Model Improvement

In this lecture…

- We will learn how to improve the classification model by
    - Pruning features
    - Rebalancing the dataset
    - Using alternative algorithms
    - Performing a search over the algorithms' hyperparameters

Getting the importance of a feature

- Important to understand HOW a model works
- For some models (e.g., decision trees) this is straightforward
- For others (e.g., Support Vector Machines or Neural Network) hard if not impossible

Feature importance for Naive Bayes

- Given clf our fitted model, we can use the following code:

```python
#Getting feature importance
neg_class_prob_sorted = clf.feature_log_prob_[0,: ].argsort()[::-1]
pos_class_prob_sorted = clf.feature_log_prob_[1,: ].argsort()[::-1]
print("Ham top 20 features:",np.take(count_vect.get_feature_names_out(), neg_class_prob_sorted[: 20]))
print("Spam top 20 features:",np.take(count_vect.get_feature_names_out(), pos_class_prob_sorted[: 20]))
```

Discussion

- The method `clf.feature_log_prob_` Computes the likelihood of features given a class, i.e., $P(x_i | y)$
- Classes “0” and “1” for us correspond to “spam” and “ham”
- `argsort()` returns the indexes of the sorted elements without sorting them. For example if you do numpy.argsort([3,4,2]) you will get [2,0,1]
- `[::-1]` reverts the list
- `NumPy.take()` extracts from the first list the elements specified as indexes in the second list
  - The first list contains the list of words in the vocabulary, obtained by `count_vect.get_feature_names_out()`
  - The second list contains the indexes for the top 20 probabilities, obtained as `neg_class_prob_sorted[: 20]`

Output

```
Ham top 20 features: ['get' 'come' 'got' 'call' 'know' 'like'
'time' 'good' 'dai' 'go' ‘love' 'lor' 'need' 'sorri' 'want'
'home' 'you' 'on' 'later' 'still']

Spam top 20 features: ['call' 'free' 'txt' 'mobil' 'text' 'stop'
'claim' 'www' 'prize' ‘repli'  'now' 'min' 'award' 'win'
'cash' 'servic' 'new' 'urgent' 'get' 'tone']
```

Feature selection

- Aims at selecting a subset of features to be used by the classifier
- Often (but not always) it helps to boost the classifier’s performances
- In scikit-learn we will use algorithms available in the feature_selection module
- Note: some machine learning algorithms (e.g., decision trees) already do feature selection themselves

Feature selection in scikit-learn

Some examples of feature selection techniques:

- Univariate feature selection: select features that better correlate with the dependent variable and do not correlate with each other
- Removing features with low variance
- Recursive feature elimination: once the model is obtained, its coefficient provide the importance of each feature
  - First, the model is trained with all features
  - Then, features are recursively removed until the desired number of features is reached

Which feature to choose?

↓

Y	X1	X2	X3
spam	0.1	10	5
ham	0.11	2	1
spam	0.11	11	6

Removing features with low variance

- If a feature does not have much variation in the dataset, then it might not very useful to discriminate one datapoint from the other
- Therefore, we might rather remove it

Example: univariate feature selection

- We use the SelectKBest feature selection algorithm with the χ^2 test to determine which feature differs more among the categories of the dependent variable

- As you know the χ^2 test determines whether the proportions of non-negative values for a feature differ among documents belonging to different categories (ham and spam in our example)

- For numerical feature you may try f_classif instead of chi2 (it does the ANOVA test) which compares two or more distributions and tells whether they significantly differ

Python Implementation

<code to create the tf-idf training set goes here>

```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2

#selecting the best 2000 features
selector = SelectKBest(chi2, k=2000)
X_new=selector.fit_transform(X_train_tfidf, y_train)


clf = MultinomialNB()
clf.fit(X_new, y_train)
#Performing the prediction

#indexing the test set
X_new_counts = count_vect.transform(X_test)
X_new_tfidf = tfidf_transformer.transform(X_new_counts)
# folds the test set into the selected features
# (i.e., it removes unused features)
X_new_sel=selector.transform(X_new_tfidf)
#performing the actual prediction
predicted = clf.predict(X_new_sel)

print(predicted)
print(np.mean(predicted==y_test))
```

Select the top x% features

```python
from sklearn.feature_selection import SelectPercentile
from sklearn.feature_selection import chi2

selector = SelectPercentile(chi2,percentile=40)
X_new=selector.fit_transform(X_train_tfidf, y_train)


clf = MultinomialNB()
clf.fit(X_new, y_train)


#Performing the prediction
#indexing the test set
X_new_counts = count_vect.transform(X_test)
X_new_tfidf = tfidf_transformer.transform(X_new_counts)
# folds the test set into the selected features
# (i.e., it removes unused features)
X_new_sel=selector.transform(X_new_tfidf)
#performing the actual prediction
predicted = clf.predict(X_new_sel)

print(predicted)
print(np.mean(predicted==y_test))
```

More…

- Data analytics course

Rebalancing the training set

Unbalanced classifiers

- If the training set is unbalanced towards certain classes, the learned model could be biased toward majority classes

- Therefore, it might classify very well those majority classes, but never classify properly other classes

- A proper classifier should be able to perform well on all classes (or at least on most classes) in which it should classify

For example…

Assuming the training set is as follows:

Spam	Ham
10	2000

And the test set is as follows:

Spam	Ham
10	1000

The prediction…

Could tend to predict almost always “Ham”, if not always “Ham”

```
pred      ham  spam

actual

ham      1000     0

spam        9     1
```

The prediction…

Even worse…

```
pred      ham  spam

actual

ham      1000     0

spam       10     0
```

What to do?

- Create a balanced training set
- Note: you can do whatever you want on the training set… transform it as you wish
- **The important thing is to never ever touch the test set!!!!**

How?

- **Undersampling**: reduce the majority class in the training set, by only selecting a random set of elements
    - Not good if the minority classes are too small
- **Oversampling**: increase the size of minority classes, by
    - Duplicating the existing samples (does not help for all classifiers)
    - Creating artificial samples similar to existing ones (see next)

Undersampling

Spam	Ham
10	2000

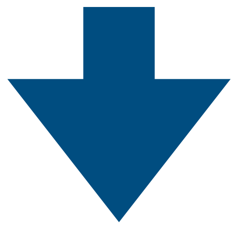
↓

Spam	Ham
10	10

As said it may not work here

Where undersampling may work

Spam	Ham
1000	2000



Spam	Ham
1000	1000

Oversampling

Spam	Ham
10	2000

Spam	Ham
2000	2000

10 real
1990 either duplicate or artificial

Creating artificial samples

- There exist techniques that create artificial samples, that are near the real ones
- What does it mean near?
- If a sample is a vector of features, another vector is near to that one if its distance is small
- The distance can be, for example, the Euclidean distance

$$dist\left(P, Q\right) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \ldots + (p_n - q_n)^2}$$

Example

Original vector

Artificial, Distance=0.55, very close

Artificial Distance=4.17, Not good

driver	car	race	speed
1.3	0	2.4	0.8
1.1	0.1	1.9	0.7
0.2	3	0	2

SMOTE

- Synthetic Minority Oversampling Technique
- Technique to generate artificial samples
- Available in scikit-learn

Rebalancing in Python

Undersampling…

<code to create the tf-idf training set goes here>

```
from imblearn.under_sampling import RandomUnderSampler
from collections import Counter
# instantiates the undersampler. “majority” means that
# the majority class will be undersampled to match the minority one
undersample = RandomUnderSampler(sampling_strategy='majority')
# undersamples the training set
X_new_train, y_train = undersample.fit_resample(X_train_tfidf, y_train)
# prints the dataset composition
counter=Counter(y_train)
print(counter)
```

<code to fit the model and perform the prediction goes here>

Random Oversampling...

<code to create the tf-idf training set goes here>

```
from imblearn.over_sampling import RandomOverSampler
from collections import Counter
#instantiate the random oversampler class. "minority" means that the
# minority class will be oversampled to match the majority class
oversample = RandomOverSampler(sampling_strategy='minority')

# Rebalances the training set by resampling
X_new_train, y_train = oversample.fit_resample(X_train_tfidf, y_train)
# prints the dataset composition
counter=Counter(y_train)
print(counter)
```

<code to fit the model and perform the prediction goes here>

SMOTE Oversampling…

<code to create the tf-idf training set goes here>

```
from imblearn.over_sampling import SMOTE
from collections import Counter

#instantiate the SMOTE oversampler
oversample = SMOTE()

# Rebalances the training set by creating artificial instances
# of the minority class.
X_new_train, y_train = oversample.fit_resample(X_train_tfidf, y_train)
# prints the dataset composition
counter=Counter(y_train)
print(counter)
```

<code to fit the model and perform the prediction goes here>

Using alternative machine learning algorithms

Other ML algorithms

- Studying many of them is out of scope of this course
- You might study ML algorithms at
  "data analytics"
- We will see in the following SVM and later neural networks

Some examples of other algorithms

- K-nearest-neighbour classifier: determines the class of a point based on the class of the K neighbour points

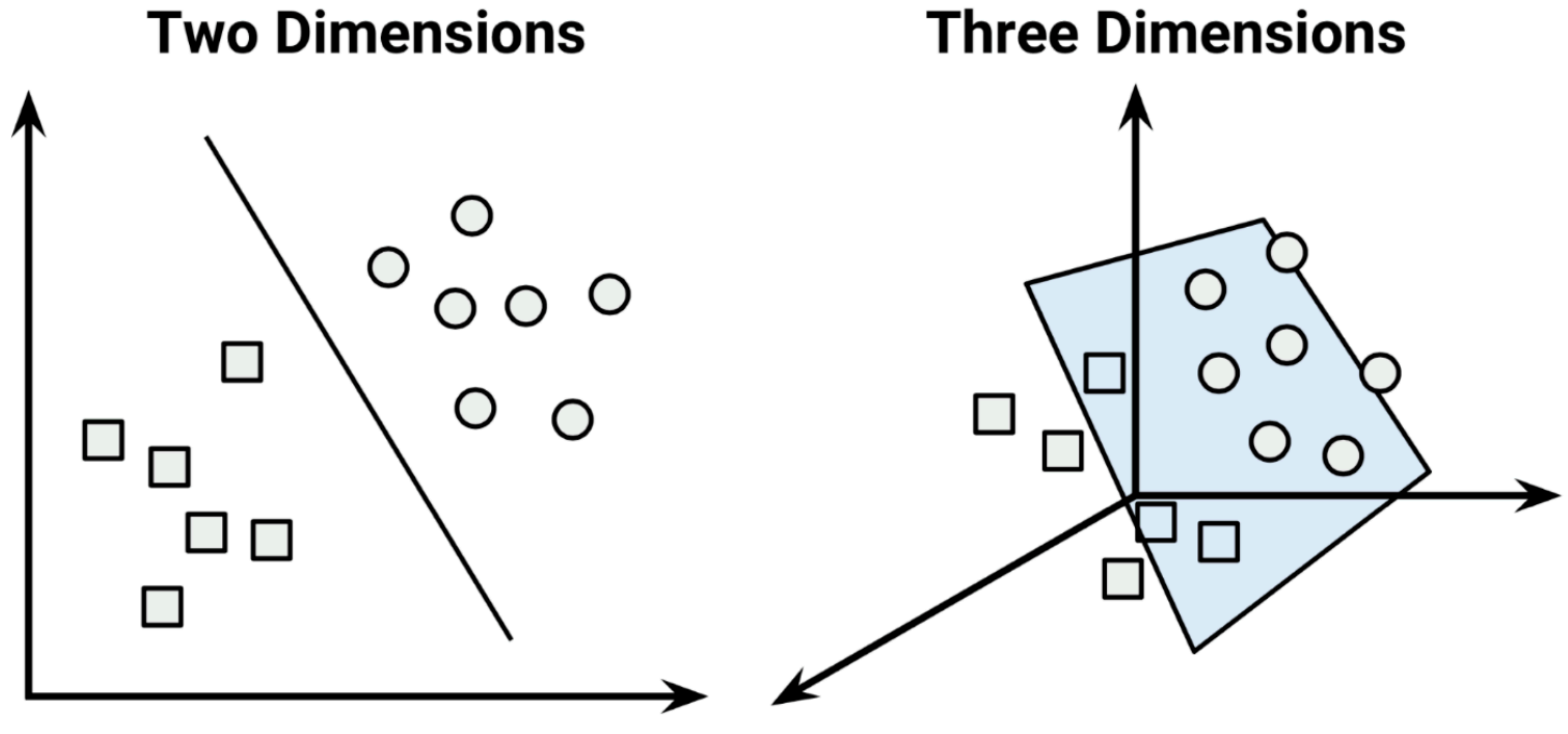
- Decision tree: performs nested separations of the feature space creating rules on feature value thresholds, so that the variability of the created partitions are minimized

- Ensemble classifiers (e.g. Random Forests): instead of creating a single classifier, create multiple classifiers that better fit different parts of the training set. Then run all of them on the test set use a voting mechanism to decide

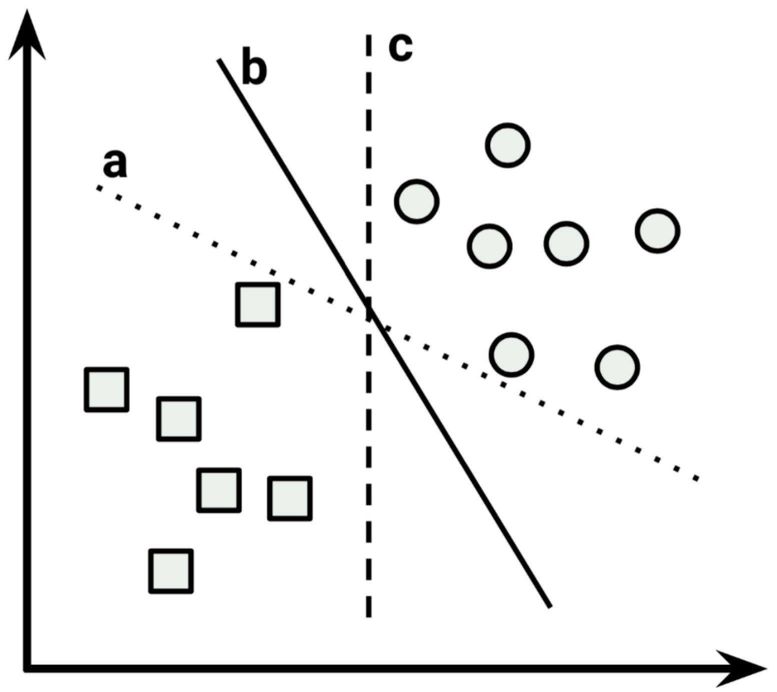
Support Vector Machines (SVM)

- Imagine a SVM as a surface that separates data belonging to different classes

- This surface is called hyperplane



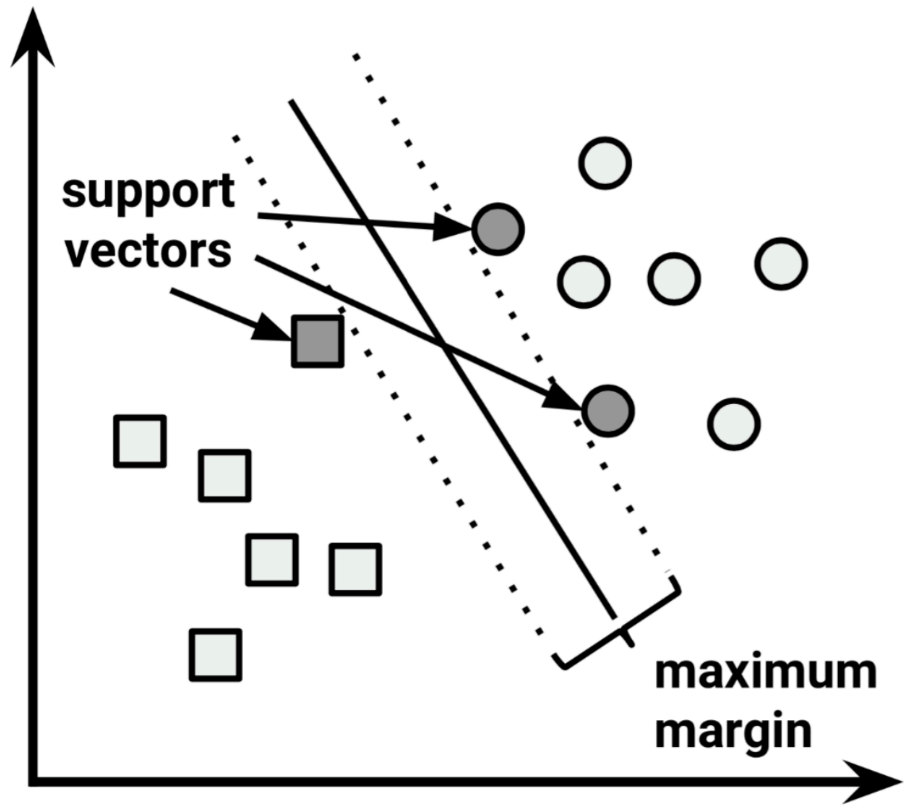
How is an hyperplane chosen?



- Search for a Maximum Margin Hyperplane (MMH)

- The hyperplane that creates the maximum separation between the two classes in the training set

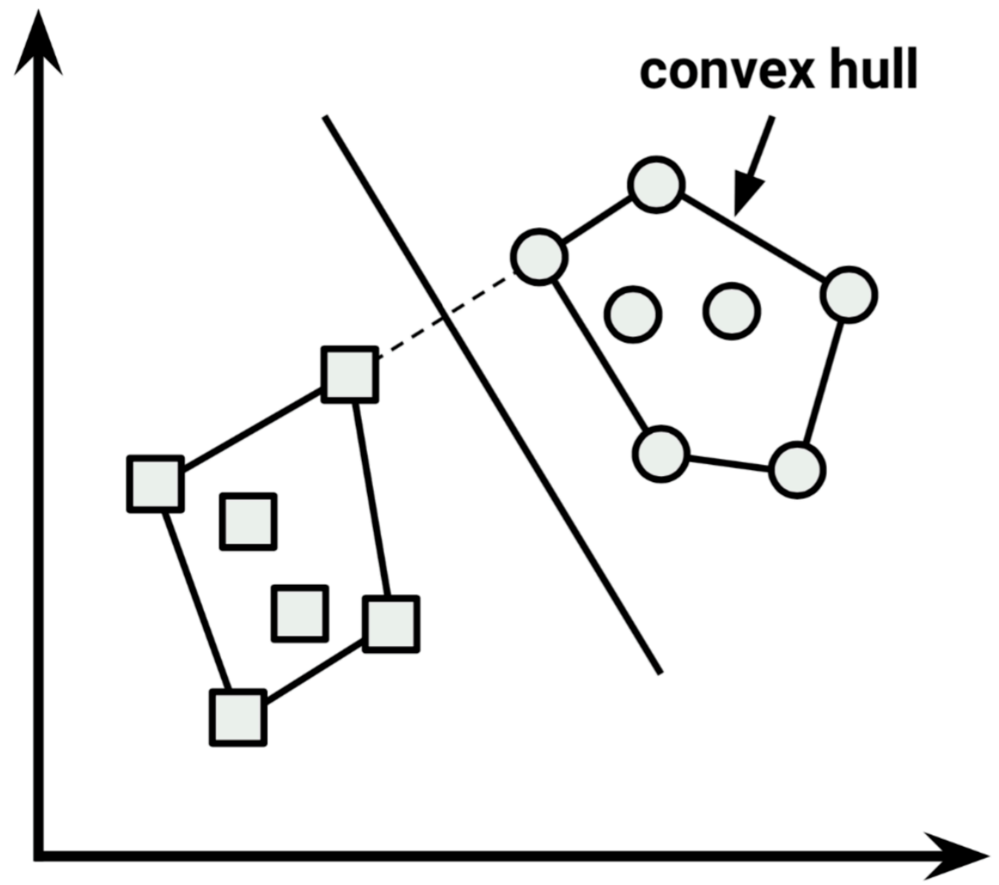
Support Vectors



- Points from each class closest to the hyperplane
- They help to define where the hyperplane should be
  - We won’t study in detail how support vectors are determined

Linearly separable data

- This is an easy case…
- Boundaries of points belonging to each class known as convex hull
- Finding this hyperplanes involves quadratic optimization



Otherwise…

- The process tries to find two parallel hyperplanes that separate the data
- Like finding the thickest mattress between those datapoints
- More specifically the goal is to find a set of weights $\overrightarrow{w}$ that specify two hyperplanes, as follows

$$\overrightarrow{w} \cdot \vec{x} + b \geq 1$$

$$\overrightarrow{w} \cdot \vec{x} + b \leq -1$$

such that all the points of one class fall above the first hyperplane and the points of the other class beneath the second

Maximizing the distance

- In general the distance between two planes $ax + by + cz + d_1 = 0$ and $ax + by + cz + d_2 = 0$ is:

$$\frac{|d_1 - d_2|}{\sqrt{a^2 + b^2 + c^2}}$$

- Which, in our case, becomes $\dfrac{2}{||\vec{w}||}$

- Where $||\vec{w}||$ is the Euclidean norm (the distance from the origin to the vector $\vec{w}$

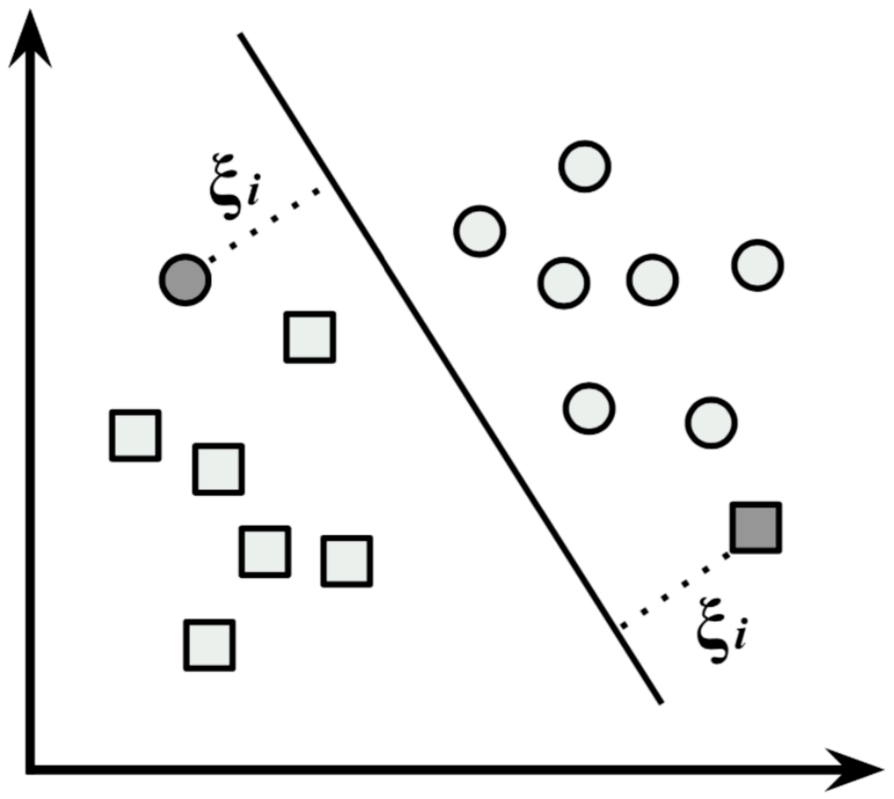
- To maximize the distance, we need to minimize $\vec{w}$

What if variables cannot be separated?

- Use of slack variables

- The algorithm creates a soft margin allowing some points to fall on the wrong side

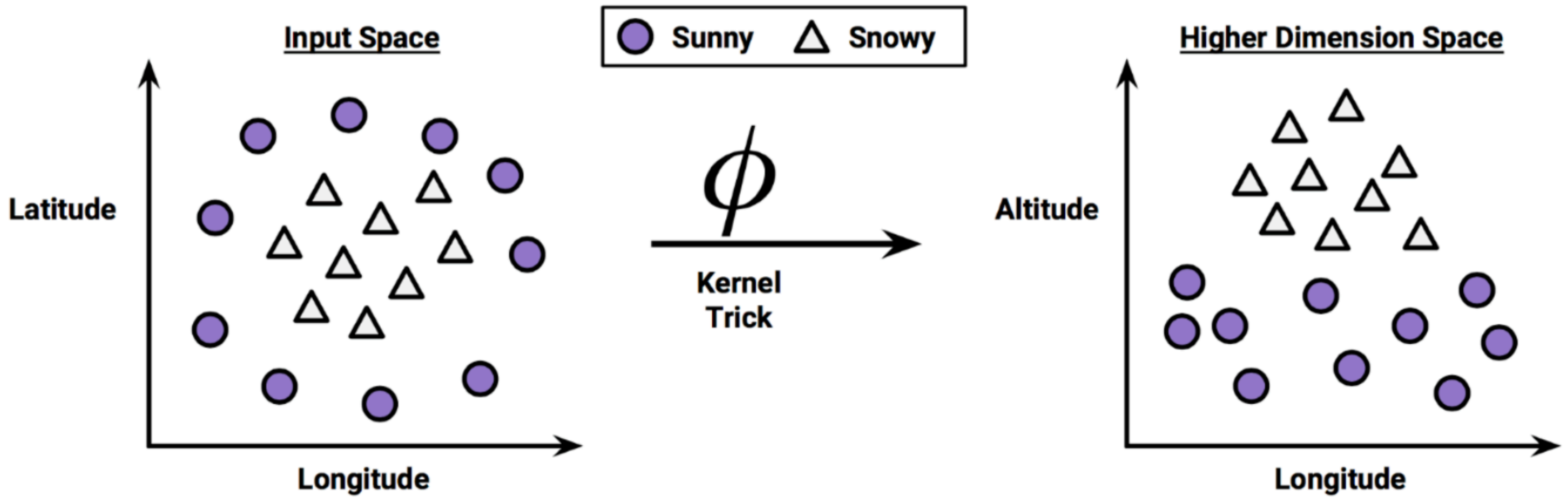
- This, in the optimization, introduces a cost penalty that the algorithm tries to minimize



Nonlinear space

- Slack variables are not the only way of coping with non-separable spaces
- Sometimes the relationship between variables is just non-linear, and this creates lack of separation
- Using a function, named kernel, the relationship between variables can be transformed

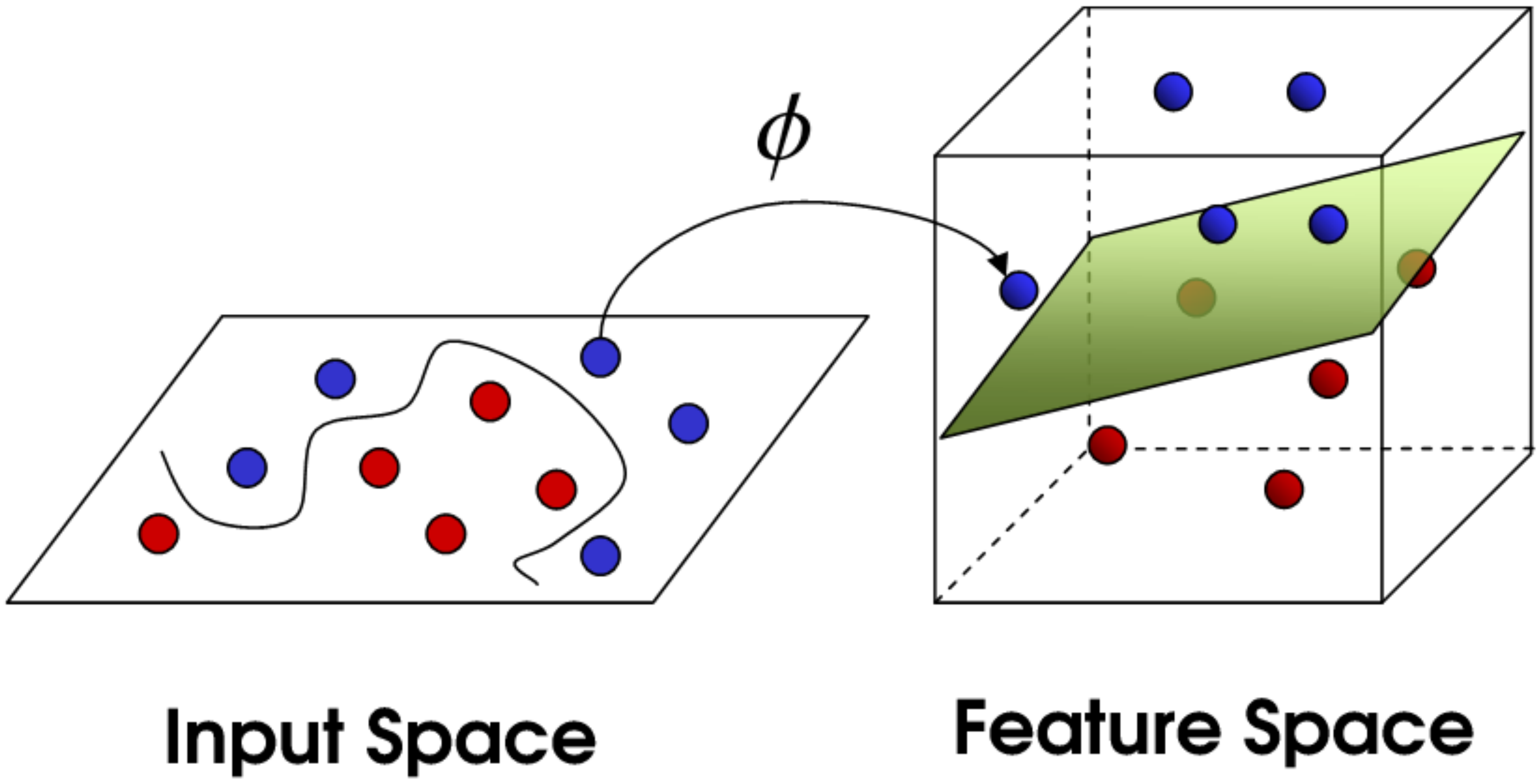
Kernel Transformation



How does the kernel work?

- We map the m-sized space into a (richer) n-feature space
- Then the kernel defines how to compute the dot product in the new space

Kernel Trick



How does the kernel work?

- It creates new artificial features that express relationship between the measured characteristics
- For example, an Altitude feature can be expressed as an interaction between latitude and longitude
- In this way, the SVM learns concepts not measured by the original data

SVM with nonlinear kernels: pros and cons

Strengths	**Weaknesses**
Work for both classification and numerical problems	Difficult to calibrate: finding the best combinations of kernels requires to set various combinations of models and parameters
Not very influenced by noisy data, and robust to overfitting	Can be slow to train, especially with many features
Easier to use then neural networks	Not interpretable
Overall, good performances	

Kernel functions

- A kernel is denoted by the Greek letter phi

$$K\left(\overrightarrow{x_i}, \overrightarrow{x_j}\right) = \phi(\overrightarrow{x_i}) \cdot \phi(\overrightarrow{x_j})$$

- It takes two vectors and combines them with a dot product returning a single number

- A kernel function computes the dot product of the result of two functions without even knowing what the functions are

Example

- Consider a kernel $k(x, y) = (1 + x^T y)^2$

- Given $x = (x_1, x_2)$ and $y = (y_1, y_2)$

- Expanding it we obtain:

$$\left(1 + x_1y_1 + x_2y_2\right)^2 = 1 + x_1^2y_1^2 + x_2^2y_2^2 + 2x_1y_1 + 2x_2y_2 + 2x_1x_2y_1y_2$$

  The latter corresponds to the dot product of

$$\left(1, x_1^2, x_2^2, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2\right) \text{ and } \left(1, y_1^2, y_2^2, \sqrt{2}y_1, \sqrt{2}y_2, \sqrt{2}y_1y_2\right)$$

- Therefore, if we consider $\varphi(x_1, x_2) = \left(1, x_1^2, x_2^2, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2\right)$ and $\varphi(y_1, y_2) = \left(1, y_1^2, y_2^2, \sqrt{2}y_1, \sqrt{2}y_2, \sqrt{2}y_1y_2\right)$

  Then $k(x, y) = \varphi(x)^T\varphi(y)$ computes the dot product in a six-dimensional space without visiting such a space

Linear kernel

- Does not transform the data at all

$$K\left(\overrightarrow{x_i}, \overrightarrow{x_j}\right) = \overrightarrow{x_i} \cdot \overrightarrow{x_j}$$

Polynomial kernel

- Applies a simple nonlinear transformation to the data:

$$K\left(\overrightarrow{x_i}, \overrightarrow{x_j}\right) = \left(\overrightarrow{x_i} \cdot \overrightarrow{x_j} + 1\right)^d$$

- It requires to calibrate the order d

Sigmoid kernel

- As we will see, makes the SVM very similar to the activation function of a neural network:

$$K\left(\overrightarrow{x_i}, \overrightarrow{x_j}\right) = tanh\left(\kappa \overrightarrow{x_i} \overrightarrow{x_j} - \delta\right)$$

- It requires to calibrate the parameters κ and δ

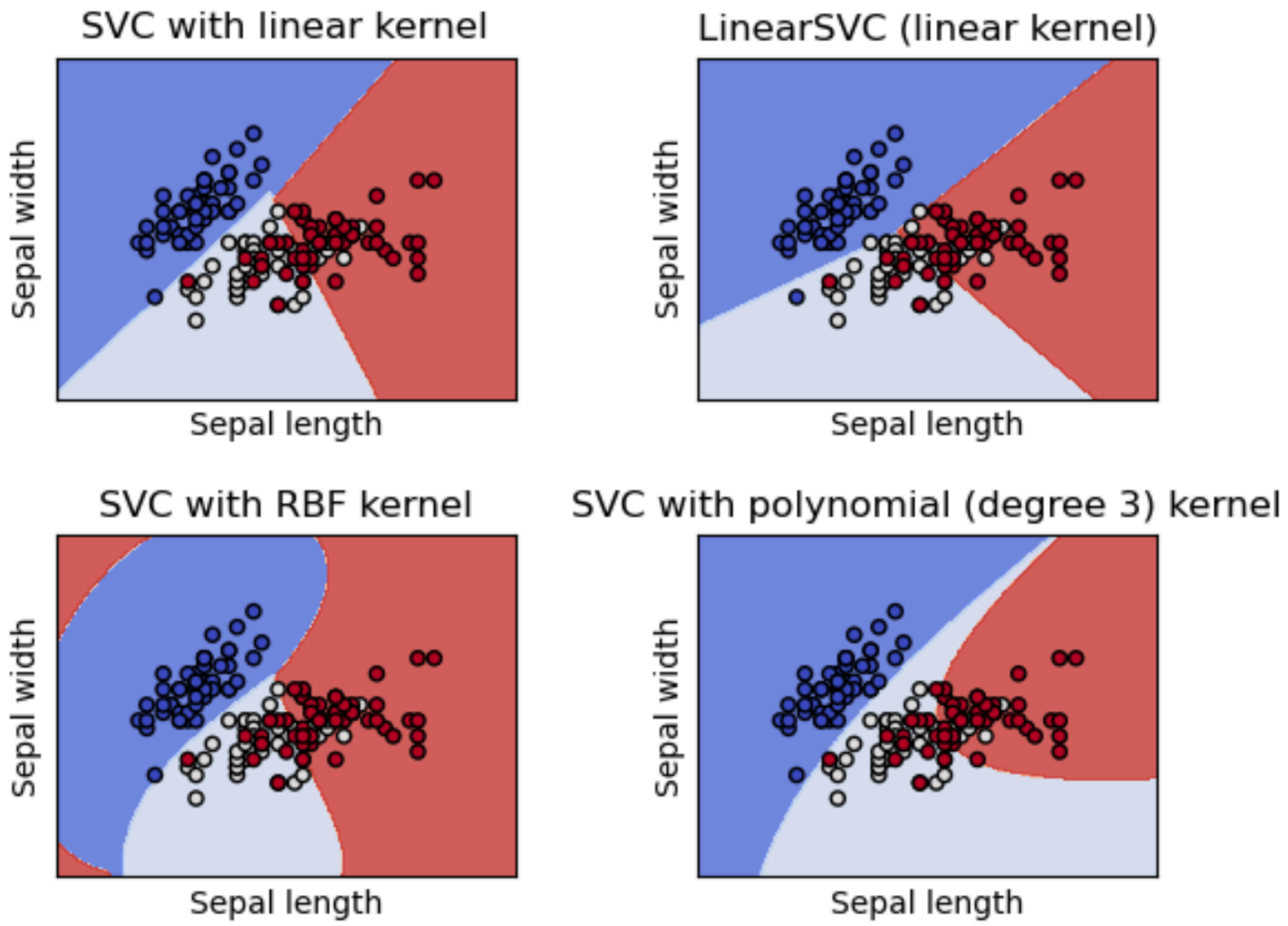
Gaussian kernel

- Performs well for many tasks

$$K\left(\overrightarrow{x_i}, \overrightarrow{x_j}\right) = e^{\frac{-\left\| \overrightarrow{x_i} - \overrightarrow{x_j} \right\|^2}{2\sigma^2}}$$

Separation with different kernels

From https://scikit-learn.org/stable/modules/svm.html



Which kernel to use?

- No particular rule, unfortunately
- This often involves training and evaluating SVM on a validation dataset
- The good news is that for **text, at least for vector space representations, a linear kernel is just enough**
- This is because the matrix is strongly sparse

SVM in Python

As told, changing algorithm is very simple, you just need to change the classifier instancing and then the code is exactly the same used for Naive Bayes:

```python
from sklearn import svm
clf = svm.SVC(kernel='linear')
```

Calibrating kernels and parameters

- See https://scikit-learn.org/stable/modules/svm.html#kernel-functions
- linear: <x,x'> (no parameters)
- polynomial: $(\gamma < x, x' > + r)^d$, need to calibrate γ (gamma), r (coef0), and d (degree)
- rbf: $exp(-\gamma||x - x'||^2)$, need to calibrate γ (gamma)>0
- sigmoid: $tanh(\gamma < x, x' > + r)$, need to calibrate γ (gamma) and r (coef0)

What to calibrate?

- Lets' consider the RBF kernel
- Two parameters must be considered: C and gamma.
- C: trades off misclassification of training examples against simplicity of the decision surface
    - C is common to all SVM kernels
    - A low C makes the decision surface smooth (too low might produce low performances)
    - A high C aims at classifying all training examples correctly (too high might get to overfitting)
- gamma defines how much influence a single training example has. The larger gamma is, the closer other examples must be to be affected.

Exhaustive grid search

- We search parameters across all possible combinations of given values
- We use the GridSearchCV class from sklearn.model_selection
- This classes works on the training set and creates training and validation folds through the training set to perform hyperparameter calibration

Grid search - (1)

```
from sklearn.model_selection import GridSearchCV
from sklearn import svm
from time import time

#creates a model instance with no parameters
svc=svm.SVC()

# prints the list of parameters for the model
print(svm.SVC().get_params().keys())

# create a dictionary with possible values for some parameters
parameters = {
    'C': [1, 10, 100, 1000],
    'gamma': [0.001, 0.0001],
    'kernel': ['rbf','linear']
}
```

Discussion

- First we create a variable containing the model instance, without caring about its parameters

- `svm.SVC().get_params().keys()` returns all hyperparameters of the model

- It prints:
  ```
  dict_keys(['C', 'break_ties', 'cache_size',
  'class_weight', 'coef0', 'decision_function_shape',
  'degree', 'gamma', 'kernel', 'max_iter',
  'probability', 'random_state', 'shrinking', 'tol',
  ‘verbose'])
  ```

- Then, we create a dictionary with only the parameters we are interested to calibrate

Grid search - (2)

```
#instantiates the grid search
# using the svc model and the parameters above defined
grid_search = GridSearchCV(svc, parameters, n_jobs=-1, verbose=10)

print("Performing grid search...")
print("parameters:")
print(parameters)
t0 = time()
# Starts the grid search
grid_search.fit(X_train_tfidf, y_train)
# Prints the required time
print("done in %0.3fs" % (time() - t0))
print()

# Prints the best score
print("Best score: %0.3f" % grid_search.best_score_)
print("Best parameters set:")
best_parameters = grid_search.best_estimator_.get_params()
for param_name in sorted(parameters.keys()):
    print("\t%s: %r" % (param_name, best_parameters[param_name]))
```

Discussion

- We instantiate the GridSearchCV() passing to it the model and the grid search parameters
    - verbose=10 prints outcome of each iteration. Using a lower verbose outputs less. verbose=1 outputs nothing
- Then, the fit() function starts the calibration
- The grid_search.best_score_ gets the best evaluation score achieved (default is accuracy, but you can change it when instantiating GridSearchCV)
- grid_search.best_estimator_.get_params() returns a dictionary with values for the calibrated parameters

Grid search - (3)

```python
#instantiating the model using the grid search best estimator
clf= grid_search.best_estimator_
clf.fit(X_train_tfidf, y_train)

#indexing the test set
X_new_counts = count_vect.transform(X_test)
X_new_tfidf = tfidf_transformer.transform(X_new_counts)

#performing the actual prediction
predicted = clf.predict(X_new_tfidf)

from sklearn import metrics
print(pd.crosstab(y_test,predicted))
print(metrics.classification_report(y_test, predicted))
```

Discussion

- `grid_search.best_estimator_` contains the best estimator found by the grid search

Output

```
Fitting 5 folds for each of 16 candidates, totalling 80 fits
done in 10.324s

Best score: 0.968
Best parameters set:
  C: 1000
  gamma: 0.001
  kernel: 'rbf'

col_0   ham  spam
type
ham    1569    14
spam     31   221

              precision    recall  f1-score   support

         ham       0.98      0.99      0.99      1583
        spam       0.94      0.88      0.91       252

    accuracy                           0.98      1835
   macro avg       0.96      0.93      0.95      1835
weighted avg       0.98      0.98      0.98      1835
```

Alternative to exhaustive search

- Random generating many possible configurations, where each parameter is drawn from a distribution
- Using the class RandomizedSearchCV

What to change in the source code

```python
from sklearn.model_selection import RandomizedSearchCV
from sklearn import svm
from time import time

#creates a model instance with no parameters
svc=svm.SVC()

# create a dictionary with possible values for some parameters
from sklearn.utils.fixes import loguniform

parameters = {
'C': loguniform(1e0, 1e3),
'gamma': loguniform(1e-4, 1e-3),
'kernel': ['rbf'],
}

#instantiates the Random search
# using the svc model and the parameters above defined
grid_search = RandomizedSearchCV(svc, parameters, random_state=0, n_jobs=-1, verbose=10)
```

Discussion

- We specify that parameters are sampled over a log distribution, e.g. loguniform(1e0, 1e3)

  means randomply sampling 1, 10, 100

- Note that you can still specify a list of values

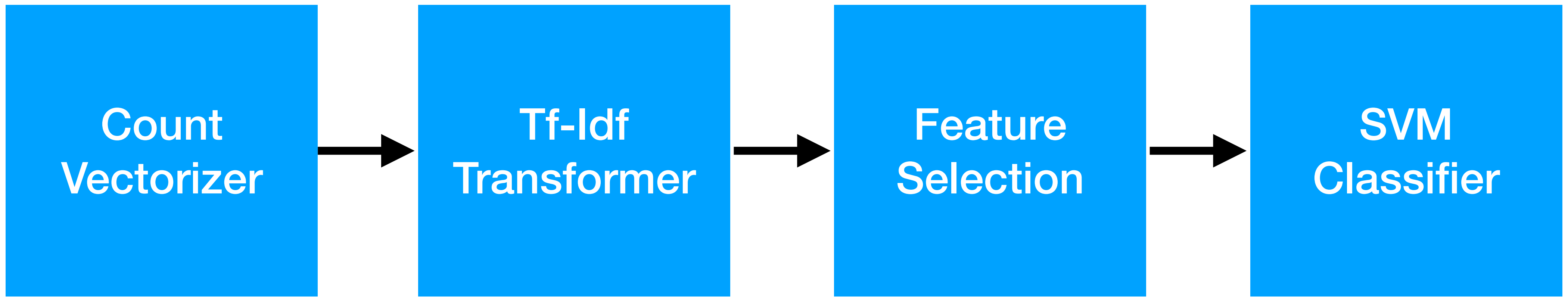
- Various alternatives:

```
from sklearn.utils.fixes import loguniform
from scipy.stats import uniform
parameters = {
 #'C': loguniform(1e0, 1e3), # log distribution
 #'C': [0, 1, 2, 3, 4, 5, 6], #list of values
 'C' : uniform(0,100), #uniform distribution
 'gamma': loguniform(1e-4, 1e-3),
 'kernel': ['rbf'],
}
```

Creating a pipeline

What is a pipeline?

A pipeline allows you to instantiate (and also configure, using for example Grid Search) a machine learning process as a sequence of blocks



We start from here…

- From the usual script
- This time, we just split train and test, without applying the Vectorizers

```
#applies transformText to all rows of text
dataset['text'] = dataset['text'].map(transformText)
#print(dataset['text'].head())


## Split the data
from sklearn.model_selection import train_test_split

#separate the test set
X_train, X_test, y_train, y_test = train_test_split(dataset['text'], dataset['type'],
                                                    test_size=0.33, random_state=10)
print ("Training Sample Size:", len(X_train), ' ', "Test Sample Size:" ,len(X_test))
```

1 - Creating the pipeline

```python
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.feature_selection import SelectPercentile
from sklearn.feature_selection import chi2
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn import svm
from time import time

#creates a model instance with no parameters
svc=svm.SVC()

# defines the steps of the pipeline, each with
# a name and the model object
pipeline = Pipeline(
    [
        ("vect", CountVectorizer()),
        ("tfidf", TfidfTransformer()),
        ("selector",SelectPercentile()),
        ("clf", svc),
    ]
)
```

Discussion

- The pipeline is composed of a list of tuples
- Each tuple contains:
  - a symbolic name we give to the stage
  - the instance of the estimator/transformer

2 - Setting possible parameters

```
# create a dictionary with possible values for some parameters
# each parameter name is composed as
# pipelineStepName__componentParameter
parameters = {
    "vect__ngram_range": ((1, 1), (1, 2)),
    "vect__min_df": (20,30,40),
    'tfidf__use_idf': (True, False),
    'selector__score_func': [chi2], #selector function needs a list
    'selector__percentile': (20,30,40),
    'clf__C': [1, 10, 100, 1000],
    'clf__gamma': [0.001, 0.0001],
    'clf__kernel': ['rbf','linear']
}
```

Discussion

- When specifying parameter, each parameter needs to be named composing, using a double underscore "__"
    - The stage name
    - The parameter name for the specific component
- For example `vect__ngram_range` is the "ngram_range" parameter of the vect stage, i.e., of CountVectorizer()

3 - Performing the grid search

- The same as before
- However, the first parameter is the pipeline and not a model

```python
#instantiates the grid search
# using the svc model and the parameters above defined
grid_search = GridSearchCV(pipeline, parameters, n_jobs=-1, verbose=10)

print("Performing grid search...")
print("parameters:")
print(parameters)
t0 = time()
# Starts the grid search
grid_search.fit(X_train, y_train)
# Prints the required time
print("done in %0.3fs" % (time() - t0))
print()

# Prints the best score
print("Best score: %0.3f" % grid_search.best_score_)
print("Best parameters set:")
best_parameters = grid_search.best_estimator_.get_params()
for param_name in sorted(parameters.keys()):
    print("\t%s: %r" % (param_name, best_parameters[param_name]))
```

4 - Creating and testing the model

- Again, exactly as before
- Also, we don’t need to transform X_test as the pipeline does it for us

```python
#Creating the model:

#instantiating the model using the grid search best parameters
clf=best_pipe = grid_search.best_estimator_
clf.fit(X_train, y_train)

#performing the actual prediction
predicted = clf.predict(X_test)

from sklearn import metrics
print(pd.crosstab(y_test,predicted))
print(metrics.classification_report(y_test, predicted))
```

Adding a customized transformer to the pipeline

scikit-learn transformer APIs

- fit() fits the data to the model contained in the transformer. For example, if the transformer is a machine-learning algorithm, it trains the algorithm. If it is an indexer, it learns the vocabulary

- transform() performs the data transformation through the transformer

- fit_transform() performs both a fit and a transform

Where to use which?

- For example CountVectorizer needs fit_transform() because it first learns the vocabulary, and then transforms the strings into a document-term matrix
- A classifier, e.g. MultinomialBayes() or SBC() just performs a fit()

Creating a simple transformer

- Inherits from BaseEstimator class, which provides the set_param and get_param methods
- Inherits from TransformerMixin class, which provides the fit_transform() method, once you implement the fit() and transform() ones
- Therefore, multiple inheritance

Example: text processing transformer

```python
from sklearn.base import BaseEstimator,TransformerMixin

class PreprocessTransformer(BaseEstimator,TransformerMixin):
    def __init__(self,stop=True,stripNum=True,minSize=3,stemming=True):
        self.stop=stop
        self.stripNum=stripNum
        self.minSize=minSize
        self.stemming=stemming
    def fit(self,x, y=None):
        return self
    def transform(self,x, y=None):
        xc=x.copy()
        xc = xc.map(self.transformText)
        return xc
```

The transformText method

```python
def transformText(self,text):
    stops = set(stopwords.words("english"))
    # Convert text to lowercase
    text = text.lower()
    # Strip multiple whitespaces
    text = gensim.corpora.textcorpus.strip_multiple_whitespaces(text)
    if self.stop:
        # Removing all the stopwords
        filtered_words = [word for word in text.split() if word not in stops]
        # Preprocessed text after stop words removal
        text = " ".join(filtered_words)
    # Remove the punctuation
    text = gensim.parsing.preprocessing.strip_punctuation(text)
    if self.stripNum:
        # Strip all the numerics
        text = gensim.parsing.preprocessing.strip_numeric(text)
    if self.minSize>0:
        # Removing all the words with less than 3 characters
        text = gensim.parsing.preprocessing.strip_short(text, minsize=self.minSize)
    # Strip multiple whitespaces
    text = gensim.corpora.textcorpus.strip_multiple_whitespaces(text)
    # Stemming
    if self.stemming:
        text=gensim.parsing.preprocessing.stem_text(text)
    return text
```

Discussion:

- We create the class PreprocessTransformer which performs a multiple inheritance:
  ```
  class PreprocessTransformer(BaseEstimator,TransformerMixin)
  ```
- We create the class PreprocessTransformer which performs a multiple inheritance
- The __init__() method initializes all the preprocessing parameters
- The fit() method takes x and y as inputs, and does nothing
- The transform() method creates a copy of the x parameter, and applies the transformation on all rows of x
- The transformText() method performs the preprocessing based on the class attribute values

Splitting training and test (as usual)

```
## Split the data
from sklearn.model_selection import train_test_split
#separate the test set
X_train, X_test, y_train, y_test = train_test_split(dataset['text'],
dataset['type'],
                                                                       test_size=0.33, random_state=10)
print ("Training Sample Size:", len(X_train), ' ', "Test Sample Size:" ,len(X_test))
```

Creating the pipeline

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.feature_selection import SelectPercentile
from sklearn.feature_selection import chi2
from sklearn.pipeline import Pipeline
from sklearn import svm

# defines the steps of the pipeline, each with
# a name and the model object
clf = Pipeline(
    [
        ("prep", PreprocessTransformer()),
        ("vect", CountVectorizer()),
        ("tfidf", TfidfTransformer()),
        (“selector",SelectPercentile(score_func=chi2, percentile=30)),
        ("clf", svm.SVC(kernel=“rbf",C=1000, gamma=0.0001)),
    ]
)
```

Note

- In this case we have instantiated the pipeline by setting the hyperparameters manually
- However, the example can be adapted to use GridSearch for setting the parameters for all stages, including the customized stage

Fitting and predicting...

```
clf.fit(X_train, y_train)

#performing the actual prediction
predicted = clf.predict(X_test)

from sklearn import metrics
print(pd.crosstab(y_test,predicted))
print(metrics.classification_report(y_test, predicted))
```

Minor Note

If you use a customized transformer in a GridSearch, you need to set the number of jobs=1 (there is currently a bug in the library)