text_valid, hidden_valid)
```
but this causes a traceback with this error:
```
 RuntimeError: Expected hidden[0] size (2, 10, 512), got (2, 64, 512)
```
Any thoughts/ideas?

I guess you are using softmax with NLLLoss, which may be causing this problem. Try using logsoftmax for your output. The logsoftmax output are the values that the NLLloss uses as inputs to compute the loss. However, when making predictions you'd want to use the torch.exp function on your outputs in order to see the right probabilities

Your loss should be around 0.8 or lower in order to get good results. But this won't tell you much about the model, that's why you have to print out the accuracy as well.

Here is an example of how to compute the accuracy in pytorch:
```
accuracy = 0
for features, labels in valid_batches:
    ....
    logps, hidden = model(text_batch, hidden)

    topval, topclass = torch.exp(logps).topk(1)
    accuracy += torch.sum(topclass.squeeze() == labels)

print('Accuracy:', accuracy/len(valid_batches))
```

NLLloss works with the logsoftmax outputs, so you don't need to compute the exp on the outputs in order to calculate the NLLloss, but for the accuracy you should.

PADDING

Hi, in the predict function one should also pad the data with zeros in order to get an input of size (40, 1) , where 40 is the sequence length and 1 is the batch size ? in this case since I am predicting a single sentence the batch should be 1 correct?

```
-- Left padding
sequence_length = 40
padded_token_tensor = torch.zeros((sequence_length, 0), dtype=torch.int64)
start_idx = max(sequence_length - len(token_tensor), 0)
padded_token_tensor[start_idx:] = token_tensor[:sequence_length]

# -- Adding a batch dimension
padded_token_tensor = padded_token_tensor.unsqueeze(dim=1)
print(padded_token_tensor.size())
```

you don't have to pad your inputs in the predict function, just use the .view(-1, 1) method on your token_tensor to get thetext_input

PREDICT

1.
this is what I am doing in predict,
```
def predict(text, model, vocab):
    """
    Make a prediction on a single sentence.

    Parameters
    ----------
        text : The string to make a prediction on.
        model : The model to use for making the prediction.
        vocab : Dictionary for word to word ids. The key is the word and the value
is the word id.

    Returns
```

```
    -------
        pred : Prediction vector
    """

    # TODO Implement

    tokens = preprocess(text)
    # Filter non-vocab words
    tokens = [word for word in tokens if word in vocab]
    # Convert words to ids
    tokens = [vocab[word] for word in tokens]
    token_tensor=torch.tensor(tokens)
    # Adding a batch dimension
    sequence_length = 30
    padded_token_tensor = torch.zeros((sequence_length), dtype=torch.int64)
    start_idx = max(sequence_length - len(token_tensor), 0)
    padded_token_tensor[start_idx:] = token_tensor[:sequence_length]

    # -- Adding a batch dimension
    padded_token_tensor = padded_token_tensor.unsqueeze(dim=1)
    print(padded_token_tensor.size())
    text_input = padded_token_tensor
    # Get the NN output
    batch_size=padded_token_tensor.size(0)
    hidden = model.init_hidden(batch_size)
    if(train_on_gpu):
        padded_token_tensor=padded_token_tensor.cuda()
    logps, _ = model(padded_token_tensor,hidden)
    # Take the exponent of the NN output to get a range of 0 to 1 for each label.
    pred = logps.exp()

    return pred
```

*You are not required to add padding in your **predict** function*

***Tip:** Use the vocab inside your first loop, like this:*

```
tokens = [vocab[word] for word in tokens if word in vocab]
```

*Also you shouldn't move your inputs nor your model to the **GPU** in the predict function*

I guess part of the confusion comes from this demand?

Adding a batch dimension

```
text_input = none
```

what exactly is expected?

*Remember your inputs should be dim **(seq_length, batch_size)** and your text comes with just the dim **(seq_length,)** so all you have to do is add a batch dimension that'll be (1) because you have only one sample. You can do it by using the **view** method in your **token_tensor** like this:*

```
token_tensor.view(-1, 1)
```

*Also, because you have only one sample your init_hidden should have **batch_size=1***

2.

now I still have a minor problem with predict:

```
    # Filter non-vocab words
    tokens = [vocab[word] for word in tokens if word in vocab]
    token_tensor=torch.tensor(tokens)
    # Adding a batch dimension
    text_input = token_tensor.view(1,-1)
    hidden=model.init_hidden(1)
    logps, _ = model(token_tensor,hidden)
    # Take the exponent of the NN output to get a range of 0 to 1 for each label.
```

```
    pred = logps.exp()
```

1. I don't really understand why we don't need to left pad
2. this logic above gives error "expected dimension [0,-1] found 1"

the inputs of your model are shape (seq_length, batch_size) or (batch_size, seq_length)?
If your inputs are shape (seq_length, batch_size) your reshape should be .view(-1,1).

*And for your left pad question, since this is one batch and the LSTM layer accepts any sequence length, you can just go ahead and pass the tokens without padding. This won't cause you any trouble (Because it's just one sample **batch size = 1**).*

this is what I ended up doing:

```
tokens = preprocess(text)
tokens = [vocab[word] for word in tokens if word in vocab]
token_tensor=torch.tensor(tokens)
text_input = token_tensor.view(-1,1)
hidden=model.init_hidden(text_input.size(1))
hidden = tuple([each.data for each in hidden])
```

It looks good to me, though you don't need the tuple([each.data for each in hidden]) part in the predict function.

you dont need it for prediction. you can just set hidden to model's init hidden method

```
hidden = model.init_hidden( ... )
```

FORWARD FUNC

Hello, I am very confused by the fact that we do not provide batch first input to the model as in the lecture notebook, could you please help me to check what I am doing wrong the forward function of the model

```
Forward function
    batch_size = nn_input.size(1)
    # embeddings and lstm_out
    nn_input = nn_input.long()
    embeds = self.embedding(nn_input)
    lstm_out, hidden_state = self.lstm(embeds, hidden_state)

    # stack up lstm outputs
    lstm_out = lstm_out.contiguous().view(-1, self.lstm_size)
    #print lstm_out.size()
    #sys.exit(-1)

    # dropout and fully-connected layer
    out = self.dropout(lstm_out)
    out = self.fc(out)
    # sigmoid function
    log_softmax_out = self.log_softmax(out)
    print(log_softmax_out.size())

    # last label
    #log_softmax_out = log_softmax_out.view(batch_size, -1)
    p
    log_softmax_out = log_softmax_out[-1, :] # get last labels
    # Return log prob and hidden state
    return log_softmax_out, hidden
```

this project is slightly different from the one in the lectures, as you saw, the input has to come in shape (seg_length, batch_size) instead of (batch_size, seg_length). Also, you don't want to

*reshape the `lstm_out`. Instead what you want is to get the **last output from the sequence**, and if the outputs from the LSTM are shape `(seq_length, batch_size, lstm_size)`, you'd want to use `lstm_out[-1, : , : ]` instead of `lstm_out.contiguous()`.... with this, you don't have to do any other reshaping, just pass it through the fc and return the softmax as it is. Extra tip: The dropout layer should wrap the `fc` instead of the `lstm_out`*

I ended up using LSTM dropout at 0.. I used a droupout layer and played around with 0.1 and 0.5 to see different results before submitting

FILTERED WORDS

well, the only thing I changed recently was *#(is this in training section?)*

```
filtered_words = [word for word in bow if word not in K_most_common and
(random.random() < (1 - p_drop[word]) )]
```

to

```
filtered_words = [word for word in bow if word not in K_most_common and
(random.random() > (1 - p_drop[word]) )]
```

DATA SPLIT MISMATCH

This error:

```
RuntimeError: Expected hidden[0] size (2, 7, 512), got (2, 64, 512)
```

*Is caused because the last batch of your testing it is coming with a different size than the actual batch_size. This happens because the dataset not always splits evenly. In order to fix this error you have to **initialize the hidden state inside the batch loop (training and valid loops) and use labels.shape[0] as the input batch_size**, like this:*

```
val_h = model.init_hidden(labels.shape[0])
```

Thanks for your solution. I am having the exact same problem as Hahnsang but changing `val_h = model.init_hidden(labels.shape[0]) ` is still not working for me. It will run for a few minutes then stop out with the above shape error

Make sure you are initializing hidden in side the for loop. like this

```
with torch.no_grad():
     for inputs, labels in dataloader(valid_features, valid_labels,
        batch_size=batch_size, sequence_length=seq_len, shuffle=True):
                inputs, labels = inputs.to(device), labels.to(device)
                val_h = model.init_hidden(labels.shape[0]
```

if you do it before the for loop your last batch in the test run is smaller than what is expected.

Make sure that the same is true for training model like this on Line 22

```
hidden = model.init_hidden(labels.shape[0])
```

TRAINING SPEED

Project 6 : Training took more than 5 hours in GPU instead of less than an hour ?!? I'm stuck (this post in knowledge has full model code posted…)

Try increasing your batch size from 64 to 512 or 1024 that made a big difference for me. Also make sure you doing this in your validation loop as well

```
for each in hidden:
        each.to(device)
```

MODULE 7: COMBINE SIGNALS 4 ENHANCED ALPHA

Lesson 15: Overview of ML Techniques

Machine Learning can be broken down into 3 main categories:

1. Supervised Learning
2. Unsupervised Learning
3. Reinforcement Learning

Supervised Learning

Is where the training data has the outcomes already labelled, which enable the algorithm to learn how to associate labels with new unseen data.

Supervised learning has 2 types: classification and regression.

Classifications predicts a *category* an example might fit into, like spam email or not, but also more than binary categories, like type of dog etc.. So the labels are *classes.*

Regressions predict a *numerical* outcome, such as the price of a house or the height of a person. So the labels are *values*.

Unsupervised and Reinforcement Learning

UNSUPERVISED LEARNING

In unsupervised ML, the data has no labels for the correct outcomes. The applications are vast: to reducing the dimensionality of a dataset to a fewer useful features, to grouping similar items, to building music recommendation systems.

REINFORCEMENT LEARNING

Are algorithms that learn from taking certain actions and receiving some reward or penalty for those actions. Applications examples are self-driving cars, and game playing agents such as AlphaGo.

Lesson 16: Decision Trees

Tree Anatomy

Internal nodes are points along the tree where the predictor space is **split**.

Terminal nodes or leaves are the nodes at the bottom of the tree, which are not split (in the sense that the leaves are at the *bottom* of the tree, decision trees are *upside down*).

Branches are the segments of the tree that connect the nodes.

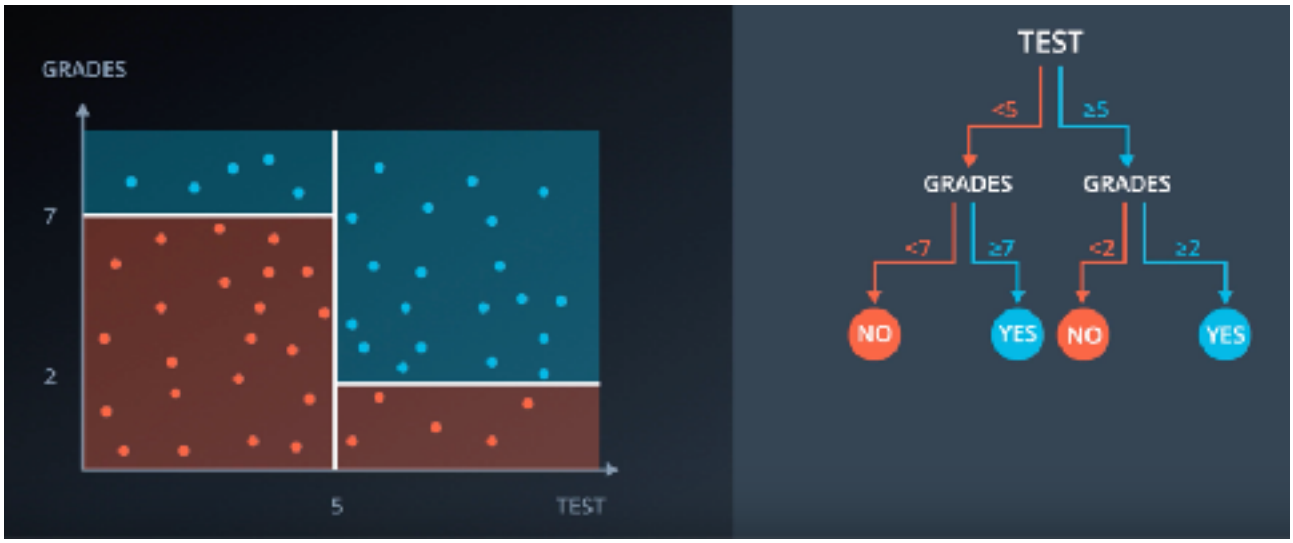
The depth is the number of levels in the tree.

This example has 2 internal nodes, 3 leaves and 2 levels. (are levels always the same number as internal nodes then? No, because you can have more than one node per level, see next…)

Example: Students Admissions
First we would want to find the horizontal or vertical line that does the best job at correctly splitting (classifying) the data. Here, the vertical line does a better job than the horizontal line, so that is based on a position on the x-axis, ie: the category is based on the test scores.

This still leaves incorrectly classified dots, so now we can use a horizontal line to split the remaining data as a second additional classification requirement. In this case, the horizontal line is positioned using the y-axis, ie: grades.

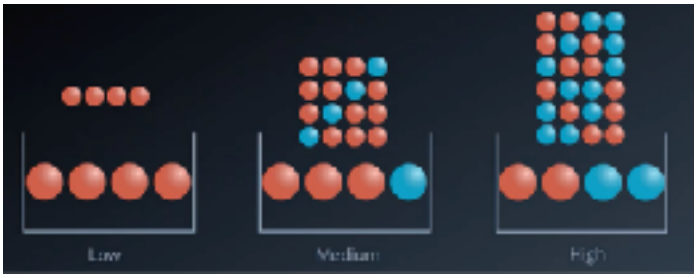
This is the process for constructing a decision tree. The first split is the first node (test) with the decision threshold from the position of the vertical line on the x-axis (the test scores), and the second split is the second level with 2 'grade' nodes (the number of times you use grades [draw a line] to split the remaining data).



Entropy

Gives example of ice, water, and vapour, where entropy measures the amount of freedom a particle has to move around. The entropy of ice is low, water is medium and vapor is high.

This same framework can be applied to probabilities. Taking 2 types of balls and 3 buckets, one where all 4 balls are same, a second bucket where one ball is different, and a last bucket which has two types of one ball and two types of another ball. The first bucket has low entropy, the second medium and third high entropy, as thought of by how many combinations of arrangments/orders of balls you can make in each of the buckets. This is not the exact definition of entropy, but helps in understanding the concept.



Another way to express entropy is in how much knowledge we have about the type of ball in each bucket. In the first bucket with low entropy we have high knowledge about the type of ball since there is only one type, and so on. So entropy is the measure of uncertainty (noise) and the opposite of information (signal).

ENTROPY FORMULA

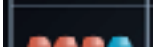
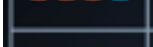
Using the medium bucket of balls, try to replicate a '3-red, 1-blue' pattern by picking at random, with repetition and replacement (putting the picked ball back in the bucket; means the picking are all independent events).

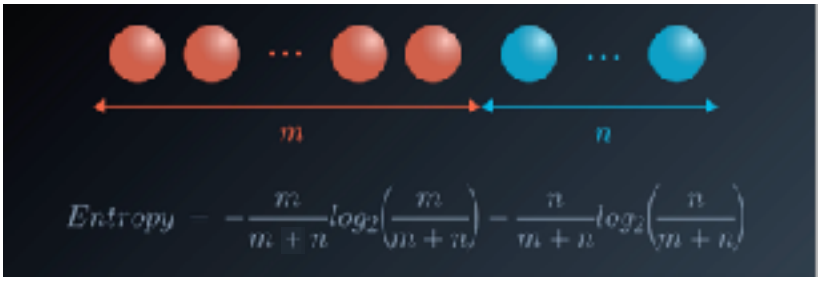
When events are independent, the probability of picking all 4 red is the probability of picking 1 red X 4. So the first bucket, the prob of picking a red ball is 100%, so the prob of 4 reds is 1x1x1x1 = 1.

The second bucket, the prob of picking red is 3/4 or 75%, so the prob of 4 reds is 3/4 x 3/4 x 3/4 x 3/4 = 3^4 / 4^4 = 27/256. Similarly, the prob of matching the '3-red, 1-blue' arrangement can be calculated as 3^3/4^3 x 1/4 (which is prob of picking blue) = 0.105.

The third bucket to match its arrangement of '2-red, 2-blue' would be: 0.5^2 x 0.5^2 = 0.5^4 = 0.0625.

In practice, products of many small numbers creates problems (results in diminishingly small numbers). Sums are better, and products can be turned into sums by using log. This class uses log_base 2. Since logs of < 1 produce negative numbers, we simply take the negative of log, so -log_base 2. Finally, when using sums, to get the entropy we take the average of the sums.

	P (red)	P (blue)	P (winning)	$-\log_2$ (P winning)	Entropy (average)
	1	0	$1 \times 1 \times 1 \times 1 = 1$	$0 + 0 + 0 + 0 = 0$	0
	0.75	0.25	$0.75 \times 0.75 \times 0.75 \times 0.25 = 0.105$	$0.415 + 0.415 + 0.415 + 2 = 3.245$	0.81
	0.5	0.5	$0.5 \times 0.5 \times 0.5 \times 0.5 = 0.0625$	$1 + 1 + 1 + 1 = 4$	1



So the general formula for entropy is:

What is the entropy for a bucket with a ratio of four red balls to ten blue balls?

red balls blue balls

- 4/14 * log2 (4/14) - 10/14 * log2(10/14) = 0.8632 *numpy is np.log2(4/14) etc….*

We can state this in terms of probabilities instead for the number of red balls as p_1 and the number of blue balls as p_2:

p_1 = m / m+n p_2 = n / m+n

So:

entropy = — p_1 * log2(p_1) — p_2 * log2(p_2)

This entropy equation can be extended to the multi-class case, where we have three or more possible values:

$$entropy = -p_1 \log_2(p_1) - p_2 \log_2(p_2) - ... - p_n \log_2(p_n) = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

The minimum value is still 0, when all elements are of the same value. The maximum value is still achieved when the outcome probabilities are the same, but the upper limit increases with the number of different outcomes. (For example, you can verify the maximum entropy is 2 if there are four different possibilities, each with probability 0.25.)

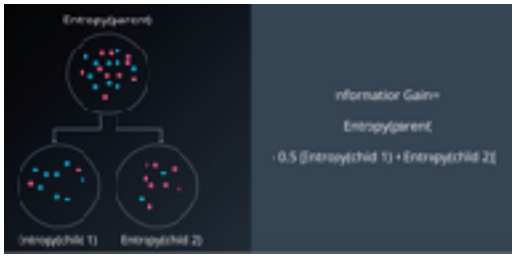
If we have a bucket with 8 red balls, 3 blue balls, and 2 yellow balls, what is the entropy of the set of balls?

= 1.3346 (see Entropy.py file where I defined multiclass entropy function)

Information Gain

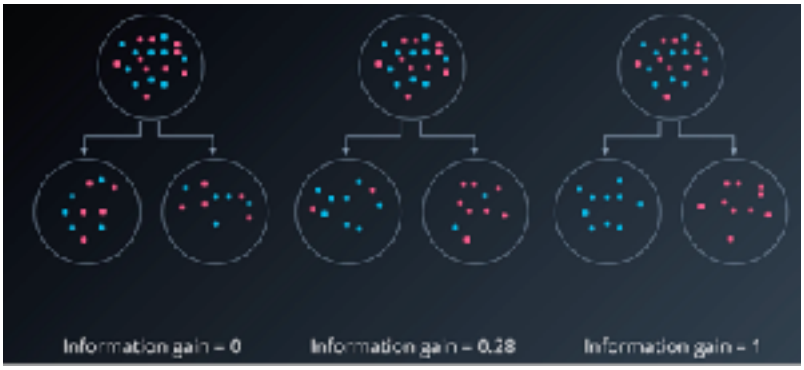
Defined as the difference between the entropy of the parent node and the weighted average entropy of the child nodes:

(this slide example has equal weighted child nodes…)



$$\text{Information Gain} = \text{Entropy}(Parent) - \left[\frac{m}{m+n}\text{Entropy}(Child_1) + \frac{n}{m+n}\text{Entropy}(Ch\right.$$

Here are the examples from the lecture:



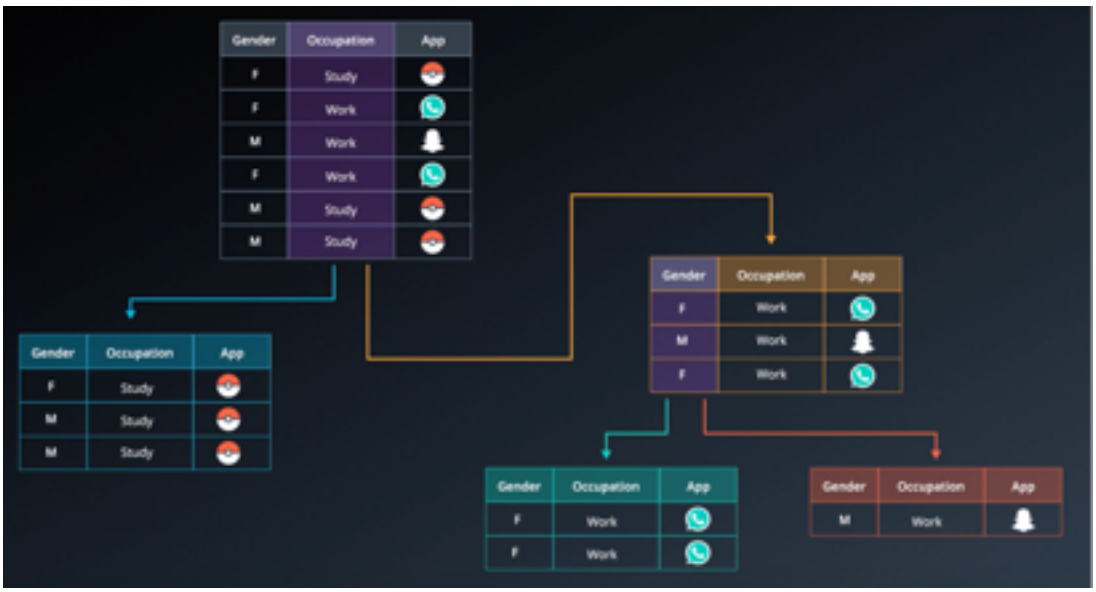
Here is the example decision tree for recommending an app:

- *Calculate the entropy of the 'app' column (which will be the **parent** node):*
  - *There are 3 Pokemon; 2 whatsapp; 1 snapchat; so the entropy calc is:*
    - *-3/6 * log2(3/6) - 1/3 * log2(1/3) - 1/6 * log2(1/6) = 1.46*

- *Calculate the entropy of the '**occupation**' column (which will be a child node):*
  - *'Work' has 2 whatsapp, 1 Snapchat; so entropy = -2/3 * log2(2/3) - 1/3 * log2(1/3) = 0.918*
  - *'Study' has all Pokemon, so entropy = 0*
  - *So their average entropy is 0.459*
  - *So information gain here is 1.46 - 0.459 = **1.01***

- *Calculate the entropy of the '**gender**' column (which will be a child node):*
  - *'Male' has 2 Pokemon, 1 Snapchat; so entropy = -2/3*log2(2/3) - 1/3*log2(1/3) = 0.918*
  - *'Female' has same ratio so same entopy: -1/2 * log2(1/2) - 1/2 * log2(1/2) = 0.918*
  - *So their average entropy is also 0.918*
  - *So information gain here is 1.46 - 0.918 = **0.542***

So splitting by the occupation columns gives a higher information gain so the decision tree algorithm will choose to split on that column first, and gender second



The same approach can be taken with continuous data (rather than this discrete data example) as we saw in the beginning with 'student admissions'.

Gini Impurity

There is another alternative for measuring the quality of a split. If there are ***k*** classes, and ***p_hat_k*** is the fraction of observations from class ***k*** classified by the node, we can calculate ***G*** (the Gini index) for the node:

$$G = \sum_{k=1}^{K} \hat{p}_k(1 - \hat{p}_k)$$

The Gini index takes on a small value if all of the proportions are close to zero or one. You can think of it as a measure of node *purity*—if the value is small, the node mostly contains observations from a single class. It turns out that the Gini index and entropy are quite similar numerically.

To measure the *increase in purity* of a split using the Gini index, calculate the Gini index on the parent node and subtract the weighted average of the Gini indexes of the child nodes:

$$G_{\text{increase}} = G_{\text{parent}} - \sum_{\text{children}} (\text{fraction of observations})_{\text{child}} \times G_{\text{child}}$$

Scikit-learn supports both the Gini impurity and information gain metrics for evaluating the quality of splits, via the `criterion` hyperparameter.

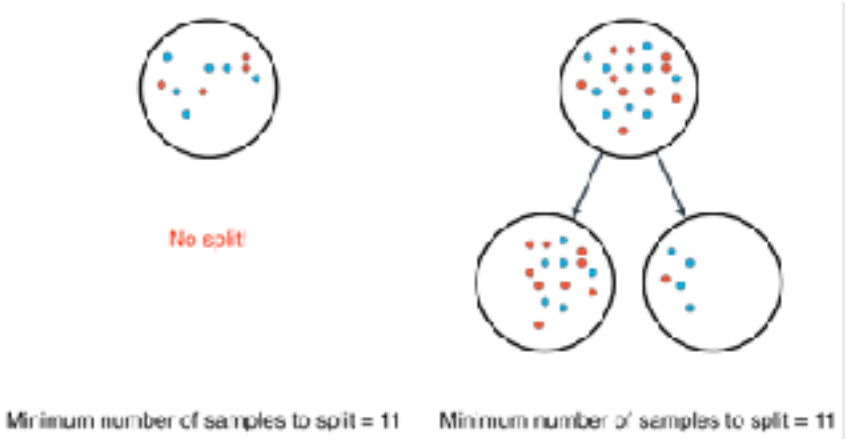
Hyperparameters for Decision Trees

In order to create decision trees that will generalize to new problems well, we can tune a number of different aspects about the trees. We call the different aspects of a decision tree "hyperparameters". These are some of the most important hyperparameters used in decision trees:

MAXIMUM DEPTH
The maximum depth of a decision tree is simply the largest possible length between the root to a leaf. A tree of maximum length ***k*** can have at most *2^k* leaves:

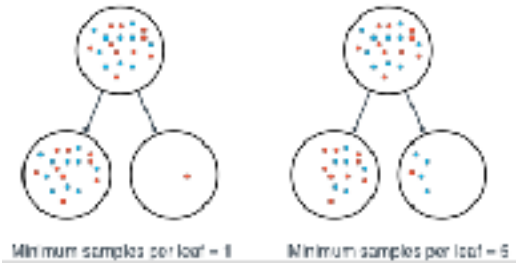
MINIMUM NUMBER OF SAMPLES TO SPLIT
A node must have at least `min_samples_split` samples in order to be large enough to split. If a node has fewer samples than `min_samples_split` samples, it will not be split, and the splitting process stops.

However, `min_samples_split` doesn't control the minimum size of leaves. As you can see in the example on the right, above, the parent node had 20 samples, greater than `min_samples_split` = 11, so the node was split. But when the node was split, a child node was created with that had 5 samples, less than `min_samples_split` = 11.



MINIMUM NUMBER OF SAMPLES PER LEAF
When splitting a node, one could run into the problem of having 99 samples in one of them, and 1 on the other. This will not take us too far in our process, and would be a waste of resources and time. If we want to avoid this, we can set a minimum for the number of samples we allow on each leaf.

This number can be specified as an integer or as a float. If it's an integer, it's the minimum number of samples allowed in a leaf. If it's a float, it's the minimum percentage of samples allowed in a leaf. For example, 0.1, or 10%, implies that a particular split will not be allowed if one of the leaves that results contains less than 10% of the samples in the dataset.



If a threshold on a feature results in a leaf that has fewer samples than `min_samples_leaf`, the algorithm will not allow *that* split, but it may perform a split on the same feature at a *different threshold*, that *does* satisfy `min_samples_leaf`.

Visualizing a Decision Tree

Once we have created a decision tree using sklearn, we can easily visualize it by exporting the tree in Graphviz format, using Graphviz open source graph visualization software.

export_graphviz
Use `sklearn.tree.export_graphviz()` to export the tree into DOT format. **DOT** is GraphViz's text file format. It includes human-readable syntax that describes the appearance of the tree graph, including the content of subtrees and the appearance of nodes (i.e. color, width, label).

So for example, assume `model` is an instance of `DecisionTreeClassifier()`, and you've already called `model.fit()`. Then export to DOT format as follows:

```
dot_data = export_graphviz(model)
```

actually, dot_data requires a ton more parameter inputs for the code example they gave…

There are a lot of options you can specify at this step, which you can explore in the documentation **here**. In particular, you can save the data to a file, you can specify whether and how to label the nodes, and you can rotate the tree.

graphviz.Source
To render a ready-made DOT source code string, create a `Source` object holding your DOT string.

```
from graphviz import Source
graph = graphviz.Source(dot_data)
```

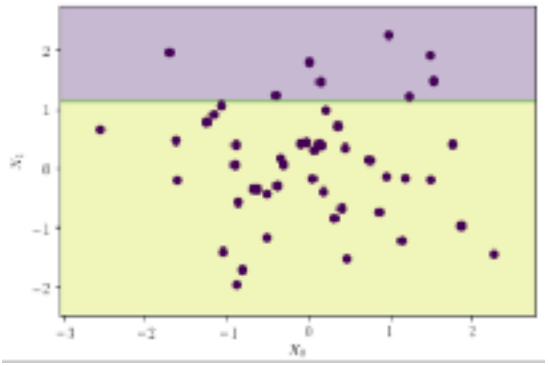
Then, display the graph directly in the Jupyter notebook:

```
graph
```

Example

So, for example, if we create the following small dataset,

```
rng = np.random.RandomState(0)
X = rng.normal(size=(50, 2))
```



and set a target variable, y, equal to 0 by default, and equal to 1 if X_1 > 1.2,

```
y = np.zeros(X.shape[0], dtype=np.int)
y[X[:, 1] > 1.2] = 1
```

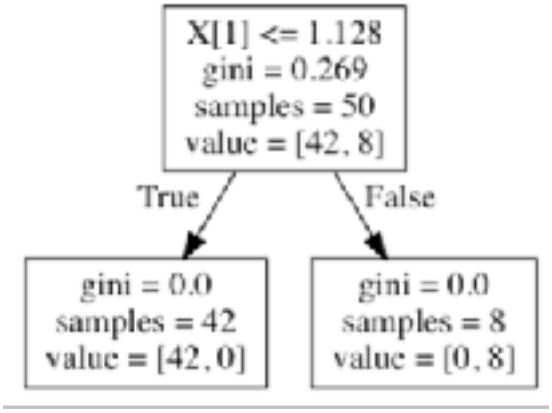
and fit a tree to it,

```
tree = DecisionTreeClassifier().fit(X, y)
```

we can run the following code,

```
from sklearn.tree import export_graphviz
from graphviz import Source

treedot = export_graphviz(tree, out_file=None)
treegraph = Source(treedot)
treegraph
```

and we'll then see the following:



Lesson 17: Model Testing and Evaluation

Confusion Matrix

Sometimes in the literature, you'll see False Positives and False Negatives as Type 1 and Type 2 errors. Here is the correspondence:

- **Type 1 Error (Error of the first kind, or False Positive):** In the medical example, this is when we misdiagnose a healthy patient as sick.
- **Type 2 Error (Error of the second kind, or False Negative):** In the medical example, this is when we misdiagnose a sick patient as healthy.

ACCURACY
Total correct classifications (ie: positive and negative) / total classifications (ie: # examples)

For above confusion matrix, this is: (1000 + 8000) / 10,000 = 90%

```
from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_pred)
```

When accuracy won't work
Imbalanced data sets, like credit card fraud

- It has very high accuracy, but doesn't catch any of the fraudulent transactions

FALSE NEGATIVES AND POSITIVES
Need to decide, based on domain knowledge, which is a more costly mistake. If, like in the sick patients example, a false negative is a worse mistake, you want to test for **high recall**. And if, as in the spam email example, a false positive (non-spam gets moved to spam) is worse, you need to test for **high precision**.

PRECISION
Using the healthy/sick matrix, precision is measured as, out of the all the patients diagnosed as sick (1st col), how many were classified correctly? 1000/ 1800 = 55.5%

- Although low, it is not the most important measure

For spam email, 100 were correct (true positive) out of 130 sent to spam folder (TP+FN) = 76.9%

- Not high enough considering precision is what we want out of a spam filter

So precision is: TP / (TP + FN)

RECALL
Recall asks: out of the points that were labeled as positive, how many were correctly predicted as positive?

For the healthy/sick matrix, how many were correctly diagnosed as sick?

1000 were labeled positive (sick) out of 1000+200 that actually were sick = 83.3% recall

- Which is the 'sick' cell divided by the 'sick' row
- Score is not good, model is sending too many sick people home (misclassifying)

So recall is: TP / (TP+FP)

Types of Errors

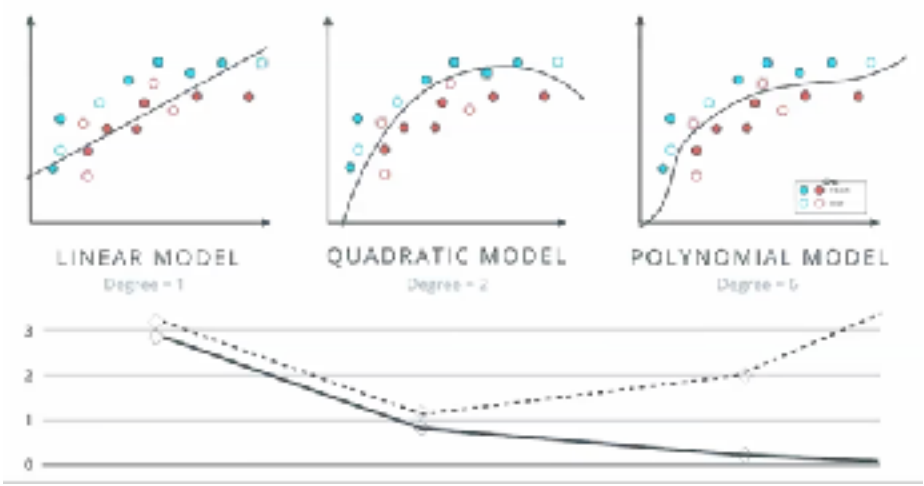
Oversimplifying the problem (killing Godzilla with a fly swatter) = underfitting
- Doesn't do well on training set
- Is called bias error

Overcomplicating the problem (killing a fly with a bazooka) = overfitting
- Doesn't do well on test set
- Is called variance error

MODEL COMPLEXITY GRAPH

Solid dots = train set; empty dots = test set

Line below scatter plots is called the model complexity graph
- It shows counts of train and test errors, in this case for the polynomial overfitting the training set and performing poorly on the test set



Cross Validation

Using the model complexity graph to decide on the best model breaks the rule of not using the test set. Enter cross validation.

Just add an additional hold out (validation) set to use for the model complexity graph / model hyperparams tuning, and the final test set remains untouched.

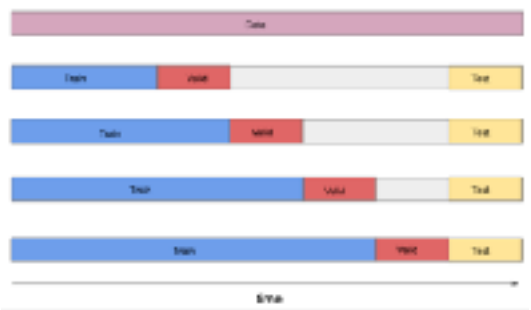
K-FOLD CROSS VALIDATION

Helps to avoid throwing away too much of the dataset to hold out sets and not having enough for training.

In sklearn, create a KFold object with the size of the data and the size of the fold, and is good to randomize the folds by setting shuffle (if not time series):

```
from sklearn.model_selection import KFold
kf = KFold(12, 3, shuffle=True)

for train_indices, test_indices in kf:
 print train_indices, test_indices
```

For time series we need to modify the k fold approach this way (right):



This way, each training set consists only of observations that occurred prior to the observations that form the validation set. Likewise, both the training and validation sets consist only of observations that occurred prior to the observations that form the test set. Thus, no future observations can be used in constructing the forecast.

VALIDATION FOR FINANCIAL DATA

Furthermore, when working with financial data, we can bring practitioners' knowledge of markets and financial data to bear on our validation procedures. We know that since markets are competitive, factors decay over time; signals that may have worked well in the past may no longer work well by the current time. For this reason, we should generally test and validate on the most recent data possible, as testing on the recent past could be considered the most demanding test.

It's possible that the design of the model may cause it to perform better or worse in different market regimes; so the most recent time period may not be in a market regime in which the model would perform well. But generally, we still prefer to use most recent data to test if the model would work in the time most similar to the present. In practice, of course, before investing a lot of money in a strategy, we would allow time to elapse without changing the model, and test its performance with this true out-of-sample data: what's known as "paper trading".

In summary, most common practice is to keep a block of data from the most recent time period as your *test* set.

Then, the data are split into train, valid and test sets according to the following schematic:

When working with data that are indexed by asset and day, it's important not to split data for the same day, but for different assets, among sets. This would manifest as a subtle form of lookahead bias. For example, say data from Coca-Cola and Pepsi for the same day ended up in different sets. Since they are very similar companies, one might expect their share price trends to be correlated. If the model were trained on data from one company, and then validated on data from the other company, it might "learn" about a price movement that affects both companies, and therefore have artificially inflated performance on the validation set.

*** I'm not clear what they are trying to say here? I think just split on calendar dates across time zones? Not a % split… (has to do with multi-indexing, so don't split in the middle of universe of stocks by % index, make sure to index by the specific date…)*

LEARNING CURVES

- Underfit models (high bias) training curves and CV curves converge but at a high level
- Good models coverge at a low point
- Overfit models (high variance) do not converge (because train error stays low due to overfitting while CV error stays high as model does not generalize)

Lesson 18: Random Forests

Ensemble Methods

To make decision trees more powerful (their prediction accuracy is usually far inferior to other ML models) and reduce overfitting (which they tend to do), we can combine several weaker models (in this case, individual decision trees) together to make a more powerful model. The constituent models are called **weak learners**, while the combined model is called the **strong learner**. Combining many models together to yield a more powerful model is called **ensembling**.

A key part of ensembling, though, is that the constituent models are not the same. In fact, ensembles tend to yield better results when the constituent models are very *different*. So how do we use the same dataset to grow many *different* trees? There are actually many ways.

- For every tree, create a new dataset by drawing a random subset of **rows** from the original dataset. Train the tree on this new dataset.
- For every tree, create a new dataset by drawing a random subset of rows from the original dataset **with replacement**. Train the tree on this new dataset.
- For every tree, create a new dataset by drawing a random subset of **columns** from the original dataset. Train the tree on this new dataset.

These are examples of "perturbations"—ways to "shake up" the constituent trees in order to ensure that they are different from each other. These are all examples of ways to introduce "perturbations" *randomly* and *independently*.

In contrast, there's another class of methods where perturbations (on a given training set) are chosen deterministically and serially, with the nth perturbation depending strongly on all of the previously generated rules. So that we have more time to talk about the random and independent methods, we're not going to talk more about the deterministic and serial methods for now.

PERTURBATIONS ON COLUMNS

As we previously discussed, one way to generate different trees is to subsample the set of *columns* the trees use to generate splits. When random subsets of the dataset are drawn as random subsets of features, then the method is known as the **Random Subspace Method**.

Importance of Random Column Selection

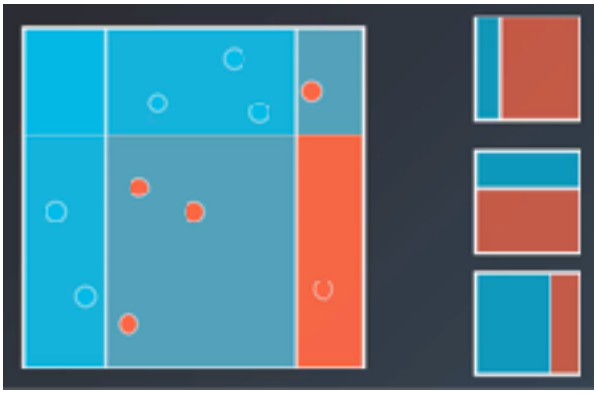
Sometimes one feature will dominate in finance. If you don't apply some type of random feature selection, then your trees will not be that different (i.e., they will be correlated) and that reduces the benefit of ensembling.

What features are typically dominant? Well, we'll talk about this more later when we talk about feature engineering, but when we use random forests for alpha combination, some of our features are alpha factors. **Classical, price-driven factors, like mean reversion or momentum factors, often dominate.**

You may also see that **features that define industry sectors or market "regimes"** (periods defined, for example, by high or low market volatility or other market-wide trends) **are towards the root of the tree.**

PERTURBATIONS ON ROWS

Another way to generate different trees is to grow each tree on a random subset of the original dataset's *rows*. Subsets can be generated with or without replacement. When it's done with replacement, it's called **bagging**, and when it's done without replacement, it's called **pasting**. *Bagging* is short for **bootstrap aggregating**.



The right-hand splits are from random subsets of rows (with replacement), which are then combined on the left. When combined, the highest votes wins, ie, whatever the majority of the weak learners predict is what we use in our final strong learner model.

Random Forests

Random Forests are ensemble prediction algorithms that use *both* random column and random row selection. Each tree in the ensemble is created as follows:

Step 1: bagging

- If the number of rows in the training dataset is N, generate the dataset for each constituent tree by choosing N rows at random — but with replacement — from the original data.
  - This is called bootstrap sampling
  - You end up with another dataset with the same number of rows 'N', but because the rows are selected at random, some rows may be repeated and others excluded

Step 2: growing the tree

- If there are M columns (features) in the training dataset, pick a number m<<M.
- At each split (node), select m columns at random out of the M and use the best split of possible splits on these m columns to split the node.
- Grow the tree to the largest extent possible (max depth), ie: until leaves are 'pure' (can't be split anymore, contain all the same class)
  - The value of m is held constant during the forest growing.
  - m is known as the `max_features` parameter, and the **default value is sqrt(M).**
- Repeat to grow more trees to build a forest

When you have all your trees, you just let each of the trees vote when you need to classify a new example (make a prediction), the highest voted outcome becomes the prediction.

For a **regression** tree model, use the **average** value of the ensemble of trees' predictions. For a classification model, use the mode of the ensemble of trees' predictions.

To learn more, check out **this** paper.

Out of Bag Estimate

When using bagging in creating random forest, the random row sampling of the bootstrap process roughly uses only about 2/3 of the rows from the original data set (with replacement, so new data set still has same number of rows as original, just with some repeats). The 1/3 unused rows for a given weak-learner tree are called 'out-of-bag' (OOB) observations.

The OOB score is the aggregate score of all the trees that did not have that observation in their bag.

For a given observation we can create an OOB score, as though they were trees being used to make label predictions. Since an observation (row) is not in 1/3 of the bags (data set used for a tree), then if ***B*** total observations is **100**, then ***OOB predictions*** will be **100 * 1/3**.

These predictions can be aggregated to make a single OOB prediction for each observation. For a regression model they can average, and for classification, a majority vote can be used. Then the OOB error (score) can be calculated for the entire data set. In doing so, it is like we have obtained a test error score since the observations in the trees being evaluated were not in the model bags (ie: the model was not fit using the OOB observations).

Choosing Hyperparameters

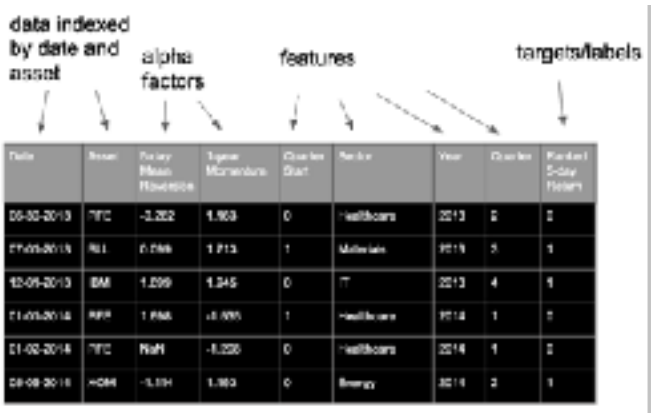
Let's say we are trying to choose the `min_samples_leaf` hyperparameter, and want to avoid overfitting. How many training samples would we choose to be the minimum per leaf? In non-financial and non-time series machine learning, setting this hyperparameter is fairly straightforward: you use grid search cross-validation to find the value that maximizes the model's performance on validation data.

When you have time-series data, you typically don't use cross-validation because usually you just want a single validation dataset that is as close in time as possible to the present. If you have a problem with high signal-to-noise, then you can try a bit of parameter tuning on the single validation set. In finance, though, you have time series data *and* you have low signal-to-noise. Therefore, you have one validation set and if you were to try a bunch of parameter values on this validation set, you would almost surely be overfitting.

As such, you need to set the parameter with some judgement and minimal trials. Later, we'll discuss a bit more about how we make this choice in the project.

Random Forests for Alpha Combination

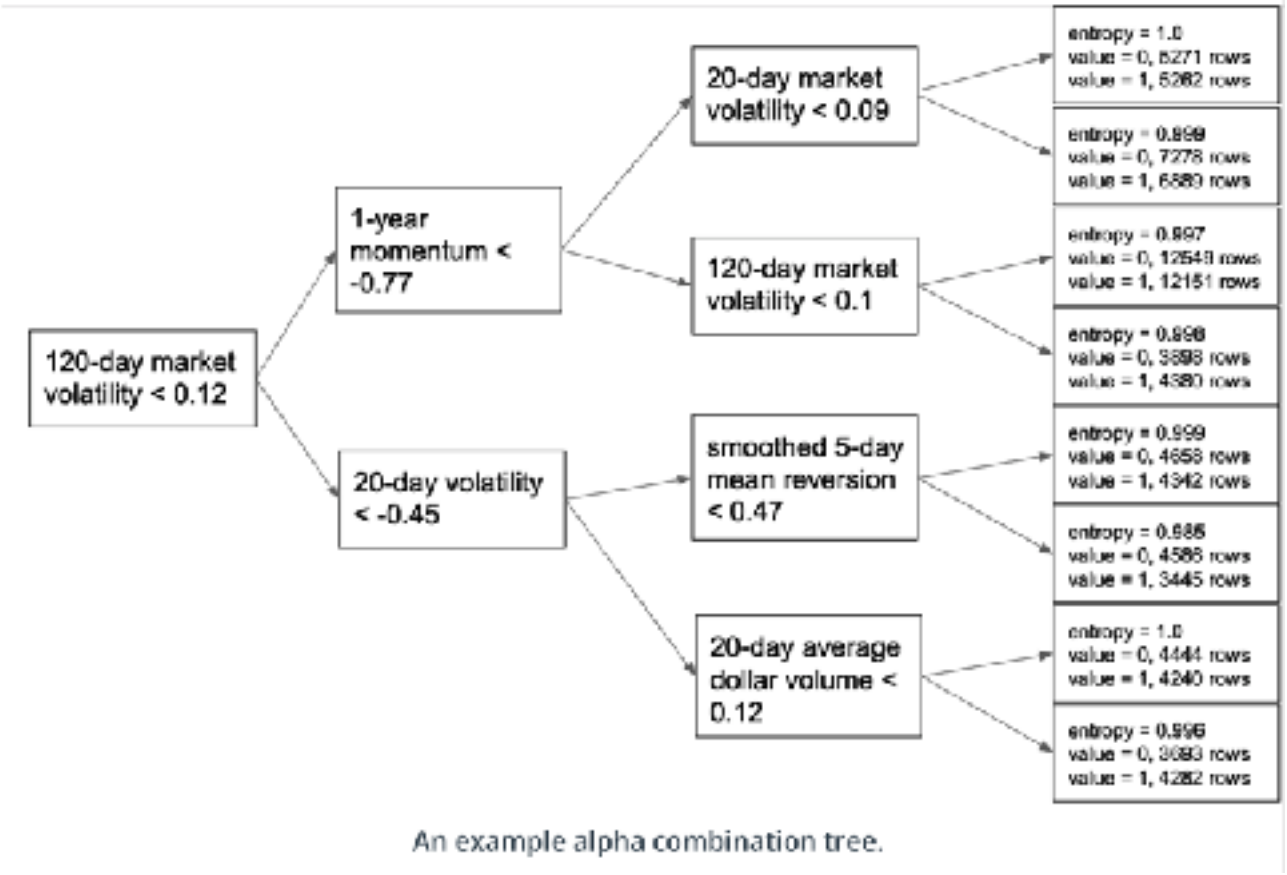
So we've seen how random forests are used for certain problems, like predicting a consumer's app preference from personal data. How do we use random forests for alpha combination?



For this type of problem, we have data like this: each row is indexed by both date and asset. We typically have several alpha factors, and we then calculate "features", which provide the random forest model additional information. For example, we may calculate date features, which the algorithm could use to learn that certain factors are

particularly predictive during certain periods.

What are we trying to predict? We're trying to predict asset returns—but not their decimal values! We rank them relative to each other into only two buckets, such that we essentially predict winners and losers on the day. The next lesson is all about feature engineering, so let's move on to learn more about features and labels in more detail!



An example alpha combination tree.

Lesson 19: Feature Engineering

Universal Quant Features

These are slightly more general than alpha factors, which are ideally buy or sell signals. Features are similar but don't have as high a requirement to be predictive on their own. In fact, they are in most case meant to be combined together with other features to build predictive power — ie, a conditional factor (like idiosyncratic volatility we say before, which could be interpreted as magnifying a signal). Universal quant features are standard/common 'conditional factors' that serve to enhance a signal. Here are a couple useful ones.

STOCK VOLATILITY
The one month and six month volatility (annualized) can be a useful feature.

STOCK DOLLAR VOLUME
When volume is higher than normal and coincides with a signal, it is likely that the signal is more meaningful than if the volume is lower. We will also use one month and six month time frames.

Market Regimes

Refers to changing market environments that broadly affect how stocks behave over a period of time. For example, range-bound, trending, etc. and it is helpful to generate features that capture changes in these market conditions. Market dispersion and market volatility are two such features.

MARKET DISPERSION
The variation of performance of the stocks within a universe. If dispersion is low/(high) then it is more difficult/(impactful) using a long-short strategy.

Market dispersion can be calculated as a standard deviation, ie: how tight or loose the distribution of returns are for the stocks in the universe. Market dispersion measures how much investors are making a distinction between stocks versus common market factors. We will again use 1 month and 6 month time periods for measurement.

MARKET VOLATILITY
The volatility of the stock universe of a time horizon. Theoretically, when market volatility is high it is usually due to higher perceived risk which market participants manage by trading more, creating temporary imbalances which **mean revert** once their liquidity demand dries up (ie: once they trade less).

Similarly, when market volatility is low, it could be expected that idiosyncratic stock factors will drive stock movements more, as a stock's specific fundamentals are not drowned out by high market volatility.

Using 1 month and 6 month periods of market volatility, we can assess whether short-term volatility is higher or lower than long-term volatility. A way to approximate the market is to take the equal-weighted average of all the stocks in the defined universe, then calculate the std dev of these returns over a time horizon. Key takeaway is mean reversion strategies are more likely during market regimes with high market volatility, while low market volatility regimes will likely better reflect individual stock fundamentals.

Sector or Industry as Feature

Other than stocks within a sector or industry moving together, its also useful to know that certain alpha factors may or may not be relevant depending on the sector. For instance, a value factor does not make much sense for a bank, nor return on sales etc. We can train a model to learn when some factors are relevant or not.

(see Jupiter notebook lesson, where sector is added as a feature)

Date Parts

There are many seasonal and calendar affects in the market, such as quarter-end window dressing, year-end window-dressing, short-term traders closing out positions before the weekend etc. It can be useful, therefore, to know which dates represent the end of the week/quarter/year etc.

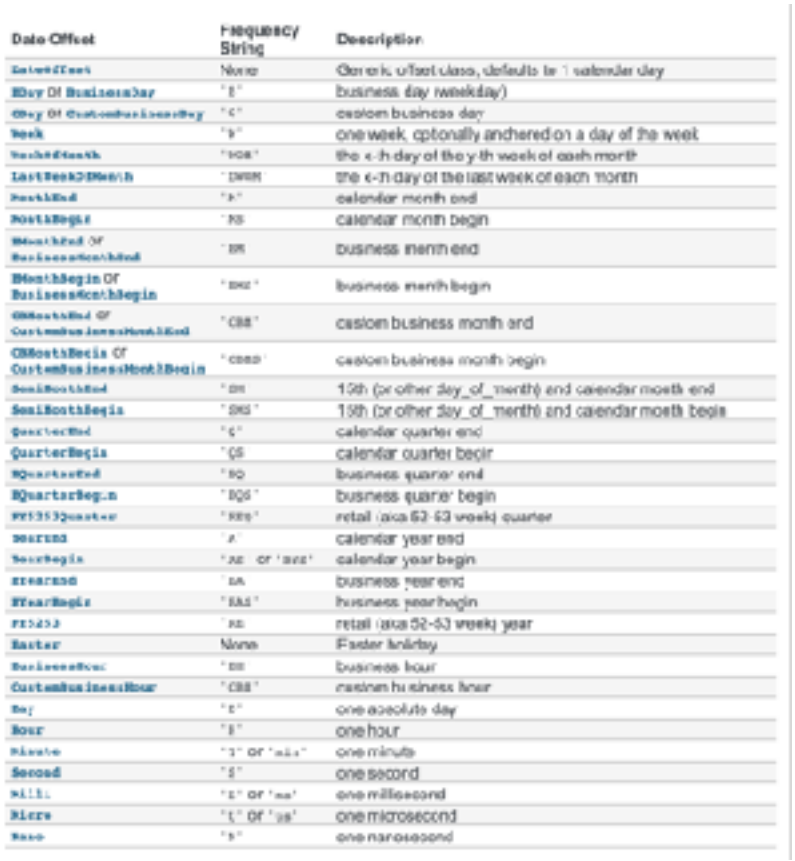
We can do this by accessing the 'dates' from the index of the data frame:

```
all_factors.index.get_level_values(0)    # get_level_values is index level; 1 is tickers
```

Then use datetime attributes:

- The `month` attribute is a numpy array with a 1 for January, 2 for February ... 12 for December etc.
- We can use a comparison operator such as `==` to return True or False.
- It's usually easier to have all data of a similar type (numeric), so we recommend converting booleans to integers.
  The numpy ndarray has a function `.astype()` that can cast the data to a specified type.
  For instance, `astype(int)` converts False to 0 and True to 1.

Pandas date freq shortcuts:



Date Offset	Frequency String	Description
[illegible]	None	Generic offset class, defaults to 1 calendar day
BDay or BusinessDay	'B'	business day (weekday)
CDay or CustomBusinessDay	'C'	custom business day
Week	'W'	one week, optionally anchored on a day of the week
WeekOfMonth	'WOM'	the x-th day of the y-th week of each month
LastWeekOfMonth	'LWOM'	the x-th day of the last week of each month
MonthEnd	'M'	calendar month end
MonthBegin	'MS'	calendar month begin
BMonthEnd or BusinessMonthEnd	'BM'	business month end
BMonthBegin or BusinessMonthBegin	'BMS'	business month begin
CBMonthEnd or CustomBusinessMonthEnd	'CBM'	custom business month end
CBMonthBegin or CustomBusinessMonthBegin	'CBMS'	custom business month begin
SemiMonthEnd	'SM'	15th (or other day_of_month) and calendar month end
SemiMonthBegin	'SMS'	15th (or other day_of_month) and calendar month begin
QuarterEnd	'Q'	calendar quarter end
QuarterBegin	'QS'	calendar quarter begin
BQuarterEnd	'BQ'	business quarter end
BQuarterBegin	'BQS'	business quarter begin
FY5253Quarter	'REQ'	retail (aka 52-53 week) quarter
YearEnd	'A'	calendar year end
YearBegin	'AS' or 'BYS'	calendar year begin
BYearEnd	'BA'	business year end
BYearBegin	'BAS'	business year begin
FY5253	'RE'	retail (aka 52-53 week) year
Easter	None	Easter holiday
BusinessHour	'BH'	business hour
CustomBusinessHour	'CBH'	custom business hour
Day	'D'	one absolute day
Hour	'H'	one hour
Minute	'T' or 'min'	one minute
Second	'S'	one second
Milli	'L' or 'ms'	one millisecond
Micro	'U' or 'us'	one microsecond
Nano	'N'	one nanosecond

Targets (Labels)

Output of ML models. In term 1 we used forward returns as targets, but the low signal-noise ratio means we should convert these into a simpler signal for the model to learn from.

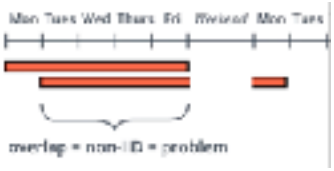
One way could be to classify returns as either positive of negative (2 labels). The downside from this is that we then can't distinguish between the strength (or weakness) of those returns.

A way around this is to require certain thresholds to bucket returns into, like low/medium/high positive returns.

But the next problem becomes how to determine these thresholds. Constant thresholds may be relevant or appropriate in some market regimes, but not in others. Having constant thresholds also doesn't allow us to neutralize market risk, since in a strong positive market, all stock returns would be positive and none in the negative buckets.

A nice solution is to quantize the returns into their categories, (kind of like rank.zscore does?).

Lesson 20: Overlapping Labels



The Non-IID Problem

The calculation for labels must be independent and identically distributed (IID):

ML model work under the assumption that rows containing the example data and labels are IID, ie: non-overlapping.

In the case of decision trees, if the bootstrap samples are made from highly overlapping samples, then the observations in the bag are likely to be similar to each other, and also similar to the observations outside the bag. This will result in correlated predictions, which in turn means an increase in the error rate == basically a case of overfitting.

POTENTIAL SOLUTIONS

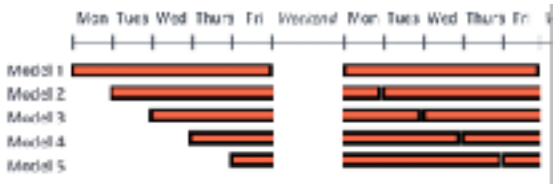
1. A simple solution would be to make sure calculations for weekly returns don't overlap, but then you lose a significant (4/5) of data to work with.

2. Use the overlapping data, but adjust the bagging procedure inside the random forest model. The idea is to pick a smaller number of samples in each bag in order to reduce the amount of redundant information

   Changing the number of samples in sklearn RandomForestClassifier is not an option. But sklearn does have a model called BaggingClassifier, which uses the max_samples parameter to adjust sample size.

   One recommendation is to reduce the size of each bag to be a fraction of the number of rows of the original dataset. For the fraction, use 1 divided by the number of labels that overlap at each time point, on average. So if you were using weekly returns, the fraction would be ⅕, or 0.2.

3. Use ***n*** different non-overlapping models to make predictions (where ***n*** is the number of overlaps), and then ensemble the models together for a final result.



Lesson 21: Feature Importance

Feature Importance in sklearn

Sklearn measures feature importance in decision trees by measuring and comparing impurity between the parent and its child nodes, after splitting on a feature. If after splitting on a feature it is able to reduce the impurity in the child nodes, then it stands to reason that that is a measure of its importance as a feature.

*** see sklearn_exercise notebook*

WHEN FEATURE IMPORTANCE MEASURES ARE INCONSISTENT
There are many of methods for interpreting machine learning models, and for measuring feature importance. Many of these methods can be inconsistent, which means that the features that are most important may not always be given the highest feature importance score. We noticed this in the prior coding exercise, where there were two equally important features that form the "AND" operator, but one was given a feature importance of 0.33 because it was used for splitting the tree first, and the other was given a score of 0.67 because it was used for splitting second.

This is the motivation for using the latest feature attribution method, Shapley Additive Explanations, which we'll see next.

If you wish to explore the concept of consistent feature attribution further, here's a blog post that discusses some of the inconsistency seen in feature importance calculation methods.
Interpretable Machine Learning with XGBoost

For project 7 insights:
https://datascience.stackexchange.com/questions/16693/interpreting-decision-tree-in-context-of-feature-importances
It is not necessary that a more important feature is a higher node in the decision tree.

This is simply because different criteria (e.g. Gini Impurity, Entropy-Information Gain, MSE etc) may be used at each of two these cases (splitting vs importance).

For example, at `SkLearn` you may choose to do the splitting of the nodes at the decision tree according to the Entropy-Information Gain criterion (see `criterion` & `'entropy'` at `SkLearn`) while the importance of the features is given by Gini Importance which is the mean decrease of the Gini Impurity for a given variable across all the trees of the random forest (see `feature_importances_` at `SkLearn` and here).

If I am right, at `SkLearn` the same applies even if you choose to do the splitting of the nodes at the decision tree according to the Gini Impurity criterion while the importance of the features is given by Gini Importance because Gini Impurity and Gini Importance are not identical (see also this and this on Stackoverflow about Gini Importance).

In scikit-learn the feature importance is the decrease in node impurity. The key is that it measures the importance only at a node level. Then, all the nodes are weighted by how many samples reach that node.

So, if only a few samples end up in the left node after the first split, this might not mean that J is the most important feature because the gain on the left node might only affect very few samples. If you additionally print out the number of samples in each node you might get a better picture of what is going on.

Shapely Additive Explanations

This is the state-of-the-art for interpreting ML models and is based on coalition game theory. For intuition, imagine you have a basketball team of three players (A, B and C), and you want to determine the contribution of each player to the final score of the team. One way would be to see how the team performs with player A, and how without player A. And try every possible sequence for all the players, measuring the difference for each combination.

This is the essence of Shapely Additive Explanations — assesses the performance of a model with and without a given feature to assess the strength of that feature, and do that for every possible combination of features.

This is a local test of feature importance for every observation rather than a global test such as what sklearn does. Feature importance may not be the same for each data point (such as in loan applications where each data point is an individual person), so local feature importance calculates the importance for each data point or observation. SHAP assigns each feature an importance value for a particular prediction.

Global feature importance refers to single ranking of features for the model. For our purposes we will use global feature importance and aggregate local feature importances into global importance.

*** the calculate_shap notebook has a great step-by-step breakdown of the formula to understand everything that is going on…*

Project 7

MY SUBMISSION COMMENTS:
1. It looks as though the accuracies of models ran with non-overlapping samples is exactly the same as when ran with non-overlapping-estimates -- is this correct?

2. It looks like the number of stocks changes (see the notebook). Each day should have the same number of stocks (c.470 or so), but the number of rows / number stocks is not a whole number, suggesting there may be survivorship bias?

3. How are we to interpret a final test accuracy of c.51%, essentially a coin toss, yet we still get a nice Sharpe ratio? The accuracy suggests this is not a great model, but the Sharpe ratio says otherwise?

Thanks! :]

implement the `non_overlapping_estimators` function.

You should not use the same dataset to fit all the classifiers. Please use `non_overlapping_samples` to build a list of none overlapping subsets of the data with different starting index. And then use each one of them to feed a classifier instead of using the same data to feed all classifiers.

Recall from the lessons that if we want to use weekly returns, we cannot simply use a sliding 5-day window of weekly returns to train the random forests — the overlapping dates will lead to drastic overfitting as seen by the oob scores.

The first attempt to mitigate this was in non_overlapping_samples where no sliding window is used, just the weekly returns, but this means only 1/5 of the data is being used.

This non_overlapping_estimators method brings back the sliding window approach, but trains the forests on each non-overlapping pass and collects the results. The final result can then be calculated as the average of all the forests trained. For example, the first pass would have Fri-to-Fri returns only to train the first forests, the second pass would have Mon-to-Mon returns for training, etc.

So your xx, yy = non_overlapping_samples(x, y, n_skip_samples, start_i=0) needs to go inside the loop, and the loop should be looping over the n_skip_samples. The clf.fit would then become classifiers.fit and iterated with the loop.

MY QUERIES ABOUT THE TARGET IN THE PROJECT
I have a question about the targets used in project 7.

Section: Target (after Regime Features in the notebook)
We created two different quantized versions of our weekly return — the first (return_5d) only quantizes the returns into two categories (positive or negative), and the second (return_5d_p) quantizes returns into 25 categories (allowing for a range of positive or negative returns to target).

Section: Shift Target
Later on when we shift the quantized return and add it to all_factors as the name ‘target’, we use the first quantized version, ie: return_5d (with only positive or negative categories of returns) to add as the last column of all_factors.

Section: IID Check of Target

When the IID section is run, we add target_1 through 4 and target_p (which second quantized target, shifted — but note that it is called target_p).

Section: Train/Valid/Test Splits (the cell implementing the function…)
Finally, when we create the train/valid/test splits, the code uses the 'target' label — not 'target_p'. Why is this? The lesson on this (lesson 19, video 10) clearly describes the limitations of using a binary quantization and instructs to use more quantiles.

Is this a mistake in the project? Should we use 2 quantiles or should we use more? Thanks!

TRAIN TEST SPLIT
I found the following quite helpful for creating the date splits https://stackoverflow.com/questions/38250710/how-to-split-data-into-3-sets-train-validation-and-test

This solution isn't directly applicable since the Dataframe is resampled. But **np.split** was quite useful for partitioning the dates.

```
getdates = all_x.index.levels[0]
```

```
train_dates, valid_dates, test_dates = np.split(dates, [int(len(dates) * train_size), int(len(dates) * (train_size + test_size))])
```

Other Stuff

Research Papers

Omar H.
Anant: the source most people point to is arxiv. https://arxiv.org/archive/q-fin is searchable.

Bryant M.
SSRN
https://papers.ssrn.com/sol3/topten/topTenResults.cfm?groupingtype=2&groupingId=203

Diff Frequency Time Series

https://stats.stackexchange.com/questions/63971/how-to-combine-time-series-based-features-with-different-frequencies

1. As suggested by @ChuckKillerDoll, you could find aggregate / derive features from your current measures, but chances are you will lose information by doing so. Another way to go about it, is to create three separate models and train a model for each of the frequency information matrices individually. These produce output scores S_1, S_2, S_3, you can then combine these in a new ensemble model. The easiest way to combine them is in a linear model: $S = a_1 S_1 + a_2 S_2 + a_3 S_3$

Here ***S*** is the final output score and you still need to learn the weights on a validation set. You could of course, plug the scores into more complicated models ... The main disadvantage of this technique is that you lose some of the covariance information.

2. I would try a state-space model. The simplest possible form would be: —>>

$$\begin{pmatrix}\theta_{1t}\\ \theta_{2t}\\ \theta_{3t}\end{pmatrix} = \begin{pmatrix}I & 0 & 0\\ 0 & I & 0\\ 0 & 0 & I\end{pmatrix}\begin{pmatrix}\theta_{1,t-1}\\ \theta_{2,t-1}\\ \theta_{3,t-1}\end{pmatrix} + \begin{pmatrix}\eta_{1t}\\ \eta_{2t}\\ \eta_{3t}\end{pmatrix}$$

$$\begin{pmatrix}Y_{1t}\\ Y_{2t}\\ Y_{3t}\end{pmatrix} = \begin{pmatrix}I & 0 & 0\\ 0 & I & 0\\ 0 & 0 & I\end{pmatrix}\begin{pmatrix}\theta_{1t}\\ \theta_{2t}\\ \theta_{3t}\end{pmatrix} + \begin{pmatrix}\varepsilon_{1t}\\ \varepsilon_{2t}\\ \varepsilon_{3t}\end{pmatrix}$$

where all the Y_{it}, $theta_{it}$ etc. are to be understood as vectors of the same dimensions of your three time series.

You can fit such model --a multivariate random walk plus noise-- even if not all the *Y*'s are observed at all times, and estimate the state vector at all possible *t*'s. This removes the problem of the different frequencies of the three time series.

If the situation warrants, you might also fit a different model. For instance, if there is some redundancy among the components of Y_{it}, you might want to use $theta_{it}$ with $\dim(\theta_{it}) < \dim(Y_{it})$ and fit what would be essentially a dynamic factor analysis model.

https://stats.stackexchange.com/questions/138957/regression-with-different-frequency
Mi(xed) da(ta) s(ampling) (MIDAS).
Midas regressions, popularized by Eric Ghysels, are a second option. There are two main ideas here. The first is frequency alignment. The second is to tackle the curse of dimensionality by specifying an appropriate polynomial. The unrestricted MIDAS model is the simplest from within the class of models and can be estimated by ordinary least squares. There is some MiDAS implementation in python too: github.com/mikemull/midaspy

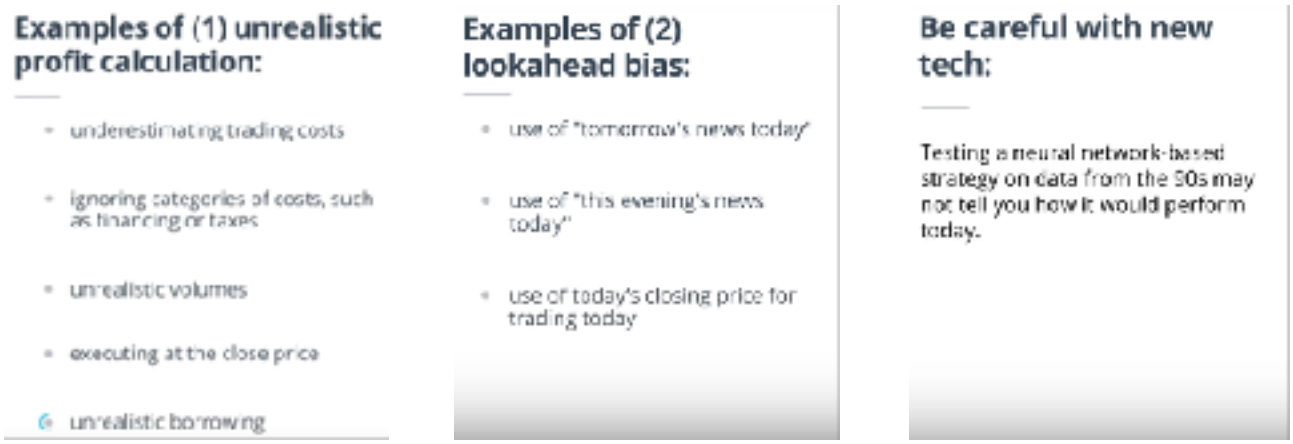
Kalman filter methods.
This is a state-space modelling approach, which involves treating the low-frequency data as containing NAs and filling them in using a Kalman filter. This is my personal preference, but it does have the difficulty of specifying the correct state-space model.

For a more in-depth look at the pros and cons of these methods, refer to State Space Models and MIDAS Regressions by Jennie Bai, Eric Ghysels and Jonathan H. Wright (2013).

MODULE 8: BACKTESTING

Lesson 25: Intro to Backtesting

Backtest Validity



Backtest Best Practices

1. Use cross-validation to achieve just the right amount of model complexity.
2. Always keep an out-of-sample test dataset. You should only look at the results of a test on this dataset once all model decisions have been made. If you let the results of this test influence decisions made about the model, you no longer have an estimate of generalization error.
3. Be wary of creating multiple model configurations. If the Sharpe ratio of a backtest is 2, but there are 10 model configurations, this is a kind of multiple comparison bias. This is different than repeatedly tweaking the parameters to get a sharpe ratio of 2.
4. Be careful about your choice of time period for validation and testing. Be sure that the test period is not special in any way.
5. Be careful about how often you touch the data. You should only use the test data once, when your validation process is finished and your model is fully built. Too many tweaks in response to tests on validation data are likely to cause the model to increasingly fit the validation data.
6. Keep track of the dates on which modifications to the model were made, so that you know the date on which a provable out-of-sample period commenced. If a model hasn't changed for 3 years, then the performance on the past 3 years is a measure of out-of-sample performance.

Traditional ML is about fitting a model until it works. Finance is different—you can't keep adjusting parameters to get a desired result. Maximizing the in-sample sharpe ratio is not good—it would probably make out of sample sharpe ratio worse. It's very important to follow good research practices.

Structural Changes

How does one split data into training, validation, and test sets so as to avoid bias induced by structural changes? It's not always better to use the most recent time period as the test set,

sometimes it’s better to have a random sample of years in the middle of your dataset. You want there to be nothing SPECIAL about the hold-out set. If the test set was the quant meltdown or financial crisis—those would be special validation sets. If you test on those time periods, you would be left with the unanswerable question: was it just bad luck? There is still some value in a strategy that would work every year except during a financial crisis.

Alphas tend to decay over time, so one can argue that using the past 3 or 4 years as a hold out set is a tough test set. Lots of things work less and less over time because knowledge spreads and new data are disseminated. Broader dissemination of data causes alpha decay. A strategy that performed well when tested on a hold-out set of the past few years would be slightly more impressive than one tested on a less recent time period.

Gradient Boosting

In our exercise about overfitting, we're going to use a type of model that we haven't yet encountered in the course, but that's popular and well-known, and has been used successfully in machine learning competitions: gradient boosted trees. Here we're going to give you a short introduction to gradient boosting so that you have an intuition for how the model works.

We've already studied ensembling; well, boosting is another type of ensembling, or combining weak learners into a strong learner. It's also typically done with decision trees as the weak learners. The video below will give you a quick introduction to boosting by telling you about the first successful boosting algorithm, Adaboost.

This video only scratched the surface, but you can see that Adaboost basically works in the following way:

- Fit an additive model (ensemble) in a forward stage-wise manner.
- In each stage, introduce a weak learner to compensate the shortcomings of existing weak learners.
- In Adaboost,“shortcomings” are identified by high-weight datapoints (this is what is meant in the video by making the points “bigger”).

Gradient boosting is very similar. In essence, it allows the user to customize the method used to identify the "shortcomings" of existing weak learners (the **cost function**). If you want to learn more about the details, try this page.

Lesson 26: Optimization with Transaction Costs

Barra Data

FACTORS¶
Here's a partial list of the barra factors in our dataset and their definitions. These are collected from documentation by Barra. There are style factors and industry factors. The industry factors will be used as risk factors. You can consider using the style factors as alpha factors. Any factors not used as alpha factors can be included in the risk factors category.

Style factors¶

- **beta**: Describes market risk that cannot be explained by the Country factor. The Beta factor is typically the most important style factor. We calculate Beta by time-series regression of stock excess returns against the market return.
- **1 day reversal**
- **dividend yield:** Describes differences in stock returns attributable to stock's historical and predicted dividend-to-price ratios.
- **downside risk** (maximum drawdown)
- **earnings quality:** Describes stock return differences due to the accrual components of earnings.
- **earnings yield:** Describes return differences based on a company's earnings relative to its price. Earnings Yield is considered by many investors to be a strong value signal. The most important descriptor in this factor is the analyst-predicted 12-month earnings-to-price ratio.
- **growth:** Differentiates stocks based on their prospects for sales or earnings growth. The most important descriptor in this factor is the analyst predicted long-term earnings growth. Other descriptors include sales and earnings growth over the previous five years.
- **leverage**: Describes return differences between high and low-leverage stocks. The descriptors within this style factor include market leverage, book leverage, and debt-to-assets ratio.
- **liquidity**: Describes return differences due to relative trading activity. The descriptors for this factor are based on the fraction of total shares outstanding that trade over a recent window.
- **long-term reversal**: Describes common variation in returns related to a long-term (five years ex. recent thirteen months) stock price behavior.
- **Mid capitalization**: Describes non-linearity in the payoff to the Size factor across the market-cap spectrum. This factor is based on a single raw descriptor: the cube of the Size exposure. However, because this raw descriptor is highly collinear with the Size factor, it is orthogonalized with respect to Size. This procedure does not affect the fit of the model, but does mitigate the confounding effects of collinearity, while preserving an intuitive meaning for the Size factor. As described by Menchero (2010), the Mid Capitalization factor roughly captures the risk of a "barbell portfolio" that is long mid-cap stocks and short small-cap and large-cap stocks.
- **Momentum** – Differentiates stocks based on their performance over the trailing 12 months. When computing Momentum exposures, we exclude the most recent returns in order to avoid the effects of short-term reversal. The Momentum factor is often the second strongest factor in the model, although sometimes it may surpass Beta in importance.
- **Profitability** – Combines profitability measures that characterize efficiency of a firm's operations and total activities.
- **Residual Volatility** – Measures the idiosyncratic volatility anomaly. It has three descriptors: (a) the volatility of daily excess returns, (b) the volatility of daily residual returns, and (c) the cumulative range of the stock over the last 12 months. Since these descriptors tend to be highly collinear with the Beta factor, the Residual Volatility factor is orthogonalized with respect to the Beta and Size factors.
- **seasonality**
- **sentiment**

- **Size** – Represents a strong source of equity return covariance, and captures return differences between large-cap and small-cap stocks. We measure Size by the log of market capitalization.
- **Short term reversal**
- **Value**
- **Prospect** -- is a function of skewness and maximum drawdown.
- **Management Quality** -- is a function of the following:
    - Asset Growth: Annual reported company assets are regressed against time over the past five fiscal years. The slope coefficient is then divided by the average annual assets to obtain the asset growth.
    - Issuance Growth Annual reported company number of shares outstanding regressed against time over the past five fiscal years. The slope coefficient is then divided by the average annual number of shares outstanding.
    - Capital Expenditure Growth: Annual reported company capital expenditures are regressed against time over the past five fiscal years. The slope coefficient is then divided by the average annual capital expenditures to obtain the capital expenditures growth.
    - Capital Expenditure: The most recent capital expenditures are scaled by the average of capital expenditures over the last five fiscal years.

Industry Factors¶

- aerospace and defense
- airlines
- aluminum and steel
- apparel
- Automotive
- banks
- beta (market)
- beverage and tobacco
- biotech & life science
- building products
- chemicals
- construction & engineering
- construction & machinery
- construction materials
- commercial equipment
- computer & electronics
- commercial services
- industrial conglomerates
- containers (forest, paper, & packaging)
- distributors
- diversified financials
- electrical equipment
- electrical utility
- food & household products & personal
- food & staples retailing
- gas & multi-utilities
- healthcare equipment and services
- health services
- home building
- household durables
- industry machinery
- non-life insurance
- leisure products
- leisure services
- life insurance
- managed healthcare
- multi-utilities
- oil & gas conversion
- oil & gas drilling
- oil & gas equipment
- oil and gas export
- paper
- pharmaceuticals
- precious metals
- personal products
- real estate
- restaurants
- road & rail
- semiconductors
- semiconductors equipment
- software
- telecommunications
- transportation
- wireless
- SPTY* and SPLTY* are various industries

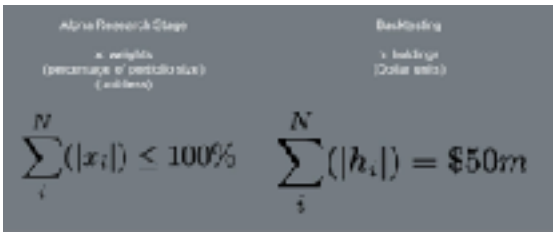
https://www.msci.com/factor-index

Backtest Considerations / Constraints

TIME OFFSETS
Use 2-day delay due to information and execution lag.

Holdings in dollars

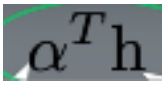
For the backtest, the positions for each asset will now be in units of dollars, instead of weights. To make this distinction, we'll use the letter "h" for holdings, instead of the "x" that we had used for weights in prior projects and lessons.



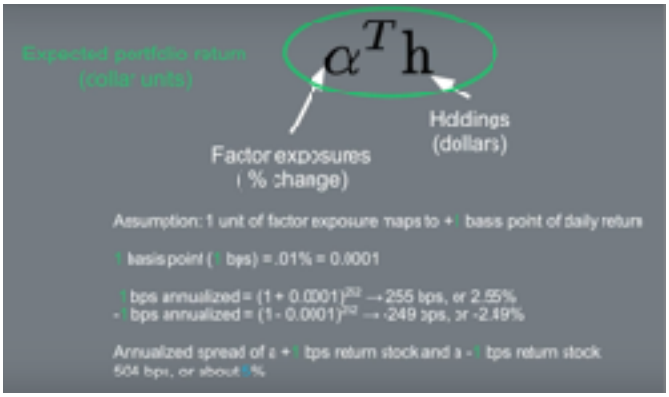
When backtesting, we need to use a specific portfolio size (in dollars), because the size affects trading costs, as position sizes become larger and more difficult to trade. So strategies that work on small portfolios may not work on larger ones.

SCALING ALPHA FACTOR

Expected portfolio return — the dot product of the transpose of the alpha vector and the holding vector — is expressed in dollar units:



The factor exposures are denoted as % change. Barra uses a -5 to +5 range, mean 0 (transformed to be close to normally distributed). The sizing maps 1 unit of factor exposure to 1 basis point. So, to illustrate, a +/-1bps return annualized is equivalent to +/-2.5%, or c.5% range. This means a stock with a factor exposure of 1 bp vs another with -1 bp would have a expected return difference of 5%.



To scale the alpha vector so that the expected portfolio return is in units of dollars, multiply the alpha vector by 1 basis point (1/10,000).

TRANSACTION COSTS

The midpoint price can serve as the benchmark from which we estimate the transaction cost. For each child order that is traded, the difference between the price paid and this benchmark is a measure of the trade cost, which is called "**slippage**." We can think of slippage as the sum of two kinds of price impact. There's the temporary price impact, and permanent price impact.

Temporary impact refers to the spread paid above the midpoint. Permanent impact refers to the price impact of trading in the stock, ie: moving the bid-ask spread prices.

Transaction Cost Formula

Use a 'linear impact' model.

The assumption that we'll use is that when the trade is **1% of Average Daily Volume, the price changes by 10 basis point**. The transaction cost is the percent change in price times the amount traded.

There is also evidence to support a square root estimation, where the impact diminishes with the size of the trade (ie: impact does not keep inceasing linearly).

Linear Impact Model

Definitions:

'Trade' is the change in dollar position size:

'Trade size' is trade divided by Avg $ volume:

'Impact' is the % change in price after trading

$$Trade_{i,t} = h_{i,t} - h_{i,t-1}$$

$$TradeSize_{i,t} = \frac{Trade_{i,t}}{ADV_{i,t}}$$

Price Impact

So impact on a unit basis is:

$$\frac{\%\Delta price_{i,t}}{TradeSize_{i,t}} = \frac{10bps}{1\%}$$

Which can be simplified as:

$$\%\Delta price_{i,t} = \frac{h_{i,t} - h_{i,t-1}}{10 \times ADV_{i,t}}$$

For convenience, we collect the term not related to the holdings and express as lambda:

$$\lambda_{i,t} = \frac{1}{10 \times ADV_{i,t}}$$

So the % change in price can be estimated as:

$$\%\Delta price_{i,t} = \lambda(h_{i,t} - h_{i,t-1})$$

Cost Impact

Cost is the % change in price X amount traded:

$$tcost_{i,t} = \%\Delta price_{i,t} \times (h_{i,t} - h_{i,t-1})$$

And since the % price change has the same trade term, the cost formula can be simplified as:

$$tcost_{i,t} = \lambda_{i,t} \times (h_{i,t} - h_{i,t-1})^2$$

And the cost impact across all **portfolio trades** would simply be:

Linear Impact Model
Portfolio of stocks (sum across all stocks)

$$tcost_t = \sum_{i}^{N} \lambda_{i,t}(h_{i,t} - h_{i,t-1})^2$$

Optimization Without Constraints

Best to avoid or limit where possible as can cause issues such as:

- Slowing down optimization
- Optimizer may not support constraints (non-linear?)

Alternative to constraints can be achieved by putting them directly into the objective function.

USING THE OBJECTIVE FUNCTION INSTEAD OF CONSTRAINTS
Assume an optimization objective function has the following terms to minimize:

Objective Function = common Risk + Specific Risk - Expected Return + Transaction Costs

We have some goals that we previously satisfied using constraints:
- Market neutral
- Position Size
- Portfolio Diversification

Can you think about which term in the objective function also serves a similar purpose as these constraints?

How objective function can 'constrain' the portfolio
Definition recall:
- Common risk = portfolio risk explained by the factors
- Specific risk = risk due to individual assets in the portfolio
- Market neutral = sum of longs equals sum of shorts

Market Neutral
When the optimizer minimizes the common risk, it is also adjusting the differences between longs and shorts, thereby effectively imposing a market neutral constraint.

Position Size (should be portfolio size??)
The risk aversion parameter of the objective function seeks to reduce the overall risk exposure (common and specific) which has a similar impact to imposing a leverage constraint.

Portfolio Diversification (max holding size)
Having the optimizer limit the specific (or idiosyncratic risk) serves to limit the single asset holding size and therefore increase diversification.

Risk Factor Matrix

Its a good idea to always be suspicious about covariance's and correlations in general and to ask whether the covariance or correlations are giving more insight or just adding more noise. It can sometimes be worthwhile to reduce(shrink) the covariance values in a cov matrix while leaving the diagonal variances unchanged, or even zeroing out the covariances altogether.

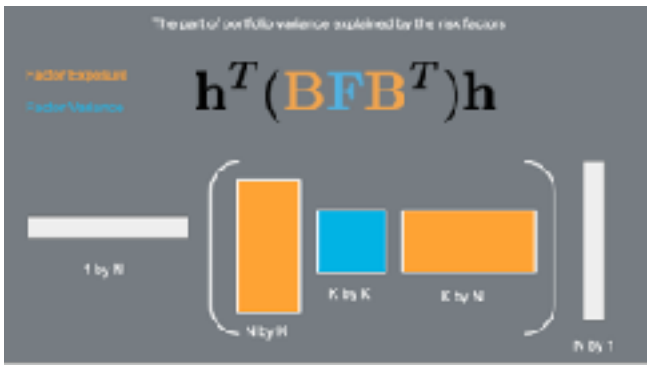
This is true because the portfolio optimizer is trying to reduce the risk exposures (variance) to factor 1 and factor 2 (etc.) so it become less important how those two risk factors co-vary together.

The benefits of throwing away the covariance information is that it makes the model more simple, and the diagonal matrix makes matrix operations more computationally efficient. Backtesting is computationally expensive and time consuming, which gets in the way of iterating and improving.

AVOID N BY N MATRIX
This is the formula for factor model of portfolio variance:

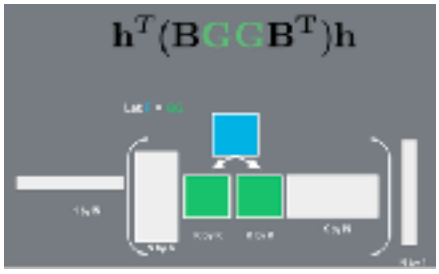
And this focuses on the risk factors:

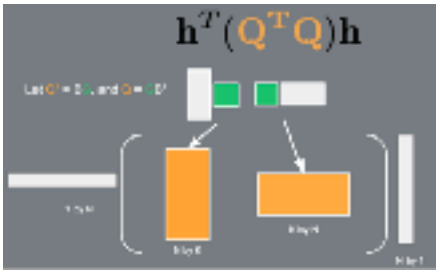
F is the *factor variance matrix*, wrapped on both sides by the *factor exposures matrices* **B** and **B^T,** which is then wrapped again on both sides by the *portfolio holdings vectors*.

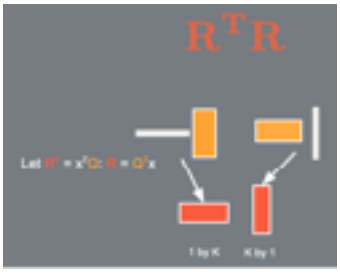


When multiplied out, you get a very large square matrix, N x N, where **n is the number of assets**. If you have 2000 stocks, it could easily take hours to run optimization or end up with memory overflow.

Steps to avoid N x N matrix







Risk Aversion Parameter (*K*)

Is set to target a particular size of portfolio. Size could mean gross market value, value at risk, volatiltiy, leverage, or any number of things. Increasing(decreasing) the risk aversion parameter decreases(increases) the size of the portfolio. So the the risk aversion parameter allows targeting of portfolio size.

The GMV is the sum of the absolute value of both the long and shorts, also called the LO1 norm.

$$GMV = \sum_{i}^{N}(|h_i|)$$

Objective Function, Gradient and Optimizer

When doing optimization, you need usually write the objective function and then its gradient, and then input both of these into an optimizer of your choice. Its good practice to be able to write the objective function and gradient in code even though there are libraries out there for it.

OPTIMIZATION OVERVIEW IN SCIPY

This **Scipy link "Mathematical optimization"** is a good reference on the various optimizers that are available in scipy. In addition to the L-BFGS method (which is designed to efficiently handle large scale problems), other optimizers worth trying are Powell, Nelder-Mead, and Conjugate Gradient optimizers.

This **course website** also contains insightful explanation of various optimization algorithms.

Exercise: optimization_w_tcosts

FILES
covariance 2003-2008 = covariance
pandas-frames 2003-2008 = data (12,252 stocks X 92 cols [mostly factors])
price 2003-2008 = daily_returns

DICTIONARIES
Open year 2003 files (and 2004 for price) into dictionaries of their daily dataframes:
- data { pandas-frames 2003} covariance { 2003} daily_return { 2003-04}
  - Key = date; Value = df

MERGE DATA AND DLYRETURN INTO '`frames`' DICT
- First add dlyreturn date column to daily_return df
- Shift dlyreturn date + 2 days when merging to account for transaction/trading lag

CALCULATE FACTOR RETURNS
- **Recall:**
  - a factor's factor return times its factor exposure gives the part of a stock's return that is explained by that factor
  - Barra data contains the factor exposure of each factor.
  - Use regression to estimate the factor returns of each factor, on each day
  - This is a **cross-sectional regression**, because it's a cross section of stocks, for a single time period.
    - **Beta** = factor exposure; independant var
    - **r** = stock return; dependent var
    - **f** = factor return; dependent co-efficient to estimate

$$r_{i,t} = \sum_{j=1}^{k} (\beta_{i,j,t-2} \times f_{j,t})$$
where $i = 1 \ldots N$ (N assets),
and $j = 1 \ldots k$ (k factors).

- **Set universe MCAP > $1bn**
  - Universe df named '`estu`' : 1756 stocks
- **Deal with Outliers:** Winsorize
- **Get list of USFASTD factor names:** `list(filter(lambda x: "USFASTD_" in x, n))`
- **Run ols to calculate factor returns:**
  - **get_formula( )** func arranges USFASTD factor names into a formula string for multiple ols regression
  - `DlyReturn` is the dependent variable, whereas the `USFAST...` columns are the independent variables.
  - Return **results.params**

ESTIMATE FACTOR RETURNS OF ALPHA FACTORS
- Choose alpha factors
  - `alpha_factors = ["USFASTD_1DREVRSL", "USFASTD_EARNYILD", "USFASTD_VALUE", "USFASTD_SENTMT"]`
- Loop through the date df's in `frames` dict, calling `estimate_factor_returns` and populating a new dict, `facret`

COMBINE FACTOR RETURNS INTO DF
- For each date, and for each **factor return**, get the value (df) from the dictionary and put it into the dataframe:

```
for dt in my_dates:
    for alp in alpha_factors:
        facret_df.at[dt, alp] = facret[dt.strftime('%Y%m%d')][alp]
```

PORTFOLIO OPTIMIZATION FOR A SINGLE PERIOD

The optimization will want to know about the **prior trading day's portfolio holdings, also called holdings.** The previous day's holdings will be used **to estimate the size of the trades due to position changes**, which in turn **helps us estimate transaction costs**. We'll start with an **initial holding of zero for a single stock**.

The reason we'll use a single stock is that the estimation universe chosen on each day will include all stocks that have holdings on the previous day. So we want to keep this list small when we first start out, else we'll keep many stocks that may fall below the 1 billion market cap threshold, just because they were chosen in the initialization of the backtest.

We'll want to choose a stock that is likely to satisfy the 1 billion market cap threshold on any day. So let's choose the stock with the largest market cap.

- This whole process seems way to convoluted, maybe project will reveal why???

Shows largest MCAP:

```
estu.sort_values('IssuerMarketCap',ascending=False)[['Barrid','IssuerMarketCap']].head()
```

Make prev holdings df with the largest MCAP stock:

```
previous_holdings = pd.DataFrame(data = {"Barrid" : estu.sort_values('IssuerMarketCap',
                                    ascending=False)[['Barrid']].iloc[0][0],
                                    "x.opt.previous" : pd.Series(0)})
```

Get a single day's data to be used for portfolio optimization:

```
dt = my_dates[0]  # grab a date
date = dt.str
df = frames[date]
```

Add previous holdings to the single day df

```
df = df.merge(previous_holdings, how = 'left', on = 'Barrid')
```

Clean NaN's all to zeros in x.opt.previous column

Clean specific risk

Barra calculates specific risk for each asset. If the value in the data is zero, this may be due to missing data rather than the specific risk actually being zero. So we'll set zero values to the **median**, to make sure our model is more realistic.

```
# This is a very concise bool condition and value-setting syntax.....(familiarize)
df.loc[df['SpecRisk'] == 0]['SpecRisk'] = median(df['SpecRisk'])
```

Modify MCAP > $1bn filter to include previous day MCAP levels

```
universe = df.loc[(df['IssuerMarketCap'] >= 1e9) | (df['x.opt.previous'] !=
0)].copy()
# solution code seems more clunky: (abs(df['x.opt.previous']) > 0)
```

Remove daily returns from universe df to ensure they can't be looked at

```
universe = universe.drop(columns = 'DlyReturn')
```

Get risk factors

- These are all the remaining factors that were not chosen as alpha factors

Save initial holdings (pre-optimization) as an array

```
h0 = np.asarray( universe['x.opt.previous'] )
h = h0.copy( )
```

MATRIX OF RISK FACTOR EXPOSURES B

- Extract risk factor columns into a risk factor matrix
  - Can use patsy.dmatrices
    - Uses formula string like ols: `SpecRisk ~ 0 + USFASTD_AERODEF + USFASTD_AIRLINES + ...`
    - Then drops `SpecRisk 'outcome'` as not needed, only keep `predictors`
  - Or just slice on risk columns and assign as np.array??? (seems much easier….)
  - Assign risk factor matrix to **`B`** and its transpose to **`BT`**

FACTOR COVARIANCE MATRIX F

Notice that we'll run into some issues where the covariance data doesn't exist. One important point to remember is that we need to order the factors in the covariance matrix F so that they match up with the order of the factors in the factor exposures matrix B (col names). Note that covariance data is in percentage units squared, so to use decimals, so we'll rescale it to convert it to decimal.

- Just get the diagonal, rest is not important….

```
## extract a diagonal element from the factor covariance matrix
def get_cov_version1(cv, factor1, factor2):
       return(cv.loc[(cv.Factor1==factor1) & (cv.Factor2==factor2),"VarCovar"].iloc[0])

def diagonal_factor_cov_version1(date, B):
    """
    Notice that we'll use the order of column names of the factor exposure matrix
    to set the order of factors in the factor covariance matrix
    """
    cv = covariance[date]
    k = np.shape(B)[1] # no. of columns; B.shape[1] if using df
    Fm = np.zeros([k,k])
    for i in range(0,k):
        for j in range(0,k):
            fac1 = B.design_info.column_names[i]  #B.columns.tolist() if using df…
            fac2 = B.design_info.column_names[j]  #B.columns.tolist() if using df…
            # Convert from percentage units squared to decimal
            Fm[i,j] = (0.01**2) * get_cov_version1(cv, fac1, fac2)
    return(Fm)
```

** be sure to use the `DataDate` to get the factor covariance matrix:

```
date = str(int(universe['DataDate'][1]))
F = diagonal_factor_cov_version1(date, B)
```

SCALE FACTOR EXPOSURES

Note that the terms that we're calculating for the objective function will be in dollar units. So the expected return $-a^T h$ will be in dollar units. The *h* vector of portfolio holdings will be in dollar units. We'll make an assumption that a factor exposure of 1 maps to 1 basis point of daily return on that stock. So we'll convert the factor values into decimals, rescaling the alpha factors by dividing by 10,000.

- Sum across the rows of the alpha factor returns to get the day's return
- Scale that sum by 1/10,000 so that the $a^T \mathbf{h}$ expression is in dollar units

CALCULATING COMMON RISK

Recall that the common risk term looks like this: $\mathbf{h}^T \mathbf{B} \mathbf{F} \mathbf{B}^T \mathbf{h}$

Where h is the vector of portfolio holdings, B is the factor exposure matrix, and F is the factor covariance matrix. This calculation forms an N by N matrix, which is computationally expensive, and may lead to memory overflow for large values of N, **so should be avoided.**

square root of a matrix.

We can find a matrix $\mathbf{B}$ that's the matrix square root of another matrix $\mathbf{A}$, which means that if we matrix multiply $\mathbf{BB}$, we'd get back to the original matrix $\mathbf{A}$.

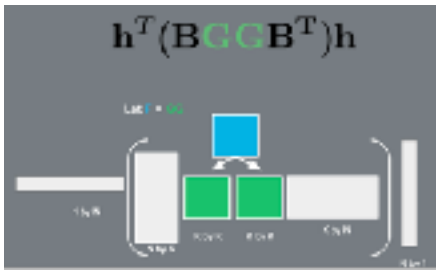
Find $\mathbf{Q}$ such that $\mathbf{Q}^T\mathbf{Q}$ is the same as $\mathbf{BFB}^T$. Let's let $\mathbf{G}$ denote the square root of matrix $\mathbf{F}$, so that $\mathbf{GG} = \mathbf{F}$.

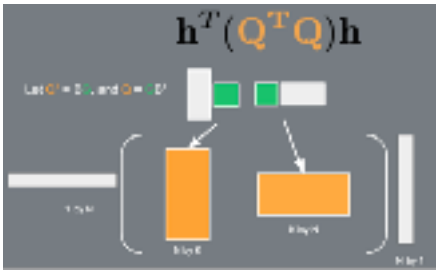
Then the expression for the covariance matrix of assets, $\mathbf{BFB}^T$, can be written as $\mathbf{BGGB}^T$.

Let's let $\mathbf{Q} = \mathbf{GB}^T$ and let $\mathbf{Q}^T = \mathbf{BG}$, which means we can rewrite $\mathbf{BGGB}^T = \mathbf{Q}^T\mathbf{Q}$, and the common risk term is $\mathbf{h}^T\mathbf{Q}^T\mathbf{Qh}$

Also, note that we don't have to calculate $\mathbf{BFB}^T$ explicitly, because the actual value we wish to calculate in the objective function will apply the holdings $\mathbf{h}$ to the covariance matrix of assets.

Steps to avoid N x N matrix





- Get matrix square root of F, name it G
  ```
  G = scipy.linalg.sqrtm(F)
  ```
- Calculate Q (should be short and wide) and Q.T (should be tall and narrow)
  ```
  Q = np.matmul(G, BT)
  QT = np.matmul(B, G)   # or Q.transpose()
  ```

So the original formula of $h^T BFB^T h$ became
$h^T BGGB^T h$, where $GG = F$.

And then, if we let $Q^T = BG$ and $Q = GB^T$:
$h^T Q^T Qh$

Let $R = Qh$ and $R^T = h^T Q^T$:

The risk term becomes:
$R^T R$, where $R^T = h^T Q$ and $R = Q^T h$

So an important point here is that we don't want to multiply $Q^T Q$ itself, because this creates the large N by N matrix. We want to multiply $h^T Q^T$ and Qh separately, creating vectors of length k (k is number of risk factors).

- Calc R = Qh
  ```
  R = np.matmul(Q, h)
  ```
- Calc R.T = Q'h'
  ```
  RT = np.matmul(h.T,QT)  # or R.transpose()??
  ```

Put it all together: calculate common risk term efficiently

Notice how we avoided creating a full N by N matrix

Also, notice that if we have Q, we can take its transpose to get Q^T instead of doing the matrix multiplication.

Similarly, if we have R, which is a vector, we notice that $R^T R$ is the same as taking the dot product. In other words, it's squaring each element in the vector R, and adding up all the squared values.

$R^T R = \sum_i^k (r_i^2)$

```
G = scipy.linalg.sqrtm(F)
Q = np.matmul(G, BT)
R = np.matmul(Q, h)
common_risk = np.sum( R ** 2)
```

SPECIFIC RISK TERM

The portfolio's variance that is specific to each asset is found by combining the holdings with the specific variance matrix:

$h^T S h$, where h^T is a 1 by N vector, S is an N by N matrix, and h is an N by 1 vector.

Recall that S is a diagonal matrix, so all the off-diagonals are zero. So instead of doing the matrix multiplication, we could save computation by working with the vector containing the diagonal values.

$h^T S h = \sum_i^N (h_i^2 \times S_i)$ because S is a diagonal matrix.

```
## TODO: specific variance : rescale it and then square to get specific variance
specVar = (0.01 * universe['SpecRisk']) ** 2
# TODO: specific risk term (include holdings)
spec_risk_term = np.dot(h**2, specVar)
```

MAXIMIZE PORTFOLIO RETURNS

Since the alpha vector α is supposed to be indicative of future asset returns, when we look at a portfolio of assets, the weighted sum of these alphas $\alpha^T \mathbf{h}$ is predictive of the portfolio's future returns. We want to maximize the portfolio's expected future returns, so we want to minimize the negative of portfolio's expected returns $-\alpha^T \mathbf{h}$

Lesson 27: Attribution

Attribution Reporting

Let's take this opportunity to look at an example attribution report in a fund's documentation. In this first example, we can see the risk of a portfolio decomposed using a fundamental risk model. Fundamental factors are factors based on common sources of risk. Their meaning remains the same over time, even if the factor exposures are updated daily. The total predicted active risk of 3.63% annual volatility can be split into a specific/idiosyncratic component, which accounts for 39% of variance (as calculated using a variance decomposition), and a factor component, which can be further split into the contributions of 3 fundamental factors.

	Pred Risk (% Ann)	% of Variance
Total Active Risk	3.63%	100%
Specific Active Risk	2.25%	39%
Factor Active Risk (Fund)	2.84%	61%
Style (Fund)	1.95%	34%
Industry (Fund)	1.77%	29%
Market (Fund)	0.25%	-1%

You can do the same sort of attribution with statistical risk factors, but the individual risk factors are hard to interpret. In the example below, most of the risk is attributed to Statistical Factors 2, 1 and 6, however this does not immediately provide a lot of insight. Additional analysis would seek to understand whether other, interpretable factors are similar to Statistical Factor 6.

	Factor Exposure	Pred Risk (% Ann)	% of Variance
Total Active Risk		4.42%	100%
Specific Active Risk		2.48%	31%
Factor Active Risk		3.65%	69%
Statistical Factor 2	0.0191%	2.30%	27.0%
Statistical Factor 1	-0.0132%	1.55%	12.3%
Statistical Factor 6	-0.0128%	1.42%	10.3%
Statistical Factor 10	0.0096%	1.08%	5.91%
Statistical Factor 8	-0.0091%	1.07%	5.91%
Statistical Factor 13	0.0070%	0.80%	3.30%

Understanding Portfolio Characteristics

There are a few other things we can calculate that help us understand our portfolio's performance. For each time period:

- GMV (gross market value) is the sum of the absolute value of your holdings, long and short. This is a good gauge of the overall size of your portfolio. GMV = $\sum_i |h_i|$.
- Net holdings tell you the relative balance of long to short positions. Net holdings = $\sum_i h_i$.
- You can also calculate the total long and short holdings. Total long = $\sum_{h_i>0} h_i$. total short = $\sum_{h_i<0} h_i$.
- Total dollars traded tells you the approximate value of the trades you made. It can help you get a sense of how much trading you're doing, and the value of your trades relative to your holdings. Dollars traded = $\sum_i |h_{i,t} - h_{i,t-1}|$.
