FoLT Turtorial 5 Summary

PART 1: Decision Tree Classifier

Part a: Features

- To train a classifier we first need to decide on the relevant features we want to use and how to extract those features.
- `gender_features` is a function that takes a name and extract its `features` and return them in a dict
- Code:

```
def gender_features(name):
    vowels = "aeiouy"
    number_vowels = len([letter for letter in name.lower() if letter in vowels])
    return {
        'first_letter': name[0],
        'last_letter': name[-1],
        'number_vowels': number_vowels,
        'length' : len(name),
        'trigram_start' : name[0:3],
        'trigram_end' : name[-3:]
    }
```

- How many Features?
  - Too many = increase classifier ability to capture more complex connections and patterns **but** lead to overfitting, the model becomes too specific to the training data and performs poorly on new, unseen data.
  - Too few = classifier simpler and therefore more understandable **but** can lead to underfitting, the model is too simple and fails to capture important patterns and relationships in the data.

Part b: Train/Test Split

- We are gonna make a dataset to train and test the classifier

1. `Train Dataset`: How?
   - we have to store all (name, gender) pairs of the name corpus
   - We can access all male names of the corpus with `names.words('male.txt')`
   - Lowercase all names of the corpus
   - store all (name, gender) pairs (male and female) in a variable `gender_names`
2. `Suffling`:
   - We should shuffle the dataset consisting of (name, gender) pairs before splitting them into train and test sets
   - Otherwise, the classifier might be biased because:
     - the trainset will mainly contain male names
     - **while** the test set only contains female names.
3. `Splitting The dataset`:
   - Create a list of tuples `feature_set` which pairs the feature dictionary of a name (use the function `gender_features`) with the assigned gender of the name.
   - 80% of the feature set should be stored in the variable train_set
   - The remaining 20% will be used as a test_set
   Code:

```
#Step 1
male_names = [(name.lower(), 'male') for name in names.words('male.txt')]
female_names = [(name.lower(), 'female') for name in names.words('female.txt')]
gender_names = male_names + female_names
#Step 2
random.seed(0)
random.shuffle(gender_names)
#Step 3
dataset_length = len(feature_set)
train_set = feature_set[:int(0.8 *dataset_length)]
test_set = feature_set[int(0.8 * dataset_length):]
```

Part c: Model Training

- Train the `classifier nltk.DecisionTreeClassifier` with the training set
- Code:

```
classifier = nltk.DecisionTreeClassifier.train(train_set)
```

- **Important Question**: If a classifier always `assigns` the `most common class`, we consider it as the `most simple classifier`. Now, recap how the accuracy is calculated and what accuracy the simplest classifier can achieve. The `simplest classifier in binary decision problems` is often called the `"majority class baseline"`
- What is the accuracy of the majority class baseline in our gender classification task?

```
female_name_dist: 0.6295317220543807
male_name_dist: 0.3704682779456193
```

- **Answer**: Looking at the `distribution of female and male names` in the dataset, we find that about `63%` of all names in the corpus are `female`, which is the `majority class`. A `majority class baseline classifier` would always predict "female" for any sample. Thus, the `accuracy` of such a classifier would be `63%` because it `would predict correctly 63% of the time`

Part d: Evaluation

- We want to evaluate our classifier now. For this we want to assess the classifier on classifying female names by using the metrics `Precision`, `Recall` and the `F1-Score`, so we need to:
  - save the list of correct labels (gender) for each name into the variable `gold_labels`
  - save the predictions of the classifier in the variable `predictions` by using `classifier.classify(features)`
- Code:

```
gold_labels = [gender for features, gender in test_set]
predictions = [classifier.classify(features) for features, gender in test_set]
```

- Now we need to calculate the `true_positives`, `false_positives` and `false_negatives` for calculating the `Precision` and `Recall` later
- Code:

```
true_positives = sum(pred == 'female' and gold == 'female' for pred, gold in
zip(predictions, gold_labels))
false_positives = sum(pred == 'male' and gold == 'female' for pred, gold in
zip(predictions, gold_labels))
false_negatives = sum(pred == 'female' and gold == 'male' for pred, gold in
zip(predictions, gold_labels))
```

- Calculate `precision`, `recall` and `F1-Score`
- Code(**With Formulas**):

```
precision = true_positives / (true_positives + false_positives)
recall = true_positives / (true_positives + false_negatives)
f1_score = 2 * (precision * recall) / (precision + recall)
print("precision: ", precision)
print("recall: ", recall)
print("f1_score: ", f1_score)
```

- Result:

```
precision: 0.8234106962663976
recall: 0.7576601671309192
f1_score: 0.7891682785299808
```

- **k-fold cross-validation** is a technique in machine learning to assess the accuracy of a model:
  - It involves dividing the data into 'k' subsets. The model is trained on k-1 subsets and tested on the remaining subset.
  - This process is repeated k times, each time with a different subset as the test set.
  - The average of the k results is taken to estimate the model's performance.
  - This method helps to use all data for both training and testing, ensuring a more reliable performance estimate.
- **Important Question**: How does k-fold cross validation help us in assessing the `performance of a machine learning model`, and what are its `benefits` for checking how the `model performs on different parts of the data`?
- **Answer**: Benefits->
  - Better use of scarce data.
  - An overview of how the performance varies across different training sets gives an overall better evaluation of the model.
  - Additionally, the models consistency can be examined more precisely to avoid overfitting or underfitting in the model.

Part 2: Neural Networks

Part a: Neural Network Classifier

- Comparison:

```
The accuracy of the decision tree classifier : 0.723725613593455
The accuracy of the Neural Network classifier : 0.8074260541220893
```

- After using a NN on the same dataset we came with this conclusion:
  - The neural network has a better accuracy than the decision tree classifier.
  - Neural Networks are better at using the features and capturing more complex relationships in the data and yield therefore higher results.
  - In general they also tend to generalize better.
  - As a drawback, Neural Networks need a lot of data and are more complex in setting up and training.

Part b: Embeddings

- **QnA Part 1**
  1. Word embeddings are not able to capture the syntactical features of words.
     - False) Word embeddings capture syntactical and morphological features.
  2. Morphologically similar words are located nearby in the vector space.
     - True) Morphology is captured in embedding spaces to some degree.

3. Word embeddings consist of word vectors.
    - False) Word vectors and word embeddigs are synonyms.
4. Semantically similar words are located nearby in the vector space.
    - Semantics are captured in embedding spaces.
- **QnA Part 2**
  1. Predict the context of a word using self-supervision.
      - True) Self-supervision means that the supervision is induced from the textual data itself, without the need for manual annotation.
  2. Train on a manually annotated dataset.
      - False)
  3. Word embeddings are 1-hot vectors.
      - False)
  4. There is no need for training.
      - False)

Part c: Word2Vec

- What is `Word2Vec` :
  - word2vec is a model that learns word embeddings from a large corpus of text
  - It is a shallow, two-layer neural network that processes text
  - Its input is a text corpus and its output is a set of vectors: feature vectors for words in that corpus
  - The model consists of two layers: an input layer and an output layer
    - The input layer is a list of unique words in the corpus
    - The output layer is a set of vectors (one vector for each word in the input layer)
      - The vectors in the output layer are the embeddings we are looking for

1. **Load word2vec model**

- We are gonna use `en_core_web_md`: The model contains 300-dimensional vectors for 685k unique words and phrases
- Explaining the functions:
  - `token.has_vector`: True if the token has a vector representation
  - `token.vector_norm`: The L2 norm of the token's vector
  - `token.is_oov`: True if the token is out-of-vocabulary
- Code:

```
nlp = spacy.load('en_core_web_md')
doc = nlp("This is a sentence we will use to test the model for FoLT.")
for token in doc:
    print(f"Token: {token.text}, has vector: {token.has_vector}, vector norm: {token.vector_norm}, and shape: {token.vector.shape}, is OOV: {token.is_oov}")
```

2. **Identify similar words**
   - We are gonna use `similarity()` that gives us a value between 0 and 1, that reflects how similar the two tokens are

- Code + Example:

```
doc = nlp("Like football and basketball")
for token1 in doc:
    for token2 in doc:
        if (token1 != token2):
            print(token1.text, token2.text, token1.similarity(token2))
```

✓ 0.0s

```
Like football -0.06370789557695389
Like and -0.09568611532449722
Like basketball 0.04521557688713074
football Like -0.06370789557695389
football and 0.18923026323318481
football basketball 0.8091733455657959
and Like -0.09568611532449722
and football 0.18923026323318481
and basketball 0.12495683878660202
basketball Like 0.04521557688713074
basketball football 0.8091733455657959
basketball and 0.12495683878660202
```

3. **Opposites**
   - Opposites doesn't mean low similairty score, because antonyms usually occur in a similar context
   - Code + Example:

```
opposites = [("good", "bad"), ("happy", "sad"), ("love", "hate")]
for word1, word2 in opposites:
    print(word1, word2, nlp(word1).similarity(nlp(word2)))
```

✓ 0.0s

```
good bad 0.739188906927192
happy sad 0.5034751138271376
love hate 0.5708349903422567
```